afpdb - a numpy-based PDB structure manipulation library

The aim for **afpdb** is to help us efficiently manipulate protein structures in Python. Our aim is to combine the convenience found in PyMOL's selection concept and the computational power found in BioPython. Implementing **afpdb** is based on DeepMind's PDB structure representation, which is used in their AlphaFold application. We found DeepMind's numpy-centric implementation of a Protein structure class to be significantly more convenient than other similar Python libraries. We, therefore, tweaked DeepMind's code and developed a wrapper class also called Protein. Our Protein class contains a data member `data_prt`, which points to a DeepMind's Protein object. 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 and P

chain for the antigen. fake.pdb contains two short chains, the L chain contains 4 residues including two with insertion code and 3 missing residues, and the H chain contains two residues.

If you are not using our Linux environment, run the following first (however, you cannot run this notebook):

`source /da/NBC/ds/bin/bashrc.ci`

If you want to install afpdb into your own Python environment, the installation package is at `/da/NBC/ds/zhoyyi1/afpdb`. However, some measurement and visualization methods may not work due to the dependencies on other 3rd party packages.

We also assume you have always run the following Python codes at the beginning of your session: (If you install afpdb, use "import afpdb.util as util".)

This document can be found in `/da/NBC/ds/zhoyyi1/afpdb/jupyter`.

In [1]:
```
%reload_ext autoreload
%autoreload 2

import afpdb
if afpdb.__file__.startswith("/da/NBC/ds"):
    import util
    import myalphafold.common.residue_constants as afres
    import myalphafold.common.protein as afprt
else:
    import afpdb.util as util
    import afpdb.myalphafold.common.residue_constants as afres
    import afpdb.myalphafold.common.protein as afprt
from afpdb import Protein
import numpy as np
fn = "/da/NBC/ds/lib/example_files/5cil.pdb"
fk = "/da/NBC/ds/lib/example_files/fake.pdb"
```

Demo

To convince you afpdb is worth learning, let us look at the following example, where we find binding residues around antigen chain P. We extract all the residues involved in the binding and save them into a separate PDB file. The structure display confirms the extraction was correct. These manipulations are not so straightforward using PyMOL or BioPython.

In [2]:
```
# load PDB file
p=Protein(fn)

# show sequence
print(p.seq_dict(), "\n")

# identify H,L chain residues within 4A to antigen P chain
binders, df_dist=p.rs_around("P", dist=4)
```

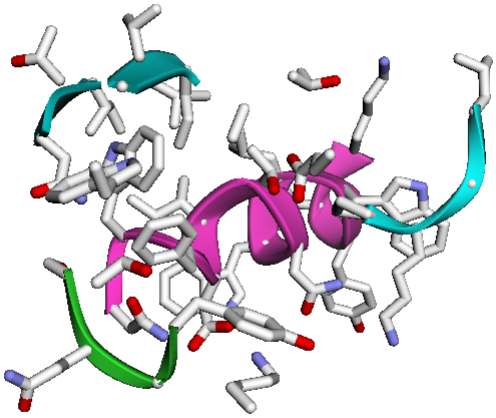
```
# show the distance of binder residues to antigen P chain
df_dist.display()

# create a new PDB file only containing the antigen and binder residues
p=p.extract_by_contig(p.rs_or(binders, "P"))
p.save("binders.pdb")

p.show(show_sidechains=True)
```

{'L': 'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTIS
RLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV', 'H': 'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPG
RGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVS
S', 'P': 'NWFDITNWLWYIK'}

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
P	6	6	242	T	H	98	98	208	E	2.63625	OG1	OE2
P	4	4	240	D	L	33	33	32	K	2.81482	OD1	NZ
P	2	2	238	W	L	95	95	94	S	2.91194	N	OG
P	1	1	237	N	L	92	92	91	Y	2.9295	ND2	O
P	7	7	243	N	H	108	108	218	K	3.03857	ND2	CE
P	1	1	237	N	L	94	94	93	Q	3.08749	OD1	C
P	2	2	238	W	H	56	56	166	I	3.23098	NE1	O
P	1	1	237	N	L	93	93	92	G	3.2578	ND2	C
P	10	10	246	W	H	106	106	216	L	3.30522	NE1	O
P	2	2	238	W	H	58	58	168	N	3.32092	CE3	ND2
P	6	6	242	T	H	30	30	140	T	3.37051	CG2	O
P	10	10	246	W	H	107	107	217	G	3.4175	NE1	C
P	7	7	243	N	H	109	109	219	P	3.47407	N	CG
P	2	2	238	W	H	50	50	160	V	3.50075	CZ2	N
P	2	2	238	W	H	49	49	159	G	3.51053	CZ2	C
P	3	3	239	F	L	97	97	96	S	3.54476	CZ	CB
P	5	5	241	I	H	51	51	161	I	3.58825	CG2	CD1
P	9	9	245	L	H	54	54	164	L	3.70006	CD1	CD1
P	6	6	242	T	H	32	32	142	A	3.78418	CG2	N
P	3	3	239	F	H	46	46	156	W	3.85663	CZ	CZ2
P	2	2	238	W	H	57	57	167	T	3.99331	NE1	C



Internal Data Structure

Data Members

To better understand analysis methods provided by afpdb, it is helpful to first discuss how the data in a PDB file is stored within the DeepMind's Protein object (modified by us). 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. The maximum residue id can be 9999 if we allow insertion codes with two letters, otherwise, 99999 if all insertion codes are single letters.

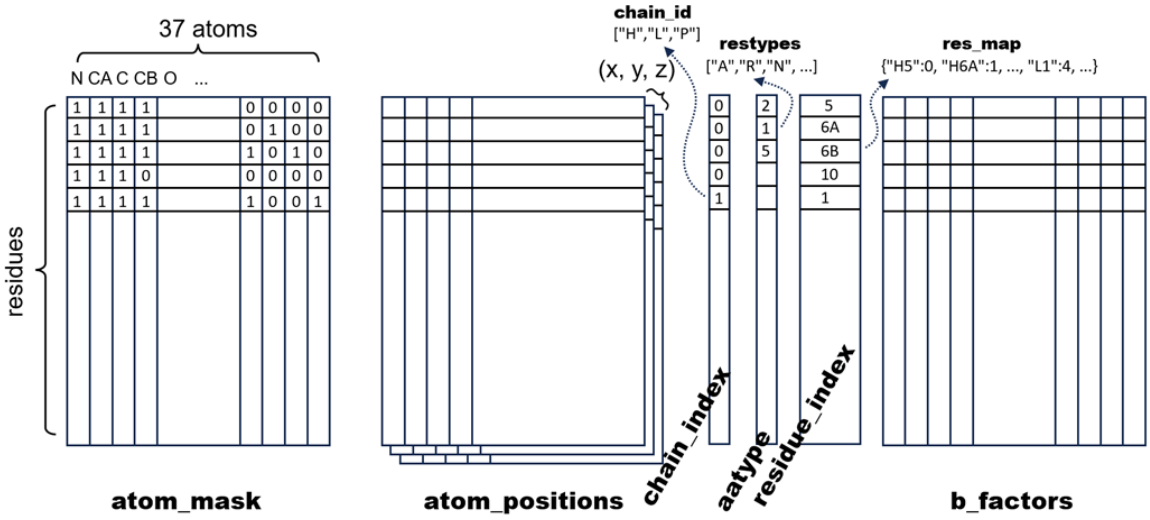
```
  # 0-indexed number corresponding to the chain in the protein that
this residue
  # belongs to.
  chain_index: np.ndarray  # [num_res]

  # 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. We decided to keep the chain
names in the PDB file
  chain_id: np.ndarray # [num_chains]
```

We also added a dictionary res_map, which maps the full residue id, in the format of {chain}{residue_id}{code}, into its residue_index.

Illustration



Notice the first axis of all ndarrays are the number of residues. The second axis can be either one or the number of possible atom types (37).

Example

Let us exam these data structures using fake.pdb example. 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 use fake.pdb to better understand the internal data elements. We first read in fake.pdb, there is a warning indicating there are residues with insertion codes: 6A and 6B. This alerts

the user that there may be duplicates in the integer portion of the residue ID, so the user should not rely on the assumption that integer IDs are all unique.

DeepMind's original code did not support insertion codes. We have made modifications to eliminate this limitation.

In [3]:
```
p=Protein(fk)
```

```
Warning: residues with insertion code: L6A, L6B
```

Our protein object p contains a data member data_prt, which points to AlphaFold's protein object g. Let us examine the data members in g.

In [4]:
```
g=p.data_prt
g.residue_index
```

Out[4]:
```
array(['5', '6A', '6B', '10', '3', '4'], dtype='<U6')
```

In this tutorial, **resn** refers to the residue_index, a string with an optional insertion code. **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. A protein object contains a data member p. `res_map` , which can map **resn** into **resi**.

In [5]:
```
print(g.chain_index, "\n")
print(g.chain_id, "\n")
print(p.chain_id())
```

```
[0 0 0 0 1 1]

['L', 'H']

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

Chain_index shows the first 4 residues belong to chain index 1, and the next 2 residues belong to chain index 0. 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 relying on the assumption that 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? DeepMind's code did not store chain names and chains were named as A, B, C, etc. We have patched the DeepMind's code to add a new data member `chain_id` into DeepMind's Protein class, we also added a method `chain_id()` to afpdb's Protein class.

We have a total of 6 residues, three missing residues (H7-9) are not found in the PDB file, therefore, we do not store them in the numpy arrays. Missing residues can only be inferred based on the gaps in the integer part of the residue IDs.

In [6]:
```
print(g.aatype, "\n")
[ afres.restypes[x] for x in g.aatype ]
```

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

Out[6]:
```
['E', 'I', 'V', 'Q', 'L', 'V']
```

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 37 possible atom types when all 20 amino acid types are combined (not counting hydrogens), DeepMind uses an array of 6 x 37 to store atom 3D coordinates. Therefore, the atom_positions is a 3-dimensional numpy array, where the last dimension is for (X, Y, Z) coordinates:

In [7]:
```
g.atom_positions.shape
```

Out[7]:
```
(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, DeepMind's code uses 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 complicated hierarchical navigation along of chain/residue/atom tree is needed compared to the BioPython API.

We thus need to know what elements in the atom_positions array contain real data using a binary array atom_mask:

In [8]:
```
print(g.atom_mask[:, afres.atom_order['CA']], "\n")
print(g.atom_positions[:, afres.atom_order['CA']], "\n")
print(g.atom_mask[:, afres.atom_order['CG']], "\n")
print(g.atom_positions[:, afres.atom_order['CG']], "\n")
```

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

[[  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]]

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

[[  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.        ]]
```

We see that all six residues have CA (C_α) atom, but only 1st, 4th, and 5th residues (E, Q, L) have CG (C_γ). So the zero coordinates for those atoms with mask 0 should be ignored (these atoms are not present in the structure). Although atoms with coordinates at the origin are likely masked-out atoms, we should not rely on that assumption. In fact, when AlphaFold 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, instead of assuming their positions can be any numbers.

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

In [9]:
```
g.b_factors.shape
```

Out[9]:
```
(6, 37)
```

To summarize, we store all key PDB information into a few numpy arrays: `atom_positions, atom_mask, residue_index, chain_index, b_factors,` and `chain_id`. We also have a helper dictionary `res_map`.

These arrays make structure manipulations a breeze.

Contig

Based on how afpdb uses ndarray to store structure data, we can see to access a few residues and a few atoms is as easy as access certain rows and columns in these ndarrays. To specify what residues are of interest, we use a string format called **contig**.

Single Residue

`CHAIN ID` without space

In PDB, each residue is assigned an integer id. In some cases, a residue ID can be followed by an insertion code, which is an alphabet letter. 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 ID alone is not unique, because the same residue ID 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.

Single Fragment

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

To specify a continuous range of residues, we specify the start the end residues without the need to enumerate all residues inbetween. 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 from say 5 in the PDB file). H98- covers 98 to the last residue. H98-100A does not include 100B. H98-100B includes both 100A and 100B.

A chain can have missing residues, e.g., the PDB structure may contain residues 92, 93, 98, 99, 100A, 100B, and 101, where 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 it includes missing residues.

Multiple Fragments

To specify multiple fragments within the same chain:

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

Chain letter should only appears 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.

```
CONTIG = IN-CHAIN FRAGMENTS[:[IN-CHAIN FRAGMENTS]]*
```

We can specify multiple 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 in-chain fragments using the general contig syntax, e.g., H-5,10-15:L-10 is equivalent to H-5:H10-15:L-10.

Note

If there are multiple fragments within a chain, we should try to keep them together, so that these segments are not separated by segments from other chains. We do not reorder the underlying residue selection, as method such as `align` might be sensitive to the order. Some methods may not work as expected if the assumption is broken, as they might rely on a method called `chain_pos`, which assumes all residues colocated on the same chain appear sequentially. Do worry too much as the code does check and print a warning if the input contig breaks this assumption. In that case, you can canonicalize a selection, which is described later.

inplace

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

In [10]:
```
p=Protein(fn)
# inplace=False by default
q=p.extract_by_contig("H:L")
print(p.chain_id(), q.chain_id(), p==q, "\n")

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

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

['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 make a clone:

In [11]:
```
p=Protein(fn)
print(p.chain_id(), "\n")
q=p.clone()
# we modify q in place
q.extract_by_contig("P", inplace=True)
# q is modified
print(q.chain_id(), "\n")
# p is untouched
print(p.chain_id())
```

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

['P']

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

Read/Write

The convenience of afpdb can be demonstrated by how we read/write protein structures.

In [12]:
```
p=Protein(fn)
print(p.chain_id())
p.save("test.pdb")
out=util.unix("ls -l test.pdb")
```

```
['L', 'H', 'P']
-rw-rw-r--. 1 zhoyyi1 ux_nbc_ds 155439 Nov 25 10:23 test.pdb
```

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

In [13]:
```
p=Protein(fn, contig="H:L")
print(p.chain_id(), "\n")
p=Protein(fn, contig="H-5:L-10")
print(p.seq())
```

```
['H' 'L']

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.

Below are a few examples:

In [14]:
```
print("Load a local PDB file")
# support both .pdb and .ent extensions
print(fn)
p=Protein(fn)
print(p.chain_id(), "\n")

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

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

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())
```

```
Load a local PDB file
/da/NBC/ds/lib/example_files/5cil.pdb
['L', 'H', 'P']

Create Protein from another afpdb.Protein object
['L', 'H', 'P']

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

Create Protein from a str containing the content of a PDB file
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
ATOM      4  CB  GLU L   1       5.794 -13.368 -17.248  1.00  1.00           C
...
['L', 'H', 'P']
```

In [15]:
```
print("Save as .cif, save Protein into a .cif file\n")
p=Protein(fn)
p.save("test.cif")
print("Create Protein from a local .cif file")
p=Protein("test.cif")
print(p.chain_id(), "\n")
print("Create with a PDB 4-letter code")
p=Protein("1crn")
print(p.seq(), "\n")
print("Create with a EMBL AlphaFold model code")
p=Protein("Q2M403")
print(p.seq())
p.show()
```

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

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

Create with a PDB 4-letter code
TTCCPSIVARSNFNVCRLPGTPEAICATYTGCIIIPGATCPGDYAN

Create with a EMBL AlphaFold model code
MVVVSLQCAIVGQAGSSFDVEIDDGAKVSKLKDAIKAKNATTITGDAKDLQLFLAKQPVEDESGKEVVPVYRPSAEEMKEESFK
WLPDEHRAALKLVEGESDDYIHALTAGEPILGSKTLTTWFYTKNNMELPSSEQIHVLVVVPDGAGGSASDTSRMDRLFDKVDKV
YEHTVLSKRTRYVHSEMNSAKGNILLNDLKIRISPVDTVKFAGGVPTPAKEFKWKSDRTEEQQKEPYREYVVANIGDVLTNNKL
CVVGVEKGANILTVEVPGRDIVLAGRTDMIVLSDIAQKFPHYLPHLPGVRMLIEVKKVVTTASEFQALSELIALDIIVTESVMA
LLTNLTNHWQFFWVSRKSDDRVIIETTTLIAPGEAFAVIRTLLDQSPSAGAEVSLPCFEKPVKRQKLSQLLPSISEASGSSGIR
ESIERYYDIASMLGPDLEMARAVASQVARSIPTLSYFS
```

Sequence

Extraction

In [16]:
```
p=Protein(fn)
p.seq()
```

Out[16]:
```
'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEP
EDFAVYYCQQYGQSLSTFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVI
PLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS:NWFDIT
NWLWYIK'
```

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

EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDF
TFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTIT
NSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS:NWFDITNWLWYIK

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

Here the order of the chain sequences is the same as how they occur in the PDB file. However, when structures are manipulated by apps, such as PyMOL, the chain order may be changed. The chain order may also 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:

In [17]:
```
p.seq_dict()
```

Out[17]:
```
{'L': 'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLT
ISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV',
 'H': 'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRST
STAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS',
 'P': 'NWFDITNWLWYIK'}
```

Missing Residues

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

In [18]:
```
q=p.extract_by_contig("H1-10:H21-30")
print(q.data_prt.residue_index, "\n")
print("Number of residues in numyp arrays:", q.data_prt.atom_positions.shape, "\n")
print(q.seq(), "\n")
print(q.seq_dict(gap="."), "\n")
```

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

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

VQLVQSGAEVXXXXXXXXXXCKASGGSFST

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

The number of missing residues is determined based on residue numberng.n We first create a new object by taking the first tenresiduese from chain H, skipping 10 residues, thentakinge 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 residueshavesnot corresponding elements in the internal numpy array (the array only contains 20 residue in totalss, the discontinuity in the residue index allows us to figure out that10n residues are missing. Therefore, seq() and seq_dict()outputs ten "X". You can change the display of "X" by the gap argumen or suppress gap by ""t.

In [19]:
```
p=Protein(fk)
print(p.data_prt.residue_index, "\n")
print(p.seq_dict(), "\n")
print(p.seq_dict(gap=""))
```

```
Warning: residues with insertion code: L6A, L6B
['5' '6A' '6B' '10' '3' '4']

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

{'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.

Length

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

In [20]:
```
p=Protein(fn)
print("Total residue counts:", len(p), "\n")
print("Residue counts per chains:", p.len_dict())
```

```
Total residue counts: 250

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

Search

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

In [21]:
```
p=Protein(fn)
contig=p.search_seq('LTI')
print(contig, "\n")
print(p.extract_by_contig(contig).seq(), "\n")
```

```
L74-76:H54-56

LTI:LTI
```

If returns a contig that matches all occurrences. The matches are in bold below:
EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDF
VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPL***LTI***TNYAPRFQGRITITAD
NWFDITNWLWYIK

Mutagenesis

To perform mutagenesis on a structure, we thread a new sequence onto an existing Protein object.

In [22]:
```
p=Protein(fk)
p.thread_sequence({"H":"LG", "L":"AIVD"}, "m.pdb", relax=0, cores=1)
```

```
Warning: residues with insertion code: L6A, L6B
Warning: residues with insertion code: L6A, L6B
MUTATE PyMOL>  L 5 GLU > ALA
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.047 (4 to 4 atoms)
 Mutagenesis: no rotamers found in library.
MUTATE PyMOL>  L 10 GLN > ASP
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.027 (4 to 4 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 9 rotamers loaded.
 Rotamer 9/9, strain=2.82
MUTATE PyMOL>  H 4 VAL > GLY
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.023 (3 to 3 atoms)
 Mutagenesis: no rotamers found in library.
m.pdb
Warning: residues with insertion code: L6A, L6B
{'H': 'LG', 'L': 'AIVD'}
Warning: residues with insertion code: L6A, L6B
###JSON STARTS
{"output_pdb": "m.pdb", "ok": true, "output_equal_target": false, "input": {"L": "EI
VQ", "H": "LV"}, "output": {"H": "LG", "L": "AIVXXXD"}, "target": {"H": "LG", "L":
"AIVD"}, "relax": false, "residues_with_missing_atom": {}, "mutations": [["L", 5, "G
LU", "ALA"], ["L", 10, "GLN", "ASP"], ["H", 4, "VAL", "GLY"]]}
###JSON END
```

Out[22]:
```
{'output_pdb': 'm.pdb',
 'ok': True,
 'output_equal_target': False,
 'input': {'L': 'EIVQ', 'H': 'LV'},
 'output': {'H': 'LG', 'L': 'AIVXXXD'},
 'target': {'H': 'LG', 'L': 'AIVD'},
 'relax': False,
 'residues_with_missing_atom': {},
 'mutations': [('L', 5, 'GLU', 'ALA'),
  ('L', 10, 'GLN', 'ASP'),
  ('H', 4, 'VAL', 'GLY')]}
```

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.

From the output JSON data, we see L5 was "E" and it was replaced with "A". L6A-L6B was not changed, therefore, their side chain coordinates remain the same. L10 was replaced by "D". On the chain H, H6 was "G" and was 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 with Amber. This is accomplished by providing `relax = 1`.

The method `thread_sequence()` is often used to create a full-atom structure for RFDiffusion-based protein design. As RFDiffusion only generates a protein backbone, it uses Glycine to represent all residues in the output PDB file, where there are no side chain atoms. ProteinMPNN then generates a protein sequence based on the backbone structure. To generate a full-atom structure, we thread the ProteinMPNN sequence onto the RFDiffusion structure.

When we use RFDiffusion to design a partial structure, e.g., CDR loops with the antibody framework fixed. The PDB output from RFDiffusion contains the original framework residues, with the CDR residues replaced by Gly. Unfortunately, the PDB file only contains backbone atoms, and the framework side chain atoms are lost. Therefore, to thread the whole sequence onto the RFDiffusion PDB structure, we need to provide the wild-type structure that contains the framework side chain coordinates.

Two other arguments are needed in this case. `side_chain_pdb` specifies the original wild-type PDB file, so that the computer can align this original structure onto the RFDiffusion output, and then copy the side chain coordinates for framework residues. When RFDiffusion generates its output, chains are named A, B, C, etc. without preserving the original chain name. This is understandable, 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 `chain_map`, 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.

In [23]:
```
q=Protein("/da/NBC/ds/lib/example_files/5cil_rfdiffuse_H2.pdb")
print(q.residues_with_missing_atoms(), "\n")
seq_MPNN={
          'A':'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGS
          'B':'NWFDITNWLWYIK',
          'C':'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRIT
        }
q.thread_sequence(seq_MPNN, 'test.pdb', relax=0, seq2bfactor=True, cores=8, side_ch
q=Protein("test.pdb")
print("\nResidues with missing atoms:", q.residues_with_missing_atoms(), "\n")
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 42, 43, 44, 45, 46, 47,
48, 49, 51, 52, 53, 54, 55, 56, 58, 59, 60, 61, 62, 63, 65, 67, 69, 70, 71, 72, 73,
74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 93, 94, 95,
96, 97, 98, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 11
5, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 131, 132, 1
33, 134, 135, 136, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 150, 151, 152,
153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 165, 167, 168, 169, 170, 173,
174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 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, 239, 241, 243, 244, 245,
246, 247, 248, 249]

RMSD after backbone alignment: 0.1905358799813354
Generated new input PDB:  /tmp/_THREADkjeg5imx.pdb
MUTATE PyMOL>  C 98 GLY > HIS
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 9 rotamers loaded.
 Rotamer 3/9, strain=9.52
MUTATE PyMOL>  C 99 GLY > LEU
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 9 rotamers loaded.
 Rotamer 6/9, strain=10.42
MUTATE PyMOL>  C 100 GLY > VAL
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 1/3, strain=27.81
MUTATE PyMOL>  C 101 GLY > ARG
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 81 rotamers loaded.
 Rotamer 39/81, strain=37.17
MUTATE PyMOL>  C 102 GLY > THR
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 3/3, strain=40.63
MUTATE PyMOL>  C 103 GLY > VAL
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
```

```
 Rotamer 3/3, strain=47.80
MUTATE PyMOL>  C 105 GLY > SER
PyMOL>refresh_wizard
Selected!
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 1/3, strain=122.20
MUTATE PyMOL>  C 107 GLY > SER
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 1/3, strain=78.86
MUTATE PyMOL>  C 108 GLY > ASN
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 18 rotamers loaded.
 Rotamer 13/18, strain=92.73
MUTATE PyMOL>  C 109 GLY > PRO
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.008 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 2 rotamers loaded.
 Rotamer 1/2, strain=74.34
MUTATE PyMOL>  C 110 GLY > GLU
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 27 rotamers loaded.
 Rotamer 17/27, strain=45.93
MUTATE PyMOL>  C 111 GLY > MET
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 27 rotamers loaded.
 Rotamer 5/27, strain=27.73
MUTATE PyMOL>  C 113 GLY > ASP
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 9 rotamers loaded.
 Rotamer 7/9, strain=44.32
MUTATE PyMOL>  C 114 GLY > VAL
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
```

```
 Rotamer 3/3, strain=20.24
MUTATE PyMOL>  C 115 GLY > VAL
PyMOL>refresh_wizard
Selected!
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 3/3, strain=15.03
test.pdb
{'A': 'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTIS
RLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV', 'B': 'NWFDITNWLWYIK', 'C': 'VQLVQSGAEVKRPGSSVTV
SCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCARhlvrtv
gsgsnpemgdvvWGQGTLVTVSS'}
###JSON STARTS
{"output_pdb": "test.pdb", "ok": true, "output_equal_target": true, "input": {"A":
"EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPED
FAVYYCQQYGQSLSTFGQGTKVEVKRTV", "B": "NWFDITNWLWYIK", "C": "VQLVQSGAEVKRPGSSVTVSCKASG
GSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCARGGGGGGGGGGGG
GGGGGGWGQGTLVTVSS"}, "output": {"A": "EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPR
LLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV", "B": "NWFDITNWLW
YIK", "C": "VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITAD
RSTSTAYLELNSLRPEDTAVYYCARHLVRTVGSGSNPEMGDVVWGQGTLVTVSS"}, "target": {"A": "EIVLTQSPG
TQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYG
QSLSTFGQGTKVEVKRTV", "B": "NWFDITNWLWYIK", "C": "VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSW
VRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCARhlvrtvgsgsnpemgdvvWGQG
TLVTVSS"}, "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", "SE
R"], ["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

Residues with missing atoms: []
```

In the RFDiffusion output file 5cil_rfdiffuse_H2.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 relative side chain coordinates for all upper-case residues except the lower-case residues in the H3 CDR loop. The side chains 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.

Chain

Chain Name and Order

In the PDB file, chains go from L, to H, to P. In our latest version, chain_id() will return chain names in the same order. In the older version, the chain_id() returned chain names alphabetically sorted. We therefore have a legacy method chain_list(), which always returns

chain names in the same order as how residues appear. chain_list() should not be needed going forward.

In [24]:
```
p=Protein(fn)
print(p.chain_id(), "\n")
print(p.data_prt.chain_index, "\n")

print(p.chain_list())
```

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

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

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

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

In [25]:
```
p=Protein(fn)
print("Original chain order:", p.chain_id(), "\n")
p.reorder_chains(["P","L","H"])
q=p.data_prt
print(p.seq(), "\n")
print(p.chain_id(), "\n")
print(q.chain_index, "\n")
print(q.residue_index, "\n")
print(p.chain_list())
```

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

NWFDITNWLWYIK:EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGT
DFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV:VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGR
GLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVSS

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

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

['1' '2' '3' '4' '5' '6' '7' '8' '9' '10' '11' '12' '13' '1' '2' '3' '4'
 '5' '6' '7' '8' '9' '10' '11' '12' '13' '14' '15' '16' '17' '18' '19'
 '20' '21' '22' '23' '24' '25' '26' '27' '28' '29' '30' '31' '32' '33'
 '34' '35' '36' '37' '38' '39' '40' '41' '42' '43' '44' '45' '46' '47'
 '48' '49' '50' '51' '52' '53' '54' '55' '56' '57' '58' '59' '60' '61'
 '62' '63' '64' '65' '66' '67' '68' '69' '70' '71' '72' '73' '74' '75'
 '76' '77' '78' '79' '80' '81' '82' '83' '84' '85' '86' '87' '88' '89'
 '90' '91' '92' '93' '94' '95' '96' '97' '98' '99' '100' '101' '102' '103'
 '104' '105' '106' '107' '108' '109' '110' '111' '1' '2' '3' '4' '5' '6'
 '7' '8' '9' '10' '11' '12' '13' '14' '15' '16' '17' '18' '19' '20' '21'
 '22' '23' '24' '25' '26' '27' '28' '29' '30' '31' '32' '33' '34' '35'
 '36' '37' '38' '39' '40' '41' '42' '43' '44' '45' '46' '47' '48' '49'
 '50' '51' '52' '53' '54' '55' '56' '57' '58' '59' '60' '61' '62' '63'
 '64' '65' '66' '67' '68' '69' '70' '71' '72' '73' '74' '75' '76' '77'
 '78' '79' '80' '81' '82' '83' '84' '85' '86' '87' '88' '89' '90' '91'
 '92' '93' '94' '95' '96' '97' '98' '99' '100' '101' '102' '103' '104'
 '105' '106' '107' '108' '109' '110' '111' '112' '113' '114' '115' '116'
 '117' '118' '119' '120' '121' '122' '123' '124' '125' '126']

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

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

Residue Index

The method chain_pos() is an important utility method. It returns the start and end ndarray 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.

In [26]:

```
p=Protein(fn)
print(p.chain_pos(), "\n")
q=p.data_prt
print(q.residue_index[108:114], "\n")
print(q.chain_index[108:114])
```

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

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

[0 0 0 1 1 1]
```

The chain_index for element 110 is the last residue of chain L, its residue id 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 id 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:

In [27]:
```
q.atom_positions[237:250].shape
```

Out[27]:
```
(13, 37, 3)
```

Residue Renumbering

Residue can be modified. There are a number of renumber 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
```

In [28]:
```
p=Protein(fk)
q=p.data_prt
print(q.chain_index, "\n")

p._renumber(None)
print(q.residue_index, "\n")

p._renumber('RESTART')
print(q.residue_index, "\n")

p._renumber('CONTINUE')
print(q.residue_index, "\n")

p._renumber('GAP200')
print(q.residue_index, "\n")

p._renumber('GAP10')
print(q.residue_index, "\n")

p._renumber('NOCODE')
print(q.residue_index)
```

```
Warning: residues with insertion code: L6A, L6B
[0 0 0 0 1 1]

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

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

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

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

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

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

We generally do not need to call renumbering directly, as it is a utility method used by other methods. This is why the method name starts with "_". 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 missing residues are always preserved.

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:

In [29]:
```
p=Protein(fn)
q, c_pos=p.merge_chains(gap=200, inplace=False)
print(q.seq_dict(gap="."), "\n")
print(c_pos, "\n")

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

```
{'L': 'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTIS
RLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRT
V.....
.....
.....VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVI
PLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVS
S.....
.....
.....NWFDITNWLWYIK'}

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

{'L': 'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTIS
RLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV', 'H': 'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPG
RGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCAREGTTGDGDLGKPIGAFAHWGQGTLVTVS
S', '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.

Selection

Selection is a powerful concept in PyMOL and we support both atom selections and residue selections; it is the major feature missing in BioPython. As the ndarray stores residues as rows and atoms as columns in the first two axes, the combination of a residue selection and an atom selection results in a new numpy array with two axes. We do not support the selection that combines C_α from residue 1 and C_β from residue 2, all residues selected must share the same set of atom selections. We do not see this as a limitation in practice.

Atom Selection

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. None or empty string means all 37 atoms.

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

To create an atom selection, use `ats()` (we could not name it as(), because "as" is a reserved word in Python). For methods relying on atom selections, they treat None as all atoms.

In [30]:
```
p=Protein()
print(p.ats("N,CA,C,O"), "\n")
print(p.ats(["N","CA","C","O"]), "\n")
print(p.ats(""), "\n")
print(p.ats(None), "\n")
```

```
[0 1 2 4]

[0 1 2 4]

None

None
```

Residue Selection

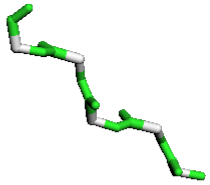
Users use a contig string previously described to select residues. To create a residue selection use the `rs()` method. As the total number of residues equals the `axis=0` dimension of `atom_positions` and `atom_mask` arrays, a residue selection object is literally an integer numpy.ndarray internally. E.g., rs=np.array([0,1,9]). Therefore, `atom_positions[rs, p.ats('N,CA,C,O')]` will capture the 3D coordinates of the 4 backbone atoms for residue 1st, 2nd, and 10th.

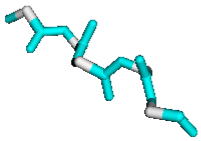
The following is an example where we only extract the backbone atoms from chain H and L and save them into a new PDB file.

In [31]:
```
p=Protein(fn)
p.extract_by_contig("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")
print(p.residues_with_missing_atoms(l_dict=True), "\n")
p.show(style="stick", show_sidechains=True)
```

```
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
ATOM     14  CA  VAL H   4      25.112   9.983 -16.986  1.00  1.00           C
...

{'H': [0, 1, 2, 3, 4], 'L': [5, 6, 7, 8, 9]}
```





Because we only have backbone atoms, all residues have missing atoms.

We implement methods in a way that if a method takes a selection as its argument, the selection can be either a selection string or a ndarray selection object. Users can use any format that is the most convenient in the programming context

Atom selection can be complemented with `ats_not`, which mostly is used internally. Residue selections can be manipulated by all major boolean operators as illustrated below.

In [32]:
```
# as_not reverse the atom selection
aa=p.ats('N,CA,C,O')
print(p.ats2str(aa), "\n")
print(p.ats_not(aa), "\n")
print(p.ats2str(p.ats_not(aa)))
```

```
N,CA,C,O

[ 3  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
 28 29 30 31 32 33 34 35 36]

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,NH2,OH,CZ,CZ2,CZ3,NZ,OXT
```

In [33]:
```
# residue selections
p=Protein(fn)
rs1=p.rs("H1-3")
rs2=p.rs("H3-5:L-3")
print("1st rs:", rs1)
print("2nd rs:", rs2, "\n")
print("and:", p.rs_and(rs1, rs2), "\n")
print("or:", p.rs_or(rs1, rs2), "\n")
# as we not H:L, the only residues left are P chain residues
print("not:", p.rs_not("H:L"), p.data_prt.chain_index[p.rs_not("H:L")], "\n")
# we can specify the residue universe using full_set, this can reduce the size of r
# in the example below, we restrict the not selection within chain L
```

```
print("not with full set:", p.rs_not("L-100", full_rs="L"), "\n")
print("notin:", p.rs_notin(rs1, rs2), "\n")
print(p.rs2str(p.rs_or(rs1, rs2)))
```

```
1st rs: [111 112 113]
2nd rs: [113 114 115   0   1   2]

and: [113]

or: [111 112 113 114 115   0   1   2]

not: [237 238 239 240 241 242 243 244 245 246 247 248 249] [2 2 2 2 2 2 2 2 2 2 2 2
2]

not with full set: [100 101 102 103 104 105 106 107 108 109 110]

notin: [111 112]

L1-3:H1-5
```

`rs2str` returns the canonical contig for a selection, where residues are ordered by chain and by ascending residue index. If we need to canonicalize a selection, this can be done:

In [34]:
```
p=Protein(fn)
# an unusal selection
rs=p.rs("L1-5:P12-13:L6-10")
try:
    p.extract_by_contig(rs)
except Exception as e:
    print("extract_by_contig does not work if residues are not sorted by chain.")
    print(e)
print("Canonical contig:", p.rs2str(rs), "\n")
# the canonicalized contig can be used by extract_by_contig
print(p.extract_by_contig(p.rs2str(rs)).data_prt.residue_index)
```

```
WARNING> contig should keep the segments for the same chain together L6-10, if possi
ble!
extract_by_contig does not work if residues are not sorted by chain.
The residues for chain {c} are not continous!
Canonical contig: L1-10:P12-13

['1' '2' '3' '4' '5' '6' '7' '8' '9' '10' '12' '13']
```

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 3.5A around P chain residues. DeepMind's Protein object does not contain hydrogens. We compute distances among all possible atom pairs and use the minimum atom-atom distance as the final distance of 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 rs_or() method.

There are two returned variables, the first is a new residue selection of the neighboring residues and the second is a dataframe object. The dataframe object provides the details about all residues 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 (contig format), resn_i_ is for the integer part of the residue names without insertion code and it is an integer type. resi_ is for the internal numpy array index (i.e., residue selection). 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.

In [35]:
```
p=Protein(fn)
rs=p.rs("P")
rs_nbr, t=p.rs_around(rs, dist=3.5)
print(p.rs2name(rs_nbr), "\n")
print(p.rs2str(rs_nbr))
t
```

```
['L33', 'L92', 'L93', 'L94', 'L95', 'H30', 'H56', 'H58', 'H98', 'H106', 'H107', 'H10
8', 'H109']

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

Out[35]:

	chain_a	resn_a	resn_i_a	resi_a	res_a	chain_b	resn_b	resn_i_b	resi_b	res_b	
2709	P	6	6	242	T	H	98	98	208	E	2.63
419	P	4	4	240	D	L	33	33	32	K	2.81
1223	P	2	2	238	W	L	95	95	94	S	2.91
1183	P	1	1	237	N	L	92	92	91	Y	2.92
2840	P	7	7	243	N	H	108	108	218	K	3.03
1209	P	1	1	237	N	L	94	94	93	Q	3.08
2159	P	2	2	238	W	H	56	56	166	I	3.23
1196	P	1	1	237	N	L	93	93	92	G	3.25
2817	P	10	10	246	W	H	106	106	216	L	3.30
2185	P	2	2	238	W	H	58	58	168	N	3.32
1825	P	6	6	242	T	H	30	30	140	T	3.37
2830	P	10	10	246	W	H	107	107	217	G	3.41
2853	P	7	7	243	N	H	109	109	219	P	3.47

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

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 residues in H95-106, we cannot use the *resn* column for filtering. Instead, we can accomplish this with the *resn_i* column:

In [36]:
```
print("resn cannot be used for filtering, as it's string type, str greater than '95
t2=t[(t.chain_b=="H")&(t.resn_b>="95")&(t.resn_b<="106")]
t2.display()
print("\n")
print("resn_i is an integer column")
t2=t[(t.chain_b=="H")&(t.resn_i_b>=95)&(t.resn_i_b<=106)]
t2.display()
```

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


resn_i is an integer column
      chain_a      resn_a    resn_i_a    resi_a  res_a    chain_b      resn_b    res
n_i_b    resi_b  res_b       dist  atom_a    atom_b
----  ---------  --------  ----------  --------  -------  ---------  --------  -----
-----  --------  -------  -------  --------  --------
2709  P                6           6       242  T        H               98
98       208  E        2.63625  OG1       OE2
2817  P               10          10       246  W        H              106
106       216  L        3.30522  NE1       O
```

In [37]:
```
p=Protein(fn)
rs=p.rs("P")
# select residues on chain H that have CA-CA distance within 8A.
rs_nbr, t=p.rs_around(rs, dist=8, within_rs="H", ats="CA")
print(p.rs2name(rs_nbr))
t
```

```
['H30', 'H31', 'H32', 'H51', 'H58', 'H108', 'H109', 'H110', 'H111']
```

Out[37]:

	chain_a	resn_a	resn_i_a	resi_a	res_a	chain_b	resn_b	resn_i_b	resi_b	res_b	
1432	P	3	3	239	F	H	111	111	221	G	5.57
1410	P	7	7	243	N	H	109	109	219	P	5.70
395	P	6	6	242	T	H	31	31	141	Y	5.91
382	P	6	6	242	T	H	30	30	140	T	6.08
1400	P	10	10	246	W	H	108	108	218	K	6.22
408	P	6	6	242	T	H	32	32	142	A	6.73
742	P	2	2	238	W	H	58	58	168	N	7.06
655	P	6	6	242	T	H	51	51	161	I	7.58
1423	P	7	7	243	N	H	110	110	220	I	7.69

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("out.pdb", p.html())`.

Within Jupyter Notebook, p.show() will display the structure as we do in 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].

In [38]:
```
p=Protein(fn)
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 prediction quality pLDDT, 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.

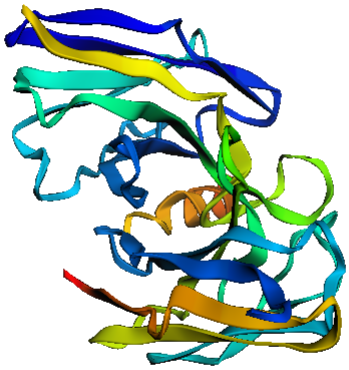
In [39]:
```
p=Protein(fk)
# read b_factors as a numpy array
print(p.b_factors())
# set b_factors by providing a numpy array
import numpy as np
p.b_factors(np.random.random((len(p))), "\n")
```

```
Warning: residues with insertion code: L6A, L6B
[1. 1. 1. 1. 1. 1.]
```

Out[39]:
```
array([0.20033806, 0.62992363, 0.96355851, 0.83482605, 0.9890843 ,
       0.51382198])
```

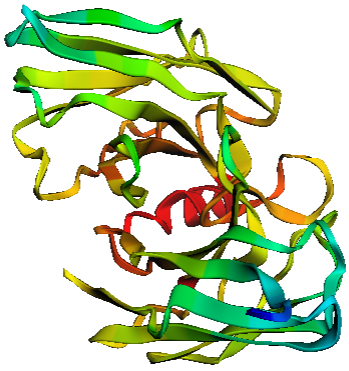
To color by b-factors, set `color = "b"` in the show() method. However, please normalize b-factor ndarray to be within [0, 1] to get the expected coloring effect.

In [40]:
```
p=Protein(fn)
n=len(p)
p.data_prt.b_factors[:,:]=np.sin(np.arange(n)/n*np.pi).reshape(-1,1)
p.show(style="cartoon", color="b")
```



It may be more convenient to get or set b-factors for each chain, as one array containing all residues from all chains can be troublesome to use. We can use b_factors_by_chain to achieve this.

In [41]:
```
p=Protein(fn)
rs, dist=p.rs_around("P", dist=1e8)
dist.sort_values("resi_b", inplace=True)
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":np.zeros(len(p.rs("P"))), "H":dist[dist.chain_b=="H"].dist.values, "L":dist[
# set b_factors
p.b_factors_by_chain(b)
p.show(color="b")
```



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

In [42]:
```
# 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()
```

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.

Measurement

Distance

The `rs_around` method already uses the distance measurement method rs_dist() internally.

Give two residue groups, `rs_a` and `rs_b`, which 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.

In [43]:
```
p=Protein(fk)
print(p.seq_dict())
print(p.data_prt.residue_index, "\n")

t=p.rs_dist('L', 'H')
t.display()
print("\n")
# we restricted the distance measurement to CA-CA distance, notice the distance are
```

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

```
Warning: residues with insertion code: L6A, L6B
{'L': 'EIVXXXQ', 'H': 'LV'}
['5' '6A' '6B' '10' '3' '4']
```

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
L	10	10	3	Q	H	3	3	4	L	21.2251	OE1	CD1
L	5	5	0	E	H	3	3	4	L	22.9552	OE1	CD1
L	6A	6	1	I	H	3	3	4	L	23.3484	O	CD1
L	6B	6	2	V	H	3	3	4	L	25.339	C	CD1
L	10	10	3	Q	H	4	4	5	V	25.5449	OE1	N
L	5	5	0	E	H	4	4	5	V	27.1606	OE1	C
L	6A	6	1	I	H	4	4	5	V	28.2433	O	N
L	6B	6	2	V	H	4	4	5	V	30.2065	C	N

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
L	6A	6	1	I	H	3	3	4	L	28.5645	CA	CA
L	5	5	0	E	H	3	3	4	L	28.8874	CA	CA
L	10	10	3	Q	H	3	3	4	L	28.9785	CA	CA
L	6B	6	2	V	H	3	3	4	L	29.0182	CA	CA
L	6A	6	1	I	H	4	4	5	V	30.6464	CA	CA
L	10	10	3	Q	H	4	4	5	V	30.719	CA	CA
L	5	5	0	E	H	4	4	5	V	30.7279	CA	CA
L	6B	6	2	V	H	4	4	5	V	30.8542	CA	CA

Instead of residue distance, we can also measure the distance between all-atom combinations. Our ultimate aim here is to add a column for interaction types, such as clash, electrostatic, hydrogen bonds, etc.

In [44]:
```
# We list distances between all atom pairs of two residues
p=Protein(fk)
t=p.atom_dist('L6A', 'H3')
t[:15].display()
print("\n...\n")

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

Warning: residues with insertion code: L6A, L6B

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
L	6A	6	1	I	H	3	3	4	L	23.3484	O	CD1
L	6A	6	1	I	H	3	3	4	L	24.169	CB	CD1
L	6A	6	1	I	H	3	3	4	L	24.4316	CD1	CD1
L	6A	6	1	I	H	3	3	4	L	24.4987	C	CD1
L	6A	6	1	I	H	3	3	4	L	24.578	CG2	CD1
L	6A	6	1	I	H	3	3	4	L	24.6116	N	CD1
L	6A	6	1	I	H	3	3	4	L	24.6718	CG1	CD1
L	6A	6	1	I	H	3	3	4	L	24.8596	CA	CD1
L	6A	6	1	I	H	3	3	4	L	24.8703	O	CG
L	6A	6	1	I	H	3	3	4	L	25.4454	O	CD2
L	6A	6	1	I	H	3	3	4	L	25.4494	O	CB
L	6A	6	1	I	H	3	3	4	L	25.6839	CB	CG
L	6A	6	1	I	H	3	3	4	L	25.9234	CD1	CG
L	6A	6	1	I	H	3	3	4	L	26.0208	C	CG
L	6A	6	1	I	H	3	3	4	L	26.0872	CG2	CG

...

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
L	6A	6	1	I	H	3	3	4	L	26.9663	O	CA
L	6A	6	1	I	H	3	3	4	L	27.2071	O	O
L	6A	6	1	I	H	3	3	4	L	27.4731	O	C
L	6A	6	1	I	H	3	3	4	L	27.4823	O	N
L	6A	6	1	I	H	3	3	4	L	28.1425	C	CA
L	6A	6	1	I	H	3	3	4	L	28.2995	N	CA
L	6A	6	1	I	H	3	3	4	L	28.4143	C	O

```
 38  L          6A                  6         1  I        H                  3
3         4  L        28.5645  CA        CA
  2  L          6A                  6         1  I        H                  3
3         4  L        28.6462  C         N
 76  L          6A                  6         1  I        H                  3
3         4  L        28.6705  C         C
148  L          6A                  6         1  I        H                  3
3         4  L        28.7272  N         O
  0  L          6A                  6         1  I        H                  3
3         4  L        28.8623  N         N
 74  L          6A                  6         1  I        H                  3
3         4  L        28.8824  N         C
149  L          6A                  6         1  I        H                  3
3         4  L        28.9565  CA        O
  1  L          6A                  6         1  I        H                  3
3         4  L        29.0662  CA        N
 75  L          6A                  6         1  I        H                  3
3         4  L        29.1546  CA        C
```

RMSD

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

In [45]:
```
p=Protein(fk)
q=p.translate([3.0,4.0,0.0], inplace=False)
print(p.center(), q.center(), p.center()-q.center(), "\n")
# consider all matched atoms, rs='' or rs=None mean all residues
print(q.rmsd(p, '', ''), "\n")
# consider CA only
print(q.rmsd(p, '', '', ats='CA'))
```

```
Warning: residues with insertion code: L6A, L6B
Warning: residues with insertion code: L6A, L6B
[ 15.54949999  -8.0205001  -15.39166681] [ 18.54949999  -4.0205001  -15.39166681] [-
3. -4.  0.]

5.000000001

5.000000001
```

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".

Solvent-Accessible Surface Area (SASA)

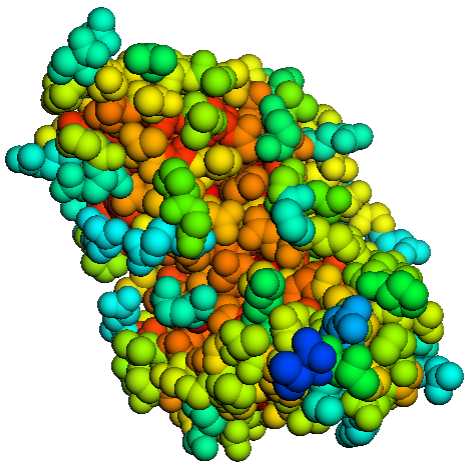
Use in_chains to specify the chains to be used. For example, if all chains are included in the antibody-antigen complex, CDR binding residues will not be accessible to water. But if antigen is excluded, CDR will be exposed.

In [46]:
```
p=Protein(fn)
# consider all chains
t=p.sasa()
sasa=p.sasa().SASA.values
p.b_factors(sasa/sasa.max())
p.show(style="sphere", color="b")

t_all=t[(t.chain=="L")][90:105]
t_all.display()
print("\n")
# ignore antigen
t=p.sasa("H:L")
t_ag=t[(t.chain=="L")][90:105]
t_ag.display()
print("\n")
t_ag['DELTA_SASA']=t_ag['SASA'].values-t_all['SASA'].values
t_ag.display()

binder, t=p.rs_around(p.rs("P"), dist=4)
print("\n")

# rs2name convert a residue selection into residue name, rs2namei convert a residue
print([x for x in p.rs2name(binder) if x.startswith("L")])
```



	chain	resn	resn_i	resi	SASA
90	L	91	91	90	1.07146
91	L	92	92	91	1.07146
92	L	93	93	92	19.4224
93	L	94	94	93	61.1719
94	L	95	95	94	41.9112
95	L	96	96	95	37.9599
96	L	97	97	96	3.21438
97	L	98	98	97	12.0763
98	L	99	99	98	17.3902
99	L	100	100	99	0
100	L	101	101	100	124.88
101	L	102	102	101	14.2013
102	L	103	103	102	2.14292
103	L	104	104	103	103.555
104	L	105	105	104	3.21438

	chain	resn	resn_i	resi	SASA
216	L	91	91	90	1.07146
217	L	92	92	91	36.3292
218	L	93	93	92	33.8961
219	L	94	94	93	78.6055
220	L	95	95	94	80.9365
221	L	96	96	95	37.9599
222	L	97	97	96	8.04489
223	L	98	98	97	12.0763
224	L	99	99	98	17.3902
225	L	100	100	99	0
226	L	101	101	100	124.88
227	L	102	102	101	14.2013
228	L	103	103	102	2.14292
229	L	104	104	103	103.555
230	L	105	105	104	3.21438

	chain	resn	resn_i	resi	SASA	DELTA_SASA
216	L	91	91	90	1.07146	0
217	L	92	92	91	36.3292	35.2578
218	L	93	93	92	33.8961	14.4736
219	L	94	94	93	78.6055	17.4336
220	L	95	95	94	80.9365	39.0253
221	L	96	96	95	37.9599	0
222	L	97	97	96	8.04489	4.83051
223	L	98	98	97	12.0763	0
224	L	99	99	98	17.3902	0
225	L	100	100	99	0	0
226	L	101	101	100	124.88	0
227	L	102	102	101	14.2013	0
228	L	103	103	102	2.14292	0
229	L	104	104	103	103.555	0
230	L	105	105	104	3.21438	0

```
['L33', 'L92', 'L93', 'L94', 'L95', 'L97']
```

We can see we can either use rs_around to select interface residues, or calculate $\Delta ASAS$.

Secondary Structures - DSSP

The 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.

In [47]:
```
p=Protein(fn)
print(p.dssp(), "\n")
ss=p.dssp(simplify=True)
print(ss, "\n")
# set helix to 1, otherwise 0
for k,v in ss.items():
    ss[k]=1-(np.array(list(v))=="a").astype(int)
p.b_factors_by_chain(ss)
p.show(color="b")
```

```
{'L': '---EE-SSSEEEE-TT--EEEEEEESS--GGG-EEEEEE-TTS--EEEEBTTTB--TTS-TTEEEEEETTEEEEEES
S--GGG-SEEEEEE-SSSS-EE---EEEEE----', 'H': '-EEEE---EEE-TT--EEEEEEEESS-STTS-EEEEEE-TT
S-EEEEEEEEGGGTEEEE-GGGTTTEEEEEETTTTEEEEEE-S--GGG-EEEEEEEEESSSSSSS-EEEEEEEE---EEEEE-
-', 'P': '-GGGHHHHHHHH-'}

{'L': '---bb-cccbbbb-cc--bbbbbbbcc--aaa-bbbbbb-ccc--bbbbbcccb--ccc-ccbbbbbbccbbbbbbc
c--aaa-cbbbbbb-cccc-bb---bbbbb----', 'H': '-bbbb---bbb-cc--bbbbbbbbcc-cccc-bbbbbb-cc
c-bbbbbbbbaaacbbbb-aaacccbbbbbbccccbbbbbb-c--aaa-bbbbbbbbbbccccccc-bbbbbbbb---bbbbb-
-', 'P': '-aaaaaaaaaaa-'}
```

Internal Coordinate

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

In [48]:
```
p=Protein(fk)
t=p.internal_coord(rs="L")
t
```

```
Warning: residues with insertion code: L6A, L6B
Warning: residues with insertion code: L6A, L6B
chain break at GLN  10  due to MaxPeptideBond (1.4 angstroms) exceeded
```

Out[48]:

	chain	resn	resn_i	resi	aa	-1C:N	N:CA	CA:C	phi	psi	
0	L	5	5	0	E	NaN	1.483377	1.532696	NaN	127.190158	
1	L	6A	6	1	I	1.313512	1.449593	1.512209	-83.955214	121.806014	177.4
2	L	6B	6	2	V	1.325801	1.438910	1.527408	-92.621830	NaN	174.4
3	L	10	10	3	Q	NaN	1.451717	1.535136	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 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):

In [49]:
```
p=Protein(fk)
t=p.internal_coord(rs="L", MaxPeptideBond=1e6)
t
```

```
Warning: residues with insertion code: L6A, L6B
Warning: residues with insertion code: L6A, L6B
```

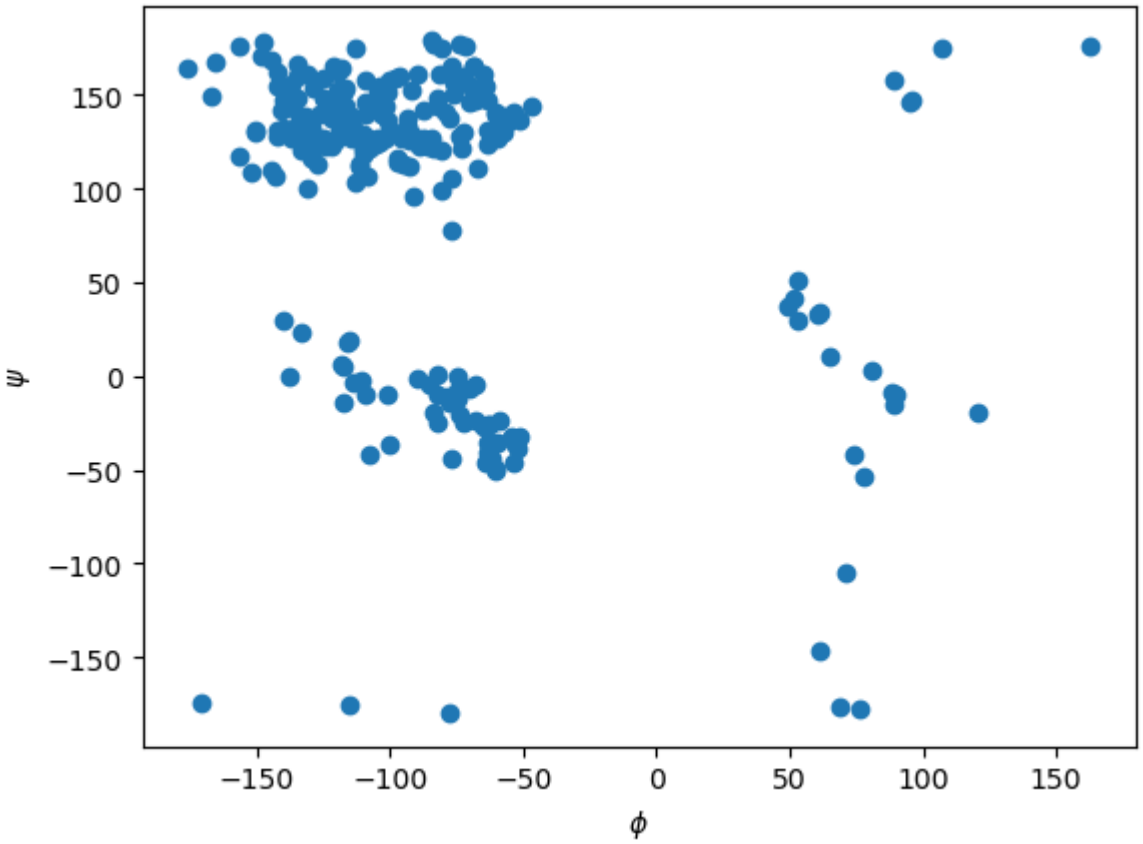
Out[49]:

	chain	resn	resn_i	resi	aa	-1C:N	N:CA	CA:C	phi	psi	
0	L	5	5	0	E	NaN	1.483377	1.532696	NaN	127.190158	
1	L	6A	6	1	I	1.313512	1.449593	1.512209	-83.955214	121.806014	17
2	L	6B	6	2	V	1.325801	1.438910	1.527408	-92.621830	176.913905	17
3	L	10	10	3	Q	7.310299	1.451717	1.535136	-165.152326	NaN	-14

In [50]:

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

```
Ramachandran Plot
```



Object Manipulation

Move Object

We can recenter, translate, and rotate an object.

In [51]:
```
p=Protein(fk)
print("Original center:", p.center(), "\n")
q=p.center_at([3.0,4.0,5.0], inplace=False)
print("New center after recentering:", q.center(), "\n")
q=p.translate([1.,0.,-1.], inplace=False)
print("New center after translate:", q.center(), "\n")
# rotate 90 degrees around axis [1,1,1]
print("RMSD before rotation:", q.rmsd(p, None, None, "CA"), "\n")
# we only rotate 5 degrees
q=p.rotate([1,1,1], 5, inplace=False)
print("Old center:", p.center(), "New center:", q.center(), "\n")
print("RMSD after rotation:", q.rmsd(p, ats="CA"), "\n")
# Warning, rotate is done using the origin as the center, that's why the rotation d
# 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], inplace=False)
# 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"), "\n")
```

```
Warning: residues with insertion code: L6A, L6B
Original center: [ 15.54949999  -8.0205001  -15.39166681]

Warning: residues with insertion code: L6A, L6B
New center after recentering: [3. 4. 5.]

Warning: residues with insertion code: L6A, L6B
New center after translate: [ 16.54949999  -8.0205001  -16.39166681]

RMSD before rotation: 1.414213565908629

Warning: residues with insertion code: L6A, L6B
Old center: [ 15.54949999  -8.0205001  -15.39166681] New center: [ 15.10944354  -6.4
4301224 -16.52909822]

RMSD after rotation: 2.1901944111953062

Warning: residues with insertion code: L6A, L6B
RMSD (center before rotate): 0.9061366909433876
```

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.

In [52]:
```
p=Protein(fn)
print("Old center:", p.center(), "\n")
p.rotate(np.random.random(3), np.random.random()*180)
p.show()
p.reset_pos()
p.show()
print("New center:", p.center())
```

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

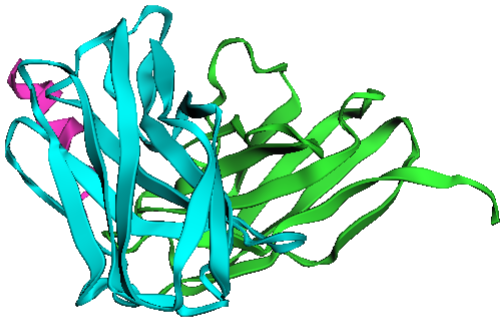
```
New center: [-2.84217094e-17  9.57811608e-15 -3.67617048e-15]
```

Align

It often makes sense to align two objects before making RMSD measurement.

In [53]:
```
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, '', '', 'CA'))
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 before alignment: 45.854280383765364
```

```
RMSD after alignment: 0.00010000030524755665
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  3.11646735e-07]]
```

Split & Merge Objects

To split an object into multiple objects, just use extract_by_contig. To merge multiple structure objects into one, use merge().

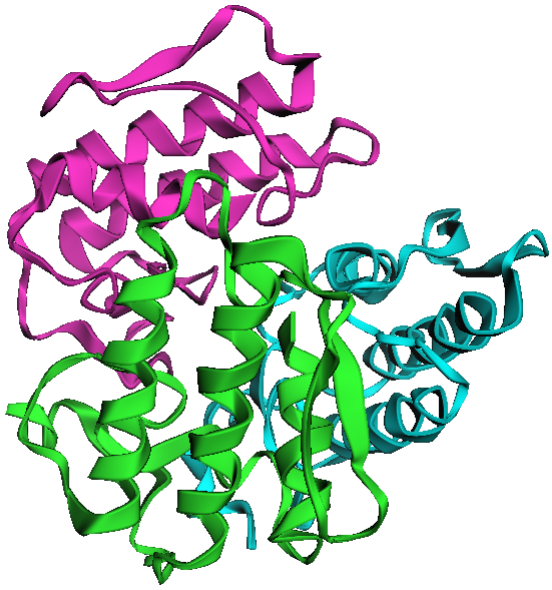
In [54]:

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

print("Exact chain into a new object\n")
Q=[ p.extract_by_contig(x, inplace=False) for x in p.chain_id() ]
print("Object 1:", Q[0].seq_dict(), "\n")
# for object 2 & 3, computer RMSD, then align them against object 1 and print RMSD
for i,q in enumerate(Q[1:]):
    # we intentionally rename chains to A for all object, the merge method will be
    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')
    print("RMSD after align:", q.rmsd(Q[0], None, None, ats='N,CA,C,O'), "\n")

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

{'A': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLT
CKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKARCQ', 'B': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGS
GTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKAR
CQ', 'C': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQ
GTLTCKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKARCQ'}



Exact chain into a new object

Object 1: {'A': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTY
SYECSQGTLTCKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKARCQ'}

Object 2: {'A': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTY
SYECSQGTLTCKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKARCQ'}
RMSD to Object 1: 29.52079507422956

RMSD after align: 0.0001

Object 3: {'A': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTY
SYECSQGTLTCKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKARCQ'}
RMSD to Object 1: 29.52079507422956

RMSD after align: 0.0001

Rename chain: object 1, A to B
Rename chain: object 2, A to C
{'A': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLT
CKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKARCQ', 'B': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGS
GTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQGTLTCKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKAR
CQ', 'C': 'NLYQFKNMIKCTVPSRSWWDFADYGCYCGRGGSGTPVDDLDRCCQVHDNCYNEAEKISGCWPYFKTYSYECSQ
GTLTCKGDNNACAASVCDCDRLAAICFAGAPYNDNNYNIDLKARCQ'}

Developer's Notes

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 ndarray. Similarly, if we have an atom selection argument named 'ats', do `ats=self.ats(ats)`. When we use an atom selection to index atom_positions or atom_mask, check if the selection is None.

Please use `extract_by_contig` 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_by_contig` 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. There are helper methods rs2name and rs2name2 help convert a residue selection into resn and resn_i. Chain name can be obtained by

`self.data_prt.chain_id[self.data_prt.chain_index[rs]]`. Residue types (sequence) can be obtained by `self.rs2aa`.

inplace

To support `inplace`, the idiom is to use: `obj = self if inplace else self.clone()`, then use obj to manipulate the structure.

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.

In [ ]: