CHAPTER 19

Working with Sequence Data

LEARNING GOAL: You can manipulate DNA, RNA, and protein sequences using Biopython.

19.1 IN THIS CHAPTER YOU WILL LEARN

- How to create a sequence object
- How to reverse and transcribe a DNA sequence
- How to translate an RNA sequence into a protein sequence
- How to create a sequence record
- How to read sequence files in different formats
- How to write formatted sequence files

19.2 STORY: HOW TO TRANSLATE A DNA CODING SEQUENCE INTO THE CORRESPONDING PROTEIN SEQUENCE AND WRITE IT TO A FASTA FILE

19.2.1 Problem Description

In Chapter 4 you learned how to parse sequence files using elementary operations on strings. For example, you learned that the “>” symbol in FASTA formatted files at the first position of a row starts the header of the record and contains the sequence ID and some concise annotations.

You also learned that you can transcribe a DNA sequence by replacing the `Ts` by `Us` and translate it using a dictionary for the genetic code (see Chapter 5, Section 5.3.1). Here, these and other actions are accomplished using Biopython modules and methods. Biopython gives you a shortcut for accomplishing these tasks by providing tools to manipulate sequences and sequence files, thus making it very simple to work with different file formats, to annotate sequence records, to write them to files, etc. In the following Python session, four modules from the `Bio` package are used: `Seq` is needed to create a sequence object; `IUPAC` is used to assign a biological alphabet (e.g., DNA or protein) to a sequence object; `SeqRecord` allows creating a sequence record object equipped with ID, annotation, description, etc.; and `SeqIO` provides methods to read and write formatted sequence files.

19.2.2 Example Python Session

```
from Bio import Seq
from Bio.Alphabet import IUPAC
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO
# read the input sequence
dna = open("hemoglobin-gene.txt").read().strip()
dna = Seq.Seq(dna, IUPAC.unambiguous_dna)
# transcribe and translate
mrna = dna.transcribe()
protein = mrna.translate()
# write the protein sequence to a file
protein_record = SeqRecord(protein,\
    id='sp|P69905.2|HBA_HUMAN',\
    description="Hemoglobin subunit alpha, human")
outfile = open("HBA_HUMAN.fasta", "w")
SeqIO.write(protein_record, outfile,"fasta")
outfile.close()
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

19.3 WHAT DO THE COMMANDS MEAN?

In Section 19.2.2, the DNA coding sequence from the hemoglobin subunit alpha is read from a plain text file (hemoglobin-gene.txt; the sequence is identical to the one in Appendix C, Section C.2, “A Single Nucleotide Sequence File in FASTA Format”). It is then transcribed into an mRNA sequence, translated into a peptide sequence, and written to the `HBA_HUMAN.fasta` output file. Let's look at the objects imported in the Python session one by one.

19.3.1 The `Seq` Object

The first imported name is `Seq`, which is a module within the `Bio` library. The `Seq.Seq` class creates a *sequence* object, i.e., a sequence associated with an alphabet attribute, which specifies the kind of sequence stored in the object. You can create `Seq` objects with or without specifying an alphabet:

```
>>> from Bio import Seq
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC")
>>> my_seq
Seq('AGCATCGTAGCATGCAC', Alphabet())
```

Biopython contains a set of precompiled alphabets that cover all biological sequence types. IUPAC-defined alphabets (http://www.chem.qmw.ac.uk/iupac) are the most frequently used. If you want to use alphabets, you have to import the `IUPAC` module from the `Bio.Alphabet` module (as in Section 19.2.2). It contains the alphabets `IUPACUnambiguousDNA` (basic ACGT letters), `IUPACAmbiguousDNA` (includes ambiguous letters), `ExtendedIUPACDNA` (includes modified bases), `IUPACUnambiguousRNA`, `IUPACAmbiguousRNA`, `IUPACProtein` (IUPAC standard amino acids), and `ExtendedIUPACProtein` (includes selenocysteine, X, etc.). The `dna` variable defined in Section 19.2.2 is a sequence object characterized by the `IUPAC.unambiguous_dna` alphabet.

Transcribing and Translating Sequences

Methods of `Seq` objects are designed specifically for biological sequences; for example, you can obtain the transcription of a DNA sequence using the `transcribe` method, as shown in Section 19.2.2:

```
>>> my_seq.transcribe()
Seq('AGCAUCGUAGCAUGCAC', RNAAlphabet())
```

The `transcribe` method just changes the `T`s to `U`s and sets the alphabet to RNA. It assumes that the input DNA sequence is the coding strand.

If you have a template strand and want to perform the transcription, you need to get the reverse complement first and then transcribe it:

```
>>> dna = Seq.Seq("AGCATCGTAGCATGCAC", IUPAC.unambiguous_dna)
>>> cdna = dna.reverse_complement()
>>> cdna
Seq('GTGCATGCTACGATGCT', IUPACUnambiguousDNA())
>>> mrna = codingStrand.transcribe()
>>> mrna
Seq('GUGCAUGCUACGAUGCU', IUPACUnambiguousRNA())
```

Or in a single command line:

```
>>> dna.reverse_complement().transcribe()
Seq('GUGCAUGCUACGAUGCU', IUPACUnambiguousRNA())
```

These methods are available for sequences assigned to protein alphabets as well, but their execution will raise errors.

A DNA or RNA sequence object can also be translated into a protein sequence. To this aim, a number of genetic codes are available through the `CodonTable` module of the `Bio.Data` module:

```
>>> from Bio.Data import CodonTable
```

You can access the tables through the dictionaries available in the `CodonTable` module (the list of dictionaries available can be visualized using the `dir()` function). For example, the `unambiguous_dna_by_name` dictionary makes it possible to access the set of DNA codon tables by name (e.g., `"Standard"`, `"Vertebrate Mitochondrial"`, etc.).

```
>>> from Bio.Data import CodonTable
>>> standard_table = \
... CodonTable.unambiguous_dna_by_name["Standard"]
```

In contrast, the `unambiguous_dna_by_id` uses a numerical identifier (`1` corresponds to the `"Standard"` codon table, `2` to the `"Vertebrate Mitochondrial"`, etc.). All NCBI-defined alphabets are available and identified by the NCBI table identifier (see www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi). By default, Biopython translation will use the standard genetic code (corresponding to the NCBI table ID 1).

If you print the `standard_table` variable, you will get the codon table from Figure 19.1. Codon table objects have other useful attributes as well, such as start and stop codons:

```
>>> standard_table.start_codons
['TTG', 'CTG', 'ATG']
>>> standard_table.stop_codons
['TAA', 'TAG', 'TGA']
>>> mito_table = \
... CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mito_table.stop_codons
['TAA', 'TAG', 'AGA', 'AGG']
```

The `translate` method translates an RNA or DNA sequence using either the default or a specified genetic code and returns a `Seq` object, the alphabet of which will contain additional information. In the following

```
  | 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
--+---------+---------+---------+---------+--
```

FIGURE 19.1 The DNA unambiguous codon table.

example, the output alphabet contains information related to the presence of stop codons in the translated sequence:

```
>>> mrna = \
... Seq.Seq('AUGGCCAUUGUA AUGGGCCGCUGAA AGGGAUAG',\
... IUPAC.unambiguous_rna)
>>> mrna.translate(table = "Standard")
Seq('MAIVMGR*KG*', HasStopCodon(IUPACProtein(), '*'))
>>> mrna.translate(table = "Vertebrate Mitochondrial")
Seq('MAIVMGRWKG*', HasStopCodon(IUPACProtein(), '*'))
```

By default, all stop codons encountered during the translation of the nucleic acid sequence are returned as stars ("*"). Notice that there are two stop codons in the mRNA sequence if the standard codon table (`table = "Standard"`) is used (UGA and UAG) and only one if the vertebrate mitochondrial codon table is used (`table = "Vertebrate Mitochondrial"`). In fact, the UGA codon is recognized as a tryptophan (W) in the mitochondrion. You can impose the translation to stop at the first encountered stop codon:

```
>>> mrna.translate(to_stop = True, table = 1)
Seq('MAIVMGR', IUPACProtein())
>>> mrna.translate(to_stop = True, table = 2)
Seq('MAIVMGRWKG', IUPACProtein())
```

19.3.2 Working with Sequences as Strings

You can manipulate sequence objects in Biopython in the same way as strings. For example, you can index, slice, split, convert the sequence to uppercase or lowercase, count occurrences of characters, and so on:

```
>>> from Bio import Seq
>>> my_seq = Seq.Seq("AGCATCGTA GCATGCAC")
>>> my_seq[0]
'A'
>>> my_seq[0:3]
Seq('AGC', Alphabet())
>>> my_seq.split('T')
[Seq('AGCA', Alphabet()), Seq('CG', Alphabet()),
    Seq('AGCA', Alphabet()), Seq('GCAC', Alphabet())]
>>> my_seq.count('A')
5
>>> my_seq.count('A') / float(len(my_seq))
0.29411764705882354
```

Notice that when you slice a `Seq` object, or you split it, the methods return not just strings but other `Seq` objects. Sequence objects can also be concatenated, but only if their alphabets are compatible (or are generic alphabets):

```
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC", IUPAC.unambiguous_dna)
>>> my_seq_2 = Seq.Seq("CGTC", IUPAC.unambiguous_dna)
>>> my_seq + my_seq_2
Seq('AGCATCGTAGCATGCACCGTC', IUPACUnambiguousDNA())
```

You can search the sequence for the occurrence of specific substrings using the `find` method. If the subsequence is not found, Python will return `-1`; if the subsequence is found, the position of the leftmost matching character in the target sequence is returned:

```
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC", IUPAC.unambiguous_dna)
>>> my_seq.find("TCGT")
4
>>> my_seq.find("TTTT")
-1
```

It is also possible to search for patterns represented by regular expressions using the Python `re` module or using the Biopython module `Bio.Motif` (see the Biopython tutorial).

Finally, Biopython provides a number of functions, such as `transcribe()` or `translate()`, that can be used on strings directly:

```
>>> my_seq_str = "AGCATCGTAGCATGCAC"
>>> Bio.Seq.transcribe(my_seq_str)
'AGCAUCGUAGCAUGCAC'
>>> Bio.Seq.translate(my_seq_str)
'SIVAC'
>>> Bio.Seq.reverse_complement(my_seq_str)
'GTGCATGCTACGATGCT'
```

19.3.3 The `MutableSeq` Object

`Seq` objects behave similarly to Python strings, in the sense that they are immutable. Therefore, if you try to reassign a residue in a sequence object, you will get an error message. Biopython provides the `MutableSeq` object to create mutable sequence objects:

```
>>> my_seq = Seq.Seq("AGCATCGTAGCATG", IUPAC.unambiguous_dna)
>>> my_seq[5] = "T"
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
TypeError: 'Seq' object does not support item assignment
>>> my_seq = \
... Seq.MutableSeq("AGCATCGTAGCATG", IUPAC.unambiguous_dna)
>>> my_seq[5] = "T"
>>> my_seq
MutableSeq('AGCATTGTAGCATG', IUPACUnambiguousDNA())
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

You cannot use methods such as `reverse()` or `remove()` on `Seq` objects, but you can use them on `MutableSeq` objects.

Finally, it is possible to convert an immutable `Seq` object into a mutable one, and vice versa, using the `tomutable()` method of `Seq` objects and the `toseq()` method of `MutableSeq` objects:

```
>>> my_mut_seq = my_seq.tomutable()
>>> my_mut_seq
MutableSeq('AGCATCGTAGCATGCAC', IUPACUnambiguousDNA())
>>> my_seq = my_mut_seq.toseq()
>>> my_seq
Seq('AGCATCGTAGCATGCAC', IUPACUnambiguousDNA())
```

Because `MutableSeq` objects can change (much like lists or sets), they cannot be used as dictionary keys, while `Seq` objects can.

19.3.4 The `SeqRecord` Object

The `SeqRecord` class provides a container for a sequence and its annotation. In the Python session in Section 19.2.2, the protein sequence in the `protein_seq` variable, obtained by translating the `mRNA` sequence object, is converted into a `SeqRecord` object:

```
protein_record = \
SeqRecord(protein_seq,id = 'sp|P69905.2|HBA_HUMAN', \
description = "Hemoglobin subunit alpha, Homo sapiens")
```

The arguments passed to `SeqRecord` to create the object are a `Seq` object (stored in the `protein_seq` variable), an `id` and a `description`, which must be both strings. `SeqRecord` objects allow associating features to a sequence object, such as the identifier or a description. The available features are as follows:

- `seq`: This is a biological sequence, typically in the form of a `Seq` object.
- `id`: This is the primary ID used to identify the sequence.

- `name`: This is a "common" molecule name.
- `description`: This is a description of the sequence/molecule.
- `letter_annotations`: This is a dictionary with per-residue annotations. Keys are the type of annotation (e.g., "secondary structure"), and values are Python sequences (lists, tuples, or strings) having the same length as the sequence, where each element is a per-residue annotation (e.g., secondary structure type indicated with a single character: S = strand, H = helix, etc.). It is useful for assigning quality scores, secondary structure or accessibility preferences, etc. to residues.
- `annotations`: This is a dictionary of additional information about the sequence. The keys are the type of information, and the information is contained in the value.
- `features`: This is a list of `SeqFeature` objects, with more structured information about sequence features (e.g., position of genes on a genome, or domains on a protein sequence; see following text).
- `dbxrefs`: This is a list of database cross-references.

Such features can be manually created by the user or imported from a database record (e.g., a GenBank or SwissProt file; see also Chapter 20). In Section 19.2.2, a `SeqRecord` object associated with a `Seq` object was created with an ID and a description. Both features can be retrieved directly:

```
>>> protein_record.id
'sp|P69905.2|HBA_HUMAN'
>>> protein_record.description
'Hemoglobin subunit alpha, Homo sapiens'
```

Features can also be assigned on the fly:

```
>>> protein_record.name = "Hemoglobin"
```

The `annotation` attribute is an empty dictionary that can be used to store all kinds of information that do not fall into the categories already provided by `SeqRecord`:

```
>>> protein_record.annotations["origin"] = "human"
>>> protein_record.annotations["subunit"] = "alpha"
>>> protein_record.annotations
{'origin': 'human', 'subunit': 'alpha'}
```

Similarly, the `letter_annotations` attribute is an empty dictionary the values of which must be strings, lists, or tuples of exactly the same length of the sequence:

```
>>> protein_record.letter_annotations[\
..."secondary structure"] = \
...'HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHSSSSSS...'
```

Converting `SeqRecord` *Objects to File Formats*

Once you have set attributes for your sequence, you can convert it to some of the most popular storage formats for sequences by using the `format` method:

```
>>> print protein_record.format("fasta")
>sp|P69905.2|HBA_HUMAN Hemoglobin subunit alpha, Homo sapiens
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHG
KKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTP
AVHASLDKFLASVSTVLTSKYR*
```

See what happens if you use the `"genbank"` format:

```
>>> print protein_record.format("genbank")
```

Finally, you can slice a `SeqRecord` object: where possible, the annotation will be sliced accordingly (e.g., `letter_annotations`), but some features (e.g., `dbxrefs`) will not be extended to the sliced object. Two `SeqRecord` objects can be also concatenated by adding them into a new `SeqRecord`. The new object will inherit some of the features that are identical in the two parent `SeqRecord` objects (e.g., `id`), whereas some others are not inherited in any case (such as `annotations`).

19.3.5 The `SeqIO` Module

In Chapter 4, we introduced procedures to parse files using standard Python commands. Here, you will see how to read and write sequence files using Biopython. The `SeqIO` module is very useful to parse many common file formats and write annotated sequences to standard file formats. In Section 19.2.2, `protein_record` (a `SeqRecord` object) is written to the `HBA_HUMAN.fasta` output file in FASTA format.

The Biopython `SeqIO` module provides parsers for many common file formats. These parsers extract information from an input file (either local or retrieved from a database) and automatically convert it into a `SeqRecord` object. `SeqIO` also provides a method to write `SeqRecord` objects to conveniently formatted files.

Parsing Files

There are two methods for sequence file parsing: `SeqIO.parse()` and `SeqIO.read()`; both of them require two mandatory arguments and one optional argument:

1. a file (mandatory; also called a "handle" object) that specifies where the data must be read from (could be a filename, a file opened for reading, data downloaded from a database using a script, or the output of another piece of code);
2. a string indicating the format of the data (mandatory; e.g., `"fasta"` or `"genbank"`; a full list of supported formats is available at http://biopython.org/wiki/SeqIO);
3. an argument that specifies the alphabet of the sequence data (optional).

The difference between the two methods `SeqIO.parse()` and `SeqIO.read()` is that `SeqIO.parse()` returns an iterator that produces `SeqRecord` objects from an input file of several records. You can use the iterator like a list in `for` or `while` loops. See Example 19.2. If you have a file containing a single record, you have to use `SeqIO.read()` instead. It returns a `SeqRecord` object. While `SeqIO.parse()` can process any number of records in the input handle, `SeqIO.read()` parses only single-record files by first checking whether there is only one record in the handle and raises an error if this condition is not met.

Q & A: WHAT IS AN ITERATOR?

An iterator is a data structure that produces a series of entries (e.g., `SeqRecord` objects). It can be used like a list in loops, but technically it is not a list. An iterator has no length, and it cannot be indexed and sliced. You can only request the next object from it. When you do it, the iterator looks to see if there are more records available. This way, the iterator does not need to keep all records in the memory all the time.

Parsing Large Sequence Files

The usage of iterators is a way to parse large files without consuming large amounts of memory. For a big number of records, you can use `SeqIO.index()`, a method that needs two arguments: a record filename and a file format. The `SeqIO.index()` method returns a dictionary-like object that gives you access to all records without keeping all data in the memory. The dictionary keys are the IDs of the records, and the values contain the entire record, which can be accessed using the attributes `id`, `description`, etc. When a particular record is accessed, the record content is parsed on the fly. This method allows you to manipulate huge files, with a little cost in flexibility and speed. Notice that these dictionary-like objects are read-only, meaning that once they are created, no records can be inserted or removed.

Writing Files

The `SeqIO.write()` method writes one or more `SeqRecord` objects to a file in the format specified by the user. The method requires three arguments: one or more `SeqRecord` objects, a handle object (i.e., a file opened with the `"w"` modality) or a filename to write to, and a sequence format (e.g., `"fasta"` or `"genbank"`).

The first argument can be a list, an iterator, or an individual `SeqRecord`, as shown in Section 19.2.2. When writing GenBank files, the alphabet must be set for the sequence.

Concluding Remarks

In some cases, you may prefer to use traditional programming, e.g., if you want a customized parser or when you have a nonstandard format that Biopython fails to parse. In other cases, you may find it more convenient to use the `SeqIO` module, e.g., when you have to index large files. In both cases, you have to be aware that file formats change occasionally and that they may contain unexpected characters, lines, and exceptions, which could break even the best-designed parser.

19.4 EXAMPLES

Example 19.1 Using the `Bio.SeqIO` Module to Parse a Multiple Sequence FASTA File

In the following example, the multiple sequence FASTA file shown in Appendix C, Section C.4, "A Multiple Sequence File in FASTA Format" is parsed:

```
from Bio import SeqIO
fasta_file = open("Uniprot.fasta","r")
for seq_record in SeqIO.parse(fasta_file, "fasta"):
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)
fasta_file.close()
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

The code writes the identifiers, sequences, and lengths for all three entries in the FASTA file:

```
sp|P03372|ESR1_HUMAN
Seq('MTMTLHTKASGMALL HQIQGNELEPLNRPQLKIPLER
PLGEVYLDSSKPAVYNY...ATV', SingleLetterAlphabet())
595
sp|P62333|PRS10_HUMAN
Seq('MADPRDKALQDYRK KLLEHKEIDGRLKELREQLKELT
KQYEKSENDLKALQSVG...KPV', SingleLetterAlphabet())
389
sp|P62509|ERR3_MOUSE
Seq('MDSVELCLPESFS LHYEEELLCRMSNKDRHIDSSCSS
FIKTEPSSPASLTDSVN...AKV', SingleLetterAlphabet())
458
```

Since the handle is a file, it is a good habit to close it when the processing is done. Remember that the iterator "empties" the file, meaning that to scan the records another time, the file must be closed, then opened again, and then used again as the handle argument of `SeqIO.parse()`. You can also use `SeqIO.parse()` by omitting the explicit creation of the handle and directly passing a filename or a complete path to `SeqIO.parse`. For example,

```
>>> for seq_record in SeqIO.parse("Uniprot.fasta", "fasta"):
… print seq_record.id
sp|P03372|ESR1_HUMAN
sp|P62333|PRS10_HUMAN
sp|P62509|ERR3_MOUSE
```

You can also parse records one by one using the `next()` method of the iterator:

```
>>> uniprot_iterator = SeqIO.parse("Uniprot.fasta","fasta")
>>> uniprot_iterator.next().id
```

```
'sp|P03372|ESR1_HUMAN'
>>> uniprot_iterator.next().id
'sp|P62333|PRS10_HUMAN'
```

When all records have been read, the `next()` method will return either `None` or a `StopIteration` exception (depending on your Biopython version).

Example 19.2 Using the `SeqIO` Module to Parse a Record File and Store Its Content in a List or a Dictionary

You can easily store all records from a file in a list:

```
from Bio import SeqIO
uniprot_iterator = SeqIO.parse("Uniprot.fasta", "fasta")
records = list(uniprot_iterator)
print records[0].id
print records[0].seq
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

This code generates the output:

```
sp|P03372|ESR1_HUMAN
MTMTLHTKASGMALLHQIQGNELEPLNRPQLKI…
```

Alternatively, you can use a dictionary, the keys of which are the record IDs and the values of which contain the record information:

```
uniprot_iterator = SeqIO.parse("Uniprot.fasta", "fasta")
records = SeqIO.to_dict(uniprot_iterator)
print records['sp|P03372|ESR1_HUMAN'].id
print records['sp|P03372|ESR1_HUMAN'].seq
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

This code generates the same output as above.

Example 19.3 Using `SeqIO.index()`to Parse a Big File

The usage of the `index` method helps process large sequence files that don't fit into the memory at the same time. In the example, the file from Example 19.1 will be read.

```
records = SeqIO.index("Uniprot.fasta","fasta")
print records.keys()
print len(records['sp|P03372|ESR1_HUMAN'].seq)
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

This code produces the output:

```
['sp|P03372|ESR1_HUMAN', 'sp|P62333|PRS10_HUMAN',
'sp|P62509|ERR3_MOUSE']
595
```

Example 19.4 Converting between Sequence File Formats

You can convert sequence file formats by combining the `Bio.SeqIO.parse()` and `Bio.SeqIO.write()` methods. The following script converts a GenBank file to a FASTA file:

```
from Bio import SeqIO
genbank_file = open ("AY810830.gbk", "r")
output_file = open("AY810830.fasta", "w")
records = SeqIO.parse(genbank_file, "genbank")
SeqIO.write(records, output_file, "fasta")
output_file.close()
```

Source: Adapted from code published by A.Via/K.Rother under the Python License.

Notice that if you do not close the output file, the writing cannot be completed.

19.5 TESTING YOURSELF

Exercise 19.1 Parse a Single Sequence Record

Read a single record from a FASTA formatted file, extract its ID and its sequence, and print them.

Exercise 19.2 Build and Write a `SeqRecord` Object to a File

Use the sequence ID and the sequence from Exercise 19.1 to create a `SeqRecord` object. Manually add a customized description. Write the `SeqRecord` object to a file in FASTA format and to a second file in GenBank format. Note that for GenBank format an alphabet must be assigned when creating the `Seq` object.

Exercise 19.3

Parse a multiple-record file and write to a file only the IDs of all records.

Exercise 19.4 Write GenBank Sequences to Separate Files

Parse a multiple-record file in GenBank format and write each record to a separate file in FASTA format. Use the IDs of the entries to create filenames.

Hint: You can manually create the input file by going to the GenBank website.

Exercise 19.5 Format Conversion

Try to convert the *protein* sequence FASTA formatted file of Example 19.1 (or a similar one) into GenBank format. What happens?