Lecture 21: 3D analysis II

Biopython interface for opening pdb files

Biopython includes a module for reading, writing, and analyzing PDB files.

http://biopython.org/wiki/The_Biopython_Structural_Bioinformatics_FAQ

```
conda install -c bioconda biopython
```

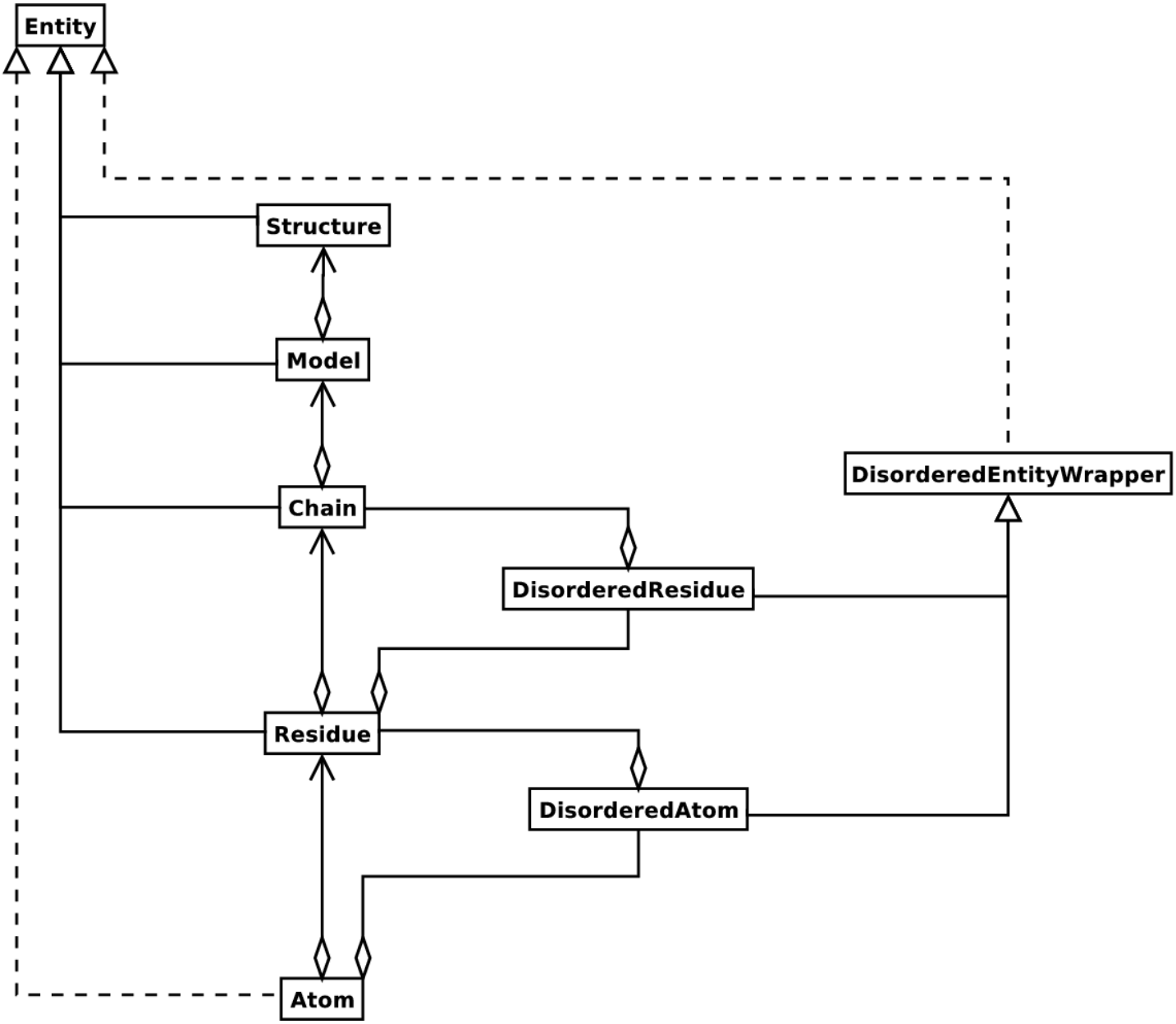
```
>>> import Bio.PDB
```

Pdb files are opened and parsed using the PDBParser object to create a Structure object:

```
>>> pdbp = Bio.PDB.PDBParser(QUIET = True)
>>> type(pdbp)
<class 'Bio.PDB.PDBParser.PDBParser'>
>>> structure = pdbp.get_structure('GluRec', '3RIF.pdb')
>>> type(structure)
<class 'Bio.PDB.Structure.Structure'>
>>> dir(structure)
['__class__',  '__contains__',  '__delattr__',  '__delitem__',  '__dict__',  '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__module__', '__ne__',
'__new__',  '__reduce__',  '__reduce_ex__',  '__repr__',  '__setattr__',  '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_sort', 'add', 'child_dict', 'child_list',
'copy',  'detach_child',  'detach_parent',  'full_id',  'get_atoms',  'get_chains',
'get_full_id',  'get_id',  'get_iterator',  'get_level',  'get_list',  'get_models',
'get_parent',  'get_residues',  'has_id',  'header',  'id',  'insert',  'level',  'parent',
'set_parent', 'transform', 'xtra']
```

To understand how the Structure object is organized, you have to understand SMCRA:

Structure – 1 structure per PDB
Model – Typically one model per PDB for protein crystal structures (if there were more than one arrangement of the proteins you wouldn't get a crystal. However, for structures generated through other approaches, like NMR, there can be multiple models per Structure.
Chain – Often multiple chains per Model
Residue – This is the level of the amino acid
Atom – This is the level of the atom.



The above diagram should help you understand the parent/child relationship between the different objects

```
Structure -> Parent of Models
Models -> Parent of Chains -> Child of Structure
Chain -> Parent of Residue -> Child of Model
Atom -> Child of Residue

>>> structure.child_list
[<Model id=0>]
>>> structure.child_list[0]
<Model id=0>

>>> model.child_list
[<Chain id=A>, <Chain id=B>, <Chain id=C>, <Chain id=D>, <Chain id=E>, <Chain id=F>,
<Chain id=G>, <Chain id=H>, <Chain id=I>, <Chain id=J>, <Chain id=K>, <Chain id=L>,
<Chain id=M>, <Chain id=N>, <Chain id=O>]
>>> model.parent
<Structure id=GluRec>
```

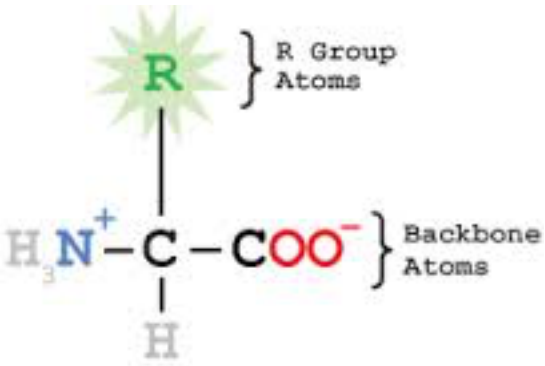
The above shows one approach to accessing the various objects. You can reach down all the way to atoms if you like:

```
>>> structure.child_list[0].child_list[0].child_list[0].child_list[0]
<Atom N>
```

Alternatively, if you know the name of the id, you can use slicing access the child objects:

```
>>> structure[0]
<Model id=0>
>>> structure[0]['A']
<Chain id=A>
```

Residues



Residues the subunits of a protein, typically holding one of twenty amino acides. It is worth exploring the residue objects in more details. Let's grab three different residues using the child_list approach:

```
>>> res1 = structure.child_list[0].child_list[0].child_list[0]
>>> res2 = structure.child_list[0].child_list[0].child_list[1]
>>> res3 = structure.child_list[0].child_list[0].child_list[2]

>>> res1
<Residue SER het=  resseq=1 icode= >
>>> res2
<Residue ASP het=  resseq=2 icode= >
>>> res3
<Residue SER het=  resseq=3 icode= >
```

If you want to know which type of amino acid each residue is, you can use the resname attribute:

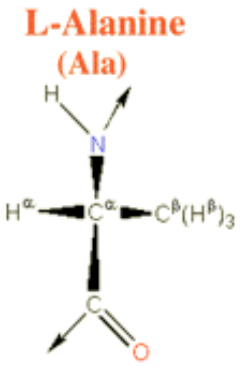
```
>>> res1.resname
'SER'
>>> res2.resname
'ASP'
>>> res3.resname
‘SER’
```

The residue number is a bit more tricky to get to:

```
>>> res1.id[1]
1
>>> res2.id[1]
2
>>> res3.id[1]
3
```

A few more things to point out. 1) Each residue has a different number of atoms. 2) Hydrogen atoms are not included in the pdb file (why?), and 3) Atoms name follow a standard name scheme.

```
>>> res1.child_list
[<Atom N>, <Atom CA>, <Atom C>, <Atom O>, <Atom CB>, <Atom OG>]
>>> res2.child_list
[<Atom N>, <Atom CA>, <Atom C>, <Atom O>, <Atom CB>, <Atom CG>, <Atom OD1>, <Atom OD2>]
```



Atoms

```
>>> atom1 = res1.child_list[0]
>>> atom2 = res1.child_list[1]
>>> atom1
<Atom N>
>>> atom2
<Atom CA>
```

Atoms have a few specific properties, namely element and coordinates. They are the only thing that have a physical position. To access these two attributes:

```
>>> atom1.element
'N'
>>> atom2.element
'C'
>>> atom1.coord
array([-16.503,  40.889,  71.556], dtype=float32)
>>> atom2.coord
array([-17.14 ,  40.762,  70.248], dtype=float32)
```

Atoms also have an overloaded subtraction method that allows you to easily determine how far apart two atoms are from each other (in Angstroms):

```
>>> atom2 - atom1
1.4603963
```

Finally, it can be useful to determine if two atoms belong to the same residue. You can do this using the parent attribute:

```
>>> atom3 = res2.child_list[0]
>>> atom1.parent
<Residue SER het=  resseq=1 icode= >
>>> atom2.parent
<Residue SER het=  resseq=1 icode= >
>>> atom3.parent
<Residue ASP het=  resseq=2 icode= >
>>> atom1.parent == atom2.parent
```

```
True
>>> atom1.parent == atom3.parent
False
```

Other ways to access atoms/residues

All objects can iterate through children objects (or grandchildren) using the following methods:

get_models()
get_chains()
get_residues()
get_atoms()

For example to get all of the residues of a structure:

```
>>> for res in structure.get_residues():
...     print(res)
...     break
...
<Residue SER het=  resseq=1 icode= >
```

To get all the residues of a chain:

```
>>> chain
<Chain id=A>
>>> for res in chain.get_residues():
...     print(res)
...     break
...
<Residue SER het=  resseq=1 icode= >
```