**Introduction to the Bioconductor Project**

**1. The Bioconductor Project**

Hi! I'm Paula Martinez, and I'll be your instructor. I am a data scientist and I train people on how to analyze their data more efficiently. Also, I am a bioinformatician, interested in genomic variation that leads to diversity. For these kinds of analyses, I love using R and Bioconductor, particularly because of the possibilities to work with different datasets and to collaborate within the community. During this introduction, you will discover the most commonly used Bioconductor packages. We will start with installation and at the end of this chapter, you will be able to retrieve whole genome sequences, using Bioconductor.

**2. Bioconductor**

The Bioconductor project is an open source repository for R packages, datasets and workflows that are specific for analyzing biological data. The Bioconductor project is a useful extension on CRAN, the R Archive, because it provides us with the software tools to explore, understand, and solve simple and complex molecular biology questions. Hence, Bioconductor's tagline is "open source software for bioinformatics".

1. 1 Bioconductor (www.bioconductor.org)

**3. What do we measure and why?**

Molecular biology questions are usually about either the structure or the function of each of the building blocks of an organism, and very often how they interconnect to one another. In this course, you will learn commonly used packages that will help you understand the structure of biological data. That is, you will find out more about the elements, their regions, their size and order, and how they relate to other data. Other Bioconductor courses on DataCamp will teach you more about the functions of the building blocks, such as gene expression and regulation, and how these affect phenotypes such as health/disease, evolution, and much more.

**4. How to install Bioconductor packages?**

The Bioconductor package collection forms its own repository and has a release schedule different from the R Archive. Because Bioconductor has it's own base functions and updates, packages are installed differently. To install Bioconductor packages you need two lines of code, as shown on the slide. First, use the function source on the script, BiocLite-dot-R, from Bioconductor-dot-org This script will install the BiocInstaller package. Then, use the function biocLite() with the name of the package you want to install. Once you source the BiocLite, you will be informed if any new versions of Bioconductor are available, and will also see a prompt to update your R version if needed. Updating packages, regularly, is important to get the new features. In case there are upgrades on packages or dependencies, you will be asked to update all, some, or none of the packages.

**5. Bioconductor version and package version**

Because Bioconductor is in constant development, you can check the version of Bioconductor using the syntax BiocInstaller::bioVersion() or if you already loaded the BiocInstaller package you can call the function biocVersion() directly. To load a package use the function library like with CRAN packages. It's important for reproducibility to always check the versions of your packages. You can use the function sessionInfo() which gives you a list of all the loaded packages and their versions, or you can inquire the version of each package using packageVersion() and the packageName. Finally, if you are interested to know if you have out-of-date packages, use the function biocValid().

## 1. The Role of S4 in Bioconductor

R is a functional programming language. Because S4 is extensively used in Bioconductor packages we will cover the basics here. But bear in mind, this is a big enough topic for a book. The two most used systems in R are S3 and S4, in order to understand better S4 we will start by shortly explaining S3-characteristics, and then making a comparison with S4.

## 2. S3

Most of CRAN packages use the S3 system, which is simple but powerful. It gives flexibility to the user and it has an interactive nature. This means it reacts differently, depending on the input, by using a generic function to decide which method to call. In R a good example is the plot() function. It actually has about 29 different kinds of plots which will be interpreted depending on your input. This is fantastic and makes R very flexible for the user. However, S3 is not good at evaluating and validating types, hence when the user gives an unexpected argument, the generic function will try all its options before it returns an error. Other annoyances are the poor use of naming conventions and how inheritance differs by the input.

## 3. S4

The S4 system motivation is to implement an object-oriented style of programming. The base concept is to define the data first and then 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. As you will see in more detail in the coming chapters, biological representations are complex and very often interconnected. Bioconductor then, recommends re-using methods and classes before implementing new representations. S4 classes have a formal definition, hence are a lot better to check input types, because of their definition and inheritance. You can create a new object from a class like in the example. We created a new genome description. This object will contain slots to describe it. S4 requires a bit more work to implement, but it serves to extend the code so that others can reuse the framework.

## 4. Is it S4 or not?

How do we know whether a definition of an object is S4 or not? Use the function isS4() It returns a logical. Another way to check if an object is S4 is by using the `str` function. If it is S4 it will start with 'Formal class'. On the other hand, S3 objects are those that respond FALSE to isS4() and don't have a formal class definition.

## 5. S4 class definition

An S4 class describes a representation of an object with a name and slots (also called methods or fields), helpful for validation. A class optionally describes its inheritance (usually with the parameter contains). A class allows us to define all the characteristics concerning an object and it gives us code reusability. For example, we create a class using setClass() Its name is MyEpicProject with three slots: ini, end and milestone. This class inherits from the class MyProject. This means we can reuse slots from it.

## 6. S4 accessors

Basic information of an S4 object is accessed through accessor-functions, also called methods. As we have seen, a class definition includes slots for describing an object. The function dot-S4methods() used with main classes gives you a summary of its accessors. For other subclasses use the showMethods() function, but this gives you a breakdown, which might be a bit too long to look at. If you want an object summary, use the accessor function show(). You will use some of these accessors in the coming exercises.

## 1. Introducing biology of genomic datasets

Hi!, this is to introduce a bit of cell biology, talking about organisms, genomes, and the yeast genome.

## 2. Organisms

An organism is a complex structure of interconnected elements that integrate the overall functioning of the being. Organisms diversity is immense, from unicellular to multicellular, with a nucleus and without it, different membranes, and systems, different life cycles and more. In our time, we study organisms in detail by sequencing genomes and dissecting its elements to find interesting functions.

## 3. What is DNA, what is a genome?

All organisms have a genome which makes up what they are, and it dictates responses to external influences. A genome is the complete genetic material of an organism stored mostly in the chromosomes, it's known as the blueprint of the living. A genome is made of long sequences of DNA, based on a four-letter-alphabet, T, A, G and C.

## 4. Genome elements

We are interested in locating and describing specific locations in a genome because this allows us to learn about diversity, evolution, hereditary changes, and more. To understand this better we subdivide a genome. The written information in a genome uses the DNA alphabet. Think of a genome as a set of books and each book is a chromosome. Chromosome numbers on each genome are highly variable. Usually, chromosomes come in pairs, but multiple sets are very common too. Each chromosome has ordered genetic sequences, think of chapters in a book. To find specific genetic instructions we look at genes. These are like the pages in a book, containing a recipe to make proteins. Some genes will produce proteins but some won't. These are called coding and non-coding genes. Coding genes are expressed through proteins responsible for specific functions. Proteins come up following a two-step process, DNA-to-RNA, a step known as transcription, while the RNA-to-protein is a step called translation.

## 5. Yeast

As an example, we are going to study the Yeast genome, a single cell microorganism. The fungus that people love. Yeast is used for fermentation and production of beer, bread, kefir, kombucha and other foods, as well as used for bioremediation. Its scientific name is Saccharomyces cerevisiae or s. cerevisiae. Yeast is a very well studied organism, due to its fast development, many experiments use it as model.

## 6. Yeast genome

The yeast genome is a dataset available from UCSC. We have picked this genome because it has a small size. In the following exercises, you will find out more about this genome. For example, you can call a specific genome version and assign it to an object. As shown in the second line. The BSgenome package provides us with many genome datasets. To get a list of the BSgenome available datasets, use the function available-dot-genomes(). Then, using common accessors functions, you can learn about the genome, for example, the number of chromosomes using length(), the names of the chromosomes using names(), and the length of each chromosome by DNA base pairs, using seqlengths().

## 7. Get sequences

Specific genes or regions are interesting because of their functions. You can retrieve sections of a genome with the function getSeq(). The minimum argument required is a BSgenome object. The first example will give you all the sequences in the yeast genome. Then, you can specify some other parameters, to select sequences from chromosome M use "chrM". Next, you can specify the locations of the subsequences to extract, using start, end, or width. Using, end equals 10, it selects the first 10 base pairs of each chromosome of the genome.

## 1. Introduction to Biostrings

Bioconductor is all about handling biological datasets in the most efficient way. As you get more familiar with your biological project and/or experiment, you will notice how big datasets can be. The Biostrings package came to Bioconductor 13 years ago, and it implements algorithms for fast manipulation of large biological sequences. It is so important, that more than 200 Bioconductor packages depend on it. Hence, Biostrings is at the top 5% of Bioconductor downloads.

## 2. Biological string containers

Biological datasets are represented by characters, and these sequences can be extremely large. Biostrings is a useful package because it implements memory efficient containers, especially for sub-setting and matching. Also, these containers can have subclasses. For example, a BString subclass for Big String can store a big sequence of strings as single object or collection. The package Biostrings implements two generic containers, also known as virtual classes; these are XString and XStringSet, from which other subclasses will inherit. To learn more about these classes and how they connect to one another, use the function showClass(), like in the example.

## 3. Biostring alphabets

BioStrings are biology-oriented containers and use a predefined alphabet for storing a DNA sequence, an RNA sequence, or a sequence of amino acids. The DNA\_BASES alphabet has the four bases (A, C, G and T) The RNA\_BASES replace the T for U) and the AA\_STANDARD contains the 20 Amino Acid letters, each is built from 3 consecutive RNA bases. In addition, Biostrings alphabets are based on two code representations: IUPAC\_CODE\_MAP and AMINO\_ACID\_CODE which contains the bases plus extra characters and symbols. It is important to know these code representations so you know which kind of string you are using or need to use depending on the sequence alphabet.

1. 1 For more information IUPAC DNA codes http://genome.ucsc.edu/goldenPath/help/iupac.html

## 4. Transcription and translation

Now that we now the alphabets, let's talk about the two processes that convert sequences from one alphabet to the other. First, a double-stranded DNA gets split. This single strand gets transcribed into RNA, complementing each base. But remember, RNA uses U instead of T. Every three RNA bases translate to a new amino acid. This translation follows the genetic code table to produce new molecules.

## 5. Transcription DNA to RNA

Using BStrings we start with a short DNA sequence saved in a DNAString object. Then, transcription is the process in which a particular segment of DNA is copied into RNA. Using RNAString, it will change all of the T's from the dna\_seq to U's in the rna\_seq, keeping the same sequence length. Remember, you could also begin with a Set if you want to transcribe multiple sequences at the same time.

## 6. Translation RNA to amino acids

To translate RNA sequences into Amino Acid sequences, we need the translation codes stored in RNA\_GENETIC\_CODE and applied by the translate function. In the example, rna\_seq is translated into MIS\*. Where three RNA bases return one Amino Acid. Hence, translation always returns a shorter sequence.

## 7. Shortcut translate DNA to amino acids

Transcription and translation are two separated processes in real life. But, in coding, there is a shortcut. The function translate also accepts DNA Strings and it automatically transcribes to RNA before translating the sequence to Amino Acids, providing the same results.

## 8. The Zika virus

For this chapter, you will use the Zika Virus genome to interact with the package biostrings. The Zika virus genome is very small, of about 10 thousand base pairs. A virus needs a host to live in. The Zika virus is common in tropical areas around the world and it spreads through mosquitoes or blood.

## 1. Sequence handling

As you will begin to discover, Biostrings have many sophisticated string handling utilities for sequence analysis. In this video, we will go on a little tour of these functionalities and continue using the Zika virus genome sequence.

## 2. Single vs set

In the previous exercises, you have been using DNAStrings and DNAStringSets. I wanted you to get a grip on how to handle these two. As a recap, any Xstring will hold one single sequence of a predefined alphabet. However, when you want to store and handle multiple sequences or collections you will use a StringSet. Remember that Sets can have sequences of varying lengths.

## 3. Create a stringSet and collate it

How do we go from a set to a string, and vice versa? First, we read a sequence file with the function readDNAStringSet(). You will notice that this object is like a list with a length of 1, because it only contains one sequence, and its width is the total number of letters in this sequence. Then, to convert a StringSet into a single string, use the function unlist() to collate the elements. In the example, the resulting DNAstring has a length of 10794 characters, but DNAStrings have no width.

## 4. From a single sequence to a set

In case you want to construct a set from a single sequence, use the function DNAStringSet, and specify the sequence, here zikaVirus\_seq, and the subsequences start and end, or the start and width as numeric vectors. Notice the example, zikaSet has three subsequences each having a sequence width of 100 letters. This has been specified by start and end.

## 5. Complement sequence

Did you notice that all the time we have been using DNAStrings or sets, they are one single sequence instead of a double DNA sequence? Well, because you can computationally derive the complement of the sequence we do not need to do the sequencing in both strands. You can generate the sequence when needed using the function complement().

## 6. Rev a sequence

Rev() is a function from package base but it is frequently used with Biostrings. This example uses, for demonstration, the zikashortSet with only 2 sequences (seq1 and seq2) each having 18 letters. Notice how rev() has changed the sequence order, from top to bottom. You can use this function with any string. This is useful to reverse the order of your sequences at the same time, mainly when building a genome reference.

## 7. Reverse a sequence

The reverse() function from IRanges reverses from right to left each sequence of the set. So you can generate the opposite strand of a sequence.

## 8. Reverse complement

Connecting what we have learned so far we have the function reverseComplement(). This function is useful for both DNA and RNA strings. It is equivalent to using the two functions we learned before at once; reverse and complement one inside of the other. As shown in the example, the advantage is that the reverseComplement() function is faster and more memory efficient.

## 9. Recap

For this lesson, I made a distinction when using an XString or an XStringSet. This table shows you the list of functions you've learned and which functions are specific for each string container. unlist() is used with sets, to collate the elements into a single sequence. length() depends on the container. width() is only used for sets and gives you the number of characters per sequence. complement() returns the paired strand of a given sequence. rev() will act as reverse on a single sequence and will reorder a set from bottom to top. reverse() changes the order of a sequence or a set of sequences from right to left. Finally, reverseComplement() is an efficient function which combines reverse and complement together.

## 1. Why are we interested in patterns?

Why are we interested in patterns? Patterns are usually peculiar, interesting, and fun. Think about a sunflower head, the stripes on a zebra, and fingerprints. Patterns in Biology are outstanding and we can learn more about them using sequencing!

## 2. Sequence code

Sequence patterns in the DNA help us find interesting things, such as sequence repeats, protein end codons, poly-A tails, conserved sequences, binding sites, and more. Our goal in analyzing sequence patterns is to discover their occurrence frequency, periodicity, and length.

## 3. What can we find with patterns?

Where does a gene start, where does a protein end, which regions make a gene expressed or silent, which regions are conserved between organisms, and what is the overall genetic variation, are common questions solved by sequence pattern matching.

## 4. Pattern matching

The Biostrings package has a few search functions, which find all the occurrences of a pattern in a subject sequence. The pattern tends to be a short sequence and the subject, a longer sequence. Occurrences may allow the presence of mismatches and can be very stringent or fuzzy. The function matchPattern() compares one single string to another single string. On the other hand, vmatchPattern() is used for multiple sequences matching, for example, when using sets. Each of these functions will return a different object as a result, but the match will be the same. You will see how in the next exercises.

## 5. Palindromes

Palindromes are sequences that read the same backwards as forwards. For example, "never odd or even" also reads "never odd or even" backwards. In biology, palindromes occur at sites highlighting important reactions (for example, binding sites and sites interrupted by restriction enzymes). Biostrings comes with a handy function called findPalindromes() for you to use. This function is used with a single sequence.

## 6. Not new biology

This chapter includes many biological concepts new for you, which we won't cover in detail. But, you can always learn more with targeted reading. The first is the genetic code, which is a table that describes which three RNA letters translate to one amino acid. The Genetic code was first described by Nirenberg in 1963. Then, how translation might differ according to the reading frame, was first described by Streisinger in 1966. The abstract introduces you to new terms and how a different sequence is translated depending on the start point.

## 7. Translation has six possibilities

This is a real example of how translation varies according to the start of the sequence, and how you can make sure to translate all possibilities. From a single DNA string, there are 6 possible string frames. 3 are positive strands and 3 negative strands. A negative strand is the reverse complement of a positive sequence strand. Because translation needs three bases for an amino acid, you get a completely different amino acid sequence, depending on where you start. That is why for translation, we move one base at the time. That is called a single base sliding window. As you can see, each DNA reading frame translates to a different amino acid sequence.

## 8. Conserved regions in the Zika virus

Now the super exciting part! coming back to our Zika virus example. You will be in charge of finding a very conserved sequence in the family of Flaviviruses, from which the Zika Virus is part of. First some facts: The Zika Virus has a positive strand genome. It can live in different host cells. For example, Humans, monkeys, and mosquitoes. The Flaviviruses family share a common structure, which means their sequences are very similar. The virus structure has only 11 proteins. In the last exercise, you'll be in search of one of these proteins using what you have learned so far!

## 1. Sequence Ranges

Nowadays, we can obtain billions of sequences for less than a thousand dollars. That means that large sequencing projects are in need of analysis, and that's where your knowledge of sequence analysis comes to play. Bioconductor packages provide convenient structures and function for representing, manipulating, and annotating genomic data. This chapter will walk you through GenomicRanges and how to use it to target specific sequences of interest.

## 2. IRanges with numeric arguments

Let's start with the IRanges package, which provides the fundamental infrastructure and operations for manipulating intervals of sequences. Keep in mind that we sequence data to understand its components, functions, structures, and how they evolve. First, we load the IRanges package, and then create myIRanges using the IRanges() function and defining the start and end. As shown in the output, this is an IRanges object with 1 range, starting at position 20, and ending at position 30. There are multiple ways to construct IRanges. You will learn the basic ways here, and then you can practice on your own.

## 3. More IRanges examples

We will create the same IRanges object using two different sets of arguments. The first example specifies start and width. The second specifies the start and end. The missing argument can be calculated using the equation shown below. Also, notice how values can be recycled to the length of the longest argument. In the second example, end = 30 is recycled twice, as there are 2 values to start ranges.

## 4. Rle - run length encoding

Another way to construct IRanges is by using a Rle definition. Rle stands for Run-length encoding. The Rle() function computes and stores the lengths and values of a vector or factor. Rle objects are general S4 containers, used to save long vectors with repetitions, more efficiently. For example, we have a vector of some numbers, the total length of the vector is 8. The Rle of this vector is stored in 5 runs because there are 3 elements of 2 and 2 elements of 3, which are repeated consecutively. This is quite useful to represent sequence ranges as very commonly they will have repetitions.

## 5. IRanges with logical vector

So far, you've learned how to create IRanges with numeric arguments defined as start, end, or width. You can also create IRanges using a logical vector to define which elements of a sequence will be kept or skipped. This example uses a vector of logical elements as the start of the range. The first two elements are skipped, hence starting at position 3, then selecting the two following elements positioned third and fourth in the sequence. This range has a width of 2. This is particularly useful when you want to skip elements of a sequence you will use the logical value FALSE. Even better, you can create this logical vector based on a condition.

## 6. IRanges with logical Rle

You can use the Rle definition to create an IRanges object with multiple ranges. The gi logical vector of 7 elements is converted to an Rle object. The resulting IRanges has two ranges. The first with 3 elements and the second with two elements corresponding to the elements equal TRUE in the Rle.

## 7. In summary

IRanges are hierarchical data structures and can contain metadata, this is quite useful to store genes, transcripts, polymorphisms, GC content, and more. To construct an IRanges object you can provide start, end or width as numeric vectors. Also, the start argument can be a logical vector or logical Rle. Remember Rle stands for Run length encoding and it uses storage efficient definition. IRanges arguments fill in the blanks of sequence length by using the equation width equals end minus start plus 1.

## 1. Gene of interest using Genomic Ranges

Let's start a scientific search for a gene of interest while learning more about genomic ranges.

## 2. Examples of genomic intervals

When we work 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 instead of having only one sequence like with IRanges, we can have sets of sequences using genomic ranges. Additionally, biological relevant features are included as metadata in GRanges. Examples of genome intervals are reads aligned to a reference, genes of interest, exonic regions, SNPs, regions of transcription or binding sites, like those regions studied using RNA or ChIP seq.

## 3. Genomic Ranges

The GenomicRanges package has the class GRanges a type of container used to save genomic intervals per chromosome. The bare minimum arguments are chromosome name, start and end of the interval. You can define them using a character. The basic difference between IRanges and GRanges is that each range is associated with a chromosome and a strand. In addition to metadata per range, like score and GC percentage, GRanges also includes additional metadata such as interval names, using seqnames() sequence lengths, and genome, stored in seqinfo.

## 4. From data to GRanges

You will most likely have sequence intervals stored in a data frame, it can also be a tibble or another table-like structure. The first three columns are the minimum requirements to construct a GRanges object: chromosome or seqname, start and end. You can also add strand (which takes values positive, negative, unknown or missing) and associated metadata like scores and CG frequency. As in the example, you will transform a data frame using the function as(), providing as input a dataframe object and the class GRanges in quotes.

## 5. Genomic Ranges accessors

When using Genomic Ranges you can get, add, and update extra information using its accessors. To see a list of the available accessors, use the function methods and specify class GRanges. Out of the many accessors listed by methods a few very useful ones are: seqnames, used for chromosome names. ranges, which will return an IRanges object. mcols, to display additional metadata per range. seqinfo stores a summary of the sequence information . And genome stores the genome name. It is important to notice that most accessors are both setter and getter functions. Another important fact to highlight is that you can notice the reuse of some accessors between classes thanks to inheritance within S4 definitions.

## 6. Gene of interest: ABCD1

Let's now talk about our gene of interest, a gene with an easy name to remember, ABCD1. ABCD1 is located at the end of chromosome X long arm. It encodes a protein relevant for the well functioning of brain and lung cells in mammals. Chromosome X is about 156 million base pairs long and our gene is located in a small interval, around the 153 million base pairs mark.

## 7. Chromosome X GRanges

Let's now prepare our data to explore. We will use a human reference, version hg38, from the transcripts database, provided by UCSC and accessed through Genomic Features. We saved the data set into an object called hg. The human reference is about 3 billion bases long and since our gene of interest is located in chromosome X we can subset the reference using the genes function. We then add a filter argument set to a list named tx\_chrom equal to chrX in quotes. This returns a GRanges object with 983 genes to explore by gene\_id. There are other filters which you will use in the next exercises.

## 1. Manipulating collections of GRanges

Let's now explore Genomic Ranges functions, some of which will return a collection of Ranges called Genomic Ranges Lists.

## 2. GRangesList

The GRangesList class is a container, particularly efficient for storing a large collection of GRanges objects. To construct these special Lists, you can use the function as(), and give it a list to be converted into a GRangesList. You can also create a GRangesList by listing multiple GRranges objects. Inversely, to convert back to GRanges use the function unlist(). Finally, to find useful accessors use the methods() function with (class = "GRangesList") in quotes.

## 3. When to use lists?

You might ask yourself why would we use these lists? GRangesList serves to store compound features of a larger object, in which you can perform operations. Some examples of GRangesLists are: transcripts by gene, exons by transcripts, read alignments, and sliding windows.

## 4. Break a region into smaller regions

Sliding windows are useful to split a GRanges object into sub-elements. This function uses width and step parameters. Width is the total number of letters for each new range, and Step is the distance between ranges. This returns a GRangesList. In the example, each gene has been split into new ranges of width 20,000 bases, and the distance between ranges is 10,000 bases because of the step. Each range has an overlap of 10,000 bases because of width - step. In most cases, the last range will be shorter.

## 5. Genomic features and TxDb

Genes, transcripts, and exons are genomic features. The GenomicFeatures package retrieves and manages this information from providers like UCSC and BioMart. These annotated features are useful for ChIP-seq, RNA-seq and annotation analyses. GenomicFeatures uses transcript database-objects to store metadata, manage genomic locations, and relationships between features and its identifiers. Bioconductor provides built-in packages for the most used transcript databases. For the example, we will use the TxDb for known human genes version 38. Here is a trimmed output displaying the most important information of a TxDb object.

## 6. Genes, transcripts, exons

Let's now learn how to extract genomic features from a TxDb object. First, load the TxDb library and store the object. If you are interested only in a subset of chromosomes its recommended to pre-filter using seqlevels(). Notice, that this is not the only way of filtering. Here we will show two extracting functions: transcripts and exons. There are three others: genes, cds, and promoters). All of them receive a TxDb object and optional parameters columns and filter. Columns are to select column names, filter uses a condition on a column. Filter and columns receive a named list of vectors and the valid names are listed at the end of the slide.

## 7. Exons by transcripts

Each gene has one or more transcripts, and each transcript has a set of exons. To find the exons in this transcript, retrieve all the exons by transcript using the function exonsBy() where tx is short for a transcript. Then select the transcript, with id 179-161. The figure shows the exons on this transcript. Each purple region is an exon and in between exons we see introns. This transcript shows 10 exons and we see their widths as a numeric vector. Pretty neat!

## 8. Overlaps

To find genes of interest in a larger interval or a collection of intervals, you will use overlaps. Counting, finding and subsetting overlaps between objects containing genomic ranges are useful and fundamental to annotating genomic features. The following functions have been optimized for iterations. CountOverlaps(), findOverlaps() and subsetByOverlaps() need basically two objects to be compared - a query and subject, which are either a GRanges or GRangesList objects. Overlaps might be complete when the query matches completely, or partial if the match is a subset of the query.

## 1. Introducing ShortRead

Hey! welcome back, we will talk about two Bioconductor packages to explore sequence data quality! The examples used for this chapter are plant sequence files!

## 2. Plant genomes

Plant genomes are usually big datasets, so today we are going to explore a small genome model, Arabidopsis thaliana. This was the first plant species to be completely sequenced, having a genome size of 135 Megabase pairs.

## 3. Sequencing companies

We are living in the time, where large-scale DNA sequencing is used to answer biological questions revolving around gene expression, mutations, hereditary conditions, and more. Since the cost of sequencing is steadily decreasing, the volume of data is steadily increasing. Different technologies for sequencing are developed by various sequencing companies. In the next examples, we will work with Illumina sequences, as Illumina continues to cover about 50% of sequencing projects worldwide.

1. 1 Dan Koboldt massgenomics.org

## 4. fastq vs fasta

How do we store sequences? We do so, using text. There are two main text formats fastQ and fastA, the main difference is that fastQ files include quality encoding per sequenced letter. Both formats are used to store DNA or protein sequences together with sequence names. In detail, fastQ files are the standard for storing large-scale sequencing also called high-throughput sequencing. Each sequence read on a fastq file will be described in four lines. The first starts with an '@' sign and a sequence identifier or description. Then comes the raw sequence string. Followed by line 3 with a '+' sign and the sequence identifier. Finally, line 4 encodes the quality values of the sequence, with one encoding value per sequenced letter. Common file extensions are fastq or only fq. A fasta file contains two lines per sequence read. The first line starts with the right arrow and a unique sequence identifier and the second line, the raw sequence string. Common file extensions are fasta, fa, or seq.

## 5. fasta

ShortRead provides us with readFasta() which reads all FASTA-formatted files in a directory Path followed by a pattern. It can read compressed or uncompressed files. It returns a single object representation of class ShortRead. This class stores and manipulates uniform-length short read sequences and their identifiers. Use methods with class ShortRead to get a list of accessors. Lastly, writeFasta() writes an object to a single file given a file name. It can also compress on the fly.

## 6. fastq

Similarly, readFastq() reads all FASTQ-formatted files in a directory Path followed by a pattern. It functions like readFasta() with two additional arguments, qualityType and filter. Fastq files include sequence quality on the fourth line. The encoding of quality depends on the technology and the version used. Again, use methods() to see the available accessors of this class. writeFastq() writes an object of class ShortReadQ to a single file, additionally, it can append new sequences to an existing file and save a compressed version with the extension dot-gz. Be mindful to not forget adding the extension to the name.

## 7. fastq sample

Sequence files can hold from one sequence to millions of sequences! Often, you will like to work with a subset of these sequences. When sampling, usually is a good idea to set the seed to collect the same sample during re-runs. The function FastqSampler() draws a subset from a fastq file, with a given length, in this example 500 reads. Then, yield is a function to extract the sample from the stored file. The time of sampling might differ depending on the size of the file. After this, you can also explore other parameters and similar functions on your own, for example, length.

## 1. Sequence quality

You have learned that short reads can be stored in different formats. The most used is the fastq format, which contains, both the sequenced bases and their quality values. Now you are going to learn new functions from ShortRead to assess the quality of your sequenced reads.

## 2. Quality scores - Phred table

FASTQ files, store quality scores to register error probabilities, in other words, the accuracy of the sequencing, as shown in this Phread qualities table. Here you can see that scores equal to or above 30 are 99-point-9% accurate, hence are considered of good quality! Because only one base in 1000 might be wrong.

## 3. Encoding - Phred +33

Let's now have a detailed look at each encoding and the score that corresponds to it. Phred scores are represented efficiently with encoding, as a single ASCII character. Originally, they were used as Sanger FASTQ standard, and now are the de facto standard for representing sequencing quality in different platforms. The encoding translated to scores usually ranges from 2 to 40. However, you can see higher scores. You should also keep in mind that other encodings existed before this standard and varied depending on the sequencing platform.

## 4. fastq quality

fastq files encode quality scores on a class FastqQuality as a BStringSet using the function quality() you can obtain the Quality of a sequence. For this example, you have Illumina sequences using Phred+33 standard.

## 5. Exploring quality encoding and scores

\*\*Every\*\* sequence analysis should start with a quality assessment. Let's check the first read in the sequence using sread() Then we can see the quality of each letter with quality() which will give us the encoding values To transform the encoding into scores we can use the function PhredQuality() given the quality of the reads, then convert the pred quality pq into an Integer list to get numeric scores. Remember, a score of 30 is considered of good quality as it means the accuracy of base call is 99-point-9% From this example, with the exception of 3 base calls (with scores 29, 27, and 18), all of the other bases are of very good quality!

## 6. Quality assessment

The qa() function from ShortRead is super nifty because it gives you lots of summary assessments about your sequence file or files. qa() not only works with the files that you have already read, but it can also be used with files to be read given a directory path and a pattern. The Shortreadqqa class gives an overview of read-counts, base calls, read quality scores, base quality, most frequent sequences, and their distribution. You can call each of these by name using a list syntax call with double square brackets, to get a summary of each evaluation. Or for a bigger picture, use the function report() with browseURL)() to see a collection of graphs and tables in a browser.

## 7. Alphabet by cycle

Using accessors for ShortRead classes we can explore in detail the sequences. Let's have a look at the nucleotide frequency per cycle and some transformations for further analysis. First, take a look at the alphabet of the short read object and remember the DNA Biostrings alphabet. Then use the function alphabetbyCycle() and you will get all letter frequencies per cycle. You will subset only the first four rows as those are the nucleotides A, C, G, T, and then transpose them. Optionally convert this into a tibble and add cycle numbers, and voila! You have a ready to visualize tibble, of nucleotides by cycle.

## 1. Match and filter

This has been great so far. Let's now learn how to do matching and filtering at the time of reading big files. This will save you time and resources!

## 2. Duplicate sequences

Finding identical sequences in your file should trigger an alarm, especially if there are many duplicates. A few biological duplicates naturally occur in nature.However, another source of duplication is commonly resulting from PCR amplification in library preparation. Also when sequencing the same molecule more than once produces duplicates. If sequencing has errors or contamination you might find that 30% to 70% of your reads are identical copies of each other. If you are working with whole genome sequencing or exome sequencing, it is recommended to remove duplicates or at least mark them as they will have adverse effects in scaffolding and discovering large-scale genomic variations. With this in mind, you should set a threshold of acceptable duplicate percentage when working with RNA-seq of ChIP-seq because you do not want to get rid of biological relevant sequences.

## 3. srduplicated

This example contains 100% duplication. To check for duplicates, you can use the srduplicated() function which returns a logical value for each read in the file. TRUE means the read is duplicated. If you use this with table(), you get the counts of the duplicated reads. If of all the reads are duplicated, like in this example, this is a clear mistake. One way to clean these reads is by sub-setting all those reads that are marked as not duplicated with a condition in the vector. Finally, if you use table and srduplicated() on the clean reads you will get no duplicates!

## 4. Creating your own filters

What about your own filters? srfilter() is a function to construct your own personalized ShortRead filters. It accepts a single argument (in the example, fqsample) and returns a logical vector used to select elements of fqsample satisfying a condition. The filter is saved in an object for convenience, here called readWidthCutOff. Then srFilter() receives a function of x, followed by a condition. Extra parameters can be specified before calling the filter, for example, minWidth. The name of the filter is optional. Lastly, the filter is applied on fqsample using the filter to subset. A filter can even be called at the time of reading the fastq file with the parameter named filter.

## 5. nFilter

Now you know that you can create your own custom filters! But, I wanted you to also take advantage of some built-in filters from the ShortRead package. For example, nFilter() has a threshold parameter representing the maximum number of N's allowed on each read. The second parameter dot-name is optional and add a custom name to your filter. As in the previous exercise, you could use the filter as a subsetting function. Or, like in this example, you can use the filter directly when reading the fastq files. This is a very fast cleaning step.

## 6. idFilter and polynFilter

Similarly, if you know which ids are the ones you need you can use, for example, the idFilter() function. It uses a regular expression to select specific ids, also called sequence names. myFilterID will select those id's that contain :3:1. Other optional parameters are dot-name, fixed, and exclude, the last two logical parameters influence how the matching occurs. Another nice filter to include is the polynFilter(). Here, myfilterPolyA accepts a maximum of 10 continuous A's per sequence read and filters everything else. Like before, you can use any of these filters at the time of reading the file, hence only reading what you need (or use them as subsetting).

## 1. Multiple and parallel sequence quality assessment

When dealing with big files, we want to save time and resources. The Rqc package from Bioconductor is a Quality Control Tool for High-Throughput Sequencing Data. It performs parallel processing of entire files to help you assess their quality.

## 2. Rqc

The package Rqc from Bioconductor will give you a summary report of all your sequence files at once! It uses some of the basic packages that you have learned so far, like Biostrings, IRanges, and S4Vectors, in addition to packages that you will discover in the other Bioconductor courses, such as Rsamtools, GenomicAlignments, and GenomicFiles Its functionality is combined and flavored with CRAN packages like knitr, markdown, and ggplot2 to create a clean report with multiple graphics. In addition, it uses Rcpp for parallel processing.

## 3. rqcQA

The Rqc function for Quality Assessment is named rqcQA. It will receive, as input, the full location of the fastq files, either compressed or uncompressed. The resulting object is a list. Each item will store one of the input files quality assessment - you can check them using names(). For each of the files, you will get a RqcResultSet class object.

## 4. rqcQA arguments

One super useful argument for running rqcQA is workers, so you can read files in parallel using your available cores. If you have 8 cores in your computer, the recommended is to use up to 6 workers. This result will only save the quality assessment, not the files. If you want to see the quality assessment in a subset of the input, use the argument sample. In the example, we selected 500 reads. The recommendation is to set a seed before calling a sample. If your sequencing project was called "single-end" then you are set, because by default rqcQA treats all files as single-end. If you have "paired-end", it means you have two files per sample id. To build pairs use a numeric vector, like in the example. So the first two files are one pair and the second two are the next pair.

## 5. rqcReport and rqcResultSet

To create a Quality Control HTML Report use rqcReport(), the main difference between this and the previous report you learned from ShortRead is that it accepts a custom template, in R markdown for generating your custom reports. Then use browseURL() to show the report in a browser. Additionally, you can pick what parts of the rqcResultSet to display, by using accessors listed in the methods function with class rqcResultSet in quotes.

## 6. perFileInformation

This summary table is the output of the function perFileInformation() using qaRqc. It shows the files that you are going to explore in the last exercise. These were downloaded from the Sequence Read Archive (SRA). Each file has about 2 million reads. To share a QA report, you don't need to copy the files. You can save the results of the Quality assessment and share it.

## 7. Plot functions

Rqc has 12 plotting functions for you to use! These are listed here and they are easy to find because they all start with rqc and finish with Plot! A cool tip is to call a function individually and save it as a PDF. You will have a clean plot of your quality assessment!

## 8. rqcCycleBaseCallsLinePlot

I can't tell you how useful is this package without showing you any of the plots. Here is a line multi-plot of base calls distribution per cycle. This was created using the function rqcCycleBaseCallsLinePlot() with the QA object - only one line of code.

## 9. Keep calm

So what are you going to do the next time you get a hold of a ton of fastq files? KEEP CALM and use a parallel quality assessment!

## 1. Congratulations!

Congratulations for getting through the “Introduction to Bioconductor” course! You have made great progress in understanding the structure of biological elements by using some of the most used Bioconductor packages.

## 2. You learned...

In this course, you learned how to install packages from Bioconductor by using the BiocInstaller package. You learned basic techniques for reading, manipulating and filtering raw genomic data using BioStrings, GenomicRanges, and ShortRead. You worked with BSgenome and TxDb built-in datasets. Then used these to identify patterns by using matching functions. Finally, you were able to check the quality of sequence files using ShortRead and Rqc.

## 3. You explored

In addition to learning how to use these Bioconductor packages, you explored a variety of organisms like fungi, viruses, humans, and plants! The aim of demonstrating Bioconductor functionality with real examples is for you to reuse the concepts, containers, and functions, and apply those to your own biological datasets.

## 4. Keep learning!

Nonetheless, it was super fun walking through your first steps with this fantastic resource, the Bioconductor project. If you would like to continue your learning using R for bioinformatics, feel free to explore other Bioconductor courses at DataCamp.