Bioinformatics using Python for Biologists

9.1 Biopython - Introduction

Biopython (http://biopython.org/wiki/Main_Page) is a collection of modules for computational molecular biology, which allows performing most of the basic (and in many cases also advanced) tasks required in a bioinformatics project.
The most common tasks that can be performed by using Biopython include:

- parsing (i.e. extracting information) of the most common file formats for gene and protein sequences, protein structures, PubMed records, etc.;
- download files from repositories such as NCBI, ExPASy, etc.;
- run (locally or remotely) popular bioinformatics algorithms such as Blast, Clustalw, etc.;
- run Biopython implementations of algorithms for clustering, machine learning, data analysis, data visualization.

It also provides classes that the Python programmer can use to handle data (such as sequences) and methods to perform operations on them (such as translation, complement, etc.). The Biopython Tutorial and Cookbook (http://biopython.org/DIST/docs/tutorial/Tutorial.html) is a good starting point to get a grasp of what Biopython can do for you. More advanced tutorials can be found for specific packages (for example the Structural Biopython package, http://biopython.org/DIST/docs/cookbook/biopdb_faq.pdf).

9.1.1 You can use Biopython in several ways

Biopython is not a program itself; it is a collection of tools for Python bioinformatics programming. In most cases, the Python programmer can build a research pipeline completely by using Biopython, or by writing new code for some specific tasks and using Biopython for more standard operations, or can modify Biopython open-source code to better adapt it to his/her own needs. Python excels as a "glue language" which can stick together other peoples' programs, functions, classes, etc.

Write your own code when:

- The algorithm implementation or coding is interesting to you;
- Biopython data structure mapping is too complex for your task;
- Biopython does not provide tools for your specific task;
- You want to have a complete control of what your code does.

Use Biopython when:

- Its modules and/or methods fit your needs;
- Your task is unchallenging or boring: Why waste your time? Don't "re-invent the wheel" unless you're doing it as a learning project;
- Your task will take you a lot of effort to write.

Extend Biopython (i.e. modify the Biopython source code) when:

- The Biopython modules and/or methods almost does what you need, but not exactly fit your need;
- But: it might be challenging for the beginner. It can be difficult to read and understand someone else's code.

Remember:

- When doing bioinformatics, keep Biopython in mind; It is very powerful;
- Browse the documentation; become familiar with its capabilities;
- Use help(), type(), dir() and other built-in features to explore Biopython modules.

9.1.2 Installing Biopython

Biopython is not part of the official Python distribution, and as such must be downloaded (from http://biopython.org/wiki/Download) and installed independently.
Prerequisite for Biopython is the installation of the NumPy package (downloadable from http://numpy.scipy.org/).
While Biopython installation is usually pain-free (follow the instructions at http://biopython.org/DIST/docs/install/Installation.html), NumPy can be more problematic, especially on Macs, for which there exist specially prepared installers (at http://stronginference.com/scipy-superpack/). Be careful: Biopython and NumPy versions are coordinated, meaning that specific Biopython releases must be installed for specific NumPy releases.
Additional packages can optionally be installed to allow Biopython to perform additional tasks (mainly for graphical outputs and plots of various kind).

9.1.3 Let's get started

To show the basic logic behind most of the Biopython modules, let's create a sequence object and see the methods associated to it. The module **`Bio`** is the main Biopython module.

```
>>> import sys
>>> sys.path.append("/Users/fabrizio/source/biopython-1.57")
>>> import Bio
>>> dir(Bio)
['BiopythonDeprecationWarning', 'MissingExternalDependencyError',
'MissingPythonDependencyError', '__builtins__', '__doc__',
'__file__', '__name__', '__package__', '__path__', '__version__']
>>> Bio.__version__
'1.57'
>>> help(Bio)
Help on package Bio:

NAME
    Bio - Collection of modules for dealing with biological data
in Python.

FILE
    /Users/fabrizio/source/biopython-1.57/Bio/__init__.py

DESCRIPTION
    The Biopython Project is an international association of
developers
    of freely available Python tools for computational molecular
biology.

    http://biopython.org


PACKAGE CONTENTS
    Affy (package)
    Align (package)
    AlignIO (package)
    Alphabet (package)
    Application (package)
    Blast (package)
    CAPS (package)
    Clustalw (package)
    Cluster (package)
    Compass (package)
    Crystal (package)
    Data (package)
    DocSQL
    Emboss (package)

:
```

If it is not automatically during the installation, you must set the PYTHONPATH to point at the installation folder, or alternatively you can set the **`sys.path`** (like in the previous example).

9.2 The `Seq` class

The **`Bio`** module **`Seq`** provides high-level data structures to store and process sequence objects. Let's import Seq from Bio:

```
>>> from Bio import Seq
>>> dir(Seq)
['Alphabet', 'CodonTable', 'IUPAC', 'MutableSeq', 'Seq',
'UnknownSeq', '__builtins__', '__doc__', '__docformat__',
'__file__', '__name__', '__package__', '_dna_complement_table',
'_maketrans', '_rna_complement_table', '_test', '_translate_str',
'ambiguous_dna_complement', 'ambiguous_rna_complement', 'array',
'back_transcribe', 'reverse_complement', 'string', 'sys',
'transcribe', 'translate']
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC")
>>> my_seq
Seq('AGCATCGTAGCATGCAC', Alphabet())
>>> print my_seq
AGCATCGTAGCATGCAC
>>> my_seq.alphabet
Alphabet()
>>> dir(my_seq)
['__add__', '__class__', '__cmp__', '__contains__',
'__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__getitem__', '__hash__', '__init__',
'__len__', '__module__', '__new__', '__radd__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_data',
'_get_seq_str_and_check_alphabet', 'alphabet', 'back_transcribe',
'complement', 'count', 'data', 'endswith', 'find', 'lower',
'lstrip', 'reverse_complement', 'rfind', 'rsplit', 'rstrip',
'split', 'startswith', 'strip', 'tomutable', 'tostring',
'transcribe', 'translate', 'ungap', 'upper']
```

The **`Seq.Seq`** objects are associated with an *alphabet* object that specifies the kind of sequence stored into the object. In our example the alphabet is not set, meaning that it is not specified whether the sequence is DNA or protein. Biopython contains a set of precompiled alphabets that cover all biological sequence types. Usually, IUPAC (http://www.chem.qmw.ac.uk/iupac) defined alphabets are the most used. They are:

- IUPACUnambiguousDNA (basic GATC letters)
- IUPACAmbiguousDNA (+ ambiguity letters)
- ExtendedIUPACDNA (+ modified bases)
- IUPACUnambiguousRNA
- IUPACAmbiguousRNA
- IUPACProtein (IUPAC standard AA)
- ExtendedIUPACProtein (+ selenocysteine, X, etc)

```
>>> from Bio.Alphabet import IUPAC
>>> dir(IUPAC)
['Alphabet', 'ExtendedIUPACDNA', 'ExtendedIUPACProtein',
'IUPACAmbiguousDNA', 'IUPACAmbiguousRNA', 'IUPACData',
'IUPACProtein', 'IUPACUnambiguousDNA', 'IUPACUnambiguousRNA',
'__builtins__', '__doc__', '__file__', '__name__', '__package__',
'ambiguous_dna', 'ambiguous_rna', 'extended_dna',
'extended_protein', 'protein', 'unambiguous_dna',
'unambiguous_rna']
>>> IUPAC.unambiguous_dna.letters
'GATC'
>>> IUPAC.unambiguous_rna.letters
'GAUC'
>>> IUPAC.ambiguous_dna.letters
'GATCRYWSMKHBVDN'
>>> IUPAC.extended_dna.letters
'GATCBDSW'
>>> IUPAC.protein.letters
'ACDEFGHIKLMNPQRSTVWY'
```

Now we can create a new instance of **my_seq**, this time specifying that it is indeed a DNA sequence:

```
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGCATCGTAGCATGCAC', IUPACUnambiguousDNA())
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

Methods associated with the **my_seq** object allow basic string manipulation. For example, we can index, slice, split, convert the sequence upper or lower-case, count occurrences of characters, and so on:

```
>>> my_seq[0]
'A'
>>> my_seq[5]
'C'
>>> my_seq[5:10]
Seq('CGTAG', IUPACUnambiguousDNA())
>>> my_seq.split("A")
[Seq('', IUPACUnambiguousDNA()), Seq('GC',
IUPACUnambiguousDNA()), Seq('TCGT', IUPACUnambiguousDNA()),
Seq('GC', IUPACUnambiguousDNA()), Seq('TGC',
IUPACUnambiguousDNA()), Seq('C', IUPACUnambiguousDNA())]
>>> my_seq.count("A")
5
>>> my_seq.count("A")/float(len(my_seq))
0.29411764705882354
```

Note 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 by adding them, but only if their alphabets are compatible (unless the sequences are assigned generic alphabets):

```
>>> my_seq_2= Seq.Seq("CGTC",IUPAC.unambiguous_dna)
>>> my_seq+my_seq_2
Seq('AGCATCGTAGCATGCACCGTC', IUPACUnambiguousDNA())
>>> my_seq_2= Seq.Seq("CGTC",IUPAC.protein)
>>> my_seq+my_seq_2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Bio/Seq.py", line 271, in __add__
TypeError: Incompatible alphabets IUPACUnambiguousDNA() and
IUPACProtein()
```

You can search the sequence for 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 described by regular expression, by using the Python **`re`** module, or by using the Biopython module **`Bio.Motif`** (see Chapter 13 of the Biopython tutorial).

Others methods are designed specifically for biological sequences; for example, you can get the reverse complement of a DNA sequence using the **`reverse_complement`** method, or compute its transcription using the **`transcribe`** method:

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

NB: the **`transcribe`** method just changes the Ts with Us and set the alphabet to RNA. The **`transcribe`** method assumes that the input DNA sequence is the coding strand, and not the template strand. If you have the template strand and want to perform a real transcription, you need to get the reverse complement first and then transcribe it:

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

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

```
>>> prot_seq = Seq.Seq("GETEREEGARCVNMHLAVADEAPIL",IUPAC.protein)
>>> dir(prot_seq)
['__add__', '__class__', '__cmp__', '__contains__',
'__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__getitem__', '__hash__', '__init__',
'__len__', '__module__', '__new__', '__radd__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_data',
'_get_seq_str_and_check_alphabet', 'alphabet', 'back_transcribe',
'complement', 'count', 'data', 'endswith', 'find', 'lower',
'lstrip', 'reverse_complement', 'rfind', 'rsplit', 'rstrip',
'split', 'startswith', 'strip', 'tomutable', 'tostring',
'transcribe', 'translate', 'ungap', 'upper']
>>> prot_seq.transcribe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Bio/Seq.py", line 828, in transcribe
ValueError: Proteins cannot be transcribed!
```

A DNA or RNA sequence objects can be also translated into a protein. To do that, a number of different genetic codes are available. Some of them are assigned names (such as “Standard”, “Vertebrate Mitochondrial”) that can be used to identify them; additionally, all NCBI-defined alphabets are included, and identified by the NCBI table identifier (see http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi). By default, translation will use the standard genetic code (corresponding to the NCBI table id 1)

```
>>> from Bio.Data import CodonTable
>>> print CodonTable.unambiguous_dna_by_name["Standard"]
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
--+---------+---------+---------+---------+--

>>> standard_table = CodonTable.unambiguous_dna_by_id[1]
>>> standard_table
<Bio.Data.CodonTable.NCBICodonTableDNA instance at 0x100540758>
>>> standard_table.start_codons
['TTG', 'CTG', 'ATG']
>>> standard_table.stop_codons
['TAA', 'TAG', 'TGA']
>>> mito_table = CodonTable.unambiguous_dna_by_id[2]
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mito_table.stop_codons
['TAA', 'TAG', 'AGA', 'AGG']
```

The **`translate`** method translates a RNA or DNA sequence using the specified or default genetic code, and returns a **`Seq`** object, whose alphabet might contain additional information, for example regarding the presence of stop codons in the translated sequence:

```
>>> mrna_seq = Seq.Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGAUAG', \
... IUPAC.unambiguous_rna)
>>> mrna_seq.translate(table="Standard")
Seq('MAIVMGR*KG*', HasStopCodon(IUPACProtein(), '*'))
>>> mrna_seq.translate(table=1)
Seq('MAIVMGR*KG*', HasStopCodon(IUPACProtein(), '*'))
>>> dna_seq=mrna_seq.back_transcribe()
>>> dna_seq
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGATAG', IUPACUnambiguousDNA())
>>> dna_seq.translate(table=1)
Seq('MAIVMGR*KG*', HasStopCodon(IUPACProtein(), '*'))
>>> print CodonTable.unambiguous_dna_by_name["Vertebrate
Mitochondrial"]
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
--+---------+---------+---------+---------+--
>>> mrna_seq.translate(table="Vertebrate Mitochondrial")
Seq('MAIVMGRWKG*', HasStopCodon(IUPACProtein(), '*'))
```

In our example, there are two stop codons in the mRNA sequence. The first stop codon is TGA, which is recognized as a triptophan (W) in the mitochondrion. By default, all stop codons encountered during the translation of the nucleic acid sequence are returned as stars ("*"). We can require the translation to stop at the first encountered stop codon:

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

9.3 The `MutableSeq` class

Seq objects behave similarly to Python strings, in the sense that they are immutable. Biopython provides another object, the **`MutableSeq`**, which instead can be changed:

```
>>> my_seq = Seq.Seq("AGCATCGTAGCATG", IUPAC.unambiguous_dna)
>>> my_seq[5]
'C'
>>> my_seq[5]="T"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Seq' object does not support item assignment
>>> my_mut_seq = Seq.MutableSeq("AGCATCGTAGCATG", \
IUPAC.unambiguous_dna)
>>> my_mut_seq
MutableSeq('AGCATCGTAGCATG', IUPACUnambiguousDNA())
>>> my_mut_seq[5] = "T"
>>> my_mut_seq
MutableSeq('AGCATTGTAGCATG', IUPACUnambiguousDNA())
>>> my_seq.reverse()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Seq' object has no attribute 'reverse'
>>> my_mut_seq.reverse()
>>> my_mut_seq
MutableSeq('GTACGATGCTACGA', IUPACUnambiguousDNA())
>>> my_seq.remove("G")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Seq' object has no attribute 'remove'
>>> my_mut_seq.remove("G")
>>> my_mut_seq
MutableSeq('TACGATGCTACGA', IUPACUnambiguousDNA())
```

It is possible to convert an immutable **`Seq`** object into a mutable one, and vice-versa:

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

NB: **`MutableSeq`** objects can not be used as dictionary keys, while **`Seq`** objects can.

9.4 Working with strings

Finally, for those users that would rather work with strings than with **`Seq`** objects, Biopython provides a number of **`Seq`** object methods that can be used on strings as well.

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

9.5 The `SeqRecord` class

The **`SeqRecord`** class is a high-level class that allows higher level features such as identifiers and features to be associated with a sequence. These features are:

- **`seq`**: The sequence itself, typically a Seq object;
- **`id`**: The primary ID used to identify the sequence;
- **`name`**: A "common" name/id for the sequence;
- **`description`**: A human readable description or expressive name for the sequence;
- **`letter_annotations`**: Holds per-letter-annotations using a (restricted) dictionary of additional information about the letters in the sequence. The keys are the name of the information, and the information is contained in the value as a Python sequence (i.e. a list, tuple or string) with the same length as the sequence itself (used for example for quality scores, secondary structure, etc.);
- **`annotations`**: A dictionary of additional information about the sequence. The keys are the name of the information, and the information is contained in the value;
- **`features`**: A list of SeqFeature objects (we'll see them later), with more structured information about the features on a sequence (e.g. position of genes on a genome, or domains on a protein sequence);
- **`dbxrefs`**: A list of database cross-references.

Such features can be created "by hand" by the user, or imported from a database record (for example a GenBank or SwissProt file). Let's see how to create a **`SeqRecord`** object associated to a **`Seq`** object:

```
>>> import Bio
>>> from Bio import Seq
>>> from Bio import SeqRecord
>>> my_seq = Seq.Seq("CGACGGACTCAG", IUPAC.unambiguous_dna)
>>> my_seq_r = SeqRecord.SeqRecord(my_seq)
>>> my_seq_r
SeqRecord(seq=Seq('CGACGGACTCAG', Alphabet()), id='<unknown id>',
name='<unknown name>', description='<unknown description>',
dbxrefs=[])
>>> my_seq_r.seq
Seq('CGACGGACTCAG', Alphabet())
>>> my_seq_r.id
'<unknown id>'
>>> my_seq_r.id = "AC12345"
>>> my_seq_r.description = "My gene"
```

Features can be added to the **SeqRecord** object directly when it is instanced:

```
>>> my_seq_r = SeqRecord.SeqRecord(my_seq, \
... id="AC12345",description="My gene")
```

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

```
>>> my_seq_r.annotations
{}
>>> my_seq_r.annotations["origin"] = "mouse"
>>> my_seq_r.annotations["date"] = "2011-04-28"
>>> my_seq_r.annotations
{'origin': 'mouse', 'date': '2011-04-28'}
>>> print my_seq_r.annotations["origin"]
mouse
```

Similarly, the **letter_annotations** attribute is an empty dictionary whose values must be lists of tuples of exactly the same length of the sequence:

```
>>> my_seq_r.letter_annotations["phred scores"] = \
... [20,34,40,12,22,20,20,20,34,40,12,22]
>>> my_seq_r.letter_annotations["conservation"] = \
... [0.5,0.3,1,1,0.8,0.9,1,1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Bio/SeqRecord.py", line 35, in __setitem__
TypeError: We only allow python sequences (lists, tuples or
strings) of length 12.
```

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

```
>>> print my_seq_r.format("fasta")
>AC12345 My gene
CGACGGACTCAG
>>> print my_seq_r.format("genbank")
LOCUS       AC12345                   12 bp    DNA
UNK 01-JAN-1980
DEFINITION  My gene
ACCESSION   AC12345
VERSION     AC12345
KEYWORDS    .
SOURCE      .
  ORGANISM  .
            .
FEATURES             Location/Qualifiers
ORIGIN
        1 cgacggactc ag
//
```

9.5.1 Slicing and adding `SeqRecord` objects

If you are interested in only a particular region of a gene or a protein, you can slice its sequence but also its associated **`SeqRecord`** object. Note that some features, such as **`annotations`** and **`dbxrefs`**, are not extended to the sliced **`SeqRecord`**.

```
>>> left_seq_r = my_seq_r[:5]
>>> left_seq_r.seq
Seq('CGACG', IUPACUnambiguousDNA())
>>> left_seq_r.letter_annotations["phred scores"]
[20, 34, 40, 12, 22]
>>> left_seq_r.annotations
{}
```

Two **`SeqRecord`** objects can be concatenated by adding them into a new **`SeqRecord`**. Some features will be inherited by the new object, in some cases only if they were identical in the two parent **`SeqRecord`** objects (for example the **`id`** attribute). Some other features are concatenated if present in both parent objects (for example identical keys in the **`letter_annotations`** attribute), while others are not inherited in any case (such as **`annotations`**).

```
>>> my_seq_2 = \
... Seq.Seq("ACGTGTCATACA",IUPAC.unambiguous_dna)
>>> my_seq_r2 = SeqRecord.SeqRecord(my_seq_2, \
... id="AC12345",description="A gene fragment")
>>> my_seq_r2.letter_annotations["phred scores"] \
... = [10,20,30,12,15,21,23,45,32,32,30,11]
>>> my_seq_c = my_seq_r + my_seq_r2
>>> my_seq_c.id
'AC12345'
>>> my_seq_c.letter_annotations
{'phred scores': [20, 34, 40, 12, 22, 20, 20, 20, 34, 40, 12, 22,
10, 20, 30, 12, 15, 21, 23, 45, 32, 32, 30, 11]}
>>> my_seq_c.annotations
{}
```