



ProDy

Protein Dynamics & Sequence Analysis

ProDy Tutorial

Release 1.5.1

Ahmet Bakan

December 24, 2013

CONTENTS

1	Introduction	1
1.1	Structural Ensemble Analysis	1
1.2	Elastic Network Models	1
1.3	Trajectory Analysis	1
1.4	Visualization	2
2	How to Start	3
2.1	Using ProDy	3
2.2	Interactive Usage	3
2.3	Using Documentation	4
3	ProDy Basics	6
3.1	File Parsers	6
3.2	Analysis Functions	7
3.3	Plotting Functions	7
3.4	Protein Structures	8
3.5	Atom Groups	9
3.6	ProDy Verbosity	10
4	Atom Groups	11
4.1	Building an Atom Group	11
4.2	Storing data in AtomGroup	15
5	Atom Selections	18
5.1	Atom Selections	18
5.2	Operations on Selections	20
6	Hierarchical Views	24
6.1	Hierarchical Views	24
6.2	Chains	25
6.3	Residues	27
6.4	Atoms	29
6.5	State Changes	29
7	Structure Analysis	30
7.1	Measure geometric properties	30
7.2	Compare and align structures	30
7.3	Writing PDB files	31
8	Dynamics Analysis	33

8.1	PCA Calculations	33
8.2	ANM Calculations	34
8.3	Comparative Analysis	35
8.4	Output Data Files	36
8.5	External Data	36
8.6	Plotting Data	36
8.7	More Examples	37
9	Sequence Analysis	38
9.1	Access Pfam	38
9.2	Parse MSA	38
9.3	Sequences	38
9.4	Analysis	39
10	Applications Tutorial	41
10.1	Align PDB files	41
10.2	ANM calculations	41
10.3	PCA calculations	42

INTRODUCTION

ProDy is an application programming interface (API) designed for structure-based analysis of protein dynamics, in particular for inferring protein dynamics from large heterogeneous structural ensembles. It comes with several command line applications (*ProDy Applications*¹) and graphical user interface for visualization (*Normal Mode Wizard*²). This tutorial shows core features of *ProDy* and some basic analysis tasks. You can find links to more detailed and advanced tutorials below.

1.1 Structural Ensemble Analysis

ProDy is primarily designed for analysis of *large* heterogeneous structural datasets for a protein composed of sequence homologs, mutants, or ligand bound structures that have with missing loops or terminal residues. Dominant patterns in structural variability are extracted by principal component analysis (PCA) of the ensemble. Helper functions allow for comparison of dynamics inferred from experiments with theoretical models and simulation data. For detailed usage examples see *Ensemble Analysis*³.

1.2 Elastic Network Models

ProDy can be used for normal mode analysis (NMA) of protein dynamics based on elastic network models (ENMs). Flexible classes allow for developing and using customized gamma functions in ENMs and numerous helper functions allow for comparative analysis of experimental and theoretical datasets. See *Elastic Network Models*⁴ for detailed usage examples.

1.3 Trajectory Analysis

In addition to analysis of experimental data and theoretical models, *ProDy* can be used to analyze trajectories from molecular dynamics simulations, such as for performing essential dynamics analysis (EDA). *ProDy* supports DCD file format, but trajectories in other formats can be parsed using other Python packages and analyzed using *ProDy*. See *Trajectory Analysis*⁵ for detailed usage examples.

¹<http://prody.csb.pitt.edu/manual/apps/prody/index.html#prody-apps>

²http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

³http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#ensemble-analysis

⁴http://prody.csb.pitt.edu/tutorials/enm_analysis/index.html#enm-analysis

⁵http://prody.csb.pitt.edu/tutorials/trajectory_analysis/index.html#trajectory-analysis

1.4 Visualization

Finally, results from *ProDy* calculations can be visualized using NMWiz, which is a [VMD](#)⁶ plugin GUI. NMWiz can also be used for submitting *ProDy* calculations for molecules in VMD. See [NMWiz Tutorial](#)⁷ for analysis of various types of datasets and visualization of protein dynamics.

⁶<http://www.ks.uiuc.edu/Research/vmd>

⁷http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/index.html#nmwiz-tutorial

HOW TO START

2.1 Using ProDy

ProDy can be used in a number of ways:

1. interactively in a Python shell,
2. as a command line program via *ProDy Applications*¹,
3. from within VMD via *Normal Mode Wizard*²,
4. or as a toolkit for developing new software.

2.1.1 Python for beginners

Familiarity with Python programming language will help when using *ProDy*. If you are new to Python, or to programming, you may start with one of the following tutorials:

- [The Python Tutorial](#)³
- [Python Scientific Lecture Notes](#)⁴
- [A Primer on Python for Life Science Researchers](#)⁵

2.2 Interactive Usage

In the rest of this tutorial, we assume that you will be typing commands in a Python shell. *ProDy* will automatically download PDB files and save them to current working directory, so you may want start Python from inside of a directory that you make for this tutorial:

```
$ mkdir prody_tutorial  
$ cd prody_tutorial
```

¹<http://prody.csb.pitt.edu/manual/apps/prody/index.html#prody-apps>

²http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

³<http://docs.python.org/tutorial/>

⁴<http://scipy-lectures.github.com/>

⁵<http://www.ploscompbiol.org/article/info%3Adoi%2F10.1371%2Fjournal.pcbi.0030199>

2.2.1 Start Python shell

For best interactive usage experience, we strongly recommend that you use [IPython](http://ipython.org)⁶ instead of the standard Python shell. IPython shell provides many user-friendly features, such as dynamic introspection and help, and also convenient integration of [Numpy](http://www.numpy.org)⁷ and [Matplotlib](http://matplotlib.org)⁸.

If you have installed IPython, type in:

```
$ ipython
```

If you also installed Matplotlib, use:

```
$ ipython --pylab
```

--pylab option will import Matplotlib and Numpy automatically, and is equivalent to the following:

```
In [1]: from pylab import *
```

```
In [2]: ion() # turn interactive mode on
```

If you don't have IPython yet, use:

```
$ python
```

On Windows, after you make the directory, make a `Shift+right click` in it in Windows Explorer and then select *Open command window here* option. Then start `C:\Python27\python.exe`. Alternatively, you may run **IDLE (Python GUI)** or **Python (command line)** from the start menu.

2.2.2 Import from ProDy

We import all *ProDy* functions and classes into the current namespace as follows:

```
In [3]: from prody import *
```

There are other ways to import *ProDy* contents. You may use `import prody as pd` and prefix all functions calls with `pd.`, if you prefer not to overcrowd the target namespace. Alternatively, if you want to use contents of a specific module, such as `prody.proteins`, you can use `from prody.proteins import *`. You should, however, avoid using `from prody.proteins.pdbfile import *`, because location of methods in submodules may change without notice.

2.3 Using Documentation

ProDy documentation is quite comprehensive and you can access it in a number of different ways. In interactive sessions, API reference can be accessed using the built-in Python function `help()`⁹:

```
help(select) # help on select module
help(fetchPDB) # help on parsePDB function
```

This function prints documentation on screen, and you will need to type `q` to exit from help view. If you are using the interactive Python shell (IPython), you can also get help using `?`:

⁶<http://ipython.org>

⁷<http://www.numpy.org>

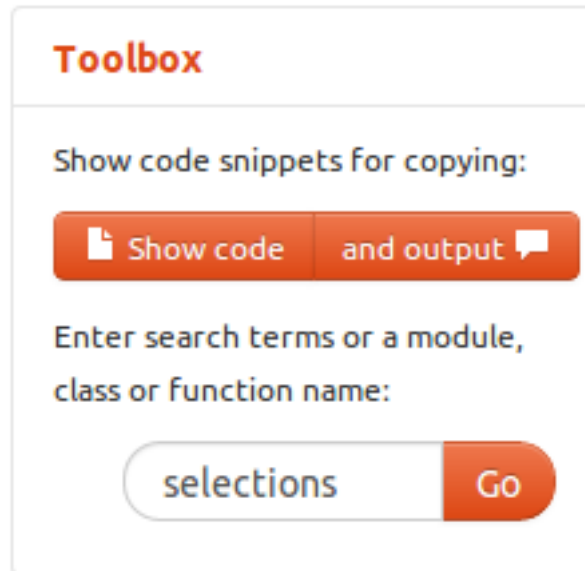
⁸<http://matplotlib.org>

⁹<http://docs.python.org/library/functions.html#help>

```
In [4]: ? fetchPDB
Type:      function
String Form:<function fetchPDB at 0x4ed0668>
File:      /home/abakan/Code/ProDy/prody/proteins/localpdb.py
Definition: fetchPDB(*pdb, **kwargs)
Docstring:
Return path(s) to PDB file(s) for specified *pdb* identifier(s).  Files
will be sought in user specified *folder* or current working director, and
then in local PDB folder and mirror, if they are available.  If *copy*
is set **True**, files will be copied into *folder*.  If *compressed* is
**False**, all files will be decompressed.  See :func:`pathPDBFolder` and
:func:`pathPDBMirror` for managing local resources, :func:`.fetchPDBviaFTP`
and :func:`.fetchPDBviaFTP` for downloading files from PDB servers.
```

2.3.1 Searching documentation

You can search entire documentation, including manual and tutorial pages, by typing in a keyword, function, or class name. Try searching for *selections* to get to *Atom Selections*¹⁰, for example.



2.3.2 Copying code snippets

When reading online documentation, you can use *Show code* button on the right hand side panel to display only code snippets. From this view, you can copy code directly into a file, i.e. click *Select* and then **Ctrl+C** to have the text in your clipboard. To return to the documentation click the *Close* button.



¹⁰<http://prody.csb.pitt.edu/manual/reference/atomic/select.html#selections>

PRODY BASICS

We start with importing everything from ProDy package:

```
In [1]: from prody import *
```

```
In [2]: from pylab import *
```

```
In [3]: ion()
```

Functions and classes are named such that they should not create a conflict with any other package. In this part we will familiarize with different categories of functions and methods.

3.1 File Parsers

Let's start with parsing a protein structure and then keep working on that in this part. File parser function names are prefixed with `parse`. You can get a list of parser functions by pressing `TAB` after typing in `parse`:

```
In [4]: parse<TAB>
parseArray      parseHeatmap    parseNMD         parsePDBStream
parseSTRIDE     parseDCD        parseMSA         parsePDB
parsePQR        parseSparseMatrix parseDSSP        parseModes
parsePDBHeader  parsePSF
```

When using `parsePDB()`, usually an identifier will be sufficient, If corresponding file is found in the current working directory, it will be used, otherwise it will be downloaded from PDB servers.

Let's parse structure `1p38`¹ of p38 MAP kinase (MAPK):

```
In [5]: p38 = parsePDB('1p38') # returns an AtomGroup object
```

```
In [6]: p38 # typing in variable name will give some information
```

```
Out[6]: <AtomGroup: 1p38 (2962 atoms)>
```

We see that this structure contains 2962 atoms.

Now, similar to listing parser function names, we can use tab completion to introspect `p38` object:

```
In [7]: p38.num<TAB>
p38.numAtoms    p38.numChains    p38.numFragments p38.numSegments
p38.numBonds     p38.numCoordsets p38.numResidues
```

¹<http://www.pdb.org/pdb/explore/explore.do?structureId=1p38>

This action printed a list of methods with *num* prefix. Let's use some of them to get information on the structure:

```
In [8]: p38.numAtoms()
Out[8]: 2962
```

```
In [9]: p38.numCoordsets() # returns number of models
Out[9]: 1
```

```
In [10]: p38.numResidues() # water molecules also count as residues
Out[10]: 480
```

3.2 Analysis Functions

Similar to parsers, analysis function names start with `calc`:

```
In [11]: calc<TAB>
calcADPAxes          calcCrossProjection  calcMSF              calcRMSF
calcADPs             calcCumulOverlap    calcOccupancies     calcRankorder
calcANM              calcDeformVector    calcOmega            calcShannonEntropy
calcAngle            calcDihedral         calcOverlap          calcSqFlucts
calcCenter           calcDistance         calcPerturbResponse  calcSubspaceOverlap
calcCollectivity     calcFractVariance   calcPhi              calcTempFactors
calcCovOverlap       calcGNM              calcProjection       calcTransformation
calcCovariance       calcGyradius         calcPsi              calcRMSD
calcCrossCorr        calcMSAOccupancy
```

Let's read documentation of `calcGyradius()` function and use it to calculate the radius of gyration of p38 MAPK structure:

```
In [12]: ? calcGyradius
Type:      function
String Form:<function calcGyradius at 0x4d422a8>
File:      /home/abakan/Code/ProDy/prody/measure/measure.py
Definition: calcGyradius(atoms, weights=None)
Docstring: Calculate radius of gyration of *atoms*.
```

```
In [13]: calcGyradius(p38)
Out[13]: 22.057752024921747
```

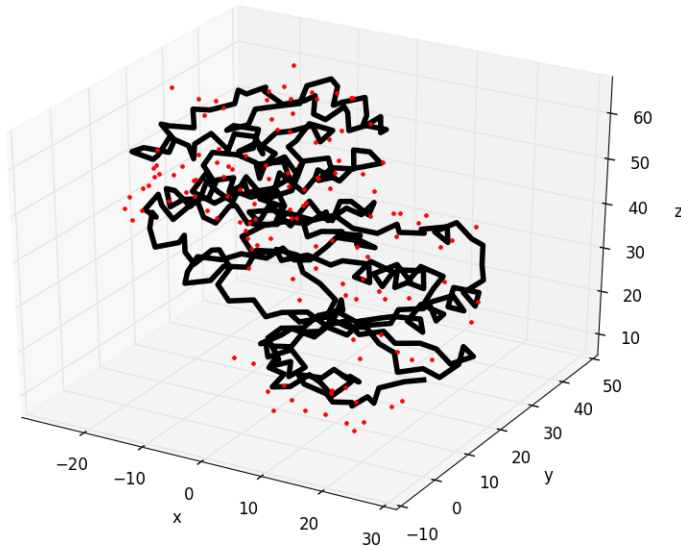
3.3 Plotting Functions

Likewise, plotting function names have `plot` prefix and here is a list of them:

```
In [14]: show<TAB>
showContactMap      showEllipsoid        showNormedSqFlucts  showScaledSqFlucts
showCrossCorr       showFractVars        showOccupancies     showShannonEntropy
showCrossProjection showHeatmap          showOverlap         showSqFlucts
showCumulFractVars  showMSAOccupancy     showOverlapTable    showProjection
showCumulOverlap    showMode             showProtein
showDiffMatrix      showMutinfoMatrix
```

We can use `showProtein()` function to make a quick plot of p38 structure:

```
In [15]: showProtein(p38);
```



This of course does not compare to any visualization software that you might be familiar with, but it comes handy to see what you are dealing with.

3.4 Protein Structures

Protein structures (.pdb files) will be the standard input for most *ProDy* calculations, so it is good to familiarize with ways to access and manage PDB file resources.

3.4.1 Fetching PDB files

First of all, *ProDy* downloads compressed PDB files when needed. If you prefer saving decompressed files, you can use `fetchPDB()` function as follows:

```
In [16]: fetchPDB('1p38', compressed=False)
Out[16]: '1p38.pdb'
```

Note that *ProDy* functions that fetch files or output files return filename upon successful completion of the task. You can use this behavior to shorten the code you need to write, e.g.:

```
In [17]: parsePDB(fetchPDB('1p38', compressed=False)) # same as p38 parsed above
Out[17]: <AtomGroup: 1p38 (2962 atoms)>
```

We downloaded and save an uncompressed PDB file, and parsed it immediately.

3.4.2 PDB file resources

Secondly, *ProDy* can manage local mirror of PDB server or a local PDB folders, as well as using a server close to your physical location for downloads:

- One of the [wwPDB²](http://www.wwpdb.org/) FTP servers in US, Europe or Japan can be picked for downloads using `wwPDBServer()`.
- A local PDB mirror can be set for faster access to files using `pathPDBMirror()`.
- A local folder can be set for storing downloaded files for future access using `pathPDBFolder()`.

If you are in the Americas now, you can choose the PDB server in the US as follows:

```
In [18]: wwPDBServer('us')
```

If you would like to have a central folder, such as `~Downloads/pdb`, for storing downloaded PDB files (you will need to make it), do as follows:

```
In [19]: mkdir /home/abakan/Downloads/pdb;
```

```
In [20]: pathPDBFolder('/home/abakan/Downloads/pdb')
```

Note that when these functions are used, ProDy will save your settings in `.prodyrc` file stored in your home folder.

3.5 Atom Groups

As you might have noticed, `parsePDB()` function returns structure data as an `AtomGroup` object. Let's see for `p38` variable from above:

```
In [21]: p38
Out[21]: <AtomGroup: 1p38 (2962 atoms)>
```

Data from this object can be retrieved using `get` methods. For example:

```
In [22]: p38.getResnames()
Out[22]:
array(['GLU', 'GLU', 'GLU', ..., 'HOH', 'HOH', 'HOH'],
      dtype='|S6')
```

```
In [23]: p38.getCoords()
Out[23]:
array([[ 28.492,   3.212,  23.465],
       [ 27.552,   4.354,  23.629],
       [ 26.545,   4.432,  22.489],
       ...,
       [ 18.872,   8.33 ,  36.716],
       [-22.062,  21.632,  42.029],
       [  1.323,  30.027,  65.103]])
```

To get a list of all methods use tab completion, i.e. `p38.<TAB>`. We will learn more about atom groups in the following chapters.

3.5.1 Indexing

An individual `Atom` can be accessed by indexing `AtomGroup` objects:

²<http://www.wwpdb.org/>

```
In [24]: atom = p38[0]
```

```
In [25]: atom
```

```
Out[25]: <Atom: N from 1p38 (index 0)>
```

Note that all `get/set` functions defined for `AtomGroup` instances are also defined for `Atom` instances, using singular form of the function name.

```
In [26]: atom.getResname()
```

```
Out[26]: 'GLU'
```

3.5.2 Slicing

It is also possible to get a slice of an `AtomGroup`. For example, we can get every other atom as follows:

```
In [27]: p38[::2]
```

```
Out[27]: <Selection: 'index 0:2962:2' from 1p38 (1481 atoms)>
```

Or, we can get the first 10 atoms, as follows:

```
In [28]: p38[:10]
```

```
Out[28]: <Selection: 'index 0:10:1' from 1p38 (10 atoms)>
```

3.5.3 Hierarchical view

You can also access specific chains or residues in an atom group. Indexing by a single letter identifier will return a `Chain` instance:

```
In [29]: p38['A']
```

```
Out[29]: <Chain: A from 1p38 (480 residues, 2962 atoms)>
```

Indexing atom group with a chain identifier and a residue number will return `Residue` instance:

```
In [30]: p38['A', 100]
```

```
Out[30]: <Residue: ASN 100 from Chain A from 1p38 (8 atoms)>
```

See *Atomic classes*³ for details of indexing atom groups and *Hierarchical Views* (page 24) for more on hierarchical views.

3.6 ProDy Verbosity

Finally, you might have noticed that ProDy prints some information to the console after parsing a file or doing some calculations. For example, PDB parser will print what was parsed and how long it took to the screen:

```
@> 1p38 (./1p38.pdb.gz) is found in the target directory.
@> 2962 atoms and 1 coordinate sets were parsed in 0.08s.
```

This behavior is useful in interactive sessions, but may be problematic for automated tasks as the messages are printed to `stderr`. The level of verbosity can be controlled using `confProDy()` function, and calling it as `confProDy(verbosity='none')` will stop all information messages permanently.

³<http://prody.csb.pitt.edu/manual/reference/atomic/index.html#atomic>

ATOM GROUPS

Below example shows how to build an `AtomGroup` from scratch. We start by importing everything from the ProDy package and the NumPy package:

```
In [1]: from prody import *
```

```
In [2]: from pylab import *
```

```
In [3]: ion()
```

4.1 Building an Atom Group

The best way to start constructing an `AtomGroup` is by setting the coordinates first. Number of atoms will be automatically set according to the size of the coordinate data array:

```
In [4]: wtr1 = AtomGroup('Water')
```

```
In [5]: wtr1
```

```
Out[5]: <AtomGroup: Water (0 atoms; no coordinates)>
```

```
In [6]: coords = array([[1, 0, 0], [0, 0, 0], [0, 0, 1]], dtype=float)
```

```
In [7]: coords
```

```
Out[7]:
```

```
array([[ 1.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  1.]])
```

```
In [8]: wtr1.setCoords(coords)
```

```
In [9]: wtr1
```

```
Out[9]: <AtomGroup: Water (3 atoms)>
```

4.1.1 Attributes

Attributes must be passed in a list or an array whose size is the same as the number of atoms.

```
In [10]: wtr1.setNames(['H', 'O', 'H'])
```

```
In [11]: wtr1.setResnums([1, 1, 1])
```

```
In [12]: wtr1.setResnames(['WAT', 'WAT', 'WAT'])
```

Accessing data will return a copy of the data:

```
In [13]: wtr1.getNames()
Out [13]:
array(['H', 'O', 'H'],
      dtype='<S6')

```

4.1.2 Atoms

Atoms are represented by instance of Atom.

Iteration

Atoms in an AtomGroup can be iterated over

```
In [14]: for a in wtr1: a
```

Indexing

Atoms in an atom group can be accessed via indexing:

```
In [15]: a = wtr1[0]
```

```
In [16]: a
Out [16]: <Atom: H from Water (index 0)>
```

```
In [17]: a.getCoords()
Out [17]: array([ 1.,  0.,  0.])
```

4.1.3 Coordinate sets

Let's add another coordinate set to the atom group:

```
In [18]: wtr1.addCoordset(array([[0, 1, 0], [0, 0, 0], [0, 0, 1]], dtype=float))
```

```
In [19]: wtr1
Out [19]: <AtomGroup: Water (3 atoms; active #0 of 2 coordsets)>
```

Note that number of coordinate sets is now 2, but active coordinate set index is still 0. Active coordinate set index can be changed for AtomGroup

```
In [20]: a.setACSIndex(1)
```

```
In [21]: a
Out [21]: <Atom: H from Water (index 0; active #1 of 2 coordsets)>
```

Changing active coordinate set for an atom group, does not affect the active coordinate set of the atom group:

```
In [22]: wtr1
Out [22]: <AtomGroup: Water (3 atoms; active #0 of 2 coordsets)>
```

Coordinates for the atom group will be returned from the active coordinate set

```
In [23]: a.getCoords()
Out[23]: array([ 0.,  1.,  0.]
```

Iterations

Coordinate sets can also be iterated over for `Atom` and `AtomGroup` instances:

```
In [24]: for xyz in a.iterCoordsets(): xyz
```

4.1.4 Copying and Merging

Now let's make another copy of this water:

```
In [25]: wtr2 = wtr1.copy()
```

```
In [26]: wtr2
Out[26]: <AtomGroup: Water (3 atoms; active #0 of 2 coordsets)>
```

Translate copy

Let's translate the coordinates of `wtr2` so that it does not overlap with `wtr1`

```
In [27]: wtr2.setCoords(wtr2.getCoords() + 2)
```

```
In [28]: wtr2.getCoords()
Out[28]:
array([[ 3.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  3.]])
```

Above operation only translated the coordinate set at index 0

```
In [29]: wtr2.setACSIndex(1)
```

```
In [30]: wtr2.getCoords()
Out[30]:
array([[ 0.,  1.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  1.]])
```

```
In [31]: wtr2.setCoords(wtr2.getCoords() + 2) # translate the 2nd coordset as well
```

Change attributes

Before we merge `wtr1` and `wtr2`, let's change resid's of `wtr2`:

```
In [32]: wtr2.setResnums( [2, 2, 2] )
```

```
In [33]: wtr2.getResnums()
Out[33]: array([2, 2, 2])
```

We can do this in an alternate way too:


```
In [34]: wtr2.select('all').setResnums(2)
```

```
In [35]: wtr2.getResnums()
Out [35]: array([2, 2, 2])
```

Note that the following won't work:

```
In [36]: wtr2.setResnums(2)
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-36-50a17a045d31> in <module>()
----> 1 wtr2.setResnums(2)
```

```
/home/abakan/Code/ProDy/prody/atomic/fields.pyc in setMethod(self, data)
    261 def wrapSetMethod(fn):
    262     def setMethod(self, data):
--> 263         return fn(self, data)
    264     return setMethod
```

```
/home/abakan/Code/ProDy/prody/atomic/atomgroup.pyc in setData(self, array, var, dtype, ndim, none, f
   1152         if self._n_atoms == 0:
   1153             self._n_atoms = len(array)
-> 1154         elif len(array) != self._n_atoms:
   1155             raise ValueError('length of array must match number '
   1156                               'of atoms')
```

```
TypeError: object of type 'int' has no len()
```

Merge two copies

Let's merge two water atom groups:

```
In [37]: wtrs = wtr1 + wtr2
```

```
In [38]: wtrs
Out [38]: <AtomGroup: Water + Water (6 atoms; active #0 of 2 coordsets)>
```

```
In [39]: wtrs.getCoords()
```

```
Out [39]:
array([[ 1.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  1.],
       [ 3.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  3.]])
```

```
In [40]: wtrs.getNames()
```

```
Out [40]:
array(['H', 'O', 'H', 'H', 'O', 'H'],
      dtype='<S6')

```

```
In [41]: wtrs.getResnums()
```

```
Out [41]: array([1, 1, 1, 2, 2, 2])
```

4.1.5 Hierarchical views

Hierarchical views of atom groups are represented by `HierView`.

Residues (and also chains) in an atom group can also be iterated over

```
In [42]: for res in wtrs.getHierView().iterResidues(): res
```

4.1.6 Renaming an atom group

Finally, it's possible to change the name of `wtrs` from "Water + Water" to something shorter:

```
In [43]: wtrs.setTitle('2Waters')
```

```
In [44]: wtrs
```

```
Out[44]: <AtomGroup: 2Waters (6 atoms; active #0 of 2 coordsets)>
```

4.2 Storing data in AtomGroup

Now let's get an atom group from a PDB file:

```
In [45]: structure = parsePDB('1p38')
```

In addition to what's in a PDB file, you can store arbitrary atomic attributes in `AtomGroup` objects.

4.2.1 Set a new attribute

For the purposes of this example, we will manufacture atomic data by dividing the residue number of each atom by 10:

```
In [46]: myresnum = structure.getResnums() / 10.0
```

We will add this to the atom group using `AtomGroup.setData()` method by passing a name for the attribute and the data:

```
In [47]: structure.setData('myresnum', myresnum)
```

We can check if a custom atomic attribute exists using `AtomGroup.isDataLabel()` method:

```
In [48]: structure.isDataLabel('myresnum')
```

```
Out[48]: True
```

4.2.2 Access subset of data

Custom attributes can be accessed from selections:

```
In [49]: calpha = structure.calpha
```

```
In [50]: calpha.getData('myresnum')
```

```
Out[50]:
array([[ 0.4,   0.5,   0.6,   0.7,   0.8,   0.9,   1. ,   1.1,   1.2,
         1.3,   1.4,   1.5,   1.6,   1.7,   1.8,   1.9,   2. ,   2.1,
         2.2,   2.3,   2.4,   2.5,   2.6,   2.7,   2.8,   2.9,   3. ,
```

```

3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9,
4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8,
4.9, 5. , 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7,
5.8, 5.9, 6. , 6.1, 6.2, 6.3, 6.4, 6.5, 6.6,
6.7, 6.8, 6.9, 7. , 7.1, 7.2, 7.3, 7.4, 7.5,
7.6, 7.7, 7.8, 7.9, 8. , 8.1, 8.2, 8.3, 8.4,
8.5, 8.6, 8.7, 8.8, 8.9, 9. , 9.1, 9.2, 9.3,
9.4, 9.5, 9.6, 9.7, 9.8, 9.9, 10. , 10.1, 10.2,
10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 11. , 11.1,
11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8, 11.9, 12. ,
12.1, 12.2, 12.3, 12.4, 12.5, 12.6, 12.7, 12.8, 12.9,
13. , 13.1, 13.2, 13.3, 13.4, 13.5, 13.6, 13.7, 13.8,
13.9, 14. , 14.1, 14.2, 14.3, 14.4, 14.5, 14.6, 14.7,
14.8, 14.9, 15. , 15.1, 15.2, 15.3, 15.4, 15.5, 15.6,
15.7, 15.8, 15.9, 16. , 16.1, 16.2, 16.3, 16.4, 16.5,
16.6, 16.7, 16.8, 16.9, 17. , 17.1, 17.2, 17.3, 17.4,
17.5, 17.6, 17.7, 17.8, 17.9, 18. , 18.1, 18.2, 18.3,
18.4, 18.5, 18.6, 18.7, 18.8, 18.9, 19. , 19.1, 19.2,
19.3, 19.4, 19.5, 19.6, 19.7, 19.8, 19.9, 20. , 20.1,
20.2, 20.3, 20.4, 20.5, 20.6, 20.7, 20.8, 20.9, 21. ,
21.1, 21.2, 21.3, 21.4, 21.5, 21.6, 21.7, 21.8, 21.9,
22. , 22.1, 22.2, 22.3, 22.4, 22.5, 22.6, 22.7, 22.8,
22.9, 23. , 23.1, 23.2, 23.3, 23.4, 23.5, 23.6, 23.7,
23.8, 23.9, 24. , 24.1, 24.2, 24.3, 24.4, 24.5, 24.6,
24.7, 24.8, 24.9, 25. , 25.1, 25.2, 25.3, 25.4, 25.5,
25.6, 25.7, 25.8, 25.9, 26. , 26.1, 26.2, 26.3, 26.4,
26.5, 26.6, 26.7, 26.8, 26.9, 27. , 27.1, 27.2, 27.3,
27.4, 27.5, 27.6, 27.7, 27.8, 27.9, 28. , 28.1, 28.2,
28.3, 28.4, 28.5, 28.6, 28.7, 28.8, 28.9, 29. , 29.1,
29.2, 29.3, 29.4, 29.5, 29.6, 29.7, 29.8, 29.9, 30. ,
30.1, 30.2, 30.3, 30.4, 30.5, 30.6, 30.7, 30.8, 30.9,
31. , 31.1, 31.2, 31.3, 31.4, 31.5, 31.6, 31.7, 31.8,
31.9, 32. , 32.1, 32.2, 32.3, 32.4, 32.5, 32.6, 32.7,
32.8, 32.9, 33. , 33.1, 33.2, 33.3, 33.4, 33.5, 33.6,
33.7, 33.8, 33.9, 34. , 34.1, 34.2, 34.3, 34.4, 34.5,
34.6, 34.7, 34.8, 34.9, 35. , 35.1, 35.2, 35.3, 35.4])

```

4.2.3 Make selections

Custom atomic attributes can be used in selections:

```
In [51]: mysel = structure.select('0 < myresnum and myresnum < 10')
```

```
In [52]: mysel
```

```
Out[52]: <Selection: '0 < myresnum and myresnum < 10' from lp38 (788 atoms)>
```

This gives the same result as the following selection:

```
In [53]: structure.select('0 < resnum and resnum < 100') == mysel
```

```
Out[53]: True
```

4.2.4 Save attributes

It is not possible to save custom attributes in PDB files, but `saveAtoms()` function can be used them to save in disk for later use:

```
In [54]: saveAtoms(structure)
Out[54]: '1p38.ag.npz'
```

Let's load it using `loadAtoms()` function:

```
In [55]: structure = loadAtoms('1p38.ag.npz')
```

```
In [56]: structure.getData('myresnum')
Out[56]: array([ 0.4,  0.4,  0.4, ...,  77.1,  77.3,  77.6])
```

4.2.5 Delete an attribute

Finally, when done with an attribute, it can be deleted using `AtomGroup.delData()` method:

```
In [57]: structure.delData('myresnum')
Out[57]: array([ 0.4,  0.4,  0.4, ...,  77.1,  77.3,  77.6])
```

ATOM SELECTIONS

This part gives more information on properties of `AtomGroup` objects. We start with making necessary imports. Note that every documentation page contains them so that the code within the can be executed independently. You can skip them if you have already done them in a Python session.

```
In [1]: from prody import *
```

```
In [2]: from pylab import *
```

```
In [3]: ion()
```

5.1 Atom Selections

`AtomGroup` instances have a plain view of atoms for efficiency, but they are coupled with a powerful atom selection engine. You can get well defined atom subsets by passing simple keywords or make rather sophisticated selections using composite statements. Selection keywords and grammar are very much similar to those found in [VMD](http://www.ks.uiuc.edu/Research/vmd)¹. Some examples are shown here:

5.1.1 Keyword selections

Now, we parse a structure. This could be any structure, one that you know well from your research, for example.

```
In [4]: structure = parsePDB('1p38')
```

```
In [5]: protein = structure.select('protein')
```

```
In [6]: protein
```

```
Out[6]: <Selection: 'protein' from 1p38 (2833 atoms)>
```

Using the "protein" keyword we selected 2833 atoms out of 2962 atoms. `Atomic.select()` method returned a `Selection` instance. Note that all get and set methods defined for the `AtomGroup` objects are also defined for `Selection` objects. For example:

```
In [7]: protein.getResnames()
```

```
Out[7]:  
array(['GLU', 'GLU', 'GLU', ..., 'ASP', 'ASP', 'ASP'],  
      dtype='<S6')
```

¹<http://www.ks.uiuc.edu/Research/vmd>

5.1.2 Select by name/type

We can select backbone atoms by passing atom names following "name" keyword:

```
In [8]: backbone = structure.select('protein and name N CA C O')
In [9]: backbone
Out[9]: <Selection: 'protein and name N CA C O' from 1p38 (1404 atoms)>
```

Alternatively, we can use "backbone" to make the same selection:

```
In [10]: backbone = structure.select('backbone')
```

We select acidic and basic residues by using residue names with "resname" keyword:

```
In [11]: charged = structure.select('resname ARG LYS HIS ASP GLU')
In [12]: charged
Out[12]: <Selection: 'resname ARG LYS HIS ASP GLU' from 1p38 (906 atoms)>
```

Alternatively, we can use predefined keywords "acidic" and "basic".

```
In [13]: charged = structure.select('acidic or basic')
In [14]: charged
Out[14]: <Selection: 'acidic or basic' from 1p38 (906 atoms)>
In [15]: set(charged.getResnames())
Out[15]: {'ARG', 'ASP', 'GLU', 'HIS', 'LYS'}
```

5.1.3 Composite selections

Let's try a more sophisticated selection. We first calculate the geometric center of the protein atoms using `calcCenter()` function. Then, we select the $C\alpha$ and $C\beta$ atoms of residues that have at least one atom within 10 Å away from the geometric center.

```
In [16]: center = calcCenter(protein).round(3)
In [17]: center
Out[17]: array([ 1.005, 17.533, 40.052])
In [18]: sel = structure.select('protein and name CA CB and same residue as '
.....:                          '((x-1)**2 + (y-17.5)**2 + (z-40.0)**2)**0.5 < 10')
.....:
In [19]: sel
Out[19]: <Selection: 'protein and nam...)**2)**0.5 < 10' from 1p38 (66 atoms)>
```

Alternatively, this selection could be done as follows:

```
In [20]: sel = structure.select('protein and name CA CB and same residue as '
.....:                          'within 10 of center', center=center)
.....:
In [21]: sel
Out[21]: <Selection: 'index 576 579 5... 1687 1707 1710' from 1p38 (66 atoms)>
```

5.1.4 Selections simplified

In interactive sessions, an alternative to typing in `.select('protein')` or `.select('backbone')` is using dot operator:

```
In [22]: protein = structure.protein
```

```
In [23]: protein
```

```
Out[23]: <Selection: 'protein' from 1p38 (2833 atoms)>
```

You can use dot operator multiple times:

```
In [24]: bb = structure.protein.backbone
```

```
In [25]: bb
```

```
Out[25]: <Selection: '(backbone) and (protein)' from 1p38 (1404 atoms)>
```

This may go on and on:

```
In [26]: ala_ca = structure.protein.backbone.resname_ALA.calpha
```

```
In [27]: ala_ca
```

```
Out[27]: <Selection: '(calpha) and ((...and (protein)))' from 1p38 (26 atoms)>
```

5.1.5 More examples

There is much more to what you can do with this flexible and fast atom selection engine, without the need for writing nested loops with comparisons or changing the source code. See the following pages:

- *Atom Selections*² for description of all selection keywords
- *Intermolecular Contacts*³ for selecting interacting atoms

5.2 Operations on Selections

Selection objects can be used with bitwise operators:

5.2.1 Union

Let's select β -carbon atoms for non-GLY amino acid residues, and α -carbons for GLYs in two steps:

```
In [28]: betas = structure.select('name CB and protein')
```

```
In [29]: len(betas)
```

```
Out[29]: 336
```

```
In [30]: gly_alphas = structure.select('name CA and resname GLY')
```

```
In [31]: len(gly_alphas)
```

```
Out[31]: 15
```

²<http://prody.csb.pitt.edu/manual/reference/atomic/select.html#selections>

³http://prody.csb.pitt.edu/tutorials/structure_analysis/contacts.html#contacts

The above shows that the p38 structure contains 15 GLY residues.

These two selections can be combined as follows:

```
In [32]: betas_gly_alphas = betas | gly_alphas

In [33]: betas_gly_alphas
Out[33]: <Selection: '(name CB and pr...nd resname GLY)' from 1p38 (351 atoms)>

In [34]: len(betas_gly_alphas)
Out[34]: 351
```

The selection string for the union of selections becomes:

```
In [35]: betas_gly_alphas.getSelstr()
Out[35]: '(name CB and protein) or (name CA and resname GLY)'
```

Note that it is also possible to yield the same selection using selection string (name CB and protein) or (name CA and resname GLY).

5.2.2 Intersection

It is as easy to get the intersection of two selections. Let's find charged and medium size residues in a protein:

```
In [36]: charged = structure.select('charged')

In [37]: charged
Out[37]: <Selection: 'charged' from 1p38 (906 atoms)>

In [38]: medium = structure.select('medium')

In [39]: medium
Out[39]: <Selection: 'medium' from 1p38 (751 atoms)>

In [40]: medium_charged = medium & charged

In [41]: medium_charged
Out[41]: <Selection: '(medium) and (charged)' from 1p38 (216 atoms)>

In [42]: medium_charged.getSelstr()
Out[42]: '(medium) and (charged)'
```

Let's see which amino acids are considered charged and medium:

```
In [43]: set(medium_charged.getResnames())
Out[43]: {'ASP'}
```

What about amino acids that are medium or charged:

```
In [44]: set((medium | charged).getResnames())
Out[44]: {'ARG', 'ASN', 'ASP', 'CYS', 'GLU', 'HIS', 'LYS', 'PRO', 'THR', 'VAL'}
```

5.2.3 Inversion

It is also possible to invert a selection:


```

In [45]: only_protein = structure.select('protein')

In [46]: only_protein
Out[46]: <Selection: 'protein' from 1p38 (2833 atoms)>

In [47]: only_non_protein = ~only_protein

In [48]: only_non_protein
Out[48]: <Selection: 'not (protein)' from 1p38 (129 atoms)>

In [49]: water = structure.select('water')

In [50]: water
Out[50]: <Selection: 'water' from 1p38 (129 atoms)>

```

The above shows that 1p38 does not contain any non-water hetero atoms.

5.2.4 Addition

Another operation defined on the `Select` object is addition (also on other `AtomPointer` derived classes). This may be useful if you want to yield atoms in an `AtomGroup` in a specific order. Let's think of a simple case, where we want to output atoms in 1p38 in a specific order:

```

In [51]: protein = structure.select('protein')

In [52]: water = structure.select('water')

In [53]: water_protein = water + protein

In [54]: writePDB('1p38_water_protein.pdb', water_protein)
Out[54]: '1p38_water_protein.pdb'

```

In the resulting file, the water atoms will precedes the protein atoms.

5.2.5 Membership

Selections also allows membership test operations:

```

In [55]: backbone = structure.select('protein')

In [56]: calpha = structure.select('calpha')

```

Is *calpha*⁴ a subset of *backbone*⁵?

```

In [57]: calpha in backbone
Out[57]: True

```

Or, is water in protein selection?

```

In [58]: water in protein
Out[58]: False

```

Other tests include:

⁴<http://prody.csb.pitt.edu/manual/reference/atomic/flags.html#term-calpha>

⁵<http://prody.csb.pitt.edu/manual/reference/atomic/flags.html#term-backbone>

```
In [59]: protein in structure
Out[59]: True
```

```
In [60]: backbone in structure
Out[60]: True
```

```
In [61]: structure in structure
Out[61]: True
```

```
In [62]: calpha in calpha
Out[62]: True
```

5.2.6 Equality

You can also check the equality of selections. Comparison will return `True` if both selections refer to the same atoms.

```
In [63]: calpha = structure.select('protein and name CA')
```

```
In [64]: calpha2 = structure.select('calpha')
```

```
In [65]: calpha == calpha2
Out[65]: True
```

HIERARCHICAL VIEWS

This part describes how to use hierarchical views. We start by importing everything from the ProDy package:

```
In [1]: from prody import *
```

6.1 Hierarchical Views

Then we parse a structure to get an `AtomGroup` instance which has a plain view of atoms:

```
In [2]: structure = parsePDB('3mkb')
```

```
In [3]: structure  
Out[3]: <AtomGroup: 3mkb (4776 atoms)>
```

A hierarchical view of the structure can be simply get by calling the `AtomGroup.getHierView()` method:

```
In [4]: hv = structure.getHierView()
```

```
In [5]: hv  
Out[5]: <HierView: AtomGroup 3mkb (4 chains, 946 residues)>
```

6.1.1 Indexing

Indexing `HierView` instances return `Chain`:

```
In [6]: hv['A']  
Out[6]: <Chain: A from 3mkb (254 residues, 1198 atoms)>
```

```
In [7]: hv['B']  
Out[7]: <Chain: B from 3mkb (216 residues, 1193 atoms)>
```

```
In [8]: hv['Z'] # This will return None, which means chain Z does not exist
```

The length of the `hv` variable gives the number of chains in the structure:

```
In [9]: len(hv)  
Out[9]: 4
```

```
In [10]: hv.numChains()  
Out[10]: 4
```

It is also possible to get a Residue by directly indexing the HierView instance:

```
In [11]: hv['A', 100]
Out[11]: <Residue: MET 100 from Chain A from 3mkb (8 atoms)>
```

Insertion codes can also be passed:

```
In [12]: hv['A', 100, 'B']
```

But this does not return anything, since residue 100B does not exist.

6.1.2 Iterations

One can iterate over HierView instances to get chains:

```
In [13]: for chain in hv:
.....:     chain
.....:
```

It is also possible to get a `list()`¹ of chains simply as follows:

```
In [14]: chains = list(hv)

In [15]: chains
Out[15]:
[<Chain: A from 3mkb (254 residues, 1198 atoms)>,
 <Chain: B from 3mkb (216 residues, 1193 atoms)>,
 <Chain: C from 3mkb (245 residues, 1189 atoms)>,
 <Chain: D from 3mkb (231 residues, 1196 atoms)>]
```

6.1.3 Residues

In addition, one can also iterate over all residues:

```
In [16]: for i, residue in enumerate(hv.iterResidues()):
.....:     if i == 4: break
.....:     print(residue)
.....:
ALA 1
PHE 2
THR 3
GLY 4
```

6.2 Chains

```
In [17]: chA = hv['A']
```

```
In [18]: chA
Out[18]: <Chain: A from 3mkb (254 residues, 1198 atoms)>
```

Length of the chain equals to the number of residues in it:

¹<http://docs.python.org/library/functions.html#list>

```
In [19]: len(chA)
Out[19]: 254
```

```
In [20]: chA.numResidues()
Out[20]: 254
```

6.2.1 Indexing

Indexing a Chain instance returns a Residue instance.

```
In [21]: chA[1]
Out[21]: <Residue: ALA 1 from Chain A from 3mkb (5 atoms)>
```

If a residue does not exist, None is returned:

```
In [22]: chA[1000]
```

```
In [23]: chA[1, 'A'] # Residue 1 with insertion code A also does not exist
```

If residue with given integer number does not exist, None is returned.

6.2.2 Iterations

Iterating over a chain yields residues:

```
In [24]: for i, residue in enumerate(chA):
.....:     if i == 4: break
.....:     print(residue)
.....:
ALA 1
PHE 2
THR 3
GLY 4
```

Note that water atoms, each constituting a residue, are also part of a chain if they are labeled with that chain's identifier.

This enables getting a `list()`² of residues simply as follows:

```
In [25]: chA_residues = list(chA)
```

```
In [26]: chA_residues[:4]
Out[26]:
[<Residue: ALA 1 from Chain A from 3mkb (5 atoms)>,
 <Residue: PHE 2 from Chain A from 3mkb (11 atoms)>,
 <Residue: THR 3 from Chain A from 3mkb (7 atoms)>,
 <Residue: GLY 4 from Chain A from 3mkb (4 atoms)>]
```

```
In [27]: chA_residues[-4:]
Out[27]:
[<Residue: HOH 471 from Chain A from 3mkb (1 atoms)>,
 <Residue: HOH 485 from Chain A from 3mkb (1 atoms)>,
 <Residue: HOH 490 from Chain A from 3mkb (1 atoms)>,
 <Residue: HOH 493 from Chain A from 3mkb (1 atoms)>]
```

²<http://docs.python.org/library/functions.html#list>

6.2.3 Get data

All methods defined for `AtomGroup` class are also defined for `Chain` and `Residue` classes:

```
In [28]: chA.getCoords()
Out [28]: array([[ -2.139,  17.026, -13.287],
             [ -1.769,  15.572, -13.111],
             [ -0.296,  15.257, -13.467],
             ...,
             [ -5.843,  17.181, -16.86 ],
             [-13.199,  -9.21 , -49.692],
             [ -0.459,   0.378, -46.156]])

In [29]: chA.getBetas()
Out [29]: array([ 59.35,  59.14,  58.5 , ...,  57.79,  47.77,  40.77])
```

6.2.4 Selections

Finally, you can select atoms from a `Chain` instance:

```
In [30]: chA_backbone = chA.select('backbone')

In [31]: chA_backbone
Out [31]: <Selection: '(backbone) and (chain A)' from 3mkb (560 atoms)>

In [32]: chA_backbone.getSelstr()
Out [32]: '(backbone) and (chain A)'
```

As you see, the selection string passed by the user is augmented with “chain” keyword and identifier automatically to provide internal consistency:

```
In [33]: structure.select(chA_backbone.getSelstr())
Out [33]: <Selection: '(backbone) and (chain A)' from 3mkb (560 atoms)>
```

6.3 Residues

```
In [34]: chA_res1 = chA[1]

In [35]: chA_res1
Out [35]: <Residue: ALA 1 from Chain A from 3mkb (5 atoms)>
```

6.3.1 Indexing

Residue instances can be indexed to get individual atoms:

```
In [36]: chA_res1['CA']
Out [36]: <Atom: CA from 3mkb (index 1)>

In [37]: chA_res1['CB']
Out [37]: <Atom: CB from 3mkb (index 4)>

In [38]: chA_res1['X'] # if atom does not exist, None is returned
```

6.3.2 Iterations

Iterating over a residue instance yields `Atom` instances:

```
In [39]: for i, atom in enumerate(chA_res1):
.....:     if i == 4: break
.....:     print(atom)
.....:
Atom N (index 0)
Atom CA (index 1)
Atom C (index 2)
Atom O (index 3)
```

This makes it easy to get a `list()`³ of atoms:

```
In [40]: list(chA_res1)
Out[40]:
[<Atom: N from 3mkb (index 0)>,
 <Atom: CA from 3mkb (index 1)>,
 <Atom: C from 3mkb (index 2)>,
 <Atom: O from 3mkb (index 3)>,
 <Atom: CB from 3mkb (index 4)>]
```

6.3.3 Get data

All methods defined for `AtomGroup` class are also defined for `Residue` class:

```
In [41]: chA_res1.getCoords()
Out[41]:
array([[ -2.139,  17.026, -13.287],
       [ -1.769,  15.572, -13.111],
       [ -0.296,  15.257, -13.467],
       [  0.199,  14.155, -13.155],
       [ -2.752,  14.639, -13.898]])

In [42]: chA_res1.getBetas()
Out[42]: array([ 59.35,  59.14,  58.5 ,  59.13,  59.02])
```

6.3.4 Selections

Finally, you can select atoms from a `Residue` instance:

```
In [43]: chA_res1_bb = chA_res1.select('backbone')

In [44]: chA_res1_bb
Out[44]: <Selection: '(backbone) and ... and (chain A))' from 3mkb (4 atoms)>

In [45]: chA_res1_bb.getSelstr()
Out[45]: '(backbone) and (resnum 1 and (chain A))'
```

Again, the selection string is augmented with the chain identifier and residue number (*resnum*⁴).

³<http://docs.python.org/library/functions.html#list>

⁴<http://prody.csb.pitt.edu/manual/reference/atomic/fields.html#term-resnum>

6.4 Atoms

The lowest level of the hierarchical view contains `Atom` instances.

```
In [46]: chA_res1_CA = chA_res1['CA']
```

```
In [47]: chA_res1_CA
```

```
Out[47]: <Atom: CA from 3mkb (index 1)>
```

Get atomic data

All methods defined for `AtomGroup` class are also defined for `Atom` class with the difference that method names are singular (except for coordinates):

```
In [48]: chA_res1_CA.getCoords()
```

```
Out[48]: array([-1.769,  15.572, -13.111])
```

```
In [49]: chA_res1_CA.getBeta()
```

```
Out[49]: 59.140000000000001
```

6.5 State Changes

A `HierView` instance represents the state of an `AtomGroup` instance at the time it is built. When chain identifiers or residue numbers change, the state that hierarchical view represents may not match the current state of the atom group:

```
In [50]: chA.setChid('X')
```

```
In [51]: chA
```

```
Out[51]: <Chain: X from 3mkb (254 residues, 1198 atoms)>
```

```
In [52]: hv['X'] # returns None, since hierarchical view is not updated
```

```
In [53]: hv.update() # this updates hierarchical view
```

```
In [54]: hv['X']
```

```
Out[54]: <Chain: X from 3mkb (254 residues, 1198 atoms)>
```

When this is the case, `HierView.update()` method can be used to update hierarchical view.

STRUCTURE ANALYSIS

ProDy comes with many functions that can be used to calculate structural properties and compare structures. We demonstrate only some of these functions. For more detailed examples, see *Structure Analysis*¹ tutorial.

```
In [1]: from prody import *
```

```
In [2]: from pylab import *
```

```
In [3]: ion()
```

7.1 Measure geometric properties

Let's parse a structure:

```
In [4]: p38 = parsePDB('1p38')
```

Functions for analyzing structures can be found in `measure`² module. For example, you can calculate phi (ϕ) and psi (ψ) for the 10th residue, or the radius of gyration of the protein as follows:

```
In [5]: calcPhi(p38[10,])  
Out [5]: -115.5351427673999
```

```
In [6]: calcPsi(p38[10,])  
Out [6]: 147.49025666398765
```

```
In [7]: calcGyradius(p38)  
Out [7]: 22.057752024921747
```

7.2 Compare and align structures

You can also compare different structures using some of the methods in `proteins` module. Let's parse another p38 MAP kinase structure

```
In [8]: bound = parsePDB('1zz2')
```

You can find similar chains in structure 1p38 and 1zz2 using `matchChains()` function:

¹http://prody.csb.pitt.edu/tutorials/structure_analysis/index.html#structure-analysis

²<http://prody.csb.pitt.edu/manual/reference/measure/index.html#prody.measure>

```

In [9]: apo_chA, bnd_chA, seqid, overlap = matchChains(p38, bound)[0]

In [10]: apo_chA
Out[10]: <AtomMap: Chain A from 1p38 -> Chain A from 1zz2 from 1p38 (337 atoms)>

In [11]: bnd_chA
Out[11]: <AtomMap: Chain A from 1zz2 -> Chain A from 1p38 from 1zz2 (337 atoms)>

In [12]: seqid
Out[12]: 99.40652818991099

In [13]: overlap
Out[13]: 96

```

Matching $C\alpha$ atoms are selected and returned as `AtomMap` instances. We can use them to calculate RMSD and superpose structures.

```

In [14]: calcRMSD(bnd_chA, apo_chA)
Out[14]: 72.930230869465859

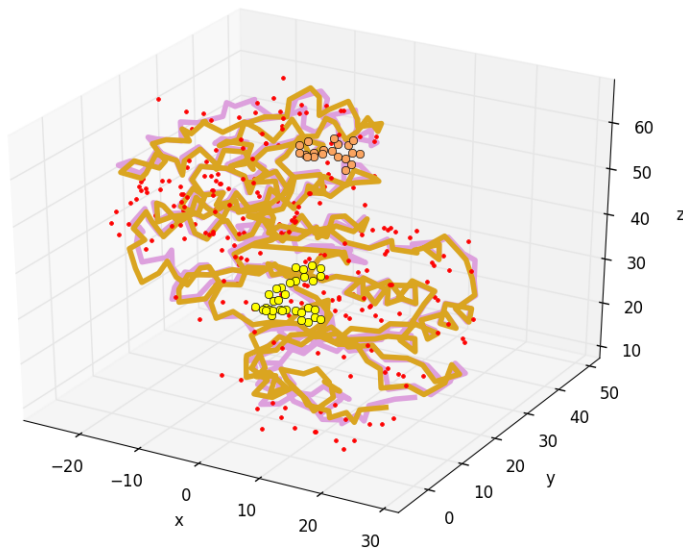
In [15]: bnd_chA, transformation = superpose(bnd_chA, apo_chA)

In [16]: calcRMSD(bnd_chA, apo_chA)
Out[16]: 1.86280149086955

In [17]: showProtein(p38);

In [18]: showProtein(bound);

```



7.3 Writing PDB files

PDB files can be written using the `writePDB()` function. The function accepts objects containing or referring to atomic data.

Output selected atoms:

```
In [19]: writePDB('1p38_calphas.pdb', p38.select('calpha'))  
Out[19]: '1p38_calphas.pdb'
```

Output a chain:

```
In [20]: chain_A = p38['A']  
  
In [21]: writePDB('1p38_chain_A.pdb', chain_A)  
Out[21]: '1p38_chain_A.pdb'
```

As you may have noticed, this function returns the file name after it is successfully written. This is a general behavior for ProDy output functions. For more PDB writing examples see *Write PDB file*³.

³http://prody.csb.pitt.edu/tutorials/structure_analysis/pdbfiles.html#writpdb

DYNAMICS ANALYSIS

In this section, we will show how to perform quick PCA and ANM analysis using a solution structure of Ubiquitin¹. If you started a new Python session, import ProDy contents:

```
In [1]: from prody import *
```

```
In [2]: from pylab import *
```

```
In [3]: ion()
```

8.1 PCA Calculations

Let's perform principal component analysis (PCA) of an ensemble of NMR models, such as 2k39². First, we prepare the ensemble:

```
In [4]: ubi = parsePDB('2k39', subset='calpha')
```

```
In [5]: ubi_selection = ubi.select('resnum < 71')
```

```
In [6]: ubi_ensemble = Ensemble(ubi_selection)
```

```
In [7]: ubi_ensemble.iterpose()
```

Then, we perform the PCA calculations:

```
In [8]: pca = PCA('Ubiquitin')
```

```
In [9]: pca.buildCovariance(ubi_ensemble)
```

```
In [10]: pca.calcModes()
```

This analysis provides us with a description of the dominant changes in the structural ensemble. Let's see the fraction of variance for top ranking 4 PCs:

```
In [11]: for mode in pca[:4]:
.....:     print(calcFractVariance(mode).round(2))
.....:
0.13
0.09
```

¹<http://en.wikipedia.org/wiki/Ubiquitin>

²<http://www.pdb.org/pdb/explore/explore.do?structureId=2k39>

0.08
0.07

PCA data can be saved on disk using `saveModel()` function:

```
In [12]: saveModel(pca)
Out [12]: 'Ubiquitin.pca.npz'
```

This function writes data in binary format, so is an efficient way of storing data permanently. In a later session, this data can be loaded using `loadModel()` function.

8.2 ANM Calculations

Anisotropic network model (ANM) analysis can be performed in two ways:

The shorter way, which may be suitable for interactive sessions:

```
In [13]: anm, atoms = calcANM(ubi_selection, selstr='calpha')
```

The longer and more controlled way:

```
In [14]: anm = ANM('ubi') # instantiate ANM object
```

```
In [15]: anm.buildHessian(ubi_selection) # build Hessian matrix for selected atoms
```

```
In [16]: anm.calcModes() # calculate normal modes
```

```
In [17]: saveModel(anm)
Out [17]: 'ubi.anm.npz'
```

*Anisotropic Network Model (ANM)*³ provides a more detailed discussion of ANM calculations. The above longer way gives more control to the user. For example, instead of building the Hessian matrix using uniform force constant and cutoff distance, customized force constant functions (see *Custom Gamma Functions*⁴) or a pre-calculated matrix (see `ANM.setHessian()`) may be used.

Individual Mode instances can be accessed by indexing the ANM instance:

```
In [18]: slowest_mode = anm[0]
```

```
In [19]: print( slowest_mode )
Mode 1 from ANM ubi
```

```
In [20]: print( slowest_mode.getEigval().round(3) )
1.714
```

Note that indices in Python start from zero (0). 0th mode is the 1st non-zero mode in this case. Let's confirm that normal modes are orthogonal to each other:

```
In [21]: (anm[0] * anm[1]).round(10)
Out [21]: 0.0
```

```
In [22]: (anm[0] * anm[2]).round(10)
Out [22]: 0.0
```

³http://prody.csb.pitt.edu/tutorials/enm_analysis/anm.html#anm

⁴http://prody.csb.pitt.edu/tutorials/enm_analysis/gamma.html#gamma

As you might have noticed, multiplication of two modes is nothing but the `dot()`⁵ product of mode vectors/arrays. See *Normal Mode Algebra*⁶ for more examples.

8.3 Comparative Analysis

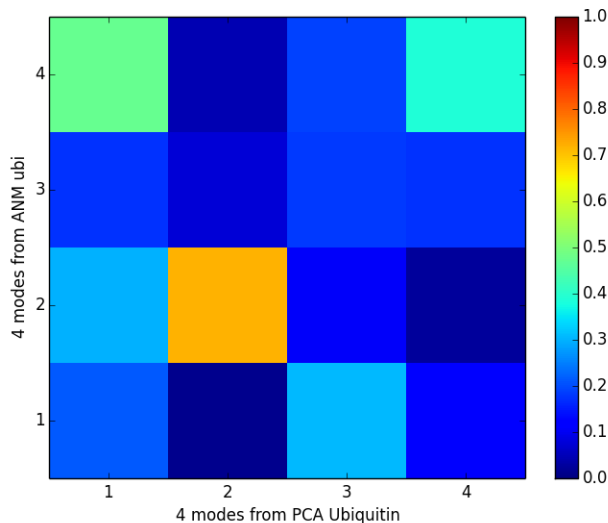
ProDy comes with many built-in functions to facilitate a comparative analysis of experimental and theoretical data. For example, using `printOverlapTable()` function you can see the agreement between experimental (PCA) modes and theoretical (ANM) modes calculated above:

```
In [23]: printOverlapTable(pca[:4], anm[:4])
Overlap Table
```

		ANM ubi			
		#1	#2	#3	#4
PCA Ubiquitin #1		-0.21	+0.30	-0.17	-0.47
PCA Ubiquitin #2		+0.01	+0.72	+0.08	+0.05
PCA Ubiquitin #3		+0.31	+0.11	+0.18	+0.19
PCA Ubiquitin #4		+0.11	-0.02	-0.17	-0.39

Output above shows that PCA mode 2 and ANM mode 2 for ubiquitin show the highest overlap (cosine-correlation).

```
In [24]: showOverlapTable(pca[:4], anm[:4]);
```



This was a short example for a simple case. *Ensemble Analysis*⁷ section contains more comprehensive examples for heterogeneous datasets. *Analysis*⁸ shows more analysis function usage examples and *Dynamics Analysis*⁹ module documentation lists all of the analysis functions.

⁵<http://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html#numpy.dot>

⁶http://prody.csb.pitt.edu/tutorials/enm_analysis/normalmodes.html#normalmode-operations

⁷http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#pca

⁸http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_analysis.html#pca-xray-analysis

⁹<http://prody.csb.pitt.edu/manual/reference/dynamics/index.html#dynamics>

8.4 Output Data Files

The `writeNMD()` function writes PCA results in NMD format. NMD files can be viewed using the *Normal Mode Wizard*¹⁰ VMD plugin.

```
In [25]: writeNMD('ubi_pca.nmd', pca[:3], ubi_selection)
Out[25]: 'ubi_pca.nmd'
```

Additionally, results can be written in plain text files for analysis with other programs using the `writeArray()` function:

```
In [26]: writeArray('ubi_pca_modes.txt', pca.getArray(), format='%8.3f')
Out[26]: 'ubi_pca_modes.txt'
```

8.5 External Data

Normal mode data from other NMA, EDA, or PCA programs can be parsed using `parseModes()` function for analysis.

In this case, we will parse ANM modes for p38 MAP Kinase calculated using *ANM server*¹¹ as the external software. We use `oanm_eigvals.txt` and `oanm_slwevs.txt` files from the ANM server.

You can either download these files to your current working directory from here or obtain them for another protein from the ANM server.

```
In [27]: nma = parseModes(normalmodes='oanm_slwevs.txt',
.....: eigenvalues='oanm_eigvals.txt',
.....: nm_usecols=range(1,21), ev_usecols=[1], ev_usevalues=range(6,26))
.....:
```

```
In [28]: nma
Out[28]: <NMA: oanm_slwevs (20 modes; 351 atoms)>
```

```
In [29]: nma.setTitle('lp38 ANM')
```

```
In [30]: slowmode = nma[0]
```

```
In [31]: print(slowmode.getEigval().round(2))
0.18
```

8.6 Plotting Data

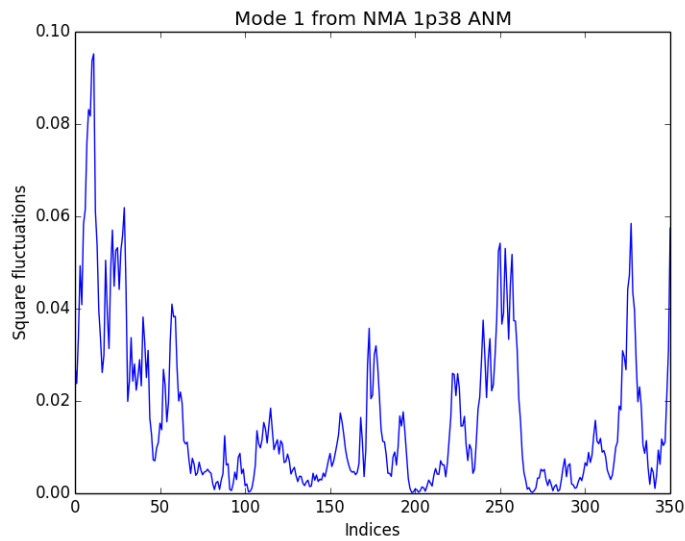
If you have *Matplotlib*¹², you can use functions whose name start with `show` to plot data:

```
In [32]: showSqFlucts(slowmode);
```

¹⁰http://prody.csb.pitt.edu/tutorials/nmwiz_tutorial/intro.html#nmwiz

¹¹<http://ignmtest.cccb.pitt.edu/cgi-bin/anm/anm1.cgi>

¹²<http://matplotlib.org>



*Plotting*¹³ shows more plotting examples and *Dynamics Analysis*¹⁴ module documentation lists all of the plotting functions.

8.7 More Examples

For more examples see *Elastic Network Models*¹⁵ and *Ensemble Analysis*¹⁶ tutorials.

¹³http://prody.csb.pitt.edu/tutorials/ensemble_analysis/xray_plotting.html#pca-xray-plotting

¹⁴<http://prody.csb.pitt.edu/manual/reference/dynamics/index.html#dynamics>

¹⁵http://prody.csb.pitt.edu/tutorials/enm_analysis/index.html#enm-analysis

¹⁶http://prody.csb.pitt.edu/tutorials/ensemble_analysis/index.html#ensemble-analysis

SEQUENCE ANALYSIS

Evol component of ProDy package brought new fast, flexible, and efficient features for handling multiple sequence alignments and analyzing sequence evolution. Here, we just give a brief introduction to these features. For more detailed examples, see *Evol Tutorial*¹.

```
In [1]: from prody import *
```

```
In [2]: from pylab import *
```

```
In [3]: ion()
```

9.1 Access Pfam

First, let's fetch an MSA file from Pfam² database:

```
In [4]: filename = fetchPfamMSA('pkinase', alignment='seed')
```

```
In [5]: filename
```

```
Out [5]: 'pkinase_seed.sth'
```

We downloaded the seed alignment for Protein Kinase (Pkinase³) family.

9.2 Parse MSA

As you might guess, we will parse this file using `parseMSA()` function:

```
In [6]: msa = parseMSA(filename)
```

```
In [7]: msa
```

```
Out [7]: <MSA: pkinase_seed (54 sequences, 476 residues)>
```

9.3 Sequences

You can access Sequence objects by indexing MSA:

¹http://prody.csb.pitt.edu/tutorials/evol_tutorial/index.html#evol-tutorial

²<http://pfam.sanger.ac.uk/>

³<http://pfam.sanger.ac.uk/family/Pkinase>

```
In [8]: seq = msa[0]
```

```
In [9]: seq
```

```
Out[9]: <Sequence: CHK1_SCHPO (pkinase_seed[0]; length 476; 263 residues and 213 gaps)>
```

```
In [10]: print(seq)
```

```
YHIGREIGTGAFASV..RLCYDDNAKI.....YAVKQVFNKK.HATSCMNAGVWARR.....MASEIQLHKLCNG....HKN..I..I
```

You can also slice MSA objects and iterate over sequences:

```
In [11]: for seq in msa[:4]:
.....:     repr(seq)
.....:
```

9.4 Analysis

Evol component includes several functions for calculating conservation and coevolution properties of amino acids, which are shown in *Evol Tutorial*⁴. Here, let's take a look at `calcMSAOccupancy()` and `showMSAOccupancy()` functions:

```
In [12]: occ = calcMSAOccupancy(msa, count=True)
```

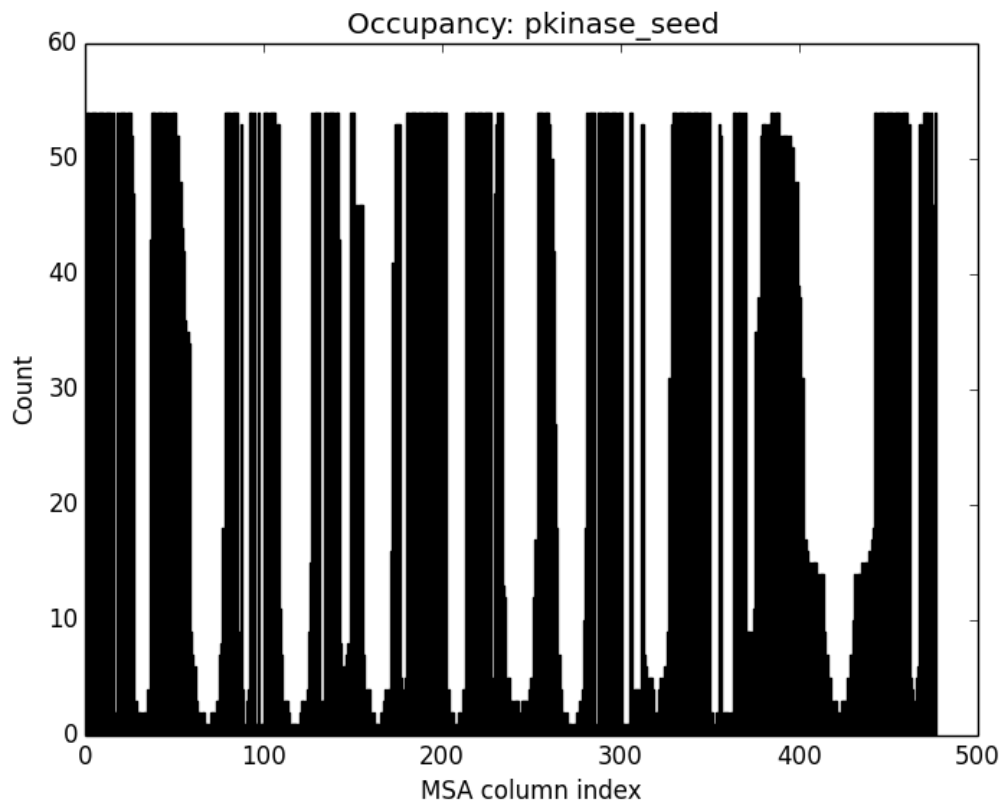
```
In [13]: occ.min()
```

```
Out[13]: 1.0
```

This shows that an amino acid is present only in one of the sequences in the MSA.

```
In [14]: showMSAOccupancy(msa, count=True);
```

⁴http://prody.csb.pitt.edu/tutorials/evol_tutorial/index.html#evol-tutorial



You see that many residues are not present in all sequences. You will see how to refine such MSA instances in *Evol Tutorial*⁵.

⁵http://prody.csb.pitt.edu/tutorials/evol_tutorial/index.html#evol-tutorial

APPLICATIONS TUTORIAL

You can use *ProDy Applications*¹ to perform some automated tasks, such as ANM/GNM/PCA calculations, fetching and aligning PDB files, making atom selections, or identify contacts. ProDy applications are handled by a script that comes with all installation packages. You can run the script from a central location such as `/usr/local/bin`:

```
$ prody -h
```

or from the current working directory:

```
$ ./prody -h
```

or:

```
$ python prody -h
```

or on Windows:

```
$ C:\Python27\python.exe C:\Python27\Scripts\prody -h
```

These lines will print available ProDy applications. You can get more help on a specific commands as follows:

```
$ prody anm -h
```

10.1 Align PDB files

*prody align*² command can be used to download and align structures for given PDB identifiers:

```
$ prody align 1p38 1r39 1zz2
```

Structures will be automatically downloaded from wwPDB FTP servers and saved in the current working directory. Additionally, you can configure ProDy to use a local mirror of PDB or to store downloaded files in a local folder. See *ProDy Basics* (page 6) part of the tutorial.

10.2 ANM calculations

*prody anm*³ can be used to perform ANM calculations:

¹<http://prody.csb.pitt.edu/manual/apps/prody/index.html#prody-apps>

²<http://prody.csb.pitt.edu/manual/apps/prody/align.html#prody-align>

³<http://prody.csb.pitt.edu/manual/apps/prody/anm.html#prody-anm>

```
$ prody anm lp38 -a -A
```

-a and -A options will make ProDy output all data and figure files.

10.3 PCA calculations

*prody anm*⁴ can be used to perform PCA calculations. The following example will perform PCA calculations for C α atoms of the p38 MAP kinase using files:

- ProDy Tutorial files (ZIP)
- ProDy Tutorial files (TGZ)

```
$ tar -xzf p38_trajectory.tar.gz
$ prody pca -a -A --select calpha --pdb p38.pdb p38_100frames.dcd
```

Acknowledgments

Continued development of Protein Dynamics Software *ProDy* is supported by NIH through R01 GM099738 award. Development of this tutorial is supported by NIH funded Biomedical Technology and Research Center (BTRC) on *High Performance Computing for Multiscale Modeling of Biological Systems (MMBios*⁵) (P41 GM103712).

⁴<http://prody.csb.pitt.edu/manual/apps/prody/anm.html#prody-anm>

⁵<http://mmbios.org/>