Basics on Analyzing Next Generation Sequencing Data with R and Bioconductor

... 

Thomas Girke

December 14, 2013
Overview

String Handling Utilities in R’s Base Distribution

Sequence Handling with Bioconductor

Range Operations
Outline

Overview

String Handling Utilities in R’s Base Distribution

Sequence Handling with Bioconductor

Range Operations
Biosequence Analysis in R and Bioconductor

R Base

- Some basic string handling utilities. Wide spectrum of numeric data analysis tools.

Bioconductor

- Bioconductor packages provide much more sophisticated string handling utilities for sequence analysis.
  - Biostrings: general sequence analysis environment
  - ShortRead: pipeline for short read data
  - IRanges: low-level infrastructure for range data
  - GenomicRanges: high-level infrastructure for range data
  - GenomicFeatures: managing transcript centric annotations
  - BSgenome: genome annotation data
  - biomaRt: interface to BioMart annotations
  - rtracklayer: Annotation imports, interface to online genome browsers

Interface for non-R sequence analysis tools

- e.g. short read aligners
Outline

Overview

String Handling Utilities in R’s Base Distribution

Sequence Handling with Bioconductor

Range Operations
Basic String Matching and Parsing

String matching.

```r
> myseq <- c("ATGCAGACATAGTG", "ATGAACATAGATCC", "GTACAGATCAC") # Sample sequence data set.
> myseq[grep("ATG", myseq)] # String searching with regular expression support.

[1] "ATGCAGACATAGTG" "ATGAACATAGATCC"

> pos1 <- regexpr("AT", myseq) # Searches 'myseq' for first match of pattern "AT".
> as.numeric(pos1); attributes(pos1)$match.length # Returns position information of matches.

[1] 1 1 7
[1] 2 2 2

> pos2 <- gregexpr("AT", myseq) # Searches 'myseq' for all matches of pattern "AT".
> as.numeric(pos2[[1]]); attributes(pos2[[1]])$match.length # Returns positions of matches in first sequence.

[1] 1 9
[1] 2 2

> gsub("^ATG", "atg", myseq) # String substitution with regular expression support.

[1] "atgCAGACATAGTG" "atgAACATAGATCC" "GTACAGATCAC"
```

Positional parsing.

```r
> nchar(myseq) # Computes length of strings.

[1] 14 14 11

> substring(myseq[1], c(1,3), c(2,5)) # Positional parsing of several fragments from one string.

[1] "AT" "GCA"

> substring(myseq, c(1,4,7), c(2,6,10)) # Positional parsing of many strings.

[1] "AT" "AAC" "ATCA"
```
Create any number of random DNA sequences of any length.

```r
> rand <- sapply(1:100, function(x) paste(sample(c("A","T","G","C"), sample(10:20), replace=T), collapse=""))
> rand[1:3]

[1] "TTTACTAGCGAGTCGACAAA" "CCAGATGCAAGAACCCAAT" "GTAGCTACGAGTAG"
```

Enumerate sequences to check for duplicates.

```r
> table(c(rand[1:4], rand[1]))

    AACTACTAACA CCAGATGCAAGAACCCAAT GTAGCTACGAGTAG TTTACTAGCGAGTCGACAAA

     1     1     1     2
```

Extract any number of pseudo reads from the following reference. Note: this requires Biostrings.

```r
> library(Biostrings)
> ref <- DNAString(paste(sample(c("A","T","G","C"), 100000, replace=T), collapse=""))
> randstart <- sample(1:(length(ref)-15), 1000)
> randreads <- Views(ref, randstart, width=15)
> rand_set <- DNAStringSet(randreads)
> unlist(rand_set)

15000-letter "DNAString" instance
```

```r
seq: CTGGTTTACAGGCAATGCTGGGATTCTCCAAAAATTGACCGGATCGGCCTCATGGCCACGGCTGATTAACGATTCAGCCAATTAACGCTATTATCCATCTAATTCTC...
```
Important Data Objects in Biostrings

**XString** for single sequence
- DNAString: for DNA
- RNAString: for RNA
- AAString: for amino acid
- BString: for any string

**XStringSet** for many sequences
- DNAStringSet: for DNA
- RNAStringSet: for RNA
- AAStringSet: for amino acid
- BStringSet: for any string

**QualityScaleXStringSet** for many sequences plus quality data
- QualityScaledDNAStringSet: for DNA
- QualityScaledRNAStringSet: for RNA
- QualityScaledAAStringSet: for amino acid
- QualityScaledBStringSet: for any string
Download the following sequences to your current working directory and then import them into R:

```r
> dir.create("data")
> myseq <- readDNAStringSet("data/AE004437.ffn")
> myseq[1:3]
```

A DNAStringSet instance of length 3

<table>
<thead>
<tr>
<th>width</th>
<th>seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>1206 ATGACTCGCGCGGTCTCGTGCCTGCGGTGCCGCGCCTCCAGCCATTGTACTGCCCCTGGCCGCAGTGGCTGCCGCTCCGATTGC...AGCG</td>
</tr>
<tr>
<td>[2]</td>
<td>666  ATGAGCATCATCGAAGAGGCGTGGTCAAACGGTACGAAACCGGTGCCGAGACAGTCGAGGCGCTGAAGCCGGCTGCTGACTTCTCGCCG...AACATCG</td>
</tr>
<tr>
<td>[3]</td>
<td>1110 ATGGCGTGGCGGAACCTCGGCGGGAACCGCGTGGCGACTGCGCTGCCGGATCGTGATCGGTGTATCGATCGCATCGATCGGGG...TTCC</td>
</tr>
</tbody>
</table>

```r
> sub <- myseq[grep("99.*", names(myseq))]
> length(sub)
```

```
[1] 170
```

```r
> writeXStringSet(sub, file="AE004437sub.ffn", width=80)
```

Open exported sequence file AE004437sub.ffn in a text editor.
The XString stores the different types of biosequences in dedicated containers:

```r
> library(Biostrings)
> d <- DNAString("GCATAT-TAC")
> d

10-letter "DNAString" instance
seq: GCATAT-TAC

> d[1:4]

4-letter "DNAString" instance
seq: GCAT

> r <- RNAString("GCAUAU-UAC")
> r <- RNAString(d) # Converts d into RNAString object.
> p <- AAString("HCWYHH")
> b <- BString("I store any set of characters. Other XString objects store only the IUPAC characters.")
```
Working with XStringSet Containers

XStringSet containers allow to store many biosequences in one object:

```r
> dset <- DNAStringSet(c("GCATATTAC", "AATCGATCC", "GCATATTAC"))
> names(dset) <- c("seq1", "seq2", "seq3") # Assigns names
> dset[1:2]

A DNAStringSet instance of length 2
  width seq
[1] 9 GCATATTAC
[2] 9 AATCGATCC

> width(dset) # Returns the length of each sequences
[1] 9 9 9

> d <- dset[[1]] # The [[ subsetting operator returns a single entry as XString object
> dset2 <- c(dset, dset) # Appends/concatenates two XStringSet objects
> dsetchar <- as.character(dset) # Converts XStringSet to named vector
> dsetone <- unlist(dset) # Collapses many sequences to a single one stored in a DNAString container

Sequence subsetting by positions:

> DNAStringSet(dset, start=c(1,2,3), end=c(4,8,5))

A DNAStringSet instance of length 3
  width seq
[1] 4 GCAT
[2] 7 ATCGATC
[3] 3 ATA
```
The XMultipleAlignment class stores the different types of multiple sequence alignments:

```r
> origMAlign <- readDNAMultipleAlignment(filepath = system.file("extdata", + "msx2_mRNA.aln", package = "Biostrings"), format = "clustal")
> origMAlign

DNAMultipleAlignment with 8 rows and 2343 columns

[1] -----TCCCGTCTCCGAGCACAAAAAATTTGAGTCGCCTGCGCGCCGTTGCCAGCGAAGTGGAGTCGCCGCCGAGCTACGTAGGGCAGAGAAGTCA-T...GAAGAGTTATCTCTTATTCTGAATT--AAATTAAGC--ATTTGTTTTATTGCAGTAAAGTTTGTCCAAACTCACAATTAAAAAAAAAAAAAAAAA gi|84452153|ref|N...
[2] ---------------------------------------------------------------------------------------------A-T...----------------------------------------------------------------------------------------------- gi|208431713|ref|...
[3] -----------------------------------------------------------------------------------GAGAGAAGTCA-T...----------------------------------------------------------------------------------------------- gi|118601823|ref|...
[4] ----------------------AAAAGTTGGAGTCTTCGCTTGAGAGTTGCCAGCGGAGTCGCGCGCCGACAGCTACGTAGGGCAGAGAAGTCA-T...GAAGAGTTATTTCTTATCTCTTACTCTGAATTAAATTAAAATATTTTATTGCAGT---------------------------------------- gi|114326503|ref|...
[5] ---------------------------------------------------------------------------------------------A-T...GAAGAGTTATTTCTTATCTCATACTCTGAATTAAATTAAAATGTTTTATTGCAGTAAA------------------------------------- gi|119220589|ref|...
[6] ---------------------------------------------------------------------------------------------A-T...----------------------------------------------------------------------------------------------- gi|148540149|ref|...
[7] --------------CGGCTCCGCAGCGCCTCACTCGCAGCTCCCGCGCGAGGCCCGAGGCCAGAGGCCACGCGCGCTACCGCTCCCCGCGGCCCCGCCCTC-C...----------------------------------------------------------------------------------------------- gi|45383056|ref|N...
[8] GGGGGAGACTTCAGAAGTTGTGTCTCTCCTCCGCTGATAAACAGTTGAGATGCAGCATATTATTATTACCTTTAGGACAAGTTGTAATGTGTTCGTCAGC...-------
```
Complement, reverse, and reverse & complement of sequences:

```r
> randset <- DNAStringSet(rand)
> complement(randset[1:2])

A DNAStringSet instance of length 2
  width seq
[1]  20 AAATGATCGCTCAGCTGTTT
[2]  19 GGTCTACGTTCTTTGGTTA

> reverse(randset[1:2])

A DNAStringSet instance of length 2
  width seq
[1]  20 AAACAGCTGAGCGATCATTT
[2]  19 TAACCCAAGAACGTAGACC

> reverseComplement(randset[1:2])

A DNAStringSet instance of length 2
  width seq
[1]  20 TTTGTCGACTCGCTAGTAAA
[2]  19 ATTGGGTTCTTGCATCTGG

Translate DNA sequences into proteins:

> translate(randset[1:2])

A AAStringSet instance of length 2
  width seq
[1]  6 FTSEST
[2]  6 PDARTQ
Pattern Matching

Pattern matching with mismatches

```r
> mypos <- matchPattern("ATGGTG", myseq1[[1]], max.mismatch=1) # Finds pattern matches in reference
> countPattern("ATGGCT", myseq1[[1]], max.mismatch=1) # Counts only the corresponding matches
> vcountPattern("ATGGCT", myseq1, max.mismatch=1) # Counts only the matches in many sequences
> tmp <- c(DNAStringSet("ATGGTG"), DNAStringSet(mypos)) # Results shown in DNAStringSet object
> consensusMatrix(tmp)[1:4,] # Returns a consensus matrix for query and hits.
> myvpos <- vmatchPattern("ATGGCT", myseq1, max.mismatch=1) # Finds all pattern matches in reference
> myvpos # The results are stored as MIndex object.
> Views(myseq1[[1]], start(myvpos[[1]]), end(myvpos[[1]])) # Retrieves the result for single entry
> sapply(seq(along=myseq1), function(x)
+ as.character(Views(myseq1[[x]], start(myvpos[[x]]), end(myvpos[[x]])))) # All matches.
```

Pattern matching with regular expression support

```r
> myseq <- DNAStringSet(c("ATGCAGACATAGTG", "ATGAACATAGATCC", "GTACAGATCAC"))
> myseq[grep("^ATG", myseq, perl=TRUE)] # String searching with regular expression support
> pos1 <- regexpr("AT", myseq) # Searches 'myseq' for first match of pattern "AT"
> as.numeric(pos1); attributes(pos1)$match.length # Returns position information of matches
> pos2 <- gregexpr("AT", myseq) # Searches 'myseq' for all matches of pattern "AT"
> as.numeric(pos2[[1]]); attributes(pos2[[1]])$match.length # Match positions in first sequence
> DNAStringSet(gsub("^ATG", "NNN", myseq)) # String substitution with regular expression support
```
> pwm <- PWM(DNAStringSet(c("GCT", "GGT", "GCA")))
> library(seqLogo); seqLogo(t(t(pwm) * 1/colSums(pwm)))

> chr <- DNAString("AAAGCTAAAGGTAAAGCAAAA")
> matchPWM(pwm, chr, min.score=0.9) # Searches sequence for PWM matches with score better than min.score.

Views on a 21-letter DNAString subject
subject: AAAGCTAAAGGTAAAGCAAAA
views:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
<th>view</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>3</td>
<td>[GCT]</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>3</td>
<td>[GGT]</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>3</td>
<td>[GCA]</td>
</tr>
</tbody>
</table>
Sequence and Quality Data: FASTQ Format

4 lines per sequence

1. ID
2. Sequence
3. ID
4. Base call qualities (Phred scores) as ASCII characters

Example of 3 Illumina reads in FASTQ format:

```plaintext
@SRR038845.3 HWI-EAS038:6:1:0:1938 length=36 CAACGAGTTTCACCTTGGCCGACAGGCCCGGGGTAA +SRR038845.3 HWI-EAS038:6:1:0:1938 length=36 BA@7>B=>:>>7@7@>>9=BAA?;>52;>:9=8.=A @SRR038845.41 HWI-EAS038:6:1:0:1474 length=36 CCAATGATTTTTTTCCGTGTTCAGAATACGGTTAA +SRR038845.41 HWI-EAS038:6:1:0:1474 length=36 BCCBA@BB@BBBBAB@B9B@=BABA@A:@693:0B= @SRR038845.53 HWI-EAS038:6:1:1:360 length=36 GTTCAAAAAGAAGTAAATTGTGTCAATAGAAAACCTC +SRR038845.53 HWI-EAS038:6:1:1:360 length=36 BBCBBBBBB@@BAB?BBBBBCB>BBBBAA8>BBBBAA@
```
Phred quality scores are integers from 0-50 that are stored as ASCII characters after adding 33. The basic R functions rawToChar and charToRaw can be used to interconvert among their representations.

```r
> phred <- 1:9
> phreda <- paste(sapply(as.raw((phred)+33), rawToChar), collapse=""); phreda

[1] ""#$%&'()*

> as.integer(charToRaw(phreda))-33

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

```r
dset <- DNAStringSet(sapply(1:100, function(x) paste(sample(c("A","T","G","C"), 20, replace=T), collapse=""))) # Creates random sample sequence.
myqlist <- lapply(1:100, function(x) sample(1:40, 20, replace=T)) # Creates random Phred score list.
myqual <- sapply(myqlist, function(x) toString(PhredQuality(x))) # Converts integer scores into ASCII characters.
myqual <- PhredQuality(myqual) # Converts to a PhredQuality object.
dsetq1 <- QualityScaledDNAStringSet(dset, myqual) # Combines DNAStringSet and quality data in QualityScaledDNAStringSet object.
dsetq1[1:2]
```

A QualityScaledDNAStringSet instance containing:

- A DNAStringSet instance of length 2
  - width seq
    - [1] 20 CTAATTATACTCTTCACGAT
    - [2] 20 TGAGAATTACATCGGCTCACT

- A PhredQuality instance of length 2
  - width seq
    - [1] 20 +,9FD6@02&8B'923'5(()
    - [2] 20 GI:B("?:*36&:$I.3G@
Processing FASTQ Files with ShortRead

Basic usage of ShortReadQ objects. To make the following sample code work, download and unzip this file to your current working directory.

```r
> library(ShortRead)
> fastq <- list.files("data", "*.fastq$"'); fastq <- paste("data/", fastq, sep="")
> names(fastq) <- paste("flowcell6_lane", 1:length(fastq), sep="_")
> (fq <- readFastq(fastq[1])) # Imports first FASTQ file

  class: ShortReadQ
  length: 1000 reads; width: 36 cycles

> countLines(dirPath="./data", pattern=".fastq$")/4 # Counts numbers of reads in FASTQ files

SRR038845.fastq  SRR038846.fastq  SRR038848.fastq  SRR038850.fastq
  1000  1000  1000  1000

> id(fq)[1] # Returns ID field

  A BStringSet instance of length 1
    width seq

> sread(fq)[1] # Returns sequence

  A DNAStringSet instance of length 1
    width seq
  [1]  36      CAACGAGTTCACACCTTGGCCGACAGGCCCGGGTAA

> quality(fq)[1] # Returns Phred scores

  class: FastqQuality
  quality:
    A BStringSet instance of length 1
      width seq
  [1]  36       BA@7>B=>:>>7@7@>>9=BAA?;>52;>;9=8.=A

> as(quality(fq), "matrix")[1,1:12] # Coerces Phred scores to numeric matrix

  [1] 33 32 31 22 29 33 28 29 25 29 29 22

> ShortReadQ(sread=sread(fq), quality=quality(fq), id=id(fq)) # Constructs a ShortReadQ from components

  class: ShortReadQ
  length: 1000 reads; width: 36 cycles
```
Quality Reports of FASTQ Files

The following seeFastq/seeFastqPlot functions generate and plot a series of useful quality statistics for a set of FASTQ files.

```r
> library(ggplot2)
> source("http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/My_R_Scripts/fastqQuality.R")
> fqlist <- seeFastq(fastq=fastq, batchsize=800, klength=8)  # For real data set batchsize to at least 10^5
> seeFastqPlot(fqlist[1:2], arrange=c(1,4,7))
```

Handles many samples in one PDF file. For more details see here [Link](http://faculty.ucr.edu/~tgirke/Documents/R_BioCond/My_R_Scripts/fastqQuality.R)
ShortRead contains various FASTQ quality report functions

```r
> sp <- SolexaPath(system.file('extdata', package='ShortRead'))
> fl <- file.path(analysisPath(sp), "s_1_sequence.txt")
> fls <- c(fl, fl)
> coll <- QCollate(QAFastqSource(fls), QAReadQuality(), QAAdapterContamination(),
+                  QANucleotideUse(), QAQualityUse(), QASequenceUse(), QAFrequentSequence(n=10),
+                  QANucleotideByCycle(), QAQualityByCycle())
> x <- qa2(coll, verbose=TRUE)
> res <- report(x)
> if(interactive())
+    browseURL(res)
```
Adaptor trimming

> fqtrim <- trimLRPatterns(Rpattern="GCCCGGGTAA", subject=fq)
> sread(fqtrim)[1:2]

A DNAStringSet instance of length 2
  width seq
[1] 26 CAACGAGTTCAACCTTGCGACAG
[2] 36 CCAATGATTTTTTTTGGTACGATACCGTAA

Read counting and duplicate removal

> tables(fq)$distribution # Counts read occurrences

  nOccurrences nReads
1        1     948
2        2     26

> sum(srduplicated(fq)) # Identifies duplicated reads

[1] 26

> fq[!srduplicated(fq)]

class: ShortReadQ
length: 974 reads; width: 36 cycles
Filtering and Trimming FASTQ Files with ShortRead II

Trimming low quality tails

```r
> cutoff <- 30
> cutoff <- rawToChar(as.raw(cutoff+33))
> sread(trimTails(fq, k=2, a=cutoff, successive=FALSE))[1:2]
```

A DNAStringSet instance of length 2

<table>
<thead>
<tr>
<th>width</th>
<th>seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>CAAC</td>
</tr>
<tr>
<td>20</td>
<td>CCAATGATTTTTTCCGTGT</td>
</tr>
</tbody>
</table>

Removal of reads with \( x \) Phred scores below a threshold value

```r
> cutoff <- 30
> qcount <- rowSums(as(quality(fq), "matrix") <= 20)
> fq[qcount == 0] # Number of reads where all Phred scores \( \geq 20 \)
```

class: ShortReadQ
length: 349 reads; width: 36 cycles

Removal of reads with \( x \) Ns and/or low complexity segments

```r
> filter1 <- nFilter(threshold=1) # Keeps only reads without Ns
> filter2 <- polynFilter(threshold=20, nuc=c("A","T","G","C")) # Removes reads with \( \geq 20 \) of one nucleotide
> filter <- compose(filter1, filter2)
> fq[filter(fq)]
```

class: ShortReadQ
length: 989 reads; width: 36 cycles
Memory Efficient FASTQ Processing

Streaming through FASTQ files with FastqStreamer and random sampling reads with FastqSampler

```r
> fq <- yield(FastqStreamer(fastq[1], 50)) # Imports first 50 reads
> fq <- yield(FastqSampler(fastq[1], 50)) # Random samples 50 reads
```

Streaming through a FASTQ file while applying filtering/trimming functions and writing the results to a new file.

```r
> f <- FastqStreamer(fastq[1], 50)
> while(length(fq <- yield(f))) {
+     fqsub <- fq[grepl("^TT", sread(fq))]
+     writeFastq(fqsub, paste(fastq[1], "sub", sep="_"), mode="a")
+ }
> close(f)
```
Exercise I

Task 1 Write a demultiplexing function that accepts any number of barcodes and splits a FASTQ file into as many subfiles as there are barcodes. At the same time the function should remove low quality tails from the reads. The following function accomplishes the first step. Expand this function so that it performs the second step as well.

Sample code:

```r
> demultiplex <- function(x, barcode, nreads) {
+   f <- FastqStreamer(x, nreads)
+   while(length(fq <- yield(f))) {
+     for(i in barcode) {
+       pattern <- paste("~", i, sep="")
+       fqsub <- fq[grepl(pattern, sread(fq))]
+       if(length(fqsub) > 0) {
+         writeFastq(fqsub, paste(x, i, sep="_"), mode="a")
+       }
+     }
+   }
+   close(f)
+ }
> demultiplex(x=fastq[1], barcode=c("TT", "AA", "GG"), nreads=50)
```
Outline

Overview

String Handling Utilities in R's Base Distribution

Sequence Handling with Bioconductor

Range Operations
Important Data Objects for Range Operations

- IRanges: stores range data only (IRanges library)
- GRanges: stores ranges and annotations (GenomicRanges library)
- GRangesList: list version of GRanges container (GenomicRanges library)
Range Data Are Stored in IRanges and GRanges Containers

Constructing GRanges Objects

```r
> library(GenomicRanges); library(rtracklayer)
> gr <- GRanges(seqnames = Rle(c("chr1", "chr2", "chr1", "chr3"), c(1, 3, 2, 4)), ranges = IRanges(1:10, end = 7:16, ... 3, 2)), score = 1:10, GC = seq(1, 0, length = 10)) # Example of creating a GRanges object with its constructor function.
+ asRangedData=FALSE) # Imports a simplified GFF3 genome annotation file.
> seqlengths(gff) <- end(ranges(gff[which(values(gff)[,"type"] == "chromosome"),]))
> names(gff) <- 1:length(gff) # Assigns names to corresponding slot.
> gff[1:4,]
```

GRanges with 4 ranges and 5 metadata columns:

```
  seqnames ranges strand | source type score phase 
   <Rle>  <IRanges> <Rle> | <factor> <factor> <numeric> <integer>
1  Chr1   [ 1, 30427671] + | TAIR10 chromosome <NA> <NA> ID=AT1G01010;Note=p
2  Chr1   [3631, 5899]   + | TAIR10 gene   <NA> <NA> ID=AT1G01010.1;Parent=AT1G01010.1;Name=AT1G01010.1
3  Chr1   [3631, 5899]   + | TAIR10 mRNA   <NA> <NA> ID=AT1G01010.1-Protein;Name=AT1G...
4  Chr1   [3760, 5630]   + | TAIR10 protein <NA> <NA> ID=AT1G01010.1-Protein;Name=AT...
```

```
seqlengths:
  Chr1  Chr2  Chr3  Chr4  Chr5  ChrC  ChrM
  30427671 19698289 23459830 18585056 26975502 154478 366924
```

```r
> gff_rd <- as(gff, "RangedData") # Coerces GRanges object to RangedData class.
> gff_gr <- as(gff_rd, "GRanges") # Coerces RangedData object to GRanges class.
```
Utilities for Range Containers

Accessor and subsetting methods for GRanges objects

> c(gff[1:2], gff[401:402]) # GRanges objects can be concatenated with the c() function.
> seqnames(gff); ranges(gff); strand(gff); seqlengths(gff) # Accessor functions
> start(gff[1:4]); end(gff[1:4]); width(gff[1:4]) # Direct access to IRanges components
> values(gff); values(gff)[, "type"] # Accessing metadata component.
> gff[elementMetadata(gff)[, "type"] == "gene"] # Returns only gene ranges.

Useful utilities for GRanges objects

> gff <- gff[values(gff)$type != "chromosome"] # Remove chromosome ranges
> strand(gff) <- "*" # Erases the strand information
> reduce(gff) # Collapses overlapping ranges to continuous ranges.
> gaps(gff) # Returns uncovered regions.
> disjoint(gff) # Returns disjoint ranges.
> coverage(gff) # Returns coverage of ranges.
> findOverlaps(gff, gff[1:4]) # Returns the index pairings for the overlapping ranges.
> countOverlaps(gff, gff[1:4]) # Counts overlapping ranges
> subsetByOverlaps(gff, gff[1:4]) # Returns only overlapping ranges

GRangesList Objects

> sp <- split(gff, seq(along=gff)) # Stores every range in separate component of a GRangesList object
> split(gff, seqnames(gff)) # Stores ranges of each chromosome in separate component.
> unlist(sp) # Returns data as GRanges object
> sp[1:4, "type"] # Subsetting of GRangesList objects is similar to GRanges objects.
> lapply(sp[1:4], length); sapply(sp[1:4], length) # Looping over GRangesList objects similar to lists
Storing annotation ranges in *TranscriptDb* databases makes many operations more robust and convenient.

```r
> library(GenomicFeatures)
> txdb <- makeTranscriptDbFromGFF(file="data/gff3.gff",
+ format="gff3",
+ dataSource="TAIR",
+ species="Arabidopsis thaliana")
> saveDb(txdb, file="./data/TAIR10.sqlite")
> txdb <- loadDb("./data/TAIR10.sqlite")
> tr <- transcripts(txdb)
> GRList <- transcriptsBy(txdb, by = "gene")
```
Generate *TranscriptDb* from BioMart

Alternative sources for creating *TranscriptDb* databases are BioMart, Bioc annotation packages, UCSC, etc. The following shows how to create a *TranscriptDb* from BioMart.

```r
> library(GenomicFeatures); library("biomaRt")
> txdb <- makeTranscriptDbFromBiomart(biomart = "plants_mart_20", dataset = "athaliana_eg_gene")
```

The following steps are useful to find out what is available on BioMart.

```r
> listMarts() # Lists BioMart databases
> mymart <- useMart("plants_mart_20") # Select one, here plants_mart_20
> listDatasets(mymart) # List datasets available in the selected BioMart database
> mymart <- useMart("plants_mart_20", dataset="athaliana_eg_gene")
> listAttributes(mymart) # List available features
> getBM(attributes=c("ensembl_gene_id", "description"), mart=mymart)[1:4,]
```
Efficient Sequence Parsing with `getSeq`

The following parses all annotation ranges provided by GRanges object (e.g. gff) from a genome sequence stored in a local file.

```r
> rand <- DNAStringSet(sapply(unique(as.character(seqnames(gff))), function(x) paste(sample(c("A","T","G","C"), 200000, replace=T), collapse="")))
> writeXStringSet(DNAStringSet(rand), ".\data\test")
> getSeq(FaFile("./data/test"), gff)
```

A DNAStringSet instance of length 449

width seq names
[1] 200000 GATTGCACATGGCAACACCGTGTAGATGGTAGTTACGTGTATGCGTCAAGCATAGAATGGTACCGGAGTGCAAGCGTGTAATACG...TC
[2] 2269 TAATCTAGACTACCGCTTCCCCCTTTTATCGGTGTCTTGAGCCAGCTCTAAAGGTGCTGCTATAGCGAGACGGCGCTTTTAGC...TA
[3] 2269 TAATCTAGACTACCGCTTCCCCCTTTTATCGGTGTCTTGAGCCAGCTCTAAAGGTGCTGCTATAGCGAGACGGCGCTTTTAGC...TA
[4] 1871 AACCACGGGAAACTAAGACCCGAAGGGATGGCATGCTAGGTTCGACAAGGAGCGAGCCCTAAATACAGAGTTCTAGCATCCTTTAATACCGGAGG...GT
[5] 283 TAATCTAGACTACCGCTTCCCCCTTTTATCGGTGTCTTGAGCCAGCTCTAAAGGTGCTGCTATAGCGAGACGGCGCTTTTAGC...TA
... ... ...
[445] 324 TTAACCCACCATGTCGCTCAGAGCTAAGGCTTCGCTTCTTTTGTAGGCTGCTCGTAATACCGAGACGGCGCTTTTAGC...TA
[446] 324 TTAACCCACCATGTCGCTCAGAGCTAAGGCTTCGCTTCTTTTGTAGGCTGCTCGTAATACCGAGACGGCGCTTTTAGC...TA
[447] 324 TTAACCCACCATGTCGCTCAGAGCTAAGGCTTCGCTTCTTTTGTAGGCTGCTCGTAATACCGAGACGGCGCTTTTAGC...TA
[448] 324 TTAACCCACCATGTCGCTCAGAGCTAAGGCTTCGCTTCTTTTGTAGGCTGCTCGTAATACCGAGACGGCGCTTTTAGC...TA
[449] 324 TTAACCCACCATGTCGCTCAGAGCTAAGGCTTCGCTTCTTTTGTAGGCTGCTCGTAATACCGAGACGGCGCTTTTAGC...TA
Exercise II

**GFF** from *Halobacterium sp*

**Genome** from *Halobacterium sp*

**Task 2** Extract gene ranges, parse their sequences from genome and translate them into proteins

**Task 3** Reduce overlapping genes and parse their sequences from genome

**Task 4** Generate intergenic ranges and parse their sequences from genome

**Useful commands**

```r
> chr <- readDNAStringSet("data/AE004437.fna")
> gff <- import("data/AE004437.gff", asRangedData=FALSE)
> gffgene <- gff[values(gff)[,"type"]="gene"]
> gene <- DNAStringSet(Views(chr[[1]], IRanges(start(gffgene), end(gffgene))))
> names(gene) <- values(gffgene)[,"locus_tag"]
> pos <- values(gffgene[strand(gffgene) == "+"])[,"locus_tag"]
> p1 <- translate(gene[names(gene) %in% pos])
> names(p1) <- names(gene[names(gene) %in% pos])
> neg <- values(gffgene[strand(gffgene) == "]-"))[,"locus_tag"]
> p2 <- translate(reverseComplement(gene[names(gene) %in% neg]))
> names(p2) <- names(gene[names(gene) %in% neg])
> writeXStringSet(c(p1, p2), "mypep.fasta")
```