SBI

Biopython: Structures

from Bio.PDB import *

Documentation
http://biopython.org/wiki/The_Biopython_Structural_Bioinformatics_FAQ

Protein Structure File Formats

1. mmcif: file extension .cif - https://files.rcsb.org/view/5JQ3.cif
2. pdb: specially formatted text file - https://files.rcsb.org/view/5JQ3.pdb

Load a pdb file:

```
>>> parser = PDBParser()

>>> parser = PDBParser(PERMISSIVE=1)
#the permissive flag indicates that a number of common problems associated
with PDB files will be ignored.

>>> struct = parser.get_structure("5jq3", "5jq3.pdb")
# four warnings
>>> struct
<Structure id=5jq3>
>>> print type(struct)
<class 'Bio.PDB.Structure.Structure'>
```

The header of the structure is a dictionary:

```
>>> print struct.header.keys()
['structure_method', 'head', 'journal', 'journal_reference',
'compound', 'keywords', 'name', 'author', 'deposition_date',
'release_date', 'source', 'resolution', 'structure_reference']
```

Download required files from the PDB:

```
>>> pdbl = PDBList()
>>> pdbl.retrieve_pdb_file('1FAT')
Downloading PDB structure '1FAT'...
'/home/stefano/scratch/fa/pdb1fat.ent'
```

Bio.PDB classes:

- **Structure**: describes one or more models of the same structure (NMR)
  - Structure.get_models() : iterator over the models.

```
>>> it = struct.get_models()
```

```
>>> it
<generator object get_models at 0x7f318f0b3fa0>
>>> models = list(it)
>>> models
```

    - Structure also acts as a dict, given a model ID returns the corresponding model object. struct[0]

```
>>> struct[0]
<Model id=0>
```

- **Model**: describes one 3D conformation. It contains one or more chains.
    - Model.get_chain() : iterator over the chains

```
>>> chains = list(models[0].get_chains())
>>> chains
[<Chain id=A>, <Chain id=B>]
```

    - Also acts as a dict, mapping from chain IDs to Chain objects.

```
>>> model = models[0]
>>> model["B"]
<Chain id=B>
```

- **Chain**: describes a proper polypeptide structure.
    - Chain.get_residues() : iterator over the residues.

```
>>> residues = list(chains[0].get_residues())
>>> len(residues)
338
```

- **Residue**: holds the atoms that belong to an aminoacid.
    - Residue.get_atom() : returns an iterator over the atoms.

```
>>> atoms = list(residues[0].get_atom())
>>> atoms
[<Atom N>, <Atom CA>, <Atom C>, <Atom O>, <Atom CB>, <Atom OG>]
```

- **Atom**: holds the 3D coordinate of an atom, as a Vector.

```
>>> vector = atoms[0].get_vector()
<Vector -68.82, 17.89, -5.51>
>>> coords = list(vector.get_array())
[-68.81500244, 17.88899994, -5.51300001]

>>> a.get_name() # atom name (spaces stripped, e.g. "CA")
```

```
>>> a.get_id() # id (equals atom name)
>>> a.get_coord() # atomic coordinates
>>> a.get_fullname() # atom name (with spaces, e.g. ".CA.")
```

Other methods of the Vector allow to compute the angle() with another vector, compute its norm(), map it to the unit sphere with normalize() and multiply the coordinates by a matrix with left_multiply().

Example of iteration:

```
parser = PDBParser()
struct = parser.get_structure("target", "5jq3.pdb")
for model in struct:
    for chain in model:
        for res in chain:
            for atom in res:
                print atom
```

Other examples:

```
>>> struct[0]["B"][502]["CA"].get_vector()

for res in model.get_residues():
    print residue

>>> atoms = chain.get_atoms()
>>> for atom in atoms:
        ... print(atom)

model = chain.get_parent()
```

You can also get a list of all child Entities of a parent Entity object. Note that this list is sorted in a specific way (e.g. according to chain identifier for Chain objects in a Model object).

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

You can also get the parent from a child:

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

The full id is a tuple containing all id's starting from the top object (Structure) down to the current object. A full id for a Residue object e.g. is something like:

```
>>> full_id = residue.get_full_id()
>>> print(full_id) ("1abc", 0, "A", ("", 10, "A"))
```

To get the entity's id, use the get_id method:

```
>>> entity.get_id()
```

You can check if the entity has a child with a given id by using the has_id

method:
>>> entity.has_id(entity_id)

The length of an entity is equal to its number of children:
>>> nr_children = len(entity)

You can also use the Selection.unfold_entities function to get all residues from a structure:
>>> res_list = Selection.unfold_entities(structure, 'R')

To go up:
>>> residue_list = Selection.unfold_entities(atom_list, 'R')
>>> chain_list = Selection.unfold_entities(atom_list, 'C')

Properties of atoms and residues:
http://biopython.org/wiki/
The_Biopython_Structural_Bioinformatics_FAQ#analysis

Residue ID:
Is a tuple with three elements:

- The **hetero-flag**: this is "H_" plus the name of the hetero-residue (e.g. 'H_GLC' in the case of a glucose molecule), or 'W' in the case of a water molecule.
- The **sequence identifier** in the chain, e.g. 100
- The **insertion code**, e.g. 'A'. The insertion code is sometimes used to preserve a certain desirable residue numbering scheme. A Ser 80 insertion mutant (inserted e.g. between a Thr 80 and an Asn 81 residue) could e.g. have sequence identifiers and insertion codes as follows: Thr 80 A, Ser 80 B, Asn 81. In this way the residue numbering scheme stays in tune with that of the wild type structure. The id of the above glucose residue would thus be ('H_GLC', 100, 'A'). If the hetero-flag and insertion code are blank, the sequence identifier alone can be used:

```
# Full id
residue = chain[(' ', 100, ' ')]
# Shortcut id
residue = chain[100]
```

Writing a PDB file

Given a Structure object ->

```
struct = ... # a Structure

io = PDBIO()
io.set_structure(struct)
io.save("output.pdb")
```

If you want to write out a part of the structure, make use of the Select class (also in PDBIO). Select has four methods:

- accept_model(model)
- accept_chain(chain)
- accept_residue(residue)
- accept_atom(atom)

```
class GlySelect(Select):
    def accept_residue(self, residue):
        if residue.get_name()=='GLY':
            return 1
        else:
            return 0

io = PDBIO()
io.set_structure(s)
io.save('gly_only.pdb', GlySelect())
```

```
class ChainSelect(Select):
        def __init__(self, chain):
            self.chain = chain

        def accept_chain(self, chain):
            if chain.get_id() == self.chain:
                return 1
            else:
                return 0

    chains = ['A','B','C']
    p = PDBParser(PERMISSIVE=1)
    structure = p.get_structure(pdb_file, pdb_file)

for chain in chains:
        pdb_chain_file = 'pdb_file_chain_{}.pdb'.format(chain)
        io_w_no_h = PDBIO()
        io_w_no_h.set_structure(structure)
        io_w_no_h.save('{}'.format(pdb_chain_file), ChainSelect(chain))
```

*the Dice module contains a handy extract function that writes out all residues in a chain between a start and end residue.

STRUCTURE ALIGNMENT

The most basic superposition algorithms work by **rototranslating** the two

proteins so to maximize their alignment. In particular, the quality of the alignment is measured by the Root Mean Squared Error (RMSD).

The Bio.PDB module supports RMSD-based structural alignment via the Superimposer class, let's see how.

1. Extracting the list of atoms from the two chains:

```
>>> atoms_a = list(struct[0]["A"].get_atoms())
>>> atoms_b = list(struct[0]["B"].get_atoms())
```

2. Make sure the two lists have the same length:

```
>>> atoms_a = atoms_a[:len(atoms_b)]
```

3. Create a superimposer object, the second structure will be aligned on top of the first one (atoms_a are fixed).

```
>>> si = Superimposer()
>>> si.set_atoms(atoms_a, atoms_b)
```

MAGIC! the superimposition occurs automatically.

Accessing itemps of the superimposer object:

- RMSD:

```
>>> print si.rms
26.463458137
```

- Rotation matrix and translation vector:

```
>>> rotmatrix = si.rotran[0]
>>> rotmatrix
array([[-0.43465167,  0.59888164,  0.67262078],
       [ 0.63784155,  0.73196744, -0.23954504],
       [-0.63579563,  0.32490683, -0.70014246]])
>>> transvector = si.rotran[1]
>>> transvector
```

Finally, we can apply the optimal rototranslation to the movable atoms:

```
array([-92.66668863,  31.1358793 ,  17.84958153])
>>> si.apply(atoms_b)
```

Extracting polypeptides from a Structure object:

```
# Using C-N
>>> ppb=PPBuilder()
>>> for pp in ppb.build_peptides(structure):
        ... print(pp.get_sequence())
        ...
# Using CA-CA
```

```
>>> ppb=CaPPBuilder()
>>> for pp in ppb.build_peptides(structure):
        ... print(pp.get_sequence())
        ...
```

However, it is possible to use PolypeptideBuilder to build Polypeptide objects from Model and Chain objects as well.

11.6.4 Determining atom-atom contacts

Use NeighborSearch to perform neighbor lookup. The neighbor lookup is done using a KD tree module written in C (see Bio.KDTree), making it very fast. It also includes a fast method to find all point pairs within a certain distance of each other.

11.6.6 Mapping the residues of two related structures onto each other

First, create an alignment file in FASTA format, then use the StructureAlignment class. This class can also be used for alignments with more than two structures.

CALCULATING DISTANCES

```
def calc_residue_dist(residue_one, residue_two) :
    """Returns the C-alpha distance between two residues"""
    diff_vector  = residue_one["CA"].coord - residue_two["CA"].coord
    return numpy.sqrt(numpy.sum(diff_vector * diff_vector))

def calc_dist_matrix(chain_one, chain_two) :
    """Returns a matrix of C-alpha distances between two chains"""
    answer = numpy.zeros((len(chain_one), len(chain_two)), numpy.float)
    for row, residue_one in enumerate(chain_one) :
        for col, residue_two in enumerate(chain_two) :
            answer[row, col] = calc_residue_dist(residue_one, residue_two)
    return answer
```

Seq Object

Alignments:

```
>>> from Bio import pairwise2
>>> alignments = pairwise2.align.globalxx("ACCGT", "ACG")
>>> from Bio.pairwise2 import format_alignment
>>> print(format_alignment(*alignments[0]))
ACCGT
|||||
A-CG-
  Score=3
<BLANKLINE>
```

Each alignment is a tuple consisting of the two aligned sequences, the score, the start and the end positions of the alignment (in global alignments the start is always 0 and the end the length of the alignment).

```
>>> len(alignments)
80
#80 different alignments with the same score

>>> print(alignments[0])
('MV-LSPADKTNV---K-A--A-WGKVGAHAG...YR-', 'MVHL-----T--
PEEKSAVTALWGKV----...Y-H', 72.0, 0, 217)
```