Introduction to Biopython

Python libraries for computational molecular biology

http://www.biopython.org

Biopython functionality and tools

- The ability to parse bioinformatics files into Python utilizable data structures
- Support the following formats:
  - Blast output
  - Clustalw
  - FASTA
  - PubMed and Medline
  - ExPASy files
  - SCOP
  - SwissProt
  - PDB
- Files in the supported formats can be iterated over record by record or indexed and accessed via a dictionary interface

Biopython functionality and tools

- Code to deal with on-line bioinformatics destinations (NCBI, ExPASy)
- Interface to common bioinformatics programs (Blast, ClustalW)
- A sequence obj dealing with seqs, seq IDs, seq features
- Tools for operations on sequences
- Tools for dealing with alignments
- Tools to manage protein structures
- Tools to run applications

The Biopython module name is **Bio**

It must be downloaded and installed (**http://biopython.org/wiki/Download**)

You need to install **numpy** first

```
>>>import Bio
```

Program

- **Introduction to Biopython**
  - Sequence objects (I)
  - Sequence Record objects (I)
  - Protein structures (PDB module) (II)
- **Working with DNA and protein sequences**
  - Transcription and Translation
- **Extracting information from biological resources**
  - Parsing Swiss-Prot files (I)
  - Parsing BLAST output (I)
  - Accessing NCBI's Entrez databases (II)
  - Parsing Medline records (II)
- **Running external applications (e.g. BLAST) locally and from a script**
  - Running BLAST over the Internet
  - Running BLAST locally
- **Working with motifs**
  - Parsing PROSITE records
  - Parsing PROSITE documentation records

Introduction to Biopython (I)

- Sequence objects
- Sequence Record objects

Sequence Object

- Seq objects vs Python strings:
  - They have different methods
  - The Seq object has the attribute **`alphabet`** (biological meaning of Seq)

```
>>> import Bio
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> print my_seq
Seq('AGTACACTGGT', Alphabet())
>>> my_seq.alphabet
Alphabet()
>>>
```

The **`alphabet`** attribute

- Alphabets are defined in the **`Bio.Alphabet`** module
- We will use the IUPAC alphabets (http://www.chem.qmw.ac.uk/iupac)
- **`Bio.Alphabet.IUPAC`** provides definitions for DNA, RNA and proteins + provides extension and customization of basic definitions:
  - **`IUPACProtein (IUPAC standard AA)`**
  - **`ExtendedIUPACProtein (+ selenocysteine, X, etc)`**
  - **`IUPACUnambiguousDNA (basic GATC letters)`**
  - **`IUPACAmbiguousDNA (+ ambiguity letters)`**
  - **`ExtendedIUPACDNA (+ modified bases)`**
  - **`IUPACUnambiguousRNA`**
  - **`IUPACAmbiguousRNA`**

The **alphabet** attribute

```
>>> import Bio
>>> 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()
>>> my_seq = Seq("AGTACACTGGT", IUPAC.protein)
>>> my_seq
Seq('AGTACACTGGT', IUPACProtein())
>>> my_seq.alphabet
IUPACProtein()
>>>
```

Sequences act like strings

```
>>> my_seq = Seq("AGTAACCCTTAGCACTGGT", IUPAC.unambiguous_dna)
>>> for index, letter in enumerate(my_seq):
...     print index, letter
...
0 A
1 G
2 T
3 A
4 A
5 C
...etc
>>> print len(my_seq)
19
>>> print my_seq[0]
A
>>> print my_seq[2:10]
Seq('TAACCCTT', IUPACProtein())
>>> my_seq.count('A')
5
>>> 100*float(my_seq.count('C')+my_seq.count('G'))/len(my_seq)
47.368421052631582
```

Turn Seq objects into strings

You may need the plain sequence string (e.g. to write to a file or to insert into a database)

```
>>> my_seq = Seq("AGTAACCCTTAGCACTGGT", IUPAC.unambiguous_dna)
>>>>>> str(my_seq)
'AGTAACCCTTAGCACTGGT'
>>> print my_seq
AGTAACCCTTAGCACTGGT
>>> fasta_format_string = ">DNA_id\n%s\n"% my_seq
>>> print fasta_format_string
>DNA_id
AGTAACCCTTAGCACTGGT
# Biopython 1.44 or older
>>>my_seq.tostring()
'AGTAACCCTTAGCACTGGT'
```

Concatenating sequences

You can't add sequences with incompatible alphabets (protein sequence and DNA sequence)

```
>>> dna_seq = Seq("AGTAACCCTTAGCACTGGT", IUPAC.unambiguous_dna)
>>> protein_seq = Seq("KSMKPPRTHLIMHWIIL", IUPAC.IUPACProtein())
>>> protein_seq + dna_seq
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/home/abarbato/biopython-1.53/build/lib.linux-x86_64-
2.4/Bio/Seq.py", line 216, in __add__
    raise TypeError("Incompatable alphabets %s and %s" \
TypeError: Incompatable alphabets IUPACProtein() and
IUPACUnambiguousDNA()
BUT, if you give generic alphabet to dna_seq and protein_seq:
>>> from Bio.Alphabet import generic_alphabet
>>> dna_seq.alphabet = generic_alphabet
>>> protein_seq.alphabet = generic_alphabet
>>> protein_seq + dna_seq
Seq('KSMKPPRTHLIMHWIILAGTAACCCTTAGCACTGGT', Alphabet())
```

Changing case

Seq objects have **`upper()`** and **`lower()`** methods

```
>>> from Bio.Alphabet import generic_dna
>>> dna_seq = Seq("acgtACGT", generic_dna)
>>> dna_seq.upper()
Seq('ACGTACGT', DNAAlphabet())
>>> dna_seq.lower()
Seq('acgtacgt', DNAAlphabet())
>>>
```

Note that the IUPAC alphabets are for upper case only

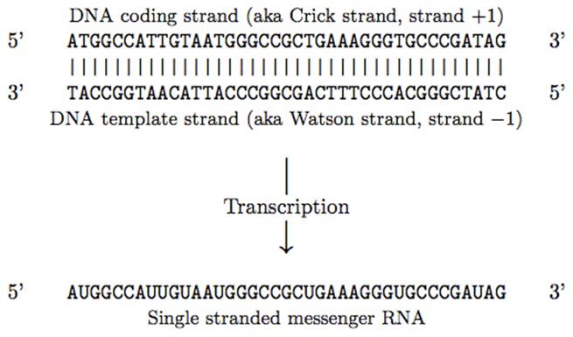
Nucleotide sequences and (reverse) complements

Seq objects have **`upper()`** and **`lower()`** methods

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> dna_seq = Seq("AGTAACCCTTAGCACTGGT", IUPAC.unambiguous_dna)
>>> dna_seq.complement()
Seq('TCATTGGGAATCGTGACCA', IUPACUnambiguousDNA())
>>> dna_seq.reverse_complement()
Seq('ACCAGTGCTAAGGGTTACT', IUPACUnambiguousDNA())
```

Note that these operations are not allowed with protein alphabets

Transcription



Transcription

```
>>> coding_dna =
Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG",
IUPAC.unambiguous_dna)
>>> template_dna = coding_dna.reverse_complement()
>>> template_dna
Seq('CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT',
IUPACUnambiguousDNA())
>>> messenger_rna = coding_dna.transcribe()
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG',
IUPACUnambiguousRNA())
```

Note: all this does is a switch T --> U and adjust the alphabet.

The Seq object also includes a back-transcription method:

```
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG',
IUPACUnambiguousDNA())
```

Translation

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG',
IUPAC.unambiguous_rna)
>>> 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.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
>>>
```

Translation with different translation tables

Translation tables available in Biopython are based on those from the NCBI.

By default, translation will use the standard genetic code (NCBI table id 1)

If you deal with mitochondrial sequences:

```
>>> coding_dna.translate(table="Vertebrate Mitochondrial")
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
>>> coding_dna.translate(table=2)
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

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

```
>>> coding_dna.translate(to_stop = True)
Seq('MAIVMGR', IUPACProtein())
>>> coding_dna.translate(table=2,to_stop = True)
Seq('MAIVMGRWKGAR', IUPACProtein())
```

Translation tables

Translation tables available in Biopython are based on those from the NCBI.

`http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi`

By default, translation will use the standard genetic code (NCBI table id 1)

```
>>> from Bio.Data import CodonTable
>>> standard_table =
CodonTable.unambiguous_dna_by_name["Standard"]
>>> mito_table =
CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
#Using the NCB table ids:
>>>standard_table = CodonTable.unambiguous_dna_by_id[1]
>>> mito_table = CodonTable.unambiguous_dna_by_id[2]
```

Translation tables

```
>>>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
--+---------+---------+---------+---------+--
```

Translation tables

```
>>> 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
--+---------+---------+---------+---------+--
```

MutableSeq objects

Like Python strings, Seq objects are immutable

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq =
Seq('CGCGCGGGTTTATGATGACCCAAATATAGAGGGCACAC',
IUPAC.unambiguous_dna)
>>> my_seq[5] = 'A'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

However, you can convert it into a mutable sequence (a **`MutableSeq`** object)

```
>>> mutable_seq = my_seq.tomutable()
>>> mutable_seq
MutableSeq('CGCGCGGGTTTATGATGACCCAAATATAGAGGGCACAC',
IUPACUnambiguousDNA())
```

MutableSeq objects

You can create a mutable object directly

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq =
MutableSeq('CGCGCGGGTTTATGATGACCCAAATATAGAGGGCACAC',
IUPAC.unambiguous_dna)
>>> mutable_seq[5] = 'A'
>>> mutable_seq
MutableSeq('CGCGCAGGTTTATGATGACCCAAATATAGAGGGCACAC',
IUPACUnambiguousDNA())
```

A **`MutableSeq`** object can be easily converted into a read-only sequence:

```
>>> new_seq = mutable_seq.toseq()
>>> new_seq
Seq('CGCGCAGGTTTATGATGACCCAAATATAGAGGGCACAC',
IUPACUnambiguousDNA())
```

Sequence Record objects

The **`SeqRecord`** class is defined in the **`Bio.SeqRecord`** module

This class allows higher level features such as identifiers and features to be associated with a sequence

```
>>> from Bio.SeqRecord import SeqRecord
>>> help(SeqRecord)
```

class SeqRecord(__builtin__.object)
A SeqRecord object holds a sequence and information about it.

Main attributes:
id - Identifier such as a locus tag (string)
seq - The sequence itself (Seq object or similar)

Additional attributes:
name - Sequence name, e.g. gene name (string)
description - Additional text (string)
dbxrefs - List of db cross references (list of strings)
features - Any (sub)features defined (list of SeqFeature objects)
annotations - Further information about the whole sequence (dictionary)

Most entries are strings, or lists of strings.

letter_annotations -
Per letter/symbol annotation (restricted dictionary). This holds Python sequences (lists, strings or tuples) whose length matches that of the sequence. A typical use would be to hold a list of integers representing sequencing quality scores, or a string representing the secondary structure.

You will typically use **Bio.SeqIO** to read in sequences from files as **SeqRecord** objects. However, you may want to create your own **SeqRecord** objects directly:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> TMP = Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF')
>>> TMP_r = SeqRecord(TMP)
>>> TMP_r.id
'<unknown id>'
>>> TMP_r.id = 'YP_025292.1'
>>> TMP_r.description = 'toxic membrane protein'
>>> print TMP_r
ID: YP_025292.1
Name: <unknown name>
Description: toxic membrane protein
Number of features: 0
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF',
Alphabet())
>>> TMP_r.seq
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF',
Alphabet())
```

You can also create your own **SeqRecord** objects as follows:

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Alphabet import IUPAC
>>> record
SeqRecord(seq=Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQ
TEVAVF', IUPACProtein()), id='YP_025292.1', name='HokC',
description='toxic membrane protein', dbxrefs=[])
>>> print record
ID: YP_025292.1
Name: HokC
Description: toxic membrane protein
Number of features: 0
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF',
IUPACProtein())
>>>
```

The **format()** method

It returns a string containing your cord formatted using one of the output file formats supported by **Bio.SeqIO**

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Alphabet import generic_protein
>>> rec =
SeqRecord(Seq("MGSNKSKPKDASQRRRSLEPSENVHGAGGAFPASQTPSKPASADGHRGPSA
AFVPPAAEPKLFGGFNSSDTVTSPQRAGALAGGVTTFVALYDYESRTETDLSFKKGERLQIVNNTR
KVDVREGDWWLAHSLSTGQTGYIPS", generic_protein), id = "P05480",
description = "SRC_MOUSE Neuronal proto-oncogene tyrosine-protein
kinase Src: MY TEST")
>>> print rec.format("fasta")
>P05480 SRC_MOUSE Neuronal proto-oncogene tyrosine-protein kinase
Src: MY TEST
MGSNKSKPKDASQRRRSLEPSENVHGAGGAFPASQTPSKPASADGHRGPSAAFVPPAAEP
KLFGGFNSSDTVTSPQRAGALAGGVTTFVALYDYESRTETDLSFKKGERLQIVNNTRKVD
VREGDWWLAHSLSTGQTGYIPS
```

```
Seq1 "ACTGGGAGCTAGC"
Seq2 "TTGATCGATCGATCG"
Seq3 "GTGTAGCTGCT"
```

INPUT FILE

```
F = open("input.txt")
for line in F:
        <parse line>
        <get seq id>
        <get description>
        <get sequence>
        <get other info>
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein
rec = SeqRecord(Seq(<sequence>, alphabet),id
= <seq_id>, description = <description>)
Format_rec = rec.format("fasta")
Out.write(Format_rec)
```

SCRIPT.py

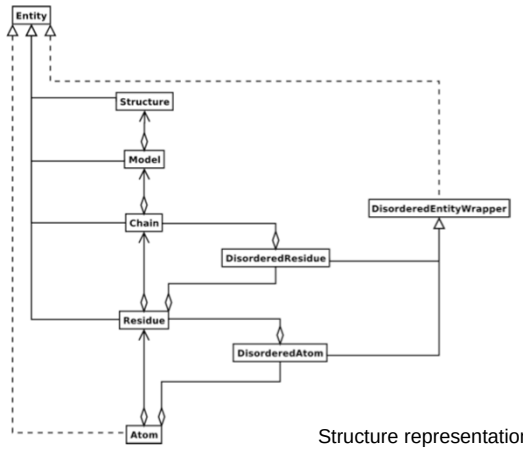
OUTPUT FILE

```
>P05480 SRC_MOUSE Neuronal proto-oncogene tyrosine-protein
kinase Src: MY TEST
MGSNKSKPKDASQRRRSLEPSENVHGAGGAFPASQTPSKPASADGHRGPSAAFVPPAAEP
KLFGGFNSSDTVTSPQRAGALAGGVTTFVALYDYESRTETDLSFKKGERLQIVNNTRKVD
```

Protein Structures

Protein structures (PDB module)

Structure
Model
Chain
Residue
Atom



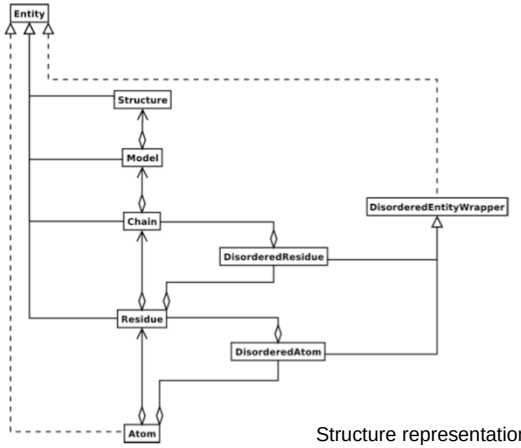
Structure representation

```
ATOM      1  N   PRO A 134      33.747  46.582  69.532  1.00 56.70           N
ATOM      2  CA  PRO A 134      34.692  45.947  70.456  1.00 55.76           C
ATOM      3  C   PRO A 134      35.040  44.558  69.953  1.00 55.49           C
ATOM      4  O   PRO A 134      34.894  43.546  70.722  1.00 55.72           O
ATOM      5  CB  PRO A 134      33.839  45.768  71.707  1.00 56.31           C
ATOM      6  CG  PRO A 134      32.384  45.405  71.086  1.00 56.74           C
ATOM      7  CD  PRO A 134      32.401  45.963  69.637  1.00 56.69           C
ATOM      8  N   ALA A 135      35.457  44.461  68.683  1.00 53.77           N
ATOM      9  CA  ALA A 135      35.892  43.141  68.150  1.00 52.13           C
ATOM     10  C   ALA A 135      37.099  42.612  68.967  1.00 50.42           C
ATOM     11  O   ALA A 135      37.255  41.387  69.178  1.00 51.42           O
ATOM     12  CB  ALA A 135      36.196  43.216  66.650  1.00 51.84           C
ATOM     13  N   SER A 136      37.939  43.538  69.439  1.00 48.55           N
ATOM     14  CA  SER A 136      38.987  43.178  70.385  1.00 46.48           C
ATOM     15  C   SER A 136      38.789  43.888  71.746  1.00 45.81           C
ATOM     16  O   SER A 136      38.568  45.056  71.764  1.00 44.23           O
ATOM     17  CB  SER A 136      40.330  43.486  69.763  1.00 46.19           C
ATOM     18  OG  SER A 136      41.344  43.236  70.671  1.00 42.35           O
...
ATOM   1885  N   ALA B 135      38.255  -6.602 103.702  1.00 44.10           N
ATOM   1886  CA  ALA B 135      37.061  -6.004 102.994  1.00 44.16           C
ATOM   1887  C   ALA B 135      37.338  -5.794 101.533  1.00 44.32           C
ATOM   1888  O   ALA B 135      36.433  -5.890 100.696  1.00 45.88           O
ATOM   1889  CB  ALA B 135      36.673  -4.681 103.600  1.00 43.74           C
ATOM   1890  N   SER B 136      38.566  -5.418 101.235  1.00 43.63           N
ATOM   1891  CA  SER B 136      38.999  -5.265  99.848  1.00 42.10           C
ATOM   1892  C   SER B 136      40.151  -6.245  99.680  1.00 42.68           C
ATOM   1893  O   SER B 136      41.069  -6.309 100.494  1.00 43.55           O
ATOM   1894  CB  SER B 136      39.485  -3.837  99.612  1.00 41.77           C
```

Introduction to Biopython (II)

Protein structures (PDB module)

Structure
Model
Chain
Residue
Atom



Structure representation

Protein structures (PDB module)

This data structure is indispensable for a good interpretation of the data present in a file that describes the structure (PDB or MMCIF)

If the hierarchy cannot represent the content of a structure file, it is fairly certain that the file contains an error or at least does not describe the structure unambiguously.

If a SMCRA cannot be generated, there is reason to suspect a problem

```
child_entity = parent_entity[child_id]
```

Model Chain Residue Atom	Structure Model Chain Residue	Model = 0 Chain = A Residue = ‘SER’ Atom = ‘CA’

```
child_list = parent_entity.get_list()
```

Structure Model Chain Residue	Model Chain Residue Atom

```
parent_entity = child_entity.get_parent()
```

```
full_id = residue.get_full_id()
print full_id
(“1H8A”, 0, “A”, (“”, 10, “A”)
```

The structure with id “1H8A”
The Model with id 0
The Chain with id “A”
The Residue with id (“ ”, 10, “A”)

```
#Get the entity’s id
entity.get_id()
#Check if there is a child with a given id
entity.has_id(entity_id)
#Get number of children
nr_chikdren = len(entity)
```

Structure object
It is at the top of the hierarchy. Its id is a user given string. It contains a number of Model children.

Model object
Its id is an integer, which is derived from the position of the model in the parsed file. It stores a list of Chain children.

Chain object
Its id is the chain identifier in the structure file, and can be any string. Each chain in a Model has a unique id. It stores a list of Residue children.

Residue object
It contains a string with residue name (“ALA”). Its id is composed of 3 parts: **(hetero field, sequence identifier, insertion code)**. It stores a set of Atom children.

Atom object
It stores the data associated with an atom and has no children. Its id is the atom name (“CA”). Its objects can be used to extract specific information.

```
from Bio.PDB.PDBParser import PDBParser
p = PDBParser(PERMISSIVE = 1)
structure_id = '2H8L'
filename = '2H8L.pdb'
s = p.get_structure(structure_id, filename)
```

```
#print p.get_header()
#print p.get_trailer()
first_model = s[0]
print type(first_model)
```

```
chain_A = first_model["A"]
print chain_A
```

```
#print chain_A.child_dict.keys()
for k in chain_A.child_dict.keys():
        print chain_A[k], chain_A[k].get_id(), chain_A[k].get_resname()
        print chain_A[k].get_list()
#print chain_A.get_list()
```

```
res266 = chain_A[266]
#print dir(res266)
print res266.get_id()
print res266.get_resname()
print res266.get_segid()
for atom in res266.get_list():
    print atom.get_name(),atom.get_id(),atom.get_coord(),atom.get_bfactor()
```

```
from Bio.PDB.PDBParser import PDBParser
from math import sqrt

def distance(coords1, coords2):
        dx = (coords1[0] - coords2[0])**2
        dy = (coords1[1] - coords2[1])**2
        dz = (coords1[2] - coords2[2])**2
        RMSD = sqrt(dx + dy + dz)
        return RMSD

p = PDBParser(PERMISSIVE = 1)
structure_id = '2H8L'
filename = '2H8L.pdb'
s = p.get_structure(structure_id, filename)
first_model = s[0]
chain_A = first_model["A"]
chain_B = first_model["B"]
for resA in chain_A.get_list():
        if resA.has_id("CA"):
                caA = resA["CA"]
                coordsA = caA.get_coord()
                #print caA, coordsA
                for resB in chain_B.get_list():
                        if resB.has_id("CA"):
                                caB = resB["CA"]
                                coordsB = caB.get_coord()
                                D = distance(coordsA,coordsB)
                                if D < 6: print resA.resname, resA.get_id()[1],\
                                                resB.resname, resB.get_id()[1], D
```

Extracting information from biological resources: parsing Swiss-Prot files, PDB files, ENSEMBLE records, blast output files, etc.

- Sequence I/O
  - Parsing or Reading Sequences
  - Writing Sequence Files

`Bio.SeqIO`

A simple interface for working with assorted file formats in a uniform way

```
>>>from Bio import SeqIO
>>>help(SeqIO)
```

Bio.SeqIO.parse()

Reads in sequence data as SeqRecord objects.
It expects two arguments.

- A *handle* to read the data form. It can be:
  - a file opened for reading
  - the output from a command line program
  - data downloaded from the internet

- A lower case string specifying the sequence format (see http://biopython.org/wiki/SeqIO for a full listing of supported formats).

The object returned by `Bio.SeqIO` is an iterator which returns `SeqRecord` objects

```
>>> from Bio import SeqIO
>>> handle = open("P05480.fasta")
>>> for seq_rec in SeqIO.parse(handle, "fasta"):
...     print seq_rec.id
...     print repr(seq_rec.seq)
...     print len(seq_rec)
...
sp|P05480|SRC_MOUSE
Seq('MGSNKSKPKDASQRRRSLERGPSA...ENL', SingleLetterAlphabet())
541
>>> handle.close()
```

```
>>> for seq_rec in SeqIO.parse(handle, "genbank"):
...     print seq_rec.id
...     print repr(seq_rec.seq)
...     print len(seq_rec)
...
U49845.1
Seq('GATCCTCCATATACAACGGTACGGAA...ATC', IUPACAmbiguousDNA())
5028
>>> handle.close()
```

Candida albicans genomic DNA, chromosome 7, complete sequence

```
>>> from Bio import SeqIO
>>> handle = open("AP006852.gbk")
>>> for seq_rec in SeqIO.parse(handle, "genbank"):
...     print seq_rec.id
...     print repr(seq_rec.seq)
...     print len(seq_rec)
...
AP006852.1
Seq('CCACTGTCCAATACCCCCAACAGGAAT...TGT', IUPACAmbiguousDNA())
949626
>>>
```

Using list comprehension:

```
>>>handle = open("AP006852.gbk")
>>>identifiers=[seq_rec.id for seq_rec in SeqIO.parse(handle,"genbank")]
>>>handle.close()
>>>identifiers
['AP006852.1']
>>>
```

Here we do it using the **`sprot_prot.fasta`** file

```
>>> from Bio import SeqIO
>>> handle = open("sprot_prot.fasta")
>>> ids = [seq_rec.id for seq_rec in SeqIO.parse(handle,"fasta")]
>>> ids
['sp|P24928|RPB1_HUMAN', 'sp|Q9NVU0|RPC5_HUMAN',
'sp|Q9BUI4|RPC3_HUMAN', 'sp|Q9BUI4|RPC3_HUMAN',
'sp|Q9NW08|RPC2_HUMAN', 'sp|Q9H1D9|RPC6_HUMAN',
'sp|P19387|RPB3_HUMAN', 'sp|O14802|RPC1_HUMAN',
'sp|P52435|RPB11_HUMAN', 'sp|O15318|RPC7_HUMAN',
'sp|P62487|RPB7_HUMAN', 'sp|O15514|RPB4_HUMAN',
'sp|Q9GZS1|RPA49_HUMAN', 'sp|P36954|RPB9_HUMAN',
'sp|Q9Y535|RPC8_HUMAN', 'sp|O95602|RPA1_HUMAN',
'sp|Q9Y2Y1|RPC10_HUMAN', 'sp|Q9H9Y6|RPA2_HUMAN',
'sp|P78527|PRKDC_HUMAN', 'sp|O15160|RPAC1_HUMAN',…,
'sp|Q9BWH6|RPAP1_HUMAN']
>>> ]
```

Iterating over the records in a sequence file

Instead of using a for loop, you can also use the **`next()`** method of an iterator to step through the entries

```
>>> handle = open("sprot_prot.fasta")
>>> rec_iter = SeqIO.parse(handle, "fasta")
>>> rec_1 = rec_iter.next()
>>> rec_1
SeqRecord(seq=Seq('MHGGGPPSGDSACPLRTIKRVQFGVLSPDELKRMSVTEGGIKYPET
TEGGRPKL...EEN', SingleLetterAlphabet()),
id='sp|P24928|RPB1_HUMAN', name='sp|P24928|RPB1_HUMAN',
description='sp|P24928|RPB1_HUMAN DNA-directed RNA polymerase II
subunit RPB1 OS=Homo sapiens GN=POLR2A PE=1 SV=2', dbxrefs=[])
>>> rec_2 = rec_iter.next()
>>> rec_2
SeqRecord(seq=Seq('MANEEDDPVVQEIDVYLAKSLAEKLYLFQYPVRPASMTYDDIPHLS
AKIKPKQQ...VQS', SingleLetterAlphabet()),
id='sp|Q9NVU0|RPC5_HUMAN', name='sp|Q9NVU0|RPC5_HUMAN',
description='sp|Q9NVU0|RPC5_HUMAN DNA-directed RNA polymerase III
subunit RPC5 OS=Homo sapiens GN=POLR3E PE=1 SV=1', dbxrefs=[])
>>> handle.close()
```

`Bio.SeqIO.read()`

If your file has one and only one record (e.g. a GenBank file for a single chromosome), then use the **`Bio.SeqIO.read()`**.
This will check there are no extra unexpected records present

```
>>> rec_iter = SeqIO.parse(open("1293613.gbk"), "genbank")
>>> rec = rec_iter.next()
>>> print rec
ID: U49845.1
Name: SCU49845
Description: Saccharomyces cerevisiae TCP1-beta gene, partial cds, and Axl2p
(AXL2) and Rev7p (REV7) genes, complete cds.
Number of features: 6
/sequence_version=1
/source=Saccharomyces cerevisiae (baker's yeast)
/taxonomy=['Eukaryota', 'Fungi', 'Ascomycota', 'Saccharomycotina',
'Saccharomycetes', 'Saccharomycetales', 'Saccharomycetaceae', 'Saccharomyces']
/keywords=['']
/references=[Reference(title='Cloning and sequence of REV7, a gene whose function
is required for DNA damage-induced mutagenesis in Saccharomyces cerevisiae', ...),
Reference(title='Selection of axial growth sites in yeast requires Axl2p, a novel
plasma membrane glycoprotein', ...), Reference(title='Direct Submission', ...)]
/accessions=['U49845']
/data_file_division=PLN
/date=21-JUN-1999
/organism=Saccharomyces cerevisiae
/gi=1293613
Seq('GATCCTCCATATACAACGGTATCTCCACCTCAGGTTTAGATCTCAACAACGGAA...ATC',
IUPACAmbiguousDNA())
```

Sequence files as lists

```
>>> from Bio import SeqIO
>>> handle = open("ncbi_gene.fasta")
>>> records = list(SeqIO.parse(handle, "fasta"))
>>> >>> records[-1]
SeqRecord(seq=Seq('gggggggggggggggggatcactctctttcagtaacctcaac...c
cc', SingleLetterAlphabet()), id='A10421', name='A10421',
description='A10421 Synthetic nucleotide sequence having a human
IL-2 gene obtained from pILOT135-8. : Location:1..1000',
dbxrefs=[])
```

Sequence files as dictionaries

```
>>> handle = open("ncbi_gene.fasta")
>>> records = SeqIO.to_dict(SeqIO.parse(handle, "fasta"))
>>> handle.close()
>>> records.keys()
['M69013', 'M69012', 'AJ580952', 'J03005', 'J03004', 'L13858',
'L04510', 'M94539', 'M19650', 'A10421', 'AJ002990', 'A06663',
'A06662', 'S62035', 'M57424', 'M90035', 'A06280', 'X95521',
'X95520', 'M28269', 'S50017', 'L13857', 'AJ345013', 'M31328',
'AB038040', 'AB020593', 'M17219', 'DQ854814', 'M27543', 'X62025',
'M90043', 'L22075', 'X56614', 'M90027']
>>> seq_record = records['X95521']
'X95521 M.musculus mRNA for cyclic nucleotide phosphodiesterase :
Location:1..1000'
```

Parsing sequences from the net

Handles are not always from files

Parsing GenBank records from the net

```
>>>from Bio import Entrez
>>>from Bio import SeqIO
>>>handle = Entrez.efetch(db="nucleotide",rettype="fasta",id="6273291")
>>>seq_record = SeqIO.read(handle,”fasta”)
>>>handle.close()
>>>seq_record.description
```

Parsing SwissProt sequence from the net

```
>>>from Bio import ExPASy
>>>from Bio import SeqIO
>>>handle = ExPASy.get_sprot_raw("6273291")
>>>seq_record = SeqIO.read(handle,”swiss”)
>>>handle.close()
>>>print seq_record.id
>>>print seq_record.name
>>>prin seq_record.description
```

Indexing really large files

`Bio.SeqIO.index()` returns a dictionary without keeping everything in memory.

It works fine even for million of sequences

The main drawback is less flexibility: it is read-only

```
>>> from Bio import SeqIO
>>> recs_dict = SeqIO.index("ncbi_gene.fasta", "fasta")
>>> len(recs_dict)
34
>>> recs_dict.keys()
['M69013', 'M69012', 'AJ580952', 'J03005', 'J03004', 'L13858', 'L04510',
'M94539', 'M19650', 'A10421', 'AJ002990', 'A06663', 'A06662', 'S62035',
'M57424', 'M90035', 'A06280', 'X95521', 'X95520', 'M28269', 'S50017',
'L13857', 'AJ345013', 'M31328', 'AB038040', 'AB020593', 'M17219', 'DQ854814',
'M27543', 'X62025', 'M90043', 'L22075', 'X56614', 'M90027']
>>> print recs_dict['M57424']
ID: M57424
Name: M57424
Description: M57424 Human adenine nucleotide translocator-2 (ANT-2) gene,
complete cds. : Location:1..1000
Number of features: 0
Seq('gagctctggaatagaatacagtagaggcatcatgctcaaagagagtagcagatg...agc',
SingleLetterAlphabet())
```

Writing sequence files

`Bio.SeqIO.write()`

This function takes three arguments:

1. some SeqRecord objects
2. a handle to write to
3. a sequence format

```
from Bio.Seq import Seq
from Bio.SeqRecors import SeqRecord
from Bio.Alphabet import generic_protein
Rec1 = SqRecord(Seq("ACCA…",generic_protein), id="1", description="")
Rec1 = SqRecord(Seq("CDRFAA",generic_protein), id="2", description="")
Rec1 = SqRecord(Seq("GRKLM",generic_protein), id="3", description="")
My_records = [Rec1, Rec2, Rec3]
from Bio import SeqIO
handle = open("MySeqs.fas","w")
SeqIO.write(My_records, handle, "fasta")
handle.close()
```

Converting between sequence file formats

We can do file conversion by combining **`Bio.SeqIO.parse()`** and **`Bio.SeqIO.write()`**

```
from Bio import SeqIO
>>> In_handle = open ("AP006852.gbk", "r")
>>> Out_handle = open("AP006852.fasta", "w")
>>> records = SeqIO.parse(In_handle, "genbank")
>>> count = SeqIO.write(records, Out_handle, "fasta")
>>> count
1
>>>
>>> In_handle.close()
>>> Out_handle.close()
```

Running external applications (e.g. BLAST) locally and from a script

Running BLAST locally

```
blastProgram -p ProgramName -d Database -i QueryFile -o  BLASTreportOutputFile
```

Using the shell command line

```
$blastall -p blastp -d nr -i P05480.fasta -o Blastall.out
```

From a script

Using os.system()

```
>>>import os
>>>S = "blastall -p blastp -d nr -i P05480.fasta -o Blastall.out"
>>>os.system(S)
```

Using Biopython

```
>>>from Bio.Blast.Applications import BlastallCommandline
>>>cline = BlastallCommandline(program ="blastp",
>>>infile="P05480.fasta", database="nr", output="Blastall.out" )
>>> print cline
>>>blastall -d nr -i P05480.fasta -o Blastall.out -p blastp
>>>os.system(str(cline))
```

Running BLAST over the Internet

Bio.Blast.NCBIWWW

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn","nr","8332116")
>>> save_file = open("qblast_blastn.out", "w")
>>> save_file.write(result_handle.read())
>>> save_file.close()
>>> result_handle.close()
```

Parsing the BLAST output

You can get BLAST output in XML in various ways: for the parser it does not matter how the output was generated as long as it is in XML format

- Use Biopython to run BLAST over the internet
- Use Biopython to run BLAST locally
- Do the BLAST search yourself on the NCBI site through your web browser, and then save the results (choose XML format for the output)
- Run BLAST locally without using Biopython, and save the output in a file (choose XML format for the output file)

Parsing the BLAST output

Bio.Blast.NCBIXML

```
>>> result_handle = open("qblast_blastn.out")
>>> from Bio.Blast import NCBIXML
#If you expect a single BLAST result
>>> blast_record = NCBIXML.read(result_handle)

#If you have lots of results
>>> blast_records = NCBIXML.parse(result_handle)
```