Chapter 4

Genomic Annotation Resources in R/Bioconductor

Marc R.J. Carlson, Hervé Pagès, Sonali Arora, Valerie Obenchain, and Martin Morgan

Abstract

Annotation resources make up a significant proportion of the Bioconductor project (Huber et al., Nat Methods 12:115–121, 2015). And there are also a diverse set of online resources available which are accessed using specific packages. Here we describe the most popular of these resources and give some high level examples on how to use them.

Key words Annotation, Next-generation sequencing, R, Bioconductor, Genomics

1 Introduction

Annotations in Bioconductor have traditionally been used near the end of an analysis. After the bulk of the data analysis, annotations would be used interpretatively to learn about the most significant results. But increasingly, they are also used as a starting point or even as an intermediate step to help guide a study that is still in progress. In addition to this, what it means for something to be an annotation is also becoming less clear than it once was. It used to be clear that annotations were only those things that had been established after multiple different studies had been performed (such as the primary role of a gene product). But today many large data sets are treated by communities in much the same way that classic annotations once were: as a reference for additional comparisons.

Another change that is underway with annotations in Bioconductor is in the way that they are obtained. In the past annotations existed almost exclusively as separate annotation packages [2–4]. Today packages are still an enormous source of annotations. The current repository contains over 800 annotation packages (http://bioconductor.org/packages/release/BiocViews.html#___AnnotationData). Here is a table that summarizes some of the more

Ewy Mathé and Sean Davis (eds.), *Statistical Genomics: Methods and Protocols*, Methods in Molecular Biology, vol. 1418, DOI 10.1007/978-1-4939-3578-9_4, © Springer Science+Business Media New York 2016

Table 1
Summary of popular Annotation Objects

Object type	Example package name	Contents
OrgDb	org.Hs.eg.db	Gene based information for *Homo sapiens*
TxDb	TxDb.Hsapiens.UCSC.hg19.knownGene	Transcriptome ranges for *Homo sapiens*
OrganismDb	Homo.sapiens	Composite information for *Homo sapiens*
BSgenome	BSgenome.Hsapiens.UCSC.hg19	Genome sequence for *Homo sapiens*

important classes of annotation objects that are often accessed using packages (Table 1).

But in spite of the popularity of annotation packages, annotations are increasingly also being pulled down from Web services like biomaRt [5–7] or from the AnnotationHub [8]. And both of these represent enormous resources for annotation data.

In part because of the rapidly evolving landscape, it is currently impossible in a single chapter to cover every possible annotation or even every kind of annotation present in Bioconductor. So this chapter instead goes over the most popular annotation resources and describe them in a way intended to expose common patterns used for accessing them. The hope is that a user with this information will be able to make educated guesses about how to find and use additional resources that will inevitably be added later. Topics that are covered include the following:

- Using the AnnotationHub
- OrgDb Objects
- TxDb Objects
- OrganismDb Objects
- BSgenome Objects
- Using the biomaRt Web service
- Creating annotation objects

2 Materials

In this chapter we make use of several Bioconductor packages. You can install them with by using biocLite() like so:

```
source("http://bioconductor.org/biocLite.R")
biocLite("AnnotationHub")
```

```
biocLite("Rattus.norvegicus")
biocLite("TxDb.Dmelanogaster.UCSC.dm3.ensGene")
biocLite("BSgenome.Dmelanogaster.UCSC.dm3")
biocLite("biomaRt")
```

The usage of the installed packages is described in detail within Subheading 3.

3 Methods

3.1 Using the AnnotationHub

The top of the list for learning about annotation resources is the relatively new AnnotationHub package [8]. The AnnotationHub was created to provide a convenient access point for end users to find a large range of different annotation objects for use with Bioconductor. Resources found in the AnnotationHub are easy to discover and are presented to the user as familiar Bioconductor data objects. Because it is a recent addition, the AnnotationHub allows access to a broad range of annotation like objects, some of which may not have been considered annotations even a few years ago. To get started with the AnnotationHub users only need to load the package and then create a local AnnotationHub object like this:

```
library("AnnotationHub")
ah <- AnnotationHub()
```

The very first time that you call the AnnotationHub, it will create a cache directory on your system and download the latest metadata for the hubs current contents. From that time forward, whenever you download one of the hubs data objects, it will also cache those files in the local directory so that if you request the information again, you will be able to access it quickly.

The show method of an AnnotationHub object will tell you how many resources are currently accessible using that object as well as give a high level overview of the most common kinds of data present.

```
ah
## AnnotationHub with 19268 records
## # snapshotDate(): 2015-03-26
## # $dataprovider: UCSC, Ensembl, BroadInstitute, NCBI, Haemcode, dbSNP, ...
## # $species: Homo sapiens, Mus musculus, Bos taurus, Pan troglodytes, Da...
## # $rdataclass: GRanges, FaFile, OrgDb, ChainFile, CollapsedVCF, Inparan...
## # additional mcols(): taxonomyid, genome, description, tags,
## #  sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["AH169"]]'
##
##        title
##  AH169   | Meleagris_gallopavo.UMD2.69.cdna.all.fa
##  AH170   | Meleagris_gallopavo.UMD2.69.dna.toplevel.fa
```

```
##  AH171   | Meleagris_gallopavo.UMD2.69.dna_rm.toplevel.fa
##  AH172   | Meleagris_gallopavo.UMD2.69.dna_sm.toplevel.fa
##  AH173   | Meleagris_gallopavo.UMD2.69.ncrna.fa
##  ...       ...
##  AH28851 | Tursiops_truncatus.turTru1.77.gtf
##  AH28852 | Vicugna_pacos.vicPac1.77.gtf
##  AH28853 | Xenopus_tropicalis.JGI_4.2.77.gtf
##  AH28854 | Xiphophorus_maculatus.Xipmac4.4.2.77.gtf
##  AH28855 | RNA-Sequencing and clinical data for 7706 tumor samples fro...
```

As you can see from the object above, there are a LOT of different resources available. So normally when you get an AnnotationHub object the first thing you want to do is to filter it to remove unwanted resources.

Fortunately, the AnnotationHub has several different kinds of metadata that you can use for searching and subsetting. To see the different categories all you need to do is to type the name of your AnnotationHub object and then tab complete from the "$" operator. And to see all possible contents of one of these categories you can pass that value in to unique like this:

```
unique(ah$dataprovider)
## [1] "Ensembl"                    "EncodeDCC"
## [3] "UCSC"                       "dbSNP"
## [5] "Inparanoid8"                "NCBI"
## [7] "BroadInstitute"             "NHLBI"
## [9] "ChEA"                       "Pazar"
## [11] "NIH Pathway Interaction Database" "RefNet"
## [13] "Haemcode"                  "GEO"
```

One of the most valuable ways in which the data is labeled is according to the kind of R object that will be returned to you.

```
unique(ah$rdataclass)
## [1] "FaFile"      "GRanges"     "CollapsedVCF"
## [4] "Inparanoid8Db"  "OrgDb"      "TwoBitFile"
## [7] "ChainFile"   "SQLiteConnection" "data.frame"
## [10] "biopax"     "VcfFile"     "ExpressionSet"
```

Once you have identified which sorts of metadata you would like to use to find your data of interest, you can then use the subset or query methods to reduce the size of the hub object to something more manageable. For example you could select only those records where the string "GRanges" was in the metadata. As you can see GRanges are one of the more popular formats for data that comes from the AnnotationHub.

```
grs <- query(ah, "GRanges")
grs
```

```
## AnnotationHub with 12390 records
## # snapshotDate(): 2015-03-26
```

```
## # $dataprovider: UCSC, BroadInstitute, Haemcode, Ensembl, Pazar, EncodeDCC
## # $species: Homo sapiens, Mus musculus, Bos taurus, Pan troglodytes, Ca...
## # $rdataclass: GRanges
## # additional mcols(): taxonomyid, genome, description, tags,
## #  sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["AH3166"]]'
##
##       title
##  AH3166 | wgEncodeRikenCageSknshraCellPapTssHmm
##  AH3912 | wgEncodeUwDgfTregwb78495824Hotspots
##  AH3913 | wgEncodeUwDgfTregwb78495824Pk
##  AH4368 | wgEncodeUwDnaseWi38PkRep1
##  AH4369 | wgEncodeUwDnaseWi38PkRep2
##  ...   ...
##  AH28850 | Tupaia_belangeri.TREESHREW.77.gtf
##  AH28851 | Tursiops_truncatus.turTru1.77.gtf
##  AH28852 | Vicugna_pacos.vicPac1.77.gtf
##  AH28853 | Xenopus_tropicalis.JGI_4.2.77.gtf
##  AH28854 | Xiphophorus_maculatus.Xipmac4.4.2.77.gtf
```

Or you can use subsetting to only select for matches on a specific field

```
grs <- ah[ah$rdataclass == "GRanges",]
```

The subset function is also provided.

```
orgs <- subset(ah, ah$rdataclass == "OrgDb")
orgs
```

```
## AnnotationHub with 1145 records
## # snapshotDate(): 2015-03-26
## # $dataprovider: NCBI
## # $species: 'Nostoc azollae'_0708, Acaryochloris marina_MBIC11017, Acet...
## # $rdataclass: OrgDb
## # additional mcols(): taxonomyid, genome, description, tags,
## #  sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["AH12818"]]'
##
##       title
##  AH12818 | org.Pseudomonas_mendocina_NK-01.eg.sqlite
##  AH12819 | org.Streptomyces_coelicolor_A3(2).eg.sqlite
##  AH12820 | org.Cricetulus_griseus.eg.sqlite
##  AH12821 | org.Streptomyces_cattleya_NRRL_8057_=_DSM_46488.eg.sqlite
##  AH12822 | org.Cavia_porcellus.eg.sqlite
##  ...   ...
##  AH13958 | org.Ochotona_princeps.eg.sqlite
##  AH13959 | org.Aeromonas_veronii_B565.eg.sqlite
##  AH13960 | org.Oryctolagus_cuniculus.eg.sqlite
##  AH13961 | org.Tetraodon_nigroviridis.eg.sqlite
##  AH13962 | org.Burkholderia_gladioli_BSR3.eg.sqlite
```

And if you really need access to all the metadata you can extract it as a DataFrame using mcols() like so:

```
meta <- mcols(ah)
```

Also if you are a fan of GUI's you can use the display method to look at your data in a browser and return selected rows back as a smaller AnnotationHub object like this:

```
sah <- display(ah)
```

Calling this method will produce a Web based interface like the one pictured here (Fig. 1).

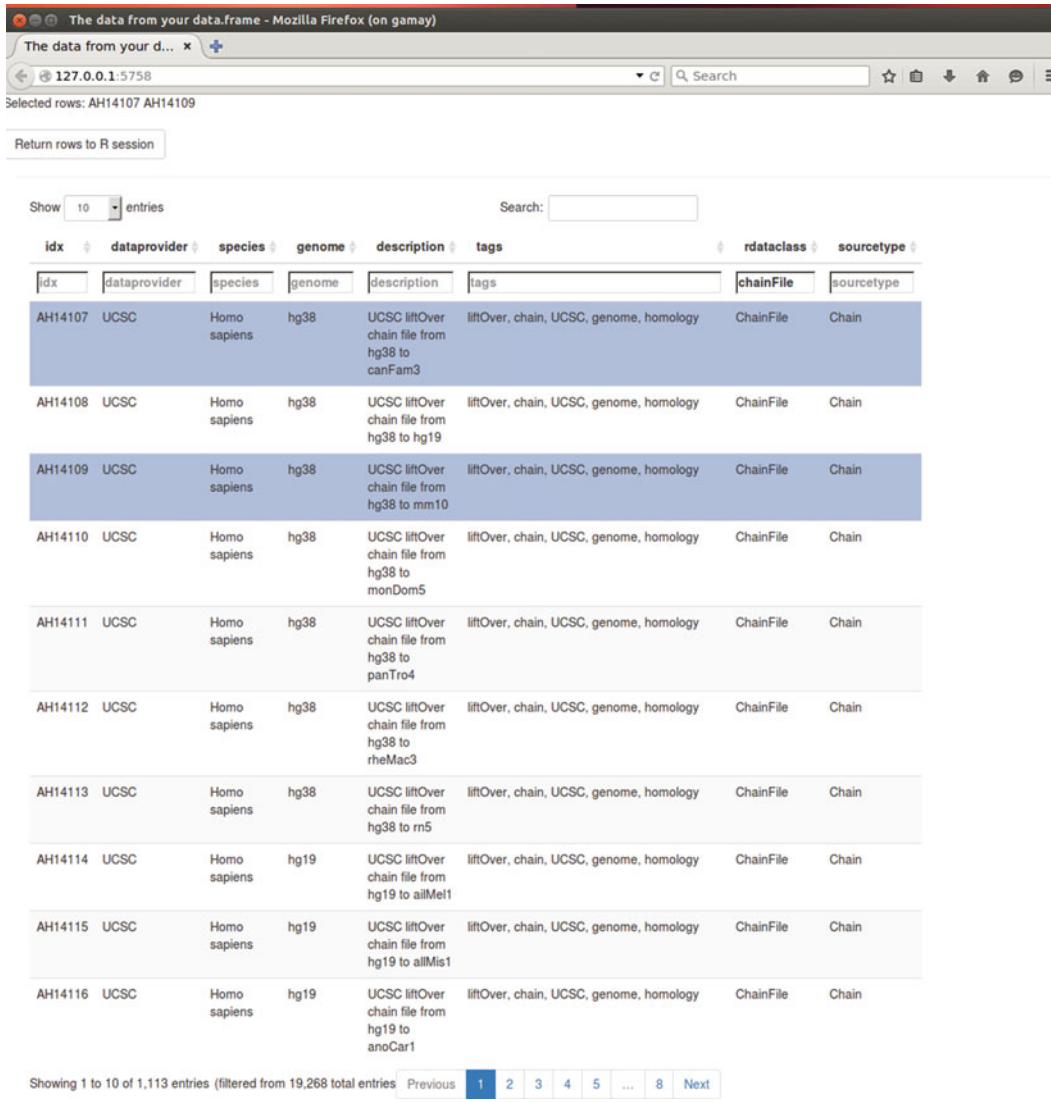

Fig. 1 The display() method will open a user friendly GUI in a local Web browser (if available)

Once you have the AnnotationHub object pared down to a reasonable size, and are sure about which records you want to retrieve, then you only need to use the '[[' operator to extract them. Using the '[[' operator, you can extract by numeric index (1,2,3) or by AnnotationHub ID. If you choose to use the former, you simply extract the element that you are interested in. So for our chain example, you might just want to first one like this:

```
res <- grs[[1]]
## require("GenomicRanges")
head(res, n=3)
## GRanges object with 3 ranges and 5 metadata columns:
##    seqnames         ranges strand |
##      <Rle>       <IRanges>  <Rle> |
##  [1]  chr11 [65266509, 65266572]   + |
##  [2]   chr1 [156675257, 156675415]   - |
##  [3]  chr10 [33247091, 33247233]   - |
##                        name   score   level
##                     <character> <integer> <numeric>
##  [1] chr11:65266509:65266572:+:0.04:1.000000     0 20993.670
##  [2] chr1:156675257:156675415:-:0.35:1.000000     0 11580.471
##  [3] chr10:33247091:33247233:-:0.32:1.000000     0  8098.093
##     signif  score2
##    <numeric> <integer>
##  [1] 3.36e-10      0
##  [2] 3.54e-10      0
##  [3] 3.82e-10      0
##  ------
##  seqinfo: 24 sequences from hg19 genome
```

Or you might have decided that you want to see the data for the green spotted pufferfish by that you spotted in the orgs subset under the name "AH13961". That data could also be extracted like this:

```
tetra <- orgs[["AH13961"]]
## Loading required package: AnnotationDbi
## Loading required package: Biobase
## Welcome to Bioconductor
##
##   Vignettes contain introductory material; view with
##   'browseVignettes()'. To cite Bioconductor, see
##   'citation("Biobase")', and for packages 'citation("pkgname")'.
##
##
## Attaching package: 'Biobase'
##
## The following object is masked from 'package:AnnotationHub':
```

```
##
##   cache
```

```
tetra
```

```
## OrgDb object:
## | DBSCHEMAVERSION: 2.1
## | DBSCHEMA: NOSCHEMA_DB
## | ORGANISM: Tetraodon nigroviridis
## | SPECIES: Tetraodon nigroviridis
## | CENTRALID: GID
## | TAXID: 99883
## | Db type: OrgDb
## | Supporting package: AnnotationDbi
##
## Please see: help('select') for usage information
```

3.2 OrgDb Objects

At this point you may be wondering: What is this OrgDb object about? OrgDb objects are one member of a family of annotation objects that all represent hidden data through a shared set of methods. So if you look closely at the tetra object in the example above you can see that it contains data for Tetraodon nigroviridis (taxonomy ID = 99883). You can learn a little more about it by learning about the columns method.

```
columns(tetra)
## [1] "ACCNUM"    "ALIAS"    "CHR"      "ENTREZID"   "GENENAME"
## [6] "SYMBOL"    "GID"      "GO"       "EVIDENCE"   "ONTOLOGY"
## [11] "GOALL"    "EVIDENCEALL" "ONTOLOGYALL" "PMID"     "REFSEQ"
```

The columns method gives you a vector of data types that can be retrieved from the object that you call it on. So the above call indicates that there are several different data types that can be retrieved from the tetra object.

A very similar method is the keytypes method, which will list all the data types that can also be used as keys.

```
keytypes(tetra)
## [1] "ACCNUM"    "ALIAS"    "ENTREZID"   "GENENAME"   "SYMBOL"
## [6] "GID"      "GO"       "EVIDENCE"   "ONTOLOGY"   "GOALL"
## [11] "EVIDENCEALL" "ONTOLOGYALL" "PMID"     "REFSEQ"
```

In many cases most of the things that are listed as columns will also come back from a keytypes call, but since these two things are not guaranteed to be identical, we must maintain two separate methods.

Now that you can see what kinds of things can be used as keys, you can call the keys method to extract out all the keys of a given key type.

```
keys(tetra, keytype="ENTREZID")
```

```
## [1] "3453248" "3453249" "3453250" "3453251" "3453252" "3453253" "3453254"
## [8] "3453255" "3453256" "3453257" "3453258" "3453259" "3474988"
```

This is useful if you need to get all the IDs of a particular kind but the keys method has a few extra arguments that can make it even more flexible. For example, using the keys method you could also extract the gene SYMBOLS that start with "COX" like this:

```
keys(tetra, keytype="SYMBOL", pattern="COX")
## [1] "COX1" "COX2" "COX3"
```

Or if you really needed an other keytype, you can use the column argument to extract the ENTREZ GENE IDs for those gene SYMBOLS that start with "COX":

```
keys(tetra, keytype="ENTREZID", pattern="COX", column="SYMBOL")
## [1] "3453250" "3453251" "3453252"
```

But often, you will really want to extract other data that matches a particular key or set of keys. For that there are two methods which you can use. The more powerful of these is probably select. Here is how you would look up the gene SYMBOL, and REFSEQ id for the entrez gene ID "3453250".

```
select(tetra, keys="3453250", columns=c("SYMBOL","REFSEQ"),
   keytype="ENTREZID")
##  ENTREZID SYMBOL   REFSEQ
## 1 3453250  COX1 YP_254663.1
```

When you call it, select will return a data.frame that attempts to fill in matching values for all the columns you requested. However, if you ask select for things that have a many to one relationship to your keys it can result in an expansion of the data object that is returned. For example, watch what happens when we ask for the GO terms for the same entrez gene ID:

```
select(tetra, keys="3453250", columns="GO", keytype=
"ENTREZID")
##  ENTREZID     GO
## 1 3453250 GO:0046686
## 2 3453250 GO:0051597
## 3 3453250 GO:0004129
## 4 3453250 GO:0009055
## 5 3453250 GO:0020037
## 6 3453250 GO:0009060
## 7 3453250 GO:0016021
```

Because there are seven GO terms associated with the gene "3453250", you end up with seven rows in the data.frame... This can become problematic if you then ask for several columns that have a many to one relationship to the original key. If you were to

do that, not only would the result multiply in size, it would also become really hard to use. So the recommended strategy is to be selective when using select.

Sometimes you might want to look up matching results in a way that is simpler than the data.frame object that select returns. This is especially true when you only want to look up one kind of value per key. For these cases, we recommend that you look at the mapIds method. Let us look at what happens if request the same basic information as the last select call, but instead using the mapIds method:

```
mapIds(tetra, keys="3453250", column="GO", keytype=
"ENTREZID")
```

```
##   3453250
## "GO:0046686"
```

As you can see, the mapIds method allows you to simplify the result that is returned. And by default, mapIds only returns the first matching element for each key. But what if you really need all those GO terms returned when you call mapIds? Well then you can make use of the mapIds multiVals argument. There are several options for this argument, we have already seen how by default you can return only the "first" element. But you can also return a "list" or "CharacterList" object, or you can "filter" out or return "asNA" any keys that have multiple matches. You can even define your own rule (as a function) and pass that in as an argument to multiVals. Let us look at what happens when you return a list:

```
mapIds(tetra, keys="3453250", column="GO", keytype=
"ENTREZID",
   multiVals="list")
```

```
## $'3453250'
##   [1]   "GO:0046686"   "GO:0051597"   "GO:0004129"
"GO:0009055" "GO:0020037"
## [6] "GO:0009060" "GO:0016021"
```

Now you know how to extract information from an OrgDb object, you might find it helpful to know that there is a whole family of other AnnotationDb derived objects that you can also use with these same five methods (keytypes(), columns(), keys(), select(), and mapIds()). For example there are ChipDb objects, InparanoidDb objects and TxDb objects which contain data about microarray probes, inparanoid homology partners or transcript range information respectively. And there are also more specialized objects like GODb or ReactomeDb objects which offer access to data from GO and reactome. In the next section, we look at one of the more popular classes of these objects: the TxDb object.

3.3 TxDb Objects

As mentioned before, TxDb objects can be accessed using the standard set of methods: keytypes(), columns(), keys(), select(), and mapIds(). But because these objects contain information

about a transcriptome, they are often used to compare range based information to these important features of the genome [3, 4]. As a result they also have specialized accessors for extracting out ranges that correspond to important transcriptome characteristics.

Let us start by loading a TxDb object from an annotation package based on the UCSC ensembl genes track for *Drosophila*. A common practice when loading these is to shorten the long name to "txdb" (just as a convenience).

```
library("TxDb.Dmelanogaster.UCSC.dm3.ensGene")
## Loading required package: GenomicFeatures
txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
txdb
## TxDb object:
## # Db type: TxDb
## # Supporting package: GenomicFeatures
## # Data source: UCSC
## # Genome: dm3
## # Organism: Drosophila melanogaster
## # UCSC Table: ensGene
## # Resource URL: http://genome.ucsc.edu/
## # Type of Gene ID: Ensembl gene ID
## # Full dataset: yes
## # miRBase build ID: NA
## # transcript_nrow: 29173
## # exon_nrow: 76920
## # cds_nrow: 62135
## # Db created by: GenomicFeatures package from
Bioconductor
## # Creation time: 2015-03-19 14:00:50 -0700 (Thu, 19 Mar
2015)
## # GenomicFeatures version at creation time: 1.19.32
## # RSQLite version at creation time: 1.0.0
## # DBSCHEMAVERSION: 1.1
```

Just by looking at the TxDb object, we can learn a lot about what data it contains including where the data came from, which build of the fly genome it was based on and the last time that the object was updated. One of the most common uses for a TxDb object is to extract various kinds of transcript data out of it. So for example you can extract all the transcripts out of the TxDb as a GRanges object like this:

```
txs <- transcripts(txdb)
txs
## GRanges object with 29173 ranges and 2 metadata columns:
##       seqnames      ranges strand |   tx_id   tx_name
##      <Rle>   <IRanges> <Rle> | <integer> <character>
```

```
##    [1]  chr2L [7529, 9484]   +  |     1 FBtr0300689
##    [2]  chr2L [7529, 9484]   +  |     2 FBtr0300690
##    [3]  chr2L [7529, 9484]   +  |     3 FBtr0330654
##    [4]  chr2L [21952, 24237]   +  |     4 FBtr0309810
##    [5]  chr2L [66584, 71390]   +  |     5 FBtr0306539
##    ...   ...        ...   ......    ...     ...
##   [29169]  chrYHet [205196, 205372]   -   |     29166
FBtr0302914
##   [29170]  chrYHet [307129, 307365]   -   |     29167
FBtr0114289
##   [29171]  chrYHet [312456, 313714]   -   |     29168
FBtr0114243
##   [29172]  chrYHet [319739, 320997]   -   |     29169
FBtr0114244
##   [29173]  chrYHet [327052, 328489]   -   |     29170
FBtr0114245
##  ———
##  seqinfo: 15 sequences (1 circular) from dm3 genome
```

Similarly, there are also extractors for exons(), cds(), genes() and promoters(). Which kind of feature you choose to extract just depends on what information you are after. These basic extractors are fine if you only want a flat representation of these data, but many of these features are inherently nested. So instead of extracting a flat GRanges object, you might choose instead to extract a GRangesList object that groups the transcripts by the genes that they are associated with like this:

```
txby <- transcriptsBy(txdb, by="gene")
txby
## GRangesList object of length 15682:
## $FBgn0000003
## GRanges object with 1 range and 2 metadata columns:
##    seqnames       ranges strand |   tx_id   tx_name
##     <Rle>    <IRanges> <Rle> | <integer> <character>
##  [1]  chr3R [2648220, 2648518]   + |   17345 FBtr0081624
##
## $FBgn0000008
## GRanges object with 3 ranges and 2 metadata columns:
##    seqnames       ranges strand | tx_id   tx_name
##  [1]  chr2R [18024494, 18060339]   + | 7681 FBtr0100521
##  [2]  chr2R [18024496, 18060346]   + | 7682 FBtr0071763
##  [3]  chr2R [18024938, 18060346]   + | 7683 FBtr0071764
##
## $FBgn0000014
## GRanges object with 4 ranges and 2 metadata columns:
##    seqnames       ranges strand | tx_id   tx_name
##  [1]  chr3R [12632936, 12655767]   - | 21863 FBtr0306337
```

```
##  [2]  chr3R [12633349, 12653845]   - | 21864 FBtr0083388
##  [3]  chr3R [12633349, 12655300]   - | 21865 FBtr0083387
##  [4]  chr3R [12633349, 12655474]   - | 21866 FBtr0300485
##
## ...
## <15679 more elements>
## ------
## seqinfo: 15 sequences (1 circular) from dm3 genome
```

Just as with the flat extractors, there is a whole family of extractors available depending on what you want to extract and how you want it grouped. They include transcriptsBy(), exonsBy(), cdsBy(), intronsByTranscript(), fiveUTRsByTranscript(), and threeUTRsByTranscript().

When dealing with genomic data it is almost inevitable that you will run into problems with the way that different groups have adopted alternate ways of naming chromosomes. This is because almost every major repository has cooked up their own slightly different way of labeling these important features.

To cope with this, the Seqinfo object was invented and is attached to TxDb objects as well as the GenomicRanges extracted from these objects. You can extract it using the seqinfo() method like this:

```
si <- seqinfo(txdb)
si
## Seqinfo object with 15 sequences (1 circular) from dm3
genome:
##  seqnames seqlengths isCircular genome
##  chr2L    23011544   FALSE  dm3
##  chr2R    21146708   FALSE  dm3
##  chr3L    24543557   FALSE  dm3
##  chr3R    27905053   FALSE  dm3
##  chr4     1351857   FALSE  dm3
##  ...       ...    ...  ...
##  chr3LHet   2555491   FALSE  dm3
##  chr3RHet   2517507   FALSE  dm3
##  chrXHet    204112   FALSE  dm3
##  chrYHet    347038   FALSE  dm3
##  chrUextra  29004656   FALSE  dm3
```

And since the seqinfo information is also attached to the GRanges objects produced by the TxDb extractors, you can also call seqinfo on the results of those methods like this:

```
txby <- transcriptsBy(txdb, by="gene")
si <- seqinfo(txby)
```

The Seqinfo object contains a lot of valuable data about which chromosome features are present, whether they are circular or linear, and how long each one is. It is also something that will be

checked against if you try to do an operation like "findOverlaps" to compute overlapping ranges etc. So it is a valuable way to make sure that the chromosomes and genome are the same for your annotations as the range that you are comparing them to. But sometimes you may have a situation where your annotation object contains data that is comparable to your data object, but where it is simply named with a different naming style. For those cases, there are helpers that you can use to discover what the current name style is for an object. And there is also a setter method to allow you to change the value to something more appropriate. So in the following example, we are going to change the seqlevelStyle from "UCSC" to "ensembl" based naming convention (and then back again).

```
seqlevels(txdb)
## [1] "chr2L"   "chr2R"   "chr3L"   "chr3R"   "chr4"
## [6] "chrX"   "chrU"   "chrM"   "chr2LHet" "chr2RHet"
## [11] "chr3LHet"   "chr3RHet"   "chrXHet"    "chrYHet"
"chrUextra"
seqlevelsStyle(txdb)
## [1] "UCSC"
seqlevelsStyle(txdb) <- "ensembl"
seqlevels(txdb)
## [1] "2L"            "2R"
## [3] "3L"            "3R"
## [5] "4"             "X"
## [7] "U"             "dmel_mitochondrion_genome"
## [9] "2LHet"           "2RHet"
## [11] "3LHet"           "3RHet"
## [13] "XHet"           "YHet"
## [15] "Uextra"
## then change it back
seqlevelsStyle(txdb) <- "UCSC"
seqlevels(txdb)
## [1] "chr2L"   "chr2R"   "chr3L"   "chr3R"   "chr4"
## [6] "chrX"   "chrU"   "chrM"   "chr2LHet" "chr2RHet"
## [11] "chr3LHet"   "chr3RHet"   "chrXHet"    "chrYHet"
"chrUextra"
```

In addition to being able to change the naming style used for an object with seqinfo data, you can also toggle which of the chromosomes are "active" so that the software will ignore certain chromosomes. By default, all of the chromosomes are set to be "active".

```
isActiveSeq(txdb)
##   chr2L   chr2R   chr3L   chr3R   chr4   chrX   chrU
##    TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE
```

```
##    chrM chr2LHet chr2RHet chr3LHet chr3RHet  chrXHet
chrYHet
##   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE
## chrUextra
##   TRUE
```

But sometimes you might wish to ignore some of them. For example, let us suppose that you wanted to ignore the Uextra chromosome from our fly txdb. You could do that like so:

```
isActiveSeq(txdb)["chrUextra"] <- FALSE
isActiveSeq(txdb)
##   chr2L   chr2R   chr3L   chr3R   chr4   chrX   chrU
##   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE
##    chrM chr2LHet chr2RHet chr3LHet chr3RHet  chrXHet
chrYHet
##   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE
## chrUextra
##   FALSE
```

3.4 OrganismDb Objects

So what happens if you have data from multiple different Annotation objects. For example, what if you had gene SYMBOLS (found in an OrgDb object) and you wanted to easily match those up with known gene transcript names from a UCSC based TxDb object? There is an ideal tool that can help with this kind of problem and it is called an OrganismDb object [9]. On the one hand OrganismDb objects can seem more complex than OrgDb, GODb or TxDb objects because they can allow you to access all three object sources at once. But in reality they are not that complicated. What they do is just query each of these resources for you and then merge the results back together in way that lets you pretend that you only have one source for all your annotations.

To try one out let us load one that was made as an annotation package:

```
library("Rattus.norvegicus")
## Loading required package: OrganismDbi
## Loading required package: GO.db
## Loading required package: DBI
##
## Loading required package: org.Rn.eg.db
##
## Loading required package: TxDb.Rnorvegicus.UCSC.rn5.
refGene
Rattus.norvegicus
## class: OrganismDb
## Annotation resources:
## [1] "GO.db"              "org.Rn.eg.db"
```

```
## [3] "TxDb.Rnorvegicus.UCSC.rn5.refGene"
## Annotation relationships:
##   xDbs     yDbs                xKeys
## [1,] "GO.db"    "org.Rn.eg.db"           "GOID"
##   [2,]   "org.Rn.eg.db"   "TxDb.Rnorvegicus.UCSC.rn5.
refGene" "ENTREZID"
##   yKeys
## [1,] "GO"
## [2,] "GENEID"
## For more details, please see the show methods for the
component objects listed above.
```

And that is it. You can now use these objects in the same way that you use OrgDb or TxDb objects. It works the same as the base objects that it contains:

```
select(Rattus.norvegicus, keys="24152", columns=c("SYM-
BOL","TXNAME"), keytype= "ENTREZID")
```

```
##  ENTREZID SYMBOL  TXNAME
## 1  24152  Asip NM_052979
```

In fact the five methods that worked for all of the other Db objects that we have discussed (keytypes(), columns(), keys(), select (), and mapIds()) should all work for OrganismDb objects. And if the OrganismDb object was composed to include a TxDb, then the range based accessors should also work:

```
txs   <-   transcripts(Rattus.norvegicus,   columns=c
("TXNAME","SYMBOL"))
txs
```

```
## GRanges object with 18762 ranges and 2 metadata columns:
##             seqnames         ranges strand |
##               <Rle>      <IRanges>  <Rle> |
##    [1]          chr1   [388173, 401149]   + |
##    [2]          chr1   [391729, 401149]   + |
##    [3]          chr1   [688298, 696950]   + |
##    [4]          chr1   [3400376, 3428890]   + |
##    [5]          chr1   [3468471, 3478304]   + |
##    ...           ...          ...  ......
##  [18758]          chrX [153807459, 153839833]   - |
##  [18759]          chrX [154351133, 154467649]   - |
##  [18760]          chrX [154511679, 154518145]   - |
##  [18761] chrX_AABR06110762_random [119,   711]   + |
##  [18762] chrX_AABR06110835_random [5214,  10526]   + |
##          TXNAME     SYMBOL
##      <CharacterList> <CharacterList>
##    [1]  NM_001099460     Vom2r3
##    [2]  NM_001099457     Vom2r2
##    [3]  NM_001099462     Vom2r5
```

```
##    [4]  NM_001106217      Lrp11
##    [5]  NM_001128191      Nup43
##    ...       ...       ...
##  [18758]  NM_001271327     Map7d3
##  [18759]  NM_001005565     Arhgef6
##  [18760]  NM_001025663       Rbmx
##  [18761]  NM_001037554       Bex4
##  [18762]  NM_001025288     Dmrtc1a
##  ------
##  seqinfo: 2739 sequences (1 circular) from rn5 genome
```

3.5 BSgenome Objects

Another important annotation resource type is a BSgenome package [10]. There are many BSgenome packages in the repository for you to choose from. And you can learn which organisms are already supported by using the available.genomes() function.

```
library("BSgenome")
```

```
## Loading required package: Biostrings
## Loading required package: XVector
## Loading required package: rtracklayer
```

```
head(available.genomes())
```

```
## [1] "BSgenome.Alyrata.JGI.v1"
## [2] "BSgenome.Amellifera.BeeBase.assembly4"
## [3] "BSgenome.Amellifera.UCSC.apiMel2"
## [4] "BSgenome.Amellifera.UCSC.apiMel2.masked"
## [5] "BSgenome.Athaliana.TAIR.04232008"
## [6] "BSgenome.Athaliana.TAIR.TAIR9"
```

Unlike the other resources that we have discussed here, these packages are meant to contain sequence data for a specific genome build of an organism. You load one of these packages in the usual way, and each of them normally has an alias for the primary object that is shorter than the full package name (as a convenience):

```
library("BSgenome.Dmelanogaster.UCSC.dm3")
```

```
ls(2)
```

```
## [1] "BSgenome.Dmelanogaster.UCSC.dm3" "Dmelanogaster"
```

```
Dmelanogaster
```

```
## Fly genome:
## # organism: Drosophila melanogaster (Fly)
## # provider: UCSC
## # provider version: dm3
## # release date: Apr. 2006
## # release name: BDGP Release 5
## # 15 sequences:
## #  chr2L   chr2R   chr3L   chr3R   chr4   chrX   chrU
## #  chrM   chr2LHet chr2RHet chr3LHet chr3RHet chrXHet
chrYHet
```

```
## #  chrUextra
## # (use 'seqnames()' to see all the sequence names, use the
'$' or '[['
## # operator to access a given sequence)
```

The getSeq method is a useful way of extracting data from these packages. This method takes several arguments but the important ones are the first two. The first argument specifies the BSgenome object to use and the second argument (names) specifies what data you want back out. So for example, if you call it and give a character vector that names the seqnames for the object then you will get the sequences from those chromosomes as a DNAStringSet object.

```
seqNms <- seqnames(Dmelanogaster)
head(seqNms)
```

```
## [1] "chr2L" "chr2R" "chr3L" "chr3R" "chr4"  "chrX"
```

```
getSeq(Dmelanogaster, seqNms[1:2])
```

```
##  A DNAStringSet instance of length 2
##     width seq
## [1] 23011544
CGACAATGCACGACAGAGGAAGCAGAACAG...TGCAAATTTTGAT-
GAACCCCCCT TTCAAA
## [2] 21146708
GACCCGCTAGGAGATGTTGAGATTGTGAGT...CAACGTGACTGTT TGCATTCTA
GGAATTC
```

Whereas if you give the a GRanges object for the second argument, you can instead get a DNAStringSet that corresponds to those ranges. This can be a powerful way to learn what sequence was present from a particular range. For example, here we can extract the range of a specific gene of interest from *Drosophila*.

```
txby <- transcriptsBy(txdb, by="gene")
geneOfInterest <- txby[["FBgn0000008"]]
res <- getSeq(Dmelanogaster, geneOfInterest)
res
```

```
##  A DNAStringSet instance of length 3
##    width seq
## [1] 35846
CGCGGCGGTCGCATCGGAGTCGAGAACTCGA...CATACAACCTTAATATATTTA
CCTGAAAAGC
## [2] 35851
CGGCGGTCGCATCGGAGTCGAGAACTCGAAG...CCTTAATATATTTACCTGAAA
AGCAATATAC
## [3] 35409
CGAATACACAAATCAAAGCAAGTGTCTGTGT...CCTTAATATATTTACCT GAAA
AGCAATATAC
```

Additionally, the Biostrings [11] package has many useful functions for finding a pattern in a string set etc. You may not have noticed when it happened, but the Biostrings package was loaded when you loaded the BSgenome object, so these functions will already be available for you to explore.

3.6 biomaRt

Another great annotation resource is the biomaRt package [5–7]. The biomaRt package exposes a huge family of different online annotation resources called marts. Each mart is another of a set of online Web resources that are following a convention that allows them to work with this package. So the first step in using biomaRt is always to load the package and then decide which "mart" you want to use. Once you have made your decision, you will then use the useMart() method to create a mart object in your R session. Here we are looking at the marts available and then choosing to use one of the most popular marts: the "ensembl" mart.

```
library("biomaRt")
head(listMarts())
```

```
##        biomart               version
## 1      ensembl    ENSEMBL GENES 79 (SANGER UK)
## 2        snp  ENSEMBL VARIATION 79 (SANGER UK)
## 3   regulation  ENSEMBL REGULATION 79 (SANGER UK)
## 4       vega        VEGA 59 (SANGER UK)
## 5   fungi_mart_26     ENSEMBL FUNGI 26 (EBI UK)
## 6 fungi_variations_26 ENSEMBL FUNGI VARIATION 26 (EBI UK)
```

```
ensembl <- useMart("ensembl")
ensembl
```

```
## Object of class 'Mart':
## Using the ensembl BioMart database
## Using the dataset
```

Each "mart" can contain datasets for multiple different things. So the next step is that you need to decide on a dataset. Once you have chosen one, you will need to specify that dataset using the dataset argument when you call the useMart() constructor method. Here we will point to the dataset for chicken.

```
head(listDatasets(ensembl))
```

```
##             dataset
## 1    oanatinus_gene_ensembl
## 2    cporcellus_gene_ensembl
## 3    gaculeatus_gene_ensembl
## 4    lafricana_gene_ensembl
## 5 itridecemlineatus_gene_ensembl
## 6    choffmanni_gene_ensembl
##                description version
## 1  Ornithorhynchus anatinus genes (OANA5)  OANA5
```

```
## 2       Cavia porcellus genes (cavPor3) cavPor3
## 3   Gasterosteus aculeatus genes (BROADS1) BROADS1
## 4     Loxodonta africana genes (loxAfr3) loxAfr3
## 5 Ictidomys tridecemlineatus genes (spetri2) spetri2
## 6    Choloepus hoffmanni genes (choHof1) choHof1
ensembl <- useMart("ensembl",dataset="ggallus_gene_
ensembl")
ensembl
## Object of class 'Mart':
## Using the ensembl BioMart database
## Using the ggallus_gene_ensembl dataset
```

Next we need to think about attributes, values, and filters. Let us start with attributes. You can get a listing of the different kinds of attributes from biomaRt buy using the listAttributes method:

```
head(listAttributes(ensembl))
##           name      description
## 1    ensembl_gene_id    Ensembl Gene ID
## 2 ensembl_transcript_id Ensembl Transcript ID
## 3  ensembl_peptide_id  Ensembl Protein ID
## 4    ensembl_exon_id    Ensembl Exon ID
## 5      description      Description
## 6    chromosome_name    Chromosome Name
```

And you can see what the values for a particular attribute are by using the getBM method:

```
head(getBM(attributes="chromosome_name", mart=ensembl))
##  chromosome_name
## 1        1
## 2       10
## 3       11
## 4       12
## 5       13
## 6       14
```

Attributes are the things that you can have returned from biomaRt. They are analogous to what you get when you use the columns method with other objects.

In the biomaRt package, filters are things that can be used with values to restrict or choose what comes back. The "values" here are treated as keys that you are passing in and which you would like to know more information about. In contrast, the filter represents the kind of key that you are searching for. So for example, you might choose a filter name of "chromosome_name" to go with specific value of "1". Together these two argument values would request whatever attributes matched things on the first chromosome. And just as there here is an accessor for attributes, there is also an accessor for filters:

```
head(listFilters(ensembl))
##              name   description
## 1 chromosome_name Chromosome name
## 2      start Gene Start (bp)
## 3       end  Gene End (bp)
## 4  marker_start   Marker Start
## 5   marker_end   Marker End
## 6     name_2    Name 2033
```

So now you know about attributes, values, and filters, you can call the getBM method to put it all together and request specific data from the mart. So for example, the following requests gene symbols and entrez gene IDs that are found on chromosome 1 of flies:

```
res <- getBM(attributes=c("hgnc_symbol", "entrezgene"),
          filters = "chromosome_name",
          values = "1", mart = ensembl)
head(res)
##  hgnc_symbol entrezgene
## 1         425783
## 2         430443
## 3   GOLGB1     NA
## 4         426867
## 5    NME6   426866
## 6         426865
```

Of course you may have noticed that a lot of the arguments for getBM are very similar to what you do when you call select. So if it is your preference you can now also use the standard select methods with mart objects.

```
head(columns(ensembl))
## [1] "ensembl_gene_id"    "ensembl_transcript_id" "ensembl_peptide_id"
## [4] "ensembl_exon_id"    "description"      "chromosome_name"
```

3.7 Creating Annotation Objects

By now you are aware that Bioconductor has a lot of annotation resources. But it is still completely impossible to have every annotation resource prepackaged for every conceivable use. Because of this, almost all annotation objects have special functions that can be called to create those objects (or the packages that load them) from generalized data resources or specific file types. Below is a table with a few of the more popular options (Table 2).

In most cases the output for resource creation functions will be an annotation package that you can install.

And there is unfortunately not enough space to demonstrate how to call each of these functions here. But to do so is actually pretty straightforward and most such functions will be well documented with their associated manual pages and vignettes [3, 4, 10, 12]. As usual, you can see the help page for any function right inside of R.

Table 2
How to create popular Annotation Objects

If you want this	And you have this	Then you could call this to help
TxDb	Tracks from UCSC	GenomicFeatures:: makeTxDbPackageFromUCSC
TxDb	Data from biomaRt	GenomicFeatures:: makeTxDbPackageFromBiomaRt
TxDb	gff or gtf file	GenomicFeatures:: makeTxDbFromGFF
OrgDb	Custom data.frames	AnnotationForge::makeOrgPackage
OrgDb	Valid Taxonomy ID	AnnotationForge:: makeOrgPackageFromNCBI
ChipDb	org package and data. frame	AnnotationForge:: makeChipPackage
BSgenome	fasta or twobit sequence files	BSgenome:: forgeBSgenomeDataPkg

```
help("makeTxDbPackageFromUCSC")
```

If you plan to make use of these kinds of functions then you should expect to consult the associated documentation first. These kinds of functions tend to have a lot of arguments and most of them also require that their input data meet some fairly specific criteria. Finally, you should know that even after you have succeeded at creating an annotation package, you will also have to make use of the install.packages() function (with the repos argument=NULL) to install whatever package source directory has just been created.

4 Notes

The bioconductor project represents a very large and active codebase from an active and engaged community. Because of this, you should expect that the software described in this chapter will change over time and often in dramatic ways. As an example, the getSeq function that is described in this chapter is expected to a big overhaul in the coming months. When this happens the older function will be deprecated for a full release cycle (6 months) and then labeled as defunct for another release cycle before it is removed. This cycle is in place so that active users can be warned about what is happening and where they should look for the appropriate replacement functionality. But obviously, this system cannot warn end users if they have not been vigilant about updating

their software to the latest version. So please take the time to always update your software to the latest version.

In order to stay abreast of new developments users are also encouraged to explore the bioconductor website which contains many current walkthroughs and vignettes (http://bioconductor.org/), and also to ask questions on the forums (https://support.bioconductor.org/).

Acknowledgments

Research reported in this chapter was supported by the National Human Genome Research Institute of the National Institutes of Health under Award Number U41HG004059 and by the National Cancer Institute of the National Institutes of Health under Award Number U24CA180996. We also want to thank the numerous institutions who produced and maintained the data that are used for generating and updating the annotation resources described here.

References

1. Huber W, Carey VJ, Gentleman R, Anders S, Carlson M, Carvalho BS, Bravo HC, Davis S, Gatto L, Girke T, Gottardo R, Hahne F, Hansen KD, Irizarry RA, Lawrence M, Love MI, MacDonald J, Obenchain V, Oleś AK, Pagès H, Reyes A, Shannon P, Smyth GK, Tenenbaum D, Waldron L, Morgan M (2015) Orchestrating high-throughput genomic analysis with Bioconductor. Nat Methods 12:115–121
2. Pages H, Carlson M, Falcon S, Li N. AnnotationDbi: annotation database interface. R package version 1.30.0. http://bioconductor.org/packages/AnnotationDbi/. Accessed May 2015
3. Carlson M, Pages H, Aboyoun P, Falcon S, Morgan M, Sarkar D, Lawrence M. GenomicFeatures: tools for making and manipulating transcript centric annotations R package version 1.19.38. http://bioconductor.org/packages/GenomicFeatures/. Accessed May 2015
4. Lawrence M, Huber W, Pagès H, Aboyoun P, Carlson M, Gentleman R, Morgan M, Carey V (2013) Software for computing and annotating genomic ranges. PLoS Comput Biol 9, http://dx.doi.org/10.1371/journal.pcbi.1003118http://www.ploscompbiol.org/article/info%3Adoi%2F10.1371%2Fjournal.pcbi.1003118
5. Durinck S, Huber W. biomaRt: interface to BioMart databases (e.g. Ensembl, COSMIC, Wormbase and Gramene) R package version 2.23.5. http://bioconductor.org/packages/biomaRt/. Accessed May 2015
6. Durinck S, Spellman P, Birney E, Huber W (2009) Mapping identifiers for the integration of genomic datasets with the R/Bioconductor package biomaRt. Nat Protoc 4:1184–1191
7. Durinck S, Moreau Y, Kasprzyk A, Davis S, De Moor B, Brazma A, Huber W (2005) BioMart and Bioconductor: a powerful link between biological databases and microarray data analysis. Bioinformatics 21:3439–3440
8. Morgan M, Carlson M, Tenenbaum D, Arora S. AnnotationHub: client to access AnnotationHub resources. R package version 2.0.1. http://bioconductor.org/packages/AnnotationHub/. Accessed May 2015
9. Carlson M, Pages H, Morgan M, Obenchain V. OrganismDbi: software to enable the smooth interfacing of different database packages. R package version 1.10.0. http://bioconductor.org/packages/OrganismDbi/. Accessed May 2015
10. Pages H. BSgenome: infrastructure for Biostrings-based genome data packages. R package version 1.36.0. http://bioconductor.org/packages/BSgenome/. Accessed May 2015

11. Pages H, Aboyoun P, Gentleman R, DebRoy S. Biostrings: string objects representing biological sequences, and matching algorithms. R package version 2.36.0. http://bioconductor.org/packages/Biostrings/. Accessed May 2015

12. Carlson M, Pages H. AnnotationForge: code for building annotation database packages. R package version 1.10.0. http://bioconductor.org/packages/AnnotationForge/. Accessed May 2015