Practical: Annotation and Ranges

Martin Morgan (mtmorgan@fhcrc.org)

November 18-22, 2013

Contents

1 Gene annotation

1.1 Data packages

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 usually 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 1.

Exercise 1 *This exercise illustrates basic use of the 'select' interface to annotation packages.*

a. *What is the name of the org package for* Homo sapiens*? Load it. Display the OrgDb object for the org.Hs.eg.db package. Use the `columns` method to discover which sorts of annotations can be extracted from it.*
b. *Use the `keys` method to extract ENSEMBL identifiers and then pass those keys in to the `select` method in such a way that you extract the SYMBOL (gene symbol) and GENENAME information for each. Use the following ENSEMBL ids.*

Table 1: Common operations for retrieving and manipulating annotations.

Category	Function	Description
Discover	`columns`	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 `columns`
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*.

```
ensids <- c("ENSG00000130720", "ENSG00000103257", "ENSG00000156414",
            "ENSG00000144644", "ENSG00000159307", "ENSG00000144485")
```

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

```
library(org.Hs.eg.db)
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"
```

```
columns(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"
```

```
cols <- c("SYMBOL", "GENENAME")
select(org.Hs.eg.db, keys=ensids, columns=cols, keytype="ENSEMBL")
```

```
##           ENSEMBL SYMBOL
## 1 ENSG00000130720 FIBCD1
## 2 ENSG00000103257 SLC7A5
## 3 ENSG00000156414  TDRD9
## 4 ENSG00000144644  GADL1
## 5 ENSG00000159307 SCUBE1
## 6 ENSG00000144485   HES6
##                                                                  GENENAME
## 1                                                fibrinogen C domain containing 1
## 2 solute carrier family 7 (amino acid transporter light chain, L system), member 5
## 3                                                      tudor domain containing 9
## 4                                                glutamate decarboxylase-like 1
```

Table 2: Selected packages querying web-based annotation services.

Package	Description
AnnotationHub	Ensembl, Encode, dbSNP, UCSC data objects
biomaRt	http://biomart.org, Ensembl and other annotations
PSICQUIC	https://code.google.com/p/psicquic.org, protein interactions
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.
rtracklayer	http://genome.ucsc.edu, genome tracks.
GEOquery	http://www.ncbi.nlm.nih.gov/geo/, array and other data
ArrayExpress	http://www.ebi.ac.uk/arrayexpress/, array and other data

```
## 5                                          signal peptide, CUB domain, EGF-like 1
## 6                                        hairy and enhancer of split 6 (Drosophila)
```

1.2 Internet resources

A short summary of select *Bioconductor* packages enabling web-based queries is in Table 2.

Using biomaRt The *biomaRt* package offers access to the online biomart 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.

Exercise 2 *warning: This exericse requires INTERNET ACCESS*

a. *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.*
b. *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:

```
## NEEDS INTERNET ACCESS !!
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
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 3 *As an optional exercise, annotate the genes that are differentially expressed in the DESeq2 laboratory, e.g., find the `GENENAME` associated with the five most differentially expressed genes. Do these make biological sense? Can you `merge` the annotation results with the 'top table' results to provide a statistically and biologically informative summary?*

2 Genome annotation

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
Homo.sapiens For integrating *TxDb** and *org.** packages.
SNPlocs.* For model organism SNP locations derived from dbSNP.
`FaFile` (*Rsamtools*) for accessing indexed FASTA files.
SIFT.*, PolyPhen, ensemblVEP Variant effect scores.

2.1 Transcript annotation packages

Genome-centric packages are very useful for annotations involving genomic coordinates. It is straight-forward, 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 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.

2.2 *rtracklayer*

The *rtracklayer* package allows us to query the UCSC genome browser, as well as providing `import` and `export` functions for common annotation file formats like GFF, GTF, and BED.

Exercise 4 *warning: This exericse requires INTERNET ACCESS*

Here we use rtracklayer to retrieve estrogen receptor binding sites identified across cell lines in the ENCODE project. We focus on binding sites in the vicinity of a particularly interesting region of interest.

a. *Define our region of interest by creating a GRanges instance with appropriate genomic coordinates. Our region corresponds to 10Mb up- and down-stream of a particular gene.*
b. *Create a session for the UCSC genome browser*
c. *Query the UCSC genome browser for ENCODE estrogen receptor ERalpha$_a$ transcription marks; identifying the appropriate track, table, and transcription factor requires biological knowledge and detective work.*
d. *Visualize the location of the binding sites and their scores; annotate the mid-point of the region of interest.*

Solution: Define the region of interest

```
roi <- GRanges("chr10", IRanges(92106877, 112106876, names="ENSG00000099194"))
```

Create a session

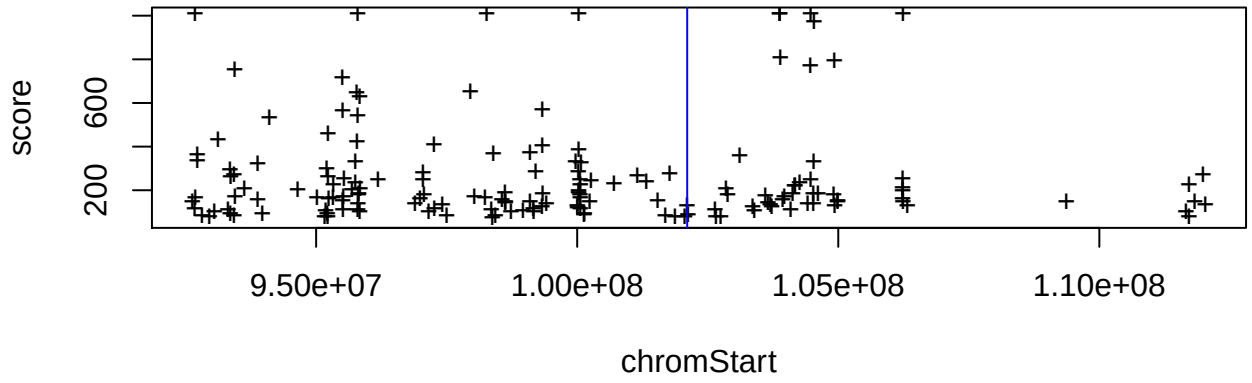
```
library(rtracklayer)
session <- browserSession()
```

Query the UCSC for a particular track, table, and transcription factor, in our region of interest

```
trackName <- "wgEncodeRegTfbsClusteredV2"
tableName <- "wgEncodeRegTfbsClusteredV2"
trFactor <- "ERalpha_a"
ucscTable <- getTable(ucscTableQuery(session, track=trackName,
    range=roi, table=tableName, name=trFactor))
```

Visualize the result

```
plot(score ~ chromStart, ucscTable, pch="+")
abline(v=start(roi) + (end(roi) - start(roi) + 1) / 2, col="blue")
```



3 Working with ranges

Start by loading the *GenomicRanges* package and defining the `plotRanges` helper function

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. These are implemented in the *GenomicRanges* package; see [1] for a more comprehensive conceptual orientation.

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

Table 3: 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.

```
genes <- GRanges(seqnames=c("chr3R", "chrX"),
                 ranges=IRanges(
                     start=c(19967117, 18962306),
                     end = c(19973212, 18962925)),
                 strand=c("+", "-"),
                 seqlengths=c(chr3R=27905053, chrX=22422827))
```

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]    chr3R [19967117, 19973212]      +
##   [2]     chrX [18962306, 18962925]      -
##   ---
##   seqlengths:
##       chr3R     chrX
##    27905053 22422827
```

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

```
?GRanges
```

and package vignettes

```
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]     chrX [18962306, 18962925]      -
##   ---
##   seqlengths:
##       chr3R     chrX
```

```
##    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    chr3R [19967117, 19973212]      +
##   FBgn0085359     chrX [18962306, 18962925]      -
##   ---
##   seqlengths:
##       chr3R     chrX
##    27905053 22422827
```

`strand` returns the strand information in a compact representation called a *run-length encoding*. 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.

Operations on ranges The *GRanges* class has many useful methods. 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 1 and summarized in Table 4.

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 1.

```
shift(ir, 5)
## IRanges of length 7
```

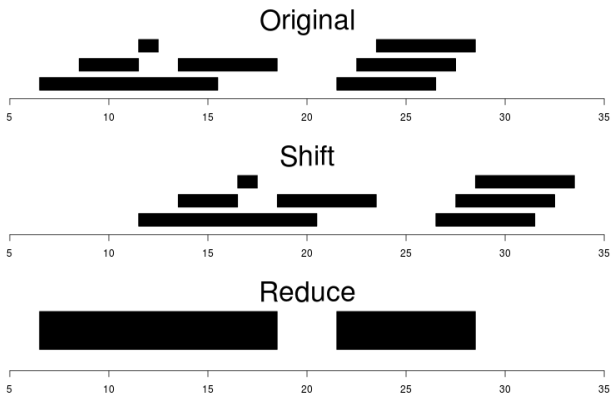


Figure 1: Ranges

```
##     start end width
## [1]    12  20     9
## [2]    14  16     3
## [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 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)

```
cvg <- coverage(ir)
cvg
## 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
## plot(as.integer(cvg), type="s", xlab="Coordinate", ylab="Depth of coverage")
```

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'.

Table 4: 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

Adding `mcols` and `metadata` 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())
```

The *GRangesList* class 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 ENSEMBL genes identified earlier can be represented as a *GRangesList*.

```
## GRangesList of length 6:
## $84929
## GRanges with 10 ranges and 2 metadata columns:
##        seqnames               ranges strand |   exon_id   exon_name
##           <Rle>            <IRanges>  <Rle> | <integer> <character>
##    [1]     chr9 [133777825, 133779710]      - |    132272        <NA>
##    [2]     chr9 [133780621, 133780800]      - |    132273        <NA>
##    [3]     chr9 [133787179, 133787275]      - |    132274        <NA>
##    [4]     chr9 [133799131, 133799267]      - |    132275        <NA>
##    [5]     chr9 [133799624, 133799783]      - |    132276        <NA>
##    [6]     chr9 [133804954, 133805433]      - |    132277        <NA>
##    [7]     chr9 [133806160, 133806183]      - |    132278        <NA>
##    [8]     chr9 [133813923, 133814035]      - |    132279        <NA>
##    [9]     chr9 [133813923, 133814239]      - |    132280        <NA>
##   [10]     chr9 [133814390, 133814455]      - |    132281        <NA>
##
## $8140
## GRanges with 10 ranges and 2 metadata columns:
##        seqnames               ranges strand | exon_id exon_name
##    [1]    chr16 [87863629, 87866631]      - |  215168      <NA>
##    [2]    chr16 [87868020, 87868197]      - |  215169      <NA>
##    [3]    chr16 [87870104, 87870253]      - |  215170      <NA>
##    [4]    chr16 [87871451, 87871547]      - |  215171      <NA>
##    [5]    chr16 [87872320, 87872423]      - |  215172      <NA>
##    [6]    chr16 [87873308, 87873431]      - |  215173      <NA>
##    [7]    chr16 [87874035, 87874079]      - |  215174      <NA>
##    [8]    chr16 [87874656, 87874761]      - |  215175      <NA>
##    [9]    chr16 [87885330, 87885455]      - |  215176      <NA>
##   [10]    chr16 [87902491, 87903100]      - |  215177      <NA>
##
## ...
## <4 more elements>
## ---
## seqlengths:
##                   chr1                 chr2 ...        chrUn_gl000249
##              249250621            243199373 ...                 38502
```

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.

3.1 Selecting gene sequences

Exercise 5 *This exercise uses annotation packages to go from gene identifiers to coding sequences.*

a. *Map from an informal gene SYMBOL, e.g., BRCA1, to ENTREZID gene identifiers using the org.Hs.eg.db package and the `select` function, use the TxDb.Hsapiens.UCSC.hg19.knownGene package and a second map to go from ENTREZID to TXNAME.*
b. *Extract the coding sequence grouped by transcript using the TxDb.Hsapiens.UCSC.hg19.knownGene package and `cdsBy` function; select just those transcripts we are interested in.*
c. *Retrieve the nucleotide sequence from the BSgenome.Hsapiens.UCSC.hg19 package using the function `extractTranscriptsFromGenome`.*

d. Verify that the coding sequences are all multiples of 3, and translate from nucleotide to amino acid sequence.

Solution: Map from gene SYMBOL to ENTREZID, and from ENTREZID to TXNAME

```
library(org.Hs.eg.db)
egid <- select(org.Hs.eg.db, "BRCA1", "ENTREZID", "SYMBOL")$ENTREZID
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
egToTx <- select(txdb, egid, "TXNAME", "GENEID")
```

```
## Warning:  'select' resulted in 1:many mapping between keys and return rows
```

Extract the releveant coding sequence, grouped by transcript

```
cdsByTx <- cdsBy(txdb, "tx", use.names=TRUE)[egToTx$TXNAME]
```

Retrieve the sequence

```
library(BSgenome.Hsapiens.UCSC.hg19)
txx <- extractTranscriptsFromGenome(Hsapiens, cdsByTx)
```

Translate to amino acid sequence

```
all(width(txx) %% 3 == 0)  # sanity check
```

```
## [1] TRUE
```

```
translate(txx)             # amino acid sequence
```

```
##   A AAStringSet instance of length 20
##      width seq
##  [1]   760 MDLSALRVEEVQNVINAMQKILECPICLELIKEPVSTK...MCEAPVVTREWVLDSVALYQCQELDTYLIPQIPHSHY*
##  [2]  1793 MSLQESTRFSQLVEELLKIICAFQLDTGLEYANSYNFA...MCEAPVVTREWVLDSVALYQCQELDTYLIPQIPHSHY*
##  [3]   174 MDAEFVCERTLKYFLGIAGGKWVVSYFWVTQSIKERKM...MCEAPVVTREWVLDSVALYQCQELDTYLIPQIPHSHY*
##  [4]   700 MDLSALRVEEVQNVINAMQKILECPICLELIKEPVSTK...CCYGPFTNMPTGCPPNCGCAARCLDRGQWLPCNWADV*
##  [5]  1817 MLKLLNQKKGPSQCPLCKNDITKRSLQESTRFSQLVEE...MCEAPVVTREWVLDSVALYQCQELDTYLIPQIPHSHY*
##  ...   ... ...
## [16]  1365 MDLSALRVEEVQNVINAMQKILECPICLELIKEPVSTK...SESQGVGLSDKELVSDDEERGTGLEENNQEEQSMDSNL
## [17]  1365 MDLSALRVEEVQNVINAMQKILECPICLELIKEPVSTK...SESQGVGLSDKELVSDDEERGTGLEENNQEEQSMDSNL
## [18]  1318 MLKLLNQKKGPSQCPLCKNDITKRSLQESTRFSQLVEE...SESQGVGLSDKELVSDDEERGTGLEENNQEEQSMDSNL
## [19]  1339 MDLSALRVEEVQNVINAMQKILECPICLELIKEPVSTK...SESQGVGLSDKELVSDDEERGTGLEENNQEEQSMDSNL
## [20]  1069 MNVEKAEFCNKSKQPGLARSQHNRWAGSKETCNDRRTP...SESQGVGLSDKELVSDDEERGTGLEENNQEEQSMDSNL
```

3.2 Summarizing overlaps

comment: This repeats an exercise from Day 1

Exercise 6 *A basic operation in RNA-seq and other work flows is to count the number of times aligned reads overlap features of interest.*

a. Load the RNAseqData.HNRNPC.bam.chr14 experiment data package and get the paths to the BAM files it contains.
b. Load the 'transcript db' package that contains the coordinates of each exon of the UCSC 'known genes' track of hg19.
c. Extract the exon coordinates grouped by gene; the result is an GRangesList object that we will discuss more latter.

d. *Use the `summarizeOverlaps` function with the exon coordinates and BAM files to generate a count of the number of reads overlapping each gene. Visit the help page `?summarizeOverlaps` to read about the counting strategy used.*
e. *The counts can be extracted from the return value of `summarizeOverlaps` using the function `assay`. This is standard R matrix. How many reads overlapped regions of interest in each sample? How many genes had non-zero counts?*

Solution: Point to BAM files

```
library(RNAseqData.HNRNPC.bam.chr14)
fls <- RNAseqData.HNRNPC.bam.chr14_BAMFILES
```

Get the gene model; this could also come from, e.g., a GFF or GTF file.

```
## library(parallel); options(mc.cores=detectCores())
library(TxDb.Hsapiens.UCSC.hg19.knownGene)
ex <- exonsBy(TxDb.Hsapiens.UCSC.hg19.knownGene, "gene")
```

Summarize the number of reads overlapping each region of interest

```
counts <- summarizeOverlaps(ex, fls)
colSums(assay(counts))
```

```
## ERR127306 ERR127307 ERR127308 ERR127309 ERR127302 ERR127303 ERR127304 ERR127305
##    340669    373302    371666    331540    313817    331160    331639    329672
```

```
sum(rowSums(assay(counts)) != 0)
```

```
## [1] 528
```

References

[1] Michael Lawrence, Wolfgang Huber, Hervé Pagès, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin T. Morgan, and Vincent J. Carey. Software for computing and annotating genomic ranges. *PLoS Comput Biol*, 9(8):e1003118, 08 2013. URL: http://dx.doi.org/10.1371%2Fjournal.pcbi.1003118, doi:10.1371/journal.pcbi.1003118.