MSc Applied Bioinformatics
alexey.larionov@cranfield.ac.uk

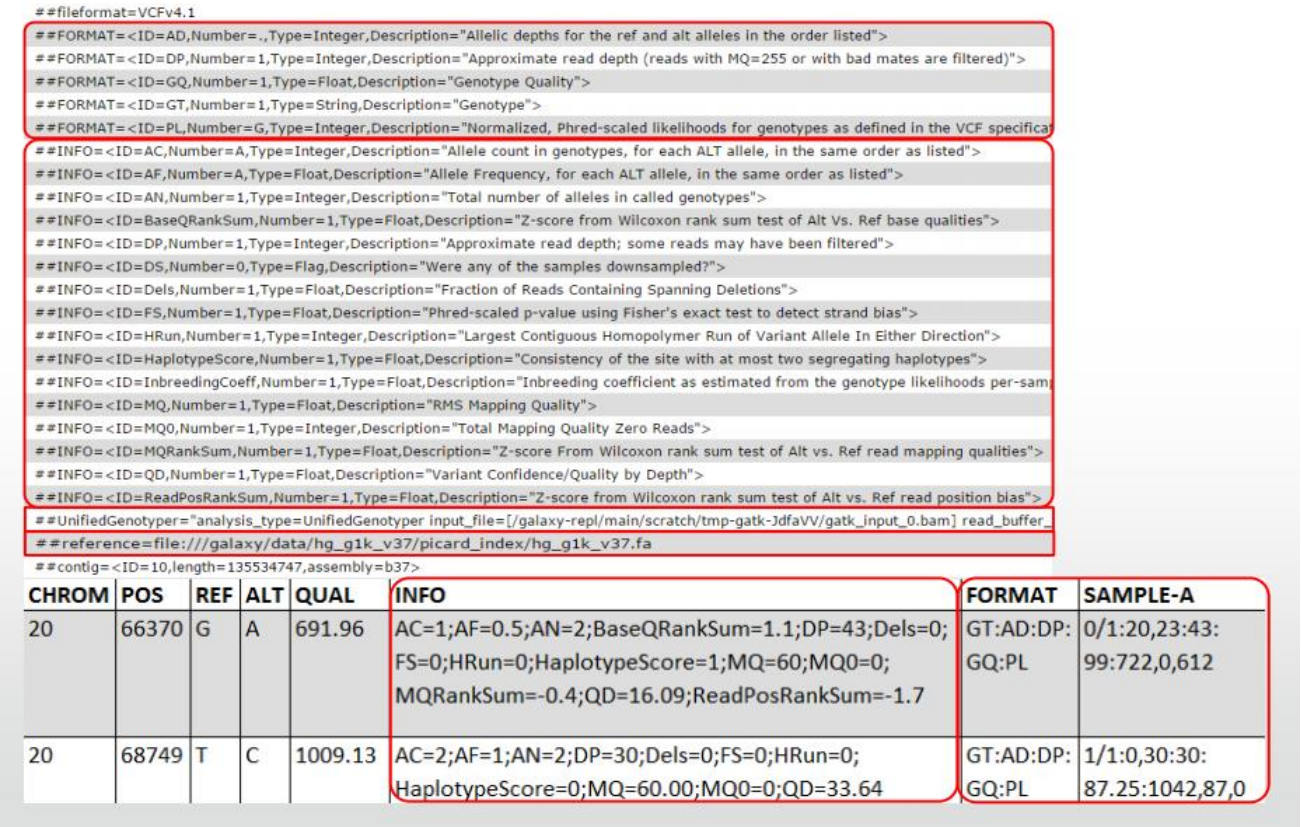
Practical 6: Handling VCF and XML data files

Apart from sequence data there are lots of other file formats that you may encounter in your journey through bioinformatics. In this practical we will consider two of those file formats: VCF and XML, writing Python code to extract and analyse data from these file types.

Task 1: Manipulating Variant Call Format files

Variant Call Format (**VCF**) is widely used in bioinformatics for recording variants (mutations) in the genes. The format was developed with the advent of large-scale DNA sequencing projects (such as the 1000 Genomes Project). VCF files store information about genetic variations observed in a sample relatively to a "reference genome". The "reference human genome" includes about 3 billion nucleotides. It is maintained by the Genome Reference Consortium (https://www.ncbi.nlm.nih.gov/grc); it is aimed to represent a genome of a "typical human" (if such human exists :) Each real human has just 5-10 million variations from the reference. VCF format provides a way to record these variants. Importantly, VCF files can store information about multiple samples in one file.

This is an example of a VCF file (with different sections highlighted by red frames):



```
##fileformat=VCFv4.1
##FORMAT=<ID=AD,Number=.,Type=Integer,Description="Allelic depths for the ref and alt alleles in the order listed">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Approximate read depth (reads with MQ=255 or with bad mates are filtered)">
##FORMAT=<ID=GQ,Number=1,Type=Float,Description="Genotype Quality">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=PL,Number=G,Type=Integer,Description="Normalized, Phred-scaled likelihoods for genotypes as defined in the VCF specifica
##INFO=<ID=AC,Number=A,Type=Integer,Description="Allele count in genotypes, for each ALT allele, in the same order as listed">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency, for each ALT allele, in the same order as listed">
##INFO=<ID=AN,Number=1,Type=Integer,Description="Total number of alleles in called genotypes">
##INFO=<ID=BaseQRankSum,Number=1,Type=Float,Description="Z-score from Wilcoxon rank sum test of Alt Vs. Ref base qualities">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Approximate read depth; some reads may have been filtered">
##INFO=<ID=DS,Number=0,Type=Flag,Description="Were any of the samples downsampled?">
##INFO=<ID=Dels,Number=1,Type=Float,Description="Fraction of Reads Containing Spanning Deletions">
##INFO=<ID=FS,Number=1,Type=Float,Description="Phred-scaled p-value using Fisher's exact test to detect strand bias">
##INFO=<ID=HRun,Number=1,Type=Integer,Description="Largest Contiguous Homopolymer Run of Variant Allele In Either Direction">
##INFO=<ID=HaplotypeScore,Number=1,Type=Float,Description="Consistency of the site with at most two segregating haplotypes">
##INFO=<ID=InbreedingCoeff,Number=1,Type=Float,Description="Inbreeding coefficient as estimated from the genotype likelihoods per-sam
##INFO=<ID=MQ,Number=1,Type=Float,Description="RMS Mapping Quality">
##INFO=<ID=MQ0,Number=1,Type=Integer,Description="Total Mapping Quality Zero Reads">
##INFO=<ID=MQRankSum,Number=1,Type=Float,Description="Z-score From Wilcoxon rank sum test of Alt vs. Ref read mapping qualities">
##INFO=<ID=QD,Number=1,Type=Float,Description="Variant Confidence/Quality by Depth">
##INFO=<ID=ReadPosRankSum,Number=1,Type=Float,Description="Z-score from Wilcoxon rank sum test of Alt vs. Ref read position bias">
##UnifiedGenotyper="analysis_type=UnifiedGenotyper input_file=[/galaxy-repl/main/scratch/tmp-gatk-JdfaVV/gatk_input_0.bam] read_buffer_
##reference=file:///galaxy/data/hg_g1k_v37/picard_index/hg_g1k_v37.fa
##contig=<ID=10,length=135534747,assembly=b37>
```

CHROM	POS	REF	ALT	QUAL	INFO	FORMAT	SAMPLE-A
20	66370	G	A	691.96	AC=1;AF=0.5;AN=2;BaseQRankSum=1.1;DP=43;Dels=0; FS=0;HRun=0;HaplotypeScore=1;MQ=60;MQ0=0; MQRankSum=-0.4;QD=16.09;ReadPosRankSum=-1.7	GT:AD:DP: GQ:PL	0/1:20,23:43: 99:722,0,612
20	68749	T	C	1009.13	AC=2;AF=1;AN=2;DP=30;Dels=0;FS=0;HRun=0; HaplotypeScore=0;MQ=60.00;MQ0=0;QD=33.64	GT:AD:DP: GQ:PL	1/1:0,30:30: 87.25:1042,87,0

Depending on the type of sequencing, for human data, VCF files may have a size from some Megabytes (for a targeted sequencing of a single individual), to many hundred of Gigabytes (for populational whole genome sequencing studes).

VCF file starts with a header that contains metadata (including information about the types of data included in the file, and about the used reference genome) followed by a table with actual information about the variants observed in the studied sample(s).

Typically, the header includes many keywords starting with ##. Some keywords are self-explanatory, such as **`fileformat`**, **`fileDate`** and **`reference`**. The other keywords describe the fields (columns) used in the body of the file later (the table with information about variants), e.g. **`INFO`**, **`FILTER`**, and **`FORMAT`**. The last line of the VCF header (starts with #) includes the column names for the variants table that follows the header.

The body of VCF follows the header. Each line corresponds to a variant and is tab separated. Each line includes 8 mandatory columns, plus an unlimited number of optional columns that may be used to record information about presence or absence of each variant the reported sample(s). When the additional columns are used, the first optional column is describes the format of the data in the "samples" columns that follow. You will explore this format in detail during the NGS modules later. If you are interested, you can find the current VCF specification here:
https://samtools.github.io/hts-specs/

Enough of the theory, let's do some programming. We will use a variant file (**`sample.cnv.chr11.vcf`**) of a patient with some abnormalities in chromosome 11. The VCF has been generated from a program called PINDEL, which detects large variants in the DNA sequence known as Copy Number Variation (or CNV). We will clean the VCF file and write it into a simple tab-delimited file, which you can share with your non-bioinformatician colleagues.

First, open the VCF file with any text editor, read the header and understand what different fields mean. By the way, if you prefer exploring tab-separated files in VS-Code, you may like to install ***Rainbow CSV*** extension.

In the new cleaned tab-delimited file we will exclude the VCF header and rearrange the information to get the output file like this:

CHROM	START	END	REF	ALT	FILTER	HOMLEN	HOMSEQ	SVLEN	SVTYPE	NTLEN
11	255986	255994	CATGGATGG	C	PASS	40	ATGGATGGATGGATGGATGGATGGATGGATGGATGG	8	DEL	
11	854730	854731	AG	A	PASS	4	GGGG	1	DEL	

Make a new Python file in your project folder. Then, open the VCF file:

```
vcf_file_name = r"Your File Location\sample.cnv.chr11.vcf"
vcf_file = open(vcf_file_name,'r')
```

Make the name for output file where you will write your output and open it for writing (in real life, you may provide the input and output file names using command line arguments):

```
txt_file_name = "File Location\\sample.cnv.chr11.txt"
```

Make and open the output file using 'w' for "write" (as we are going to write to the file):

```
txt_file = open(txt_file_name,'w')
```

Add the required header to the text file:

What is this?

```
txt_file.write("CHROM\tSTART\tEND\tREF\tALT\tFILT\tHOMLEN\t" + \
               "HOMSEQ\tSVLEN\tSVTYPE\tNTLEN\n")
```

Now, let's read the VCF file one line at a time:

```
for line in vcf_file:
```

To extract the required information, we don't need any header lines. So, we will ignore any line that starts with **"#"**

```
    if not line.startswith("#"): What is indentation in this line?
```

For each VCF line we need to initialise variables to capture the required data:

```
        end = '' # What is indentation in this line?
        homlen = ''
        homseq = ''
        svlen = ''
        svtype = ''
        ntlen = ''
```

Q: Why we need to initialise (only) these variables for each line?

Each line has new line character in the end, which we do not need in our output:

```
        l = line.rstrip()
```

Note the variable name "l" that I used in the line above. This is one of the worst variable names possible! It is not informative, and it is very easy to mix-up with the digit 1. Please, use another variable name in your script! (I will keep using "`line`" :)

The body of VCF file is TAB-delimited. So, split each line based on TAB (**"\t") :**

```
        line_split = line.split('\t')
```

Then extract information from the line, capturing the 8 mandatory VCF fields:

```
            chrom = line_split[0] # Indentation: 2 levels
            start = line_split[1]
            ref = line_split[3]
            alt = line_split[4]
            filt = line_split[6]
            info = line_split[7]
```

According to the VCF specification, the main VCF fields are delimited with TAB. However, the **INFO** field is additionally sub-delimited with semi-colons (**";"**). So, we should further split this field by **";"** :

```
            info_split = info.split(';')
```

Now, after the **INFO** field has been split, we will extract the attributes from it:

```
            for attr in info_split:
```

The attributes in INFO field are recorded as "Name=Value". So, to separate the names and values we should further split each attribute by "=" :

```
                attr_spl = attr.split('=') # Indentation: 3 levels
```

Then record the values to correct variables (note using "if" in one line):

```
                if attr_spl[0]=='END': end = attr_spl[1]
                if attr_spl[0]=='HOMLEN': homlen = attr_spl[1]
                if attr_spl[0]=='HOMSEQ': homseq = attr_spl[1]
                if attr_spl[0]=='SVLEN':svlen=attr_spl[1].strip('-')
                if attr_spl[0]=='SVTYPE': svtype = attr_spl[1]
                if attr_spl[0]=='NTLEN': ntlen = attr_spl[1]
```

Finally, write the extracted information to the output file:

What is this?

```
            txt_file.write(chrom+"\t"+start+"\t"+end+"\t"+ref+"\t"+ \
            alt+"\t"+filt+"\t"+homlen+"\t"+homseq+"\t"+svlen+"\t"+ \
            svtype+"\t"+ntlen+"\n") # Indentation: ? levels
```

Answers:

You should use two levels of indentation here (why ?)

Bac-slash could be used to split liong Python statement over several lines.

Q. What is missed in the script?

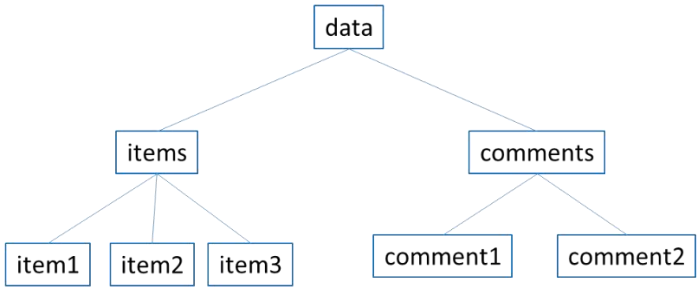
Exercise 6.1. Calculate the proportions of reads supporting alternative alleles from the VCF file (***sample.cnv.chr11.vcf***) and show the output as ***tab-separated*** file with two columns that look like this:

```
Variant    AAF
11:255986-255994    0.75
```

Where, "`11:255986-255994`" is the variant and `0.75` is the alternative allele fraction. Alternative allele fraction (AAF) is calculated from AD (alternative allele count) and RD (reference allele count): `AAF = AD/(RD + AD)`

Task 2: Handling XML file using Python

eXtensible Markup Language (XML) is a way of representing hierarchical heterogeneous data (see Lectures 1 and 6):




```
<data>
    <items>
        <item name="item1"></item>
        <item name="item2"></item>
        <item name="item3"></item>
    </items>
    <comments>
        <comment id="1"></comment>
        <comment id="2"></comment>
    </comments>
</data>
```

Strictly speaking, XML could be used to store any data. However, the simpler formats (like a tab-separated file) could be preferred for a data that is naturally represented by a table. XML was developed by the World Wide Web Consortium (W3C). You may think of XML as about a stricter version of HTML (if you are familiar with HTML :) However, XML is widely used for many tasks beyond the Internet applications. You may see more details about XML here https://en.wikipedia.org/wiki/XML (or in many other sources :) Luckily, you do not need to know all the details of the XML data format because Python has several well-developed libraries for dealing with this data type.

Have a look at the provided "***Q9JJE1.xml***" file to see how XML can be applied to store bioinformatics data. This XML file was retrieved from **UniProt** (one of the largest public databases for protein sequences). Make yourself familiar with the content of this file. In some sense, the content is self-explanatory.

MSc Applied Bioinformatics
alexey.larionov@cranfield.ac.uk

You can see that an XML file is very well structured. It starts with a short (optional) section that may be used by the programs that interpret XML files (so called XML parsers). In the ***"Q9JJE1.xml"*** file this section includes only one line with the XML version and with information about the character encoding (this section is called "prolog"). Often, XML files also contain a comment with a web link to the detailed description, explaining what elements should be expected later in the file (e.g. see the second line in the ***"pubmed_24009636.xml"*** file). Such descriptions are called "XML schemas" or "Document Type Definitions". For instance, see some XML schemes used by NCBI here: https://www.ncbi.nlm.nih.gov/data_specs/dtd/

The main body of the XML files is comprised of many "***elements***". An element is a piece of information enclosed into the ***start tag*** and the ***end tag*** (including these tags and all that lies in between). Example: <taxon>Eukaryota</taxon>

Elements can be **nested** inside each other.

Elements have **attributes**, which are placed within the start tag. In the example below, the underlined text is the element attribute:

```
<name type="scientific">Mus musculus</name>
```

Some elements may have more than one attribute, e.g.:

```
<sequence length="393" checksum="E0C0CC2E1F189B8A"> ...
```

During this practical session we will use `ElementTree` : a popular light-weight Python XML parser. There is a very similar another Python XML parser, called `cElementTree`. Initially, `cElementTree` was developed to improve the speed and memory efficiency by using C-language at the back-end. However, the recent versions of `ElementTree` provide the equivalent efficiency, so using `cElementTree` does not give any advantage any longer, and it will be deprecated in future. At the moment, you may use any of these parsers: the program interface of both parsers is the same.

A typical workflow of extracting data from an XML file includes two steps:

1. Parsing the file into an iterable object (list or *iterator*)
2. Looping over this list (or *iterator*) to extract the required data

In this practical we will compare two `ElementTree` functions to parse XML files:

`ElementTree.parse()` and `ElementTree.iterparse()`

First, we will use these functions to extract protein sequence from the ***"Q9JJE1.xml"*** file.

MSc Applied Bioinformatics
alexey.larionov@cranfield.ac.uk

Parsing XML with ElementTree.parse()

As usual, make a new Python script, add the script header, then import `ElementTree` library:

```
from xml.etree import ElementTree
```

Then open the XML file and get handle to its root using `ElementTree.parse()`:

```
tree = ElementTree.parse('Q9JJE1.xml')
root=tree.getroot()
```

The root object provides access to many useful functions allowing us to navigate within the XML file. One of the most used functions is `root.findall()`:

```
nodes = root.findall('./entry/sequence')
```

It searches for elements within within XML file, and returns them in a list that can be used in standard Python functions or in a loop:

```
print('Number of sequence elements in XML file:', len(nodes))

for node in nodes:
    print(node.tag, node.text)
```

Several points may need to be explained in the code above:

First, the design of the search term (`'./entry/sequence'`) assumes that we know the hierarchy of elements within the file, and we know the generic path to the elements we want to retrieve. In this case, we look for all “sequence” elements nested into “entry” elements, which in turn is directly nested in the root (designated as the dot in the search term).

Then, you can see that `ElementTree` also allows us to retrieve the name and the text of the elements by using `node.tag` and `node.text` respectively. Actually, the attributes are also accessible:

```
    print("Checksum:", node.attrib["checksum"])
    print("Length:", node.attrib["length"])
```

Save and execute the code.

Q1: What would be searched by this term './/sequence' ?

Q2: Can you print all taxa mentioned in the *“Q9JJE1.xml”* file ?

Parsing XML with ElementTree.iterparse()

Another popular function of `ElementTree` is called `iterparse`. It allows parsing XML into an "*iterator*" (an object similar to list) consisting of tuples (**event**, **element**).

By default, `iterparse` considers the end tags as the "**events**": retrieving the entire XML **element** each time when it detects an end tag. Other elements of XML syntax may be used for parsing too (e.g. start tags). However, for simplicity, we will only use the default "end" **events** (i.e. closing tags) here.

An interesting point about `iterparse` is that it returns an ***iterator*** instead of a ***list***. Iterators are widely used in Python. They go through the data source (e.g. through an XML file) and present the required elements one-by-one, without keeping all the elements in the memory at the same time. It saves memory, but you need to reinitialise the iterator if you wish go through the same data again! Alternatively, you can store the iterator into a list. However, this may be VERY memory consuming for large XML files.

Add a new file to your project folder, start with importing `ElementTree`:

```
from xml.etree import ElementTree
```

Initialise the `iterparse` *iterator*:

```
iter_parse = ElementTree.iterparse('Your File Path\Q9JJE1.xml')
```

Loop over the tuples (***event, element)*** produced by the `iterparse` *iterator*:

```
for event, elem in iter_parse:
```

Note that the line above not only loops over the tuples; it also unpacks each tuple into two variables, called "event" and "elem". Now we can print the "events", as well as the name, text, and the attributes for the element(s) named 'sequence':

```
    if elem.tag == 'sequence':
        print(event)
        print(elem.tag, elem.text)
        print('Checksum: ', elem.attrib["checksum"])
        print('Length: ', elem.attrib["length"])
```

Note that because the `iter_parse` is an *iterator*, you cannot use it again before you re-initialise it ! You may also note that because we used the default settings, all the "events" reported by `iterparse` will be equal to "end".

Q. Can you count the number of elements in the XML file?

Warning: don't forget about reinitialising the iterparse iterator!

We only scratched the surface of XML parsing in Python !

For instance:

Another way of looping through `ElementTree.parse` tree:
using `tree.iter` instead of `tree.getroot-root.findall`

```
from  xml.etree import ElementTree
tree = ElementTree.parse('Q9JJE1.xml')

for node in tree.iter():
     if node.tag == "sequence":
          print(node.tag,node.text)
          print(f'Checksum: {node.attrib["checksum"]}')
          print(f'Length: {node.attrib["length"]}')
```

Parsing XML with `xml.dom` package

```
from xml.dom import minidom
mydoc = minidom.parse('Q9JJE1.xml')
items = mydoc.getElementsByTagName('sequence')
for elem in items:
     print(elem.firstChild.data)
     print(elem.attributes['length'].value)
     print(elem.attributes['checksum'].value)
```

etc.

Exercise 6.2: *Parse the provided Pubmed XML file "**pubmed_24009636.xml**". This file contains bibliographic information about a single article from PubMed (PubMed is the largest public database for biomedical publications). Create a reference in the following style:*

Last name, First initial. <for all authors> (Year published)
Article title. Journal, Volume (Issue), Page(s).

Well done! You have finished Practical 6.