Afpdb - An Efficient Protein Structure Manipulation Python Package

Yingyao Zhou

https://github.com/data2code/afpdb
Last updated: September 10, 2024

Contents

Introduction

The emergence of AlphaFold and subsequent protein AI models has revolutionized protein design. To maximize the probability of success, the AI-driven protein design process involves analyzing thousands of protein structures. This includes handling structure file read/write operations, aligning structures and measuring structural deviations, standardizing chain/residue labels, extracting residues, identifying mutations, and creating visualizations. However, existing programming packages do not fully address these challenges. To bridge this gap, we introduce the Afpdb module. Built upon AlphaFold's NumPy architecture and leveraging the intuitive contig syntax proposed by RFDiffusion, Afpdb streamlines structure analyses. While supplementing Biopython with dozens of methods commonly used in protein AI design but not readily available elsewhere; it also offers a user-friendly interface that seamlessly integrates with PyMOL's visualization capabilities.

This document is a tutorial on Afpdb. Throughout this tutorial, we will use two example antigen-antibody complex structures: 5cil.pdb and fake.pdb. 5cil.pdb contains three chains, L and H chains for the antibody light chain and heavy chain, respectively, and P chain for the antigen. fake.pdb contains two short chains, the L chain contains 4 residues including two with insertion codes and 3 missing residues, and the H chain contains two residues.

The best way to learn Afpdb is to open the Jupyter Notebook in Google Colab and play with the examples. This PDF is meant to clone the content of the Jupyter Notebook for the convenience of offline reading. The Jupyter Notebook does not export nicely into a PDF file, we therefore decided to recreate the content. As the Jupyter Notebook is frequently updated, this PDF version only served as a snapshot in time for user's learning purpose, therefore, the content may become outdated. If you use a Linux environment, however, we expect the notebook can run in other platforms if Afpdb has been installed.

Installation

To install the Afpdb package:

```
pip install git+https://github.com/data2code/afpdb.git
```

To enable PyMOL-dependent features

```
conda install conda-forge::pymol-open-source
```

To assign secondary structures:

```
conda install sbl::dssp
```

The Notebook contains more detailed codes for the Colab environment and for developers.

Demo

To convince you Afpdb is effective and elegant, let us look at the following example, where we first identify contact residues in the antibody-antigen interface and then extract those residues into a separate PDB file. The structure display confirms the extraction was correct. These manipulations are not so straightforward using PyMOL or Biopython.

```
# load the ab-ag complex structure using PDB code name
p=Protein("5cil")
# show key statistics summary of the structure
p.summary()
```

	Chain	Sequence	Length	#Missing Residues	#Insertion Code	First Residue Name	Last Residue Name	#Missing Backbone
0	H	VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEW...	220	20	14	2	227	0
1	L	EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPR...	212	0	1	1	211	0
2	P	NWFDITNWLWYIK	13	0	0	671	683	0

Notice that residue names do not start from "1" for chain "H" and chain "L" (see column "First Residue Name"), some residues in the CDR regions have insertion code (see column "#Insertion Code"), so that residue numbering goes from 1 to 211 for 212 residues on chain "L". It is good that learn that all residues have the complete set of backbone atoms "N,CA,C,O".

We can standardize residue naming (we use blue fonts for outputs):

```
print("Old P chain residue numbering:", p.rs("P").name(), "\n")
```

```
Old P chain residue numbering: ['671', '672', '673', '674', '675', '676', '677', '678', '679',
'680', '681', '682', '683']
```

```
q, old_num=p.renumber("RESTART")
print("New P chain residue numbering:", q.rs("P").name(), "\n")
```

```
New P chain residue numbering: ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12',
'13']
```

```
q.summary()
```

	Chain	Sequence	Length	#Missing Residues	#Insertion Code	First Residue Name	Last Residue Name	#Missing Backbone
0	H	VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEW...	220	20	14	1	226	0
1	L	EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPR...	212	0	1	1	211	0
2	P	NWFDITNWLWYIK	13	0	0	1	13	0

```
# We can further remove the insertion code by
# renumber() returns a tuple, (Protein obj, old numbers)
q=p.renumber("NOCODE")[0].renumber("RESTART")[0]
q.summary()
```

	Chain	Sequence	Length	#Missing Residues	#Insertion Code	First Residue Name	Last Residue Name	#Missing Backbone
0	H	VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEW...	220	20	0	1	240	0
1	L	EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPR...	212	0	0	1	212	0
2	P	NWFDITNWLWYIK	13	0	0	1	13	0

To predict the structure with AlphaFold, we can replace missing residues with glycines. Notice we refer to missing residues within a single chain, not using glycines to connect different chains.

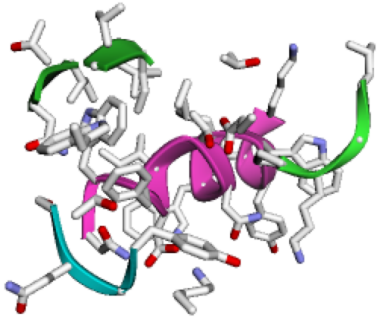
```
print("Sequence for AlphaFold modeling, with the 20 missing residues replaced by Glycine:")
print(">5cil\n"+p.seq(gap="G")+"\n")
```

```
Sequence for AlphaFold modeling, with the 20 missing residues replaced by Glycine:
>5cil
VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAR
EGTTGDGDLGKPIGAFAHWGQGTLVTVSSASTKGPSVFPLAPSGGGGGGGGTAALGCLVKDYFPEPVTVGSWGGGGNSGALTSGGVHTFPAVLQSG
SGLYSLSSVVTVPSSSLGTGGQGTYICNVNHKPSNTKVDKKGGVEP:EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIY
GASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTVAAPSVFIFPPSDEQLKSGTASVVCLLNNFYPREAKV
QWKVDNALQSGNSQESVTEQDSKDSTYSLSSTLTLSKADYEKHKVYACEVTHQGLSSPVTKSFNR:NWFDITNWLWYIK
```

```
# identify H,L chain residues within 4A to antigen P chain
binder, target, df_dist=p.rs_around("P", dist=4)
# show the distance of binder residues to antigen P chain
df_dist[:5]
```

	chain_a	resi_a	resn_a	resn_i_a	atom_a	chain_b	resi_b	resn_b	resn_i_b	atom_b	dist
90	P	437	6	6	OG1	H	97	98	98	OE2	2.636247
72	P	435	4	4	OD1	L	252	33	33	NZ	2.814822
21	P	433	2	2	N	L	314	95	95	OG	2.911941
12	P	432	1	1	ND2	L	311	92	92	O	2.929501
93	P	438	7	7	ND2	H	107	108	108	CE	3.038573

```
# create a new PDB file only containing the antigen and binder residues
p=p.extract(binder | "P")
# save the new structure into a local PDB file
p.save("binding.pdb")

# display the PDB struture, default is show ribbon and color by chains.
p.show(show_sidechains=True)
```



The above script shows a few strengths of Afpdbs:

- User friendly: We use `Protein()` to create a protein object, use "'save()"' to create a PDB file. We will explain both methods can accommodate multiple formats.
- Residue selection: We use "H:L" and "P" to select all Ab and Ag residues, we use `binder | "P"` to Boolean combine selections. Afpdb supports a "contig" syntax to select residues with their human readable labels.
- Visualization: We here show how a protein object can be visualized within Jupyter Notebook. We will explain how Afpdb integrates with PyMOL for more powerful 3D visualization and transfer the residues selections from Afpdb into PyMOL.
- AI ready: Afpdb was created to meet our own protein AI design needs. We show a brief example of how Afpdb prepares an input for AlphaFold, and we will show more sophisticated AI use cases.

Fundamental Concepts

Internal Data Structure

Data Members

From the demo, we see how fast Afpdb analyses are, which is because Afpdb uses NumPy arrays for storing protein structure data, which is very different from most existing packages relying on the model-chain-residue-atom tree structure.

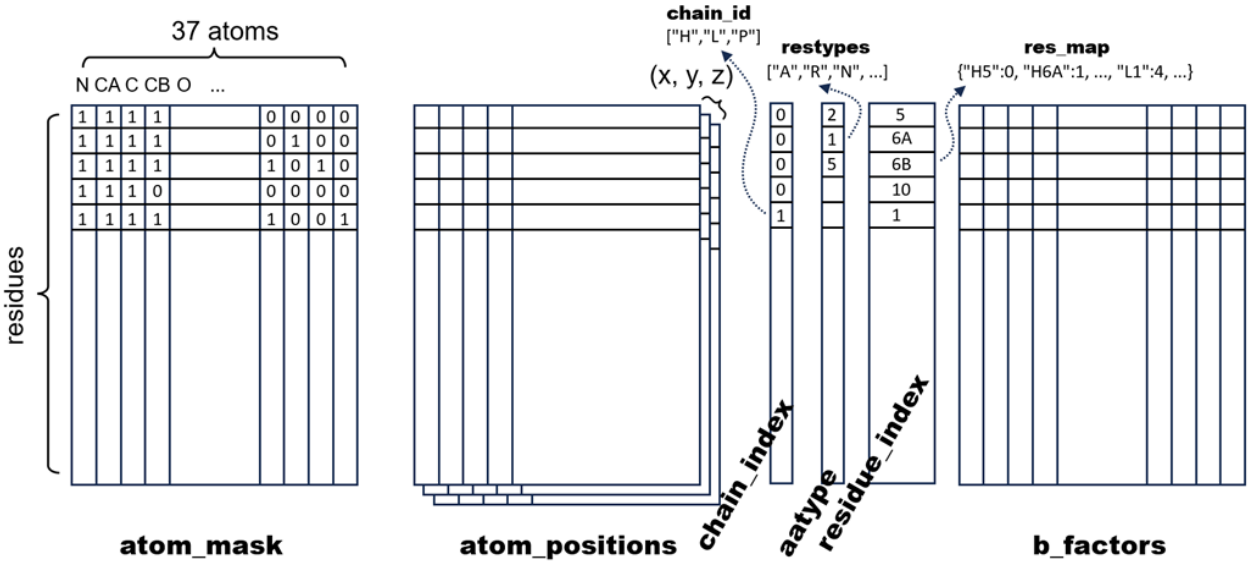
To better understand why Afpdb can manipulate protein structures efficiently, it is helpful to first discuss how the structure data in a PDB file is stored within the AlphaFold's Protein class. The following descriptions are mostly taken from DeepMind's `alphafold/common/protein/protein.py` file.

```
# Cartesian coordinates of atoms in angstroms. The atom types correspond to
# residue_constants.atom_types, i.e. the first three are N, CA, C.
atom_positions: np.ndarray  # [num_res, num_atom_type, 3]

# Amino-acid type for each residue represented as an integer between 0 and
# 20, where 20 is 'X'.
aatype: np.ndarray  # [num_res]

# Binary float mask to indicate presence of a particular atom. 1.0 if an atom
# is present and 0.0 if not. This should be used for loss masking.
atom_mask: np.ndarray  # [num_res, num_atom_type]

# Residue index as used in PDB. It is not necessarily continuous or 0-indexed.
residue_index: np.ndarray  # [num_res]
```

We modified residue_index to dtype "<U6" in order to be able to store residue id with possible insertion code. A `residue_index` was an integer in AlphaFold, in our extension, `residue_index` can include insertion code, e.g., "121", "122A", "112B". The maximum residue id can be 9999 if we allow insertion codes with two letters, otherwise, 99999 if all insertion codes are single letters.

```
# 0-indexed number corresponding to the chain in the protein that this residue
# belongs to.
chain_index: np.ndarray  # [num_res]

# B-factors, or temperature factors, of each residue (in sq. angstroms units),
# representing the displacement of the residue from its ground truth mean
# value.
b_factors: np.ndarray  # [num_res, num_atom_type]

# Chain names - this is added by us. DeepMind's original code does not memorize the chain
# name in the PDB file

# chain names became A, B, C, etc. afpdb keeps the original chain names in the PDB file
chain_id: np.ndarray # [num_chains]
```

Illustration



Notice the first axis of all NumPy arrays are the number of residues. The second axis can be either one or the number of possible non-hydrogen atom types (which is 37).

The above is AlphaFold's Protein class, developers access this data via Afpdb's Protein class, which is a wrapper of AlphaFold's protein class.

In Afpdb's Protein class, it contains an attribute called `data`, which points to an AlphaFold's Protein instance.

```
from afpdb.afpdb import Protein
p=Protein(fk)
# p.data points to AlphaFold's Protein object
print(p.data.atom_positions[:, 1])

# outputs
# [[  6.30200005 -14.27600002 -18.36100006]
#  [  9.34000015 -16.29100037 -17.33600044]
#  [  9.67800045 -16.51799965 -13.59899998]
#  [ 16.89500046 -17.8920002   -7.03000021]
#  [ 25.96999931   6.87099981 -19.03800011]
#  [ 25.11199951   9.9829998  -16.98600006]]

print(p.res_map)

# outputs
# {'L5': 0, 'L6A': 1, 'L6B': 2, 'L10': 3, 'H3': 4, 'H4': 5}
```

In the example above, we obtain the (X, Y, Z) coordinates of the CA atoms (i.e., we specify 1 for the 2nd axis of `atom_positions`) corresponding to the first six residues via accessing `p.data`.

To further speed up structure manipulations, Afpdb's Protein class maintains a dictionary called `res_map`, which maps the full residue id, in the format of `{chain}{residue_index}`, into its residue position. `{chain}{residue_index}` is the user-friendly format for identifying a specific residue, e.g., "C122A" is the residue index "122A" on chain "C". This residue corresponds to an integer index in the NumPy arrays internally. Developers should remember to call `self._make_res_map()` to update this mapping dictionary, if a method requires the underlying mapping to be refreshed after changing the positions/types of residues for any location. The example above shows how the six full residue names are mapped to the NumPy arrays.

For the rest of the tutorial, when we use the term "Protein" class, we refer to Afpdb's Protein class, which can be imported using `from afpdb.afpdb import Protein`. We will no longer mention AlphaFold's Protein class to avoid confusion, infact, users do not need to directly access `p.data`, only developers need to be aware that `p.data` points to multiple NumPy arrays behind the scenes.

Example

Let us further explain the previous example and examine `p.data` NumPy arrays using fake.pdb.

The following is the fake.pdb file, which contains 4-residue chain L and a 2-residue chain H.

```
MODEL     1
ATOM      1  N   GLU L   5       5.195 -14.817 -19.187  1.00  1.00           N
ATOM      2  CA  GLU L   5       6.302 -14.276 -18.361  1.00  1.00           C
ATOM      3  C   GLU L   5       7.148 -15.388 -17.731  1.00  1.00           C
ATOM      4  CB  GLU L   5       5.794 -13.368 -17.248  1.00  1.00           C
ATOM      5  O   GLU L   5       6.658 -16.231 -17.006  1.00  1.00           O
ATOM      6  CG  GLU L   5       6.934 -12.664 -16.494  1.00  1.00           C
ATOM      7  CD  GLU L   5       6.461 -11.817 -15.327  1.00  1.00           C
ATOM      8  OE1 GLU L   5       7.282 -11.138 -14.677  1.00  1.00           O
ATOM      9  OE2 GLU L   5       5.243 -11.804 -15.070  1.00  1.00           O
ATOM     10  N   ILE L   6A      8.444 -15.321 -17.934  1.00  1.00           N
ATOM     11  CA  ILE L   6A      9.340 -16.291 -17.336  1.00  1.00           C
ATOM     12  C   ILE L   6A      9.657 -15.849 -15.925  1.00  1.00           C
ATOM     13  CB  ILE L   6A     10.604 -16.433 -18.162  1.00  1.00           C
ATOM     14  O   ILE L   6A     10.192 -14.739 -15.685  1.00  1.00           O
ATOM     15  CG1 ILE L   6A     10.228 -16.847 -19.590  1.00  1.00           C
ATOM     16  CG2 ILE L   6A     11.540 -17.469 -17.523  1.00  1.00           C
ATOM     17  CD1 ILE L   6A     11.401 -17.319 -20.426  1.00  1.00           C
ATOM     18  N   VAL L   6B      9.339 -16.725 -14.982  1.00  1.00           N
ATOM     19  CA  VAL L   6B      9.678 -16.518 -13.599  1.00  1.00           C
ATOM     20  C   VAL L   6B     11.024 -17.188 -13.330  1.00  1.00           C
ATOM     21  CB  VAL L   6B      8.569 -17.028 -12.666  1.00  1.00           C
ATOM     22  O   VAL L   6B     11.242 -18.372 -13.679  1.00  1.00           O
ATOM     23  CG1 VAL L   6B      8.960 -16.919 -11.194  1.00  1.00           C
ATOM     24  CG2 VAL L   6B      7.268 -16.234 -12.927  1.00  1.00           C
ATOM     40  N   GLN L  10      15.587 -17.776  -7.649  1.00  1.00           N
ATOM     41  CA  GLN L  10      16.895 -17.892  -7.030  1.00  1.00           C
ATOM     42  C   GLN L  10      16.721 -18.330  -5.569  1.00  1.00           C
ATOM     43  CB  GLN L  10      17.616 -16.572  -7.093  1.00  1.00           C
ATOM     44  O   GLN L  10      16.270 -17.557  -4.746  1.00  1.00           O
ATOM     45  CG  GLN L  10      17.963 -16.094  -8.483  1.00  1.00           C
ATOM     46  CD  GLN L  10      18.781 -14.822  -8.460  1.00  1.00           C
ATOM     47  NE2 GLN L  10      20.052 -14.951  -8.083  1.00  1.00           N
ATOM     48  OE1 GLN L  10      18.284 -13.727  -8.786  1.00  1.00           O
ATOM    862  N   LEU H   3      27.368   6.440 -19.107  1.00  1.00           N
ATOM    863  CA  LEU H   3      25.970   6.871 -19.038  1.00  1.00           C
ATOM    864  C   LEU H   3      25.761   7.794 -17.840  1.00  1.00           C
ATOM    865  CB  LEU H   3      25.089   5.647 -18.873  1.00  1.00           C
ATOM    866  O   LEU H   3      25.979   7.398 -16.661  1.00  1.00           O
ATOM    867  CG  LEU H   3      25.225   4.606 -19.964  1.00  1.00           C
ATOM    868  CD1 LEU H   3      24.282   3.430 -19.748  1.00  1.00           C
ATOM    869  CD2 LEU H   3      24.962   5.190 -21.355  1.00  1.00           C
ATOM    870  N   VAL H   4      25.291   9.008 -18.090  1.00  1.00           N
ATOM    871  CA  VAL H   4      25.112   9.983 -16.986  1.00  1.00           C
ATOM    872  C   VAL H   4      23.672  10.399 -16.963  1.00  1.00           C
ATOM    873  CB  VAL H   4      25.976  11.257 -17.222  1.00  1.00           C
ATOM    874  O   VAL H   4      23.143  10.875 -17.973  1.00  1.00           O
ATOM    875  CG1 VAL H   4      25.686  12.319 -16.177  1.00  1.00           C
ATOM    876  CG2 VAL H   4      27.466  10.902 -17.258  1.00  1.00           C
TER     877      VAL H   4
ENDMDL
END
```

We first read in fake.pdb, there is a warning indicating there are residues with insertion codes: 6A and 6B. This alerts users that there may be duplicates in the integer portion of the residue ID, so it is a good habit

not to rely on the assumption that residue integer IDs are all unique. AlphaFold's original code did not support insertion codes. Afpdb made modifications to eliminate this limitation.

```
p=Protein(fk)
print(p.data.warning(), "\n")
```

```
{'renamed_res': [], 'insertion_res': [('L6A', 'ILE'), ('L6B', 'VAL')], 'unknown_res': []}
```

```
# show the sequences by chain
print(p.seq_dict())
```

```
{'L': 'EIVXXXQ', 'H': 'LV'}
```

```
# print the total number of residues
print("Total length:", len(p), "\n")
```

```
Total length: 6
```

```
print("Residue labels:", p.rs().name(), "\n")
```

```
Residue labels: ['5', '6A', '6B', '10', '3', '4']
```

```
print("Their chain labels:", p.rs().chain())
```

```
Their chain labels: ['L', 'L', 'L', 'L', 'H', 'H']
```

Here `p.rs()` is a "residue selection" that points to all residues. The residue selection, to be explained later, can be used to fetch the metadata for residues (their labels, chain names, amino acid types, etc). The most convenient way to access residue identifiers as a `DataFrame` object is to use its `df()` method.

	resi	chain	resn	namei	code	aa	unique_name
0	0	L	5	5		E	L5
1	1	L	6A	6	A	I	L6A
2	2	L	6B	6	B	V	L6B
3	3	L	10	10		Q	L10
4	4	H	3	3		L	H3
5	5	H	4	4		V	H4

Our protein object p contains an attribute `data`, which points to AlphaFold's NumPy arrays. First, any warnings during the parsing of the PDB file can be extracted using `p.data.warning()`, which includes unrecognized residues, special forms that are renamed, and residues with an insertion code.

Let us examine the backend NumPy data within `p.data`. Here `p.rs()` is a "residue selection" that points to all residues. The residue selection, to be explained later, can be used to fetch the metadata for residues (their labels, chain names, amino acid types, etc.).

```
g=p.data
# we used p.rs() to select ALL residues
# we used p.rs().name() to get all residue labels, which is stored in g.residue_index
print(g.residue_index, "\n")
```

```
['5' '6A' '6B' '10' '3' '4']
```

```
# p.res_map maps each unique residue label to their row indices in the NumPy arrays.
print(p.res_map, "\n")
```

```
{'L5': 0, 'L6A': 1, 'L6B': 2, 'L10': 3, 'H3': 4, 'H4': 5}
```

```
# we used p.rs().chain() to get the chain name of each residue
# chain information is stored in g.chain_index
```

```
print(g.chain_index, "\n")
```

```
[0 0 0 0 1 1]
```

```
# To obtain the list of chain names in the order of their appearance in the PDB file,
# we should use the chain_id() method instead
print(p.rs().chain(), "\n")
```

```
['L', 'L', 'L', 'L', 'H', 'H']
```

```
print(p.chain_id())
```

```
['L', 'H']
```

`g.chain_index` shows the first 4 residues belong to chain index 0, and the next 2 residues belong to chain index 1. AlphaFold uses Biopython to parse PDB files, Chains are numbered in the order of their appearance in the PDB file, so chain L is indexed as chain 0 and chain H is indexed as chain 1. It is important to remember that the best practice is to access data without assuming how chains appear in a certain order in the PDB or in Protein objects. It is safer to access structure or sequence data by their chain names.

What are the chain names for 0 and 1? AlphaFold's original code did not store chain names and chains were named as A, B, C, etc. Afpdb has improved AlphaFold's code to add a new data member `chain_id()` into its Protein class.

Array `aatype` stores the identity of amino acids as defined in alphafold.common.residue_constants.py:

```
restypes = ['A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T',
'W', 'Y', 'V']
```

So index 6 translates into "E".

There are a total of 6 residues. By assuming residues are numbered sequentially, we imply three residues (7, 8, and 9) are missing in the PDB file. Missing residues are not stored in the NumPy arrays; they can only be inferred based on the gaps in the integer portion of the residue IDs.

```
# afres contains protein structure constants defined by AlphaFold
import afpdb.myalphafold.common.residue_constants as afres
print(g.aatype, "\n")
```

```
[ 6  9 19  5 10 19]
```

```
[ afres.restypes[x] for x in g.aatype ]
```

```
['E', 'I', 'V', 'Q', 'L', 'V']
```

Users do not have to use `afres`, residue types can be extracted with the residue selection object:

```
# Notice three missing residues are represented as 3 "X", but they are not there in the protein s
tructure
# notice sequences from different chains are separated by ":"
print(p.seq(), "\n")
```

```
EIVXXXQ:LV
```

```
# retrieve sequence information via residue selection
p.rs().aa()
```

```
['E', 'I', 'V', 'Q', 'L', 'V']
```

There are different types of metadata for residues. In Afpdb, we use the following convention:

- **resn** refers to the residue_index, a string with an optional insertion code without chain name.
- **resi** refers to the internal numpy array indices, e.g., the resi for resn = '6A' is 1.
- **resn_i** refers to the integer portion of the resn, resn_i for residue 6B is integer 6. `p.res_map` maps the unique residue name, (chain + **resn**), into **resi**.

Note: Early Afpdb users use `p.data_prt`, although this is still supported for now, please switch to use `p.data` going forward.

There are a total of 37 possible atom types when all 20 amino acid types are combined (not counting hydrogens), AlphaFold uses an array of 6 x 37 to store atom 3D coordinates (we have 6 residues in this example). Therefore, the atom_positions is a 3-dimensional NumPy array, where the last dimension is for (X, Y, Z) coordinates:

```
g.atom_positions.shape
```

```
(6, 37, 3)
```

No residue will have all 37 atoms, they all have N, CA, C, and O, but do not necessarily have CB, CG, etc. Therefore, AlphaFold's code uses slightly more memory than needed to store the coordinates. As PDB data are small, trading extra memory to gain convenience and speed with NumPy arrays is wise. This is why implementing new computations in Afpdb can be done quickly, as no boilerplate loops are required to navigation the model/chain/residue/atom tree to access coordinate data, like what is the required for using the Biopython API.

As not all array members are used, we thus need to know what atoms in the `atom_positions` array contain real data using a binary array `atom_mask`:

```
print(ATS.i('CA'), "\n")
```

```
1
```

```
print(g.atom_mask[:, ATS.i('CA')], "\n")
```

```
[1. 1. 1. 1. 1. 1.]
```

```
print(g.atom_positions[:, ATS.i('CA')], "\n")
```

```
[[  6.30200005 -14.27600002 -18.36100006]
 [  9.34000015 -16.29100037 -17.33600044]
 [  9.67800045 -16.51799965 -13.59899998]
 [ 16.89500046 -17.8920002   -7.03000021]
 [ 25.96999931   6.87099981 -19.03800011]
 [ 25.11199951   9.9829998  -16.98600006]]
```

```
print(g.atom_mask[:, ATS.i('CG')], "\n")
```

```
[1. 0. 0. 1. 1. 0.]
```

```
print(g.atom_positions[:, ATS.i('CG')], "\n")
```

```
[[  6.93400002 -12.66399956 -16.49399948]
 [  0.           0.           0.        ]
 [  0.           0.           0.        ]
 [ 17.96299934 -16.09399986  -8.4829998 ]
 [ 25.22500038   4.60599995 -19.9640007 ]
 [  0.           0.           0.        ]]
```

```
str(~ p.ats('CA'))
```

```
'N,C,CB,O,CG,CG1,CG2,OG,OG1,SG,CD,CD1,CD2,ND1,ND2,OD1,OD2,SD,CE,CE1,CE2,CE3,NE,NE1,NE2,OE1,OE2,CH
```

```
2,NH1,NH2,OH,CZ,CZ2,CZ3,NZ,OXT'
```

We refer to an atom with its name instead of memorizing its index in the arrays. ATS is the atom selection class to be introduced later. `ATS.i` method converts an atom name into its integer index. We also show a simple `"~"` operator to demonstrate selection objects can be Boolean manipulated.

We see that all six residues have CA (Cα) atom, but only the 1st, 4th, and 5th residues (E, Q, L) have (Cγ). So the coordinates for those atoms within `atom_positions` array, but with corresponding atom_mask set to 0 should be ignored (these atoms are not present in the PDB structure). Although atoms with coordinates at the origin are likely masked-out atoms, we should not rely on that assumption. In fact, when AlphaFold2 prediction starts, all atoms are placed at the origin -- so-called black hole conformation. Therefore, we should rely on the `atom_mask` array to determine the presence of atoms and assume the coordinates of masked out atoms can be any garbage numbers.

You might worry that the extra array elements for masked atoms will create computational overhead. In the Developers' Note section, we will explain methods that convert this sparse coordinate NumPy array into a dense coordinate NumPy array. Indeed, eliminating non-existing atoms further accelerate the computation and that is used in Afpdb's analyses.

Lastly and similarly, `b_factors` are also stored in an array at the atom level. Only elements with atom_mask equaling 1 have meaningful b factors.

```
g.b_factors.shape
```

```
(6, 37)
```

To summarize, for a given Afpdb Protein object p, we store all key PDB information into a few NumPy arrays: `atom_positions`, `atom_mask`, `residue_index`, `chain_index`, `b_factors`, and `chain_id` within p.data. We also have a helper dictionary `p.res_map`.

These arrays make structure manipulations a breeze. End users do not need to access these NumPy arrays, the architecture is for developers to understand.

Contig

Based on how Afpdb uses NumPy arrays to store structure data, we can access a few residues and a few atoms is as easily as accessing certain rows and columns in these NumPy arrays. To specify what residues are of interest, we use a string format called **contig**. Contig is a string format introduced by recent protein AI packages such as RFDiffusion and ProteinMPNN, we adopt and slightly improve the syntax as detailed below.

Single Residue

`RESIDUE = CHAIN RESIDUE_INDEX` without a space in between

In PDB, each residue is assigned an integer id. In some cases, a residue label (RESIDUE_INDEX) can be followed by an insertion code, which is an alphabet letter (two letters in very rare cases). This is because when insertions were made to a residue, such as residue 100 in the antibody CDR loops, people prefer to call the new residues 100A and 100B so that the rest of the residues do not need to be renumbered.

The residue label alone without the chain name is not unique, because the same residue label can appear on different chains. Therefore, to fully specify a residue, we include their chain name. For example, a single residue contig can be H99, H100A. Remember `p.res_map` maps a residue contig into its internal array index.

Note: Contig syntax cannot handle negative residue index. Residue index should not contain a negative integer. Afpdb still can read such PDB files, but we should renumber the residues (introduced later) if we want to use the more advanced contig syntax.

Single Fragment

```
FRAGMENT = CHAIN[START ID][-][END ID]
```

To specify a continuous range of residues, we specify the start and end residues without the need to enumerate all residues in between. The chain name only occurs once in the beginning. The following are all valid contigs: H1-98, H-98, H98-, H, H98-100A, H98-100B, etc. H stands for the whole chain. H-98 has the start residue ID omitted, which means it starts from the first residue (which is not necessarily residue 1, as the numbering can start, e.g., from 5 in the PDB file fk.pdb). H98- covers 98 to the last residue. H98-100A does not include 100B. H98-100B includes both 100A and 100B. A single residue contig is a special fragment contig.

A chain can contain missing residues, e.g., the PDB structure may contain residues 92, 93, 98, 99, 100A, 100B, and 101, where the integer portion of the residue labels jumps from 93 to 98. We, therefore, assume residues 94-97 are missing. H92-99 will select all residues including 92, 93, 98, and 99, i.e., the contig is not broken simply because there maybe missing residues implied.

Multiple Same-Chain Fragments

To specify multiple fragments within the same chain:

```
IN-CHAIN_FRAGMENTS = FRAGMENT[,[START ID][-][END ID]]*
```

Chain letter should only appear once. E.g., H-5,10-15 covers residues from the beginning to residue 5 on chain H, followed by residue 10 to residue 15 on the same chain.

Note: Fragments may overlap, however, the only the unique residues end up in the residue selection object to be introduced later. We should avoid overlap, as it makes the contig string confusing.

Multiple Fragments

```
CONTIG = IN-CHAIN_FRAGMENTS[:[IN-CHAIN_FRAGMENTS]]*
```

We can specify multiple same-chain fragments by concatenating them with ":". An example contig is H-98:L:P2-5, which specifies the whole L chain, H residues from the beginning to residue ID 98, and includes residues in the P chain from 2 to 5. You can certainly describe same-chain fragments using the general contig syntax, e.g., H-5,10-15:L-10 is equivalent to H-5:H10-15:L-10. In the Demo section, we use "H:L" to specify the whole Ab chains and "P" to select the whole antigen chain. Those are valid contig strings.

If there are multiple fragments within a chain, we advise to keep them together, so that these segments are not separated by segments from other chains for better clarity. As mentioned later, `align()`, and `rmsd()` do not reorder the underlying residue selections in its arguments, as these methods requires the one-to-one residue mapping of the two lists of input residues. `extract()` methods do not reorder residues specified in the contig, however, it requires the all residues of the same chain stay together and ordered ascendingly in the contig string, otherwise, it will generate an error. This is because Afpdb makes a heavy use of a method called `chain_pos`, which assumes all residues colocated on the same chain appear sequentially. Do not worry too much as Afpdb code validates the input contig. Contig strings can be canonicalized, which is described later. When a protein is constructed based on a PDB input, Afpdb makes sure residues are ordered correctly, even if they are ordered incorrected in the source file.

Special Keywords

`None, "ALL"` match all residues. `"", "NONE", and "NULL"` matches no residue.

Note

As mentioned above, there are multiple ways of describing the same set of residues. Contig strings can be canonicalized using the `__str__` method of a residue selection object. `Print(rs)` or `str(rs)` will implicitly call the `__str__` method.

```
p=Protein(fk)
str(p.rs("L6B,5:L-6B"))
```

```
'L5-6B'
```

inplace

Many methods support a boolean argument called `inplace`, this is a practice copied from the well-known Python packaged called Pandas. `inplace=True` will modify the original protein object without allocating new memory, while `inplace=False` will leave the original object unmodified and return the modified structure as a new object.

Developers should stick to to the practice and set False as the default value, whenever `inplace` is used as the argument.

```
p=Protein(fn)
# inplace=False by default
print(p.chain_id())
```

```
['L', 'H', 'P']
```

```
# q has the two chains switched
q=p.extract("H:L")
print(p.chain_id()[:2], q.chain_id(), p==q, "\n")
```

```
['L', 'H'] ['H' 'L'] False
```

```
# p is modified if inplace=True
q=p.extract("H:L", inplace=True)
print(p.chain_id(), q.chain_id(), p==q)
```

```
['H' 'L'] ['H' 'L'] True
```

Clone

If a method does not support `inplace` and you would like to keep the original object unmodified, you can first clone a copy:

```
p=Protein(fn)
print(p.chain_id(), "\n")
```

```
['L', 'H', 'P']
```

```
q=p.clone()
# we modify q in place
q.extract("P", inplace=True)
# q is modified
print(q.chain_id(), "\n")
```

```
['P']
```

```
# p is untouched
print(p.chain_id())
```

```
['L', 'H', 'P']
```

Selection

Selection is a powerful concept in PyMOL and we support both atom selections and residue selections. We consider the concept of selection a major feature missing in Biopython, which led to tedious residue/atom enumerations and made the code less readable and more error prone. Afpdb stores residues as rows and atoms as columns in the first two axes of the NumPy arrays, a residue selection contains the row indices as its data member and an atom selection contains a column index as its data member. Therefore, a residue selection and an atom selection can be used to form a new NumPy array with two axes. Notice the same atom selection is shared by all residues, which greatly simplify the usage and backend implementation. We do not support the selection that combines Ca from residue 1 and C_b from residue 2. We do not see this as a limitation in our practice.

Atom Selection

Afpdb provides a class named `ATS` for an atom selection. `ATS` has an attribute `data`, which is a NumPy array storing the atom column indices.

There are 37 unique atoms across all 20 amino acids. The syntax for users to select atoms is by providing an atom list, or as a comma-separate string. use either `"N,CA,C,O"` or `["N","CA","C","O"]` for backbone atoms. All atom names are in upper case only.

Internally, `atom_positions` and `atom_mask` have 37 elements in their `axis=1` dimension, one per atom type. Therefore, the atom list is represented as a NumPy integer array internally. For `"N,CA,C,O"`, they are represented as `ats.data = np.array([0, 1, 2, 4])`, so `atom_positions[:, ats.data]` will select the atom coordinates for the 4 backbone atoms. This is a key benefit of using NumPy to store PDB structural data.

Note: We often use "ats" for an atom selection object, since the word "as" is a reserved word in python.

```
print(ATS("N,CA,C,O"), "\n")
```

```
N,CA,C,O
```

```
print("ATS data member:", ATS("N,CA,C,O").data, "\n")
```

```
ATS data member: [0 1 2 4]
```

```
# Atoms are reordered in the selection object
print(ATS(["CA","C","N","O"]), "\n")
```

```
N,CA,C,O
```

```
# Atoms are deduplicated in the selection object
print(ATS("CA,CA"), "\n")
```

```
CA
```

```
# special keywords
print("ALL", ATS("ALL"), "\n")
```

```
ALL
N,CA,C,CB,O,CG,CG1,CG2,OG,OG1,SG,CD,CD1,CD2,ND1,ND2,OD1,OD2,SD,CE,CE1,CE2,CE3,NE,NE1,NE2,OE1,OE2,
CH2,NH1,NH2,OH,CZ,CZ2,CZ3,NZ,OXT
```

```
print(None, ATS(None), np.all(ATS(None).data==ATS().data), "\n")
```

```
None
N,CA,C,CB,O,CG,CG1,CG2,OG,OG1,SG,CD,CD1,CD2,ND1,ND2,OD1,OD2,SD,CE,CE1,CE2,CE3,NE,NE1,NE2,OE1,OE2,
CH2,NH1,NH2,OH,CZ,CZ2,CZ3,NZ,OXT True
```

```
print("Empty string, NONE, NULL", ATS("").data, ATS("NONE").data, ATS("NULL").data, "\n")
```

```
Empty string, NONE, NULL [] [] []
```

As shown above, the order of the atoms does not matter, and duplicates are removed and column indices are sorted. We have also special keywords: `None` and `"ALL"` for selecting all atoms, `""`, `"NONE"`, and `"NULL"` for an empty selection. You can also initialize an ATS object with a `set/list/tuple/pandas.Series` of atom strings, a NumPy array, another ATS object. If a single integer is provided, it is treated as one atom index. Basically, ATS supports all sensible input formats.

If a method expects an ATS object as an argument, we can provide the value using any of the format that can be used to initialize an ATS object. For developers, the first thing the method should do is to cast the input argument into an ATS object using the following idiom:

```
ats=ATS(ats)
```

We consider it is a good practice to create an ATS object from a Protein object using `p.ats()`, so that we can avoid explicitly using the ATS class (one less class to memorize):

```
p=Protein()
ats=p.ats("N,CA,C,O")
print(ats, "\n")
```

```
N,CA,C,O
```

```
print(p.ats([0,1,2,3]), "\n")
```

```
N,CA,C,CB
```

```
print(len(ats), "\n")
```

```
4
```

```
# not_full can be empty or less than 37 atoms
print(ats.is_empty(), ats.is_full(), ats.not_full(), "\n")
```

```
False False True
```

```
# method i() converts an atom name into its integer index
print(ATS.i("CA"), ATS.i("CB"), "\n")
```

```
1 3
```

```
# in operator
print("N" in ats, "CB" in ats, 2 in ats, "\n")
```

```
True False True
```

```
print(p.ats_not("N,CA,C,O"), "\n")
```

```
CB,CG,CG1,CG2,OG,OG1,SG,CD,CD1,CD2,ND1,ND2,OD1,OD2,SD,CE,CE1,CE2,CE3,NE,NE1,NE2,OE1,OE2,CH2,NH1,N
H2,OH,CZ,CZ2,CZ3,NZ,OXT
```

```
print(p.ats2str(ATS("N,C,CA,O")), "the same as", str(ATS("N,C,CA,O")), "\n")
```

```
N,CA,C,O the same as N,CA,C,O
```

We also support the set operations &, |, ~ (not), + (the same as |), - (a-b means in a but not in b). We strongly recommend to use these Boolean operators as they make the code easier to read:

```
ats1=ATS("N,CA,C,O")
ats2=ATS("C,O,CB")
print("or:", ats1 | ats2, "\nsame as +:", ats1+ats2, "\n")
```

```
or: N,CA,C,CB,O
same as +: N,CA,C,CB,O
```

```
print("and:", ats1 & ats2, "\n")
```

```
and: C,O
```

```
print("minus:", ats1 - ats2, "\n")
```

```
minus: N,CA
```

```
print("not:", ~ats1, "\nnot not:", ~~ats1, "\n")
```

```
not:
CB,CG,CG1,CG2,OG,OG1,SG,CD,CD1,CD2,ND1,ND2,OD1,OD2,SD,CE,CE1,CE2,CE3,NE,NE1,NE2,OE1,OE2,CH2,NH1,N
H2,OH,CZ,CZ2,CZ3,NZ,OXT
not not: N,CA,C,O
```

Residue Selection

We use a contig string previously described to select residues. To create a residue selection use the `RS` class or preferrably the `p.rs()` method. A residue selection object contains a `data` member, which is literally an integer numpy.ndarray. E.g., rs.data=np.array([0,1,9]).
Therefore, `atom_positions[rs.data][ ATS('N,CA,C,O').data]` will capture the 3D coordinates of the 4 backbone atoms for residue 1st, 2nd, and 10th.

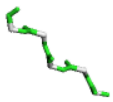
For developers, `atom_positions[rs.data, ATS('N,CA,C,O').data]` does not work, as it pairs the residue id with the corresponding atom id. We could do `atom_positions[np.ix_(rs.data, ATS('N,CA,C,O').data)]` to select the subarray.

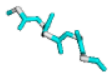
The following is an example where we only extract the backbone atoms from chain H and L and save them into a new PDB file.

```
p=Protein(fn)
p.extract("H-5:L-5", ats="C,CA,N,O", inplace=True)
p.save("bb.pdb")
print("\n".join(util.read_list("bb.pdb")[:15])+"\n...\n")
```

```
HEADER    AFPDB PROTEIN                         10-Jun-24   XXXX
MODEL     1
ATOM      1  N   VAL H   1      32.582   3.997 -22.715  1.00  1.00           N
ATOM      2  CA  VAL H   1      31.113   4.261 -22.594  1.00  1.00           C
ATOM      3  C   VAL H   1      30.727   4.737 -21.206  1.00  1.00           C
ATOM      4  O   VAL H   1      30.948   4.033 -20.247  1.00  1.00           O
ATOM      5  N   GLN H   2      30.105   5.899 -21.101  1.00  1.00           N
ATOM      6  CA  GLN H   2      29.676   6.451 -19.826  1.00  1.00           C
ATOM      7  C   GLN H   2      28.215   6.827 -20.036  1.00  1.00           C
ATOM      8  O   GLN H   2      27.888   7.435 -21.037  1.00  1.00           O
ATOM      9  N   LEU H   3      27.368   6.440 -19.107  1.00  1.00           N
ATOM     10  CA  LEU H   3      25.970   6.871 -19.038  1.00  1.00           C
ATOM     11  C   LEU H   3      25.761   7.794 -17.840  1.00  1.00           C
ATOM     12  O   LEU H   3      25.979   7.398 -16.661  1.00  1.00           O
ATOM     13  N   VAL H   4      25.291   9.008 -18.090  1.00  1.00           N

...
```

```
p.show(style="stick", show_sidechains=True)
```





Because we only have backbone atoms, all residues (except Gly, which fake.pdb does not have) have missing atoms.

```
# no residue has missing backbone atoms
print("Missing backbone:", p.rs_missing_atoms(ats="N,CA,C,O"), "\n")
```

```
Missing backbone:
```

```
# they all have missing side chain atoms, ats defaults to None -- all atoms
print("Missing side chain:", p.rs_missing_atoms(), "\n")
```

```
Missing side chain: H:L
```

Similar to ATS, RS can be initialized by a contig str or special keywords (None, "ALL" for all residues and "", "NONE", "NULL" for an empty selection). RS can also be initialized with a `set/list/tuple/pandas.Series`, a numpy array, another RS object or RL object (explained later). If a single integer is provided, it is treated as one residue index. Basically, RS supports all sensible input formats.

If a method expects an RS object as an argument, we should implement it in a way that users can provide the selection object using any of the data format that can be used to initialize an RS object. E.g., we can write `p.extract("H"), p.extract([0,1,2,3]), p.extract(np.array([2,4,6])), p.extract((2,4,6))`, or `p.extract(p.rs("H"))`, etc. The contig is the most popular format we strongly recommend, as it is the most readable and error-proof. For developers, the first thing the method should do is to cast the input argument into an RS object using the following idiom:

```
rs=RS(self, rs)
```

Unlike ATS, RS requires a Protein object as its first argument in the initialization, as a residue selection will not be meaningful without being tied to a protein structure. For this reason, we often create an RS object from a Protein object using `p.rs()` instead of using `RS(protein_obj, selection_data)`

```
# residue selections
p=Protein(fn)
rs1=RS(p, "H1-3")
rs2=RS(p, "H3-5:L-3")
print("1st rs:", rs1)
```

```
1st rs: H1-3
```

```
print("2nd rs:", rs2, "\n")
```

```
2nd rs: L1-3:H3-5
```

```
# however, we prefer to create an RS object using Protein.rs() method
rs1=p.rs("H1-3")
rs2=p.rs("H3-5:L-3")
```

```
print("1st rs:", rs1)
```

```
1st rs: H1-3
```

```
print("2nd rs:", rs2, "\n")
```

```
2nd rs: L1-3:H3-5
```

```
# Protein also provides set operations on rs objects, however, try not the use these, as the Bool
ean operators (shown in the next cell) make the code easier to read
print("and:", p.rs_and(rs1, rs2), "\n")
```

```
and: H3
```

```
print("or:", p.rs_or(rs1, rs2), "\n")
or: L1-3:H1-5
```

```
# as we not H:L, the only residues left are P chain residues
print("not:", p.rs_not("H:L"), p.data.chain_index[p.rs_not("H:L").data], "\n")
```

```
not: P [2 2 2 2 2 2 2 2 2 2 2 2 2]
```

```
# we can specify the residue universe using full_set, this can reduce the size of rs_not
# in the example below, we restrict the not selection within chain L
print("not with full set:", p.rs_not("L-100", rs_full="L"), "\n")
```

```
not with full set: L101-111
```

```
print("notin:", p.rs_notin(rs1, rs2), "\n")
```

```
notin: H1-2
```

```
print(p.rs2str(p.rs_or(rs1, rs2)))
```

```
L1-3:H1-5
```

In Afpdb, our convention is any method name starts with `rs_` is expected to return an RS object and any argument named `rs` or starts with `rs_` should take any residue selection formats whenever possible. The exception is `rs_seq()` returns a list of RS objects.

Residue selection behaves like a set, where residue indices within `rs.data` are deduplicated and sorted. All sensible set operations apply, therefore, instead of using `p.rs_or`, `p.rs_and`, we prefer to use set operators directly as shown below:

```
p=Protein(fn)
# an unusal selection
rs1=p.rs("H1-3")
rs2=p.rs("H3-5:L-3")
print("and:", rs1 & rs2, "\n")
```

```
and: H3
```

```
print("or:", rs1 | rs2, "same as", rs1+rs2, "\n")
```

```
or: L1-3:H1-5 same as L1-3:H1-5
```

```
print("not:", ~ p.rs("H:L"), "\n")
```

```
not: P
```

```
# ~ cannot take additional argument, so we need to use _not(), if rs_full needs to be specified
print("not with full set:", p.rs("L-100")._not(rs_full="L"), "\n")
```

```
not with full set: L101-111
```

```
print("notin:", rs1-rs2, "\n")
```

```
notin: H1-2
```

```
# inplace operations
rs1+=rs2
print(rs1, "\n")
```

```
L1-3:H1-5
```

```
rs1-=rs2
print(rs1, "\n")
```

```
H1-2
```

```
# more
p=Protein(fn)
print(p.rs("").is_empty(), "\n")
True
```

```
print(p.rs("ALL").is_full(), "\n")
```

```
True
```

```
print(p.rs(None).is_empty(), p.rs(None).is_full(), "\n")
```

```
False True
```

```
print(p.rs("H").is_full(), p.rs("H").is_empty(), "\n")
```

```
False False
```

```
# to combine a list of selections
rs1=p.rs("H1-3")
rs2=p.rs("H3-5:L-3")
rs3=p.rs("H:P")
rs_list=[rs1, rs2, rs3]
print(RS._or(*rs_list), "\n")
```

```
L1-3:H:P
```

```
# AND on the list
print(RS._and(*rs_list), "\n")
```

```
H3
```

```
# only the first element of the list needs to be an RS object, the rest can be any format
# we need to extract the Protein object from the first RS object
print(RS._or(rs1, "H", "P"), "\n")
```

```
H:P
```

```
# residue lookup, convert a residue name to its internal AlphaFold residue order number
print(RS.i("GLY"), RS.i("Gly"), RS.i("G"), "\n")
```

```
7 7 7
```

Residule List

Residue indices within a RS object, `rs.data`, are unique and sorted. We sometimes need duplicate residue indices or require indices to be in specific order. Residule List (RL) class is the parent class of RS, which serves the purpose.

```
p=Protein(fn)
rl=p.rl("H3-5:L-3:H1-3")
print("Not sorted and not unique:", rl.data, "\n")
```

```
WARNING> contig should keep the segments for the same chain together H1-3, if possible!
Not sorted and not unique: [113 114 115   0   1   2 111 112 113]
```

```
# compared to residue list
rs=p.rs("H3-5:L-3:H1-3")
print("Unique and ordered:", rs.data, "\n")
```

```
WARNING> contig should keep the segments for the same chain together H1-3, if possible!
Unique and ordered: [  0   1   2 111 112 113 114 115]
```

```
# cast between RS and RL
# notice we need to provide a protein object as the first arugment, when we use RS or RL to creat
e the object
print(RS(rl.p, rl), "\n")
```

```
L1-3:H1-5
```

```
print("Sorted and unique:", p.rs(rl).data, "\n")
print(p.rl(p.rs("H3-5:L-3:H1-3")).data, "\n")
```

```
Sorted and unique: [  0   1   2 111 112 113 114 115]

WARNING> contig should keep the segments for the same chain together H1-3, if possible!
[  0   1   2 111 112 113 114 115]
```

When printed as a string, RL is shown as a compact contig format.

```
print(p.rl("H3-5:L-3:H1-3"), "\n")
```

```
H3-5:L1-3:H1-3
```

```
print(p.rl("H:H:H"))
```

```
H:H:H
```

Residue list is handy for extracting key annotations for the residues, as demonstrated before:

```
p=Protein(fk)
rl=p.rl("H:L:H")
df=pd.DataFrame({"resi": rl.data})
df['chain']=rl.chain()
df['resn']=rl.name()
df['resni']=rl.namei()
df['aa']=rl.aa()
df['unique_name']=rl.unique_name()
df
```

	resi	chain	resn	resni	aa	unique_name
0	4	H	3	3	L	H3
1	5	H	4	4	V	H4
2	0	L	5	5	E	L5
3	1	L	6A	6	I	L6A
4	2	L	6B	6	V	L6B
5	3	L	10	10	Q	L10
6	4	H	3	3	L	H3
7	5	H	4	4	V	H4

This can be simplified with the `df()` method.

	resi	chain	resn	namei	code	aa	unique_name
0	113	H	3	3		L	H3
1	114	H	4	4		V	H4
2	115	H	5	5		Q	H5
3	0	L	1	1		E	L1
4	1	L	2	2		I	L2
5	2	L	3	3		V	L3
6	111	H	1	1		V	H1
7	112	H	2	2		Q	H2
8	113	H	3	3		L	H3

In the above example, we introduce a few terms:

- resi: the internal integer residue index used as the row index of backend numpy arrays
- chain: the chain name of the residue
- resn/name: the residue index, may include an insertion code, such as "6A" and "6B" in this example.
- resni: the integer part of the residue index without insertion code. This is integer type and can be used for sorting.
- Code: insertion code
- aa: amino acid letter
- unique_name/contig: the unique residue name: {chain}{residue_index}

Later we will see residue list is also used in aligning two structures, as we would like to present the one-to-one mapping between residues.

Boolean operations of RL objects. Residues in a RL object are ordered and can be duplicated, therefore, set operations such as AND/OR are not well defined. Nevertheless, it is convenient to provide some basic operations on RL.

```
rl_h=p.rl("H")
rl_l=p.rl("L")
# + and | both means concatenation of the two selections
print(rl_h + rl_l, "\n")
```

```
H:L
```

```
print(rl_h | rl_l | rl_h, "\n")
```

```
H:L:H
```

```
# to concatenate multiple RL objects
print(RL._or(rl_h, rl_l, rl_h, rl_l), "\n")
```

```
H:L:H:L
```

```
# - mean remove any residues in the first RL that appear in the second RL
print(rl_h + rl_h - p.rl("H3"), "\n")
```

```
H4:H4
```

We may notice the contig representation for two H4 residues is "H4:H4" instead of "H4,4". They are equivalent. In fact, to programmatically construct a contig str for a RL or RS object, is it often convenient to simply ":"-join residues' unique names:

```
rl=p.rl("H3:L6A-10")
print(rl.data)
```

```
[4 1 2 3]
```

```
print(":".join(rl.unique_name()))
```

```
H3:L6A:L6B:L10
```

Casting

Unless we are sure two proteins have the identical sequences, including how their chains are ordered, Boolean operation between RS/RL of two protein objects should be avoided! This is because RS/RL uses the internal indices of residues, these indices lose their meaning when they are used in the context of a different protein.

```
p=Protein(fk)
print(p.chain_id())
```

```
['L', 'H']
```

```
# object q arranges chains in a different order
q=p.extract("H:L")
print(q.chain_id())
```

```
['H', 'L']
```

```
print(p.rs("L").name(), "\n")
```

```
['5', '6A', '6B', '10']
```

```
# the following is wrong, as it use selections of two different objects, the ouput misses residue
L5
print(p.rs("L") & q.rs("L"), "\n")
```

```
L6B-10
```

```
print((p.rs("H") & q.rs("H")).is_empty(), "\n")
```

```
True
```

When we know two proteins share the same sequence(s), we can first align their chains and residues:

```
# first, let's creat two proteins with different chain names and chain orders
p=Protein(fk)
q=p.rename_chains({'H':'A', 'L':'B'})
q=q.extract('A:B')
print(p.seq(), "\n")
```

```
EIVXXXQ:LV
```

```
print(q.seq(), "\n")
```

```
LV:EIVXXXQ
```

```
# to align these two proteins, we rename the chains in q and reorder them according to object p
q=q.rename_reorder_chains(p, {'A':'H', 'B':'L'})
print(q.seq())
```

```
EIVXXXQ:LV
```

However, the recommended approach to transfer a selection to another object is to cast, which internally uses contig strings to make sure the "meaning" of the selection is transferred.

```
p=Protein(fk)
print(p.chain_id())
```

```
['L', 'H']
```

```
# object q arranges chains in a different order
q=p.extract("H:L")
print(q.chain_id())
```

```
['H' 'L']
```

```
rs_a=p.rs("L")
# L chain are in the first 4 positions of object p
print(rs_a.data, "\n")
```

```
[0 1 2 3]
```

```
# once casted into object q, the "L" selection of object q are the last 4 positions
rs_b=rs_a.cast(q)
# rs_b now belongs to object q
print(rs_b.p == q, "\n")
```

```
True
```

```
print(rs_b.data, "\n")
```

```
[2 3 4 5]
```

```
print(rs_b.name())
```

```
['5', '6A', '6B', '10']
```

Read/Write

One of our philosophies in designing Afpdb is to reduce the number of methods and arguments to bare minimum to reduce the burden of memorization. The convenience of Afpdb can be demonstrated by how we read/write protein structures. Regardless of what format a protein structure is represented, we use `Protein()` to create and use `save()` to export.

```
p=Protein(fn)
print(p.chain_id())
```

```
['L', 'H', 'P']
```

```
p.save("test.pdb")
out=util.unix("ls -l test.pdb")
```

```
-rw-r--r--  1 zhoyyi1  staff  155520 Jun 10 19:40 test.pdb
```

The constructor can optionally take a contig string if you want to only read in a subset of the structure:

```
p=Protein(fn, contig="H:L")
print(p.chain_id(), "\n")
```

```
['H' 'L']
```

```
p=Protein(fn, contig="H-5:L-10")
print(p.seq(), "\n")
```

```
VQLVQ:EIVLTQSPGT
```

```
# however, for clarity, we recommend to separate the two operations, use contig in the extract()
method explicitly.
p=Protein(fn)
print(p.extract("H:L").chain_id(), "\n")
```

```
['H' 'L']
```

```
p=Protein(fn)
print(p.extract("H-5:L-10").seq(), "\n")
```

```
VQLVQ:EIVLTQSPGT
```

The first argument in `Protein()` constructor can be quite flexible. Just be aware that only the first model is read, if the PDB file contains multiple models. AlphaFold was not designed to handle multiple conformations, afpdb can be made much more efficient by assuming only one conformation is being manipulated.

Below are a few examples:

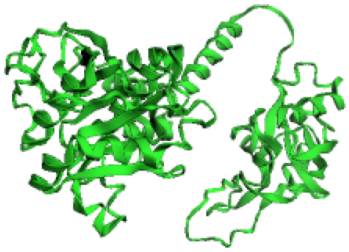
```
print("Load a local PDB file")
# support both .pdb and .ent extensions
print(fn)
p=Protein(fn)
print(p.chain_id(), "\n")
```

```
Load a local PDB file
/Users/zhoyyi1/afpdb/tutorial/example_files/5cil.pdb
['L', 'H', 'P']
```

```
print("Create Protein from another afpdb.Protein object")
q=Protein(p)
print(q.chain_id(), "\n")
```

```
Create Protein from another afpdb.Protein object
```

```
['L' 'H' 'P']

print("Create Protein from an afpdb.Protein.data object")
q=Protein(p.data)
print(q.chain_id(), "\n")

Create Protein from an afpdb.Protein.data object
['L' 'H' 'P']

print("Create Protein from a BioPython Structure object")
b=p.to_biopython()
print(type(b))
q=Protein(b)
print(q.chain_id(), "\n")

Create Protein from a BioPython Structure object
<class 'Bio.PDB.Structure.Structure'>
['L', 'H', 'P']

print("Create Protein from a str containing the content of a PDB file")
s=p.to_pdb_str()
print("\n".join(s.split("\n")[:5])+"\n...")
q=Protein(s)
print(q.chain_id())

Create Protein from a str containing the content of a PDB file
HEADER    AFPDB PROTEIN                           10-Jun-24   XXXX
MODEL     1
ATOM      1  N   GLU L   1       5.195 -14.817 -19.187  1.00  1.00           N
ATOM      2  CA  GLU L   1       6.302 -14.276 -18.361  1.00  1.00           C
ATOM      3  C   GLU L   1       7.148 -15.388 -17.731  1.00  1.00           C
...
['L', 'H', 'P']

# more examples ...
print("Save as .cif, save Protein into a .cif file\n")
p=Protein(fn)
p.save("test.cif")

Save as .cif, save Protein into a .cif file

print("Create Protein from a local .cif file")
p=Protein("test.cif")
print(p.chain_id(), "\n")

Create Protein from a local .cif file
['L' 'H' 'P']

print("Create with a PDB 4-letter code, fetch structure from PDB online.")
p=Protein("1crn")
print(p.seq(), "\n")

Create with a PDB 4-letter code, fetch structure from PDB online.
--2024-06-10 19:40:31--  https://files.rcsb.org/download/1crn.pdb1.gz
TTCCPSIVARSNFNVCRLPGTPEAICATYTGCIIIPGATCPGDYAN

print("Create with a EMBL AlphaFold model code, fetch structure from EMBL online.")
p=Protein("Q2M403")
print(p.seq())
p.show()

Create with a EMBL AlphaFold model code, fetch structure from EMBL online.
Saving to: 'AF-Q2M403-F1-model_v3.pdb'

MVVVSLQCAIVGQAGSSFDVEIDDGAKVSKLKDAIKAKNATTITGDAKDLQLFLAKQPVEDESGKEVVPVYRPSAEEMKEESFKWLPDEHRAALKLV
EGESDDYIHALTAGEPILGSKTLTTWFYTKNNMELPSSEQIHVLVVVPDGAGGSASDTSRMDRLFDKVDKVYEHTVLSKRTRYVHSEMNSAKGNILL
NDLKIRISPVDTVKFAGGVPTPAKEFKWKSDRTEEQQKEPYREYVVANIGDVLTNNKLCVVGVEKGANILTVEVPGRDIVLAGRTDMIVLSDIAQKF
PHYLPHLPGVRMLIEVKKVVTTASEFQALSELIALDIIVTESVMALLTNLTNHWQFFWVSRKSDDRVIIETTTLIAPGEAFAVIRTLLDQSPSAGAE
VSLPCFEKPVKRQKLSQLLPSISEASGSSGIRESIERYYDIASMLGPDLEMARAVASQVARSIPTLSYF
```



Sequence & Chain

Extraction

```
p=Protein(fn)
p.seq()

'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSL
STFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLE
LNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS:NWFDITNWLWYIK'
```

Notice the sequences from different chains are concatenated by colon (color in red) into one line:

```
EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLS
TFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLEL
NSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS:NWFDITNWLWYIK
```

This is a standard format used to provide multimer sequence inputs to ColabFold and AlphaFold for protein structure prediction.

Here the order of the chain sequences is the same as how they occur in the PDB file. However, it is always safer to assume that chain order may change when we convert objects among different formats. Therefore, we strongly recommend users tie chain names with their sequences by using `seq_dict()` method instead:

```
p.seq_dict()

{'L':
'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSL
STFGQGTKVEVKRTV',
 'H':
'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCA
REGTTGDGDLGKPIGAFAHWGQGTLVTVSS',
 'P': 'NWFDITNWLWYIK'}
```

Missing Residues

When there are missing residues in the structure, they will be represented as "X" in the sequence string:

```
p=Protein(fn)
q=p.extract("H1-10:H21-30")
print(len(q.data.residue_index), q.data.residue_index, "\n")

20 ['1' '2' '3' '4' '5' '6' '7' '8' '9' '10' '21' '22' '23' '24' '25' '26'
 '27' '28' '29' '30']

print("Number of residues in numpy arrays:", q.data.atom_positions.shape, "\n")
print(q.seq(), "\n")

Number of residues in numpy arrays: (20, 37, 3)
VQLVQSGAEVXXXXXXXXXXCKASGGSFST

print(q.seq_dict(gap="."), "\n")

{'H': 'VQLVQSGAEV..........CKASGGSFST'}

print(q.seq().replace("X", "G"), "\n")

VQLVQSGAEVGGGGGGGGGGCKASGGSFST

print(q.seq(gap="G"), "\n")
```

```
VQLVQSGAEVGGGGGGGGGGCKASGGSFST
```

The number of missing residues is determined based on residue numbering. Users shoudl be aware that the SEQRES section in the PDB file is not used to determine missing residues. In the above example, we first create a new object by taking the first ten residues from chain H, skipping 10 residues, then take another 10 residues. In the resultant object q, the 10-residue gap is preserved (residue index jumps from 10 to 21). Although the 10 missing residues have no corresponding elements in the internal NumPy array representation or in the saved PDB file (the array only contains 20 residues in total, the discontinuity in the residue index allows us to figure out that 10 residues are missing).
Therefore, `seq()` and `seq_dict()` outputs ten "X". You can change the display of "X" by the gap argument or suppress gap by "". For protein structure prediction, we often replace the missing residues by Glycine "G". If somehow you want to ignore missing residues, set `gap=''`.

```
p=Protein(fk)
print(p.data.residue_index, "\n")
```

```
['5' '6A' '6B' '10' '3' '4']
```

```
print(p.seq_dict(), "\n")
```

```
{'L': 'EIVXXXQ', 'H': 'LV'}
```

```
print(p.seq_dict(gap=""))
```

```
{'L': 'EIVQ', 'H': 'LV'}
```

When we determine missing residues, only the integer part of the residue ID is considered. In the above example, residue index 6 appears twice as 6A and 6B, followed by residue 10. We assume residues 7, 8, and 9 are missing.

Note: Missing residues at the N or C terminals cannot be recovered. In addition, if the residue numbering in the full structure is not numerically continuous, e.g., both residues "6A" and "6B" were removed, we can only infer that one residue "6" is missing:

```
p=Protein(fk)
q=p.extract("L5,10:H")
print(q.data.residue_index, "\n")
```

```
['5' '10' '3' '4']
```

```
print(q.seq(), "\n")
```

```
EXXXXQ:LV
```

Length

The get the total number of residues (not counting missing residues), i.e, the number of rows in the backend numpy arrays, use:

```
p=Protein(fn)
print("Total residue counts:", len(p), "\n")
```

```
Total residue counts: 250
```

```
print("Residue counts per chains:", p.len_dict())
```

```
Residue counts per chains: {'L': 111, 'H': 126, 'P': 13}
```

```
q=Protein(fk)
# missing residues are not counted as part of the length
print(q.seq(), "\n")
```

```
EIVXXXQ:LV
```

```
print(q.len_dict(), "\n")
```

```
{'L': 4, 'H': 2}
```

Search

When we have a sequence fragment and would like to locate where it came from, use:

```
p=Protein(fn)
out=p.rs_seq('LTI')
for i,m in enumerate(out):
    print(f"Matched #{i+1}, location: {m}, sequence: {m.seq()}")
```

```
Matched #1, location: L74-76, sequence: LTI
Matched #2, location: H54-56, sequence: LTI
```

The name `rs_seq()` suggests the method performs sequence search and return residue selection (RS) objects. Since there can be multiple matches, it returns a list of RS objects. We will explain residue selection objects later, for now, we just need to know that a residue selection object can be printed as a contig string and its `seq()` method returns the corresponding sequence.

The two matches are in bold below:

```
EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFT**LTI**SRLEPEDFAVYYCQQYGQSLS
TFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPL**LTI**TNYAPRFQGRITITADRSTSTAYLEL
NSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS: NWFDITNWLWYIK
```

If we are only interested in sequence search within a list of chains, provide the chain name(s) list:

```
out=p.rs_seq('LTI', in_chains="H")
print(len(out), out[0], "\n")
```

```
1 H54-56
```

The sequence search pattern can be a regular expression, e.g., to match all charged residues:

```
out=p.rs_seq(r'[RHKDE]')
# concatenate the list of matched residue selection objects into one RS object, RS._or() will be
explained later.
m=RS._or(*out)
print(m)
```

```
L1,17-18,24,33,40,46,55,61-62,71,78,80,82-83,104,106,108-109:H9,11-12,22,37,42,45,62,66,72-
73,81,86,88-89,97-98,103,105,108,115:P4,13
```

```
print(m.seq())
```

```
EERRKRRRDRDREEDKEKREKRKRRERRDREREDREDDKHDK
```

Mutagenesis

To perform mutagenesis on a structure, we thread a new sequence onto an existing Protein object. As this method requires the installation of PyMOL, it throws an exception if PyMOL is not found.

When we specify the new sequence, we only specify the new residues for which there is coordinate information, i.e., missing residues should not occur in the new sequence. Chain L's sequence was EIVXXXQ, the target sequence is AIVD instead of AIVXXXD.

The following example requires PyMOL to be installed. The installation can take some time in Google Colab.

```
install_pymol()
p=Protein(fk)
print(p.seq(), "\n")

EIVXXXQ:LV

p.thread_sequence({"L":"AIVD", "H":"LG"}, "m.pdb")

###JSON STARTS
{"output_pdb": "m.pdb", "ok": true, "output_equal_target": false, "input": {"L": "EIVQ", "H":
"LV"}, "output": {"H": "LG", "L": "AIVXXXD"}, "target": {"L": "AIVD", "H": "LG"}, "relax": false,
"residues_with_missing_atom": "", "mutations": [["L", 5, "GLU", "ALA"], ["L", 10, "GLN", "ASP"],
["H", 4, "VAL", "GLY"]]}
###JSON END
{'output_pdb': 'm.pdb',
 'ok': True,
 'output_equal_target': False,
 'input': {'L': 'EIVQ', 'H': 'LV'},
 'output': {'H': 'LG', 'L': 'AIVXXXD'},
 'target': {'L': 'AIVD', 'H': 'LG'},
 'relax': False,
 'residues_with_missing_atom': '',
 'mutations': [('L', 5, 'GLU', 'ALA'),
  ('L', 10, 'GLN', 'ASP'),
  ('H', 4, 'VAL', 'GLY')]}
```

From the output JSON data, we see L5 was "E" and it is replaced with "A". L6A-L6B was not changed, therefore, their side chain coordinates remain the same. L10 is replaced by "D". On the chain H, H6 was "G" and is replaced by "V".

The new mutated residues have their side chain torsion angles generated by PyMOL. Therefore, it generally makes sense to relax the side chain, for example, with Amber. Relax is not supported by Afpdb due to its dependency on Amber. TODO: we might investigate this more.

Mutation

If a chain name is in two proteins and the chain has the same number of residues, we can identify mutated residues by a residue-by-residue comparison. The example below identifies three mutated residues:

```
p=Protein(fk)
# m.pdb was produced with the previous thread_seq() example
q=Protein("m.pdb")
print(p.seq(), q.seq(), "\n")

EIVXXXQ:LV LG:AIVXXXD

print(p.rs_mutate(q))

L5,10:H4
```

Chain Name and Order

In the fn PDB file, chains go from L, to H, to P. chain_id() will return chain names in the same order. `p.data.chain_index` stores the chain index. Afpdb expects all residues belong to the same chain are stored together without being interrupted by another chain.

```
p=Protein(fn)
print(p.chain_id(), "\n")
```

```
['L', 'H', 'P']
```

```
print(p.data.chain_index, "\n")
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

Although we should not rely on the exact order of chains in the PDB file, we can nevertheless reorder chains, if we wish:

```
p=Protein(fn)
q=p.data
print("Original chain order:", p.chain_id(), "\n")
```

```
Original chain order: ['L', 'H', 'P']
```

```
print(p.seq(), "\n")
```

```
EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLS
TFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLEL
NSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS:NWFDITNWLWYIK
```

```
# notice the last 20 residues belong to chain L, then chain P in the q.chain_index NumPy array
print(q.chain_index[-20:], "\n")
```

```
[1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

```
print(q.residue_index[-20:], "\n")
```

```
['120' '121' '122' '123' '124' '125' '126' '1' '2' '3' '4' '5' '6' '7' '8'
 '9' '10' '11' '12' '13']
```

```
p.reorder_chains(["P","L","H"])
q=p.data
print("After chain reorder:", p.chain_id(), "\n")
```

```
After chain reorder: ['P', 'L', 'H']
```

```
print(p.seq(), "\n")
```

```
NWFDITNWLWYIK:EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPED
FAVYYCQQYGQSLSTFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRIT
ITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS
```

```
# after ordering, notice the last 20 residues belong to chain H in the q.chain_index NumPy array
print(q.chain_index[-20:], "\n")
```

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

```
print(q.residue_index[-20:], "\n")
```

```
['107' '108' '109' '110' '111' '112' '113' '114' '115' '116' '117' '118'
 '119' '120' '121' '122' '123' '124' '125' '126']

# The method reorder_chains() is no longer needed, as we can use extract():
p=Protein(fn)
p.extract("P:L:H", inplace=True)
q=p.data
print("After chain reorder:", p.chain_id(), "\n")

After chain reorder: ['P' 'L' 'H']

print(p.seq(), "\n")

NWFDITNWLWYIK:EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPED
FAVYYCQQYGQSLSTFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRIT
ITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS

print(q.chain_index[-20:], "\n")

[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]

print(q.residue_index[-20:], "\n")

['107' '108' '109' '110' '111' '112' '113' '114' '115' '116' '117' '118'
 '119' '120' '121' '122' '123' '124' '125' '126']
```

Notice after reordering, the `chain_index` array goes from 0 to 2, and the chain_id list has also been updated.

The extract() method will error if the selection list contains duplicates or residues of the same chain are not ordered.

```
p=Protein(fn)
try:
    p.extract("H:H:L")
except Exception as e:
    print(e)

There are duplicate residues: [111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
127 128
 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146
 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164
 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182
 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218
 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236]

try:
    p.extract("H5-:H1")
except Exception as e:
    print(e)

Residues within chain H are not sorted in order.
WARNING> contig should keep the segments for the same chain together H2, if possible!

try:
    p.extract("H1:L:H2")
except Exception as e:
    print(e)

Residues for chain ['H'] are not grouped together.
Treat as residue selection ...
WARNING> contig should keep the segments for the same chain together H2, if possible!

# as_rl=False, all examples are fine, b/c the contig selection will be cast into an unordered/non
-redundant residue selection
print("Treat as residue selection ...\n")
```

```
q=p.extract("H:H:L", as_rl=False)
q=p.extract("H5-:H1", as_rl=False)
q=p.extract("H1:L:H2", as_rl=False)
```

```
Treat as residue selection ...
```

As a shortcut, we can rename and reorder chains in one method. In the example below, the original chain names are L:H:P, AlphaFold's corresponding modeling output has the chain named as A:B:C. We can prepare both objects as chain H:L:P with the following:

```
p=Protein(fn)
int(p.chain_id())
```

```
['L', 'H', 'P']
```

```
print(q.chain_id())
```

```
['A', 'B', 'C']
```

```
p=p.extract("H:L:P")
q.rename_reorder_chains(p, {"A":"L", "B":"H", "C":"P"}, inplace=True)
print(p.seq() == q.seq())
```

```
True
```

```
# although we don't recommend this, since two objects have exact the same sequence, RS/RL objects
can be used across
print(p.rs("H35-50") & q.rs("H40-45"), "\n")
```

```
H40-45
```

```
# the recommended way is to use cast
print(p.rs("H35-50") & q.rs("H40-45").cast(p))
```

```
H40-45
```

Residue Labeling

The method `chain_pos()` is an important utility method. It returns the start and end NumPy indices of each chain, which can be used to access the corresponding numpy arrays. For our example structure, the complex consists of 3 chains. The array residue_index has 250 elements. The residues in chain L are stored as the first 111 elements in the numpy arrays, in Python, the indices are from 0 to 110.

Many methods relies on `chain_pos` to accelerate the computation, this is the reason why Afpdb assumes all residues of the same chain should be stored together within the backend numpy arrays. If they are not in the PDB file, their order will be fixed by Apfdb during the protein object creation.

```
p=Protein(fn)
print(p.chain_pos(), "\n")
```

```
{'L': [0, 110], 'H': [111, 236], 'P': [237, 249]}
```

```
q=p.data
print(q.residue_index[108:114], "\n")
```

```
['109' '110' '111' '1' '2' '3']
```

```
print(q.chain_index[108:114])
```

```
[0 0 0 1 1 1]
```

```
# we should avoid using p.data, all can be done by residue selection
rs=p.rs(list(range(108, 115)))
print(rs.name(), "\n")
```

```
['109', '110', '111', '1', '2', '3', '4']
```

```
print(rs.chain())
```

```
['L', 'L', 'L', 'H', 'H', 'H', 'H']
```

The chain_index for element 110 is the last residue of chain L, its residue name is '111' and chain id is 0, i.e., resn="111", resi=110, resn_i=111. Element 111 is the first residue of chain H. Its residue name is '1' and chain id is 1, i.e., resn="1", resi=111, resn_i=1.

If we only want to extract the coordinates for the 13 residues of chain P, we will use:

```
q.atom_positions[237:250].shape
```

```
(13, 37, 3)
```

Residue Renumbering

Residue index can be modified. There are a number of renumbering modes:

```
    None: keep original numbering unmodified.
    RESTART: 1, 2, 3, ... for each chain
    CONTINUE: 1 ... 100, 101 ... 140, 141 ... 245
    GAP200: 1 ... 100, 301 ... 340, 541 ... 645 # mimic AlphaFold gaps
    You can define your own gap by replacing GAP200 with GAP{number}, e.g., GAP10
    NOCODE: remove insertion code, residue 6A and 6B becomes 6, 7
```

```
p=Protein(fk)
q=p.data
print(q.chain_index, "\n")
```

```
[0 0 0 0 1 1]
```

```
p.renumber(None, inplace=True)
print(q.residue_index, "\n")
```

```
['5' '6A' '6B' '10' '3' '4']
```

```
p.renumber('RESTART', inplace=True)
print(q.residue_index, "\n")
```

```
['1' '2A' '2B' '6' '1' '2']
```

```
p.renumber('CONTINUE', inplace=True)
print(q.residue_index, "\n")
```

```
['1' '2A' '2B' '6' '7' '8']
```

```
p.renumber('GAP200', inplace=True)
print(q.residue_index, "\n")
```

```
['1' '2A' '2B' '6' '207' '208']
```

```
p.renumber('GAP10', inplace=True)
print(q.residue_index, "\n")
```

```
['1' '2A' '2B' '6' '17' '18']
```

```
p.renumber('NOCODE', inplace=True)
print(q.residue_index)
```

```
['1' '2' '3' '7' '8' '9']
```

NOCODE is used to remove insertion code, in case we need to feed a structure into a tool that cannot deal with insertion code. Notice "2A" and "2B" are renamed to "2" and "3" to avoid the collision, then residue "6" is renamed to "7" to preserve the three missing residues in the original naming.

By default, `renumber()` returns a tuple `(p_object, old_number)`. `inplace=False` is the default, i.e., the input object is not modified.

If we need a custom naming, e.g., we really want to remove the gaps for missing residues, we can provide our own custom list of residue names, using `resn()`. Without any argument, `resn()` is a read method that returns the current residue names. If providing an argument as a list or NumPy array, its length should match the number of residues.

```
p=Protein(fk)
q, old_resn=p.renumber("RESTART")
print("Old Name:", old_resn, "New Name:", q.resn(), "\n")
```

```
Old Name: ['5' '6A' '6B' '10' '3' '4'] New Name: ['1' '2A' '2B' '6' '1' '2']
```

```
# restore the old residue names
q.resn(old_resn)
print(q.resn(), "\n")
```

```
['5' '6A' '6B' '10' '3' '4']
```

```
# set custom residue names, we create new residue names as a continous integer without gap
q.resn([str(x) for x in range(1, len(q)+1)])
print(q.resn())
```

```
['1' '2' '3' '4' '5' '6']
```

Be aware that `resn()` modifies the object in place. If we only want to rename a few residues, we can use a RL object to specify the target residues.

```
p=Protein(fk)
print(p.resn())
```

```
['5' '6A' '6B' '10' '3' '4']
```

```
p.resn(["6", "7"], "L6A-6B")
print(p.resn())
```

```
['5' '6' '7' '10' '3' '4']
```

Merge & Split Chains

Before AlphaFold Multimer was created, we had to use AlphaFold monomer model to predict multimer structures. The hack is to merge chains into one single chain, where we introduce a 200 gap in the residue indices between the original chain junctions to keep AlphaFold from treating the previous C- and the next N-terminals of two contiguous chains from being mistakenly treated as neighboring residues. This could be achieved by:

```
p=Protein(fn)
q, c_pos=p.merge_chains(gap=200, inplace=False)
print(q.seq_dict(gap="."), "\n")
```

```
{'L':'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQ
YGQSLSTFGQGTKVEVKRTV.....................................................................................
```

```
..................................................................................................
.........................VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITA
DRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS.........................................
..................................................................................................
.....................................................NWFDITNWLWYIK'}

print(c_pos, "\n")

{'L': [0, 110], 'H': [111, 236], 'P': [237, 249]}

r=q.split_chains(c_pos, inplace=False)
print(r.seq_dict())

{'L':'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQ
YGQSLSTFGQGTKVEVKRTV',
'H':'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAV
YYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS',
'P': 'NWFDITNWLWYIK'}
```

As chains got merged into a single change, 200-residue gaps were introduced and shown as "." in the sequence output. We notice there is only one chain, where the first chain name in the original PDB file was used. The method returns the original `c_pos` object, which memorizes the chain structure before the merge.

After AlphaFold predicts the structure as a monomer, the output PDB file can then be used by split_chains to restore the original trimer structure. Here `c_pos` guides the method to split the monomer chain and restores the naming of the original chains.

If we send one sequence to AF without residue indices, we would actually need to replace the gap with glycines. The predicted structure will contain extra glycine linkers, those need to be removed before we can `split_chains()`. The extra glycines can be identified with `q.rsi_missing()` or `pred.rs_insertion()`, as demonstrated later in the first AI use case. The idea is outlined below:

```
p=Protein(fn)
# for the sake of saving paper space, we use a gap of 20 residues
q, c_pos=p.merge_chains(gap=20, inplace=False)
seq=q.seq(gap="G")
# obtain the positions for the glycine linkers
linker_pos=q.rsi_missing()
print("Chain breaks:", c_pos, "\n")

Chain breaks: {'L': [0, 110], 'H': [111, 236], 'P': [237, 249]}

print("Position of G:", linker_pos, "\n")

Position of G: [111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128
 129 130 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
 273 274 275 276]

print("Send the following fasta sequence to AlphaFold monomer model for prediction:\n")

Send the following fasta sequence to AlphaFold monomer model for prediction:

print(">af_pred\n"+seq, "\n")

>af_pred
EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLS
TFGQGTKVEVKRTVGGGGGGGGGGGGGGGGGGGGVQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRF
QGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSSGGGGGGGGGGGGGGGGGGGGNWFDITNWLWYIK

print("AF prediction, generate pred.pdb ...\n")
print("Read the predicted structure:")
print("    pred=Protein('pred.pdb')\n")
print("Remove glycine linkers:")
print("    pred.extract(~ RS(pred, linker_pos), inplace=True)\n")
```

```
print("Restore three chains:")
print("    pred.split_chains(c_pos)\n")

AF prediction, generate pred.pdb ...
Read the predicted structure:
    pred=Protein('pred.pdb')
Remove glycine linkers:
    pred.extract(~ RS(pred, linker_pos), inplace=True)
Restore three chains:
    pred.split_chains(c_pos)
```

The hack for merging chains is no longer needed with AlphaFold Multimer, this is required for using ESMFold web service. Also replacing missing residues within a chain by Glycine is a common practice.

Geometry, Measurement, and Visualization

Select Neighboring Residues

For a given residue selection, `rs_around` selects neighboring residues, where their closest distance is within the specified `dist` argument. In the example below, we find all residues that are within 3.5Å around P chain residues. AlphaFold's NumPy arrays do not contain hydrogens. We compute distances among all possible atom pairs and use the minimum atom-atom distance as the final distance between two residues. In this example, it identifies 5 residues on chain L and 8 residues on chain H. The method does not consider intra-selection distances, i.e., it will not include residues that are already part of the seed selection. If you want to include seed residues, just use "or" operator to combine two residue selections.

There are three returned values, the first is a new residue selection of the neighboring residues and second is a residue selection of the seed residues (i.e., source residues that have a neighbor), the third is a DataFrame object. The DataFrame object provides the details about all residue pairs, the source is marked as "a" (chain P here) and the target is marked as "b" (chain H or L). `resn_` stands for residue name, `resn_i_` is for the integer part of the residue names without the insertion code and it is an integer type. `resi_` is for the internal Numpy array index (i.e., residue selection `rs.data`). The shortest distance between two residues is listed under the `dist` column and the atom pair that contributes to this shortest distance is listed as `atom_a` and `atom_b`. The DataFrame is sorted by descending `dist` values.

```
p=Protein(fn)
rs_nbr, rs_seed, t=p.rs_around("P", dist=3.5)
print("Neighbors:", rs_nbr.name(), "\n")
```

```
Neighbors: ['33', '92', '93', '94', '95', '30', '56', '58', '98', '106', '107', '108', '109']
```

```
print("Seeds:", rs_seed.name(), "\n")
```

```
Seeds: ['1', '2', '3', '4', '6', '7', '10']
```

```
t
```

	chain_a	resi_a	resn_a	resn_i_a	atom_a	chain_b	resi_b	resn_b	resn_i_b	atom_b	dist
23	P	242	6	6	OG1	H	208	98	98	OE2	2.636247
17	P	240	4	4	OD1	L	32	33	33	NZ	2.814822
7	P	238	2	2	N	L	94	95	95	OG	2.911941
2	P	237	1	1	ND2	L	91	92	92	O	2.929501
27	P	243	7	7	ND2	H	218	108	108	CE	3.038573
4	P	237	1	1	OD1	L	93	94	94	C	3.087493
14	P	238	2	2	NE1	H	166	56	56	O	3.230981
1	P	237	1	1	ND2	L	92	93	93	C	3.257804
31	P	246	10	10	NE1	H	216	106	106	O	3.305220
13	P	238	2	2	CE3	H	168	58	58	ND2	3.320923
5	P	237	1	1	OD1	L	94	95	95	N	3.325511
22	P	242	6	6	CG2	H	140	30	30	O	3.370509
16	P	239	3	3	CE2	L	94	95	95	O	3.383096
30	P	246	10	10	NE1	H	217	107	107	C	3.417503
15	P	239	3	3	CD2	L	93	94	94	O	3.472714
26	P	243	7	7	N	H	219	109	109	CG	3.474068
18	P	242	6	6	C	H	219	109	109	CG	3.495887

To include the seed residues in the selection, use the following idiom:

```
# p.rs(t.resi_a) select all seed residues appears in the neighbor table
# resi_a contains the residue IDs for those on the P chain that interact with rs_nbr
rs_both = rs_nbr | rs_seed
print(rs_both)
```

```
L33,92-95:H30,56,58,98,106-109:P1-4,6-7,10
```

```
The distance table contains all residues pairs between seeds and their neighbors. If we only want
to see one seed per neighbor, we define drop_duplicates = True:
```

```
rs_nbr, rs_seed, t2=p.rs_around("P", dist=3.5, drop_duplicates=True)
print(rs_seed, "\n")
```

```
P1-4,6-7,10
```

```
print(rs_nbr)
t2
```

```
L33,92-95:H30,56,58,98,106-109
```

By default, the DataFrame object only keep one atom pair per residue pair. If we want to access all atom pairs within the specified distance, use `keep_atoms = True`.

The reason we provide `resn_i` is to enable us to filter the output DataFrame. The data type for `resn` is a string, as `resn` could contain insertion codes. If we are interested in rows corresponding to residues within H95-106, we cannot use the `resn` column for filtering. Instead, we can accomplish this with the `resn_i` column:

```
print("resn cannot be used for filtering, as it's string type, str greater than '95' will be all
less than '116'")
t2=t[(t.chain_b=="H")&(t.resn_b>="95")&(t.resn_b<="106")]
t2
```

```
resn cannot be used for filtering, as it's string type, str greater than '95' will be all less
than '116'
```

chain_a	resi_a	resn_a	resn_i_a	atom_a	chain_b	resi_b	resn_b	resn_i_b	atom_b	dist

```
print("resn_i is an integer column")
t2=t[(t.chain_b=="H")&(t.resn_i_b>=95)&(t.resn_i_b<=106)]
t2
```

```
resn_i is an integer column
```

	chain_a	resi_a	resn_a	resn_i_a	atom_a	chain_b	resi_b	resn_b	resn_i_b	atom_b	dist
23	P	242	6	6	OG1	H	208	98	98	OE2	2.636247
31	P	246	10	10	NE1	H	216	106	106	O	3.305220

By default, all residue atoms are considered. However, we can restrict the distance measurement using an atom selection. We can also restrict the target residues using a residue selection `rs_within`.

```
p=Protein(fn)
rs=p.rs("P")
# select residues on chain H that have CA-CA distance within 8A.
rs_nbr, rs_seed, t=p.rs_around(rs, dist=8, rs_within="H", ats="CA")
print(rs_nbr)
```

```
H30-32,51,58,108-111
```

```
t
```

	chain_a	resi_a	resn_a	resn_i_a	atom_a	chain_b	resi_b	resn_b	resn_i_b	atom_b	dist
2	P	239	3	3	CA	H	221	111	111	CA	5.573195
10	P	243	7	7	CA	H	219	109	109	CA	5.704927
4	P	242	6	6	CA	H	141	31	31	CA	5.911184
7	P	242	6	6	CA	H	140	30	30	CA	6.083017
13	P	246	10	10	CA	H	218	108	108	CA	6.221307
9	P	243	7	7	CA	H	218	108	108	CA	6.447024
6	P	242	6	6	CA	H	219	109	109	CA	6.487135
5	P	242	6	6	CA	H	142	32	32	CA	6.735140
12	P	245	9	9	CA	H	140	30	30	CA	6.884907
0	P	238	2	2	CA	H	168	58	58	CA	7.062018
3	P	240	4	4	CA	H	221	111	111	CA	7.073796
14	P	246	10	10	CA	H	219	109	109	CA	7.393139
15	P	246	10	10	CA	H	140	30	30	CA	7.531632
8	P	242	6	6	CA	H	161	51	51	CA	7.587631
1	P	239	3	3	CA	H	142	32	32	CA	7.691276
11	P	243	7	7	CA	H	220	110	110	CA	7.696134
16	P	247	11	11	CA	H	218	108	108	CA	7.910936

Display

To visualize a structure, please `save` it to a PDB file and use PyMOL. If you prefer to display the structure in an HTML file, use `util.save_string("my_protein.html", p.html())` and open it in a browser.

Within Jupyter Notebook, `p.show()` will display the structure as we do throughout this notebook.

The arguments for `show()` and html() are the same. `show_sidechains=False, show_mainchains=False, color="chain", style="cartoon", width=320, height=320)`

color can be: `"chain", "lDDT", "b", "spectrum", "ss"`. style can be: `"cartoon", "stick", "line", "sphere", "cross"`.

If colored by `lDDT`, the b-factors should be in the range of [0, 100]. If colored by `"b"`, b-factors should be normalized to [0, 1].

```
p=Protein(fn)
util.save_string("my_protein.html", p.html())
print("You can also open my_protein.html in a browser to view the protein structure.")
print("We display the protein within Jupyter notebook below")
p.show(style="cartoon", color="ss")
```

B-factors

B-factor is an embedded attribute that can be repurposed to store other computation results. E.g., AlphaFold uses b-factors to store pLDDT for the structure prediction confidence. we can use b-factors to flag the CDR loop residues. PyMOL can then color residues by b-factors, which enables us to visualize the computational results in their structural context. In Afpdb, we currently read/write b-factors at the residue level, i.e., all atoms in a residue share the same b-factor value, as we have not yet encountered the need for displaying b-factors at the atom level.

```
p=Protein(fk)
# read b_factors as a numpy array
print(p.b_factors())
```

```
[1. 1. 1. 1. 1. 1.]
```

```
# set b_factors by providing a numpy array
import numpy as np
p.b_factors(np.random.random((len(p))), "\n")
```

```
[0.0668895  0.31111105 0.04879224 0.51323109 0.13960504 0.86005201]
```

To color by b-factors, set `color = "b"` in the `show()` method. However, please normalize b-factor NumPy array to be within [0, 1] to get the expected coloring effect (although `show()` method will clip the values, it does not normalize them).

```
p=Protein(fn)
n=len(p)
p.b_factors(np.sin(np.arange(n)/n*np.pi))
p.show(style="cartoon", color="b")
```

It may be more convenient to get or set b-factors for each chain using `b_factors_by_chain()` method.

```
# in this example, we color residues based on how far they are from the P chain
p=Protein(fn)
rs, rs_seed, dist=p.rs_around("P", dist=1e8)
# important to sort the residues by their internal indices, so that the order of distance matches
 the internal b-factor array
# we use drop_duplicates to make sure each residue only occur once (to its nearest seed residue)
dist.drop_duplicates("resi_b", inplace=True)
dist.sort_values("resi_b", inplace=True)
assert len(dist)==len(p.rs("H:L"))
d_min,d_max=dist.dist.min(), dist.dist.max()
# b factors should be in [0, 1], if we want to color by b-factos.
dist["dist"]=(dist["dist"]-d_min)/(d_max-d_min)
b={"P":0, "H":dist[dist.chain_b=="H"].dist.values, "L":dist[dist.chain_b=="L"].dist.values}
# set b_factors
p.b_factors_by_chain(b)
p.show(color="b")
```

We can also provide a residue selection for `b_factors()`.

```
# Use b-factor to mimic color by chain
p=Protein(fn)
# provide b-factors as an array
p.b_factors(np.zeros(len(p.rs("H"))), rs="H")
# a number is considered a special case
p.b_factors(0.5, rs="L")
p.b_factors(1, rs="P")
p.show(color="b")
```

```
# visualize CDRs
p=Protein(fn)
# CDR sequence (Chothia definition) light chains followed by heavy chain
CDR_L=['RASQSVGNNKLA', 'GASSRPS', 'QQYGQSLST']
CDR_H=['GGSFSTY', 'IPLLTI', 'EGTTGDGDLGKPIGAFAH']
# color the framework resdues
p.b_factors(1, "H")
p.b_factors(0.9, "L")
p.b_factors(0.8, "P")
for i,(seq,chain) in enumerate(zip(CDR_L+CDR_H, ["L"]*3+["H"]*3)):
    b=i/20
    # we select the CDR fragment by sequence search
    # as rs_seq returns a list of RS, we take the first element
    rs_cdr=p.rs_seq(seq, in_chains=[chain])[0]
    p.b_factors(b, rs=rs_cdr)
p.show(color="b")
```

We can leverage PyMOL to create more powerful visualizations, which will be described next.

PyMOL Interface

PyMOL is a power structure visualization tool. Afpdb provides efficient and convenient programming interface for structure computation. An easy-to-use interface between Afpdb and PyMOL can help combine the strenght of both tools covering both structure analysis and visualization. The `PyMol()` method fills this role.

The method returns a PyMOL object, which has a `run` method enabling us to run any PyMOL commands.

```
# create a new PyMOL engine
pm=Protein().PyMOL()
pm.cmd(f"load {fn}, myobj")
p=Protein(fn)
rs_binder, rs_seed, t = p.rs_around("P", rs_within="H:L", dist=4)
# selecting all interface residues
rs_int = rs_binder | rs_seed
# convert Afpdb residue selection to a PyMOL selection command, where the selection object is nam
ed "myint"
rs_str = rs_int.str(format="PYMOL", rs_name="myint")
print(rs_str, "\n")
```

```
select myint, (chain L and resi 33+92-95+97) or (chain H and resi 30+32+46+49-51+54+56-58+98+106-
109) or (chain P and resi 1-7+9-10)

# We can also control the selection to CA only
print(rs_int.str(format="PYMOL", rs_name="myint_ca", ats="CA"), "\n")

select myint_ca, ((chain L and resi 33+92-95+97) or (chain H and resi 30+32+46+49-51+54+56-
58+98+106-109) or (chain P and resi 1-7+9-10)) and name CA
```

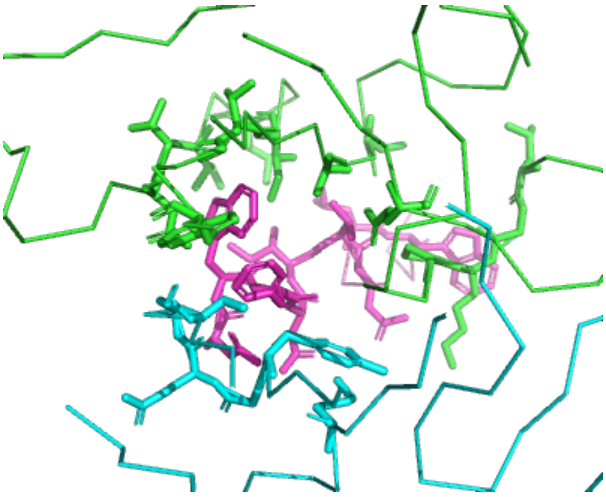
```
pm.run(f"""
# color by chain
as ribbon, myobj
util.cbc
# defines a selection named myint
{rs_str}
show sticks, myint
# focus on the selection
zoom myint
""")
pm.run("""deselect; save mypm.png; save mypm.pse""")
# dispose the PyMOL object to save system resource
pm.close()
```

```
PyMOL>load /Users/zhoyyi1/afpdb/tutorial/example_files/5cil.pdb, myobj
 CmdLoad: "" loaded as "myobj".
PyMOL>
PyMOL># color by chain
PyMOL>as ribbon, myobj
PyMOL>util.cbc
 util.cbc: color 26, (chain H)
 util.cbc: color 5, (chain L)
 Executive: Colored 939 atoms.
 util.cbc: color 154, (chain P)
 Executive: Colored 844 atoms.
 Executive: Colored 130 atoms.
PyMOL>
PyMOL># defines a selection named myint
PyMOL>select myint, (chain L and resi 33+92-95+97) or (chain H and resi 30+32+46+49-51+54+56-
58+98+106-109) or (chain P and resi 1-7+9-10)
 Selector: selection "myint" defined with 245 atoms.
PyMOL>show sticks, myint
PyMOL>
PyMOL># focus on the selection
PyMOL>zoom myint
PyMOL>
PyMOL>deselect; save mypm.png; save mypm.pse
 Ray: render time: 0.31 sec. = 11623.1 frames/hour (0.31 sec. accum.).
```

```
Save: Please wait -- writing session file...
ScenePNG: wrote 640x480 pixel image to file "mypm.png".
Save: wrote "mypm.pse".
```

The generated mypm.pse is a PyMOL session file to visualize the binding interface in 3D.

We here also automatically save the image into a PNG file as below.



As shown in this example, we can send any PyMOL commands via the `run` method. The input can be a single line or multiple lines. We can use `#` to add comments. The computation, i.e., the identification of the interface residues, can be done in Afpdb using `rs_around`. A residue selection object's `str` method casts the selection into a PyMOL selection command:

```
rs_str = rs_int.str(format="PYMOL", rs_name="myint")
```

turns the interface residues into the command that defines a selection object named `"myint"`:

```
select myint, (chain L and resi 33+92-95+97) or (chain H and resi 30+32+46+49-51+54+56-58+98+106-109) or (chain P and resi 1-7+9-10)
```

If we only want to select CA atoms, we can do:

```
rs_str = rs_int.str(format="PYMOL", rs_name="myint", ats="CA")
```

This will produce:

```
select myint, ((chain L and resi 33+92-95+97) or (chain H and resi 30+32+46+49-51+54+56-58+98+106-109) or (chain P and resi 1-7+9-10)) and name CA
```

When there are multiple objects in the PyMOL session, the argument `obj_name` can constrain the selection to that object.

PyMOL command `show sticks, myint` and `zoom myint` shows the interface residues as sticks and zoom in to focus on them. We can also provide multiple PyMOL commands into one line with `";"` as the concatenator. The example line is:

```
deselect; save mypm.png; save mypm.pse
```

Users can use all the PyMOL command they are already familiar with, but use **Afpdb** for the computational tasks that are not straightforward to do in PyMOL. Destroy the `PyMOL()` instance with `close()`, when it is no longer needed to save resource.

Note: If we already have PyMOL commands saved into a text file, we can also use `pm.script(script_file_name)` to execute it.

Distance

In a previous example, `rs_around` method already uses the distance measurement method `rs_dist` internally.

Given two residue groups, `rs_a` and `rs_b`, `rs_dist` computes the distance between all residue pairs. The distance is determined based on the two closest atoms. The output dataframe is sorted by distance and indicates which atom pair contributes to the distance measure.

```
p=Protein(fk)
print(p.seq_dict())
print(p.rs().name(), "\n")

t=p.rs_dist('L', 'H')
t
```

	chain_a	resn_a	resn_i_a	resi_a	res_a	chain_b	resn_b	resn_i_b	resi_b	res_b	dist	atom_a	atom_b
486	L	10	10	3	Q	H	3	3	4	L	21.225081	OE1	CD1
111	L	5	5	0	E	H	3	3	4	L	22.955209	OE1	CD1
201	L	6A	6	1	I	H	3	3	4	L	23.348418	O	CD1
291	L	6B	6	2	V	H	3	3	4	L	25.339045	C	CD1
488	L	10	10	3	Q	H	4	4	5	V	25.544916	OE1	N
115	L	5	5	0	E	H	4	4	5	V	27.160638	OE1	C
203	L	6A	6	1	I	H	4	4	5	V	28.243298	O	N
293	L	6B	6	2	V	H	4	4	5	V	30.206544	C	N

```
# we restricted the distance measurement to CA-CA distance, notice the distance are now longer
t=p.rs_dist('L', 'H', ats='CA')
t
```

	chain_a	resn_a	resn_i_a	resi_a	res_a	chain_b	resn_b	resn_i_b	resi_b	res_b	dist	atom_a	atom_b
2	L	6A	6	1	I	H	3	3	4	L	28.564522	CA	CA
0	L	5	5	0	E	H	3	3	4	L	28.887439	CA	CA
6	L	10	10	3	Q	H	3	3	4	L	28.978506	CA	CA
4	L	6B	6	2	V	H	3	3	4	L	29.018223	CA	CA
3	L	6A	6	1	I	H	4	4	5	V	30.646395	CA	CA
7	L	10	10	3	Q	H	4	4	5	V	30.718995	CA	CA
1	L	5	5	0	E	H	4	4	5	V	30.727932	CA	CA
5	L	6B	6	2	V	H	4	4	5	V	30.854223	CA	CA

Instead of residue distance, we can also measure the distance between all-atom combinations. Our aim in the future is to add a column for interaction types, such as clash, electrostatic, hydrogen bonds, etc. Please contact us, if you have the knowledge in how to classify the interaction type of each atom pair.

```
# We list distances between all atom pairs of two residues
p=Protein(fk)
t=p.atom_dist('L6A', 'H3')
print(f"\n... total rows: {len(t)}\n")
```

```
... total rows: 64
```

```
t[:10]
```

	chain_a	resn_a	resn_i_a	resi_a	res_a	chain_b	resn_b	resn_i_b	resi_b	res_b	dist	atom_a	atom_b
38	L	6A	6	1	I	H	3	3	4	L	23.348418	O	CD1
30	L	6A	6	1	I	H	3	3	4	L	24.169027	CB	CD1
62	L	6A	6	1	I	H	3	3	4	L	24.431554	CD1	CD1
22	L	6A	6	1	I	H	3	3	4	L	24.498690	C	CD1
54	L	6A	6	1	I	H	3	3	4	L	24.577985	CG2	CD1
6	L	6A	6	1	I	H	3	3	4	L	24.611640	N	CD1
46	L	6A	6	1	I	H	3	3	4	L	24.671778	CG1	CD1
14	L	6A	6	1	I	H	3	3	4	L	24.859585	CA	CD1
37	L	6A	6	1	I	H	3	3	4	L	24.870263	O	CG
39	L	6A	6	1	I	H	3	3	4	L	25.445370	O	CD2

```
# We list distances between all backbone atom pairs of two residues
t=p.atom_dist('L6A', 'H3', ats=['CA','C','N','O'])
t
```

	chain_a	resn_a	resn_i_a	resi_a	res_a	chain_b	resn_b	resn_i_b	resi_b	res_b	dist	atom_a	atom_b
13	L	6A	6	1	I	H	3	3	4	L	26.966275	O	CA
15	L	6A	6	1	I	H	3	3	4	L	27.207144	O	O
14	L	6A	6	1	I	H	3	3	4	L	27.473148	O	C
12	L	6A	6	1	I	H	3	3	4	L	27.482269	O	N
9	L	6A	6	1	I	H	3	3	4	L	28.142550	C	CA
1	L	6A	6	1	I	H	3	3	4	L	28.299546	N	CA
11	L	6A	6	1	I	H	3	3	4	L	28.414299	C	O
5	L	6A	6	1	I	H	3	3	4	L	28.564522	CA	CA
8	L	6A	6	1	I	H	3	3	4	L	28.646190	C	N
10	L	6A	6	1	I	H	3	3	4	L	28.670499	C	C
3	L	6A	6	1	I	H	3	3	4	L	28.727160	N	O
0	L	6A	6	1	I	H	3	3	4	L	28.862343	N	N
2	L	6A	6	1	I	H	3	3	4	L	28.882357	N	C
7	L	6A	6	1	I	H	3	3	4	L	28.956531	CA	O
4	L	6A	6	1	I	H	3	3	4	L	29.066193	CA	N
6	L	6A	6	1	I	H	3	3	4	L	29.154596	CA	C

Sometimes we compute inter-residue distance as distances between CB atoms, however, since Glyine does not have CB atom, special care needs to bee taken. We can compute distances among both CA and CB, then remove records associated with non-G CA atoms.

```
p=Protein(fn)
t=p.atom_dist("L8-10", "P1-2", ats="CA,CB")
bad_entry=((t.res_a!="G")&(t.atom_a=="CA"))|((t.res_b!="G")&(t.atom_b=="CA"))
t=t[~ bad_entry]
t
```

	chain_a	resn_a	resn_i_a	resi_a	res_a	chain_b	resn_b	resn_i_b	resi_b	res_b	dist	atom_a	atom_b
9	L	9	9	8	G	P	1	1	237	N	28.179702	CA	CB
11	L	9	9	8	G	P	2	2	238	W	29.129874	CA	CB
5	L	8	8	7	P	P	1	1	237	N	30.261281	CB	CB
7	L	8	8	7	P	P	2	2	238	W	31.916355	CB	CB
17	L	10	10	9	T	P	1	1	237	N	32.302604	CB	CB
19	L	10	10	9	T	P	2	2	238	W	33.047554	CB	CB

This logic is implemented in method cb_dist().

RMSD

To measure the root mean square deviation (RMSD) between two residue selections, we first translate one molecule by 5Å, then call `rmsd()`. The method `rmsd()` takes a target object, source residue list, target residue list, and optionally an atom selection.

```
p=Protein(fk)
q=p.translate([3.0,4.0,0.0], inplace=False)
print("Translation:", p.center(), q.center(), q.center()-p.center(), "\n")
```

```
Translation: [ 15.54949999  -8.0205001  -15.39166681] [ 18.54949999  -4.0205001  -15.39166681]
[3. 4. 0.]
```

```
# consider all matched atoms, rs='ALL' or rs=None mean all residues
print(q.rmsd(p, None, None), "\n")
```

```
5.0
```

```
# consider CA only
print(q.rmsd(p, ats='CA'))
```

```
5.0
```

The two selections should have the same number of residues, unless one selection has only one residue. Otherwise, it will error. Make sure the order of residues in the two selections is what you want, as the distance is computed for each residue pairwise in order. If two residues are of different types, only the common atoms are used, which is enforced by the `atom_mask` array. If we have different residue types, we recommend users use `ats` to explicitly restrict the computation to the backbone atoms only, e.g., `"N,CA,C,O"` or `"CA"`.

`rmsd()` and `align()` are two methods where the order of the residues in the input matters, therefore, argument `rl_a` and `rl_b` are Residue Selection (RL) instead of RS. The following example shows why it is important to make sure the residue orders should match:

```
p=Protein(fn)
print(p.chain_id(), "\n")
```

```
['L', 'H', 'P']
```

```
# protein p and q have different chain orders
q=p.extract("P:H:L")
print(q.chain_id(), "\n")
```

```
['P' 'H' 'L']
```

```
# The default behavior of using Residue List returns the right answer
print(p.rmsd(q, "H:L", "H:L", ats="CA"), "\n")
```

```
0.0
```

```
# However, if you pass in residue select, the result is wrong in this case, as residues in the RS
# has been reordered
print(p.rmsd(q, p.rs("H:L"), q.rs("H:L"), ats="CA"), "\n")
```

```
29.09069364318863
```

```
# this is because in object p, p.rs("H:L") standarize the residue index and put L residues in
# front of H residue
# in object q, p.rs("H:L") keeps H residues in front of L residues.
# Therefore, residue selection breaks the one-to-one mapping relationship between the
# two residue lists.
```

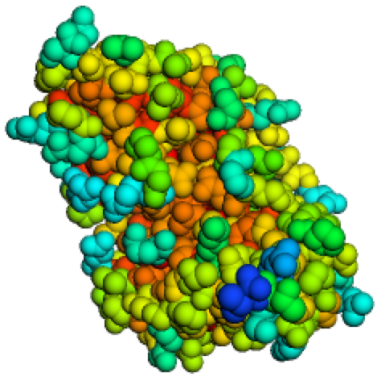
```
# This example is created to demonstrate why we need to be careful, if we use RS objects for
# the RL argument in rmsd() and align().
```

We often need to align the two structures first, before we take RMSD measurement. If we pass `align=True` into `rmsd()`, object `p` will then be first aligned to `q`, RMSD is then calculated. The `align()` method will be explained later.

Solvent-Accessible Surface Area (SASA)

We often would like to exclude some chains during the surface area computation. Although we can use `extract()` to take the chains we like, the `in_chains` argument provides more convenience. For example, if all chains are included in the antibody-antigen complex, CDR binding residues will not be accessible to water. But if the antigen chain is excluded, contact CDR residues will be exposed. Computing the different of SASA values with different `in_chains` is another popular way of identifying interface residues in addition to using `rs_around`.

```
p=Protein(fn)
# consider all chains, H:L:P, so the interface residues between H:L and P are mostly buried with
low SASA values
t_HLP=p.sasa()
sasa=p.sasa().SASA.values
p.b_factors(sasa/sasa.max())
p.show(style="sphere", color="b")
```



```
# Get residue surface of a few L-chain residues
t_all=t_HLP[(t_HLP.chain=="L")][90:105]
t_all
```

	chain	resn	resn_i	resi	aa	SASA	rSASA
90	L	91	91	90	Q	1.071459	0.005669
91	L	92	92	91	Y	1.071459	0.004679
92	L	93	93	92	G	19.422432	0.228499
93	L	94	94	93	Q	61.171949	0.323661
94	L	95	95	94	S	41.911208	0.343534
95	L	96	96	95	L	37.959864	0.210888
96	L	97	97	96	S	3.214377	0.026347
97	L	98	98	97	T	12.076282	0.082714
98	L	99	99	98	F	17.390160	0.079771
99	L	100	100	99	G	0.000000	0.000000
100	L	101	101	100	Q	124.880067	0.660741
101	L	102	102	101	G	14.201306	0.167074
102	L	103	103	102	T	2.142918	0.014678
103	L	104	104	103	K	103.554748	0.490781
104	L	105	105	104	V	3.214377	0.020090

rSASA is defined as the relative SASA, i.e., SASA of each residue is normalized by its maximum possible surface area as defined in the G-X-G peptide: SASA/MaxResidueSASA. We use the values defined in the paper: https://arxiv.org/pdf/1211.4251

```
print("\nnotice residue L92 only has a small surface area 1.071459\n")
# ignore antigen, CDR contact residues are now exposed with high SASA values
t_HL=p.sasa("H:L")
t_ag=t_HL[(t_HL.chain=="L")][90:105]
t_ag
```

	chain	resn	resn_i	resi	aa	SASA	rSASA
216	L	91	91	90	Q	1.071459	0.005669
217	L	92	92	91	Y	36.329227	0.158643
218	L	93	93	92	G	33.896076	0.398777
219	L	94	94	93	Q	78.605526	0.415902
220	L	95	95	94	S	80.936500	0.663414
221	L	96	96	95	L	37.959864	0.210888
222	L	97	97	96	S	8.044890	0.065942
223	L	98	98	97	T	12.076282	0.082714
224	L	99	99	98	F	17.390160	0.079771
225	L	100	100	99	G	0.000000	0.000000
226	L	101	101	100	Q	124.880067	0.660741
227	L	102	102	101	G	14.201306	0.167074
228	L	103	103	102	T	2.142918	0.014678
229	L	104	104	103	K	103.554748	0.490781
230	L	105	105	104	V	3.214377	0.020090

```
print("\nnotice residue L92 now has a large surface area 36.3292\n\n")
t_ag['dSASA']=t_ag['SASA'].values-t_all['SASA'].values
t_ag
```

	chain	resn	resn_i	resi	aa	SASA	rSASA	dSASA
216	L	91	91	90	Q	1.071459	0.005669	0.000000
217	L	92	92	91	Y	36.329227	0.158643	35.257768
218	L	93	93	92	G	33.896076	0.398777	14.473644
219	L	94	94	93	Q	78.605526	0.415902	17.433577
220	L	95	95	94	S	80.936500	0.663414	39.025291
221	L	96	96	95	L	37.959864	0.210888	0.000000
222	L	97	97	96	S	8.044890	0.065942	4.830513
223	L	98	98	97	T	12.076282	0.082714	0.000000
224	L	99	99	98	F	17.390160	0.079771	0.000000
225	L	100	100	99	G	0.000000	0.000000	0.000000
226	L	101	101	100	Q	124.880067	0.660741	0.000000
227	L	102	102	101	G	14.201306	0.167074	0.000000
228	L	103	103	102	T	2.142918	0.014678	0.000000
229	L	104	104	103	K	103.554748	0.490781	0.000000
230	L	105	105	104	V	3.214377	0.020090	0.000000

```
binder, rs_seed, t=p.rs_around(p.rs("P"), rs_within="L", dist=4)
print("Interface residues can also be identified with rs_around directly:", binder, "\n")
# the two methods are quite consistent with each other
```

```
Interface residues can also be identified with rs_around directly: L33,92-95,97
```

Since we often need to compute the delta in SASA, there is a method `dsasa()` specifically defined for doing all steps shown above. The example below uses `dsasa()` to identify the Ab-Ag interface residues using the ideas outlined in the previous example.

```
p=Protein(fn)
# compute the delta SASA for the interface formed between H:L and P
t=p.dsasa("H:L", "P")
p.b_factors(1-t.drSASA.values)
p.show(color="b")
```

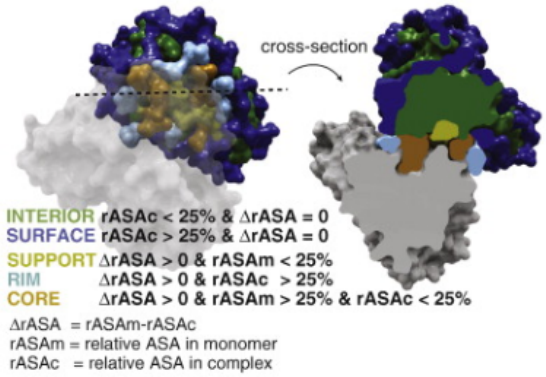
```
# this is mostly equivalent to finding interface residues with rs_around
rs_L, rs_seed,_ = p.rs_around("P", rs_within="L", dist=4)
# Let's display the result for residues on the L residues in the L-P interface
t[t.resi.isin(rs_L.data)]
```

This demonstrates that we can either use rs_around to select interface residues, or calculate $\Delta ASAS$. The `dsasa()` method not only compute the SASA changes in the normalized scale drSASA, it also classify the types of residues according to https://doi.org/10.1016/j.jmb.2010.09.028.

rSASAm is the rSASA when the two components are separated ("m" stands for monomer). rSASAc is when the two components for a complex ("c" stands for complex). Based on these two values and drSASA (rSASAm - rSASAc), residues can be classified into:

- iterior residues: No change between rSASAm and rSASAc, never accessible.

- surface residues: No change bewteen rSASAm and rSASAc, always accessible.
- core interface residues: accessible in rSASAm but inaccessible in rSASAc.
- rim interface residues: reduced SASA in the complex, but remain accessible in the complex.
- support interface residues: reduced SASA in the complex, but inaccessible in the monomer.



Secondary Structures - DSSP

Method `dssp` is a wrapper on BioPython's `Bio.PDB.DSSP` API, which requires the pre-installation of the DSSP package as described in https://en.wikipedia.org/wiki/DSSP_(algorithm).

We can install DSSP with:

```
conda install sbl::dssp
```

The secondary structure codes are:

- G = 3-turn helix (310 helix). Min length 3 residues.
- H = 4-turn helix (α helix). Minimum length 4 residues.
- I = 5-turn helix (π helix). Minimum length 5 residues.
- T = hydrogen bonded turn (3, 4 or 5 turn)
- E = extended strand in parallel and/or anti-parallel β-sheet conformation. Min length 2 residues.
- B = residue in isolated β-bridge (single pair β-sheet hydrogen bond formation)
- S = bend (the only non-hydrogen-bond based assignment).
- C = coil (residues which are not in any of the above conformations).

In the simplified mode, we map the codes into a - alpha helix, b - beta sheet, and c - coil.

```
# BioPython relies on DSSP command line program to compute secondary structures
if IN_COLAB:
    if not os.path.isfile("INSTALL_DSSP"):
        ! conda install sbl::dssp
        ! touch INSTALL_DSSP

p=Protein(fn)
# Biopython relies on command line DSSP program, it does not automatically detect the version, so
 we provides a locate_dssp() method
# locate_dss() does this for us
dssp_settings=Protein.locate_dssp()
print("DSSP command line and its version:", dssp_settings)
```

```
ss=p.dssp(simplify=True, **dssp_settings)
print(ss, "\n")
```

```
DSSP command line and its version: {'DSSP': '/Users/zhoyyi1/anaconda3/bin/mkdssp',
'dssp_version': '4.2.2.1'}
{'L': '---bb-cccbbbb-cc--bbbbbbbccPPaaa-bbbbbb-ccc--bbbbbcccb--ccc-ccbbbbbbccbbbbbbcc--aaa-
cbbbbbb-cccc-bb---bbbbb----', 'H': '-bbbbPPPbbb-cc--bbbbbbbbcc-cccc-bbbbbb-ccc-bbbbbbbbaaacbbbb-
aaaccbbbbbbcccbbbbbb-c--aaa-bbbbbbbbbbccccccc-bbbbbbbb---bbbbb--', 'P': '-aaaaaaaaaa-'}
```

```
# set helix to 1, otherwise 0
for k,v in ss.items():
    b=np.zeros(len(v))+0.5 # default to turns
    b[np.array(list(v))=="a"]=0
    b[np.array(list(v))=="b"]=0.8
    b[np.array(list(v))=="c"]=0.3
    ss[k]=b
p.b_factors_by_chain(ss)
p.show(color="b")
```

Internal Coordinate

To obtain the bond length (for backbone) and bond angles:

```
p=Protein(fk)
t=p.internal_coord(rs="L")
t
```

	chain	resn	resn_i	resi	aa	-1C:N	N:CA	CA:C	phi	psi	omg	tau	chi1	chi2	chi3	chi4	chi5
0	L	5	5	0	E	NaN	1.483377	1.532696	NaN	127.190158	NaN	112.098322	-172.925399	-176.577934	-176.247607	NaN	NaN
1	L	6A	6	1	I	1.313512	1.449593	1.512209	-83.955214	121.806014	177.408124	108.596746	-57.806173	-167.083730	NaN	NaN	NaN
2	L	6B	6	2	V	1.325801	1.438910	1.527408	-92.621830	NaN	174.410099	108.287317	175.406001	NaN	NaN	NaN	NaN
3	L	10	10	3	Q	NaN	1.451717	1.535136	NaN	NaN	NaN	109.054940	-65.144361	-175.385704	-107.025451	NaN	NaN

resi is the internal residue_index, and *resn* is residue name. The bond lengths are "-1C:N" (previous C to current N), N-CA, and CA-C, followed by all bond angles.

Notice ϕ and ω angle for the first residue is `NaN`, as there is no peptide bond for the first residue.

Similarly, ψ angle is not defined for the last residue.

The peptide bond length is `NaN` between residues 6B and 10, as there are missing residues. If we do want to know how far these two residues are, we can modify the default `MaxPeptideBond` from 1.4 to a bigger value (which can be useful for imperfect structures output by diffusion methods):

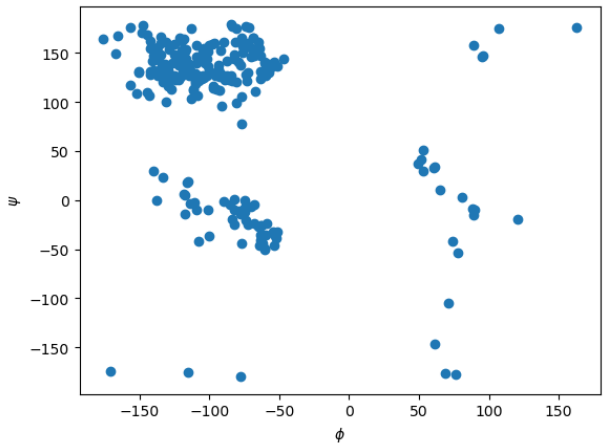
```
p=Protein(fk)
t=p.internal_coord(rs="L", MaxPeptideBond=1e6)
t
```

	chain	resn	resn_i	resi	aa	-1C:N	N:CA	CA:C	phi	psi	omg	tau	chi1	chi2	chi3	chi4	chi5
0	L	5	5	0	E	NaN	1.483377	1.532696	NaN	127.190158	NaN	112.098322	-172.925399	-176.577934	-176.247607	NaN	NaN
1	L	6A	6	1	I	1.313512	1.449593	1.512209	-83.955214	121.806014	177.408124	108.596746	-57.806173	-167.083730	NaN	NaN	NaN
2	L	6B	6	2	V	1.325801	1.438910	1.527408	-92.621830	176.913905	174.410099	108.287317	175.406001	NaN	NaN	NaN	NaN
3	L	10	10	3	Q	7.310299	1.451717	1.535136	-165.152326	NaN	-147.486795	109.054940	-65.144361	-175.385704	-107.025451	NaN	NaN

To run the example below, make sure you have matplotlib installed. If not, run in the shell:

```
pip install matplotlib
```

```
p=Protein(fn)
t=p.internal_coord()
#t.display()
from matplotlib import pyplot as plt
plt.scatter(t.phi, t.psi)
plt.xlabel('$\\phi$')
plt.ylabel('$\\psi$')
print("Ramachandran Plot")
```




```
#TODO: add the boundaries for popular engles, similar to the countours in https://www.mathworks.com/help/bioinfo/ref/ramachandran.html
```

Protein Object Manipulation

Move Objects

We can recenter, translate, and rotate an object.

```
p=Protein(fk)
print("Original center:", p.center(), "\n")
```

```
Original center: [ 15.54949999  -8.0205001  -15.39166681]
```

```
q=p.center_at([3.0,4.0,5.0])
print("New center after recentering:", q.center(), "\n")
```

```
New center after recentering: [3. 4. 5.]
```

```
q=p.translate([1.,0.,-1.])
print("New center after translate:", q.center(), "\n")
```

```
New center after translate: [ 16.54949999  -8.0205001  -16.39166681]
```

```
# rotate 90 degrees around axis [1,1,1]
print("RMSD before rotation:", q.rmsd(p, None, None, "CA"), "\n")
```

```
RMSD before rotation: 1.4142135623730951
```

```
# we only rotate 5 degrees
q=p.rotate([1,1,1], 5)
print("Old center:", p.center(), "New center:", q.center(), "\n")
```

```
Old center: [ 15.54949999  -8.0205001  -15.39166681] New center: [ 15.10944354  -6.44301224 -
16.52909822]
```

```
print("RMSD after rotation:", q.rmsd(p, ats="CA"), "\n")
```

```
RMSD after rotation: 2.190194408912404
```

```
# Warning, rotate is done using the origin as the center, that's why the rotation dramatically in
crease RMSD in the example above.
# most of the time, it's only meaningful if we center the molecule before rotation
rc=p.center()
q=p.center_at([0,0,0])
# rotate 5 degrees
q.rotate([1,1,1], 5, inplace=True)
# restore the center to the old center
q.center_at(rc, inplace=True)
# now the RMSD is much smaller
print("RMSD (center before rotate):", q.rmsd(p, None, None, "CA", align=True), "\n")
```

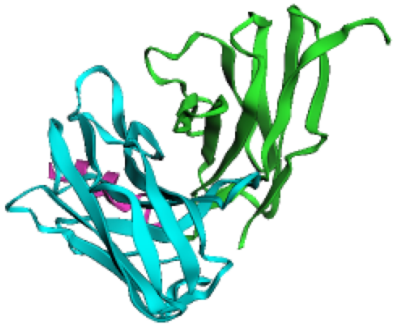
```
RMSD (center before rotate): 1.926437630612445e-12
```

To reset the position of an object, use `reset_pos()`. This will place the object center at the origin, as well as align its three PCA axes along Z, X, and Y axis, respectively.

```
p=Protein(fn)
print("Old center:", p.center(), "\n")
```

```
Old center: [ 20.15256404  -5.12843199 -15.77199195]
```

```
# rotation around a random axis direction with a random angle
p.rotate(np.random.random(3), np.random.random()*180)
p.show()
```



```
p.reset_pos()
p.show()
```

```
print("New center:", p.center())
```

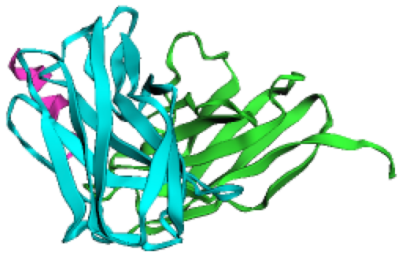
```
New center: [-9.80548975e-15 -2.61479727e-15  7.52908846e-15]
```

Align

It makes sense to align two objects before making RMSD measurement. Align method takes two residule lists (not residue selection) as it arguments. The default value is None, which refers to all residues.

```
p=Protein(fn)
q=p.translate([3.,4.,0.], inplace=False)
q.rotate([1,1,1], 90, inplace=True)
p.show()
```

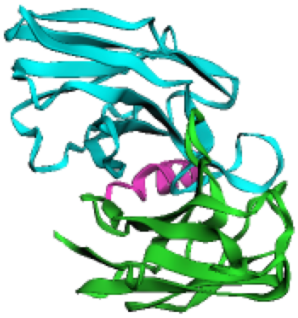
```
q.show()
```



```
print("RMSD before alignment:", q.rmsd(p, "ALL", "ALL", ats='CA'))
```

```
RMSD before alignment: 45.854280383656324
```

```
R,t=q.align(p, ats="CA")
q.show()
```



```
print("RMSD after alignment:", q.rmsd(p, ats='CA'))
print("Rotation matrix:\n", R, "\n")
print("Translation vector:", t)
```

```
RMSD after alignment: 2.470821699978209e-07
Rotation matrix:
 [[ 0.33333331 -0.24401692  0.91068361]
 [ 0.91068361  0.33333331 -0.24401693]
 [-0.24401692  0.91068361  0.33333331]]

Translation vector: [[-3.00000031e+00 -3.99999992e+00  3.11646732e-07]]
```

RMSD usually requires a pre-alignment, therefore, we can combine the two into one by providing `align=True` to `rmsd()`. Notice the coordinates of object `p` is changed as the side effect!

```
p=Protein(fn)
q=p.translate([3.,4.,0.], inplace=False)
q.rotate([1,1,1], 90, inplace=True)
print("RMSD without alignment:", p.rmsd(q), "\n")
```

```
RMSD without alignment: 45.98047321761866
```

```
print("RMSD with built-in alignment:", p.rmsd(q, align=True), "\n")
# The side effect is p has been changed by align=True
```

```
RMSD with built-in alignment: 2.493618546797601e-07
```

```
print("RMSD without alignment on the new p object:", p.rmsd(q), "\n")
```

```
RMSD without alignment on the new p object: 2.493618546797601e-07
```

Align Sequences

Both `align()` and `rmsd()` requires a pair of pre-aligned RL objects. If two proteins share the same sequence or a few spot mutations, the whole sequence can be used as the RL arguments. However, where there are insertions and deletions, it can be tedious to construct the RL pairs manually. This can happen even when we compare an experimental structure to its predicted structure, because modeling software may not be able to handle some non-canonical residues and resulting missing residues in the models. Afpdb can programmatically align two sequences and extract the aligned subset of residues as a RL pair.

We first create two proteins, one with two missing residues:

```
p=Protein(fn)
# create q with two deletions
q=p.extract(~ p.rs("H50,90"))
# by default the two sequences are algined using Biopython global alignment
x, a, b=Protein.find_aligned_positions(p.rs("H").seq(), q.rs("H").seq())
print(x[1], "\n")
#If the second sequence is a truncation of the first one, local alignment generally works better
(no penalty for ends)
x, a, b=Protein.find_aligned_positions(p.rs("H").seq(), q.rs("H40-110").seq(), is_global=False)
print(x[1])
```

```
VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS
||||||||||||||||||||||||||||||||||||||||||||||||||| |||||||||||||||||||||||||||||||||||||||||| |||||||||||||||||||||||||||||||||||
VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGG-IPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPED-AVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS
  Score=247


40 PGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPI
   |||||||||| |||||||||||||||||||||||||||||||||| |||||||||||||||||||
 1 PGRGLEWMGG-IPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPED-AVYYCAREGTTGDGDLGKPI
  Score=137
```

The method find_aligned_positons is a low-level debugging method, we generally use a higher-level method rl_align instead.

```
rl_q, rl_p=q.rl_align(p)
print(rl_p, "\n")
```

```
L:H1-49:H51-89:H91-126:P
```

```
print(rl_q)
```

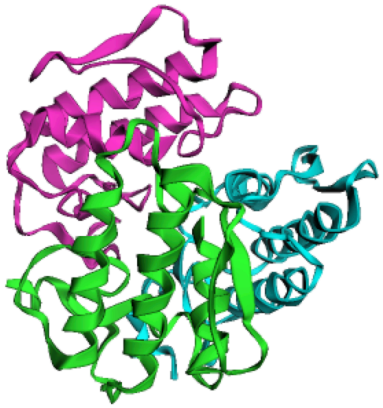
```
L:H:P
```

rl_algin uses is_global = True by default and one can adjust some scores to fine tune the alignment algorithm. In the above example, rl_align() returns a pair of RL object for all aligned residue pairs. Here since object q has H50 and H100 removed, we can see these two residues are not in the rl_p object. All residues in object q are used, therefore, its contig is L:H:P. The resultant RL object can then be used in the align() or rmsd() methods.

Split & Merge Objects

To split an object into multiple objects, just use `extract`. To merge multiple structure objects into one, use `merge`.

```
# 1a3d.pdb is a three-chain homotrimer.
p=Protein(pwd / "example_files/1a3d.pdb")
print(p.seq_dict(), "\n")
p.show()
```



```
# Extract each chain into a new object
Q=[ p.extract(x, inplace=False) for x in p.chain_id() ]
print("Object 1:", Q[0].seq_dict(), "\n")
```

```
Object 1: {'A':
'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCDCDRLA
AICFAGAPYNDNNYNIDLKARCQ'}
```

```
#for object 2 & 3, computer RMSD, then align them against object 1 and print RMSD after alignment
for i,q in enumerate(Q[1:]):
    # we intentionally rename chains to A for all object, the merge method will be able to auto r
ename chains
    q.rename_chains({'B':'A', 'C':'A'})
    print(f"Object {i+2}:", q.seq_dict())
    print("RMSD to Object 1:", q.rmsd(Q[0], None, None, ats='N,CA,C,O'), "\n")
    q.align(Q[0], ats='N,CA,C,O')
    q.translate(np.random.random(3)*1.5) # add a small jitter, so that we can later see there are
 three objects overlaid
    print("RMSD after align:", q.rmsd(Q[0], None, None, ats='N,CA,C,O'), "\n")
```

```
Object 2: {'B':
'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCDCDRLA
AICFAGAPYNDNNYNIDLKARCQ'}
RMSD to Object 1: 29.520795074060185
RMSD after align: 1.01025425274622e-14

Object 3: {'C':
'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCDCDRLA
AICFAGAPYNDNNYNIDLKARCQ'}
RMSD to Object 1: 29.520795074060185
RMSD after align: 1.061576257419e-14
```

```
q=Protein.merge(Q)
print(q.seq_dict())
```

```
{'A':'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCD
CDRLAAICFAGAPYNDNNYNIDLKARCQ',
'B':'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCDC
DRLAAICFAGAPYNDNNYNIDLKARCQ',
'C':'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCDC
DRLAAICFAGAPYNDNNYNIDLKARCQ'}
```

```
q.show()
```

Parsers for AI Models

Afpdb also provides some utility classes to parse output of AlphaFold, ColabFold, ESMFold, and ProteinMPNN. Since these utilities do not take a protein structure object as either its input or output, they are not part of the Afpdb's Protein class, but are provided in the `afpdb.aiparser` module. Due to file size constrains, we do not include the output files from these models, but instead show the example code and their abbreviated output.

Parse AlphaFold Output

```
import afpdb.util
from afpdb.aiparser import AlphaFoldParser
# points the parser to the model output folder (which should contain .pdf and .pkl files)
x=AlphaFoldParser('af_output')
# x.data is a DataFrame listing all predictions and their key metrics
x.data.display()
```

	path	name	relaxed	model	seed	pkl_path	ptm	iptm
0	input/relaxed_model_1_multimer_v3_pred_1.pdb	relaxed_model_1_multimer_v3_pred_1	True	1	1	input/result_model_1_multimer_v3_pred_1.pkl	0.863508	0.853023
1	input/relaxed_model_3_multimer_v3_pred_0.pdb	relaxed_model_3_multimer_v3_pred_0	True	3	0	input/result_model_3_multimer_v3_pred_0.pkl	0.748894	0.659703
2	input/relaxed_model_3_multimer_v3_pred_1.pdb	relaxed_model_3_multimer_v3_pred_1	True	3	1	input/result_model_3_multimer_v3_pred_1.pkl	0.746389	0.657367
3	input/relaxed_model_1_multimer_v3_pred_4.pdb	relaxed_model_1_multimer_v3_pred_4	True	1	4	input/result_model_1_multimer_v3_pred_4.pkl	0.730452	0.65088
22	input/unrelaxed_model_1_multimer_v3_pred_1.pdb	unrelaxed_model_1_multimer_v3_pred_1	False	1	1	input/result_model_1_multimer_v3_pred_1.pkl	0.863508	0.853023
23	input/unrelaxed_model_3_multimer_v3_pred_0.pdb	unrelaxed_model_3_multimer_v3_pred_0	False	3	0	input/result_model_3_multimer_v3_pred_0.pkl	0.748894	0.659703
24	input/unrelaxed_model_3_multimer_v3_pred_1.pdb	unrelaxed_model_3_multimer_v3_pred_1	False	3	1	input/result_model_3_multimer_v3_pred_1.pkl	0.746389	0.657367
25	input/unrelaxed_model_1_multimer_v3_pred_4.pdb	unrelaxed_model_1_multimer_v3_pred_4	False	1	4	input/result_model_1_multimer_v3_pred_4.pkl	0.730452	0.65088
26	input/unrelaxed_model_1_multimer_v3_pred_3.pdb	unrelaxed_model_1_multimer_v3_pred_3	False	1	3	input/result_model_1_multimer_v3_pred_3.pkl	0.729049	0.647938

As shown, the parser compiles all the .pdb files, listing the relaxed model (if available) first. The model's ptm, iptm, and corresponding pickle files (.pkl) are included. The .pkl file contains pLDDT and PAE information. pLDDT is also embedded in the .pdb files as b-factors. Predictions are ranked by relaxed status, then by iPTM and PTM.

```
print(x.get_pdb())
```

```
input/relaxed_model_1_multimer_v3_pred_1.pdb
```

```
print(x.get_ptm())
```

```
0.8635075964711556
```

```
print(x.get_iptm())
```

```
0.8530234098434448
```

```
print(x.get_plddt().shape)
```

```
(554,)
```

```
print(x.get_plddt()[:5])
```

```
[76.5977254  91.16503033 95.96849556 97.97122074 98.53807982]
```

```
print(x.get_pae().shape)
```

```
(554, 554)
```

The `get_*` methods extract the corresponding properties. Both `get_plddt` and `get_pae` return Numpy array objects. These method takes an optional `idx` argument, default to `0`, indicating the row index of the prediction user is interested in (`idx = 0` is the best prediction listed in `x.data`).

The usage of other parsers is quite similar, therefore we simply outline the code and skip their outputs.

Parse ColabFold Output

```
from afpdb.aiparser import ColabFoldParser
x=ColabFoldParser('cf_output')
x.data.display()
print(x.get_pdb())
print(x.get_ptm())
print(x.get_iptm())
print(x.get_plddt())
pae=x.get_pae()
print(pae.shape)
```

Parse ESMFold Output

```
from afpdb.aiparser import ESMFoldParser
from afpdb.afpdb import Protein
from afpdb import util
x=ESMFoldParser('esmfold_output')
x.data.display()
print(x.get_plddt())
data=(x.get_pae())
print(data.shape)
p=Protein(x.get_pdb())
print(p.len_dict())
```

Parse ProteinMPNN Output

```
from afpad.aiparser import ProteinMPNNParser
from afpdb import util
x=ProteinMPNNParser('mpnn_output')
# results are sorted by score
x.data.display()
```

```
    path               name       sample    score    global_score    T    seq_recovery  seq
--  -----------------  -------  --------  -------  --------------  ---  --------------  ---------
-----------------------------------
 6  mpnn/seqs/1crn.fa  1crn.fa         7   1.1493          1.2145  0.1          0.5238
TVCCPSSLSEYLSCLSSGTPSSSCASETGCIIISGSSCPSSY
 3  mpnn/seqs/1crn.fa  1crn.fa         4   1.1639          1.206   0.1          0.5476
TVCCPSPLSSYLSCRSSGRPDSECAKLTGCIIISGSTCPSSY
 7  mpnn/seqs/1crn.fa  1crn.fa         8   1.1695          1.2297  0.1          0.5238
TVCCPSSLSSFLACLSSGTPVEECCKETGCIVISGSTCPSSY
 8  mpnn/seqs/1crn.fa  1crn.fa         9   1.1747          1.2383  0.1          0.5238
TVCCPSSLSDYLNCLSSGTPTSICAKETGCIVISGSNCSSDY
 9  mpnn/seqs/1crn.fa  1crn.fa        10   1.1962          1.2587  0.1          0.4762
TVCCPSPLSEYLECLSSGKPPEECCKETGCIIISGSSCPSSY
 5  mpnn/seqs/1crn.fa  1crn.fa         6   1.1966          1.2474  0.1          0.5714
TVCCPSPLSEYLACRSSGTPDSLCALATGCIIISGSSCPSDY
 2  mpnn/seqs/1crn.fa  1crn.fa         3   1.1978          1.242   0.1          0.5238
TICCPSPRSEYESCLSSGTPPSSCCEETGCIIISGSTCPSSF
 4  mpnn/seqs/1crn.fa  1crn.fa         5   1.1978          1.2609  0.1          0.5476
TVCCPSPLSEYLQCLSSGTPPEECAKATGCIIISGSTCPASY
 0  mpnn/seqs/1crn.fa  1crn.fa         1   1.1982          1.2612  0.1          0.5
TICCPSPLEEYLSCLSSGTPPEECCEETGCIIISGSSCPSDY
 1  mpnn/seqs/1crn.fa  1crn.fa         2   1.2151          1.2849  0.1          0.5476
TVCCPSPLAEYLECLSSGTPPEECAKATGCIIISGSTCPSDY
```

Protein AI Design Use Cases

One effective AI protein design strategy involves generating hypothetical binder backbone structures based on a target protein (e.g., using RFDiffusion), followed by using inverse folding AI models to create binder sequences (e.g., using ProteinMPNN). These designs can then be validated using structure prediction models like AlphaFold. In order to achieve successful designs, thousands of protein structures need to be manipulated within a project. Afpdb aims to increase the productivity for such large-scale AI protein design efforts.

In this section, we demonstrate a few real use cases on how Afpdb tools help in AI protein design processes.

Handle Missing Residues in AlphaFold Prediction

In evaluating protein structure prediction models such as AlphaFold, the true experimental structure `p_exp` may contain missing residues. We often replace missing residues with glycine for AlphaFold modeling. When align the predicted structure `p_af` against `p_exp`, we need to exclude those added G residues, as they do not exist in `p_exp`.

The example below first created a fake experimental `p_exp` with five missing residues, and a fake AF-predicted `p_af` with five extra Glycines. p_exp contains 106 residues with L15-19 missing. `p_af` contains 111 residues with five extra Gs.

From `p_exp`, `rsi_missing()` returns a residue selection object that can be used to point at the extra G residues in `p_af`. Methods `align` and `rmsd` can then be done with those extra residues excluded.

```
p=Protein(fn)
miss_residues="L15-19"

# create an experimental structure for chain L with residues L15-19 missing
p_exp=p.extract( ~ p.rs(miss_residues) & "L")
print(p_exp.seq(), "\n")

EIVLTQSPGTQSLSXXXXXTLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLS
TFGQGTKVEVKRTV

print("Notice five residues XXXXX corresponds to position 15-19\n")

Notice five residues XXXXX corresponds to position 15-19

af_seq=p_exp.seq(gap="G")
print(af_seq, "\n")

EIVLTQSPGTQSLSGGGGGTLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLS
TFGQGTKVEVKRTV

print("We run AlphaFold prediction using af_seq, i.e., replacing missing residues with
Gly ...\n")

We run AlphaFold prediction using af_seq, i.e., replacing missing residues with Gly ...

# Create a fake AlphaFold predicted structure, by changing the missing residues to Glycine
p_af=p.extract("L") # extract chain L
rs=p_af.rs(miss_residues)
print(rs)

L15-19

p_af.data.aatype[rs.data]=RS.i("G") # set residue type to Gly
# np.ix_ select all rows in rs.data and all columns in ~ATS("N,CA,C,O")).data
p_af.data.atom_mask[np.ix_(rs.data, (~ATS("N,CA,C,O")).data)]=0 # Gly does not have side chain
atoms
print("Pretend p_af is the output of AlphaFold prediction\n")
```

```
Pretend p_af is the output of AlphaFold prediction

print("Verify the missing resiudes are not missing in p_af, they are Gs:\n")
print(p_af.seq(), "\n")

Verify the missing resiudes are not missing in p_af, they are Gs:
EIVLTQSPGTQSLSGGGGGTLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLS
TFGQGTKVEVKRTV

print(f"# of residues: p_exp={len(p_exp)}, p_af={len(p_af)}")

# of residues: p_exp=106, p_af=111

print("AlphaFold predicted structure has 5 more residues, which needs to be excluded for
alignment purpose.\n")

AlphaFold predicted structure has 5 more residues, which needs to be excluded for alignment
purpose.

# we now need to align the common residues, excluding the inserted G residues as they do not
exist in p_exp
# the above is equivalent to first find the missing residue indices, notice rsi_missing returns
an array
# rsi=p_exp.rsi_missng()
# warning: do not use rsi on p_exp, as rsi contains non-existing residue indices!!!
# cast rsi into a RS object for object p_af
# rs_miss=RS(p_af, rsi)

# The shortcut is to find the inserted residues in p_af with respect to p_exp
rs_miss=p_af.rs_insertion(p_exp)

# now rs_miss is meaningful, as they point to the 5 Gs
print(rs_miss, rs_miss.seq(), "\n")

L15-19 GGGGG

# now align the two structure and measure RMSD
print(p_exp.rmsd(p_af, rl_b=~rs_miss, ats="CA", align=True), "\n")

5.956938906837272e-15
```

Structure Prediction with ESMFold

We provide a `fold()` method, which uses Meta's free EMSFold web service to predict protein structures. Our method support multiple chains, as well as missing residues. The multi-chain support was based on the idea outlined in the "Merge & Split" section, i.e., by default we concatenate chains by 50 glycine residues (gap=50 can be modified) into one chain. Missing residues are replaced by glycine just as we did in the above AlphaFold example.

Please be aware that ESMFold generally produces less accurate results compared to AlphaFold. The web service limits the intermediate sequence length to maximally 400 residues. Connection can fail after a few continuously predicted, as this public resource is protected from being overused. `fold()` is a quick way of occasionally turning a sequence into a structure.

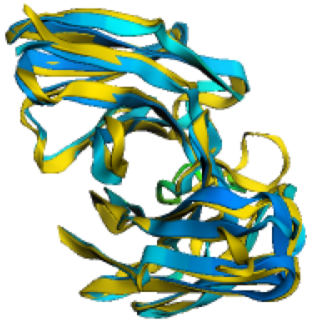
We here provide an example of predicting the antibody structure of 5CLI.

Let us try the Ab-Ag complex, which ESMFold failed to prediction.

```
p=Protein(fn)
exp=p.extract("H:L")
try:
    pred=Protein.fold(exp.seq())
    print(pred.rmsd(exp, ats="N,CA,C,O", align=True))
    exp.b_factors(0.2)
    q=Protein.merge([exp, pred])
    q.show(color="b")
```

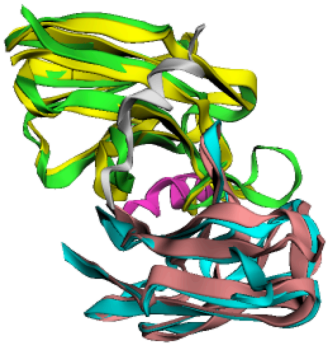
```
    #q.save(pwd / "example_files/esmfold.pdb")
except Exception as e:
    print(e)
    print("ESMFold service is not always stable. If you see error, try again!\n")
    print("We show results from a previous successful run:\n")
    p=Protien(pwd / "example_files/esmfold.pdb")
    p.show(color="b")
```

```
2.4554355609575724
```



```
p=Protein(fn)
exp=p.extract("H:L:P")
try:
    pred=Protein.fold(exp.seq())
    # algin by Ab only
    # predicted chains are A, B, C
    pred.rename_chains({"A":"H", "B":"L", "C":"P"})
    pred.align(exp, "H:L", "H:L", ats="N,CA,C,O")
    # print RMSD of antigens
    print(pred.rmsd(exp, "P", "P", ats="N,CA,C,O"))
    Protein.merge([exp, pred]).show(color="chain")
except Exception as e:
    print(e)
    print("ESMFold service is not always stable. If you see error, try again!")
```

```
41.00484574766786
```



We can see that ESMFold predicts the antibody structure well, however, it got the antigen binding mode wrong. Instead of binding to the CDR loops, it places antigen on the other end. Ab-Ag complex prediction is an extremely difficult problem, the prediction accuracy was improved in the latest AlphaFold3 model.

Create Side Chains for de novo Designed Proteins

In the current AI-based protein design, RFDiffusion only generates a protein backbone. The output PDB file only contains coordinates for N, CA, C, O atoms. Often, RFDiffusion works on an input template and was used to design only part of the structure (known as inpainting). For example, in an antibody design

application, RFDiffusion may be only used to redesign the CDR loops, while leaving the framework residues and structures untouched. In the example below, we use RFDiffusion to design CDR H3 loops with the rest of the antibody sequence fixed. For convenience, let us refer to those fixed residues approximately as framework residues and the CDR H3 loop as CDR residues without creating confusion.

In the output PDB file generated by RFDiffusion, no sidechain atom exists, even for those framework residues. The generated CDR H3 backbone are represented by glycine residues. ProteinMPNN will preserve the identities of the input framework residues, while proposing CDR H3 residues to replace those glycines. In order to visualize the full-atom model of the design antibody-antigen complex structure, we can use AlphaFold to predict the structure from the ProteinMPNN sequences. However, AlphaFold prediction is time consuming, and the predicted structure can be very different from the desired PDB structure due to the limited prediction accuracy on Ab-Ag complexes.

The method `thread_sequence()` helps us rapidly create a full-atom structure by threading the ProteinMPNN-generated sequence onto the RFDiffusion backbone-only structure. When the ProteinMPNN sequence is threaded onto the PDB backbone template, the side chain coordinates of those framework residues are now generated by PyMOL, which are sampled from their most frequent torsion angles and can be very different from their original known coordinates. For this reason, `thread_sequence()` takes an additional argument side_chain_pdb, which specifies the original PDB file that contains the side chain atoms for the framework residues (this is the PDB file used as the input for RFDiffusion). With this additonal input, the side chain coordinates for framework residues in the original PDB structure will be used instead of relying on PyMOL generation.
Argument seq2bfactor=True maps the upper/lower-case of the input sequence onto b-factors (1.0/0.5), so that we can use b-factor to distinguish the redesigned residues (lower case) from those framework residues (upper case) preserved from the input template.

There are details inside the method that make use of several other **Afpdb** methods. For example:

The predicted structure needs to be aligned with the side_chain_pdb first, before we are able to clone the coordinates of the side chain atoms. This is because the original PDB structure and the RFDiffusion-generated structure are not aligned by default.

When RFDiffusion generates its output, chains are named A, B, C, etc. with the original chain names lost. As RFDiffusion can hallucinate new chains that do not exist in the original structure, it may not be able to preserve the original chain names even if it likes to. For this reason, we also need to provide a chain_map argument, so that the threading code knows how to map the wild-type chain names into the RFDiffusion chain names in order to align them correctly.

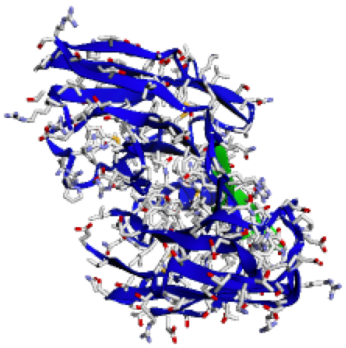
Side chain coordinate cloning for the framework residues were done by manipulating the backend NumPy arrays.

```
# we first mimic a backbone-only structure, by changing all residues to Gly
q=Protein(pwd / "example_files/5cil_rfdiffuse_H3.pdb")
# rs_missing_atoms returns all residues where their sidechain atoms are missing
no_sc=q.rs_missing_atoms()
print("All non-Gly residues have missing side-chain atoms:", util.unique(no_sc.seq()), "\n")
```

```
All non-Gly residues have missing side-chain atoms: ['T', 'E', 'R', 'N', 'K', 'Y', 'F', 'P', 'V',
'Q', 'S', 'D', 'W', 'A', 'I', 'M', 'L', 'C']
```

```
seq_MPNN={
'A':'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQY
GQSLSTFGQGTKVEVKRTV', 'B':'NWFDITNWLWYIK',
'C':'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAV
YYCARhlvrtvgsgsnpemgdvvWGQGTLVTVSS'}
# we thread seq_MPNN onto RFDiffusion output structure q
# output a new PDB file: test.pdb
# copy sidechains of framework residues from template fn
# map the A/B/C chain names in the RFDiffusion structure to the L/P/H chains in the template fn
```

```
# seq2bfactor=True means we assign b-factor of value 0.5 for lower-cased seq_MPNN residues (those
are CDR H3 residues redesigned)
# framework residues (upper case) have b-factor of value 1.0.
q.thread_sequence(seq_MPNN, 'test.pdb', seq2bfactor=True, side_chain_pdb=fn, chain_map={"H":"C",
"L":"A", "P":"B"})

###JSON STARTS
{"output_pdb": "test.pdb", "ok": true, "output_equal_target": true, "input": {"A":
"EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSL
STFGQGTKVEVKRTV", "B": "NWFDITNWLWYIK", "C":
"VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCA
RGGGGGGGGGGGGGGGGGGWGQGTLVTVSS"}, "output": {"A":
"EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSL
STFGQGTKVEVKRTV", "B": "NWFDITNWLWYIK", "C":
"VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCA
RHLVRTVGSGSNPEMGDVVWGQGTLVTVSS"}, "target": {"A":
"EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSL
STFGQGTKVEVKRTV", "B": "NWFDITNWLWYIK", "C":
"VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCA
RhlvrtvgsgsnpemgdvvWGQGTLVTVSS"}, "relax": false, "residues_with_missing_atom": "", "mutations":
[["C", 98, "GLY", "HIS"], ["C", 99, "GLY", "LEU"], ["C", 100, "GLY", "VAL"], ["C", 101, "GLY",
"ARG"], ["C", 102, "GLY", "THR"], ["C", 103, "GLY", "VAL"], ["C", 105, "GLY", "SER"], ["C", 107,
"GLY", "SER"], ["C", 108, "GLY", "ASN"], ["C", 109, "GLY", "PRO"], ["C", 110, "GLY", "GLU"],
["C", 111, "GLY", "MET"], ["C", 113, "GLY", "ASP"], ["C", 114, "GLY", "VAL"], ["C", 115, "GLY",
"VAL"]]}
###JSON END

q=Protein("test.pdb")
print("\nResidues with missing atoms:", q.rs_missing_atoms(), "\n")

Residues with missing atoms:

q.show(color="b", show_sidechains=True)
```



In the RFDiffusion output file 5cil_rfdiffuse_H3.pdb, all 250 residues have missing atoms due to their missing side chains. After threading using the wild-type PDB 5cil.pdb as the template, we copy the side chain coordinates for all upper-case residues except the lower-case residues in the H3 CDR loop. The side chain atoms for the H3 loop are generated by PyMOL by replacing Glycine with corresponding ProteinMPNN-generated residues. At the end, the final output test.pdb contains all atoms, therefore, no residue has missing atoms. The framework residues are colored in blue and the redesigned CDR H3 loop is colored in green.

Binding Score for EvoPro

EvoPro is a de novo protein design framework that iteratively evolutes protein binder sequences (https://doi.org/10.1073/pnas.2307371120). EvoPro starts from a pool of binder sequences; it uses AlphaFold to predict complex structures and then ranks the candidates by their EvoPro scores. A new canidate pool is generated with genetic algorithm based on a few top-scoring sequences. This process iterates until sequences of good scores are found.

We here demonstrate how EvoPro score, as described in the publication can be implemented straightforwardly with Afpdb. The score is the sum of three scores. The first is the placement confidence score that is the total number of interface residue pairs, weighted by AlphaFold PAE scores. The second is the fold confidence score based on AlphaFold's pLDDT scores. The third is the conformal stability score based on the RMSD between the binder structures as a monomer and as the binder of the complex.

The implementation is shown below:

```
# We score 5cli
# Ab-Ag predicted complex structure
p_complex = Protein(pwd / "example_files/5cil_AF.pdb")
p_complex.rename_chains({"A":"L", "B":"H", "C":"P"})
# load the PAE score generated by AF, PAE is the pairwise uncertainty in inter-residue distance
import json
pae = np.array(json.loads(util.read_string(pwd / "example_files/5cil_AF.json"))['pae'])
# AF uses b-factors to store pLDDT
plddt=p_complex.b_factors()
# Antibody predicted as a monomer
p_ab = Protein(pwd / "example_files/5cil_ab_AF.pdb")
p_ab.rename_chains({"A":"L", "B":"H"})

# Compute placement confidence
# identify all contact residue pairs
rs_ab, rs_ag, dist = p_complex.rs_around("P", rs_within="H:L", dist=4)
# extract PAE for contact residue pairs
# each row in dist DataFrame is one residue pair, they are weighted by PAE
# if PAE is small, we are more confident that the contact pair is real
pae_xy=pae[dist.resi_a.values, dist.resi_b.values]
pae_yx=pae[dist.resi_b.values, dist.resi_a.values]
# add negative, b/c the lower the score, the better
placement_score = -np.sum(((35-pae_xy) + (35-pae_yx))/70)

# Compute fold confidence
# pLDDT score is [0-100], scale it by ten
fold_score = -np.mean(plddt)/10

# Compute conformational stability score
# The argument is if the binder does not change its conformation after binding,
# it is more likely to bind
stability_score = p_ab.rmsd(p_complex, "H:L", "H:L", align=True)*5

# the lower the better
overall_fitness = placement_score + fold_score + stability_score

print(f"Fitness: {overall_fitness:.2f}")
print(f"Placement: {placement_score:.2f}")
print(f"Fold: {fold_score:.2f}")
print(f"Stability: {stability_score:.2f}")
```

```
Fitness: -39.96
Placement: -31.51
Fold: -9.73
Stability: 1.28
```

Developer's Note

Installation

TODO: When the package is considered stable, we need to publish afpdb into pypi, it then simply "pip install afpdb".

The instructions below are for users who would like to install Afpdb locally and for developers.

Python

If you do not have Python installed, follow the instructions on https://docs.anaconda.com/free/miniconda/ to install the latest `miniconda`. Type command `python` should launch the Python programming shell, if it is installed successfully.

Install Afpdb

```
pip install git+https://github.com/data2code/afpdb.git
```

Jupyter Notebook (optional)

To view and run this tutorial, Jupyter should be installed:

```
pip install notebook
```

Type command `jupyter notebook` to launch the Jupyter Notebook, if it is installed successfully.

This is no longer needed. However, if the embedded protein structures do not display in Jupyter after rerun the cell, install the required plugin:

```
jupyter labextension install jupyterlab_3dmol
```

PyMOL (optional)

PyMOL is the preferred application for visualizing protein structures. It is required by examples using `thread_sequence()` or `PyMOL()`. To install the open source PyMOL:

```
conda install conda-forge::pymol-open-source
```

In Colab, we also need to run:

```
conda install conda-forge::openssl=3.2.0
```

DSSP (optional)

Required for the secondary structure assignment with method dssp().

```
conda install sbl::dssp
```

There are multiple options, `sbl::dssp` suits Apple Silicon.

matplotlib (optional)

Required for the Ramachandra plot example

```
pip install matplotlib
```

Install `pytest` as a developer:

```
pip install pytest
```

Type command `pytest` within the root folder of the Afpdb package, you will run all test examples in `tests\test_all.py`.

For developers, after we fixed the bugs and passed `pytest`, we run `pip install .` to update the package under the `conda` installation.

Selection

When creating a method that takes a selection argument named `'rs'`, the first step is to convert it into an internal selection object using: `rs = self.rs(rs)`, this will convert the argument into a RS object, which has its data member storing the residue indices. Similarly, if we have an atom selection argument named `'ats'`, do `ats=self.ats(ats)`. Similarly, if we take a residue list object, we do `rl=self.rl(rl)`. When we use a residue/atom selection to index `atom_positions` or `atom_mask`, check if the selection is empty/full with `ats.is_empty() and ats.is_full()`. Empty selection often implies an error on the users' side, a full selection means you can skip the indexing, as the original array is already good.

Please use `extract()` as an example to see how we support selection arguments.

Change in residue/chain

The Protein class contains a data structure called `res_map`, which is a dictionary that maps a full residue name `"{chain}{residue_id}{code}"` into its internal ndarray index. A few methods rely on this mapping. Therefore, whenever a method renames a chain, changes chain orders, mutates a residue, or changes the full residue name and its internal index, `self._make_res_map()` should be called at the end. This is also needed in `extract()` as the underlying arrays have been changed.

Residue Identifier

When outputting a dataframe containing a residue, our recommendation is to provide all residue ID formats. This includes chain, `resn`, `resn_i`, `resi`. Please use rs_dist as an example. We often use the resi column to create a Residue List object, then use its name, namei, chain, aa methods to add additional residue annotation data. See the example under `rs_dist()`.

inplace

To support `inplace`, the idiom is to use: `obj = self if inplace else self.clone()`, then use obj to manipulate the structure. Please set `inplace=False` as the default, so that users do not have to memorize what the default is.

Extract Atom Coordinates

p.data.atom_positions contains non-existent atoms. It is often faster to compute distances between two residue sets, if we only keep the coordinates for real atoms. This is done with _get_xyz() method, which returns three variables: (`residue_indices`, `atom_indices`, `XYZ_array`).

```
In [2]:
p=Protein(fk)
rs_i, atom_i, xyz=p._get_xyz(p.rs("H"), p.ats("N,CA"))
print("Residue ID:", rs_i, p.rl(rs_i).name(), "\n")
```

```
print("Atom ID:", atom_i, [str(p.ats(x)) for x in atom_i], "\n")
print("XYZ:", xyz)
```

```
Residue ID: [4 4 5 5] ['3', '3', '4', '4']
Atom ID: [0 1 0 1] ['N', 'CA', 'N', 'CA']
XYZ: [[ 27.36800003   6.44000006 -19.10700035]
 [ 25.96999931   6.87099981 -19.03800011]
 [ 25.29100037   9.00800037 -18.09000015]
 [ 25.11199951   9.9829998  -16.98600006]]
```

Note: To extract a rectangular subarray of rows and columns, we need to use np.ix_.

```
p=Protein(fk)
# the followin is an error, as the row indice have two residues, column indices have 4 atoms
# NumPy tries to pair the indices
try:
    p.data.atom_mask[np.array([2,3]), ATS("N,CA,C,O,CB,CG").data]
except Exception as e:
    print(e)
```

```
shape mismatch: indexing arrays could not be broadcast together with shapes (2,) (6,)
```

```
# The correct way is to generate a mesh indices
print("\n")
x=p.data.atom_mask[np.ix_(np.array([2,3]), ATS("N,CA,C,O,CB,CG").data)]
print(x.shape, "\na", x, "\n")
```

```
(2, 6)
a [[1. 1. 1. 1. 1. 0.]
 [1. 1. 1. 1. 1. 1.]]
```

```
# or
print(p.data.atom_mask[np.array([2,3])][:, ATS("N,CA,C,O,CB,CG").data])
```

```
[[1. 1. 1. 1. 1. 0.]
 [1. 1. 1. 1. 1. 1.]]
```

Extract Atom Pair Coordinates

For align and rmsd, we need to extract atom coordinates in pairs, we can use `_get_xyz_pair`.

Note: If two residues have different types (their side chain atoms are different), only the common atoms are included.

```
p=Protein(fk)
# move X by 1, Y/Z remains the same
q=p.translate(np.array([1,0,-1]), inplace=False)
rs_i, atom_i, rs_j, atom_j, xyz_i, xyz_j=p._get_xyz_pair(q, p.rs("H"), q.rs("H"), ATS("N,CA"))
print("Protein i\n")
print("Residue ID:", rs_i, p.rl(rs_i).name(), "\n")
print("Atom ID:", atom_i, [str(p.ats(x)) for x in atom_i], "\n")
print("XYZ:", xyz_i)
```

```
Protein i
Residue ID: [4 4 5 5] ['3', '3', '4', '4']
Atom ID: [0 1 0 1] ['N', 'CA', 'N', 'CA']
XYZ: [[ 27.36800003   6.44000006 -19.10700035]
 [ 25.96999931   6.87099981 -19.03800011]
 [ 25.29100037   9.00800037 -18.09000015]
 [ 25.11199951   9.9829998  -16.98600006]]
```

```
print("\n\n")
print("Protein j\n")
print("Residue ID:", rs_j, p.rl(rs_j).name(), "\n")
print("Atom ID:", atom_j, [str(p.ats(x)) for x in atom_j], "\n")
print("XYZ:", xyz_j)
```

```
Protein j
```

```
Residue ID: [4 4 5 5] ['3', '3', '4', '4']
Atom ID: [0 1 0 1] ['N', 'CA', 'N', 'CA']
XYZ: [[ 28.36800003   6.44000006 -20.10700035]
 [ 26.96999931   6.87099981 -20.03800011]
 [ 26.29100037   9.00800037 -19.09000015]
 [ 26.11199951   9.9829998  -17.98600006]]
```

Caution

When we add a new method, please keep in mind that the residue index may not start from 1, a residue index may contain insertion code, there can be gaps in the residue index (missing residues), the integer part of the residue index may not be unique within a chain (e.g. 6A and 6B). You should use the file "fk" to test your method. Please also add a corresponding test method into `tests/test_all.py`.