# R and GRanges

A genomic range is a span of coordinates on a chromosome, e.g. chr4:450000-500000

This might sound simplistic at first, but most computational genomics projects operate fundamentally on genomic ranges.

I downloaded the data from the GEO record, GSM1018048. This is a ChIPseq experiment for H3K4me3, in lung adenocarcinoma cells treated with TGFß. I've called peaks and have created the following files:
GSM1018048_H3K4me3_peaks.txt   (ChIPseq peaks)
hg19genes.txt
tgf_beta_genes.txt  (this is from MSigDB)

Note that the original alignments were to hg19 so I need to use the gene annotations on hg19 coordinates.

There are quite a few ways to construct Genomic Ranges objects. We will use a method that is not the quickest, but is probably the clearest.

First, make sure that you have the GenomicRanges library.

```
> library("GenomicRanges")
```

If this loads without error, then you already have the library. Otherwise, you'll need to fetch and install it from Bioconductor:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("GenomicRanges")
> library("GenomicRanges")
```

As with all Bioconductor packages, GenomicRanges comes with quite a bit of documentation. You can find it easily:

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

First, read in the gene and peak tables:

```
> genes <- read.table("hg19genes.txt", stringsAsFactors=F)
> peaks <- read.table("GSM1018048_H3K4me3_peaks.txt",
stringsAsFactors=F, header=T)
> tgfb <- read.table("tgf_beta_genes.txt", stringsAsFactors=F)
```

Now, make the Genomic Ranges. Genomic Ranges are a specialized type of IRanges, which hold simpler intervals (start and end), so one way to construct a GenomicRanges object is to construct an IRanges and add sequence names, strands, and other metadata to it.

```
> genesGR <- GRanges(seqnames=genes[,1], IRanges(start=genes[,2],
end=genes[,3]))
> peaksGR <- GRanges(seqnames=peaks[,1], IRanges(start=peaks[,2],
end=peaks[,3]))
```

Let's explore these objects:

```
> head(peaksGR)
GRanges object with 6 ranges and 0 metadata columns:
      seqnames           ranges strand
         <Rle>        <IRanges>  <Rle>
  [1]     chr1 [713524, 714105]      *
  [2]     chr1 [714266, 714838]      *
  [3]     chr1 [715085, 715339]      *
  [4]     chr1 [762068, 763287]      *
  [5]     chr1 [805016, 805303]      *
  [6]     chr1 [839439, 840603]      *
  -------
  seqinfo: 57 sequences from an unspecified genome; no seqlengths
```

```
> str(peaksGR)
Formal class 'GRanges' [package "GenomicRanges"] with 6 slots
  ..@ seqnames       :Formal class 'Rle' [package "S4Vectors"] with 4 slots
  .. .. ..@ values         : Factor w/ 57 levels "chr1","chr10",..: 1 2 3 4 5 6 7 8 9 10 ...
  .. .. ..@ lengths        : int [1:57] 3066 1659 1650 1 1936 795 1116 1239 1285 1837 ...
  .. .. ..@ elementMetadata: NULL
  .. .. ..@ metadata       : list()
  ..@ ranges         :Formal class 'IRanges' [package "IRanges"] with 6 slots
  .. .. ..@ start          : int [1:35255] 713524 714266 715085 762068 805016 839439 858928
859506 875945 877504 ...
  .. .. ..@ width          : int [1:35255] 582 573 255 1220 288 1165 328 2002 290 450 ...
  .. .. ..@ NAMES          : NULL
  .. .. ..@ elementType    : chr "integer"
  .. .. ..@ elementMetadata: NULL
  .. .. ..@ metadata       : list()
  ..@ strand         :Formal class 'Rle' [package "S4Vectors"] with 4 slots
  .. .. ..@ values         : Factor w/ 3 levels "+","-","*": 3
  .. .. ..@ lengths        : int 35255
  .. .. ..@ elementMetadata: NULL
  .. .. ..@ metadata       : list()
  ..@ elementMetadata:Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
  .. .. ..@ rownames       : NULL
  .. .. ..@ nrows          : int 35255
  .. .. ..@ listData       : Named list()
  .. .. ..@ elementType    : chr "ANY"
  .. .. ..@ elementMetadata: NULL
  .. .. ..@ metadata       : list()
  ..@ seqinfo        :Formal class 'Seqinfo' [package "GenomeInfoDb"] with 4 slots
  .. .. ..@ seqnames   : chr [1:57] "chr1" "chr10" "chr11" "chr11_gl000202_random" ...
  .. .. ..@ seqlengths : int [1:57] NA NA NA NA NA NA NA NA NA NA ...
  .. .. ..@ is_circular: logi [1:57] NA NA NA NA NA NA ...
  .. .. ..@ genome     : chr [1:57] NA NA NA NA ...
  ..@ metadata       : list()
```

How can we access these data???

```
>  peaksGR@seqnames
factor-Rle of length 35255 with 57 runs
  Lengths:                    3066                   1659 ...                     49
  Values :                    chr1                  chr10 ...                   chrY
Levels(57): chr1 chr10 chr11 chr11_gl000202_random ... chrUn_gl000241 chrX chrY
```

And then how do we get the starts . . .?

This is a more complex object, so it has useful accessory functions (most of which are described in the documentation):

```
> head(start(peaksGR))
[1] 713524 714266 715085 762068 805016 839439
```

Which genes are near ChIPseq peaks for H3K4me3, under TGFß stimulation?
We don't want to restrict this to the body of the gene . . .  let's remake the GRanges for the genes and expand the boundaries a little:

```
> genesGR <- GRanges(seqnames=genes[,1], IRanges(start=genes[,2]-5000, 
end=genes[,3]+5000))
```

Now we'll overlap the peaks with the genes:

```
> ovl <- findOverlaps(genesGR, peaksGR)
Warning message:
In .Seqinfo.mergexy(x, y) :
  Each of the 2 combined objects has sequence levels not in the other:
  - in 'x': chr17|NT_113930.1, chr17|NT_113943.1, chr19|NT_113949.1, chr1|NT_113878.1, 
chr1|NT_167207.1, chr4|NT_113885.1, chr4|NT_113888.1, chr7|NT_113901.1, chr8|
NT_113907.1, chr8|NT_113909.1, chr9|NT_113911.1, chr9|NT_113915.1, chrMT, chrUn|
NT_113889.1, chrUn|NT_113923.1, chrUn|NT_113961.1, chrUn|NT_167208.1, chrUn|
NT_167209.1, chrUn|NT_167210.1, chrUn|NT_167211.1, chrUn|NT_167212.1, chrUn|
NT_167213.1, chrUn|NT_167214.1, chrUn|NT_167215.1, chrUn|NT_167216.1, chrUn|
NT_167217.1, chrUn|NT_167218.1, chrUn|NT_167219.1, chrUn|NT_167221.1, chrUn|
NT_167222.1, chrUn|NT_167223.1, chrUn|NT_167224.1, chrUn|NT_167230.1, chrUn|
NT_167232.1, chrUn|NT_167235.1, chrUn|NT_167236.1, chrUn|NT_167237.1, chrUn|
NT_167238.1, chrUn|NT_167243.1
```

```
  - in 'y': chr11_gl000202_random, chr17_ctg5_hap1, chr17_gl000204_random,
chr17_gl000205_random, chr19_gl000208_random, chr1_gl000191_random,
chr1_gl000192_random, chr4_gl000194_random, chr6_cox_hap2 [... truncated]
```

Yikes! what happened??
The peaks and genes are computed and annotated over the standard chromosomes as well as a bunch of unplaced contigs, and they have annotations on unplaced contigs that are present in one dataset and not the other.

When you get these messages you should evaluate them and decide whether it's important (for example, if you see that chr20, chr21, and chr22 are in one dataset and not the other, you should ask yourself whether you are accidentally comparing mouse data to human data!)

Here, we'll ignore the message.

What sort of object is this?

```
> str(ovl)
Formal class 'SortedByQueryHits' [package "S4Vectors"] with 6 slots
  ..@ from           : int [1:35751] 26 26 26 29 30 31 32 33 35 35 ...
  ..@ to             : int [1:35751] 1 2 3 4 4 4 5 5 7 8 ...
  ..@ nLnode         : int 39024
  ..@ nRnode         : int 35255
  ..@ elementMetadata: NULL
  ..@ metadata       : list()
```

```
> ?Hits
```

this means that the elements in the first GRanges (genes) are in the "from" slot, and these overlap the elements in the second GRanges (peaks), in the "to" slot.

| | |
|---|---|
| genes[26,] | peaks[1,] |
| genes[26,] | peaks[2,] |
| genes[26,] | peaks[3,] |
| genes[29,] | peaks[4,] |
| genes[30,] | peaks[4,] |

Exercise:
How many genes did we load? Do this in three ways: find a GenomicRanges function that tests the genesGR object, look at the original table (genes), that you loaded in R, and use one of the unix utilities on the original file.

How many peaks did we load?

How many genes overlap peaks? how many peaks overlap genes? why is there a discrepancy between those two numbers?

Remember that you can subset an R object by rows, columns, or logical statements.

The `ovl` object that we made is perfect for subsetting. We could subset either the `genes` object or the `genesGR` GenomicRanges object, but we didn't load the gene names into the `genesGR` metadata, so we'll use the genes object.

```
> sel <- unique(from(ovl))
> genesovl <- genes[sel,]
```

We never gave the `genes` object column names. The last column is the gene name.

How many of these genes are TGFß targets?

Look at the object that we made when we read in the TGFß targets. What if you can't remember the name of this object?

```
> ls()
> head(tgfb)
```

Notice that `tgfb` is a data frame (with only one colum), as is `genes`.
A very simple operator will tell us which elements in one vector match elements in another:

```
> testvec <- c("APC", “AAA")
> gene <- "APC"
> gene%in%testvec
[1] TRUE
> testvec%in%tgfb[,1]
[1]  TRUE FALSE
```

Now we can easily count how many items in the first vector match elements in the second vector:

```
> sum(testvec%in%tgfb[,1])
[1] 1
```

We can also subset the first vector into items that only match elements in the second vector, or only items that do not match elements in the second vector:

```
> sel <- testvec%in%tgfb[,1]
> testvec[sel]
[1] "APC"
> testvec[!sel]
```

```
[1] "AAA"
```

Exercise:
Subset the genes object into two new objects: `genes_tgfb` (genes that are in the tgfb list) and `genes_not_tgfb` (genes that are not in the tgfb list)

Make sure these two objects add up:

```
> head(genes_tgfb)
       V1        V2        V3 V4     V5
101  chr1   2160134   2241652  +    SKI
497  chr1  23884409  23886322  -    ID3
688  chr1  32757708  32799224  +  HDAC1
1406 chr1  94027348  94147385  -  BCAR3
1640 chr1 114935399 115053781  - TRIM33
3206 chr1 226124298 226129083  - LEFTY2
> head(genes_not_tgfb)
    V1    V2    V3 V4           V5
1 chr1 10954 11507  + LOC100506145
2 chr1 12613 14361  +      DDX11L4
3 chr1 14362 29370  -       WASH5P
4 chr1 30366 30503  +   MIR1302-10
5 chr1 34611 36081  -      FAM138A
6 chr1 52453 53396  +       OR4G4P
> dim(genes_tgfb)
[1] 56  5
> dim(genes_not_tgfb)
[1] 38968     5
> dim(genes)
[1] 39024     5
```

Exercise:
Use findOverlaps to create two new objects: `ovl_tgfb` and `ovl_not_tgfb`, where the first object comes from overlapping `genes_tgfb` with the peaks and the second comes from overlapping `genes_not_tgfb` with peaks. You will have to make Genomic Ranges out of these two tables first.

Take a look at your new objects:

```
> head(ovl_tgfb)
Hits object with 6 hits and 0 metadata columns:
      queryHits subjectHits
      <integer>   <integer>
  [1]         1          78
  [2]         1          79
  [3]         2         440
  [4]         2         441
  [5]         3         634
  [6]         3         635
  -------
  queryLength: 56 / subjectLength: 35255
> head(ovl_not_tgfb)
Hits object with 6 hits and 0 metadata columns:
      queryHits subjectHits
      <integer>   <integer>
  [1]        26           1
  [2]        29           4
  [3]        30           4
  [4]        32           5
  [5]        36           8
  [6]        36           9
  -------
  queryLength: 38968 / subjectLength: 35255
```

Exercise:
How many genes from the genes_tgfb ranges overlap peaks? from the genes_not_tgfb ranges? How long was each to begin with?

It looks like there might be enrichment in the `genes_tgfb` set, for overlap with H3K4me3 peaks under TGFß stimulation, versus genes in the `genes_not_tgfb` set.

Now we've created two gene sets! and we can do a competitive gene set analysis, using a Fisher's exact test (because we don't have very many observations).

| | in TGFß gene set | not in TGFß gene set |
|---|---|---|
| overlap | 53 | 16350 |
| do not overlap | 3 | 22618 |

How does this work . . . ?

```
> ?Fisher
No documentation for ‘Fisher’ in specified packages and libraries:
you could try ‘??Fisher’
> ??Fisher
> mat <- matrix(c(53,3,16350, 22618), nrow=2)
> fisher.test(mat)

	Fisher's Exact Test for Count Data

data:  mat
p-value < 2.2e-16
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
   7.918697 121.905976
sample estimates:
odds ratio
  24.43809
```

(I did not engineer this one! these are real data)

Like almost all programming languages, R supports loops and familiar flow control operators, though the syntax is slightly different from Python and shell code:

```
> tab <- read.table("primerblast.txt", header=F)
> counter <- 0
> for (i in 1:length(tab[,1])) {
     if (tab[i,3] > 90 & tab[i,4] == 23) {
           counter = counter+1 }}
```

However, loops like this are generally not the most efficient way to deal with data in R.
Exercise:
Write a one-line subsetting command that produces a data frame from the primerblast.txt data, where the third column (percent identity) is greater than 90 and the fourth column (length) is 23.