Afpdb - Developer's Note

Installation

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

For Colab users, please skip this cell.

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

1. 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.
2. Install Afpdb

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

3. Jupyter Notebook (optional) To view and run this tutorial, Jupyter should be installed:

```
pip install notebook
```

Type command `jupyter notebook` to lauch 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
```

4. 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
```

5. DSSP (optional) Required for the secondary structure prediction with method `dssp()`.

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

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

6. matplotlib (optional) Required for the Ramanchondra plot example

   ```
   pip install matplotlib
   ```

7. 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.

```
from pathlib import Path
import os
pwd=Path(os.getcwd())
IN_COLAB=str(pwd)=="/content" # we are in Google Colab

if IN_COLAB:
    pwd=Path("/content/afpdb/tutorial")
    # remove local proxy setting
    os.environ["https_proxy"]=""
    os.environ["http_proxy"]=""
    os.environ["ftp_proxy"]=""
    # install afpdb
    if not os.path.isfile("INSTALL_AFPDB"):
        ! git clone git+https://github.com/data2code/afpdb.git && cd afpdb && pip in
        ! touch INSTALL_AFPDB
    from IPython.display import Javascript
    display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight: 5
    from IPython.display import HTML, display
    def set_css():
        display(HTML('''
          <style>
            pre {
                white-space: pre-wrap;
            }
          </style>
        '''))
    get_ipython().events.register('pre_run_cell', set_css)
else: # in a local jupyter notebook
    %reload_ext autoreload
    %autoreload 2
    # we assume afpdb has been preinstall

def install_pymol():
    try:
        import pymol2
    except Exception as e:
        if not IN_COLAB:
            print("Please install PyMOL first!")
        else:
            !pip install -q condacolab
            import condacolab
            condacolab.install()
            ! conda install conda-forge::pymol-open-source
            print("Colab does not have openssl 3.2.0, install it...")
            ! conda install conda-forge::openssl=3.2.0
            import pymol2

from afpdb.afpdb import Protein,util,RS,RL,ATS
import numpy as np
import pandas as pd
import re
```

```
# two example PDB files used in this tutorial
fn = pwd / "example_files/5cil.pdb"
fk = pwd / "example_files/fake.pdb"
```

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.

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).

```
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)
```

```
Warning: residues with insertion code: L6A, L6B
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
# 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)
# 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")
# or
print(p.data.atom_mask[np.array([2,3])][:, ATS("N,CA,C,O,CB,CG").data])
```

```
Warning: residues with insertion code: L6A, L6B
shape mismatch: indexing arrays could not be broadcast together with shapes (2,)


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

[[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"), AT
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)
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)
```

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



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`.