Bioconductor
Course by James Chapman, Paula Martinez, and DataCamp

Bioconductor’s tagline is “open source software for bioinformatics”

Installing Bioconductor
```
install.packages(“BiocManager”)
BiocManager::install(“GenomicRanges”)
```

```
#load BiocManager
library(BiocManager)
#frequent updates > make sure to have the most up to date version
version()
#load package
library(GenomicRanges)
#check the version of your package
sessionInfo()
#to check for package updates
valid()
```

The Role of S4 in Bioconductor
S4 is a special type of system within R
S4 implements an object-oriented style of programming
*the basic idea here is to define the data then to work on it
once an object is defined, it is generalized to a class by defining the kind of data it contains and any actions or functions to manipulate it
biological representations are complex and interconnected > *this is why S4 links up so well with bioinformatics
Bioconductor recommends re-using methods and classes before implementing new representations
S4 classes have a formal definition and inheritance (making them better to check input types)
We can create a new object from a class
*Creating a new class from a pre-existing object allow for the inheritance of certain attributes
concept example > similar to how children can inherit hair color from their parents

Is an object S4?
```
isS4(object_name)
#or
str(object_name)
```

output > ‘Formal class’ (this represents that the object is S4)

S4 class definition > describes a representation of an object with a name and slots (also called methods or fields)
these are helpful for validation
a class optionally describes its inheritance using the parameter ‘contains’
a class allows us to define all the characteristics concerning an object
*we can then reuse when creating new objects
we create a class using setClass
example

```
MyEpicProject <- setClass(‘MyEpicProject’,
                          slots = c(ini = ‘Date’, #define slots which are helpful for validation
                                    end = ‘Date’,
                                    milestone = ‘character’),
                          contains = ‘MyProject’) #define inheritance
#this class inherits from teh class MyProject meaning we can reuse slots from MyProject
```

S4 accessors
accessor-functions also called methods > can give us some basic info on our S4 objects
*a class definition includes slots for describing an object

```
#to get a summary of the accessors of a main class
.S4methods(class = ‘GenomeDescription’)
#for subclasses
showMethods(classes = ‘GenomeDescription’, where = search())
#object summary
show(object_name)
```

example
showClass(“BSgenome”)
gives slots
also gives parent classes listed as “Extends”
also gives classes that inherit from it as “Subclasses”

example

```
show(a_genome)
Yeast genome:
# organism: Saccharomyces cerevisiae (Yeast)
# genome: sacCer3
# provider: UCSC
# release date: April 2011
# 17 sequences:
#   chrI    chrII   chrIII  chrIV   chrV    chrVI   chrVII  chrVIII chrIX
#   chrX    chrXI   chrXII  chrXIII chrXIV  chrXV   chrXVI  chrM
# (use 'seqnames()' to see all the sequence names, use the '$' or '[[' operator
# to access a given sequence)

#Investigating specific accessors
organism(a_genome)
[1] "Saccharomyces cerevisiae"
```

Biology of genomic datasets
bioinformatics > organisms are studied by sequencing genomes and dissecting its elements to find interesting functions
a genome is the complete genetic material of an organism stored mostly in the chromosomes
made of long sequences of DNA (TAGC)
the written information in a genome uses the DNA alphabet
can think of a genome as a set of books and each book is a chromosome
each chromosome has ordered genetic sequences (think of as chapters within a book)
genes are like the pages in a book containing a recipe to make proteins (coding and non-coding genes)
coding genes are expressed through proteins responsible for specific functions
Proteins come up following a two-step process:

- transcription > DNA to RNA
- translation > RNA to protein

For practice we are going to use

```
library(BSgenome.Scerevisisae. UCSC.sacCer3)
yeast <- BSgenome.Scerevisisae.UCSC.sacCer3
```

A whole bunch of available genomes for practice

```
#to find
available.genomes()
```

available accessor functions

```
length(yeast) #number of chromosomes
names(yeast) #names of the chromosomes
seqlengths(yeast) # length of each chromosome by DNA base pairs
```

Getting sequences

```
#select entire genomic sequence
getSeq(yeast)
```

#select sequence from chromosome M
getSeq(yeast, “chrM”)
#select first 10 base pairs
getSeq(yeast, end =10) #addition arguments > ‘start’ and ‘width’

Example
# Load the yeast genome
library(BSgenome.Scerevisiae.UCSC.sacCer3)

# Assign data to the yeastGenome object
yeastGenome <- BSgenome.Scerevisiae.UCSC.sacCer3

# Get the head of seqnames and tail of seqlengths for yeastGenome
head(seqnames(yeastGenome))
tail(seqlengths(yeastGenome))

# Print chromosome M, alias chrM
getSeq(yeastGenome, 'chrM')

# Count characters of the chrM sequence
nchar(getSeq(yeastGenome, 'chrM'))

Biostrings
critical package for Bioconductor
implements algorithms for fast manipulation of large biological sequences

to install:
BiocManager::install(“Biostrings”)

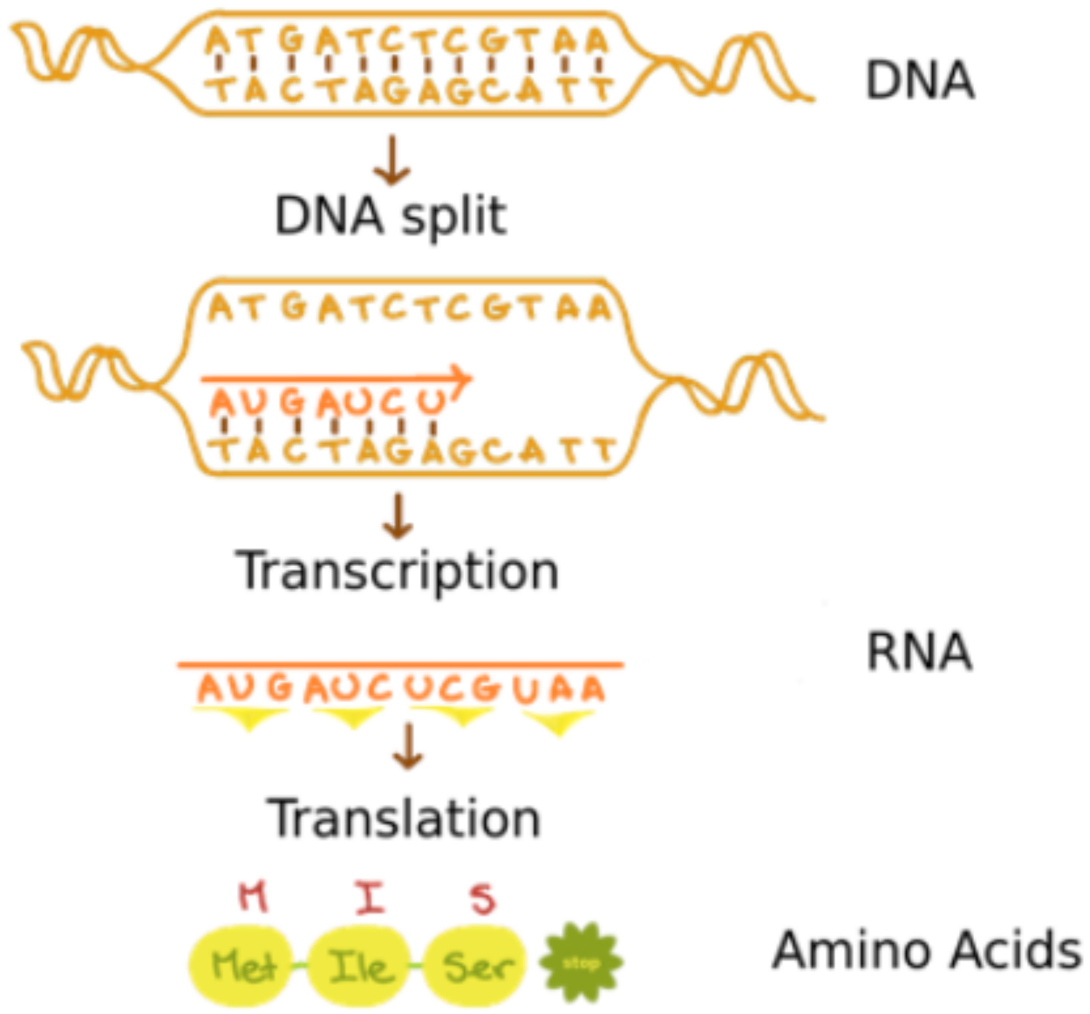
*key component to Biostrings is its containers
implements memory efficient containers > great for subsetting and matching
*these containers can have subclasses
example
BString (short fo Big String) subclass can store a big sequence of strings

Biostrings implements two generic containers from which other subclasses will inherit > XString and XStringSet

1. XString for a single sesequence of a predefined alphabet (example DNAString, RNAString, AAString (for amino acids)
2. XStringSet for many sequences even of varying lengths (example BStringSet, DNAStringSet, …)

to see inner workings:
showClass(‘XString’)

Biostring alphabets
DNA_BASES #DNA sequence
output > A C G T
RNA_BASES #RNA sequence
output > A C G U
AA_STANDARD #the 20 amino acids, each is built from 3 consecutive RNA bases
output > A R N D C Q E G H I L K M F P S T W Y V
Biostring alphabets are based on two code representations
IUPAC_CODE_MAP and AMINO_ACID_CODE
each contains the bases plus extra characters and symbols
DNA_ALPHABET #contains IUPAC_CODE_MAP
RNA_ALPHABET #contains IUPAC_CODE_MAP
AA_ALPHABET #contains AMINO_ACID_CODE



Example transcription DNA to RNA with single string sequence

```
dna_seq <- DNAString(“ATGATCTCGTAA”)
#transcription DNA to RNA string
#T’s to U’s
rna_seq <- RNAString(dna_seq)
rna_seq
output>
AUGAUCUCGUAA
#can also do this with a set
#translation RNA to AA
aa_seq <- translate(rna_seq)
#three RNA bases form one AA > this example AUG = M, AUC = I, UCG = S, UAA =
*
aa_seq
output > MIS*
```

**with coding can skip right from dna to aa with translate()

```
translate(dna_seq)
output > MIS*
```

Example

```
# Load packages
library(Biostrings)

# Check the alphabet of the zikaVirus
alphabet(zikaVirus)

# Check the alphabetFrequency of the zikaVirus > gives us letter counts
alphabetFrequency(zikaVirus)

# Check alphabet of the zikaVirus using baseOnly = TRUE > gives us the base
alphabet(zikaVirus, baseOnly = TRUE)
```

Example

```
# Unlist the set, select the first 21 letters, and assign to dna_seq
dna_seq <- subseq(unlist(zikaVirus), end = 21)
dna_seq

# Transcribe dna_seq into an RNAString object and print it
rna_seq <- RNAString(dna_seq)
rna_seq

# Translate rna_seq into an AAString object and print it
```

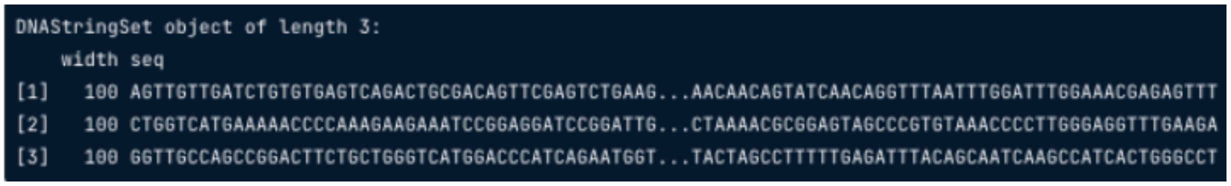
aa_seq <- translate(rna_seq)
aa_seq

# Unlist the set, select the first 21 letters, and assign to dna_seq
dna_seq <- subseq(unlist(zikaVirus), end = 21)
dna_seq

# Transcribe and translate dna_seq into an AAString object and print it
aa_seq <- translate(dna_seq)
aa_seq

Creating a StringSet and collating it
zikaVirus <- readDNAStringSet(‘data/zika.fa’)
#collate the sequence
zikaVirus_seq <- unlist(zikaVirus)
#DNAStrings do not have widths only lengths
#length represents the amount of characters within the DNAString

From a single sequence to a set
zikaSet <- DNAStringSet(zikaVirus_seq, start = c(1, 101, 201), end = c(100, 200, 300))



```
DNAStringSet object of length 3:
    width seq
[1]   100 AGTTGTTGATCTGTGTGAGTCAGACTGCGACAGTTCGAGTCTGAAG...AACAACAGTATCAACAGGTTTAATTTGGATTTGGAAACGAGAGTTT
[2]   100 CTGGTCATGAAAAACCCCAAAGAAGAAATCCGGAGGATCCGGATTG...CTAAAACGCGGAGTAGCCCGTGTAAACCCCTTGGGAGGTTTGAAGA
[3]   100 GGTTGCCAGCCGGACTTCTGCTGGGTCATGGACCCATCAGAATGGT...TACTAGCCTTTTTGAGATTTACAGCAATCAAGCCATCACTGGGCCT
```

now if we check width and length
length(zikaSet)
width(zikaSet)
output > 3
output > 100 100 100

**we only need to work with one DNA sequence because the second strand is ‘complement’
letters are paired
if we need the complement
complement(a_seq)

We can rev a sequence > making 1 2 and 2 1
this is useful when building a genome reference
example

```
zikaShortSet
```

```
DNAStringSet instance of length 2
width seq                          names
[1]    18 AGTTGTTGATCTGTGTGA        seq1
[2]    18 CTGGTCATGAAAAACCCC        seq2
```

```
rev(zikaShortSet)
```

```
 A DNAStringSet instance of length 2
width seq                          names
[1]    18 CTGGTCATGAAAAACCCC        seq2
[2]    18 AGTTGTTGATCTGTGTGA        seq1
```

We can reverse a sequence
reverse()
reverses each sequence in the set from right to left

Can get the complement and reverse it in one function
reverseComplement(rna_seq) #can be used for RNAStrings and DNAStrings

Example
# Create zikv with one collated sequence using zikaVirus
zikv <- unlist(zikaVirus)

# Check the length of zikaVirus and zikv
length(zikaVirus)
length(zikv)

# Check the width of zikaVirus
width(zikaVirus)

# Subset zikv to only the first 30 bases
subZikv <- subseq(zikv, end = 30)
subZikv

The goal of analyzing sequence patterns
-find sequence repeats
-frequency of proteins and codons
-poly-A tails
-conserved sequences
-binding sites
-and to discover occurrence frequency, periodicity, and length

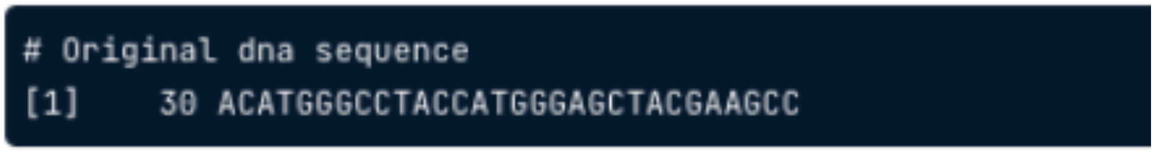
Common questions solved by sequence pattern matching?

- where a gene starts
- where a protein ends
- regions that enhance or silence gene expression
- conserved regions between organisms
- overall genetic variation

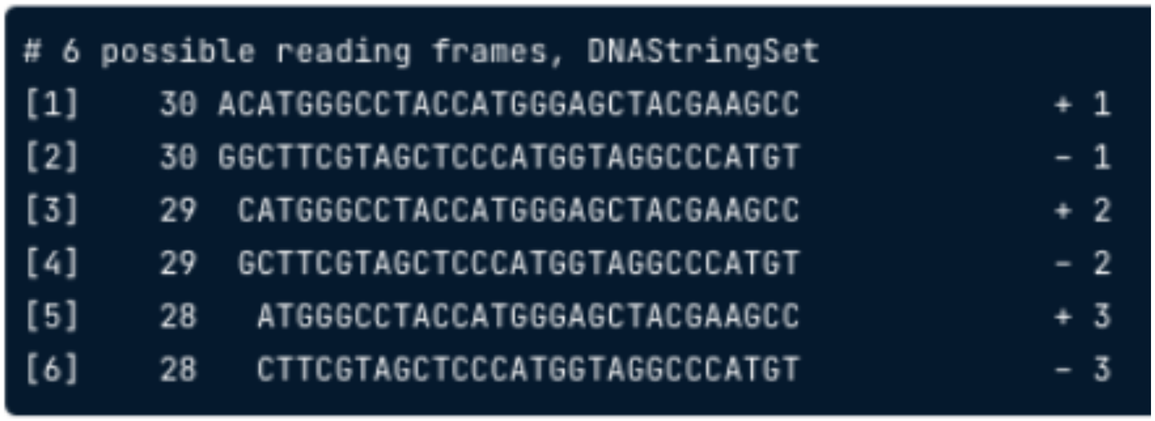
Using Biostrings
matchPattern(pattern, subject) #1 string to 1 string
#pattern tends to be a short sequence and the subject a longer sequence
vmatchPatter(pattern, subject) #for multiple sequences (1 set of strings to 1 string or 1 string to a set of strings)

Palindromes
not just a funny language thing
in biology, palindromes occur at sites highlighting binding sites and sites interrupted by restriction enzymes
with R and Biostrings:
findPalindromes() #will find palindromic regions in a single sequence

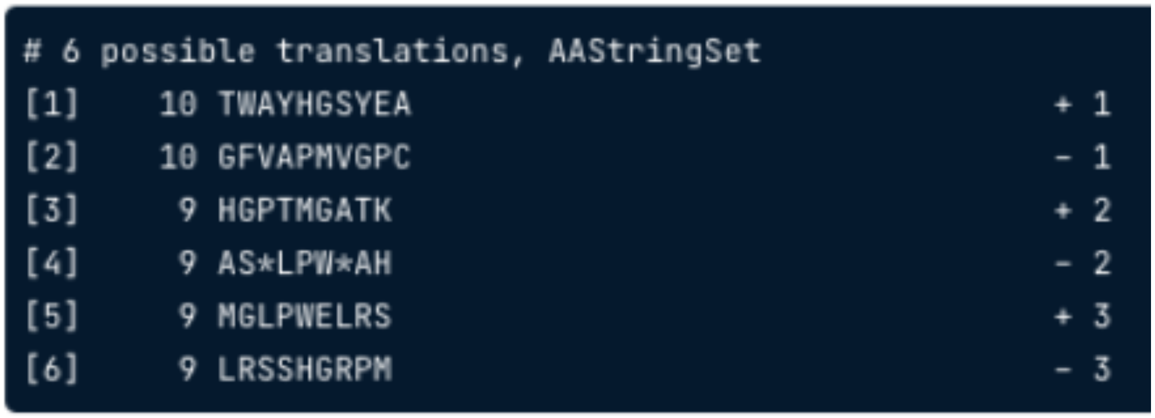
Different sequences are translated depending on the start point
*Translation has six possibilities



```
# Original dna sequence
[1]    30 ACATGGGCCTACCATGGGAGCTACGAAGCC
```



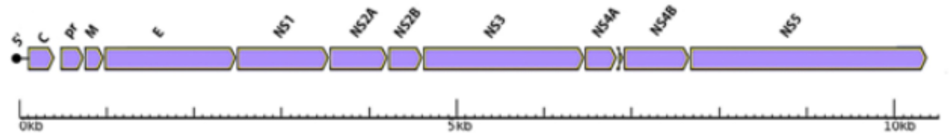
```
# 6 possible reading frames, DNAStringSet
[1]    30 ACATGGGCCTACCATGGGAGCTACGAAGCC                 + 1
[2]    30 GGCTTCGTAGCTCCCATGGTAGGCCCATGT                 - 1
[3]    29  CATGGGCCTACCATGGGAGCTACGAAGCC                 + 2
[4]    29  GCTTCGTAGCTCCCATGGTAGGCCCATGT                 - 2
[5]    28   ATGGGCCTACCATGGGAGCTACGAAGCC                 + 3
[6]    28   CTTCGTAGCTCCCATGGTAGGCCCATGT                 - 3
```



```
# 6 possible translations, AAStringSet
[1]    10 TWAYHGSYEA                                     + 1
[2]    10 GFVAPMVGPC                                     - 1
[3]     9 HGPTMGATK                                      + 2
[4]     9 AS*LPW*AH                                      - 2
[5]     9 MGLPWELRS                                      + 3
[6]     9 LRSSHGRPM                                      - 3
```

*translation varies according to the start of the sequence
from a DNA string there are 6 possible string frames (3 positive, 3 negative)
a negative strand is the reverse complement of a positive sequence strand
translation needs three bases for an amino acid > meaning you get a completely different AA sequence depending on where you start
*that is why for translation, we move one base at a time > this is called single base sliding window

Conserved regions in the Zika virus
zika has a positive strand genome
has a very conserved sequence in the family of Flaviiviruses
can live in different host cells
virus structure has only 11 proteins



Adapted figure From Mosquitos to Humans: Genetic Evolution of Zika Virus Wang, Lulan et al. Cell Host & Microbe 2016, Vol 19 5: 561-565

Example
# Print rnaframesZikaSet
rnaframesZikaSet

# Translate rnaframesZikaSet
AAzika6F <- translate(rnaframesZikaSet)
AAzika6F

# Count NS5 protein matches in AAzika6F, allowing 15 mismatches
vcountPattern(pattern = NS5, subject = AAzika6F, max.mismatch = 15)

# Subset the frame that contains the match from AAzika6F
selectedSet <- AAzika6F[3]

# Convert selectedSet into a single sequence
selectedSeq <- unlist(selectedSet)

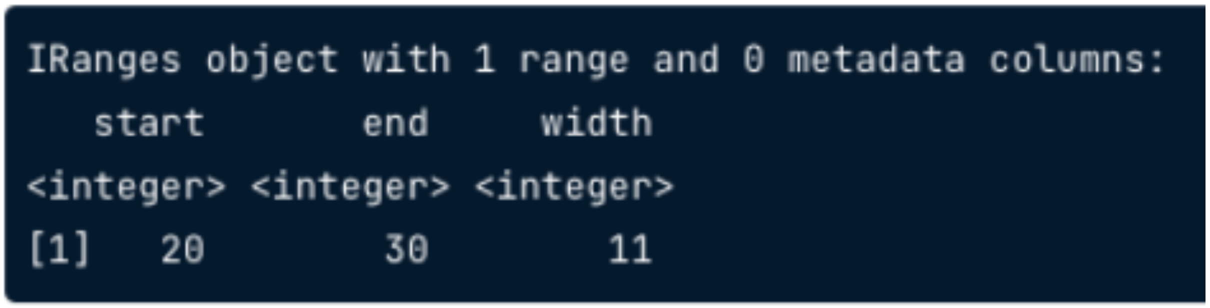
# Use vmatchPattern() with the set
vmatchPattern(pattern = ns5, subject = selectedSet, max.mismatch = 15)

# Use matchPattern() with the single sequence
matchPattern(pattern = ns5, subject = selectedSeq, max.mismatch = 15)

# Take your time to see the similarities/differences in the result.
**result for this example is the same
star t 3023 end 3347 width 325

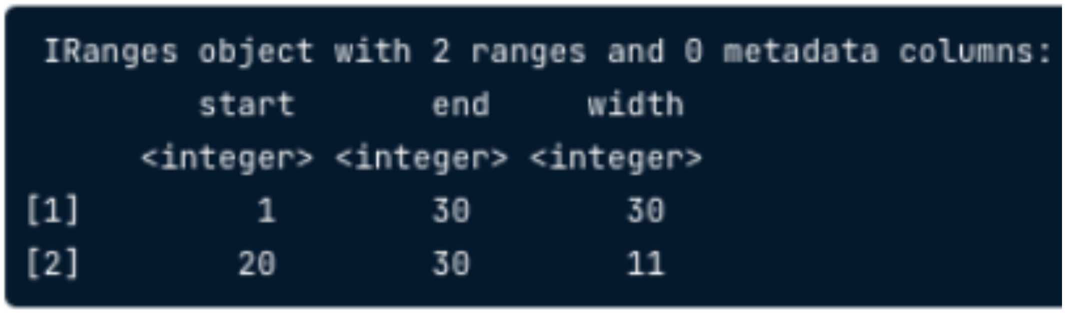
IRanges and Genomic Structures
IRanges package provides the fundamental infrastructure and operations for manipulating intervals of sequences
with Bioconductor
library(IRanges)
*a range is defined by ‘start’ and ‘end’
myIRanges <- IRanges(start = 20, end = 30)
myIRanges

output>



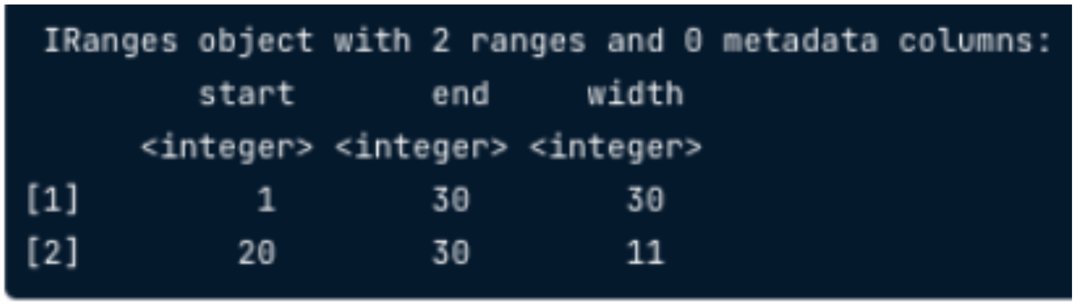
```
IRanges object with 1 range and 0 metadata columns:
   start       end     width
<integer> <integer> <integer>
[1]   20         30        11
```

further examples:
myIRanges_width <- IRanges(start = c(1, 20), width = c(30, 11)))
**width = end - start +1



```
 IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
[1]           1        30        30
[2]          20        30        11
```

myIRanges_end <- IRanges(start = c(1, 20), end = 30))
*can recycle values as ‘end’ here
**width = end - start +1



```
 IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
[1]           1        30        30
[2]          20        30        11
```

Rle - run length encoding
another way to construct IRanges
Rle() function computes and stores the legnth and values of a vector or factor
*Rle is general S4 container used to save long repetitive vectors efficiently
example:

```r
(some_numbers <- c(3, 2, 2, 2, 3, 3, 4, 2))
```

```
3 2 2 2 3 3 4 2
```

```r
(Rle(some_numbers))
```

```
numeric-Rle of length 8 with 5 runs
Lengths: 1 3 2 1 1
Values : 3 2 3 4 2
```

Rle turned the above example vector from 8 to 5 and noted seqeunce via reading the “Lengths” output
one 3, three 2s, two 3s, one 4, one 2

IRanges can also be a logical vector

```r
IRanges(start = c(FALSE, FALSE, TRUE, TRUE))
```

```
 IRanges object with 1 range and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
  [1]         3         4         2
```

skips element 1 and 2, starts on element 3, ends on element 4 for a width (4-3 +1) of 2
*this technique becomes particularly useful when you want to skip elements of a sequence
can also create this logical vector based on a condition

Can still use Rle with logical elements

```r
gi <- c(TRUE, TRUE, FALSE, FALSE, TRUE, TRUE, TRUE)
myRle <- Rle(gi)
```

```
logical-Rle of length 7 with 3 runs
Lengths:     2     2     3
Values :  TRUE FALSE  TRUE
```

```r
IRanges(start = myRle)
```

```
IRanges object with 2 ranges and 0 metadata columns:
          start       end     width
      <integer> <integer> <integer>
[1]         1         2         2
[2]         5         7         3
```

reads as 2 true, 2 false, 3 true
Rle reads as two sequences

1. starts at element 1 and ends with last TRUE at element 2 with a width of 2
2. element 3 and 4 are skipped, starts at element 5 goes to last TRUE ending on element 7 with a width of 3

Remember IRanges are hierarchical data structures that can contain metadata useful to store genes, transcripts, polymorphisms and more

Example
# Load IRanges package
library(IRanges)

# IRnum1: start - vector 1 through 5, end - 100
IRnum1 <- IRanges(start = c(1:5), end = 100)

# IRnum2: end - 100, width - 89 and 10
IRnum2 <- IRanges(end = 100, width = c(89, 10))

# IRlog1: start = Rle(c(F, T, T, T, F, T, T, T)))
IRlog1 <- IRanges(start = Rle(c(F, T, T, T, F, T, T, T)))

```
# Print objects in a list
print(list(IRnum1 = IRnum1, IRnum2 = IRnum2, IRlog1 = IRlog1))
```

Example

```
# Create the first sequence seq_1
seq_1 <- IRanges(start = 10, end = 37)

# Create the second sequence seq_2
seq_2 <- IRanges(start = c(5, 35, 50),
             end = c(12, 39, 61),
             names = LETTERS[1:3])

# Check the width of seq_1 and seq_2
width(seq_1)
[1] 28
width(seq_2)
[1]  8  5 12
# Create the first sequence seq_1
seq_1 <- IRanges(start = 10, end = 37)

# Create the second sequence seq_2
seq_2 <- IRanges(start = c(5, 35, 50),
             end = c(12, 39, 61),
             names = LETTERS[1:3])

# Check the width of seq_1 and seq_2
lengths(seq_1)
[1] 28
lengths(seq_2)
 A  B  C
 8  5 12
```

Gene of interest
genomic intervals
when working with genome data we mostly work by comparing sequence intervals to a reference
a genome is represented as a linear sequence split over multiple chromosomes > hence we have sets of sequences
*biological relevant features are included as metadata in GRanges
these genome intervals are reads aligned to a reference, genes of interest, exonic regions, SNPs, regions of transcription or binding sites

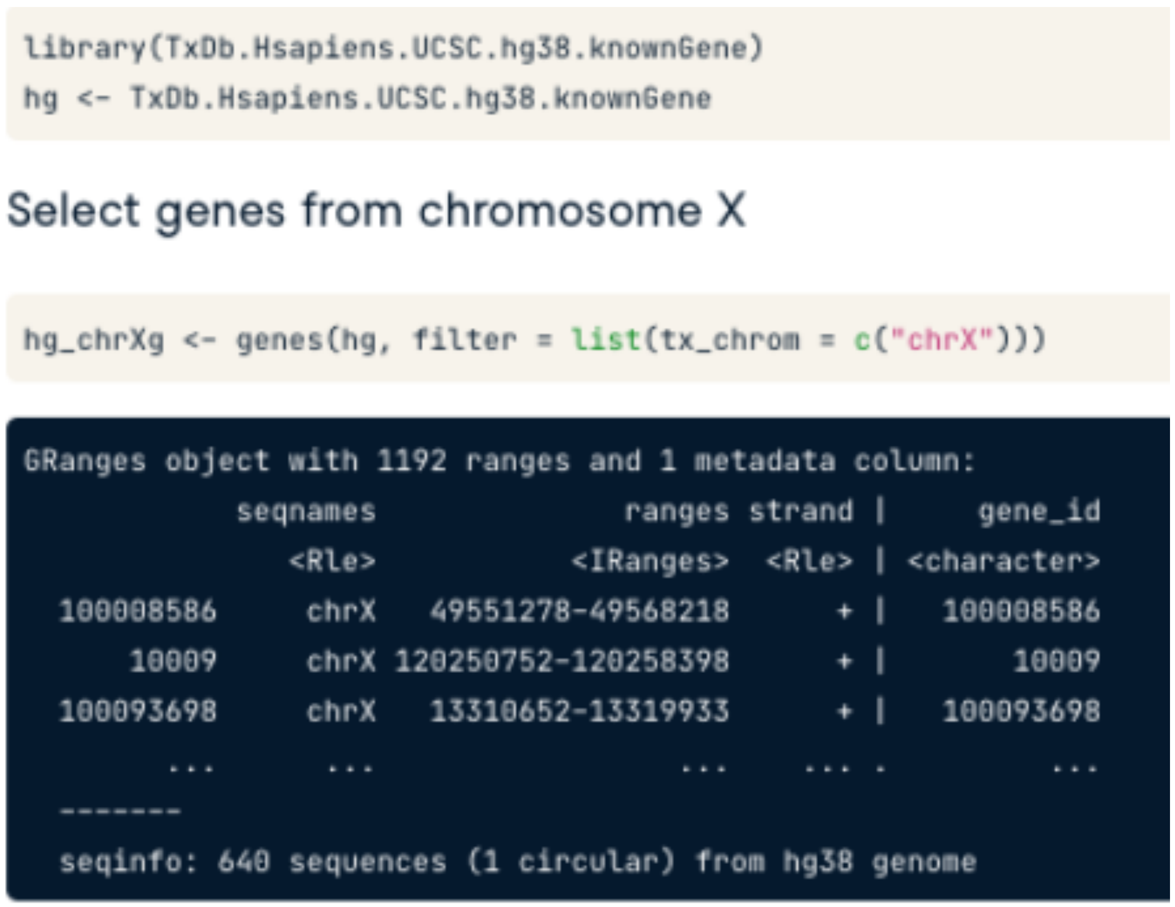
library(GenomicRanges)
uses GRanges > a type of container used to save genomic intervals per chromosome
example with bare arguments chromosome name and start and end of interval
myGR <- GRanges(“chr1:200-300”))
difference from IRanges > G is associated with a chromosome and a strand
metadata per range > score, GC percentage, interval names, seqnames, seqinfo

Sequence intervals must come in the form of a table (ie dataframe or tibble)
we can than take this dataframe and construct a GRange object
example
(myGR <- as(df, “GRanges”)) #transforms df into GRanges

Genomic Ranges accessors
methods(class = ‘GRanges) #to check available accessors
seqnames(gr) #to get chromosomes names
ranges(gr) #returns an IRanges object for ranges
mcols(gr) #to display additional metadata per range
seqinfo(gr) #display a summary of the sequence information
genome(gr) #the genome name
*Accessors can be inherited thanks to S4

Gene of interest for our practice
ABCD1
located at the end of chromosome X long arm
encodes a protein relevant for the well fucntioning of brain and lung celss in mammals
chromosome X is about 156 million base pairs long
our gene is located in a small interval around 153.70 mi bp

example using a human reference from UCSC database
subset the reference using the genes function to chromosome X



```
library(TxDb.Hsapiens.UCSC.hg38.knownGene)
hg <- TxDb.Hsapiens.UCSC.hg38.knownGene
```

Select genes from chromosome X

```
hg_chrXg <- genes(hg, filter = list(tx_chrom = c("chrX")))
```

```
GRanges object with 1192 ranges and 1 metadata column:
            seqnames              ranges strand |     gene_id
               <Rle>           <IRanges>  <Rle> | <character>
  100008586     chrX   49551278-49568218      + |   100008586
      10009     chrX 120250752-120258398      + |       10009
  100093698     chrX   13310652-13319933      + |   100093698
        ...      ...                 ...    ... .         ...
  -------
  seqinfo: 640 sequences (1 circular) from hg38 genome
```

If you would like to test other filters, valid names for this list are: "gene_id", "tx_id", "tx_name", "tx_chrom", "tx_strand", "exon_id", "exon_name", "exon_chrom", "exon_strand", "cds_id", "cds_name", "cds_chrom", "cds_strand", and "exon_rank".

Example

```
# Load human reference genome hg38
library(TxDb.Hsapiens.UCSC.hg38.knownGene)

# Assign hg38 to hg, then print it
hg <- TxDb.Hsapiens.UCSC.hg38.knownGene
hg

# Extract all positive stranded genes in chromosome X, assign to hg_chrXgp, then sort it
hg_chrXgp <- genes(hg, filter = list(tx_chrom = c("chrX"), tx_strand = "+"))
sort(hg_chrXgp)
```

Manipulating collections of GRanges

GRangesList is a container for storing a collection of GRanges

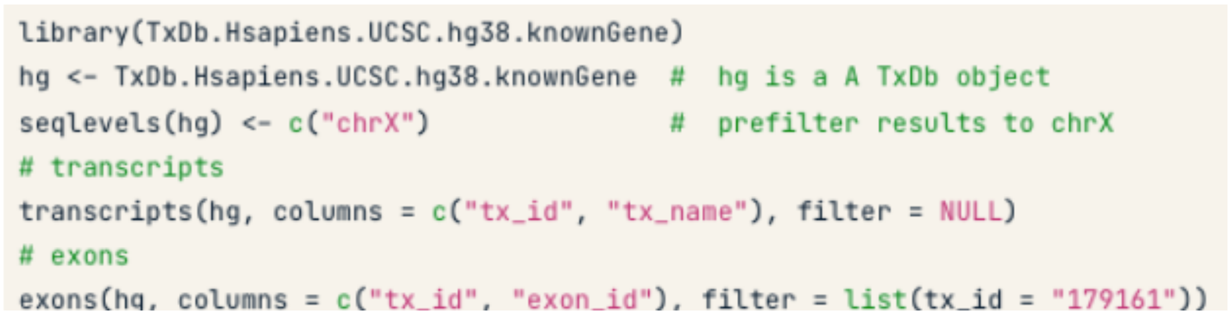
efficient for storing a large number of elements
to construct:
as(mylist, “GRangesList”)
GRangesList(myGranges1, myGRanges2,...)
to convert back:
unlist(myGRangesList)
for list of accessors:
methods(class = ‘GRangeList’)

Examples of GRangesLists
-transcripts by gene
-exons by transcripts
-read alignments
-sliding windows

Breaking a region into smaller regions
slidingWindows(hg_chrX, width = 20000, step = 10000)
#returns a GRangesList
#above splits each gene into new ranges of 20,000 bases with the distance between ranges is 10,000 bases
#each range has an overlap of 10,000 bases (width - step)

GenomicFeatures()
uses transcript database objects to store metadata, manage locations and relationships between features and its identifiers
examples genes, transcripts, and exons
managed by providers like UCSC
useful for ChP-seq, RNA-seq and annotation analyses
Bioconductor provides built-in packages for the most used transcript databases

example

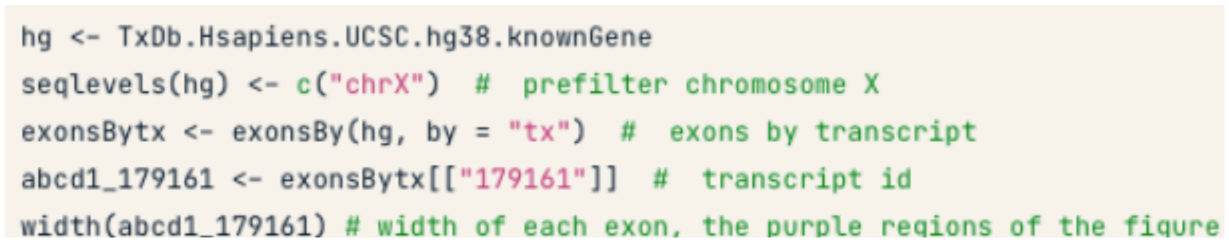


```
library(TxDb.Hsapiens.UCSC.hg38.knownGene)
hg <- TxDb.Hsapiens.UCSC.hg38.knownGene  #  hg is a A TxDb object
seqlevels(hg) <- c("chrX")               #  prefilter results to chrX
# transcripts
transcripts(hg, columns = c("tx_id", "tx_name"), filter = NULL)
# exons
exons(hg, columns = c("tx_id", "exon_id"), filter = list(tx_id = "179161"))
```

additional extracting function options > genes, cds, and promoters
‘filter’ uses a condition on a column
‘filter’ options:
"gene_id", "tx_id", "tx_name", "tx_chrom", "tx_strand", "exon_id", "exon_name",

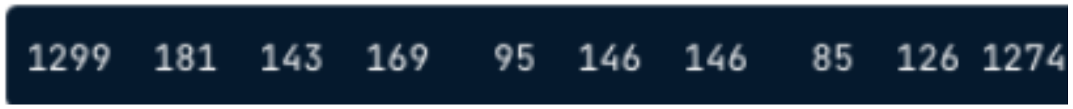
"exon_chrom", "exon_strand", "cds_id", "cds_name", "cds_chrom", "cds_strand", and "exon_rank"

Exons are coding sections of an RNA transcript, or the DNA encoding it, that are translated into protein
Each gene has one or more transcripts > each transcript has a set of exons
retrieve all the exons by transcript using the exonsBy() function
‘by’ argument response ’tx’ is short for transcript
example



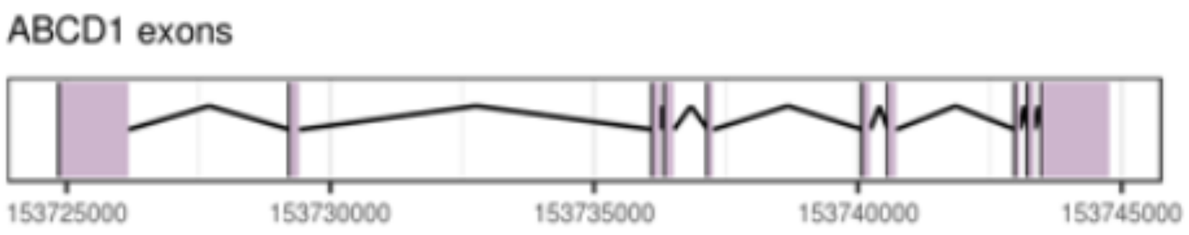
```
hg <- TxDb.Hsapiens.UCSC.hg38.knownGene
seqlevels(hg) <- c("chrX")  #  prefilter chromosome X
exonsBytx <- exonsBy(hg, by = "tx")  #  exons by transcript
abcd1_179161 <- exonsBytx[["179161"]]  #  transcript id
width(abcd1_179161) # width of each exon, the purple regions of the figure
```

output>



```
1299  181  143  169   95  146  146   85  126 1274
```

shows 10 exons, value is their widths as a numeric vector
what this looks like visually >



Overlaps
to find genes of interest in a larger interval or a collection of intervals
counting, finding, and subsetting overlaps between objects containing genomic ranges are useful to annotating genomic features
set of functions:
countOverlaps(query, subject) #results in an integer vector of counts
findOverlaps(query, subject) #results in a Hits object
subsetByOverlabps(query, subject) #returns a GRangesList object
*overlaps might be complete or partial if the is match is a subset of the query

Example
# Load the human transcripts DB to hg
library(TxDb.Hsapiens.UCSC.hg38.knownGene)

```
hg <- TxDb.Hsapiens.UCSC.hg38.knownGene

# Prefilter chromosome X "chrX" using seqlevels()
seqlevels(hg) <- c("chrX")

# Get all transcripts by gene and print it
hg_chrXt <- transcriptsBy(hg, by = "gene")
hg_chrXt

# Select gene `215` from the hg_chrXt
hg_chrXt$"215"
```

Bioconductor Packages to explore sequence data quality
genome for our example is the plant Arbidopsis thaliana
first plant species to be completely sequenced
genome size of 135 megabase pairs

fastq vs fasta
need to store sequences
store as text
two go to formats
fastQ and fastA
main difference is that fastQ files include quality encoding per sequenced letter
fastQ is the standard
fastq described in 4 lines:

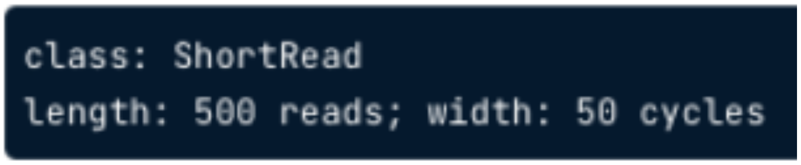
1. @ unique sequence identifier or description
2. raw sequence string
3. + optional id (sequence identifier)
4. encodes the quality values of the sequence with one encoding value per sequenced letter

common fastq file extensions > fastq, fq
fasta described in 2 lines:

1. > unique sequence identifier
2. raw sequence string

fasta
ShortRead package provides us with readFasta() which reads all Fasta-formatted files in a directory Path
library(ShortRead)
fasample <- readFast(dirPath = “data/“, pattern = “fasta”)
can read compressed or uncompressed files
returns a single object representation of class ShortRead

*class stores and manipulates unifor-length short read sequences and their identifiers
example output>



```
class: ShortRead
length: 500 reads; width: 50 cycles
```

#for accessors
methods(class = “ShortRead”)
#write and object to a single file
writeFasta(fasample, file = “data/sample.fasta”)
this can also compress on the fly

fastq
similar to fasta
same library
fqsample <- readFastq(dirPath = “data/“, pattern = “fastq”) #two additional arguments ‘qualityType’ and ‘filter’
creates class ShortReadQ
#for accessors
methods(class = “ShortReadQ”)
#to write a ShortReadQ object to a single file
writeFastq(fqsample, file = “data/sample.fastq.gz”)
be mindful of the .gz extension
this allows you to appen new sequences to an existing file and save a compressed version

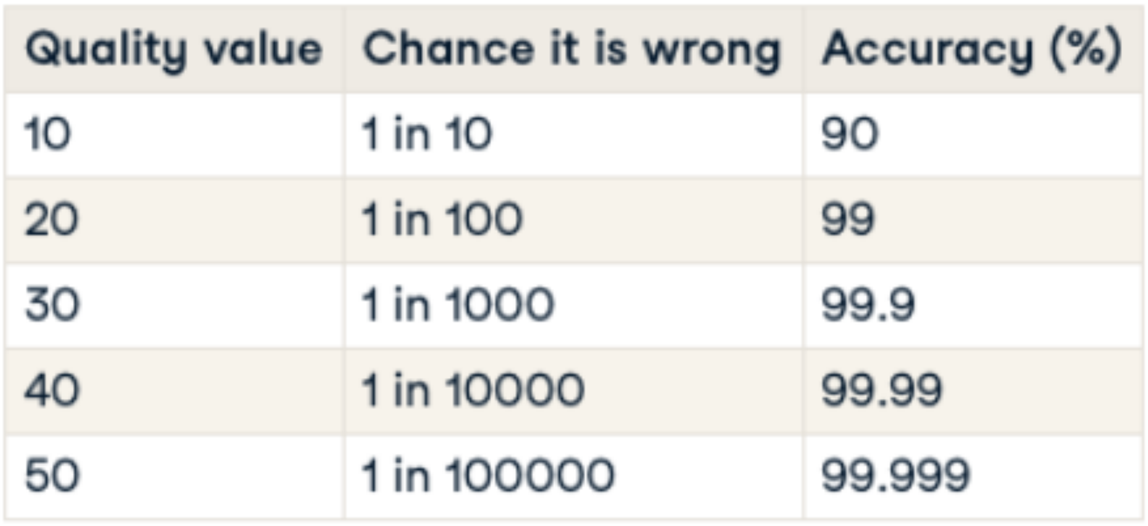
Subsetting a sample
can use ‘seed’ for repeatability
set.seed(123)
#subsample
sampler <- FastqSampler(“data/SRR1971253.fastq”, 500)
#use ‘yield’ function to extract the sampe from the stored file and save
sample_small <- yield(sampler)

Example
# Load ShortRead
library(ShortRead)

# Print fqsample
fqsample

```
# Check class of fqsample
class(fqsample)

# Check class sread fqsample
class(sread(fqsample))

# Check ids of fqsample
id(fqsample)

# Load ShortRead
library(ShortRead)

# Set a seed for sampling
set.seed(1234)

# Use FastqSampler with f and select 100 reads
fs <- FastqSampler(con = f, n = 100)

# Generate new sample yield
my_sample <- yield(fs)

# Print my_sample
my_sample
```

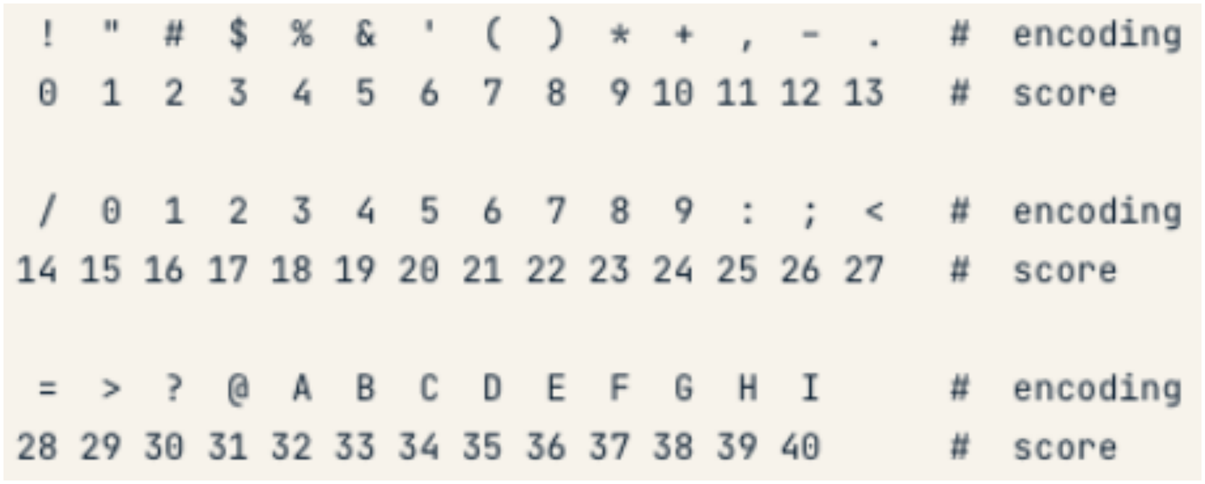
Assessing sequence and data quality
*here we are assessing accuracy
we use the Phred table



Quality value	Chance it is wrong	Accuracy (%)
10	1 in 10	90
20	1 in 100	99
30	1 in 1000	99.9
40	1 in 10000	99.99
50	1 in 100000	99.999

example what this says:
quality value 30 tells us that one base in a 1000 might be wrong

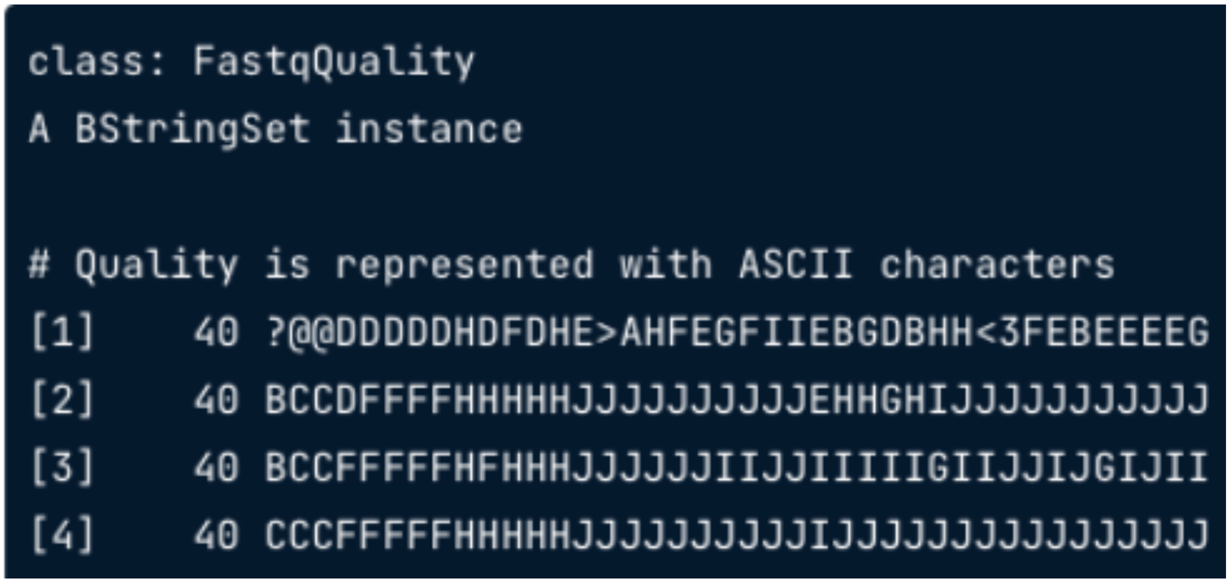
with R:
encoding(quality(fqsample))
#output of encoding characters and their scores



```
 !  "  #  $  %  &  '  (  )  *  +  ,  -  .   # encoding
 0  1  2  3  4  5  6  7  8  9 10 11 12 13   # score

 /  0  1  2  3  4  5  6  7  8  9  :  ;  <   # encoding
14 15 16 17 18 19 20 21 22 23 24 25 26 27   # score

 =  >  ?  @  A  B  C  D  E  F  G  H  I      # encoding
28 29 30 31 32 33 34 35 36 37 38 39 40      # score
```

usual range is 2-40; scores can be higher
this is the standard but other encodings exist

with R:
library(ShortRead)
#fastq files encode quality scores on a class ‘FastqQuality’
#quality() function obtains the quality of a sequence
example
quality(fqsample)



```
class: FastqQuality
A BStringSet instance

# Quality is represented with ASCII characters
[1]    40 ?@@DDDDDHDFDHE>AHFEGFIIEBGDBHH<3FEBEEEEG
[2]    40 BCCDFFFFHHHHHJJJJJJJJJEHHGHIJJJJJJJJJJ
[3]    40 BCCFFFFFHFHHHJJJJJIIJJIIIIIGIIJJIJGIJII
[4]    40 CCCFFFFFHHHHHJJJJJJJJJIJJJJJJJJJJJJJJJ
```

Exploring quality encoding

```
library(ShortRead)
sread(fqsample)[1]
# Quality is represented with ASCII characters
quality(fqsample)[1]
```

```
50 GTCCCATTTACCTCTGACTCTTTTGATGCTGCAATTGCTGCTCATATACT
50 ?@@DDDDDHDFDHE>AHFEGFIIEBGDBHH<3FEBEEEEGGIGIIGHGHC
```

```
## PhredQuality instance
pq <- PhredQuality(quality(fqsample))
# transform encoding into scores
qs <- as(pq, "IntegerList")
qs # print scores
```

```
30 31 31 35 35 35 35 35 39 35 37 35 39 36 29 32 39 37 36 38 37 40 40 36 33 38 35 33 39 39 27 18 37 36 33 36
36 36 36 38 38 40 38 40 40 38 39 38 39 34
```

sread indexed allows us to read the first line
quality() indexed gives us the encoding values
we transform the encoding values into scores with PhredQuality()
then convert the scores into numeric scores by turning it into an “IntegerList”
the community generally sees a score of 30 as of good quality
here we can see almost all of the scores are above 30

Quality Assessment

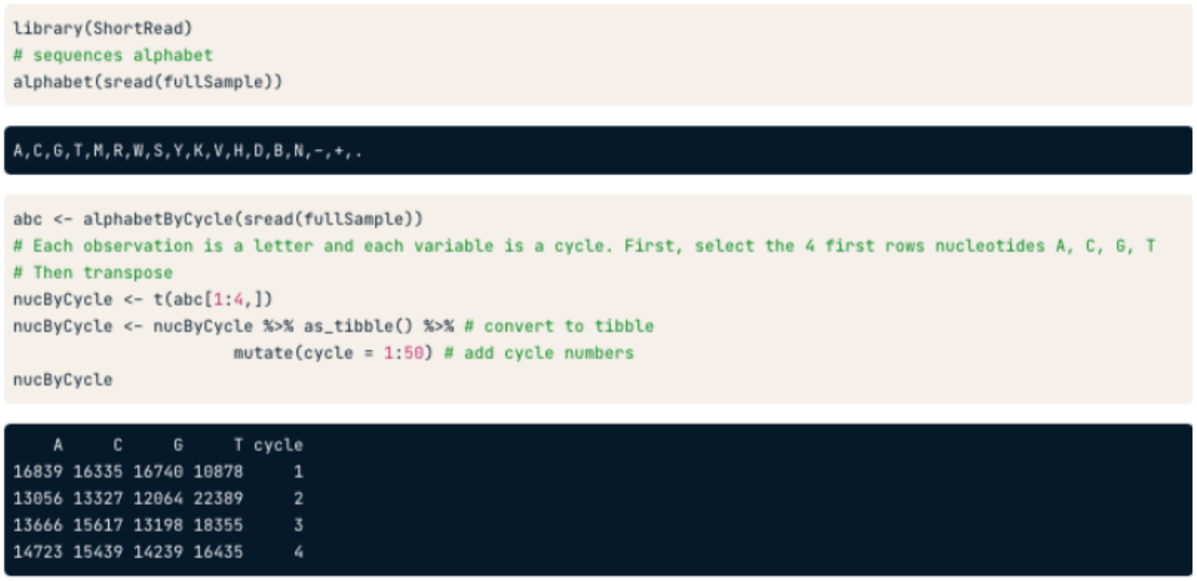
```
library(ShortRead)
# Quality assessment
qaSummary <- qa(fqsample, lane = 1)    # optional lane
# class: ShortReadQQA(10)
# Names accessible with the quality assessment summary
names(qaSummary)
```

```
 [1] "readCounts"          "baseCalls"           "readQualityScore"       "baseQuality"
 [5] "alignQuality"        "frequentSequences"   "sequenceDistribution"   "perCycle"
 [9] "perTile"             "adapterContamination"
# QA elements are accessed with qa[["name"]]
```

```
# Get a HTML report
browseURL(report(qaSummary))
```

qa() gives you lots of summary assessments about your sequence file/s
can call all these assessments with [[ ]] to get a summary of each evaluation
browseURL will gvie you a bigger picture

Example - analyzing nucleotide frequency per cycle



```
library(ShortRead)
# sequences alphabet
alphabet(sread(fullSample))
```

```
A,C,G,T,M,R,W,S,Y,K,V,H,D,B,N,-,+,.
```

```
abc <- alphabetByCycle(sread(fullSample))
# Each observation is a letter and each variable is a cycle. First, select the 4 first rows nucleotides A, C, G, T
# Then transpose
nucByCycle <- t(abc[1:4,])
nucByCycle <- nucByCycle %>% as_tibble() %>% # convert to tibble
                     mutate(cycle = 1:50) # add cycle numbers
nucByCycle
```

```
    A     C     G     T cycle
16839 16335 16740 10878     1
13056 13327 12064 22389     2
13666 15617 13198 18355     3
14723 15439 14239 16435     4
```

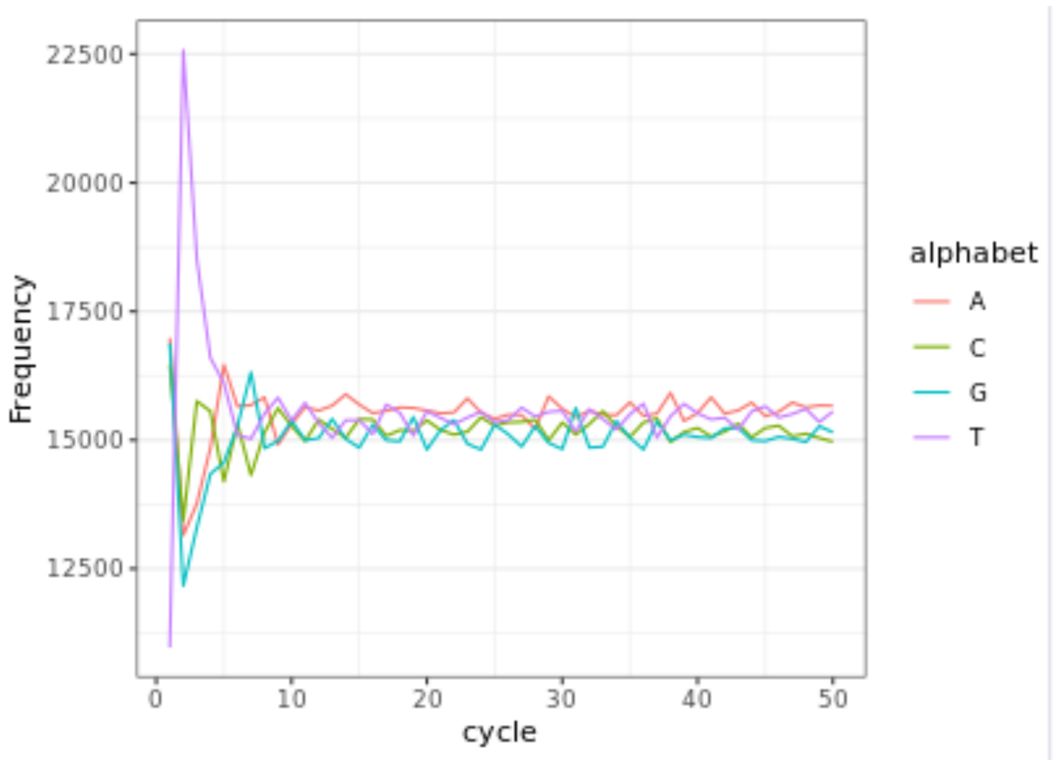
Example

```
# load ShortRead
library(ShortRead)

# Check quality
quality(fqsample)

# Check encoding of quality
encoding(quality(fqsample))

# Check baseQuality
qaSummary[['baseQuality']]
```

Example

```
# Glimpse nucByCycle
glimpse(nucByCycle)

# Create a line plot of cycle vs. count
nucByCycle %>%
  # Gather the nucleotide letters in alphabet and get a new count column
  pivot_longer(-cycle, names_to = "alphabet", values_to = "count") %>%
  ggplot(aes(x = cycle, y = count, color = alphabet)) +
  geom_line(size = 0.5 ) +
  labs(y = "Frequency") +
  theme_bw() +
  theme(panel.grid.major.x = element_blank())
```

output>



Match and filter
Duplicates should always trigger alarm
yes duplicates happen in nature
but they can also happen due to PCR amplification in library preparation
or when sequencing the same molecule more than once
these errors might lead to 30-70% identical copies in your sample
industry standard when working with whole genome sequencing or exome sequencing is to remove or mark duplicates
you can also set a threshold for acceptable duplicate percentage > useful with RNA-seq, ChIP-seq

Finding duplications
table(srduplicated(dfqsample)
returns a logical argument
use table to get counts

One way to clean-up duplicates
subset all those reads the are marked as not duplicated with a condition in the vector
cleanReads <- mydReads[srduplicated(mydReads) == FALSE]

Creating your own filters

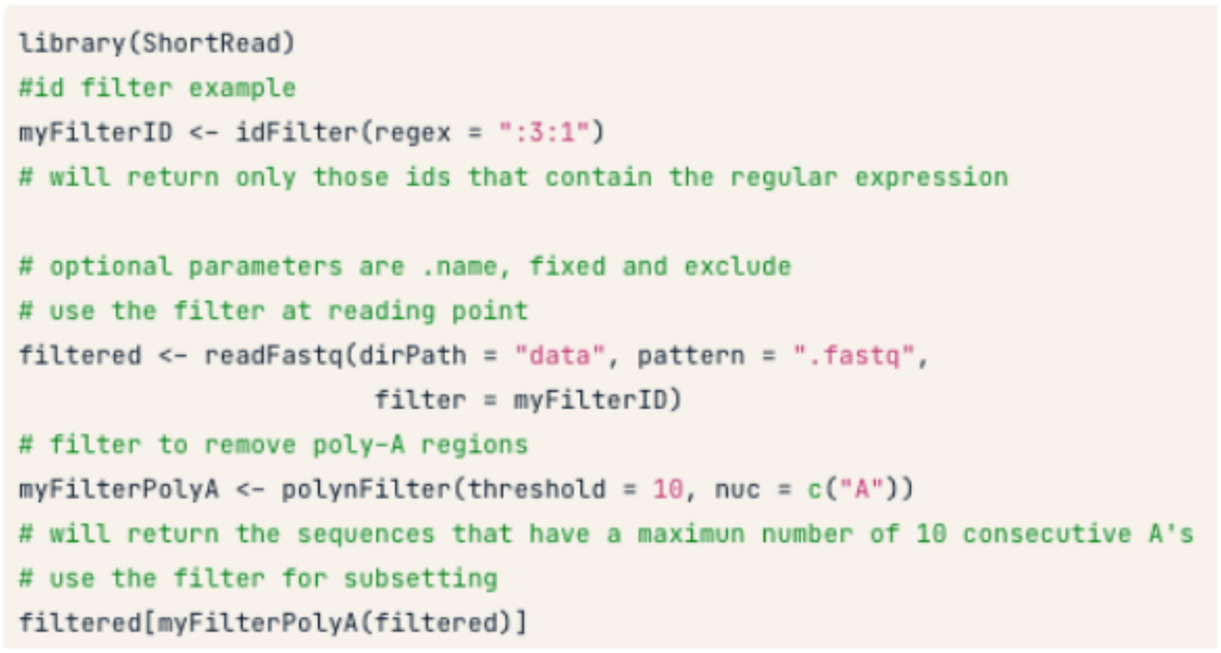
srFilter() is a function to construct your own personalized ShortRead filters
it accepts a single argument (our example - fqsample) and returns a logical vector used to select elements of fqsample satisfying a condition
example
readWidthCutOff <- srFilter(function(x) }{width(x) >= minWidth}, name = “MinWidth”)
#extra parameters can be specified before calling the filter, here minWidth
minWidth <-51
#filter is applied on fqsample using the filter to subset
fqsample[readWidthCutOff(fqsample)]

Built-in filters
nFilter
has threshold parameter representing maximum number of N’s allowed on each read
.name creates a custom name to your filter
myFilter <- nFilter(threshold = 10, .name = “cleanNFilter”)
#can use the filter directly when reading the fastq files

```
filtered <- readFastq(dirPath = “data”,
                      pattern = “.fastq”,
                      filter = myFilter)
```

filtered #will retrieve only those reads that have a maximum of 10 N’s
#works as a very fast cleaning step

IdFilter and polynFilter



```
library(ShortRead)
#id filter example
myFilterID <- idFilter(regex = ":3:1")
# will return only those ids that contain the regular expression

# optional parameters are .name, fixed and exclude
# use the filter at reading point
filtered <- readFastq(dirPath = "data", pattern = ".fastq",
                      filter = myFilterID)
# filter to remove poly-A regions
myFilterPolyA <- polynFilter(threshold = 10, nuc = c("A"))
# will return the sequences that have a maximun number of 10 consecutive A's
# use the filter for subsetting
filtered[myFilterPolyA(filtered)]
```

Example

```
# Load package ShortRead
library(ShortRead)

# Check class of fqsample
class(fqsample)

# Filter reads into selectedReads using myStartFilter
selectedReads <- fqsample[myStartFilter(fqsample)]

# Check class of selectedReads
class(selectedReads)

# Check detail of selectedReads
detail(selectedReads)
```

Example

```
# Check reads of fqsample
sread(fqsample)

# Create myFil using polynFilter
myFil <- polynFilter(threshold = 3, nuc = c("A"))

# Apply your filter to fqsample
filterCondition <- myFil(fqsample)

# Use myFil with fqsample
filteredSequences <- fqsample[filterCondition]

# Check reads of filteredSequences
sread(filteredSequences)
```

Rqc package > quality control tool for high-throughput sequencing data
deals with big files
saves time and resources
performs parallel processing

```
library(Rqc)
aqRgq <- rqcQA(fastq_files, workers = 4, sample = TRUE, n = 500)
#’workers’ defines the amount of computer cores to work in parallel
#’sample’ argument will put the quality assessment in a subset of the input
#n selects the number of reads
```

always remember to set a seed before calling a sample

resulting object is a list
names(qaRqc) will give the name of the input files quality assessment
default rqcQA treats all files as single-end
if you have two files per sample id > create a numeric vector

```
# paired-end files
pfiles <- "data/seq_11.fq" "data/seq1_2.fq" "data/seq2_1.fq" "data/seq2_2.fq"

qaRqc_paired <- rqcQA(pfiles, workers = 4, pair = c(1, 1, 2, 2)))
```

For reports with custom templates, use:

```
reportFile <- rqcReport(qaRqc, templateFile = "myReport.Rmd")
browseURL(reportFile)
```

Rqc's 12 plotting functions:

`rqcCycleAverageQualityPcaPlot()`	`rqcGroupCycleAverageQualityPlot()`
`rqcCycleAverageQualityPlot()`	`rqcReadQualityBoxPlot()`
`rqcCycleBaseCallsLinePlot()`	`rqcReadQualityPlot()`
`rqcCycleBaseCallsPlot()`	`rqcReadWidthPlot()`
`rqcCycleGCPlot()`	`rqcReadFrequencyPlot()`
`rqcCycleQualityBoxPlot()`	`rqcCycleQualityPlot()`