R / *Bioconductor* for High-Throughput Sequence Analysis

Marc Carlson, Valerie Obenchain, Hervé Pagès, Paul Shannon, Dan Tenenbaum, Martin Morgan[1]

31 May, 2013

[1]mtmorgan@fhcrc.org

Contents

This course takes a quick tour through the essentials of *R* / *Bioconductor* for high-throughput sequence analysis. We start with a refresher on using *R* efficiently, including tips for managing large data. The next stop explores how *R* / *Bioconductor* represents genome and other bioloical sequences, short reads, and alignments; we are reminded of the strengths and challenges of *R*'s S4 class system. Ranges are a central concept in *Bioconductor*'s approach to sequence analysis; we use these in a variety of contexts. The tour concludes with an exploration of how *Bioconductor* can be used to place quantitative results in to a richer biological context of annotation and visualization.

There are many books to help with using *R*, but not yet a book-length treatment of *Bioconductor* tools for sequence analysis; Recent training material is available on the *Bioconductor* web site[1] novices, one place to start is Pardis' *R for Beginners*[2]. General *R* programming recommendations include Dalgaard's *Introductory Statistics with R* [4], Matloff's *The Art of R Programming* [5], and Meys and de Vies' *R For Dummies* [6]; an interesting internet resource for intermediate *R* programming is Burns' *The R Inferno*[3].

Table 1: Approximate schedule.

9:00 - 10:30	Up to speed with *R* and *Bioconductor*
10:30 - 12:00	Sequences, reads, and alignments
12:00 - 12:30	Ranges: introduction
12:30 - 1:30	Lunch
1:30 - 3:30	Ranges: in action
3:30 - 4:15	Variants, annotations, and visualization
4:15 - 5:00	Working with large data

[1] `http://bioconductor.org/help/course-materials/`.. For *R*

[2] `http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf`

[3] `http://www.burns-stat.com/documents/books/the-r-inferno/`

Chapter 1

Efficient *R*

R is an open-source statistical programming language. It is used to manipulate data, to perform statistical analysis, and to present graphical and other results. *R* consists of a core language, additional 'packages' distributed with the *R* language, and a very large number of packages contributed by the broader community. Packages add specific functionality to an *R* installation. *R* has become the primary language of academic statistical analysis, and is widely used in diverse areas of research, government, and industry.

R has several unique features. It has a surprisingly 'old school' interface: users type commands into a console; scripts in plain text represent work flows; tools other than *R* are used for editing and other tasks. *R* is a flexible programming language, so while one person might use functions provided by *R* to accomplish advanced analytic tasks, another might implement their own functions for novel data types. As a programming language, *R* adopts syntax and grammar that differ from many other languages: objects in *R* are 'vectors', and functions are 'vectorized' to operate on all elements of the object; *R* objects have 'copy on change' and 'pass by value' semantics, reducing unexpected consequences for users at the expense of less efficient memory use; common paradigms in other languages, such as the 'for' loop, are encountered much less commonly in *R*. Many authors contribute to *R*, so there can be a frustrating inconsistency of documentation and interface. *R* grew up in the academic community, so authors have not shied away from trying new approaches. Common statistical analysis functions are very well-developed.

1.1 *R*

Opening an *R* session results in a prompt. The user types instructions at the prompt. Here is an example:

```
> ## assign values 5, 4, 3, 2, 1 to variable 'x'
> x <- c(5, 4, 3, 2, 1)
> x
```

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

The first line starts with a `#` to represent a comment; the line is ignored by *R*. The next line creates a variable `x`. The variable is assigned (using `<-`, we could have used `=` almost interchangeably) a value. The value assigned is the result of a call to the `c` function. That it is a function call is indicated by the symbol named followed by parentheses, `c()`. The `c` function takes zero or more arguments, and returns a vector. The vector is the value assigned to `x`. *R* responds to this line with a new prompt, ready for the next input. The next line asks *R* to display the value of the variable `x`. *R* responds by printing `[1]` to indicate that the subsequent number is the first element of the vector. It then prints the value of `x`.

R has many features to aid common operations. Entering sequences is a very common operation, and expressions of the form `2:4` create a sequence from `2` to `4`. Sub-setting one vector by another is enabled with `[`. Here we create an integer sequence from 2 to 4, and use the sequence as an index to select the second, third, and fourth elements of `x`

Table 1.1: Essential aspects of the *R* language.

Category	Function	Description
Vectors	`integer`, `numeric` `complex`, `character` `raw`, `factor`	Vectors holding a single type of data length 0 or more
List-like	`list`	Arbitrary collections of elements
	`data.frame`	List of equal-length vectors
	`environment`	*Pass-by-reference* data storage; hash
Array-like	`matrix`	Two-dimensional, homogeneous types
	`data.frame`	Homogeneous columns; row- and column indexing
	`array`	0 or more dimensions
Statistical	`NA`, `factor`	Essential statistical concepts, integral to the language.
Classes	'S3'	List-like structured data; simple inheritance & dispatch
	'S4'	Formal classes, multiple inheritance & dispatch
Functions	'function'	A simple function with arguments, body, and return value
	'generic'	A (S3 or S4) function with associated *methods*
	'method'	A function implementing a generic for an S3 or S4 class

```
> x[2:4]

[1] 4 3 2
```

Index values can be repeated, and if outside the domain of `x` return the special value `NA`. Negative index values remove elements from the vector. Logical and character vectors (described below) can also be used for sub-setting.

R functions operate on variables. Functions are usually vectorized, acting on all elements of their argument and obviating the need for explicit iteration. Functions can generate warnings when performing suspect operations, or errors if evaluation cannot proceed; try `log(-1)`.

```
> log(x)

[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000
```

1.1.1 Essential data types

R has a number of built-in data types, summarized in Table 1.1. These represent `integer`, `numeric` (floating point), `complex`, `character`, `logical` (Boolean), and `raw` (byte) data. It is possible to convert between data types, and to discover the type or mode of a variable.

```
> c(1.1, 1.2, 1.3)         # numeric

[1] 1.1 1.2 1.3

> c(FALSE, TRUE, FALSE)    # logical

[1] FALSE  TRUE FALSE

> c("foo", "bar", "baz")   # character, single or double quote ok

[1] "foo" "bar" "baz"

> as.character(x)          # convert 'x' to character
```

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

```
> typeof(x)                # the number 5 is numeric, not integer
```

```
[1] "double"
```

```
> typeof(2L)               # append 'L' to force integer
```

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

```
> typeof(2:4)              # ':' produces a sequence of integers
```

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

R includes data types particularly useful for statistical analysis, including `factor` to represent categories and `NA` (used in any vector) to represent missing values.

```
> sex <- factor(c("Male", "Female", NA), levels=c("Female", "Male"))
> sex
```

```
[1] Male   Female <NA>
Levels: Female Male
```

Lists, data frames, and matrices All of the vectors mentioned so far are homogeneous, consisting of a single type of element. A `list` can contain a collection of different types of elements and, like all vectors, these elements can be named to create a key-value association.

```
> lst <- list(a=1:3, b=c("foo", "bar"), c=sex)
> lst
```

```
$a
[1] 1 2 3

$b
[1] "foo" "bar"

$c
[1] Male   Female <NA>
Levels: Female Male
```

Lists can be subset like other vectors to get another list, or subset with `[[` to retrieve the actual list element; as with other vectors, sub-setting can use names

```
> lst[c(3, 1)]             # another list
```

```
$c
[1] Male   Female <NA>
Levels: Female Male

$a
[1] 1 2 3
```

```
> lst[["a"]]               # the element itself, selected by name
```

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

A data.frame is a list of equal-length vectors, representing a rectangular data structure not unlike a spread sheet. Each column of the data frame is a vector, so data types must be homogeneous within a column. A data.frame can be subset by row or column, and columns can be accessed with $ or [[.

```
> df <- data.frame(age=c(27L, 32L, 19L),
+                  sex=factor(c("Male", "Female", "Male")))
> df
```

```
  age    sex
1  27   Male
2  32 Female
3  19   Male
```

```
> df[c(1, 3),]
```

```
  age  sex
1  27 Male
3  19 Male
```

```
> df[df$age > 20,]
```

```
  age    sex
1  27   Male
2  32 Female
```

A matrix is also a rectangular data structure, but subject to the constraint that all elements are the same type. A matrix is created by taking a vector, and specifying the number of rows or columns the vector is to represent.

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

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

```
> m[c(1, 3), c(2, 4)]
```

```
     [,1] [,2]
[1,]    4   10
[2,]    6   12
```

On sub-setting, *R* coerces a single column data.frame or single row or column matrix to a vector if possible; use drop=FALSE to stop this behavior.

```
> m[, 3]
```

```
[1] 7 8 9
```

```
> m[, 3, drop=FALSE]
```

```
     [,1]
[1,]    7
[2,]    8
[3,]    9
```

An array is a data structure for representing homogeneous, rectangular data in higher dimensions.

1.1.2 S3 (and S4) classes

More complicated data structures are represented using the 'S3' or 'S4' object system. Objects are often created by functions (for example, `lm`, below), with parts of the object extracted or assigned using *accessor* functions. The following generates 1000 random normal deviates as `x`, and uses these to create another 1000 deviates `y` that are linearly related to `x` but with some error. We fit a linear regression using a 'formula' to describe the relationship between variables, summarize the results in a familiar ANOVA table, and access `fit` (an S3 object) for the residuals of the regression, using these as input first to the `var` (variance) and then `sqrt` (square-root) functions. Objects can be interrogated for their class.

```
> x <- rnorm(1000, sd=1)
> y <- x + rnorm(1000, sd=.5)
> fit <- lm(y ~ x)       # formula describes linear regression
> fit                    # an 'S3' object

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
   0.005389     0.971044

> anova(fit)

Analysis of Variance Table

Response: y
           Df Sum Sq Mean Sq F value    Pr(>F)
x           1 920.84  920.84  3923.9 < 2.2e-16 ***
Residuals 998 234.21    0.23
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

> sqrt(var(resid(fit)))  # residuals accessor and subsequent transforms

[1] 0.4841906

> class(fit)

[1] "lm"
```

Many *Bioconductor* packages implement S4 objects to represent data. S3 and S4 systems are quite different from a programmer's perspective, but fairly similar from a user's perspective: both systems encapsulate complicated data structures, and allow for methods specialized to different data types; accessors are used to extract information from the objects.

1.1.3 Functions

R has a very large number of functions; Table 1.2 provides a brief list of those that might be commonly used and particularly useful. See the help pages (e.g., `?lm`) and examples (`example(match)`) for more details on each of these functions.

R functions accept arguments, and return values. Arguments can be required or optional. Some functions may take variable numbers of arguments, e.g., the columns in a `data.frame`

Table 1.2: A selection of *R* function.

`dir`, `read.table` **(and friends)**, `scan`	List files in a directory, read spreadsheet-like data into *R*, efficiently read homogeneous data (e.g., a file of numeric values) to be represented as a matrix.
`c`, `factor`, `data.frame`, `matrix`	Create a vector, factor, data frame or matrix.
`summary`, `table`, `xtabs`	Summarize, create a table of the number of times elements occur in a vector, cross-tabulate two or more variables.
`t.test`, `aov`, `lm`, `anova`, `chisq.test`	Basic comparison of two (`t.test`) groups, or several groups via analysis of variance / linear models (`aov` output is probably more familiar to biologists), or compare simpler with more complicated models (`anova`); χ^2 tests.
`dist`, `hclust`	Cluster data.
`plot`	Plot data.
`ls`, `str`, `library`, `search`	List objects in the current (or specified) workspace, or peak at the structure of an object; add a library to or describe the search path of attached packages.
`lapply`, `sapply`, `mapply`, `aggregate`	Apply a function to each element of a list (`lapply`, `sapply`), to elements of several lists (`mapply`), or to elements of a list partitioned by one or more factors (`aggregate`).
`with`	Conveniently access columns of a data frame or other element without having to repeat the name of the data frame.
`match`, `%in%`	Report the index or existence of elements from one vector that match another.
`split`, `cut`, `unlist`	Split one vector by an equal length factor, cut a single vector into intervals encoded as levels of a factor, unlist (concatenate) list elements.
`strsplit`, `grep`, `sub`	Operate on character vectors, splitting it into distinct fields, searching for the occurrence of a patterns using regular expressions (see `?regex`, or substituting a string for a regular expression.
`install.packages`	Install a package from an on-line repository into your *R*.
`traceback`, `debug`, `browser`	Report the sequence of functions under evaluation at the time of the error; enter a debugger when a particular function or statement is invoked.

```
> y <- 5:1
> log(y)

[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000

> args(log)        # arguments 'x' and 'base'; see ?log

function (x, base = exp(1))
NULL

> log(y, base=2)   # 'base' is optional, with default value

[1] 2.321928 2.000000 1.584963 1.000000 0.000000

> try(log())       # 'x' required; 'try' continues even on error
> args(data.frame) # ... represents variable number of arguments

function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
    stringsAsFactors = default.stringsAsFactors())
NULL
```

Arguments can be matched by name or position. If an argument appears after `...`, it must be named.

```
> log(base=2, y)   # match argument 'base' by name, 'x' by position

[1] 2.321928 2.000000 1.584963 1.000000 0.000000
```

A function such as `anova` is a *generic* that provides an overall signature but dispatches the actual work to the *method* corresponding to the class(es) of the arguments used to invoke the generic. A generic may have fewer arguments than a method, as with the S3 function `anova` and its method `anova.glm`.

```
> args(anova)

function (object, ...)
NULL

> args(anova.glm)

function (object, ..., dispersion = NULL, test = NULL)
NULL
```

The `...` argument in the `anova` generic means that additional arguments are possible; the `anova` generic hands these arguments to the method it dispatches to.

1.1.4 Packages

Packages provide functionality beyond that available in base *R*. There are over 4000 packages in CRAN (comprehensive *R* archive network) and more than 670 *Bioconductor* packages. Packages are contributed by diverse members of the community; they vary in quality (many are excellent) and sometimes contain idiosyncratic aspects to their implementation. Table 1.3 outlines key base packages and selected contributed packages; see a local CRAN mirror (including the task views summarizing packages in different domains) and *Bioconductor* for additional contributed packages.

Table 1.3: Selected base and contributed packages.

Package	Description
base	Data input and manipulation; scripting and programming.
stats	Essential statistical and plotting functions.
lattice, *ggplot2*	Approaches to advanced graphics.
methods	'S4' classes and methods.
parallel	Facilities for parallel evaluation.

1.1.5 Exercise

Exercise 1

This exercise uses data describing 128 microarray samples as a basis for exploring R functions. Covariates such as age, sex, type, stage of the disease, etc., are in a data file `pData.csv`.

The following command creates a variable `pdataFiles` that is the location of a comma-separated value ('csv') file to be used in the exercise. A csv file can be created using, e.g., 'Save as...' in spreadsheet software.

```
> pdataFile <- system.file(package="SequenceAnalysisData", "extdata",
+                          "pData.csv")
```

Input the csv file using `read.table`, assigning the input to a variable `pdata`. Use `dim` to find out the dimensions (number of rows, number of columns) in the object. Are there 128 rows? Use `names` or `colnames` to list the names of the columns of `pdata`. Use `summary` to summarize each column of the data. What are the data types of each column in the data frame?

A data frame is a list of equal length vectors. Select the 'sex' column of the data frame using `[[` or `$`. Pause to explain to your neighbor why this sub-setting works. Since a data frame is a list, use `sapply` to ask about the class of each column in the data frame. Explain to your neighbor what this produces, and why.

Use `table` to summarize the number of males and females in the sample. Consult the help page `?table` to figure out additional arguments required to include `NA` values in the tabulation.

The `mol.biol` column summarizes molecular biological attributes of each sample. Use `table` to summarize the different molecular biology levels in the sample. Use `%in%` to create a logical vector of the samples that are either `BCR/ABL` or `NEG`. Subset the original phenotypic data to contain those samples that are `BCR/ABL` or `NEG`.

After sub-setting, what are the levels of the `mol.biol` column? Set the levels to be `BCR/ABL` and `NEG`, i.e., the levels in the subset.

One would like covariates to be similar across groups of interest. Use `t.test` to assess whether `BCR/ABL` and `NEG` have individuals with similar age. To do this, use a `formula` that describes the response `age` in terms of the predictor `mol.biol`. If age is not independent of molecular biology, what complications might this introduce into subsequent analysis? Use

Solution: Here we input the data and explore basic properties.

```
> pdata <- read.table(pdataFile)
> dim(pdata)
```

```
[1] 128  21
```

```
> names(pdata)
```

```
 [1] "cod"            "diagnosis"      "sex"            "age"
 [5] "BT"             "remission"      "CR"             "date.cr"
 [9] "t.4.11."        "t.9.22."        "cyto.normal"    "citog"
[13] "mol.biol"       "fusion.protein" "mdr"            "kinet"
[17] "ccr"            "relapse"        "transplant"     "f.u"
[21] "date.last.seen"
```

```
> summary(pdata)
```

```
      cod           diagnosis     sex         age              BT    
 10005  :  1   1/15/1997 :  2   F   :42   Min.   : 5.00   B2     :36  
 1003   :  1   1/29/1997 :  2   M   :83   1st Qu.:19.00   B3     :23  
 1005   :  1   11/15/1997:  2   NA's: 3   Median :29.00   B1     :19  
 1007   :  1   2/10/1998 :  2             Mean   :32.37   T2     :15  
 1010   :  1   2/10/2000 :  2             3rd Qu.:45.50   B4     :12  
 11002  :  1   (Other)   :116             Max.   :58.00   T3     :10  
 (Other):122   NA's      :  2             NA's   :5       (Other):13  
 remission                  CR           date.cr    t.4.11.        
 CR  :99   CR                :96   11/11/1997: 3   Mode :logical  
 REF :15   DEATH IN CR       : 3   1/21/1998 : 2   FALSE:86       
 NA's:14   DEATH IN INDUCTION: 7   10/18/1999: 2   TRUE :7        
           REF               :15   12/7/1998 : 2   NA's :35       
           NA's              : 7   1/17/1997 : 1                  
                                   (Other)   :87                  
                                   NA's      :31                  
  t.9.22.        cyto.normal               citog        mol.biol 
 Mode :logical   Mode :logical   normal       :24   ALL1/AF4:10  
 FALSE:67        FALSE:69        simple alt.  :15   BCR/ABL :37  
 TRUE :26        TRUE :24        t(9;22)      :12   E2A/PBX1: 5  
 NA's :35        NA's :35        t(9;22)+other:11   NEG     :74  
                                 complex alt. :10   NUP-98  : 1  
                                 (Other)      :21   p15/p16 : 1  
                                 NA's         :35                
   fusion.protein   mdr          kinet       ccr            relapse       
 p190     :17     NEG :101   dyploid:94   Mode :logical   Mode :logical  
 p190/p210: 8     POS : 24   hyperd.:27   FALSE:74        FALSE:35       
 p210     : 8     NA's:  3   NA's   : 7   TRUE :26        TRUE :65       
 NA's     :95                             NA's :28        NA's :28       



 transplant                     f.u        date.last.seen
 Mode :logical   REL              :61   1/7/1998  : 2    
 FALSE:91        CCR              :23   12/15/1997: 2    
 TRUE :9         BMT / DEATH IN CR: 4   12/31/2002: 2    
 NA's :28        BMT / CCR        : 3   3/29/2001 : 2    
                 DEATH IN CR      : 2   7/11/1997 : 2    
                 (Other)          : 7   (Other)   :83    
                 NA's             :28   NA's      :35    
```

A data frame can be subset as if it were a matrix, or a list of column vectors.

```
> head(pdata[,"sex"], 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata$sex, 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata[["sex"]], 3)

[1] M M F
Levels: F M

> sapply(pdata, class)

           cod      diagnosis            sex            age             BT
      "factor"       "factor"       "factor"      "integer"       "factor"
     remission             CR        date.cr        t.4.11.        t.9.22.
      "factor"       "factor"       "factor"      "logical"      "logical"
   cyto.normal          citog       mol.biol fusion.protein            mdr
     "logical"       "factor"       "factor"       "factor"       "factor"
         kinet            ccr        relapse     transplant            f.u
      "factor"      "logical"      "logical"      "logical"       "factor"
date.last.seen
      "factor"
```

The number of males and females, including `NA`, is

```
> table(pdata$sex, useNA="ifany")

   F    M <NA>
  42   83    3
```

An alternative version of this uses the `with` function: `with(pdata, table(sex, useNA="ifany"))`.

The `mol.biol` column contains the following samples:

```
> with(pdata, table(mol.biol, useNA="ifany"))

mol.biol
ALL1/AF4  BCR/ABL E2A/PBX1      NEG   NUP-98  p15/p16
      10       37        5       74        1        1
```

A logical vector indicating that the corresponding row is either `BCR/ABL` or `NEG` is constructed as

```
> ridx <- pdata$mol.biol %in% c("BCR/ABL", "NEG")
```

We can get a sense of the number of rows selected via `table` or `sum` (discuss with your neighbor what `sum` does, and why the answer is the same as the number of `TRUE` values in the result of the `table` function).

```
> table(ridx)

ridx
FALSE  TRUE
   17   111

> sum(ridx)

[1] 111
```

The original data frame can be subset to contain only `BCR/ABL` or `NEG` samples using the logical vector `ridx` that we created.

```
> pdata1 <- pdata[ridx,]
```

The levels of each factor reflect the levels in the original object, rather than the levels in the subset object, e.g.,

```
> levels(pdata1$mol.biol)

[1] "ALL1/AF4" "BCR/ABL"  "E2A/PBX1" "NEG"      "NUP-98"   "p15/p16"
```

These can be re-coded by updating the new data frame to contain a factor with the desired levels.

```
> pdata1$mol.biol <- factor(pdata1$mol.biol)
> table(pdata1$mol.biol)

BCR/ABL     NEG
     37      74
```

To ask whether age differs between molecular biology types, we use a formula `age ~ mol.biol` to describe the relationship ('age as a function of molecular biology') that we wish to test

```
> with(pdata1, t.test(age ~ mol.biol))

        Welch Two Sample t-test

data:  age by mol.biol
t = 4.8172, df = 68.529, p-value = 8.401e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  7.13507 17.22408
sample estimates:
mean in group BCR/ABL     mean in group NEG
             40.25000              28.07042
```

This summary can be visualize with, e.g., the `boxplot` function

```
> ## not evaluated
> boxplot(age ~ mol.biol, pdata1)
```

Molecular biology seem to be strongly associated with age; individuals in the `NEG` group are considerably younger than those in the `BCR/ABL` group. We might wish to include age as a covariate in any subsequent analysis seeking to relate molecular biology to gene expression.

1.2 *Bioconductor*

Bioconductor is a collection of *R* packages for the analysis and comprehension of high-throughput genomic data. *Bioconductor* started more than 10 years ago, and is widely used. It gained credibility for its statistically rigorous approach to microarray pre-processing and analysis of designed experiments, and integrative and reproducible approaches to bioinformatic tasks. There are now more than 670 *Bioconductor* packages for expression and other microarrays, sequence analysis, flow cytometry, imaging, and other domains. The *Bioconductor* web site provides installation, package repository, help, and other documentation.

The *Bioconductor* web site is at `bioconductor.org`. Features include:

- Introductory work flows.
- A manifest of *Bioconductor* packages arranged in BiocViews.

Table 1.4: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	*IRanges*, *GenomicRanges*, *GenomicFeatures*, *Biostrings*, *BSgenome*, *girafe*.
Input / output	*ShortRead* (fastq), *Rsamtools* (bam), *rtracklayer* (gff, wig, bed), *VariantAnnotation* (vcf), *R453Plus1Toolbox* (454).
Annotation	*GenomicFeatures*, *ChIPpeakAnno*, *VariantAnnotation*.
Alignment	*gmapR*, *Rsubread*, *Biostrings*.
Visualization	*ggbio*, *Gviz*.
Quality assessment	*qrqc*, *seqbias*, *ReQON*, *htSeqTools*, *TEQC*, *Rolexa*, *ShortRead*.
RNA-seq	*BitSeq*, *cqn*, *cummeRbund*, *DESeq*, *DEXSeq*, *EDASeq*, *edgeR*, *gage*, *goseq*, *iASeq*, *tweeDEseq*.
ChIP-seq, etc.	*BayesPeak*, *baySeq*, *ChIPpeakAnno*, *chipseq*, *ChIPseqR*, *ChIPsim*, *CSAR*, *DiffBind*, *MEDIPS*, *mosaics*, *NarrowPeaks*, *nucleR*, *PICS*, *PING*, *REDseq*, *Repitools*, *TSSi*.
Variants	*VariantAnnotation*, *VariantTools*, *gmapR*
SNPs	*snpStats*, *GWASTools*, *hapFabia*, *GGtools*
Copy number	*cn.mops*, *genoset*, *CNAnorm*, *exomeCopy*, *seqmentSeq*.
Motifs	*MotifDb*, *BCRANK*, *cosmo*, *cosmoGUI*, *MotIV*, *seqLogo*, *rGADEM*.
3C, etc.	*HiTC*, *r3Cseq*.
Microbiome	*phyloseq*, *DirichletMultinomial*, *clstutils*, *manta*, *mcaGUI*.
Work flows	*QuasR*, *easyRNASeq*, *ArrayExpressHTS*, *Genominator*, *oneChannelGUI*, *rnaSeqMap*.
Database	*SRAdb*.

- Annotation (data bases of relevant genomic information, e.g., Entrez gene ids in model organisms, KEGG pathways) and experiment data (containing relatively comprehensive data sets and their analysis) packages.
- Mailing lists, including searchable archives, as the primary source of help.
- Course and conference information, including extensive reference material.
- General information about the project.
- Package developer resources, including guidelines for creating and submitting new packages.

1.2.1 Packages for high-throughput sequence analysis

Table 1.4 enumerates many of the packages available for sequence analysis. The table includes packages for representing sequence-related data (e.g., *GenomicRanges*, *Biostrings*), as well as domain-specific analysis such as RNA-seq (e.g., *edgeR*, *DEXSeq*), ChIP-seq (e.g,. *ChIPpeakAnno*, *DiffBind*), variants (e.g., *VariantAnnotation*, *VariantTools*, and SNPs and copy number variation (e.g., *genoset*, *ggtools*).

1.2.2 S4 Classes and methods

Bioconductor makes extensive use of 'S4' classes. Essential operations are sketched in Table 1.5. *Bioconductor* has tried to develop and use 'best practices' for use of S4 classes. Usually instances are created by a call to a *constructor*, such as `GRanges` (an object representing genomic ranges, with information on sequence, strand, start, and end coordinate of each range), or are returned by a function call that makes the object 'behind the scenes' (e.g., `readFastq`). Objects can have complicated structure, but users are not expected to have to concern themselves with the internal representation, just as the details of the S3 object returned by the `lm` function are not of direct concern. Instead, one might query the object to retrieve information; functions providing this functionality are sometimes called *accessors*, e.g., `seqnames`; the data that is returned by the

Table 1.5: Using S4 classes and methods.

Best practices	
`gr <- GRanges()`	'Constructor'; create an instance of the *GRanges* class
`seqnames(gr)`	'Accessor', extract information from an instance
`countOverlaps(gr1, gr2)`	A *method* implementing a *generic* with useful functionality
Older packages	
`s <- new("MutliSet)`	A constructor
`s@annotation`	A 'slot' accessor

accessor may involve some calculation, e.g., querying a data base, that the user can remain blissfully unaware of. It can be important to appreciate that objects can be related to one another, in particular inheriting parts of its internal structure and external behavior from other classes. For instance, the *GRanges* class inherits structure and behavior from the *GenomicRanges* class. The details of structural inheritance should not be important to the user, but the fact that once class inherits from another can be useful information to know.

One often calls a function in which one or more objects are arguments, e.g., `countOverlaps` can take two *GRanges* instances. The role of the function is to transform inputs into outputs. In the case of `countOverlaps` the transformation is to summarize the number of ranges in the second argument (the `subject` function argument) overlap with ranges in the first argument (the `query`) argument. This establishes a kind of contract, e.g., the return value of `countOverlaps` should be a non-negative integer vector, with as many elements as there are ranges in the `query` argument, and with a one-to-one correspondence between elements in the `query` input argument and the output. Having established such a contract, it can be convenient to write variations of `countOverlaps` that fulfill the contract but for different objects, e.g., when the arguments are instances of class *IRanges*, which do not have information about chromosomal sequence or strand. To indicate that the same contract is being fulfilled, and perhaps to simplify software development, one typically makes `countOverlaps` a *generic* function, and implements methods for different types of arguments.

Attending course and reading vignettes pages are obviously an excellent way to get an initial orientation about classes and methods that are available. It can be very helpful, as one becomes more proficient, to use the interactive help system to discover what can be done with the objects one has or the functions one knows about.

The `showMethods` function is a key entry point into discovery of available methods, e.g., `showMethods("countOverlaps")` to show methods defined on the `countOverlaps` generic, or `showMethods(class="GRanges", where=search())` to discover methods available to transform *GRanges* instances. The definition of a method can be retrieved as

```
> selectMethod(countOverlaps, c("GRanges", "GRanges"))
```

1.2.3 Help!

S4 classes, generics, and methods To illustrate how help work with S4 classes and generics, consider the *DNAStringSet* class `complement` generic in the Biostrings package:

```
> library(Biostrings)
> showMethods(complement)

Function: complement (package Biostrings)
x="DNAString"
x="DNAStringSet"
x="MaskedDNAString"
x="MaskedRNAString"
x="RNAString"
```

Table 1.6: Using S4 classes and methods.

Help	
S4 Help	
`class(gr)`	Discover class of instance
`getClass(gr)`	Display class structure, e.g., inheritance
`showMethods(findOverlaps)`	Classes for which methods of `findOverlaps` are implemented
`showMethods(class="GRanges", where=search())`	Generics with methods implemented for the *GRanges* class, limited to currently loaded packages.
`class?GRanges`	Documentation for the *GRanges* class.
`method?"findOverlaps,GRanges,GRanges"`	Documentation for the `findOverlaps` method when the two arguments are both *GRanges* instances.
`selectMethod(findOverlaps, c("GRanges", "GRanges"))`	View source code for the method, including method 'dispatch'

Table 1.7: Common ways to improve efficiency of *R* code.

Easy	Moderate
1. Selective input	1. Know relevant packages
2. Vectorize	2. Understand algorithm complexity
3. Pre-allocate and fill	3. Use parallel evaluation
4. Avoid expensive conveniences	4. Exploit libraries and `C++` code

```
x="RNAStringSet"
x="XStringViews"
```

(Most) methods defined on the `DNAStringSet` class of *Biostrings* and available on the current search path can be found with

```
> showMethods(class="DNAStringSet", where=search())
```

Obtaining help on S4 classes and methods requires syntax such as

```
> class ? DNAStringSet
> method ? "complement,DNAStringSet"
```

The specification of method and class in the latter must not contain a space after the comma.

1.3 Efficient *R*

There are often many ways to accomplish a result in *R*, but these different ways often have very different speed or memory requirements. For small data sets these performance differences are not that important, but for large data sets (e.g., high-throughput sequencing; genome-wide association studies, GWAS) or complicated calculations (e.g., bootstrapping) performance can be important. Several approaches to achieving efficient *R* programming are summarized in Table 1.7.

1.3.1 Easy solutions

Several common performance bottlenecks often have easy solutions; these are outlined here.

Text files often contain more information, for example 1000's of individuals at millions of SNPs, when only a subset of the data is required, e.g., during algorithm development. Reading in all the data can be demanding in terms of both memory and time. A solution is to use arguments such as `colClasses` to specify the columns and their data types that are required, and to use `nrow` to limit the number of rows input. For example, the following ignores the first and fourth column, reading in only the second and third (as type `integer` and `numeric`).

```
> ## not evaluated
> colClasses <-
+   c("NULL", "integer", "numeric", "NULL")
> df <- read.table("myfile", colClasses=colClasses)
```

R is vectorized, so traditional programming `for` loops are often not necessary. Rather than calculating 100000 random numbers one at a time, or squaring each element of a vector, or iterating over rows and columns in a matrix to calculate row sums, invoke the single function that performs each of these operations.

```
> x <- runif(100000); x2 <- x^2
> m <- matrix(x2, nrow=1000); y <- rowSums(m)
```

This often requires a change of thinking, turning the sequence of operations 'inside-out'. For instance, calculate the log of the square of each element of a vector by calculating the square of all elements, followed by the log of all elements `x2 <- x^2; x3 <- log(x2)`, or simply `logx2 <- log(x^2)`.

It may sometimes be natural to formulate a problem as a `for` loop, or the formulation of the problem may require that a `for` loop be used. In these circumstances the appropriate strategy is to pre-allocate the `result` object, and to fill the result in during loop iteration.

```
> ## not evaluated
> result <- numeric(nrow(df))
> for (i in seq_len(nrow(df)))
+   result[[i]] <- some_calc(df[i,])
```

Failure to pre-allocate and fill is the second cirle of *R* hell [2].

Some *R* operations are helpful in general, but misleading or inefficient in particular circumstances. An example is the behavior of `unlist` when the list is named – *R* creates new names that have been made unique. This can be confusing (e.g., when Entrez gene identifiers are 'mangled' to unintentionally look like other identifiers) and expensive (when a large number of new names need to be created). Avoid creating unnecessary names, e.g.,

```
> unlist(list(a=1:2)) # name 'a' becomes 'a1', 'a2'

a1 a2
 1  2

> unlist(list(a=1:2), use.names=FALSE)   # no names

[1] 1 2
```

Names can be very useful for avoiding book-keeping errors, but are inefficient for repeated look-ups; use vectorized access or numeric indexing.

Table 1.8: Tools for measuring performance.

Function	Description
`identical`	Compare content of objects.
`all.equal`	
`system.time`	Time required to evaluate an expression
`Rprof`	Time spent in each function; also `summaryRprof`.
`tracemem`	Indicate when memory copies occur (*R* must be configured to support this).
rbenchmark	Packages for standardizing speed measurment
microbenchmark	

1.3.2 Moderate solutions

Several solutions to inefficient code require greater knowledge to implement.

Using appropriate functions can greatly influence performance; it takes experience to know when an appropriate function exists. For instance, the `lm` function could be used to assess differential expression of each gene on a microarray, but the *limma* package implements this operation in a way that takes advantage of the experimental design that is common to each probe on the microarray, and does so in a very efficient manner.

Using appropriate algorithms can have significant performance benefits, especially as data becomes larger. This solution requires moderate skills, because one has to be able to think about the complexity (e.g., expected number of operations) of an algorithm, and to identify algorithms that accomplish the same goal in fewer steps. For example, a naive way of identifying which of 100 numbers are in a set of size 10 might look at all 100×10 combinations of numbers (i.e., polynomial time), but a faster way is to create a 'hash' table of one of the set of elements and probe that for each of the other elements (i.e., linear time). The latter strategy is illustrated with

```
> x <- 1:100; s <- sample(x, 10)
> inS <- x %in% s
```

R supports parallel evaluation, most easily through the `mclapply` function of the *parallel* package distributed with base *R* (`mclapply` is unfortunately not available on Windows). Parallel evaluation is discussed further in Chapter 5.

R is an interpreted language, and for very challenging computational problems it may be appropriate to write critical stages of an analysis in a compiled language like C or Fortran, or to use an existing programming library (e.g., the BOOST library) that efficiently implements advanced algorithms. *R* has a well-developed interface to C or Fortran, so it is 'easy' to do this; the *Rcpp* package provides a very nice approach for those familiar with C++ concepts. This places a significant burden on the person implementing the solution, requiring knowledge of two or more computer languages and of the interface between them.

1.3.3 Measuring performance

When trying to improve performance, one wants to ensure (a) that the new code is actually faster than the previous code, and (b) both solutions arrive at the same, correct, answer. common tools used to help with assessing performance (including comparison of results from different implementations!) are in Table 1.8.

The `system.time` function is a straight-forward way to measure the length of time a portion of code takes to evaluate.

```
> m <- matrix(runif(200000), 20000)
> system.time(apply(m, 1, sum))

   user  system elapsed 
  0.092   0.000   0.090 
```

When comparing performance of different functions, it is appropriate to replicate timings to average over vagaries of system use, and to shuffle the order in which timings of alternative algorithms are calculated to avoid artifacts such as initial memory allocation. Rather than creating *ad hoc* approaches to timing, it is convenient to use packages such as *rbenchmark*:

```
> library(rbenchmark)
> f0 <- function(x) apply(x, 1, sum)
> f1 <- function(x) rowSums(x)
> benchmark(f0(m), f1(m),
+           columns=c("test", "elapsed", "relative"),
+           replications=5)
```

```
   test elapsed relative
1 f0(m)   0.451   112.75
2 f1(m)   0.004     1.00
```

Speed is an important metric, but equivalent results are also needed. The functions `identical` and `all.equal` provide different levels of assessing equivalence, with `all.equal` providing ability to ignore some differences, e.g., in the names of vector elements.

```
> res1 <- apply(m, 1, sum)
> res2 <- rowSums(m)
> identical(res1, res2)
```

```
[1] TRUE
```

```
> identical(c(1, -1), c(x=1, y=-1))
```

```
[1] FALSE
```

```
> all.equal(c(1, -1), c(x=1, y=-1),
+           check.attributes=FALSE)
```

```
[1] TRUE
```

Two additional functions for assessing performance are `Rprof` and `tracemem`; these are mentioned only briefly here. The `Rprof` function profiles *R* code, presenting a summary of the time spent in each part of several lines of *R* code. It is useful for gaining insight into the location of performance bottlenecks when these are not readily apparent from direct inspection. Memory management, especially copying large objects, can frequently contribute to poor performance. The `tracemem` function allows one to gain insight into how *R* manages memory; insights from this kind of analysis can sometimes be useful in restructuring code into a more efficient sequence.

Chapter 2

Sequences, Reads, and Alignments

2.1 DNA (and other) strings with the *Biostrings* package

The *Biostrings* package provides tools for working with sequences. The essential data structures are *DNAString* and *DNAStringSet*, for working with one or multiple DNA sequences. The *Biostrings* package contains additional classes for representing amino acid and general biological strings.

2.1.1 Whole-genome sequences

There are a diversity of packages and classes available for representing large genomes. Several include:

TxDb.* For transcript and other genome / coordinate annotation.

BSgenome For whole-genome representation. See `available.packages` for pre-packaged genomes, and the vignette 'How to forge a BSgenome data package' in the

SNPlocs.* For model organism SNP locations derived from dbSNP.

`FaFile` (*Rsamtools*) for accessing indexed FASTA files.

Homo.sapiens For integrating *TxDb** and *org.** packages.

The *BSgenome* and related packages (e.g., *BSgenome.Dmelanogaster.UCSC.dm3*) are used to represent whole-genome sequences. Table 2.1 summarizes common operations.

Exercise 2

The objective of this exercise is to calculate the GC content of chromosomes 1-22, X, and Y.

Load the BSgenome.Hsapiens.UCSC.hg19 data package, containing the UCSC representation of H. sapiens *genome assembly hg19. Discover the content of the package by evaluating* `Hsapiens`.

Use `alphabetFrequency` *to develop a function* `gcContent` *that calculates GC content of a single chromosome.*

Use the `gcContent` *function to calculate the GC content on chromosome 1. Use the function to calculation GC content on chromosomes 1-22, X, and Y.*

Solution: Here we load the *H. sapiens* genome select a single chromosome, and create a light-weight 'view' onto the sequence at a particular (arbitrary) location.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> Hsapiens
> Hsapiens[[1]]
> Views(Hsapiens[[1]], 1000000, 1001000)
```

Here is the `gcContent` helper function to calculate GC content:

Table 2.1: Operations on strings in the *Biostrings* package.

	Function	Description
Access	`length`, `names`	Number and names of sequences
	`[`, `head`, `tail`, `rev`	Subset, first, last, or reverse sequences
	`c`	Concatenate two or more objects
	`width`, `nchar`	Number of letters of each sequence
	`Views`	Light-weight sub-sequences of a sequence
Compare	`==`, `!=`, `match`, `%in%`	Element-wise comparison
	`duplicated`, `unique`	Analog to `duplicated` and `unique` on character vectors
	`sort`, `order`	Locale-independent sort, order
	`split`, `relist`	Split or relist objects to, e.g., *DNAStringSetList*
Edit	`subseq`, `subseq<-`	Extract or replace sub-sequences in a set of sequences
	`reverse`, `complement`	Reverse, complement, or reverse-complement DNA
	`reverseComplement`	
	`translate`	Translate DNA to Amino Acid sequences
	`chartr`	Translate between letters
	`replaceLetterAt`	Replace letters at a set of positions by new letters
	`trimLRPatterns`	Trim or find flanking patterns
Count	`alphabetFrequency`	Tabulate letter occurrence
	`letterFrequency`	
	`letterFrequencyInSlidingView`	
	`consensusMatrix`	Nucleotide $\times$ position summary of letter counts
	`dinucleotideFrequency`	2-mer, 3-mer, and k-mer counting
	`trinucleotideFrequency`	
	`oligonucleotideFrequency`	
	`nucleotideFrequencyAt`	Nucleotide counts at fixed sequence positions
Match	`matchPattern`, `countPattern`	Short patterns in one or many (`v*`) sequences
	`vmatchPattern`, `vcountPattern`	
	`matchPDict`, `countPDict`	Short patterns in one or many (`v*`) sequences (mismatch only)
	`whichPDict`, `vcountPDict`	
	`vwhichPDict`	
	`pairwiseAlignment`	Needleman-Wunsch, Smith-Waterman, etc. pairwise alignment
	`matchPWM`, `countPWM`	Occurrences of a position weight matrix
	`matchProbePair`	Find left or right flanking patterns
	`findPalindromes`	Palindromic regions in a sequence. Also `findComplementedPalindromes`
	`stringDist`	Levenshtein, Hamming, or pairwise alignment scores
I/0	`readDNAStringSet`	FASTA (or sequence only from FASTQ). Also `readBStringSet`, `readRNAStringSet`, `readAAStringSet`
	`writeXStringSet`	
	`writePairwiseAlignments`	Write `pairwiseAlignment` as "pair" format
	`readDNAMultipleAlignment`	Multiple alignments (FASTA, "stockholm", or "clustal"). Also `readRNAMultipleAlignment`, `readAAMultipleAlignment`
	`write.phylip`	

```
> gcContent <-
+     function(x)
+ {
+     alf <- alphabetFrequency(x, as.prob=TRUE)
+     sum(alf[c("G", "C")])
+ }
```

The `gcContent` function is really straight-forward: it invokes the function `alphabetFrequency` from the *Biostrings* package. This returns a simple vector of nucleotide probabilities. The sum of the `G` and `C` elements of this vector is the GC content of the arugment.

Here is the GC content of chromosome 1

```
> gcContent(Hsapiens[["chr1"]])
```

```
[1] 0.4174393
```

and of all chromosomes

```
> chrs <- paste0("chr", c(1:22, "X", "Y"))
> gc <- numeric(length(chrs))            # pre-allocate...
> for (chr in seq_along(chrs))           # ...and fill
+     gc[[chr]] <- gcContent(Hsapiens[[chr]])
> names(gc) <- chrs
> gc
```

```
     chr1      chr2      chr3      chr4      chr5      chr6      chr7      chr8 
0.4174393 0.4024378 0.3969427 0.3824789 0.3951629 0.3961091 0.4075131 0.4017569 
     chr9     chr10     chr11     chr12     chr13     chr14     chr15     chr16 
0.4131684 0.4158488 0.4156565 0.4081198 0.3852654 0.4088715 0.4220095 0.4478943 
    chr17     chr18     chr19     chr20     chr21     chr22      chrX      chrY 
0.4554046 0.3978497 0.4836032 0.4412572 0.4083254 0.4798807 0.3949634 0.3996504 
```

2.2 Reads

2.2.1 Short read formats

The Illumina GAII and HiSeq technologies generate sequences by measuring incorporation of florescent nucleotides over successive PCR cycles. These sequencers produce output in a variety of formats, but *FASTQ* is ubiquitous. Each read is represented by a record of four components:

```
@SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
+SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
```

The first and third lines (beginning with `@` and `+` respectively) are unique identifiers. The identifier produced by the sequencer typically includes a machine id followed by colon-separated information on the lane, tile, x, and y coordinate of the read. The example illustrated here also includes the SRA accession number, added when the data was submitted to the archive. The machine identifier could potentially be used to extract information about batch effects. The spatial coordinates (lane, tile, x, y) are often used to identify optical duplicates; spatial coordinates can also be used during quality assessment to identify artifacts of sequencing, e.g., uneven amplification across the flow cell, though these spatial effects are rarely pursued.

The second and fourth lines of the FASTQ record are the nucleotides and qualities of each cycle in the read. This information is given in 5' to 3' orientation as seen by the sequencer. A letter `N` in the sequence is used to signify bases that the sequencer was not able to call. The fourth line of the FASTQ record encodes the quality (confidence) of the corresponding base call. The quality score is encoded following one of several conventions, with the general notion being that letters later in the visible ASCII alphabet

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

are of higher quality; this is developed further below. Both the sequence and quality scores may span multiple lines.

Technologies other than Illumina use different formats to represent sequences. Roche 454 sequence data is generated by 'flowing' labeled nucleotides over samples, with greater intensity corresponding to longer runs of A, C, G, or T. This data is represented as a series of 'flow grams' (a kind of run-length encoding of the read) in Standard Flowgram Format (SFF). The *Bioconductor* package *R453Plus1Toolbox* has facilities for parsing SFF files, but after quality control steps the data are frequently represented (with some loss of information) as FASTQ. SOLiD technologies produce sequence data using a 'color space' model. This data is not easily read in to *R*, and much of the error-correcting benefit of the color space model is lost when converted to FASTQ; SOLiD sequences are not well-handled by *Bioconductor* packages.

2.2.2 Short reads in *R*

FASTQ files can be read in to *R* using the `readFastq` function from the *ShortRead* package. Use this function by providing the path to a FASTQ file. There are sample data files available in the *SequenceAnalysisData* package, each consisting of 1 million reads from a lane of the Pasilla data set.

```
> library(ShortRead)
> bigdata <- system.file("bigdata", package="SequenceAnalysisData")
> fastqDir <- file.path(bigdata, "fastq")
> fastqFiles <- dir(fastqDir, full=TRUE)
> fq <- readFastq(fastqFiles[1])
> fq
```

```
class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

The data are represented as an object of class *ShortReadQ*.

```
> head(sread(fq), 3)
```

```
  A DNAStringSet instance of length 3
    width seq
[1]    37 GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
[2]    37 GTTGTCGCATTCCTTACTCTCATTCGGGAATTCTGTT
[3]    37 GAATTTTTTGAGAGCGAAATGATAGCCGATGCCCTGA
```

```
> head(quality(fq), 3)
```

```
class: FastqQuality
quality:
  A BStringSet instance of length 3
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
> head(id(fq), 3)

  A BStringSet instance of length 3
    width seq
[1]    58 SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
[2]    57 SRR031724.2 HWI-EAS299_4_30M2BAAXX:5:1:937:1157 length=37
[3]    58 SRR031724.4 HWI-EAS299_4_30M2BAAXX:5:1:1443:1122 length=37
```

The *ShortReadQ* class illustrates *class inheritance*. It extends the *ShortRead* class

```
> getClass("ShortReadQ")

Class "ShortReadQ" [package "ShortRead"]

Slots:

Name:       quality        sread           id
Class: QualityScore DNAStringSet   BStringSet

Extends:
Class "ShortRead", directly
Class ".ShortReadBase", by class "ShortRead", distance 2

Known Subclasses: "AlignedRead"
```

Methods defined on *ShortRead* are available for *ShortReadQ*.

```
> showMethods(class="ShortRead", where="package:ShortRead")
```

For instance, the `width` can be used to demonstrate that all reads consist of 37 nucleotides.

```
> table(width(fq))

     37
1000000
```

The `alphabetByCycle` function summarizes use of nucleotides at each cycle in a (equal width) *ShortReadQ* or *DNAStringSet* instance.

```
> abc <- alphabetByCycle(sread(fq))
> abc[1:4, 1:8]

        cycle
alphabet   [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
       A  78194 153156 200468 230120 283083 322913 162766 220205
       C 439302 265338 362839 251434 203787 220855 253245 287010
       G 397671 270342 258739 356003 301640 247090 227811 246684
       T  84833 311164 177954 162443 211490 209142 356178 246101
```

FASTQ files are getting larger. A very common reason for looking at data at this early stage in the processing pipeline is to explore sequence quality. In these circumstances it is often not necessary to parse the entire FASTQ file. Instead create a representative sample

```
> sampler <- FastqSampler(fastqFiles[1], 1000000)
> yield(sampler) # sample of 1000000 reads
```

```
class: ShortReadQ
length: 1000000 reads; width: 37 cycles
```

A second common scenario is to pre-process reads, e.g., trimming low-quality tails, adapter sequences, or artifacts of sample preparation. The *FastqStreamer* class can be used to 'stream' over the fastq files in chunks, processing each chunk independently.

2.2.3 Exercise

Exercise 3
Use the file path `bigdata` *and the* `file.path` *and* `dir` *functions to locate the fastq file from [1] (the file was obtained as described in the pasilla experiment data package).*

Input the fastq files using `readFastq` *from the ShortRead package.*

Use `alphabetFrequency` *to summarize the GC content of all reads (hint: use the* `sread` *accessor to extract the reads, and the* `collapse=TRUE` *argument to the* `alphabetFrequency` *function). Draw a histogram of the distribution of GC frequencies across reads.*

Solution: Discovery:

```
> bigdata <- system.file("bigdata", package="SequenceAnalysisData")
> fl <- file.path(bigdata, "fastq", "SRR031724_1_subset.fastq")
```

Input:

```
> fq <- readFastq(fls[1])
```

GC content:

```
> alf0 <- alphabetFrequency(sread(fq), as.prob=TRUE, collapse=TRUE)
> sum(alf0[c("G", "C")])
```

```
[1] 0.5457237
```

A histogram of the GC content of individual reads is obtained with

```
> alf0 <- alphabetFrequency(sread(fq), as.prob=TRUE)
> hist(alf0[,c("G", "C")])
```

2.3 Alignments

Most down-stream analysis of short read sequences is based on reads aligned to reference genomes.

2.3.1 Alignment formats

Most main-stream aligners produce output in SAM (text-based) or BAM format. A SAM file is a text file, with one line per aligned read, and fields separated by tabs. Here is an example of a single SAM line, split into fields.

```
 [1] EAS51_66:7:68:402:50                137
 [3] seq1                                22
 [5] 99                                  35M
 [7] *                                   0
 [9] 0                                   GTGTGGTTTAACTCGTCCATGGCCCAGCATTTGGG
[11] <<<<<<<<<<<<<<;:<<<9<6;9;;&697;7&<55 MF:i:18
[13] Aq:i:66                             NM:i:1
[15] UQ:i:5                              H0:i:1
[17] H1:i:0
```

Table 2.2: Fields in a SAM record. From http://samtools.sourceforge.net/samtools.shtml

Field	Name	Value
1	QNAME	Query (read) NAME
2	FLAG	Bitwise FLAG, e.g., strand of alignment
3	RNAME	Reference sequence NAME
4	POS	1-based leftmost POSition of sequence
5	MAPQ	MAPping Quality (Phred-scaled)
6	CIGAR	Extended CIGAR string
7	MRNM	Mate Reference sequence NaMe
8	MPOS	1-based Mate POSition
9	ISIZE	Inferred insert SIZE
10	SEQ	Query SEQuence on the reference strand
11	QUAL	Query QUALity
12+	OPT	OPTional fields, format TAG:VTYPE:VALUE

Fields in a SAM file are summarized in Table 2.2. We recognize from the FASTQ file the identifier string, read sequence and quality. The alignment is to a chromosome 'seq1' starting at position 1. The strand of alignment is encoded in the 'flag' field. The alignment record also includes a measure of mapping quality, and a CIGAR string describing the nature of the alignment. In this case, the CIGAR is 36M, indicating that the alignment consisted of 36 `M`atches or mismatches, with no indels or gaps; indels are represented by `I` and `D`; gaps (e.g., from alignments spanning introns) by `N`.

BAM files encode the same information as SAM files, but in a format that is more efficiently parsed by software; BAM files are the primary way in which aligned reads are imported in to *R*.

2.3.2 Aligned reads in *R*

The `readGappedAlignments` function from the *GenomicRanges* package reads essential information from a BAM file in to *R*. The result is an instance of the *GappedAlignments* class. The *GappedAlignments* class has been designed to allow useful manipulation of many reads (e.g., 20 million) under moderate memory requirements (e.g., 4 GB).

```
> library(RNAseqData.HeLa.bam.chr14)
> bamfls <- RNAseqData.HeLa.bam.chr14_BAMFILES
> aln <- readGappedAlignments(bamfls[1])
> head(aln, 3)
```

```
GappedAlignments with 3 alignments and 0 metadata columns:
      seqnames strand       cigar    qwidth     start       end     width
         <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
  [1]    chr14      +         72M        72  19069583  19069654        72
  [2]    chr14      +         72M        72  19363738  19363809        72
  [3]    chr14      -         72M        72  19363755  19363826        72
           ngap
      <integer>
  [1]         0
  [2]         0
  [3]         0
  ---
  seqlengths:
                    chr1                 chr10 ...                 chrY
               249250621             135534747 ...             59373566
```

A *GappedAlignments* instance is like a data frame, but with accessors as suggested by the column names. It is easy to query, e.g., the distribution of reads aligning to each strand, the width of reads, or the cigar strings

```
> table(strand(aln))

     +      -      *
400242 400242      0

> table(qwidth(aln))

    72
800484

> range(width(aln))

[1]     70 404751

> head(sort(table(cigar(aln)), decreasing=TRUE))

       72M 35M123N37M 38M670N34M  64M316N8M 36M123N36M 18M123N54M
    603939        272        264        261        228        225
```

The `readGappedAlignments` function takes an additional argument, `param`, allowing the user to specify regions of the BAM file (e.g., known gene coordinates) from which to extract alignments, or other elements (such as the read sequence) to be extracted from the BAM file. The data input with these extra arguments is accessed with the `mcols` function.

```
> param <- ScanBamParam(what="seq")
> aln <- readGappedAlignments(bamfls[1], param=param)
> head(mcols(aln)$seq)

  A DNAStringSet instance of length 6
    width seq
[1]    72 TGAGAATGATGATTTCCAATTTCATCCATGTCCC...AGGACATGAACTCATCATTTTTATGGCTGCAT
[2]    72 CCCATATGTACATCAGGCCCCAGGTATACACTGG...AGGTGGACACCAGCACTCAGTTGGATACACACA
[3]    72 CCCCAGGTATACACTGGACTCCAGGTGGACACCA...CAGTTGGATACACACACTCAAGGTGGACACCAG
[4]    72 CATAGATGCAAGAATCCTCAATCAAATACTAGCA...AATTCAACAGCACATTAAAAAGATAACTTACCA
[5]    72 TAGCACACTGAATTCAACAGCACATTAAAAAGAT...ACCATGCTCAAGTGGATTTACCCCAAGGATACA
[6]    72 TGCTGGTGCAGGATTTATTCTACTAAGCAATGAG...GGATCAAATCCACTTTCTTATCTCAGGAATCAG
```

2.3.3 Exercise

Exercise 4
What is the GC content of aligned reads in each of the BAM files in `bamfls`*?*

Solution: Here is a variation of our now familiar helper function...

```
> bamGCContent <-
+     function(bamfl)
+ {
+     param <- ScanBamParam(what="seq")
+     aln <- readGappedAlignments(bamfl, param=param)
+     seq <- mcols(aln)$seq
+     alf0 <- alphabetFrequency(seq, as.prob=TRUE, collapse=TRUE)
+     sum(alf0[c("G", "C")])
+ }
```

applied to one file

```
> bamGCContent(bamfls[1])
```

```
[1] 0.5462378
```

and to all BAM files

```
> sapply(bamfls, bamGCContent)
```

```
ERR127306 ERR127307 ERR127308 ERR127309 ERR127302 ERR127303 ERR127304 ERR127305
0.5462378 0.5356166 0.5440062 0.5372698 0.5204686 0.5025465 0.5197887 0.5347569
```

Chapter 3

Ranges

Ranges describe both features of interest (e.g., genes, exons, promoters) and reads aligned to the genome. *Bioconductor* has very powerful facilities for working with ranges, some of which are summarized in Table 3.1.

3.1 The *GenomicRanges* package

Next-generation sequencing data consists of a large number of short reads. These are, typically, aligned to a reference genome. Basic operations are performed on the alignment, asking e.g., how many reads are aligned in a genomic range defined by nucleotide coordinates (e.g., in the exons of a gene), or how many nucleotides from all the aligned reads cover a set of genomic coordinates. How is this type of data, the aligned reads and the reference genome, to be represented in *R* in a way that allows for effective computation?

The *IRanges*, *GenomicRanges*, and *GenomicFeatures* *Bioconductor* packages provide the essential infrastructure for these operations; we start with the *GRanges* class, defined in *GenomicRanges*.

3.1.1 The *GRanges* class

Instances of *GRanges* are used to specify genomic coordinates. Suppose we wish to represent two *D. melanogaster* genes. The first is located on the positive strand of chromosome 3R, from position 19967117 to 19973212. The second is on the minus strand of the X chromosome, with 'left-most' base at 18962306, and right-most base at 18962925. The coordinates are *1-based* (i.e., the first nucleotide on a chromosome is numbered 1, rather than 0), *left-most* (i.e., reads on the minus strand are defined to 'start' at the left-most coordinate, rather than the 5' coordinate), and *closed* (the start and end coordinates are included in the range; a range with identical start and end coordinates has width 1, a 0-width range is represented by the special construct where the end coordinate is one less than the start coordinate).

Table 3.1: Selected *Bioconductor* packages for representing and manipulating ranges, strings, and other data structures.

Package	Description
IRanges	Defines important classes (e.g., *IRanges*, *Rle*) and methods (e.g., `findOverlaps`, `countOverlaps`) for representing and manipulating ranges of consecutive values. Also introduces *DataFrame*, *SimpleList* and other classes tailored to representing very large data.
GenomicRanges	Range-based classes tailored to sequence representation (e.g., *GRanges*, *GRangesList*), with information about strand and sequence name.
GenomicFeatures	Foundation for manipulating data bases of genomic ranges, e.g., representing coordinates and organization of exons and transcripts of known genes.

A complete definition of these genes as *GRanges* is:

```
> genes <- GRanges(seqnames=c("3R", "X"),
+                  ranges=IRanges(
+                      start=c(19967117, 18962306),
+                      end=c(19973212, 18962925)),
+                  strand=c("+", "-"),
+                  seqlengths=c(`3R`=27905053L, `X`=22422827L))
```

The components of a *GRanges* object are defined as vectors, e.g., of `seqnames`, much as one would define a *data.frame*. The start and end coordinates are grouped into an *IRanges* instance. The optional `seqlengths` argument specifies the maximum size of each sequence, in this case the lengths of chromosomes 3R and X in the 'dm2' build of the *D. melanogaster* genome. This data is displayed as

```
> genes

GRanges with 2 ranges and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]       3R [19967117, 19973212]      +
  [2]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

The *GRanges* class has many useful methods defined on it. Consult the help page

```
> ?GRanges
```

and package vignettes (especially 'An Introduction to GenomicRanges')

```
> vignette(package="GenomicRanges")
```

for a comprehensive introduction. A *GRanges* instance can be subset, with accessors for getting and updating information.

```
> genes[2]

GRanges with 1 range and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827

> strand(genes)

factor-Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : + -
Levels(3): + - *

> width(genes)
```

```
[1] 6096  620

> length(genes)

[1] 2

> names(genes) <- c("FBgn0039155", "FBgn0085359")
> genes  # now with names

GRanges with 2 ranges and 0 metadata columns:
              seqnames               ranges strand
                 <Rle>            <IRanges>  <Rle>
  FBgn0039155       3R [19967117, 19973212]      +
  FBgn0085359        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

`strand` returns the strand information in a compact representation called a *run-length encoding*, this is introduced in greater detail below. The 'names' could have been specified when the instance was constructed; once named, the *GRanges* instance can be subset by name like a regular vector.

As the `GRanges` function suggests, the *GRanges* class extends the *IRanges* class by adding information about `seqnames`, `strand`, and other information particularly relevant to representing ranges that are on genomes. The *IRanges* class and related data structures (e.g., *RangedData*) are meant as a more general description of ranges defined in an arbitrary space. Many methods implemented on the *GRanges* class are 'aware' of the consequences of genomic location, for instance treating ranges on the minus strand differently (reflecting the 5' orientation imposed by DNA) from ranges on the plus strand.

3.1.2 Operations on ranges

The *GRanges* class has many useful methods from the *IRanges* class; some of these methods are illustrated here. We use *IRanges* to illustrate these operations to avoid complexities associated with strand and seqnames, but the operations are comparable on *GRanges*. We begin with a simple set of ranges:

```
> ir <- IRanges(start=c(7, 9, 12, 14, 22:24),
+               end=c(15, 11, 12, 18, 26, 27, 28))
```

These and some common operations are illustrated in the upper panel of Figure 3.1 and summarized in Table 3.2.

Methods on ranges can be grouped as follows:

Intra-range methods act on each range independently. These include `flank`, `narrow`, `reflect`, `resize`, `restrict`, and `shift`, among others. An illustration is `shift`, which translates each range by the amount specified by the `shift` argument. Positive values shift to the right, negative to the left; `shift` can be a vector, with each element of the vector shifting the corresponding element of the *IRanges* instance. Here we shift all ranges to the right by 5, with the result illustrated in the middle panel of Figure 3.1.

```
> shift(ir, 5)

IRanges of length 7
    start end width
[1]    12  20     9
[2]    14  16     3
```

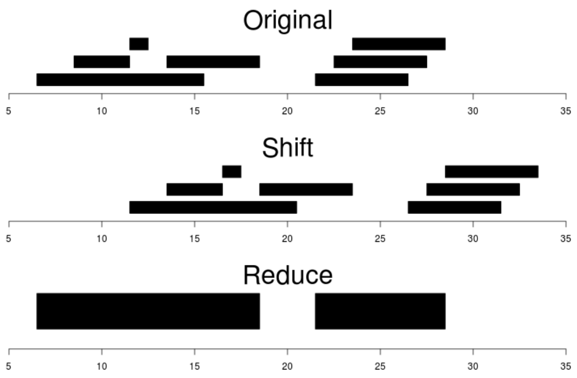


Figure 3.1: Ranges

```
[3]     17  17      1
[4]     19  23      5
[5]     27  31      5
[6]     28  32      5
[7]     29  33      5
```

Inter-range methods act on the collection of ranges as a whole. These include `disjoin`, `reduce`, `gaps`, and `range`. An illustration is `reduce`, which reduces overlapping ranges into a single range, as illustrated in the lower panel of Figure 3.1.

```
> reduce(ir)
```

```
IRanges of length 2
    start end width
[1]     7  18    12
[2]    22  28     7
```

`coverage` is an inter-range operation that calculates how many ranges overlap individual positions. Rather than returning ranges, `coverage` returns a compressed representation (run-length encoding)

```
> coverage(ir)
```

```
integer-Rle of length 28 with 12 runs
  Lengths: 6 2 4 1 2 3 3 1 1 3 1 1
  Values : 0 1 2 1 2 1 0 1 2 3 2 1
```

The run-length encoding can be interpreted as 'a run of length 6 of nucleotides covered by 0 ranges, followed by a run of length 2 of nucleotides covered by 1 range. . . '.

Between methods act on two (or sometimes more) *IRanges* instances. These include `intersect`, `setdiff`, `union`, `pintersect`, `psetdiff`, and `punion`.
The `countOverlaps` and `findOverlaps` functions operate on two sets of ranges. `countOverlaps` takes its first argument (the `query`) and determines how many of the ranges in the second argument (the `subject`) each overlaps. The result is an integer vector with one element for each member of `query`. `findOverlaps` performs a similar operation but returns a more general matrix-like structure that identifies each pair of query / subject overlaps. Both arguments allow some flexibility in the definition of 'overlap'.

Common operations on ranges are summarized in Table 3.2.

Table 3.2: Common operations on *IRanges*, *GRanges* and *GRangesList*.

Category	Function	Description
Accessors	`start`, `end`, `width`	Get or set the starts, ends and widths
	`names`	Get or set the names
	`mcols`, `metadata`	Get or set metadata on elements or object
	`length`	Number of ranges in the vector
	`range`	Range formed from min start and max end
Ordering	`<`, `<=`, `>`, `>=`, `==`, `!=`	Compare ranges, ordering by start then width
	`sort`, `order`, `rank`	Sort by the ordering
	`duplicated`	Find ranges with multiple instances
	`unique`	Find unique instances, removing duplicates
Arithmetic	`r + x`, `r - x`, `r * x`	Shrink or expand ranges `r` by number `x`
	`shift`	Move the ranges by specified amount
	`resize`	Change width, anchoring on start, end or mid
	`distance`	Separation between ranges (closest endpoints)
	`restrict`	Clamp ranges to within some start and end
	`flank`	Generate adjacent regions on start or end
Set operations	`reduce`	Merge overlapping and adjacent ranges
	`intersect`, `union`, `setdiff`	Set operations on reduced ranges
	`pintersect`, `punion`, `psetdiff`	Parallel set operations, on each `x[i]`, `y[i]`
	`gaps`, `pgap`	Find regions not covered by reduced ranges
	`disjoin`	Ranges formed from union of endpoints
Overlaps	`findOverlaps`	Find all overlaps for each `x` in `y`
	`countOverlaps`	Count overlaps of each `x` range in `y`
	`nearest`	Find nearest neighbors (closest endpoints)
	`precede`, `follow`	Find nearest `y` that `x` precedes or follows
	`x %in% y`	Find ranges in `x` that overlap range in `y`
Coverage	`coverage`	Count ranges covering each position
Extraction	`r[i]`	Get or set by logical or numeric index
	`r[[i]]`	Get integer sequence from `start[i]` to `end[i]`
	`subsetByOverlaps`	Subset `x` for those that overlap in `y`
	`head`, `tail`, `rev`, `rep`	Conventional R semantics
Split, combine	`split`	Split ranges by a factor into a *RangesList*
	`c`	Concatenate two or more range objects

3.1.3 Adding data to *GRanges*

The *GRanges* class (actually, most of the data structures defined or extending those in the *IRanges* package) has two additional very useful data components. The `mcols` function allows information on each range to be stored and manipulated (e.g., subset) along with the *GRanges* instance. The element metadata is represented as a *DataFrame*, defined in *IRanges* and acting like a standard *R data.frame* but with the ability to hold more complicated data structures as columns (and with element metadata of its own, providing an enhanced alternative to the *Biobase* class *AnnotatedDataFrame*).

```
> mcols(genes) <- DataFrame(EntrezId=c("42865", "2768869"),
+                           Symbol=c("kal-1", "CG34330"))
```

`metadata` allows addition of information to the entire object. The information is in the form of a list; any data can be provided.

```
> metadata(genes) <-
+     list(CreatedBy="A. User", Date=date())
```

3.1.4 *GRangesList*

The `GRanges` class is extremely useful for representing simple ranges. Some next-generation sequence data and genomic features are more hierarchically structured. A gene may be represented by several exons within it. An aligned read may be represented by discontinuous ranges of alignment to a reference. The *GRangesList* class represents this type of information. It is a list-like data structure, which each element of the list itself a *GRanges* instance. The gene FBgn0039155 contains several exons, and can be represented as a list of length 1, where the element of the list contains a *GRanges* object with 7 elements:

```
GRangesList of length 1:
$FBgn0039155
GRanges with 7 ranges and 2 metadata columns:
      seqnames               ranges strand |   exon_id   exon_name
         <Rle>            <IRanges>  <Rle> | <integer> <character>
  [1]    chr3R [19967117, 19967382]      + |     45515        <NA>
  [2]    chr3R [19970915, 19971592]      + |     45516        <NA>
  [3]    chr3R [19971652, 19971770]      + |     45517        <NA>
  [4]    chr3R [19971831, 19972024]      + |     45518        <NA>
  [5]    chr3R [19972088, 19972461]      + |     45519        <NA>
  [6]    chr3R [19972523, 19972589]      + |     45520        <NA>
  [7]    chr3R [19972918, 19973212]      + |     45521        <NA>

---
seqlengths:
    chr3R
 27905053
```

The *GRangesList* object has methods one would expect for lists (e.g., `length`, sub-setting). Many of the methods introduced for working with *IRanges* are also available, with the method applied element-wise.

3.2 Exercises: *GRanges* in action

See course supplement.

Chapter 4

Annotation and Visualization

4.1 Annotation

4.1.1 Genes and pathways: *org.** and friends

Bioconductor provides extensive annotation resources, summarized in Figure 4.1. These can be *gene*-, or *genome*-centric. Annotations can be provided in packages curated by *Bioconductor*, or obtained from web-based resources. Gene-centric *AnnotationDbi* packages include:

- Organism level: e.g. *org.Mm.eg.db*, *Homo.sapiens*.
- Platform level: e.g. *hgu133plus2.db*, *hgu133plus2.probes*, *hgu133plus2.cdf*.
- Homology level: e.g. *hom.Dm.inp.db*.
- System biology level: *GO.db*, *KEGG.db*, *Reactome.db*.

Organism-level ('org') packages contain mappings between a central identifier (e.g., Entrez gene ids) and other identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an org package is always of the form *org.<Sp>.<id>.db* (e.g. *org.Sc.sgd.db*) where *<Sp>* is a 2-letter abbreviation of the organism (e.g. `Sc` for *Saccharomyces cerevisiae*) and *<id>* is an abbreviation (in lower-case) describing the type of central identifier (e.g. `sgd` for gene identifiers assigned by the *Saccharomyces* Genome Database, or `eg` for Entrez gene ids). The "How to use the '.db' annotation packages" vignette in the *AnnotationDbi* package (org packages are only one type of ".db" annotation packages) is a key reference. The '.db' and most other *Bioconductor* annotation packages are updated every 6 months.

Annotation packages contain an object named after the package itself. These objects are collectively called *AnnotationDb* objects, with more specific classes named *OrgDb*, *ChipDb* or *TranscriptDb* objects. Methods that can be applied to these objects include `cols`, `keys`, `keytypes` and `select`. Common operations for retrieving annotations are summarized in Table 4.1.

4.1.2 Gene models: *TxDb.** packages

Examples of packages providign gene models include:

- *GenomicFeatures*, to represent genomic features, including constructing reproducible feature or transcript data bases from file or web resources.
- Pre-built transcriptome packages, e.g. *TxDb.Hsapiens.UCSC.hg19.knownGene* based on the *H. sapiens* UCSC hg19 knownGenes track.

Genome-centric packages are very useful for annotations involving genomic coordinates. It is straightforward, for instance, to discover the coordinates of coding sequences in regions of interest, and from these retrieve corresponding DNA or protein coding sequences. Other examples of the types of operations that

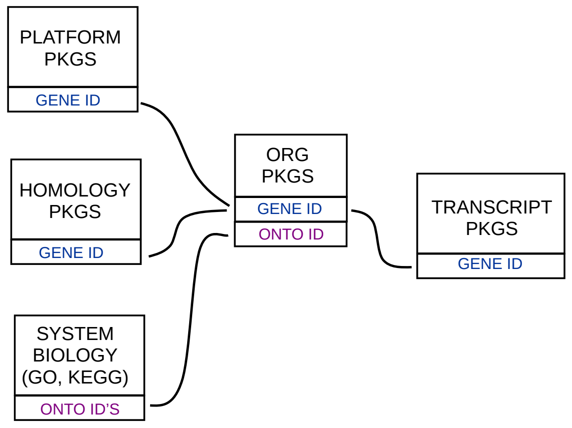


Figure 4.1: Annotation Packages: the big picture

Table 4.1: Common operations for retrieving and manipulating annotations.

Category	Function	Description
Discover	`cols`	List the kinds of columns that can be returned
	`keytypes`	List columns that can be used as keys
	`keys`	List values that can be expected for a given keytype
	`select`	Retrieve annotations matching `keys`, `keytype` and `cols`
Manipulate	`setdiff`, `union`, `intersect`	Operations on sets
	`duplicated`, `unique`	Mark or remove duplicates
	`%in%`, `match`	Find matches
	`any`, `all`	Are any `TRUE`? Are all?
	`merge`	Combine two different `data.frames` based on shared keys
*GRanges**	`transcripts`, `exons`, `cds`	Features (transcripts, exons, coding sequence) as *GRanges*.
	`transcriptsBy` , `exonsBy` `cdsBy`	Features group by gene, transcript, etc., as *GRangesList*.

Table 4.2: Selected packages querying web-based annotation services.

Package	Description
biomaRt	`http://biomart.org`, Ensembl and other annotations
rtracklayer	`http://genome.ucsc.edu`, genome tracks.
AnnotationHub	Ensembl, Encode, dbSNP, UCSC data objects
uniprot.ws	`http://uniprot.org`, protein annotations
KEGGREST	`http://www.genome.jp/kegg`, KEGG pathways
SRAdb	`http://www.ncbi.nlm.nih.gov/sra`, sequencing experiments.
GEOquery	`http://www.ncbi.nlm.nih.gov/geo/`, array and other data
ArrayExpress	`http://www.ebi.ac.uk/arrayexpress/`, array and other data

are easy to perform with genome-centric annotations include defining regions of interest for counting aligned reads in RNA-seq experiments and retrieving DNA sequences underlying regions of interest in ChIP-seq analysis, e.g., for motif characterization.

4.1.3 Tracks and on-line resources

A short summary of select *Bioconductor* packages enabling web-based queries is in Table 4.2. As an example, The *biomaRt* package offers access to the online biomart[1] resource. This consists of several data base resources, referred to as 'marts'. Each mart allows access to multiple data sets; the *biomaRt* package provides methods for mart and data set discovery, and a standard method `getBM` to retrieve data. An exercise at the end of this section walks through use of *biomaRt*.

4.1.4 Called variants with *VariantAnnotation*

A major product of DNASeq experiments are catalogs of called variants (e.g., SNPs, indels). The *VariantAnnotation* package allows us to explore this type of data. Sample data included in the package are a subset of chromosome 22 from the 1000 Genomes project. Variant Call Format (VCF; full description) text files contain meta-information lines, a header line with column names, data lines with information about a position in the genome, and optional genotype information on samples for each position.

Important operations on VCF files available with the *VariantAnnotation* package are summarized in Table 4.3.

To illustrate *VariantAnnotation*, we load the package and subset of a 1000 genomes data file.

```
> library(VariantAnnotation)
> fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
```

Variants can be easily screened for presence in dbSNP, classified to region such as `coding`, `intron`, `intergenic`, `spliceSite` etc. Amino acid coding changes are computed for the non-synonymous variants. The Ensembl Variant Effect Predictor (accessed via *ensemblVEP* provide predictions of how severely the coding changes affect protein function. Additional annotations are easily crafted using the *GenomicRanges* and *GenomicFeatures* software in conjunction with *Bioconductor* and broader community annotation resources.

The *ensemblVEP* package 'wraps' a perl script provided by Ensembl, so users need to have perl and the script installed. Instructions for installation are in the package vignette[2] (this is already configured on the AMI used in this training). Basic use is to invoke the `ensemblVEP` function with the path to a VCF file:

```
> library(ensemblVEP)
> fl <- system.file("extdata", "gl_chr1.vcf", package="VariantAnnotation")
> gr <- ensemblVEP(fl)
```

[1]urlhttp://www.biomart.org
[2]`http://bioconductor.org/packages/release/bioc/vignettes/ensemblVEP/inst/doc/ensemblVEP.pdf`

Table 4.3: Working with *VCF* files and data.

Category	Function	Description
Read	`scanVcfHeader`	Retrieve file header information
	`scanVcfParam`	Select fields to read in
	`readVcf`	Read VCF file into a VCF class
	`scanVcf`	Read VCF file into a list
Filter	`filterVcf`	Filter a VCF from one file to another
Write	`writeVcf`	Write a VCF file to disk
Annotate	`locateVariants`	Identify where variant overlaps a gene annotation
	`predictCoding`	Amino acid changes for variants in coding regions
	`summarizeVariants`	Summarize variant counts by sample
SNPs	`genotypeToSnpMatrix`	Convert genotypes to a SnpMatrix
	`GLtoGP`	Convert genotype likelihoods to genotypes
	`snpSummary`	Counts and distribution statistics for SNPs
Manipulate	`expand`	Convert CompressedVCF to ExpandedVCF
	`cbind`, `rbind`	Combine by column or row

This returns the 'consequence' of each variant.

```
> head(gr, 3)
```

```
GRanges with 3 ranges and 13 metadata columns:
             seqnames         ranges strand |   Allele            Gene
                <Rle>      <IRanges>  <Rle> | <factor>        <factor>
  rs58108140        1 [10583, 10583]      * |        A ENSG00000223972
  rs58108140        1 [10583, 10583]      * |        A ENSG00000227232
  rs58108140        1 [10583, 10583]      * |        A ENSG00000227232
                     Feature Feature_type            Consequence cDNA_position
                    <factor>     <factor>               <factor>      <factor>
  rs58108140 ENST00000456328   Transcript   upstream_gene_variant          <NA>
  rs58108140 ENST00000488147   Transcript downstream_gene_variant          <NA>
  rs58108140 ENST00000541675   Transcript downstream_gene_variant          <NA>
             CDS_position Protein_position Amino_acids   Codons
                 <factor>         <factor>    <factor> <factor>
  rs58108140         <NA>             <NA>        <NA>     <NA>
  rs58108140         <NA>             <NA>        <NA>     <NA>
  rs58108140         <NA>             <NA>        <NA>     <NA>
             Existing_variation DISTANCE CELL_TYPE
                       <factor> <factor>  <factor>
  rs58108140               <NA>     1286      <NA>
  rs58108140               <NA>     3821      <NA>
  rs58108140               <NA>     3780      <NA>
  ---
  seqlengths:
    1
   NA
```

The `VEPParam()` function controls many aspects of VEP evaluation, with arguments divided into `base`, `input`, `database`, `output`, and `filterqc` components; these are described more completely on `?VEPParam` and closely track the options documented on the VEP web site[3].

[3]http://www.ensembl.org/info/docs/variation/vep/vep_script.html

One useful option is to return the results from VEP as a `VCF` object.

```
> fl <- system.file("extdata", "structural.vcf", package="VariantAnnotation")
> param <- VEPParam(input=c(format="vcf"), output=c(vcf=TRUE))
> vep <- ensemblVEP(fl, param)
```

The 'consequence' data are then returned in a compact format as part of the VCF INFO column labeled *CSQ*

```
> head(info(vep)$CSQ, 3)
```

```
CharacterList of length 3
[[1]] -||||intergenic_variant||||||||
[[2]] deletion|ENSG00000233684|ENST00000430529|Transcript|intron_variant&nc_t...
[[3]] deletion|ENSG00000230448|ENST00000418420|Transcript|intron_variant&nc_t...
```

This can be unpacked to a more friendly *GRanges* instance with

```
> csq <- parseCSQToGRanges(vep)
> head(csq, 3)
```

```
GRanges with 3 ranges and 14 metadata columns:
            seqnames             ranges strand |  VCFRowID   Allele
               <Rle>          <IRanges>  <Rle> | <integer> <factor>
  1:2827693        1 [2827693, 2827762]      * |         1        -
   2:321682        2 [ 321682,  321682]      * |         2 deletion
   2:321682        2 [ 321682,  321682]      * |         2 deletion
                       Gene         Feature Feature_type
                   <factor>        <factor>     <factor>
  1:2827693            <NA>            <NA>         <NA>
   2:321682 ENSG00000233684 ENST00000430529   Transcript
   2:321682 ENSG00000233684 ENST00000436808   Transcript
                                                       Consequence
                                                          <factor>
  1:2827693                                     intergenic_variant
   2:321682 intron_variant&nc_transcript_variant&feature_truncation
   2:321682 intron_variant&nc_transcript_variant&feature_truncation
            cDNA_position CDS_position Protein_position Amino_acids   Codons
                 <factor>     <factor>         <factor>    <factor> <factor>
  1:2827693          <NA>         <NA>             <NA>        <NA>     <NA>
   2:321682          <NA>         <NA>             <NA>        <NA>     <NA>
   2:321682          <NA>         <NA>             <NA>        <NA>     <NA>
            Existing_variation DISTANCE CELL_TYPE
                      <factor> <factor>  <factor>
  1:2827693               <NA>     <NA>      <NA>
   2:321682               <NA>     <NA>      <NA>
   2:321682               <NA>     <NA>      <NA>
  ---
  seqlengths:
    1  2  3  4
   NA NA NA NA
```

4.1.5 Exercises

Exercise 5

What is the name of the org package for H. sapiens*? Load it. Display the OrgDb object for the* org.Hs.eg.db *package. Use the* `cols` *method to discover which sorts of annotations can be extracted from it.*

Use the `keys` *method to extract UNIPROT identifiers and then pass those keys in to the* `select` *method in such a way that you extract the SYMBOL (gene symbol) and KEGG pathway information for each.*

Use `select` *to retrieve the ENTREZ and SYMBOL identifiers of all genes in the KEGG pathway* `00310`*.*

Solution: The *OrgDb* object is named `org.Hs.eg.db`.

```
> library(org.Hs.eg.db)
> cols(org.Hs.eg.db)

 [1] "ENTREZID"     "PFAM"         "IPI"          "PROSITE"      "ACCNUM"
 [6] "ALIAS"        "CHR"          "CHRLOC"       "CHRLOCEND"    "ENZYME"
[11] "MAP"          "PATH"         "PMID"         "REFSEQ"       "SYMBOL"
[16] "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GENENAME"
[21] "UNIPROT"      "GO"           "EVIDENCE"     "ONTOLOGY"     "GOALL"
[26] "EVIDENCEALL"  "ONTOLOGYALL"  "OMIM"         "UCSCKG"

> keytypes(org.Hs.eg.db)

 [1] "ENTREZID"     "PFAM"         "IPI"          "PROSITE"      "ACCNUM"
 [6] "ALIAS"        "CHR"          "CHRLOC"       "CHRLOCEND"    "ENZYME"
[11] "MAP"          "PATH"         "PMID"         "REFSEQ"       "SYMBOL"
[16] "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GENENAME"
[21] "UNIPROT"      "GO"           "EVIDENCE"     "ONTOLOGY"     "GOALL"
[26] "EVIDENCEALL"  "ONTOLOGYALL"  "OMIM"         "UCSCKG"

> uniprotKeys <- head(keys(org.Hs.eg.db, keytype="UNIPROT"))
> cols <- c("SYMBOL", "PATH")
> select(org.Hs.eg.db, keys=uniprotKeys, cols=cols, keytype="UNIPROT")

   UNIPROT SYMBOL  PATH
1   P04217   A1BG  <NA>
2   P01023    A2M 04610
3   F5H5R8   NAT1 00232
4   F5H5R8   NAT1 00983
5   F5H5R8   NAT1 01100
6   P18440   NAT1 00232
7   P18440   NAT1 00983
8   P18440   NAT1 01100
9   Q400J6   NAT1 00232
10  Q400J6   NAT1 00983
11  Q400J6   NAT1 01100
12  A4Z6T7   NAT2 00232
13  A4Z6T7   NAT2 00983
14  A4Z6T7   NAT2 01100
```

Selecting UNIPROT and SYMBOL ids of KEGG pathway `00310` is very similar:

```
> kegg <- select(org.Hs.eg.db, "00310", c("UNIPROT", "SYMBOL"), "PATH")
> nrow(kegg)
```

```
[1] 57
```

```
> head(kegg, 3)
```

```
   PATH UNIPROT SYMBOL
1 00310  P24752  ACAT1
2 00310  Q9BWD1  ACAT2
3 00310  B4DW54  ALDH2
```

Exercise 6
(Adapted from[4]) Suppose you have a list of transcription factor binding sites on hg19. How would you obtain (a) the GC content of each site and (b) the percentage of gene promoters covered by the binding sites?

Solution: As an outline of a solution, the steps for calculating GC content might be

a. Represent the list of transcription factor binding sites ('regions of interest') as a *GRanges* instance, `roi`.
b. Load *BSgenome* and the appropriate genome package, e.g., *BSgenome.Hsapiens.UCSC.hg19*.
c. Use `getSeq` to retrieve the sequences, `seqs <- getSeq(Hsapiens, gr)`.
d. Use `alphabetFrequency(seqs)` to summarize nucleotide use, and simple *R* functions to determine GC content of each region of interest.
e. Summarize these as density plots, etc. A meaningful extension of this exercise might compare the observed GC content to the expected content, where expectation is the product of the independent G and C frequencies.

To calculate the percentage of promoters covered by binding sites, we might

a. Load the reference genome *TxDb* package, *TxDb.Hsapiens.UCSC.hg19.knownGene*.
b. Query the package for promoters using the `promoters` function, or otherwise manipulating exon or transcript coordinates to get a *GRanges* or *GRangesList* representing genomic regions of interest, `groi`.
c. Use `countOverlaps(groi, roi)` to find how many transcription factor binding sites overlap each promoter, and from there use standard *R* functions to tally the number of promoters that have zero overlaps.

Exercise 7
Load the biomaRt *package and list the available marts. Choose the* ensembl *mart and list the datasets for that mart. Set up a mart to use the* ensembl *mart and the* hsapiens_gene_ensembl *dataset.*

A biomaRt *dataset can be accessed via* `getBM`*. In addition to the mart to be accessed, this function takes filters and attributes as arguments. Use* `filterOptions` *and* `listAttributes` *to discover values for these arguments. Call* `getBM` *using filters and attributes of your choosing.*

Solution:

```
> library(biomaRt)
> head(listMarts(), 3)                      ## list the marts
> head(listDatasets(useMart("ensembl")), 3) ## mart datasets
> ensembl <-                                ## fully specified mart
+     useMart("ensembl", dataset = "hsapiens_gene_ensembl")
> head(listFilters(ensembl), 3)             ## filters
```

[4] http://www.sph.emory.edu/~hwu/teaching/bioc/bios560R.html

```
> myFilter <- "chromosome_name"
> head(filterOptions(myFilter, ensembl), 3) ## return values
> myValues <- c("21", "22")
> head(listAttributes(ensembl), 3)          ## attributes
> myAttributes <- c("ensembl_gene_id","chromosome_name")
> ## assemble and query the mart
> res <- getBM(attributes =  myAttributes, filters =  myFilter,
+              values =  myValues, mart = ensembl)
```

Use `head(res)` to see the results.

Exercise 8
Load the TxDb.Hsapiens.UCSC.hg19.knownGene annotation package, and read in the `chr22.vcf.gz` *example file from the VariantAnnotation package.*

Remembering to re-name sequence levels, use the `locateVariants` *function to identify coding variants.*

Summarize aspects of your data, e.g., did any coding variants match more than one gene? How many coding variants are there per gene ID?

Solution: Here we open the known genes data base, and read in the VCF file.

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> fl <- system.file("extdata", "chr22.vcf.gz", package="VariantAnnotation")
> vcf <- readVcf(fl, "hg19")
> seqlevels(vcf, force=TRUE) <- c("22"="chr22")
```

The next lines locate coding variants.

```
> rd <- rowData(vcf)
> loc <- locateVariants(rd, txdb, CodingVariants())
> head(loc, 3)

GRanges with 3 ranges and 7 metadata columns:
      seqnames               ranges strand | LOCATION   QUERYID      TXID
         <Rle>            <IRanges>  <Rle> | <factor> <integer> <integer>
  [1]    chr22 [50301422, 50301422]      * |   coding        24     73482
  [2]    chr22 [50301476, 50301476]      * |   coding        25     73482
  [3]    chr22 [50301488, 50301488]      * |   coding        26     73482
          CDSID      GENEID   PRECEDEID    FOLLOWID
      <integer> <character> <character> <character>
  [1]    217009       79087        <NA>        <NA>
  [2]    217009       79087        <NA>        <NA>
  [3]    217009       79087        <NA>        <NA>
  ---
  seqlengths:
   chr22
      NA
```

To answer gene-centric questions data can be summarized by gene regardless of transcript.

```
> ## Did any coding variants match more than one gene?
> splt <- split(loc$GENEID, loc$QUERYID)
> table(sapply(splt, function(x) length(unique(x)) > 1))
```

```
FALSE  TRUE
  956    15
```

```
> ## Summarize the number of coding variants by gene ID
> splt <- split(loc$QUERYID, loc$GENEID)
> head(sapply(splt, function(x) length(unique(x))), 3)
```

```
113730   1890  23209
    22     15     30
```

4.2 Visualization

4.2.1 Static

R has some great visualization packages; essential references include [3] for a general introduction, Murrell [7] for base graphics, Sarkar [9] for *lattice*, and Wickham [11] for *ggplot2*. Here we take a quick tour of visualization facilities tailed for sequence data and using *Bioconductor* approaches.

Gviz The *Gviz* package produces very elegant data organized in a more-or-less familiar 'track' format. The following exercises walk through the *Gviz* User guide Section 2. We have an opportunity to visit just one package, *Gviz*.

Load the *Gviz* package and sample *GRanges* containing genomic coordinates of CpG islands. Create a couple of variables with information on the chromosome and genome of the data (how can this information be extracted from the `cpgIslands` object?).

```
> library(Gviz)
> data(cpgIslands)
> chr <- "chr7"
> genome <- "hg19"
```

The basic idea is to create a track, perhaps with additional attributes, and to plot it. There are different types of track, and we create these one at a time. We start with a simple annotation track

```
> atrack <- AnnotationTrack(cpgIslands, name="CpG")
> plotTracks(atrack)
```

Then add a track that represents genomic coordinates. Tracks are combined during when plotted, as a simple list. The vertical ordering of tracks is determined by their position in the list.

```
> gtrack <- GenomeAxisTrack()
> plotTracks(list(gtrack, atrack))
```

We can add an ideogram to provide overall orientation...

```
> itrack <- IdeogramTrack(genome=genome, chromosome=chr)
> plotTracks(list(itrack, gtrack, atrack))
```

and a more elaborate gene model, as an *data.frame* or *GRanges* object with specific columns of metadata.

```
> data(geneModels)
> grtrack <-
+     GeneRegionTrack(geneModels, genome=genome,
+                     chromosome=chr, name="Gene Model")
> tracks <- list(itrack, gtrack, atrack, grtrack)
> plotTracks(tracks)
```

Zooming out changes the location box on the ideogram

```
> plotTracks(tracks, from=2.5e7, to=2.8e7)
```

When zoomed in we can add sequence data

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> strack <- SequenceTrack(Hsapiens, chromosome=chr)
> plotTracks(c(tracks, strack), from=26450430, to=26450490, cex=.8)
```

As the *Gviz* vignette humbly says, 'so far we have replicated the features of a whole bunch of other genome browser tools out there'. We'd like to be able integrate our data into these plots, with a rich range of plotting options. The key is the `DataTrack` function, which we demonstrate with some simulated data; this final result is shown in Figure 4.2.

```
> ## some data
> lim <- c(26700000, 26900000)
> coords <- seq(lim[1], lim[2], 101)
> dat <- runif(length(coords) - 1, min=-10, max=10)
> ## DataTrack
> dtrack <-
+     DataTrack(data=dat, start=coords[-length(coords)],
+               end= coords[-1], chromosome=chr, genome=genome,
+               name="Uniform Random")
> plotTracks(c(tracks, dtrack))
```

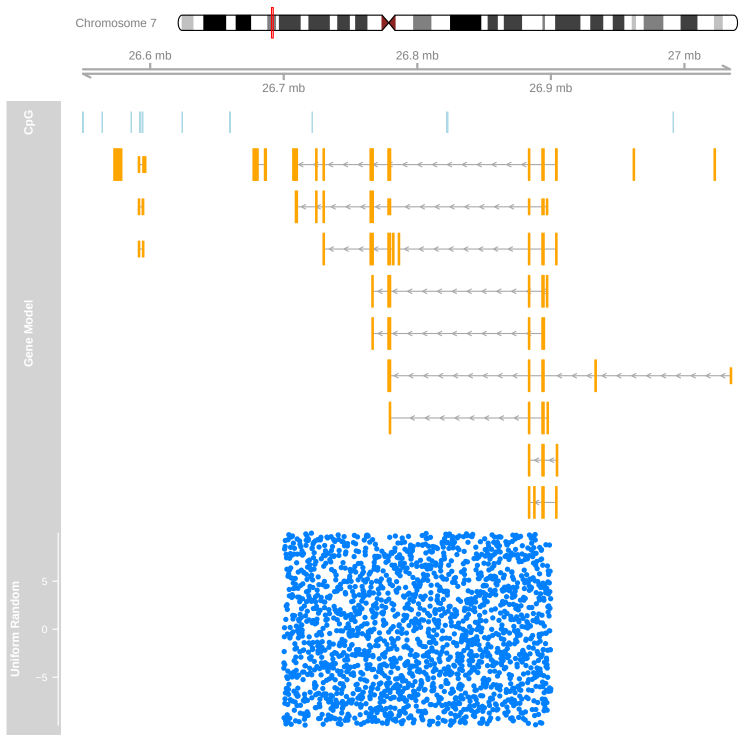
Section 4.3 of the *Gviz* vignette illustrates flexibility of the data track.

4.2.2 Interactive

The recently-introduced *shiny* package and web site[5] offers a new model for developing interactive, browser-based visualizations. These visualizations could be an excellent way to provide sophisticated exploratory or summary analysis in a very accessible way. The idea is to write a 'user interface' component that describes how a page is to be presented to users, and a 'server' that describes how the data are to be calculated or modified in responses to user choices. The programming model is 'reactive', where changes in a user choice automatically trigger re-calculations in the server. This reactive model is like in a spreadsheet with a formula, where adjusting a cell that the formula references triggers re-calculation of the formula. Just like in a spreadsheet, someone creating a *shiny* application does not have to work hard to make reactivity work. *ReportingTools* and other *Bioconductor* packages are starting to incorporate these interactive displays; a Google Summer of Code intern is working to develop *shiny*-based displays to help navigate some of the more complicated *Bioconductor* data structures.

[5]http://www.rstudio.com/shiny/

Figure 4.2: *Gviz* ideogram, genome coordinate, annotation, and data tracks.



Chapter 5

Working with Large Data

5.1 Four strategies

Bioinformatics data is now very large; it is not reasonable to expect all of a fastq or bam file, for instance, to fit into memory. How is this data to be processed? This challenge confronts us in whatever language or tool we are using. In *R* and *Bioconductor*, three main approaches are to: (1) *restrict* data input to the interesting subset of the larger data set; (2) *sample* from the large data, knowing that an appropriately sized sample will accurately estimate statistics we are interested in; (3) *iterate* through large data in chunks; and (4) use *parallel* evaluation on one or several computers. Let's look at each of these approaches.

5.1.1 Restriction

Just because a data file contains a lot of data does not mean that we are interested in all of it. In base *R*, one might use the `colClasses` argument to `read.delim` or similar function (e.g., setting some elements to `NULL`) to read only some columns of a large comma-separated value file. In addition to the obvious benefit of using less memory than if all of the file had been read in, input will be substantially faster because less computation needs to be done to coerce values from their representation in the file to their representation in *R*'s memory.

A variation on the idea of restricting data input is to organize the data on disk into a representation that facilitates restriction. In base *R*, large data might be stored in a relational data base like the *sqlite* data bases that are built in to *R* and used in the *AnnotationDbi* *Bioconductor* packages. In addition to facilitating restriction, these approaches are typically faster than parsing a plain text file, because the data base software has stored data in a way that efficiently transforms from on-disk to in-memory representation.

Let's use BAM files and the *Rsamtools* package to illustrate restriction. Rather than using simple text files ("sam" format), we use "bam" (binary alignment) files that have been indexed. Use of a binary format enhances data input speed, while the index facilitates restrictions that take into account genomic coordinates. The basic approach will use the `ScanBamParam` function to specify a restriction.

A BAM record can contain a lot of information, including the relatively large sequence, quality, and query name strings. Not all information in the BAM file is needed for some calculations. For instance, one could calculate coverage (number of nucleotides overlapping each reference position) using only the `rname` (reference sequence name), `pos`, and `cigar` fields. We could arrange to input this just information with

```
> param <- ScanBamParam(what=c("rname", "pos", "cigar"))
```

Another common restriction is to particular genomic regions, for instance known genes in an RNAseq differential expression study or to promoter regions in a ChIP-seq study. Restrictions in genome space are specified using `GRanges` objects (typically computed from some reference source, rather than entered by hand) provided as the `which` argument to `ScanBamParam`. Here we create a `GRanges` instance representing all exons on *H. sapiens* chromosome 14, and use that as a restriction to `ScanBamParam`'s `which` argument:

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> exByGn <- exonsBy(TxDb.Hsapiens.UCSC.hg19.knownGene, "gene")
> seqlevels(exByGn, force=TRUE) <- "chr14"
> gns <- unlist(range(exByGn))
> param <- ScanBamParam(which=gns)
```

The `what` and `which` restrictions can be combined with other restrictions into a single `ScanBamParam` object

```
> param <- ScanBamParam(what=c("rname", "pos", "cigar"), which=gns)
```

We can also restrict input to, e.g., paired-end reads that represent the primary (best) alignment; see the help page `?ScanBamParam` for a more complete description.

Here are some BAM files from an experiment in *H. sapiens*; the BAM files contain a subset of aligned reads

```
> library(RNAseqData.HeLa.bam.chr14)
> bamfls <- RNAseqData.HeLa.bam.chr14_BAMFILES
```

We'll read in the first BAM file, but restrict input to "cigar" to find how many reads map to the plus and minus strands. We use the "low-level" function `scanBam` to input the data

```
> param <- ScanBamParam(what="cigar")
> bam <- scanBam(bamfls[1], param=param)[[1]]
> tail(sort(table(bam$cigar)))

18M123N54M 36M123N36M  64M316N8M 38M670N34M 35M123N37M        72M
       225        228        261        264        272     603939
```

Restriction is such a useful concept that many *Bioconductor* high throughput sequence analysis functions enable doing the right thing. Functions such as `coverage,BamFile-method` use restrictions to read in the specific data required for them to compute the statistic of interest. Most "higher level" functions have a `param=ScanBamParam()` argument. For instance, the primary user-friendly function for reading BAM files is `readGappedAlignments` (this is renammed to `readGAlignments` in *Bioconductor* version 2.13) reads the most useful information, allowing the user to specify additional fields if desired.

```
> gal <- readGappedAlignments(bamfls[1])
> head(gal, 3)

GappedAlignments with 3 alignments and 0 metadata columns:
      seqnames strand       cigar    qwidth     start       end     width
         <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
  [1]    chr14      +         72M        72  19069583  19069654        72
  [2]    chr14      +         72M        72  19363738  19363809        72
  [3]    chr14      -         72M        72  19363755  19363826        72
           ngap
      <integer>
  [1]         0
  [2]         0
  [3]         0
  ---
  seqlengths:
                  chr1                 chr10 ...                  chrY
             249250621             135534747 ...              59373566

> param <- ScanBamParam(what="seq")  ## also input sequence
> gal <- readGappedAlignments(bamfls[1], param=param)
> head(mcols(gal)$seq)
```

```
  A DNAStringSet instance of length 6
    width seq
[1]    72 TGAGAATGATGATTTCCAATTTCATCCATGTCCC...AGGACATGAACTCATCATTTTTTATGGCTGCAT
[2]    72 CCCATATGTACATCAGGCCCCAGGTATACACTGG...AGGTGGACACCAGCACTCAGTTGGATACACACA
[3]    72 CCCCAGGTATACACTGGACTCCAGGTGGACACCA...CAGTTGGATACACACACTCAAGGTGGACACCAG
[4]    72 CATAGATGCAAGAATCCTCAATCAAATACTAGCA...AATTCAACAGCACATTAAAAAGATAACTTACCA
[5]    72 TAGCACACTGAATTCAACAGCACATTAAAAAGAT...ACCATGCTCAAGTGGATTTACCCCAAGGATACA
[6]    72 TGCTGGTGCAGGATTTATTCTACTAAGCAATGAG...GGATCAAATCCACTTTCTTATCTCAGGAATCAG
```

5.1.2 Sampling

R is after all a statistical language, and it sometimes makes sense to draw inferences from a sample of large data. For instance, many quality assessment statistics summarize overall properties (e.g., GC content or per-nucleotide base quality of FASTQ reads) that don't require processing of the entire data. For these statistics to be valid, the sample from the file needs to be a random sample, rather than a sample of convenience.

There are two advantages to sampling from a FASTQ (or BAM) file. The sample uses less memory than the full data. And because less data needs to be parsed from the on-disk to in-memory representation the input is faster.

The *ShortRead* package `FastqSampler` (see also `Rsamtools::BamSampler` visits a FASTQ file but retains only a random sample from the file. Here is our function to calculate GC content from a `DNAStringSet`:

```
> gcFunction <-
+     function(x)
+ {
+     alf <- alphabetFrequency(x, as.prob=TRUE)
+     rowSums(alf[,c("G", "C")])
+ }
```

and a subset of a FASTQ file with 1 million reads:

```
> bigdata <- system.file("bigdata", package="SequenceAnalysisData")
> fqfl <- dir(file.path(bigdata, "fastq"), ".fastq$", full=TRUE) #$
```

`FastqSampler` works by specifying the file name and desired sample size, e.g., 100,000 reads. This creates an object from which an independent sample can be drawn using the `yield` function.

```
> sampler <- FastqSampler(fqfl, 100000)
> fq <- yield(sampler)                    # 100,000 reads
> lattice::densityplot(gcFunction(sread(fq)), plot.points=FALSE)
> fq <- yield(sampler)                    # a different 100,000 reads
```

Generally, one would choose a sample size large enough to adequately characterize the data but not so large as to consume all (or a fraction, see section 5.1.4 below) of the memory. The default (1 million) is a reasonable starting point.

`FastqSampler` relies on the *R* random number generator, so the same sequence of reads can be sampled by using the same random number seed. This is a convenient way to sample the same read pairs from the FASTQ files typically used to represent paired-end reads

```
> ## NOT RUN
> set.seed(123)
> end1 <- yield(FastqSampler("end_1.fastq"))
> set.seed(123)
> end2 <- yield(FastqSampler("end_2.fastq"))
```

5.1.3 Iteration

Restriction may not be enough to wrestle large data down to size, and sampling may be inappropriate for the task at hand. A solution is then to iterate through the file. An example in base *R* is to open a file connection, and then read and process successive chunks of the file, e.g., reading chunks of 10000 lines

```
> ## NOT RUN
> con <- file("<hypothetical-file>.txt")
> open(f)
> while (length(x <- readLines(f, n=10000))) {
+     ## work on character vector 'x'
+ }
> close(f)
```

The pattern `length(x <- readLines(f, n=10000))` is a convenient short-hand pattern that reads the *next* 10000 lines into a variable `x` and then asks `x` it's length. `x` is created in the calling environment (so it is available for processing in the `while` loop) When there are no lines left to read, `length(x)` will evaluate to zero and the `while` loop will end.

Iteration really involves three steps: input of a 'chunk' of the data; calculation of a desired summary; and aggregation of summaries across chunks. We illustrate this with BAM files in the *Rsamtools* package; see also `FastqStreamer` in the *ShortRead* package and `TabixFile` for iterating through large VCF files (via `readVCF` in the *VariantAnnotation* package).

The first stage in iteration is to arrange for input of a chunk of data. In many programming languages one would iterate 'record-at-a-time', reading in one record, processing it, and moving to the next. *R* is not efficient when used in this fashion. Instead, we want to read in a larger chunk of data, typically as much as can comfortably fit in the available memory. In *Rsamtools* for reading BAM files we arrange for this by creating a `BamFile` (or `TabixFile` for VCF) object where we specify an appropriate `yieldSize`. Here we go for a bigger BAM file

```
> library(RNAseqData.HeLa.bam.chr14)
> bamfl <- RNAseqData.HeLa.bam.chr14_BAMFILES[1]
> countBam(bamfl)

  space start end width                 file records nucleotides
1    NA    NA  NA    NA ERR127306_chr14.bam  800484    57634848

> bf <- BamFile(bamfl, yieldSize=200000) # could be larger, e.g., 2 million
```

A `BamFile` object can be used to read data from a BAM file, e.g., using `readGappedAlignments`. Instead of reading all the data, the reading function will read just `yieldSize` records. The idea then is to open the BAM file and iterate through until no records are input. The pattern is

```
> ## initialize, e.g., for step 3 ...
> open(bf)
> while (length(gal <- readGappedAlignments(bf))) {
+     ## step 2: do work...
+     ## step 3: aggregate results...
+ }
> close(bf)
```

The second step is to perform a useful calculation on the chunk of data. This is particularly easy to do if chunks are independent of one another. For instance, a common operation is to count the number of times reads overlap regions of interest. There are functions that implement a variety of counting modes (see, e.g., `summarizeOverlaps` in the *GenomicRanges* package); here we'll go for a simple counter that arranges to tally one 'hit' each time a read overlaps (in any way) at most one range. Here is our counter function, taking as arguments a `GRanges` or `GRangesList` object representing the regions for which counts are desired, and a `GappedAlignments` object representing reads to be counted:

```
> counter <-
+     function(query, subject, ..., ignore.strand=TRUE)
+     ## query: GRanges or GRangesList
+     ## subject: GappedAlignments
+ {
+     if (ignore.strand)
+         strand(subject) <- "*"
+     hits <- countOverlaps(subject, query)
+     countOverlaps(query, subject[hits==1])
+ }
```

To set this step up a little more completely, we need to know the regions over which counting is to occur. Here we retrieve exons grouped by gene for the genome to which the BAM files were aligned:

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> query <- exonsBy(TxDb.Hsapiens.UCSC.hg19.knownGene, "gene")
```

Thus we'll add the following lines to our loop:

```
> ## initialize, e.g., for step 3 ...
> open(bf)
> while (length(gal <- readGappedAlignments(bf))) {
+     ## step 2: do work...
+     count0 <- counter(query, gal, ignore.strand=TRUE)
+     ## step 3: aggregate results...
+ }
> close(bf)
```

It is worth asking whether the 'do work' step will always be as straight-forward. The current example is easy because counting overlaps of one read does not depend on other reads, so the chunks can be processed independently of one another. *FIXME: These are paired-end reads, so counting is* not *this easy! See* `GenomicRanges::summarizeOverlaps`

The final step in iteration is to aggregate results across chunks. In our present case, `counter` always returns an integer vector with `length(query)` elements. We want to add these over chunks, and we can arrange to do this by starting with an initial vector of counts `counts <- integer(length(query))` and simply adding `count0` at each iteration. The complete iteration, packaged as a function to facilitate re-use is

```
> counter1 <-
+     function(bf, query, ...)
+ {
+     ## initialize, e.g., for step 3 ...
+     counts <- integer(length(query))
+     open(bf)
+     while (length(gal <- readGappedAlignments(bf, ...))) {
+         ## step 2: do work...
+         count0 <- counter(query, gal, ignore.strand=TRUE)
+         ## step 3: aggregate results...
+         counts <- counts + count0
+     }
+     close(bf)
+     counts
+ }
```

In action, this is invoked as

```
> bf <- BamFile(bamfl, yieldSize=500000)
> counts <- counter1(bf, query)
```

Counting is a particularly simple operation; one will often need to think carefully about how to aggregate statistics derived from individual chunks. A useful general approach is to identify *sufficient statistics* that represent a sufficient description of the data and can be easily aggregated across chunks. This is the approach taken by, for instance, the *biglm* package for fitting linear models to large data sets – the linear model is fit to successive chunks and the fit reduced to sufficient statistics, the sufficient statistics are then added across chunks to arrive at an overall fit.

5.1.4 Parallel evaluation

Each of the preceding sections addressed memory management; what about overall performance?

The starting point is really writing efficient, vectorized code, with common strategies outlined in Chapter 1. Performance differences between poorly written versus well written *R* code can easily span two orders of magnitude, whereas parallel processing can only increase throughput by an amount inversely proportional to the number of processing units (e.g., CPUs) available.

The memory management techniques outlined earlier in this chapter are important in a parallel evaluation context. This is because we will typically be trying to exploit multiple processing cores on a single computer, and the cores will be competing for the same pool of shared memory. We thus want to arrange for the collection of processors to cooperate in dividing available memory between them, i.e., each processor needs to use only a fraction of total memory.

There are a number of ways in which *R* code can be made to run in parallel. The least painful and most effective will use 'multicore' functionality provided by the *parallel* package; the *parallel* package is installed by default with base *R*, and has a useful vignette [8]. Unfortunately, multicore facilities are not available on Windows computers.

Parallel evaluation on several cores of a single Linux or MacOS computer is particularly easy to achieve when the code is already vectorized. The solution on these operating systems is to use the functions `mclapply` or `pvec`. These functions allow the 'master' process to 'fork' processes for parallel evaluation on each of the cores of a single machine. The forked processes initially share memory with the master process, and only make copies when the forked process modifies a memory location ('copy on change' semantics). On Linux and MacOS, the `mclapply` function is meant to be a 'drop-in' replacement for `lapply`, but with iterations being evaluated on different cores. The following illustrates the use of `mclapply` and `pvec`, for a toy vectorized function `f`:

```
> library(parallel)
> f <- function(i) {
+     cat("'f' called, length(i) = ", length(i), "\n")
+     sqrt(i)
+ }
> res0 <- mclapply(1:5, f, mc.cores=2)

'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1

> res1 <- pvec(1:5, f, mc.cores=2)

'f' called, length(i) =  3
'f' called, length(i) =  2
```

```
> identical(unlist(res0), res1)
```

```
[1] TRUE
```

`pvec` takes a vectorized function and distributes computation of different chunks of the vector across cores. Both functions allow the user to specify the number of cores used, and how the data are divided into chunks.

The *parallel* package does not support fork-like behavior on Windows, where users need to more explicitly create a cluster of *R* workers and arrange for each to have the same data loaded into memory; similarly, parallel evaluation across computers (e.g., in a cluster) require more elaborate efforts to coordinate workers; this is typically done using `lapply`-like functions provided by the *parallel* package but specialized for simple ('snow') or more robust ('MPI') communication protocols between workers.

Data movement and random numbers are two important additional considerations in parallel evaluation. Moving data to and from cores to the manager can be expensive, so strategies that minimize explicit movement (e.g., passing file names data base queries rather than *R* objects read from files; reducing data on the worker before transmitting results to the manager) can be important. Random numbers need to be synchronized across cores to avoid generating the same sequences on each 'independent' computation.

How might parallel evaluation be exploited in *Bioconductor* work flows? One approach when working with BAM files exploits the fact that data are often organized with one sample per BAM file. Suppose we are interested in running our iterating counter `counter1` over several BAM files. We could do this by creating a `BamFileList` with appropriate `yieldSize`

```
> fls <- RNAseqData.HeLa.bam.chr14_BAMFILES # 8 BAM files
> bamfls <- BamFileList(fls, yieldSize=500000) # yieldSize can be larger
```

and using an `lapply` to take each file in turn and performing the count.

```
> counts <- simplify2array(lapply(bamfls, counter1, query))
```

The parallel equivalent of this is simply (note the change from `lapply` to `mclapply`)

```
> options(mc.cores=detectCores())         # use all cores
> counts <- simplify2array(mclapply(bamfls, counter1, query))
> head(counts[rowSums(counts) != 0,], 3)
```

```
          ERR127306 ERR127307 ERR127308 ERR127309 ERR127302 ERR127303 ERR127304
10001           207       277       217       249       303       335       362
100128927       572       629       597       483       379       319       388
100129075        62        70        56        63        34        51        88
          ERR127305
10001           300
100128927       435
100129075        79
```

5.2 Other Large-Data Resource

The *foreach* package can be useful for parallel evaluation written using coding styles more like `for` loops rather than `lapply`. The *iterator* package is an abstraction that simplifies the notion of iterating over objects. The *Rmpi* package provides access to MPI, a structured environment for calculation on clusters. The *pbdR* formalism[1] is especially useful for well-structured distributed matrix computations.

The *MatrixEQTL* package is an amazing example implementing high performance algorithms on large data; the corresponding publication [10] is well worth studying.

Bioconductor provides an Amazon machine instance with MPI and *Rmpi* installed[2]. This can be an effective way to gain access to large computing resources.

[1] https://rdav.nics.tennessee.edu/2012/09/pbdr/
[2] http://bioconductor.org/help/bioconductor-cloud-ami/

Chapter 6

Supplement

6.1 Exercises: *GRanges* in action

Exercise 9
Load the GenomicRanges package. Open the man page for the GRanges class and run the examples in it.
Shift the ranges in `gr` by 100 positions to the right.
What method is called when doing `shift()` on a GRanges object? Find the man page for this method.

Solution: The examples in `?GRanges` can be run with:

```
> library(GenomicRanges)
> example(GRanges)

> shift(gr, 100)

GRanges with 10 ranges and 2 metadata columns:
    seqnames     ranges strand |     score                GC
       <Rle>  <IRanges>  <Rle> | <integer>         <numeric>
  a     chr1 [101, 110]      - |         1                 1
  b     chr2 [102, 110]      + |         2 0.888888888888889
  c     chr2 [103, 110]      + |         3 0.777777777777778
  d     chr2 [104, 110]      * |         4 0.666666666666667
  e     chr1 [105, 110]      * |         5 0.555555555555556
  f     chr1 [106, 110]      + |         6 0.444444444444444
  g     chr3 [107, 110]      + |         7 0.333333333333333
  h     chr3 [108, 110]      + |         8 0.222222222222222
  i     chr3 [109, 110]      - |         9 0.111111111111111
  j     chr3 [110, 110]      - |        10                 0
  ---
  seqlengths:
   chr1 chr2 chr3
   1000 2000 1500

> selectMethod("shift", "GRanges")

Method Definition:

function (x, shift = 0L, use.names = TRUE)
{
    ranges <- shift(ranges(x), shift, use.names = use.names)
    clone(x, ranges = ranges)
}
<environment: namespace:GenomicRanges>

Signatures:
        x
target  "GRanges"
defined "GenomicRanges"
```

The method for *GenomicRanges* objects is called. Consult its man page with `¿shift,GenomicRanges-method‘`.

Exercise 10
Load the RNAseqData.HeLa.bam.chr14 package and use the `readGappedAlignments` *function from the GenomicRanges package to load the reads from the 1st sequencing run.*

The object returned by `readGappedAlignments` *is a GappedAlignments object. This is a vector-like object where each element represents an alignment. In an RNA-seq experiment an alignment can contain one or more "gaps". Each gap corresponds to an intron and is represented by an N operation in the CIGAR. If we ignore the gaps, each alignment corresponds to a single genomic range. The ranges of all the alignments can be extracted with the* `granges` *function from the GenomicRanges package. Use* `granges` *to extract the genomic ranges of the alignments.*

Extract the genome-wide coverage of these genomic ranges. What kind of object is used to represent this coverage? This RleList object is a list-like object that can be manipulated like an ordinary list. Extract the list element corresponding to chromosome 14. What are the average and maximum coverage on this chromosome?

Solution:

```
> library(RNAseqData.HeLa.bam.chr14)
> bamfiles <- RNAseqData.HeLa.bam.chr14_BAMFILES
> gal <- readGappedAlignments(bamfiles[1])
```

```
> read_ranges <- granges(gal)
> read_ranges
```

```
GRanges with 800484 ranges and 0 metadata columns:
           seqnames                 ranges strand
              <Rle>              <IRanges>  <Rle>
       [1]    chr14   [19069583, 19069654]      +
       [2]    chr14   [19363738, 19363809]      +
       [3]    chr14   [19363755, 19363826]      -
       [4]    chr14   [19369799, 19369870]      +
       [5]    chr14   [19369828, 19369899]      -
       ...      ...                    ...    ...
  [800480]    chr14 [106989780, 106989851]      -
  [800481]    chr14 [106994763, 106994834]      +
  [800482]    chr14 [106994819, 106994890]      -
  [800483]    chr14 [107003080, 107003151]      +
  [800484]    chr14 [107003171, 107003242]      -
  ---
  seqlengths:
                   chr1                 chr10 ...                 chrY
              249250621             135534747 ...             59373566
```

```
> read_cvg <- coverage(read_ranges)
> read_cvg

RleList of length 93
$chr1
integer-Rle of length 249250621 with 1 run
  Lengths: 249250621
  Values :         0

$chr10
integer-Rle of length 135534747 with 1 run
  Lengths: 135534747
  Values :         0

$chr11
integer-Rle of length 135006516 with 1 run
  Lengths: 135006516
  Values :         0

$chr11_gl000202_random
integer-Rle of length 40103 with 1 run
  Lengths: 40103
  Values :     0

$chr12
integer-Rle of length 133851895 with 1 run
  Lengths: 133851895
  Values :         0

...
<88 more elements>

> read_cvg$chr14

integer-Rle of length 107349540 with 487597 runs
  Lengths: 19069582       72   294083       17 ...       19       72   346298
  Values :        0        1        0        1 ...        0        1        0

> mean(read_cvg$chr14)

[1] 6.676184

> max(read_cvg$chr14)

[1] 3900
```

Exercise 11
In the GRanges object obtained previously, each element can "hit" 0, 1 or more genes. In this exercise, we want to count the number of hits per gene. For this we first need to load a gene model based on the same reference genome that was used to align our reads.

Consult the documentation of the RNAseqData.HeLa.bam.chr14 package and find out what reference genome was used to align the reads.

The TxDb. packages in Bioconductor are a collection of annotation packages that contain various gene models for the most common organisms. Browse the list of annotation packages on the Bioconductor website and find the TxDb.* packages that are based on the same reference genome that was used to align our reads.*

Load the TxDb.Hsapiens.UCSC.hg19.knownGene package then use the `transcriptsBy` function from the GenomicFeatures package to extract the transcripts grouped by gene.

The returned object is an GRangesList object with one list element per gene. Each list element is itself a GRanges object with one element per transcript. Is this object named? What do those names mean.

Use `$` or `[[` to extract the transcripts of gene with Entrez ID 3183 (HNRNPC gene). How many transcripts in that gene?

In Bioconductor, `elementLengths` can be used on a list-like object to perform `sapply(x, length)` in a very efficient way. Use it (together with `table`) to tabulate the number of transcripts per gene.

`range` can be used on a GRangesList object to compute the range of each list element. Call it on our object containing the transcripts grouped by gene. Let's call this object `gene_ranges`. What kind of object is it? What's the range of the HNRNPC gene? Do all the list elements in `gene_ranges` have a length of 1? What can be the reasons for this?

Remove the list elements from `gene_ranges` that have a length > 1 and unlist it. What was the point of this removal?

Use `countOverlaps` on `gene_ranges` and the the GRanges object containing the genomic ranges of the alignments. Note that the RNA-seq protocol is generally not stranded so we want to ignore the strand when counting the overlaps.

Finally stick the result to `gene_ranges` as a metadata column.

Solution: The reads in *RNAseqData.HeLa.bam.chr14* were aligned to the hg19 genome. This information can be found in the description file of the package (with `packageDescription("RNAseqData.HeLa.bam.chr14")`) or in its man page (with `¿RNAseqData.HeLa.bam.chr14-package‘`).

To find the *TxDb.** packages that are based on hg19, go to http://www.bioconductor.org/, then click on *Install*, then on *Annotation Data*, then use the built-in document search of your browser (CTRL+f in Firefox and Chromium on Linux) to search for the TxDb pattern. There are currently 16 *TxDb.** packages in *Bioconductor*. 2 of them are based on the hg19 genome from UCSC.

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
> tx_by_gene <- transcriptsBy(txdb, by="gene")
> tx_by_gene

GRangesList of length 22932:
$1
GRanges with 2 ranges and 2 metadata columns:
      seqnames               ranges strand |     tx_id     tx_name
         <Rle>            <IRanges>  <Rle> | <integer> <character>
  [1]    chr19 [58858172, 58864865]      - |     68796  uc002qsd.4
  [2]    chr19 [58859832, 58874214]      - |     68797  uc002qsf.2

$10
GRanges with 1 range and 2 metadata columns:
      seqnames               ranges strand | tx_id    tx_name
  [1]     chr8 [18248755, 18258723]      + | 31203 uc003wyw.1

$100
GRanges with 1 range and 2 metadata columns:
      seqnames               ranges strand | tx_id    tx_name
  [1]    chr20 [43248163, 43280376]      - | 70442 uc002xmj.3
```

```
...
<22929 more elements>
---
seqlengths:
                  chr1                  chr2 ...        chrUn_gl000249
             249250621             243199373 ...                 38502
```

According to the man page for `transcriptsBy`, `tx_by_gene` is named with the gene ids. In the case of `txdb`, the gene ids are Entrez Gene IDs:

```
> txdb

TranscriptDb object:
| Db type: TranscriptDb
| Supporting package: GenomicFeatures
| Data source: UCSC
| Genome: hg19
| Organism: Homo sapiens
| UCSC Table: knownGene
| Resource URL: http://genome.ucsc.edu/
| Type of Gene ID: Entrez Gene ID
| Full dataset: yes
| miRBase build ID: GRCh37
| transcript_nrow: 80922
| exon_nrow: 286852
| cds_nrow: 235842
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2013-03-08 09:43:09 -0800 (Fri, 08 Mar 2013)
| GenomicFeatures version at creation time: 1.11.14
| RSQLite version at creation time: 0.11.2
| DBSCHEMAVERSION: 1.0

> tx_by_gene[["3183"]]

GRanges with 11 ranges and 2 metadata columns:
       seqnames               ranges strand   |     tx_id     tx_name
          <Rle>            <IRanges>  <Rle>   | <integer> <character>
   [1]    chr14 [21677296, 21737638]      -   |     51103  uc001vzw.3
   [2]    chr14 [21677296, 21737638]      -   |     51104  uc001vzx.3
   [3]    chr14 [21677296, 21737638]      -   |     51105  uc001vzy.3
   [4]    chr14 [21677296, 21737638]      -   |     51106  uc001vzz.3
   [5]    chr14 [21677296, 21737638]      -   |     51107  uc001waa.3
   ...      ...                  ...    ... ...       ...         ...
   [7]    chr14 [21677296, 21737638]      -   |     51109  uc001wad.3
   [8]    chr14 [21677296, 21737638]      -   |     51110  uc010ail.3
   [9]    chr14 [21677296, 21737638]      -   |     51111  uc010tlq.2
  [10]    chr14 [21678927, 21698532]      -   |     51112  uc010tlr.2
  [11]    chr14 [21678927, 21737638]      -   |     51113  uc001wae.3
  ---
  seqlengths:
                    chr1                  chr2 ...        chrUn_gl000249
               249250621             243199373 ...                 38502
```

11 transcripts in that gene.

```
> table(elementLengths(tx_by_gene))

   1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16
8344 4369 3273 2365 1595 1051  632  417  207  152  111   87   52   41   30   31
  17   18   19   20   21   22   23   24   25   26   27   28   29   30   31   32
  24   20    6   14   13    3    7   12    6    4    3    9    2    5    1    3
  33   34   35   36   38   39   40   42   44   46   47   49   52   56   57   64
   4    2    5    2    2    1    4    7    1    1    1    1    1    1    1    1
  65   67   69   72  104  120  129  175
   1    1    1    1    1    1    1    1

> gene_ranges <- range(tx_by_gene)
> gene_ranges

GRangesList of length 22932:
$1
GRanges with 1 range and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]    chr19 [58858172, 58874214]      -

$10
GRanges with 1 range and 0 metadata columns:
      seqnames               ranges strand
  [1]     chr8 [18248755, 18258723]      +

$100
GRanges with 1 range and 0 metadata columns:
      seqnames               ranges strand
  [1]    chr20 [43248163, 43280376]      -

...
<22929 more elements>
---
seqlengths:
                  chr1                  chr2 ...         chrUn_gl000249
             249250621             243199373 ...                  38502

> gene_ranges[["3183"]]

GRanges with 1 range and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]    chr14 [21677296, 21737638]      -
  ---
  seqlengths:
                    chr1                  chr2 ...         chrUn_gl000249
               249250621             243199373 ...                  38502

> table(elementLengths(gene_ranges))
```

```
    1     2     3     4     5     6     7     8 
22534   195     6     5    19    39    75    59 
```

```
> gene_ranges[elementLengths(gene_ranges) == 2]

GRangesList of length 195:
$100036519 
GRanges with 2 ranges and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]     chr9 [42717234, 42720342]      +
  [2]     chr9 [70426623, 70429731]      -

$100101116 
GRanges with 2 ranges and 0 metadata columns:
      seqnames             ranges strand
  [1]     chrY [6258442, 6279605]      +
  [2]     chrY [9590765, 9611928]      -

$100101120 
GRanges with 2 ranges and 0 metadata columns:
      seqnames             ranges strand
  [1]     chrY [6317509, 6325947]      +
  [2]     chrY [9544433, 9552871]      -


...
<192 more elements>
---
seqlengths:
                  chr1                  chr2 ...        chrUn_gl000249
             249250621             243199373 ...                 38502

> gene_ranges[elementLengths(gene_ranges) == 3]

GRangesList of length 6:
$100133331 
GRanges with 3 ranges and 0 metadata columns:
      seqnames                 ranges strand
         <Rle>              <IRanges>  <Rle>
  [1]     chr1 [   322037,    326938]      +
  [2]     chr1 [   661139,    679736]      -
  [3]     chr5 [180743354, 180753553]      +

$3119 
GRanges with 3 ranges and 0 metadata columns:
           seqnames               ranges strand
  [1]          chr6 [32627241, 32634466]      -
  [2] chr6_cox_hap2 [ 4072511,  4080111]      -
  [3] chr6_qbl_hap6 [ 3858965,  3866565]      -

$3833 
GRanges with 3 ranges and 0 metadata columns:
            seqnames               ranges strand
  [1]           chr6 [33359313, 33377699]      +
  [2]  chr6_qbl_hap6 [ 4591540,  4609700]      +
  [3] chr6_ssto_hap7 [ 4839529,  4857918]      +

...
<3 more elements>
---
seqlengths:
                  chr1                 chr2 ...        chrUn_gl000249
             249250621            243199373 ...                 38502
```

```
> gene_ranges <- gene_ranges[elementLengths(gene_ranges) == 1]
> gene_ranges <- unlist(gene_ranges)
```

`gene_ranges` is now a *GRanges* object with 1 range per gene. Not removing the list elements that have a length > 1 before unlisting would produce a *GRanges* object with possibly more than 1 range per gene.

```
> nhits_per_gene <- countOverlaps(gene_ranges, read_ranges, ignore.strand=TRUE)
```

```
> mcols(gene_ranges)$nhits <- nhits_per_gene
> gene_ranges
```

```
GRanges with 22534 ranges and 1 metadata column:
        seqnames                 ranges strand   |     nhits
           <Rle>              <IRanges>  <Rle>   | <integer>
      1    chr19 [ 58858172,  58874214]      -   |         0
     10     chr8 [ 18248755,  18258723]      +   |         0
    100    chr20 [ 43248163,  43280376]      -   |         0
   1000    chr18 [ 25530930,  25757445]      -   |         0
  10000     chr1 [243651535, 244006886]      -   |         0
    ...      ...                    ...    ... ...       ...
   9991     chr9 [114979995, 115095944]      -   |         0
   9992    chr21 [ 35736323,  35743440]      +   |         0
   9993    chr22 [ 19023795,  19109967]      -   |         0
   9994     chr6 [ 90539619,  90584155]      +   |         0
   9997    chr22 [ 50961997,  50964905]      -   |         0
  ---
  seqlengths:
                  chr1                 chr2 ...        chrUn_gl000249
             249250621            243199373 ...                 38502
```

References

[1] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*, pages 193–202, 2011.

[2] P. Burns. The R inferno. Technical report, 2011.

[3] W. Chang. *R Graphics Cookbook*. O'Reilly Media, Incorporated, 2012.

[4] P. Dalgaard. *Introductory Statistics with R*. Springer, 2nd edition, 2008.

[5] N. Matloff. *The Art of R Programming*. No Starch Pess, 2011.

[6] J. Meys and A. de Vries. *R For Dummies*. For Dummies, 2012.

[7] P. Murrell. *R graphics*. Chapman & Hall/CRC, 2005.

[8] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

[9] D. Sarkar. *Lattice: multivariate data visualization with R*. Springer, 2008.

[10] A. A. Shabalin. Matrix eqtl: ultra fast eqtl analysis via large matrix operations. *Bioinformatics*, 28(10):1353–1358, 2012.

[11] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer Publishing Company, Incorporated, 2009.