**Course: Advanced Bioinformatics**

**Module title: Sequence Object**

**Module no. : 203**

In this module, sequence objects are discussed.Biological sequences are arguably the central object in Bioinformatics, and in this module we’ll introduce the Biopython mechanism for dealing with sequences, the Seq object. Sequences are essentially strings of letters like AGTACACTGGT, which seems very natural since this is the most common way that sequences are seen in biological file formats.

There are two important differences between Seq objects and standard Python strings. First of all, they have different methods. Although the Seq object supports many of the same methods as a plain string, its translate() method differs by doing biological translation, and there are also additional biologically relevant methods like reverse\_complement(). Secondly, the Seq object has an important attribute, alphabet, which is an object describing what the individual characters making up the sequence string “mean”, and how they should be interpreted. For example, isAGTACACTGGT a DNA sequence, or just a protein sequence that happens to be rich in Alanines, Glycines, Cysteines and Threonines?

**Sequences and Alphabets**

The alphabet object is perhaps the important thing that makes the Seq object more than just a string. The currently available alphabets for Biopython are defined in the Bio.Alphabet module. We’ll use the IUPAC alphabets (<http://www.chem.qmw.ac.uk/iupac/>) here to deal with some of our favorite objects: DNA, RNA and Proteins.

Bio.Alphabet.IUPAC provides basic definitions for proteins, DNA and RNA, but additionally provides the ability to extend and customize the basic definitions. For instance, for proteins, there is a basic IUPACProtein class, but there is an additional ExtendedIUPACProtein class providing for the additional elements “U” (or “Sec” for selenocysteine) and “O” (or “Pyl” for pyrrolysine), plus the ambiguous symbols “B” (or “Asx” for asparagine or aspartic acid), “Z” (or “Glx” for glutamine or glutamic acid), “J” (or “Xle” for leucine isoleucine) and “X” (or “Xxx” for an unknown amino acid). For DNA you’ve got choices of IUPACUnambiguousDNA, which provides for just the basic letters, IUPACAmbiguousDNA (which provides for ambiguity letters for every possible situation) and ExtendedIUPACDNA, which allows letters for modified bases. Similarly, RNA can be represented by IUPACAmbiguousRNA or IUPACUnambiguousRNA.

The advantages of having an alphabet class are two fold. First, this gives an idea of the type of information the Seq object contains. Secondly, this provides a means of constraining the information, as a means of type checking.

Now that we know what we are dealing with, let’s look at how to utilize this class to do interesting work. You can create an ambiguous sequence with the default generic alphabet like this:

>>> from Bio.Seq import Seq

>>> my\_seq = Seq("AGTACACTGGT")

>>> my\_seq

Seq('AGTACACTGGT', Alphabet())

>>> my\_seq.alphabet

Alphabet()

However, where possible you should specify the alphabet explicitly when creating your sequence objects - in this case an unambiguous DNA alphabet object:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> my\_seq = Seq("AGTACACTGGT", IUPAC.unambiguous\_dna)

>>> my\_seq

Seq('AGTACACTGGT', IUPACUnambiguousDNA())

>>> my\_seq.alphabet

IUPACUnambiguousDNA()

Unless of course, this really is an amino acid sequence:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> my\_prot = Seq("AGTACACTGGT", IUPAC.protein)

>>> my\_prot

Seq('AGTACACTGGT', IUPACProtein())

>>> my\_prot.alphabet

IUPACProtein()

**Sequences act like strings**

In many ways, we can deal with Seq objects as if they were normal Python strings, for example getting the length, or iterating over the elements:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> my\_seq = Seq("GATCG", IUPAC.unambiguous\_dna)

>>> for index, letter in enumerate(my\_seq):

... print("%i %s" % (index, letter))

0 G

1 A

2 T

3 C

4 G

>>> print(len(my\_seq))

5

You can access elements of the sequence in the same way as for strings (but remember, Python counts from zero!):

>>> print(my\_seq[0]) #first letter

G

>>> print(my\_seq[2]) #third letter

T

>>> print(my\_seq[-1]) #last letter

G

The Seq object has a .count() method, just like a string. Note that this means that like a Python string, this gives a *non-overlapping* count:

>>> from Bio.Seq import Seq

>>> "AAAA".count("AA")

2

>>> Seq("AAAA").count("AA")

2

For some biological uses, you may actually want an overlapping count (i.e. 3 in this trivial example). When searching for single letters, this makes no difference:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> my\_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPAC.unambiguous\_dna)

>>> len(my\_seq)

32

>>> my\_seq.count("G")

9

>>> 100 \* float(my\_seq.count("G") + my\_seq.count("C")) / len(my\_seq)

46.875

While you could use the above snippet of code to calculate a GC%, note that the Bio.SeqUtils module has several GC functions already built. For example:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> from Bio.SeqUtils import GC

>>> my\_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPAC.unambiguous\_dna)

>>> GC(my\_seq)

46.875

Note that using the Bio.SeqUtils.GC() function should automatically cope with mixed case sequences and the ambiguous nucleotide S which means G or C.

**Slicing a sequence**

A more complicated example, let’s get a slice of the sequence:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> my\_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous\_dna)

>>> my\_seq[4:12]

Seq('GATGGGCC', IUPACUnambiguousDNA())

Two things are interesting to note. First, this follows the normal conventions for Python strings. So the first element of the sequence is 0 (which is normal for computer science, but not so normal for biology). When you do a slice the first item is included (i.e. 4 in this case) and the last is excluded (12 in this case), which is the way things work in Python, but of course not necessarily the way everyone in the world would expect. The main goal is to stay consistent with what Python does.

The second thing to notice is that the slice is performed on the sequence data string, but the new object produced is another Seq object which retains the alphabet information from the original Seq object.

Also like a Python string, you can do slices with a start, stop and *stride* (the step size, which defaults to one). For example, we can get the first, second and third codon positions of this DNA sequence:

>>> my\_seq[0::3]

Seq('GCTGTAGTAAG', IUPACUnambiguousDNA())

>>> my\_seq[1::3]

Seq('AGGCATGCATC', IUPACUnambiguousDNA())

>>> my\_seq[2::3]

Seq('TAGCTAAGAC', IUPACUnambiguousDNA())

Another stride trick you might have seen with a Python string is the use of a -1 stride to reverse the string. You can do this with a Seq object too:

>>> my\_seq[::-1]

Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())

**Turning Seq objects into strings**

If you really do just need a plain string, for example to write to a file, or insert into a database, then this is very easy to get:

>>> str(my\_seq)

'GATCGATGGGCCTATATAGGATCGAAAATCGC'

Since calling str() on a Seq object returns the full sequence as a string, you often don’t actually have to do this conversion explicitly. Python does this automatically in the print function (and the print statement under Python 2):

>>> print(my\_seq)

GATCGATGGGCCTATATAGGATCGAAAATCGC

You can also use the Seq object directly with a %s placeholder when using the Python string formatting or interpolation operator (%):

>>> fasta\_format\_string = ">Name\n%s\n" % my\_seq

>>> print(fasta\_format\_string)

>Name

GATCGATGGGCCTATATAGGATCGAAAATCGC

<BLANKLINE>

This line of code constructs a simple FASTA format record (without worrying about line wrapping).

>>> str(my\_seq)

'GATCGATGGGCCTATATAGGATCGAAAATCGC'

**3.5  Concatenating or adding sequences**

Naturally, you can in principle add any two Seq objects together - just like you can with Python strings to concatenate them. However, you can’t add sequences with incompatible alphabets, such as a protein sequence and a DNA sequence:

>>> from Bio.Alphabet import IUPAC

>>> from Bio.Seq import Seq

>>> protein\_seq = Seq("EVRNAK", IUPAC.protein)

>>> dna\_seq = Seq("ACGT", IUPAC.unambiguous\_dna)

>>> protein\_seq + dna\_seq

Traceback (most recent call last):

...

TypeError: Incompatible alphabets IUPACProtein() and IUPACUnambiguousDNA()

If you *really* wanted to do this, you’d have to first give both sequences generic alphabets:

>>> from Bio.Alphabet import generic\_alphabet

>>> protein\_seq.alphabet = generic\_alphabet

>>> dna\_seq.alphabet = generic\_alphabet

>>> protein\_seq + dna\_seq

Seq('EVRNAKACGT', Alphabet())

Here is an example of adding a generic nucleotide sequence to an unambiguous IUPAC DNA sequence, resulting in an ambiguous nucleotide sequence:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import generic\_nucleotide

>>> from Bio.Alphabet import IUPAC

>>> nuc\_seq = Seq("GATCGATGC", generic\_nucleotide)

>>> dna\_seq = Seq("ACGT", IUPAC.unambiguous\_dna)

>>> nuc\_seq

Seq('GATCGATGC', NucleotideAlphabet())

>>> dna\_seq

Seq('ACGT', IUPACUnambiguousDNA())

>>> nuc\_seq + dna\_seq

Seq('GATCGATGCACGT', NucleotideAlphabet())

You may often have many sequences to add together, which can be done with a for loop like this:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import generic\_dna

>>> list\_of\_seqs = [Seq("ACGT", generic\_dna), Seq("AACC", generic\_dna), Seq("GGTT", generic\_dna)]

>>> concatenated = Seq("", generic\_dna)

>>> for s in list\_of\_seqs:

... concatenated += s

...

>>> concatenated

Seq('ACGTAACCGGTT', DNAAlphabet())

Or, a more elegant approach is to the use built in sum function with its optional start value argument (which otherwise defaults to zero):

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import generic\_dna

>>> list\_of\_seqs = [Seq("ACGT", generic\_dna), Seq("AACC", generic\_dna), Seq("GGTT", generic\_dna)]

>>> sum(list\_of\_seqs, Seq("", generic\_dna))

Seq('ACGTAACCGGTT', DNAAlphabet())

Unlike the Python string, the Biopython Seq does not (currently) have a .join method.

**3.6  Changing case**

Python strings have very useful upper and lower methods for changing the case. As of Biopython 1.53, the Seq object gained similar methods which are alphabet aware. For example,

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import generic\_dna

>>> dna\_seq = Seq("acgtACGT", generic\_dna)

>>> dna\_seq

Seq('acgtACGT', DNAAlphabet())

>>> dna\_seq.upper()

Seq('ACGTACGT', DNAAlphabet())

>>> dna\_seq.lower()

Seq('acgtacgt', DNAAlphabet())

These are useful for doing case insensitive matching:

>>> "GTAC" in dna\_seq

False

>>> "GTAC" in dna\_seq.upper()

True

Note that strictly speaking the IUPAC alphabets are for upper case sequences only, thus:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> dna\_seq = Seq("ACGT", IUPAC.unambiguous\_dna)

>>> dna\_seq

Seq('ACGT', IUPACUnambiguousDNA())

>>> dna\_seq.lower()

Seq('acgt', DNAAlphabet())

**3.7  Nucleotide sequences and (reverse) complements**

For nucleotide sequences, you can easily obtain the complement or reverse complement of a Seq object using its built-in methods:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> my\_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous\_dna)

>>> my\_seq

Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPACUnambiguousDNA())

>>> my\_seq.complement()

Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG', IUPACUnambiguousDNA())

>>> my\_seq.reverse\_complement()

Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC', IUPACUnambiguousDNA())

As mentioned earlier, an easy way to just reverse a Seq object (or a Python string) is slice it with -1 step:

>>> my\_seq[::-1]

Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())

In all of these operations, the alphabet property is maintained. This is very useful in case you accidentally end up trying to do something weird like take the (reverse)complement of a protein sequence:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> protein\_seq = Seq("EVRNAK", IUPAC.protein)

>>> protein\_seq.complement()

Traceback (most recent call last):

...

ValueError: Proteins do not have complements!

**3.8  Transcription**

Before talking about transcription, I want to try to clarify the strand issue. Consider the following (made up) stretch of double stranded DNA which encodes a short peptide:

|  |
| --- |
|  |
|  | DNA coding strand (aka Crick strand, strand +1) |  |
| 5’ | ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG | 3’ |
|  | ||||||||||||||||||||||||||||||||||||||| |  |
| 3’ | TACCGGTAACATTACCCGGCGACTTTCCCACGGGCTATC | 5’ |
|  | DNA template strand (aka Watson strand, strand −1) |  |
|  |  |  |
|  | | |  |
|  | Transcription |  |
|  | ↓ |  |
|  |  |  |
| 5’ | AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG | 3’ |
|  | Single stranded messenger RNA |  |
|  |  |  |

The actual biological transcription process works from the template strand, doing a reverse complement (TCAG → CUGA) to give the mRNA. However, in Biopython and bioinformatics in general, we typically work directly with the coding strand because this means we can get the mRNA sequence just by switching T → U.

Now let’s actually get down to doing a transcription in Biopython. First, let’s create Seq objects for the coding and template DNA strands:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> coding\_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous\_dna)

>>> coding\_dna

Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())

>>> template\_dna = coding\_dna.reverse\_complement()

>>> template\_dna

Seq('CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT', IUPACUnambiguousDNA())

These should match the figure above - remember by convention nucleotide sequences are normally read from the 5’ to 3’ direction, while in the figure the template strand is shown reversed.

Now let’s transcribe the coding strand into the corresponding mRNA, using the Seq object’s built in transcribe method:

>>> coding\_dna

Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())

>>> messenger\_rna = coding\_dna.transcribe()

>>> messenger\_rna

Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())

As you can see, all this does is switch T → U, and adjust the alphabet.

If you do want to do a true biological transcription starting with the template strand, then this becomes a two-step process:

>>> template\_dna.reverse\_complement().transcribe()

Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())

The Seq object also includes a back-transcription method for going from the mRNA to the coding strand of the DNA. Again, this is a simple U → T substitution and associated change of alphabet:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> messenger\_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous\_rna)

>>> messenger\_rna

Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())

>>> messenger\_rna.back\_transcribe()

Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())

**3.9  Translation**

Sticking with the same example discussed in the transcription section above, now let’s translate this mRNA into the corresponding protein sequence - again taking advantage of one of the Seq object’s biological methods:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> messenger\_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous\_rna)

>>> messenger\_rna

Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())

>>> messenger\_rna.translate()

Seq('MAIVMGR\*KGAR\*', HasStopCodon(IUPACProtein(), '\*'))

You can also translate directly from the coding strand DNA sequence:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> coding\_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous\_dna)

>>> coding\_dna

Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())

>>> coding\_dna.translate()

Seq('MAIVMGR\*KGAR\*', HasStopCodon(IUPACProtein(), '\*'))

You should notice in the above protein sequences that in addition to the end stop character, there is an internal stop as well. This was a deliberate choice of example, as it gives an excuse to talk about some optional arguments, including different translation tables (Genetic Codes).

The translation tables available in Biopython are based on those [from the NCBI](http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi) (see the next section of this tutorial). By default, translation will use the *standard* genetic code (NCBI table id 1). Suppose we are dealing with a mitochondrial sequence. We need to tell the translation function to use the relevant genetic code instead:

>>> coding\_dna.translate(table="Vertebrate Mitochondrial")

Seq('MAIVMGRWKGAR\*', HasStopCodon(IUPACProtein(), '\*'))

You can also specify the table using the NCBI table number which is shorter, and often included in the feature annotation of GenBank files:

>>> coding\_dna.translate(table=2)

Seq('MAIVMGRWKGAR\*', HasStopCodon(IUPACProtein(), '\*'))

Now, you may want to translate the nucleotides up to the first in frame stop codon, and then stop (as happens in nature):

>>> coding\_dna.translate()

Seq('MAIVMGR\*KGAR\*', HasStopCodon(IUPACProtein(), '\*'))

>>> coding\_dna.translate(to\_stop=True)

Seq('MAIVMGR', IUPACProtein())

>>> coding\_dna.translate(table=2)

Seq('MAIVMGRWKGAR\*', HasStopCodon(IUPACProtein(), '\*'))

>>> coding\_dna.translate(table=2, to\_stop=True)

Seq('MAIVMGRWKGAR', IUPACProtein())

Notice that when you use the to\_stop argument, the stop codon itself is not translated - and the stop symbol is not included at the end of your protein sequence.

You can even specify the stop symbol if you don’t like the default asterisk:

>>> coding\_dna.translate(table=2, stop\_symbol="@")

Seq('MAIVMGRWKGAR@', HasStopCodon(IUPACProtein(), '@'))

Now, suppose you have a complete coding sequence CDS, which is to say a nucleotide sequence (e.g. mRNA – after any splicing) which is a whole number of codons (i.e. the length is a multiple of three), commences with a start codon, ends with a stop codon, and has no internal in-frame stop codons. In general, given a complete CDS, the default translate method will do what you want (perhaps with the to\_stop option). However, what if your sequence uses a non-standard start codon? This happens a lot in bacteria – for example the gene yaaX in E. coli K12:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import generic\_dna

>>> gene = Seq("GTGAAAAAGATGCAATCTATCGTACTCGCACTTTCCCTGGTTCTGGTCGCTCCCATGGCA" + \

... "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAAATTACAGATAGGCGATCGTGAT" + \

... "AATCGTGGCTATTACTGGGATGGAGGTCACTGGCGCGACCACGGCTGGTGGAAACAACAT" + \

... "TATGAATGGCGAGGCAATCGCTGGCACCTACACGGACCGCCGCCACCGCCGCGCCACCAT" + \

... "AAGAAAGCTCCTCATGATCATCACGGCGGTCATGGTCCAGGCAAACATCACCGCTAA",

... generic\_dna)

>>> gene.translate(table="Bacterial")

Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HR\*',

HasStopCodon(ExtendedIUPACProtein(), '\*')

>>> gene.translate(table="Bacterial", to\_stop=True)

Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',

ExtendedIUPACProtein())

In the bacterial genetic code GTG is a valid start codon, and while it does *normally* encode Valine, if used as a start codon it should be translated as methionine. This happens if you tell Biopython your sequence is a complete CDS:

>>> gene.translate(table="Bacterial", cds=True)

Seq('MKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',

ExtendedIUPACProtein())

In addition to telling Biopython to translate an alternative start codon as methionine, using this option also makes sure your sequence really is a valid CDS (you’ll get an exception if not).

**3.10  Translation Tables**

In the previous sections we talked about the Seq object translation method (and mentioned the equivalent function in the Bio.Seq module). Internally these use codon table objects derived from the NCBI information at <ftp://ftp.ncbi.nlm.nih.gov/entrez/misc/data/gc.prt>, also shown on <http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi> in a much more readable layout.

As before, let’s just focus on two choices: the Standard translation table, and the translation table for Vertebrate Mitochondrial DNA.

>>> from Bio.Data import CodonTable

>>> standard\_table = CodonTable.unambiguous\_dna\_by\_name["Standard"]

>>> mito\_table = CodonTable.unambiguous\_dna\_by\_name["Vertebrate Mitochondrial"]

Alternatively, these tables are labeled with ID numbers 1 and 2, respectively:

>>> from Bio.Data import CodonTable

>>> standard\_table = CodonTable.unambiguous\_dna\_by\_id[1]

>>> mito\_table = CodonTable.unambiguous\_dna\_by\_id[2]

You can compare the actual tables visually by printing them:

>>> print(standard\_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

--+---------+---------+---------+---------+--

and:

>>> print(mito\_table)

Table 2 Vertebrate Mitochondrial, SGC1

| 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 W | A

T | TTG L | 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 | CCG P | CAG Q | CGG R | G

--+---------+---------+---------+---------+--

A | ATT I(s)| ACT T | AAT N | AGT S | T

A | ATC I(s)| ACC T | AAC N | AGC S | C

A | ATA M(s)| ACA T | AAA K | AGA Stop| A

A | ATG M(s)| ACG T | AAG K | AGG Stop| 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(s)| GCG A | GAG E | GGG G | G

--+---------+---------+---------+---------+--

You may find these following properties useful – for example if you are trying to do your own gene finding:

>>> mito\_table.stop\_codons

['TAA', 'TAG', 'AGA', 'AGG']

>>> mito\_table.start\_codons

['ATT', 'ATC', 'ATA', 'ATG', 'GTG']

>>> mito\_table.forward\_table["ACG"]

'T'

**Comparing Seq objects**

Sequence comparison is actually a very complicated topic, and there is no easy way to decide if two sequences are equal. The basic problem is the meaning of the letters in a sequence are context dependent - the letter “A” could be part of a DNA, RNA or protein sequence. Biopython uses alphabet objects as part of each Seq object to try to capture this information - so comparing two Seq objects could mean considering both the sequence strings *and* the alphabets.

For example, you might argue that the two DNA Seq objects Seq("ACGT", IUPAC.unambiguous\_dna) and Seq("ACGT", IUPAC.ambiguous\_dna) should be equal, even though they do have different alphabets. Depending on the context this could be important.

This gets worse – suppose you think Seq("ACGT", IUPAC.unambiguous\_dna) and Seq("ACGT") (i.e. the default generic alphabet) should be equal. Then, logically, Seq("ACGT", IUPAC.protein) and Seq("ACGT") should also be equal. Now, in logic if *A*=*B* and *B*=*C*, by transitivity we expect *A*=*C*. So for logical consistency we’d require Seq("ACGT", IUPAC.unambiguous\_dna) and Seq("ACGT", IUPAC.protein) to be equal – which most people would agree is just not right. This transitivity also has implications for using Seq objects as Python dictionary keys.

Now, in everyday use, your sequences will probably all have the same alphabet, or at least all be the same type of sequence (all DNA, all RNA, or all protein). What you probably want is to just compare the sequences as strings – which you can do explicitly:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> seq1 = Seq("ACGT", IUPAC.unambiguous\_dna)

>>> seq2 = Seq("ACGT", IUPAC.ambiguous\_dna)

>>> str(seq1) == str(seq2)

True

>>> str(seq1) == str(seq1)

True

So, what does Biopython do? Well, as of Biopython 1.65, sequence comparison only looks at the sequence, essentially ignoring the alphabet:

>>> seq1 == seq2

True

>>> seq1 == "ACGT"

True

As an extension to this, using sequence objects as keys in a Python dictionary is now equivalent to using the sequence as a plain string for the key.

Note if you compare sequences with incompatible alphabets (e.g. DNA vs RNA, or nucleotide versus protein), then you will get a warning but for the comparison itself only the string of letters in the sequence is used:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import generic\_dna, generic\_protein

>>> dna\_seq = Seq("ACGT", generic\_dna)

>>> prot\_seq = Seq(``ACGT'', generic\_protein)

>>> dna\_seq == prot\_seq

BiopythonWarning: Incompatible alphabets DNAAlphabet() and ProteinAlphabet()

True

*WARNING:* Older versions of Biopython instead used to check if the Seq objects were the same object in memory. This is important if you need to support scripts on both old and new versions of Biopython. Here make the comparison explicit by wrapping your sequence objects with eitherstr(...) for string based comparison or id(...) for object instance based comparison.

**MutableSeq objects**

Just like the normal Python string, the Seq object is “read only”, or in Python terminology, immutable. Apart from wanting the Seq object to act like a string, this is also a useful default since in many biological applications you want to ensure you are not changing your sequence data:

>>> from Bio.Seq import Seq

>>> from Bio.Alphabet import IUPAC

>>> my\_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous\_dna)

Observe what happens if you try to edit the sequence:

>>> my\_seq[5] = "G"

Traceback (most recent call last):

...

TypeError: 'Seq' object does not support item assignment

However, you can convert it into a mutable sequence (a MutableSeq object) and do pretty much anything you want with it:

>>> mutable\_seq = my\_seq.tomutable()

>>> mutable\_seq

MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())

Alternatively, you can create a MutableSeq object directly from a string:

>>> from Bio.Seq import MutableSeq

>>> from Bio.Alphabet import IUPAC

>>> mutable\_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous\_dna)

Either way will give you a sequence object which can be changed:

>>> mutable\_seq

MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())

>>> mutable\_seq[5] = "C"

>>> mutable\_seq

MutableSeq('GCCATCGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())

>>> mutable\_seq.remove("T")

>>> mutable\_seq

MutableSeq('GCCACGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())

>>> mutable\_seq.reverse()

>>> mutable\_seq

MutableSeq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())

Do note that unlike the Seq object, the MutableSeq object’s methods like reverse\_complement() and reverse() act in-situ!

An important technical difference between mutable and immutable objects in Python means that you can’t use a MutableSeq object as a dictionary key, but you can use a Python string or a Seq object in this way.

Once you have finished editing your a MutableSeq object, it’s easy to get back to a read-only Seq object should you need to:

>>> new\_seq = mutable\_seq.toseq()

>>> new\_seq

Seq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())

**UnknownSeq objects**

The UnknownSeq object is a subclass of the basic Seq object and its purpose is to represent a sequence where we know the length, but not the actual letters making it up. You could of course use a normal Seq object in this situation, but it wastes rather a lot of memory to hold a string of a million “N” characters when you could just store a single letter “N” and the desired length as an integer.

>>> from Bio.Seq import UnknownSeq

>>> unk = UnknownSeq(20)

>>> unk

UnknownSeq(20, alphabet = Alphabet(), character = '?')

>>> print(unk)

????????????????????

>>> len(unk)

20

You can of course specify an alphabet, meaning for nucleotide sequences the letter defaults to “N” and for proteins “X”, rather than just “?”.

>>> from Bio.Seq import UnknownSeq

>>> from Bio.Alphabet import IUPAC

>>> unk\_dna = UnknownSeq(20, alphabet=IUPAC.ambiguous\_dna)

>>> unk\_dna

UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')

>>> print(unk\_dna)

NNNNNNNNNNNNNNNNNNNN

You can use all the usual Seq object methods too, note these give back memory saving UnknownSeq objects where appropriate as you might expect:

>>> unk\_dna

UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')

>>> unk\_dna.complement()

UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')

>>> unk\_dna.reverse\_complement()

UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')

>>> unk\_dna.transcribe()

UnknownSeq(20, alphabet = IUPACAmbiguousRNA(), character = 'N')

>>> unk\_protein = unk\_dna.translate()

>>> unk\_protein

UnknownSeq(6, alphabet = ProteinAlphabet(), character = 'X')

>>> print(unk\_protein)

XXXXXX

>>> len(unk\_protein)

**Working with strings directly**

To close this chapter, for those you who *really* don’t want to use the sequence objects (or who prefer a functional programming style to an object orientated one), there are module level functions in Bio.Seq will accept plain Python strings, Seq objects (including UnknownSeq objects) orMutableSeq objects:

>>> from Bio.Seq import reverse\_complement, transcribe, back\_transcribe, translate

>>> my\_string = "GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGTTAG"

>>> reverse\_complement(my\_string)

'CTAACCAGCAGCACGACCACCCTTCCAACGACCCATAACAGC'

>>> transcribe(my\_string)

'GCUGUUAUGGGUCGUUGGAAGGGUGGUCGUGCUGCUGGUUAG'

>>> back\_transcribe(my\_string)

'GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGTTAG'

>>> translate(my\_string)

'AVMGRWKGGRAAG\*'

You are, however, encouraged to work with Seq objects by default.