Biopython - Sequence

A sequence is series of letters used to represent an organism's protein, DNA or RNA. It is represented by Seq class. Seq class is defined in Bio.Seq module.

Let's create a simple sequence in Biopython as shown below −

```
>>> from Bio.Seq import Seq
>>> seq = Seq("AGCT")
>>> seq
Seq('AGCT')
>>> print(seq)
AGCT
```

Here, we have created a simple protein sequence **AGCT** and each letter represents **A**lanine, **G**lycine, **C**ysteine and **T**hreonine.

Each Seq object has two important attributes −

- data − the actual sequence string (AGCT)
- alphabet − used to represent the type of sequence. e.g. DNA sequence, RNA sequence, etc. By default, it does not represent any sequence and is generic in nature.

Alphabet Module

Seq objects contain Alphabet attribute to specify sequence type, letters and possible operations. It is defined in Bio.Alphabet module. Alphabet can be defined as below −

```
>>> from Bio.Seq import Seq
>>> myseq = Seq("AGCT")
>>> myseq
Seq('AGCT')
>>> myseq.alphabet
Alphabet()
```

Alphabet module provides below classes to represent different types of sequences. Alphabet - base class for all types of alphabets.

SingleLetterAlphabet - Generic alphabet with letters of size one. It derives from Alphabet and all other alphabets type derives from it.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import single_letter_alphabet
>>> test_seq = Seq('AGTACACTGGT', single_letter_alphabet)
```

```
>>> test_seq
Seq('AGTACACTGGT', SingleLetterAlphabet())
```

ProteinAlphabet − Generic single letter protein alphabet.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_protein
>>> test_seq = Seq('AGTACACTGGT', generic_protein)
>>> test_seq
Seq('AGTACACTGGT', ProteinAlphabet())
```

NucleotideAlphabet − Generic single letter nucleotide alphabet.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_nucleotide
>>> test_seq = Seq('AGTACACTGGT', generic_nucleotide) >>> test_seq
Seq('AGTACACTGGT', NucleotideAlphabet())
```

DNAAlphabet − Generic single letter DNA alphabet.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> test_seq = Seq('AGTACACTGGT', generic_dna)
>>> test_seq
Seq('AGTACACTGGT', DNAAlphabet())
```

RNAAlphabet − Generic single letter RNA alphabet.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_rna
>>> test_seq = Seq('AGTACACTGGT', generic_rna)
>>> test_seq
Seq('AGTACACTGGT', RNAAlphabet())
```

Biopython module, Bio.Alphabet.IUPAC provides basic sequence types as defined by IUPAC community. It contains the following classes −

- **IUPACProtein (protein)** − IUPAC protein alphabet of 20 standard amino acids.
- **ExtendedIUPACProtein (extended_protein)** − Extended uppercase IUPAC protein single letter alphabet including X.
- **IUPACAmbiguousDNA (ambiguous_dna)** − Uppercase IUPAC ambiguous DNA.
- **IUPACUnambiguousDNA (unambiguous_dna)** − Uppercase IUPAC unambiguous DNA (GATC).
- **ExtendedIUPACDNA (extended_dna)** − Extended IUPAC DNA alphabet.

- **IUPACAmbiguousRNA (ambiguous_rna)** − Uppercase IUPAC ambiguous RNA.
- **IUPACUnambiguousRNA (unambiguous_rna)** − Uppercase IUPAC unambiguous RNA (GAUC).

Consider a simple example for IUPACProtein class as shown below −

```
>>> from Bio.Alphabet import IUPAC
>>> protein_seq = Seq("AGCT", IUPAC.protein)
>>> protein_seq
Seq('AGCT', IUPACProtein())
>>> protein_seq.alphabet
```

Also, Biopython exposes all the bioinformatics related configuration data through Bio.Data module. For example, IUPACData.protein_letters has the possible letters of IUPACProtein alphabet.

```
>>> from Bio.Data import IUPACData
>>> IUPACData.protein_letters
'ACDEFGHIKLMNPQRSTVWY'
```

Basic Operations

This section briefly explains about all the basic operations available in the Seq class. Sequences are similar to python strings. We can perform python string operations like slicing, counting, concatenation, find, split and strip in sequences.

Use the below codes to get various outputs.

To get the first value in sequence.

```
>>> seq_string = Seq("AGCTAGCT")
>>> seq_string[0]
'A'
```

To print the first two values.

```
>>> seq_string[0:2]
Seq('AG')
```

To print all the values.

```
>>> seq_string[ : ]
Seq('AGCTAGCT')
```

To perform length and count operations.

```
>>> len(seq_string)
```

```
8
>>> seq_string.count('A')
2
```

To add two sequences.

```
>>> from Bio.Alphabet import generic_dna, generic_protein
>>> seq1 = Seq("AGCT", generic_dna)
>>> seq2 = Seq("TCGA", generic_dna)
>>> seq1+seq2
Seq('AGCTTCGA', DNAAlphabet())
```

Here, the above two sequence objects, seq1, seq2 are generic DNA sequences and so you can add them and produce new sequence. You can't add sequences with incompatible alphabets, such as a protein sequence and a DNA sequence as specified below −

```
>>> dna_seq = Seq('AGTACACTGGT', generic_dna)
>>> protein_seq = Seq('AGUACACUGGU', generic_protein)
>>> dna_seq + protein_seq
.....
.....
TypeError: Incompatible alphabets DNAAlphabet() and ProteinAlphabet()
>>>
```

To add two or more sequences, first store it in a python list, then retrieve it using 'for loop' and finally add it together as shown below −

```
>>> from Bio.Alphabet import generic_dna
>>> list = [Seq("AGCT",generic_dna),Seq("TCGA",generic_dna),Seq("AAA",generic_dna
>>> for s in list:
... print(s)
...
AGCT
TCGA
AAA
>>> final_seq = Seq(" ",generic_dna)
>>> for s in list:
... final_seq = final_seq + s
...
>>> final_seq
Seq('AGCTTCGAAAA', DNAAlphabet())
```

In the below section, various codes are given to get outputs based on the requirement.

To change the case of sequence.

```
>>> from Bio.Alphabet import generic_rna
```

```
>>> rna = Seq("agct", generic_rna)
>>> rna.upper()
Seq('AGCT', RNAAlphabet())
```

To check python membership and identity operator.

```
>>> rna = Seq("agct", generic_rna)
>>> 'a' in rna
True
>>> 'A' in rna
False
>>> rna1 = Seq("AGCT", generic_dna)
>>> rna is rna1
False
```

To find single letter or sequence of letter inside the given sequence.

```
>>> protein_seq = Seq('AGUACACUGGU', generic_protein)
>>> protein_seq.find('G')
1
>>> protein_seq.find('GG')
8
```

To perform splitting operation.

```
>>> protein_seq = Seq('AGUACACUGGU', generic_protein)
>>> protein_seq.split('A')
[Seq('', ProteinAlphabet()), Seq('GU', ProteinAlphabet()),
   Seq('C', ProteinAlphabet()), Seq('CUGGU', ProteinAlphabet())]
```

To perform strip operations in the sequence.

```
>>> strip_seq = Seq(" AGCT ")
>>> strip_seq
Seq(' AGCT ')
>>> strip_seq.strip()
Seq('AGCT')
```

Biopython - Advanced Sequence Operations

In this chapter, we shall discuss some of the advanced sequence features provided by Biopython.

Complement and Reverse Complement

Nucleotide sequence can be reverse complemented to get new sequence. Also, the complemented sequence can be reverse complemented to get the original sequence. Biopython provides two methods to do this functionality − **complement** and **reverse_complement**. The code for this is given below −

```
>>> from Bio.Alphabet import IUPAC
>>> nucleotide = Seq('TCGAAGTCAGTC', IUPAC.ambiguous_dna)
>>> nucleotide.complement()
Seq('AGCTTCAGTCAG', IUPACAmbiguousDNA())
>>>
```

Here, the complement() method allows to complement a DNA or RNA sequence. The reverse_complement() method complements and reverses the resultant sequence from left to right. It is shown below −

```
>>> nucleotide.reverse_complement()
Seq('GACTGACTTCGA', IUPACAmbiguousDNA())
```

Biopython uses the ambiguous_dna_complement variable provided by Bio.Data.IUPACData to do the complement operation.

```
>>> from Bio.Data import IUPACData
>>> import pprint
>>> pprint.pprint(IUPACData.ambiguous_dna_complement) {
   'A': 'T',
   'B': 'V',
   'C': 'G',
   'D': 'H',
   'G': 'C',
   'H': 'D',
   'K': 'M',
   'M': 'K',
   'N': 'N',
   'R': 'Y',
   'S': 'S',
   'T': 'A',
   'V': 'B',
```

```
   'W': 'W',
   'X': 'X',
   'Y': 'R'}
>>>
```

GC Content

Genomic DNA base composition (GC content) is predicted to significantly affect genome functioning and species ecology. The GC content is the number of GC nucleotides divided by the total nucleotides.

To get the GC nucleotide content, import the following module and perform the following steps −

```
>>> from Bio.SeqUtils import GC
>>> nucleotide = Seq("GACTGACTTCGA",IUPAC.unambiguous_dna)
>>> GC(nucleotide)
50.0
```

Transcription

Transcription is the process of changing DNA sequence into RNA sequence. The actual biological transcription process is performing a reverse complement (TCAG → CUGA) to get the mRNA considering the DNA as template strand. However, in bioinformatics and so in Biopython, we typically work directly with the coding strand and we can get the mRNA sequence by changing the letter T to U.

Simple example for the above is as follows −

```
>>> from Bio.Seq import Seq
>>> from Bio.Seq import transcribe
>>> from Bio.Alphabet import IUPAC
>>> dna_seq = Seq("ATGCCGATCGTAT",IUPAC.unambiguous_dna) >>> transcribe(dna_seq)
Seq('AUGCCGAUCGUAU', IUPACUnambiguousRNA())
>>>
```

To reverse the transcription, T is changed to U as shown in the code below −

```
>>> rna_seq = transcribe(dna_seq)
>>> rna_seq.back_transcribe()
Seq('ATGCCGATCGTAT', IUPACUnambiguousDNA())
```

To get the DNA template strand, reverse_complement the back transcribed RNA as given below −

```
>>> rna_seq.back_transcribe().reverse_complement()
Seq('ATACGATCGGCAT', IUPACUnambiguousDNA())
```

Translation

Translation is a process of translating RNA sequence to protein sequence. Consider a RNA sequence as shown below −

```
>>> rna_seq = Seq("AUGGCCAUUGUAAU",IUPAC.unambiguous_rna)
>>> rna_seq
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Now, apply translate() function to the code above −

```
>>> rna_seq.translate()
Seq('MAIV', IUPACProtein())
```

The above RNA sequence is simple. Consider RNA sequence, AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGA and apply translate() −

```
>>> rna = Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGA', IUPAC.unambiguous_rna)
>>> rna.translate()
Seq('MAIVMGR*KGAR', HasStopCodon(IUPACProtein(), '*'))
```

Here, the stop codons are indicated with an asterisk '*'.

It is possible in translate() method to stop at the first stop codon. To perform this, you can assign to_stop=True in translate() as follows −

```
>>> rna.translate(to_stop = True)
Seq('MAIVMGR', IUPACProtein())
```

Here, the stop codon is not included in the resulting sequence because it does not contain one.

Translation Table

The Genetic Codes page of the NCBI provides full list of translation tables used by Biopython. Let us see an example for standard table to visualize the code −

```
>>> from Bio.Data import CodonTable
>>> table = CodonTable.unambiguous_dna_by_name["Standard"]
>>> print(table)
Table 1 Standard, SGC0
   | T       | C       | A       | G       |
 --+---------+---------+---------+---------+--
 T | TTT F   | TCT S   | TAT Y   | TGT C   | T
 T | TTC F   | TCC S   | TAC Y   | TGC C   | C
 T | TTA L   | TCA S   | TAA Stop| TGA Stop| A
 T | TTG L(s)| TCG S   | TAG Stop| TGG W   | G
```

```
  --+---------+---------+---------+---------+--
  C | CTT L   | CCT P   | CAT H   | CGT R   | T
  C | CTC L   | CCC P   | CAC H   | CGC R   | C
  C | CTA L   | CCA P   | CAA Q   | CGA R   | A
  C | CTG L(s)| CCG P   | CAG Q   | CGG R   | G
  --+---------+---------+---------+---------+--
  A | ATT I   | ACT T   | AAT N   | AGT S   | T
  A | ATC I   | ACC T   | AAC N   | AGC S   | C
  A | ATA I   | ACA T   | AAA K   | AGA R   | A
  A | ATG M(s)| ACG T   | AAG K   | AGG R   | G
  --+---------+---------+---------+---------+--
  G | GTT V   | GCT A   | GAT D   | GGT G   | T
  G | GTC V   | GCC A   | GAC D   | GGC G   | C
  G | GTA V   | GCA A   | GAA E   | GGA G   | A
  G | GTG V   | GCG A   | GAG E   | GGG G   | G
  --+---------+---------+---------+---------+--
>>>
```

Biopython uses this table to translate the DNA to protein as well as to find the Stop codon.

Biopython - Sequence I/O Operations

Biopython provides a module, Bio.SeqIO to read and write sequences from and to a file (any stream) respectively. It supports nearly all file formats available in bioinformatics. Most of the software provides different approach for different file formats. But, Biopython consciously follows a single approach to present the parsed sequence data to the user through its SeqRecord object.

Let us learn more about SeqRecord in the following section.

SeqRecord

Bio.SeqRecord module provides SeqRecord to hold meta information of the sequence as well as the sequence data itself as given below −

- *seq* − It is an actual sequence.
- *id* − It is the primary identifier of the given sequence. The default type is string.
- *name* − It is the Name of the sequence. The default type is string.
- *description* − It displays human readable information about the sequence.
- *annotations* − It is a dictionary of additional information about the sequence.

The SeqRecord can be imported as specified below

```
from Bio.SeqRecord import SeqRecord
```

Let us understand the nuances of parsing the sequence file using real sequence file in the coming sections.

Parsing Sequence File Formats

This section explains about how to parse two of the most popular sequence file formats, **FASTA** and **GenBank**.

FASTA

FASTA is the most basic file format for storing sequence data. Originally, FASTA is a software package for sequence alignment of DNA and protein developed during the early evolution of Bioinformatics and used mostly to search the sequence similarity.

Biopython provides an example FASTA file and it can be accessed at https://github.com /biopython/biopython/blob/master/Doc/examples/ls_orchid.fasta.

Download and save this file into your Biopython sample directory as **'orchid.fasta'**.

Bio.SeqIO module provides parse() method to process sequence files and can be imported as follows −

```
from Bio.SeqIO import parse
```

parse() method contains two arguments, first one is file handle and second is file format.

```
>>> file = open('path/to/biopython/sample/orchid.fasta')
>>> for record in parse(file, "fasta"):
...     print(record.id)
...
gi|2765658|emb|Z78533.1|CIZ78533
gi|2765657|emb|Z78532.1|CCZ78532
..........
..........
gi|2765565|emb|Z78440.1|PPZ78440
gi|2765564|emb|Z78439.1|PBZ78439
>>>
```

Here, the parse() method returns an iterable object which returns SeqRecord on every iteration. Being iterable, it provides lot of sophisticated and easy methods and let us see some of the features.

next()

next() method returns the next item available in the iterable object, which we can be used to get the first sequence as given below −

```
>>> first_seq_record = next(SeqIO.parse(open('path/to/biopython/sample/orchid.fas
>>> first_seq_record.id 'gi|2765658|emb|Z78533.1|CIZ78533'
>>> first_seq_record.name 'gi|2765658|emb|Z78533.1|CIZ78533'
>>> first_seq_record.seq Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCG
>>> first_seq_record.description 'gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.
>>> first_seq_record.annotations
{}
>>>
```

Here, seq_record.annotations is empty because the FASTA format does not support sequence annotations.

list comprehension

We can convert the iterable object into list using list comprehension as given below

```
>>> seq_iter = SeqIO.parse(open('path/to/biopython/sample/orchid.fasta'),'fasta')
>>> all_seq = [seq_record for seq_record in seq_iter] >>> len(all_seq)
```

```
94
>>>
```

Here, we have used len method to get the total count. We can get sequence with maximum length as follows −

```
>>> seq_iter = SeqIO.parse(open('path/to/biopython/sample/orchid.fasta'),'fasta')
>>> max_seq = max(len(seq_record.seq) for seq_record in seq_iter)
>>> max_seq
789
>>>
```

We can filter the sequence as well using the below code −

```
>>> seq_iter = SeqIO.parse(open('path/to/biopython/sample/orchid.fasta'),'fasta')
>>> seq_under_600 = [seq_record for seq_record in seq_iter if len(seq_record.seq)
>>> for seq in seq_under_600:
...     print(seq.id)
...
gi|2765606|emb|Z78481.1|PIZ78481
gi|2765605|emb|Z78480.1|PGZ78480
gi|2765601|emb|Z78476.1|PGZ78476
gi|2765595|emb|Z78470.1|PPZ78470
gi|2765594|emb|Z78469.1|PHZ78469
gi|2765564|emb|Z78439.1|PBZ78439
>>>
```

Writing a collection of SqlRecord objects (parsed data) into file is as simple as calling the SeqIO.write method as below −

```
file = open("converted.fasta", "w)
SeqIO.write(seq_record, file, "fasta")
```

This method can be effectively used to convert the format as specified below −

```
file = open("converted.gbk", "w)
SeqIO.write(seq_record, file, "genbank")
```

GenBank

It is a richer sequence format for genes and includes fields for various kinds of annotations. Biopython provides an example GenBank file and it can be accessed at https://github.com /biopython/biopython/blob/master/Doc/examples/ls_orchid.fasta.

Download and save file into your Biopython sample directory as **'orchid.gbk'**

Since, Biopython provides a single function, parse to parse all bioinformatics format. Parsing

GenBank format is as simple as changing the format option in the parse method.

The code for the same has been given below −

```
>>> from Bio import SeqIO
>>> from Bio.SeqIO import parse
>>> seq_record = next(parse(open('path/to/biopython/sample/orchid.gbk'),'genbank'
>>> seq_record.id
'Z78533.1'
>>> seq_record.name
'Z78533'
>>> seq_record.seq Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...
>>> seq_record.description
'C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA'
>>> seq_record.annotations {
   'molecule_type': 'DNA',
   'topology': 'linear',
   'data_file_division': 'PLN',
   'date': '30-NOV-2006',
   'accessions': ['Z78533'],
   'sequence_version': 1,
   'gi': '2765658',
   'keywords': ['5.8S ribosomal RNA', '5.8S rRNA gene', 'internal transcribed spa
   'source': 'Cypripedium irapeanum',
   'organism': 'Cypripedium irapeanum',
   'taxonomy': [
      'Eukaryota',
      'Viridiplantae',
      'Streptophyta',
      'Embryophyta',
      'Tracheophyta',
      'Spermatophyta',
      'Magnoliophyta',
      'Liliopsida',
      'Asparagales',
      'Orchidaceae',
      'Cypripedioideae',
      'Cypripedium'],
   'references': [
      Reference(title = 'Phylogenetics of the slipper orchids (Cypripedioideae:
      Orchidaceae): nuclear rDNA ITS sequences', ...),
      Reference(title = 'Direct Submission', ...)
   ]
}
```

Biopython - Sequence Alignments

Sequence alignment is the process of arranging two or more sequences (of DNA, RNA or protein sequences) in a specific order to identify the region of similarity between them.

Identifying the similar region enables us to infer a lot of information like what traits are conserved between species, how close different species genetically are, how species evolve, etc. Biopython provides extensive support for sequence alignment.

Let us learn some of the important features provided by Biopython in this chapter −

Parsing Sequence Alignment

Biopython provides a module, Bio.AlignIO to read and write sequence alignments. In bioinformatics, there are lot of formats available to specify the sequence alignment data similar to earlier learned sequence data. Bio.AlignIO provides API similar to Bio.SeqIO except that the Bio.SeqIO works on the sequence data and Bio.AlignIO works on the sequence alignment data.

Before starting to learn, let us download a sample sequence alignment file from the Internet.

To download the sample file, follow the below steps −

Step 1 − Open your favorite browser and go to http://pfam.xfam.org/family/browse website. It will show all the Pfam families in alphabetical order.

Step 2 − Choose any one family having less number of seed value. It contains minimal data and enables us to work easily with the alignment. Here, we have selected/clicked PF18225 and it opens go to http://pfam.xfam.org/family/PF18225 and shows complete details about it, including sequence alignments.

Step 3 − Go to alignment section and download the sequence alignment file in Stockholm format (PF18225_seed.txt).

Let us try to read the downloaded sequence alignment file using Bio.AlignIO as below −

Import Bio.AlignIO module

```
>>> from Bio import AlignIO
```

Read alignment using read method. read method is used to read single alignment data available in the given file. If the given file contain many alignment, we can use parse method. parse method returns iterable alignment object similar to parse method in Bio.SeqIO module.

```
>>> alignment = AlignIO.read(open("PF18225_seed.txt"), "stockholm")
```

Print the alignment object.

```
>>> print(alignment)
SingleLetterAlphabet() alignment with 6 rows and 65 columns
MQNTPAERLPAIIEKAKSKHDINVWLLDRQGRDLLEQRVPAKVA...EGP B7RZ31_9GAMM/59-123
AKQRGIAGLEEWLHRLDHSEAIPIFLIDEAGKDLLEREVPADIT...KKP A0A0C3NPG9_9PROT/58-119
ARRHGQEYFQQWLERQPKKVKEQVFAVDQFGRELLGRPLPEDMA...KKP A0A143HL37_9GAMM/57-121
TRRHGPESFRFWLERQPVEARDRIYAIDRSGAEILDRPIPRGMA...NKP A0A0X3UC67_9GAMM/57-121
AINRNTQQLTQDLRAMPNWSLRFVYIVDRNNQDLLKRPLPPGIM...NRK B3PFT7_CELJU/62-126
AVNATEREFTERIRTLPHWARRNVFVLDSQGFEIFDRELPSPVA...NRT K4KEM7_SIMAS/61-125
>>>
```

We can also check the sequences (SeqRecord) available in the alignment as well as below −

```
>>> for align in alignment:
... print(align.seq)
...
MQNTPAERLPAIIEKAKSKHDINVWLLDRQGRDLLEQRVPAKVATVANQLRGRKRRAFARHREGP
AKQRGIAGLEEWLHRLDHSEAIPIFLIDEAGKDLLEREVPADITA---RLDRRREHGEHGVRKKP
ARRHGQEYFQQWLERQPKKVKEQVFAVDQFGRELLGRPLPEDMAPMLIALNYRNRESHAQVDKKP
TRRHGPESFRFWLERQPVEARDRIYAIDRSGAEILDRPIPRGMAPLFKVLSFRNREDQGLVNNKP
AINRNTQQLTQDLRAMPNWSLRFVYIVDRNNQDLLKRPLPPGIMVLAPRLTAKHPYDKVQDRNRK
AVNATEREFTERIRTLPHWARRNVFVLDSQGFEIFDRELPSPVADLMRKLDLDRPFKKLERKNRT
>>>
```

Multiple Alignments

In general, most of the sequence alignment files contain single alignment data and it is enough to use **read** method to parse it. In multiple sequence alignment concept, two or more sequences are compared for best subsequence matches between them and results in multiple sequence alignment in a single file.

If the input sequence alignment format contains more than one sequence alignment, then we need to use **parse** method instead of **read** method as specified below −

```
>>> from Bio import AlignIO
>>> alignments = AlignIO.parse(open("PF18225_seed.txt"), "stockholm")
>>> print(alignments)
<generator object parse at 0x000001CD1C7E0360>
>>> for alignment in alignments:
... print(alignment)
...
SingleLetterAlphabet() alignment with 6 rows and 65 columns
MQNTPAERLPAIIEKAKSKHDINVWLLDRQGRDLLEQRVPAKVA...EGP B7RZ31_9GAMM/59-123
AKQRGIAGLEEWLHRLDHSEAIPIFLIDEAGKDLLEREVPADIT...KKP A0A0C3NPG9_9PROT/58-119
ARRHGQEYFQQWLERQPKKVKEQVFAVDQFGRELLGRPLPEDMA...KKP A0A143HL37_9GAMM/57-121
```

```
TRRHGPESFRFWLERQPVEARDRIYAIDRSGAEILDRPIPRGMA...NKP A0A0X3UC67_9GAMM/57-121
AINRNTQQLTQDLRAMPNWSLRFVYIVDRNNQDLLKRPLPPGIM...NRK B3PFT7_CELJU/62-126
AVNATEREFTERIRTLPHWARRNVFVLDSQGFEIFDRELPSPVA...NRT K4KEM7_SIMAS/61-125
>>>
```

Here, parse method returns iterable alignment object and it can be iterated to get actual alignments.

Pairwise Sequence Alignment

Pairwise sequence alignment compares only two sequences at a time and provides best possible sequence alignments. **Pairwise** is easy to understand and exceptional to infer from the resulting sequence alignment.

Biopython provides a special module, **Bio.pairwise2** to identify the alignment sequence using pairwise method. Biopython applies the best algorithm to find the alignment sequence and it is par with other software.

Let us write an example to find the sequence alignment of two simple and hypothetical sequences using pairwise module. This will help us understand the concept of sequence alignment and how to program it using Biopython.

Step 1

Import the module **pairwise2** with the command given below −

```
>>> from Bio import pairwise2
```

Step 2

Create two sequences, seq1 and seq2 −

```
>>> from Bio.Seq import Seq
>>> seq1 = Seq("ACCGGT")
>>> seq2 = Seq("ACGT")
```

Step 3

Call method pairwise2.align.globalxx along with seq1 and seq2 to find the alignments using the below line of code −

```
>>> alignments = pairwise2.align.globalxx(seq1, seq2)
```

Here, **globalxx** method performs the actual work and finds all the best possible alignments in the given sequences. Actually, Bio.pairwise2 provides quite a set of methods which follows the below convention to find alignments in different scenarios.

```
<sequence alignment type>XY
```

Here, the sequence alignment type refers to the alignment type which may be *global* or *local. global* type is finding sequence alignment by taking entire sequence into consideration. local type is finding sequence alignment by looking into the subset of the given sequences as well. This will be tedious but provides better idea about the similarity between the given sequences.

- X refers to matching score. The possible values are x (exact match), m (score based on identical chars), d (user provided dictionary with character and match score) and finally c (user defined function to provide custom scoring algorithm).
- Y refers to gap penalty. The possible values are x (no gap penalties), s (same penalties for both sequences), d (different penalties for each sequence) and finally c (user defined function to provide custom gap penalties)

So, localds is also a valid method, which finds the sequence alignment using local alignment technique, user provided dictionary for matches and user provided gap penalty for both sequences.

```
>>> test_alignments = pairwise2.align.localds(seq1, seq2, blosum62, -10, -1)
```

Here, blosum62 refers to a dictionary available in the pairwise2 module to provide match score. -10 refers to gap open penalty and -1 refers to gap extension penalty.

Step 4

Loop over the iterable alignments object and get each individual alignment object and print it.

```
>>> for alignment in alignments:
... print(alignment)
...
('ACCGGT', 'A-C-GT', 4.0, 0, 6)
('ACCGGT', 'AC--GT', 4.0, 0, 6)
('ACCGGT', 'A-CG-T', 4.0, 0, 6)
('ACCGGT', 'AC-G-T', 4.0, 0, 6)
```

Step 5

Bio.pairwise2 module provides a formatting method, format_alignment to better visualize the result −

```
>>> from Bio.pairwise2 import format_alignment
>>> alignments = pairwise2.align.globalxx(seq1, seq2)
>>> for alignment in alignments:
... print(format_alignment(*alignment))
...
```

```
ACCGGT
| | ||
A-C-GT
   Score=4

ACCGGT
|| ||
AC--GT
   Score=4

ACCGGT
| || |
A-CG-T
   Score=4

ACCGGT
|| | |
AC-G-T
   Score=4

>>>
```

Biopython also provides another module to do sequence alignment, Align. This module provides a different set of API to simply the setting of parameter like algorithm, mode, match score, gap penalties, etc., A simple look into the Align object is as follows −

```
>>> from Bio import Align
>>> aligner = Align.PairwiseAligner()
>>> print(aligner)
Pairwise sequence aligner with parameters
   match score: 1.000000
   mismatch score: 0.000000
   target open gap score: 0.000000
   target extend gap score: 0.000000
   target left open gap score: 0.000000
   target left extend gap score: 0.000000
   target right open gap score: 0.000000
   target right extend gap score: 0.000000
   query open gap score: 0.000000
   query extend gap score: 0.000000
   query left open gap score: 0.000000
   query left extend gap score: 0.000000
   query right open gap score: 0.000000
   query right extend gap score: 0.000000
   mode: global
>>>
```

Support for Sequence Alignment Tools

Biopython provides interface to a lot of sequence alignment tools through Bio.Align.Applications module. Some of the tools are listed below −

- ClustalW
- MUSCLE
- EMBOSS needle and water

Let us write a simple example in Biopython to create sequence alignment through the most popular alignment tool, ClustalW.

Step 1 − Download the Clustalw program from http://www.clustal.org/download/current/ and install it. Also, update the system PATH with the “clustal” installation path.

Step 2 − import ClustalwCommanLine from module Bio.Align.Applications.

```
>>> from Bio.Align.Applications import ClustalwCommandline
```

Step 3 − Set cmd by calling ClustalwCommanLine with input file, opuntia.fasta available in Biopython package. https://raw.githubusercontent.com/biopython/biopython/master/Doc/examples/opuntia.fasta

```
>>> cmd = ClustalwCommandline("clustalw2",
infile="/path/to/biopython/sample/opuntia.fasta")
>>> print(cmd)
clustalw2 -infile=fasta/opuntia.fasta
```

Step 4 − Calling cmd() will run the clustalw command and give an output of the resultant alignment file, opuntia.aln.

```
>>> stdout, stderr = cmd()
```

Step 5 − Read and print the alignment file as below −

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("/path/to/biopython/sample/opuntia.aln", "clustal")
>>> print(align)
SingleLetterAlphabet() alignment with 7 rows and 906 columns
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA
gi|6273285|gb|AF191659.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA
gi|6273284|gb|AF191658.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA
gi|6273287|gb|AF191661.1|AF191
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA
gi|6273286|gb|AF191660.1|AF191
```

```
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA
gi|6273290|gb|AF191664.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA
gi|6273289|gb|AF191663.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA
gi|6273291|gb|AF191665.1|AF191
>>>
```

Biopython - Overview of BLAST

BLAST stands for **Basic Local Alignment Search Tool**. It finds regions of similarity between biological sequences. Biopython provides Bio.Blast module to deal with NCBI BLAST operation. You can run BLAST in either local connection or over Internet connection.

Let us understand these two connections in brief in the following section −

Running over Internet

Biopython provides Bio.Blast.NCBIWWW module to call the online version of BLAST. To do this, we need to import the following module −

```
>>> from Bio.Blast import NCBIWWW
```

NCBIWW module provides qblast function to query the BLAST online version, https://blast.ncbi.nlm.nih.gov/Blast.cgi . qblast supports all the parameters supported by the online version.

To obtain any help about this module, use the below command and understand the features −

```
>>> help(NCBIWWW.qblast)
Help on function qblast in module Bio.Blast.NCBIWWW:
qblast(
   program, database, sequence,
   url_base = 'https://blast.ncbi.nlm.nih.gov/Blast.cgi',
   auto_format = None,
   composition_based_statistics = None,
   db_genetic_code =  None,
   endpoints = None,
   entrez_query = '(none)',
   expect = 10.0,
   filter = None,
   gapcosts = None,
   genetic_code = None,
   hitlist_size = 50,
   i_thresh = None,
   layout = None,
   lcase_mask = None,
   matrix_name = None,
   nucl_penalty = None,
   nucl_reward = None,
   other_advanced = None,
   perc_ident = None,
```

```
    phi_pattern = None,
    query_file = None,
    query_believe_defline = None,
    query_from = None,
    query_to = None,
    searchsp_eff = None,
    service = None,
    threshold = None,
    ungapped_alignment = None,
    word_size = None,
    alignments = 500,
    alignment_view = None,
    descriptions = 500,
    entrez_links_new_window = None,
    expect_low = None,
    expect_high = None,
    format_entrez_query = None,
    format_object = None,
    format_type = 'XML',
    ncbi_gi = None,
    results_file = None,
    show_overview = None,
    megablast = None,
    template_type = None,
    template_length = None
)

    BLAST search using NCBI's QBLAST server or a cloud service provider.

    Supports all parameters of the qblast API for Put and Get.

    Please note that BLAST on the cloud supports the NCBI-BLAST Common
    URL API (http://ncbi.github.io/blast-cloud/dev/api.html).
    To use this feature, please set url_base to 'http://host.my.cloud.service.prov
    format_object = 'Alignment'. For more details, please see 8. Biopython – Overv

https://blast.ncbi.nlm.nih.gov/Blast.cgi?PAGE_TYPE = BlastDocs&DOC_TYPE = CloudBl

    Some useful parameters:

    - program blastn, blastp, blastx, tblastn, or tblastx (lower case)
    - database Which database to search against (e.g. "nr").
    - sequence The sequence to search.
    - ncbi_gi TRUE/FALSE whether to give 'gi' identifier.
    - descriptions Number of descriptions to show. Def 500.
    - alignments Number of alignments to show. Def 500.
    - expect An expect value cutoff. Def 10.0.
    - matrix_name Specify an alt. matrix (PAM30, PAM70, BLOSUM80, BLOSUM45).
```

```
- filter "none" turns off filtering. Default no filtering
- format_type "HTML", "Text", "ASN.1", or "XML". Def. "XML".
- entrez_query Entrez query to limit Blast search
- hitlist_size Number of hits to return. Default 50
- megablast TRUE/FALSE whether to use MEga BLAST algorithm (blastn only)
- service plain, psi, phi, rpsblast, megablast (lower case)

This function does no checking of the validity of the parameters
and passes the values to the server as is. More help is available at:
https://ncbi.github.io/blast-cloud/dev/api.html
```

Usually, the arguments of the qblast function are basically analogous to different parameters that you can set on the BLAST web page. This makes the qblast function easy to understand as well as reduces the learning curve to use it.

Connecting and Searching

To understand the process of connecting and searching BLAST online version, let us do a simple sequence search (available in our local sequence file) against online BLAST server through Biopython.

Step 1 − Create a file named **blast_example.fasta** in the Biopython directory and give the below sequence information as input

```
Example of a single sequence in FASTA/Pearson format:
>sequence A ggtaagtcctctagtacaaacacccccaatattgtgatataattaaaattatattcatat
tctgttgccagaaaaaacacttttaggctatattagagccatcttctttgaagcgttgtc

>sequence B ggtaagtcctctagtacaaacacccccaatattgtgatataattaaaattatattca
tattctgttgccagaaaaaacacttttaggctatattagagccatcttctttgaagcgttgtc
```

Step 2 − Import the NCBIWWW module.

```
>>> from Bio.Blast import NCBIWWW
```

Step 3 − Open the sequence file, **blast_example.fasta** using python IO module.

```
>>> sequence_data = open("blast_example.fasta").read()
>>> sequence_data
'Example of a single sequence in FASTA/Pearson format:\n\n\n> sequence
A\nggtaagtcctctagtacaaacacccccaatattgtgatataattaaaatt
atattcatat\ntctgttgccagaaaaaacacttttaggctatattagagccatcttctttg aagcgttgtc\n\n'
```

Step 4 − Now, call the qblast function passing sequence data as main parameter. The other parameter represents the database (nt) and the internal program (blastn).

```
>>> result_handle = NCBIWWW.qblast("blastn", "nt", sequence_data)
```

```
>>> result_handle
<_io.StringIO object at 0x000001EC9FAA4558>
```

blast_results holds the result of our search. It can be saved to a file for later use and also, parsed to get the details. We will learn how to do it in the coming section.

Step 5 − The same functionality can be done using Seq object as well rather than using the whole fasta file as shown below −

```
>>> from Bio import SeqIO
>>> seq_record = next(SeqIO.parse(open('blast_example.fasta'),'fasta'))
>>> seq_record.id
'sequence'
>>> seq_record.seq
Seq('ggtaagtcctctagtacaaacacccccaatattgtgatataattaaaattatat...gtc',
SingleLetterAlphabet())
```

Now, call the qblast function passing Seq object, record.seq as main parameter.

```
>>> result_handle = NCBIWWW.qblast("blastn", "nt", seq_record.seq)
>>> print(result_handle)
<_io.StringIO object at 0x000001EC9FAA4558>
```

BLAST will assign an identifier for your sequence automatically.

Step 6 − result_handle object will have the entire result and can be saved into a file for later usage.

```
>>> with open('results.xml', 'w') as save_file:
>>>   blast_results = result_handle.read()
>>>   save_file.write(blast_results)
```

We will see how to parse the result file in the later section.

Running Standalone BLAST

This section explains about how to run BLAST in local system. If you run BLAST in local system, it may be faster and also allows you to create your own database to search against sequences.

Connecting BLAST

In general, running BLAST locally is not recommended due to its large size, extra effort needed to run the software, and the cost involved. Online BLAST is sufficient for basic and advanced purposes. Of course, sometime you may be required to install it locally.

Consider you are conducting frequent searches online which may require a lot of time and high network volume and if you have proprietary sequence data or IP related issues, then installing it

locally is recommended.

To do this, we need to follow the below steps −

Step 1 − Download and install the latest blast binary using the given link − ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/

Step 2 − Download and unpack the latest and necessary database using the below link − ftp://ftp.ncbi.nlm.nih.gov/blast/db/

BLAST software provides lot of databases in their site. Let us download alu.n.gz file from the blast database site and unpack it into alu folder. This file is in FASTA format. To use this file in our blast application, we need to first convert the file from FASTA format into blast database format. BLAST provides makeblastdb application to do this conversion.

Use the below code snippet −

```
cd /path/to/alu
makeblastdb -in alu.n -parse_seqids -dbtype nucl -out alun
```

Running the above code will parse the input file, alu.n and create BLAST database as multiple files alun.nsq, alun.nsi, etc. Now, we can query this database to find the sequence.

We have installed the BLAST in our local server and also have sample BLAST database, **alun** to query against it.

Step 3 − Let us create a sample sequence file to query the database. Create a file search.fsa and put the below data into it.

```
>gnl|alu|Z15030_HSAL001056 (Alu-J)
AGGCTGGCACTGTGGCTCATGCTGAAATCCCAGCACGGCGGAGGACGGCGGAAGATTGCT
TGAGCCTAGGAGTTTGCGACCAGCCTGGGTGACATAGGGAGATGCCTGTCTCTACGCAAA
AGAAAAAAAAAATAGCTCTGCTGGTGGTGCATGCCTATAGTCTCAGCTATCAGGAGGCTG
GGACAGGAGGATCACTTGGGCCCGGGAGTTGAGGCTGTGGTGAGCCACGATCACACCACT
GCACTCCAGCCTGGGTGACAGAGCAAGACCCTGTCTCAAAACAAACAAATAA
>gnl|alu|D00596_HSAL003180 (Alu-Sx)
AGCCAGGTGTGGTGGCTCACGCCTGTAATCCCACCGCTTTGGGAGGCTGAGTCAGATCAC
CTGAGGTTAGGAATTTGGGACCAGCCTGGCCAACATGGCGACACCCCAGTCTCTACTAAT
AACACAAAAAATTAGCCAGGTGTGCTGGTGCATGTCTGTAATCCCAGCTACTCAGGAGGC
TGAGGCATGAGAATTGCTCACGAGGCGGAGGTTGTAGTGAGCTGAGATCGTGGCACTGTA
CTCCAGCCTGGCGACAGAGGGAGAACCCATGTCAAAAACAAAAAAAGACACCACCAAAGG
TCAAAGCATA
>gnl|alu|X55502_HSAL000745 (Alu-J)
TGCCTTCCCCATCTGTAATTCTGGCACTTGGGGAGTCCAAGGCAGGATGATCACTTATGC
CCAAGGAATTTGAGTACCAAGCCTGGGCAATATAACAAGGCCCTGTTTCTACAAAAACTT
TAAACAATTAGCCAGGTGTGGTGGTGCGTGCCTGTGTCCAGCTACTCAGGAAGCTGAGGC
AAGAGCTTGAGGCTACAGTGAGCTGTGTTCCACCATGGTGCTCCAGCCTGGGTGACAGGG
CAAGACCCTGTCAAAAGAAAGGAAGAAAGAACGGAAGGAAAGAAGGAAAGAAACAAGGAG
AG
```

The sequence data are gathered from the alu.n file; hence, it matches with our database.

Step 4 − BLAST software provides many applications to search the database and we use blastn. **blastn application requires minimum of three arguments, db, query and out. db** refers to the database against to search; **query** is the sequence to match and **out** is the file to store results. Now, run the below command to perform this simple query −

```
blastn -db alun -query search.fsa -out results.xml -outfmt 5
```

Running the above command will search and give output in the **results.xml** file as given below (partially data) −

```
<?xml version = "1.0"?>
<!DOCTYPE BlastOutput PUBLIC "-//NCBI//NCBI BlastOutput/EN"
   "http://www.ncbi.nlm.nih.gov/dtd/NCBI_BlastOutput.dtd">
<BlastOutput>
   <BlastOutput_program>blastn</BlastOutput_program>
   <BlastOutput_version>BLASTN 2.7.1+</BlastOutput_version>
   <BlastOutput_reference>Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb
      Miller (2000), "A greedy algorithm for aligning DNA sequences", J
      Comput Biol 2000; 7(1-2):203-14.
   </BlastOutput_reference>

   <BlastOutput_db>alun</BlastOutput_db>
   <BlastOutput_query-ID>Query_1</BlastOutput_query-ID>
   <BlastOutput_query-def>gnl|alu|Z15030_HSAL001056 (Alu-J)</BlastOutput_query-de
   <BlastOutput_query-len>292</BlastOutput_query-len>
   <BlastOutput_param>
      <Parameters>
         <Parameters_expect>10</Parameters_expect>
         <Parameters_sc-match>1</Parameters_sc-match>
         <Parameters_sc-mismatch>-2</Parameters_sc-mismatch>
         <Parameters_gap-open>0</Parameters_gap-open>
         <Parameters_gap-extend>0</Parameters_gap-extend>
         <Parameters_filter>L;m;</Parameters_filter>
      </Parameters>
   </BlastOutput_param>
   <BlastOutput_iterations>
      <Iteration>
         <Iteration_iter-num>1</Iteration_iter-num><Iteration_query-ID>Query_1</I
         <Iteration_query-def>gnl|alu|Z15030_HSAL001056 (Alu-J)</Iteration_query-
         <Iteration_query-len>292</Iteration_query-len>
         <Iteration_hits>
         <Hit>
            <Hit_num>1</Hit_num>
            <Hit_id>gnl|alu|Z15030_HSAL001056</Hit_id>
            <Hit_def>(Alu-J)</Hit_def>
```

```
      <Hit_accession>Z15030_HSAL001056</Hit_accession>
      <Hit_len>292</Hit_len>
      <Hit_hsps>
         <Hsp>
           <Hsp_num>1</Hsp_num>
            <Hsp_bit-score>540.342</Hsp_bit-score>
            <Hsp_score>292</Hsp_score>
            <Hsp_evalue>4.55414e-156</Hsp_evalue>
            <Hsp_query-from>1</Hsp_query-from>
            <Hsp_query-to>292</Hsp_query-to>
            <Hsp_hit-from>1</Hsp_hit-from>
            <Hsp_hit-to>292</Hsp_hit-to>
            <Hsp_query-frame>1</Hsp_query-frame>
            <Hsp_hit-frame>1</Hsp_hit-frame>
            <Hsp_identity>292</Hsp_identity>
            <Hsp_positive>292</Hsp_positive>
            <Hsp_gaps>0</Hsp_gaps>
            <Hsp_align-len>292</Hsp_align-len>

            <Hsp_qseq>
               AGGCTGGCACTGTGGCTCATGCTGAAATCCCAGCACGGCGGAGGACGGCGGAAGATTGCT
               CGACCAGCCTGGGTGACATAGGGAGATGCCTGTCTCTACGCAAAAGAAAAAAAAAATAGC
               CCTATAGTCTCAGCTATCAGGAGGCTGGGACAGGAGGATCACTTGGGCCCGGGAGTTGAG
               ACGATCACACCACTGCACTCCAGCCTGGGTGACAGAGCAAGACCCTGTCTCAAAACAAAC
            </Hsp_qseq>

            <Hsp_hseq>
               AGGCTGGCACTGTGGCTCATGCTGAAATCCCAGCACGGCGGAGGACGGCGGAAGATTGCT
               GTTTGCGACCAGCCTGGGTGACATAGGGAGATGCCTGTCTCTACGCAAAAGAAAAAAAAA
               GGTGGTGCATGCCTATAGTCTCAGCTATCAGGAGGCTGGGACAGGAGGATCACTTGGGCC
               CTGTGGTGAGCCACGATCACACCACTGCACTCCAGCCTGGGTGACAGAGCAAGACCCTGT
               AAATAA
            </Hsp_hseq>

            <Hsp_midline>
               ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
               ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
               ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
               ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
               ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
               ||||||||||||||||||||||||||||
            </Hsp_midline>
         </Hsp>
      </Hit_hsps>
   </Hit>
   ........................
   ........................
   ........................
```

```
         </Iteration_hits>
         <Iteration_stat>
            <Statistics>
               <Statistics_db-num>327</Statistics_db-num>
               <Statistics_db-len>80506</Statistics_db-len>
               <Statistics_hsp-lenv16</Statistics_hsp-len>
               <Statistics_eff-space>21528364</Statistics_eff-space>
               <Statistics_kappa>0.46</Statistics_kappa>
               <Statistics_lambda>1.28</Statistics_lambda>
               <Statistics_entropy>0.85</Statistics_entropy>
            </Statistics>
         </Iteration_stat>
      </Iteration>
   </BlastOutput_iterations>
</BlastOutput>
```

The above command can be run inside the python using the below code −

```
>>> from Bio.Blast.Applications import NcbiblastnCommandline
>>> blastn_cline = NcbiblastnCommandline(query = "search.fasta", db = "alun",
outfmt = 5, out = "results.xml")
>>> stdout, stderr = blastn_cline()
```

Here, the first one is a handle to the blast output and second one is the possible error output generated by the blast command.

Since we have provided the output file as command line argument (out = “results.xml”) and sets the output format as XML (outfmt = 5), the output file will be saved in the current working directory.

Parsing BLAST Result

Generally, BLAST output is parsed as XML format using the NCBIXML module. To do this, we need to import the following module −

```
>>> from Bio.Blast import NCBIXML
```

Now, **open the file directly using python open method** and **use NCBIXML parse method** as given below −

```
>>> E_VALUE_THRESH = 1e-20
>>> for record in NCBIXML.parse(open("results.xml")):
>>>     if record.alignments:
>>>        print("\n")
>>>        print("query: %s" % record.query[:100])
>>>        for align in record.alignments:
>>>           for hsp in align.hsps:
>>>              if hsp.expect < E_VALUE_THRESH:
```

```
>>>                 print("match: %s " % align.title[:100])
```

This will produce an output as follows −

```
query: gnl|alu|Z15030_HSAL001056 (Alu-J)
match: gnl|alu|Z15030_HSAL001056 (Alu-J)
match: gnl|alu|L12964_HSAL003860 (Alu-J)
match: gnl|alu|L13042_HSAL003863 (Alu-FLA?)
match: gnl|alu|M86249_HSAL001462 (Alu-FLA?)
match: gnl|alu|M29484_HSAL002265 (Alu-J)

query: gnl|alu|D00596_HSAL003180 (Alu-Sx)
match: gnl|alu|D00596_HSAL003180 (Alu-Sx)
match: gnl|alu|J03071_HSAL001860 (Alu-J)
match: gnl|alu|X72409_HSAL005025 (Alu-Sx)

query: gnl|alu|X55502_HSAL000745 (Alu-J)
match: gnl|alu|X55502_HSAL000745 (Alu-J)
```