import os, sys, gzip, argparse, glob, string

from Bio.PDB import *
from Bio import pairwise2

def read_pdb_files(pdb_files, options_verbose):
"""Given a pdb file, read it, remove the heteroatoms and create a dictionary with the chain ids and the structure

```
Input:
    PDB File (files argument) with a pairwise interaction

Output:
Dictionary with three elements: Chain ids (2) and the structure """

dict_with_NP={}
dict_with_PP={}
#homodimer_dict={}
#heterodimer_dict={}
pdb_parser=PDBParser(PERMISSIVE=1, QUIET=True)
# alpha_carbons=CaPPBuilder()

for file in pdb_files:
    id=file[:-4]
    structure=pdb_parser.get_structure(id,file)
    chains_ids=''.join([chain.id for chain in structure.get_chains()])
    chains=[]
    alpha_carbon_chains=0

    #Obtain the alpha carbon structure of each chain
    removeable=[]
    for chain in structure.get_chains():
        for residue in chain:
            if residue in chain:
                if residue.id[0] != ' ':
                    removeable.append(residue.id)

    #Now that heteroatoms are selected, remove them from the chain
        for residue in removeable:
            chain.detach_child(residue)
        chains.append(chain)

    #Finally, obtain the alpha carbon chain and store it
    #Check if the length of the polypeptide chain is long enough to not be considered a ligand/cofactor (not
        if len(chain)<=25 and len(next(chain.get_residues()).get_resname())<3:
            structure[0].detach_child(chain.id)
        else:
            alpha_carbon_chains+=1          # counter for number of structures
            # chain_alpha = alpha_carbons.build_peptides(chain)
            # alpha_carbon_chains.append(chain_alpha[0].get_sequence())


    # Check the if we are working with P-Pinteraction or P-Nuc interactions:
    key_chain = [x for x in structure.get_chains()][1]
    chain_type = alpha_carbons_retriever(key_chain, options_verbose)[1]

    if chain_type =="Protein":               # If P-P interaction, we need to have binary interactions (2 CA
        if alpha_carbon_chains!= 2:
            if options_verbose:
                sys.stderr.write("File %s does not have right input format." % (file))
            continue
        dict_with_PP[id]=structure

    else:
        if alpha_carbon_chains != 3:        # If P-Nuc interaction, we need to have 3 different Seqs in list
            if options_verbose:
                sys.stderr.write("File %s does not have right input format." % (file))
            continue
        dict_with_NP[id]=structure

if bool(dict_with_PP) == True:
    return (dict_with_PP, "PP")

else:
    return (dict_with_NP,"NP")
```

#==

def alpha_carbons_retriever(chain, options_verbose):
""""
Get alpha Carbons from input chains (CA for preotein sequence and C4' for DNA/RNA).

```
Argument: chain class with the atoms

Returns: - list of CA or C4 atoms
         - class of molecule we are working with

"""
nucleic_acids = ['DA','DT','DC','DG','DI','A','U','C','G','I']
RNA = ['A','U','C','G','I']
DNA = ['DA','DT','DC','DG','DI']
atoms = []

for residue in chain:
    res_type = residue.get_resname().strip()       # Get residue name
    if residue.get_id()[0] == " ":                 #Check if we are dealing with and HET entry

        if res_type not in nucleic_acids:         # If residue is not a nucleic acid
            if 'CA' not in residue:                 # If there are no alpha carbons
                if options_verbose:                 # And the verboes option has been set in the function: pr
                                                    # informing about not having CA
                    sys.stderr.write("This residue %d %s doest not have an alpha carbon" % (residue.get_id()[

            else:                                   # If there are alfa cabrons
                atoms.append(residue['CA'])
                molecule='Protein'

        elif res_type in DNA:       #Otherwise, if the residue is a deoxynucleic acid
            molecule = 'DNA'
            atoms.append(residue['C4\''])

        elif res_type in RNA:       #Finally, if the residue is a nucleic acid
            molecule = 'RNA'
            atoms.append(residue['C4\''])

return(atoms, molecule)      #Return the list of alpha carbon atoms and the molecule types
```

#==

def sequence_alignment(chain1,chain2):

"""Comparing if the pairwise interaction holds a homodimer or heterodimer"""

align=pairwise2.align.globalxx(chain1,chain2)

identity=align[0][2]/max(len(chain1),len(chain2))

return identity

#
#

alignment = pairwise2.align.globalxx(sequence1, sequence2)

return alignment

#==

def dir_path(string):
"""A function to check whether a string is a directory or not"""
if os.path.isdir(string):
return string
else:
raise NotADirectoryError(string)
#==

def transform_to_structure(dictionary,name):
"""
From a dictionary of binary interactions (heterodimer/homodimer dictionaries)
transform it to a structure class object which will be used in superimposition"""

```
structure_object=Structure.Structure(name)

i=0

for structure_chains in dictionary.values():
    structure_object.add(Model.Model(i))
    for chain in structure_chains.get_chains():
        structure_object[i].add(chain)
    i+=1

return structure_object
```

#==

def check_files(path):
"""
A function to check whether PDB input files have correct format
"""

```
work_files=[]
for file in os.listdir(path):
    if file.endswith(".pdb"):
        work_files.append(file)


if not work_files:
```

if my_pattern.match(file) == None:

```
    raise ValueError("Check the PDB input files format")
else:
    os.chdir(path)
    # print(len(work_files))
    return work_files
```

#==

def output_dir(string, options_force):
""""
A function to check whether outputfile already exists
""""
if options_force is False:
if os.path.isdir(string):
raise ValueError("Directory already exists. Please set -f to True to overwrite the directory")
else:
sys.stderr.write("Setting the output directory to %s" % (string))
os.mkdir(string)

#==

def align_chains(chain1, chain2):
""""
Run alignment for two chains of any type
Return alignment score
""""
alignment=pairwise2.align.globalxx(chain1,chain2)
alig_score=alignment[0][2]/max(len(chain1),len(chain2))

```
return alig_score
```

#==

def align_chains_peptides(chain1,chain2):
""""
A function aligning two chains with each other
Returns the final alignment score of both peptidic chains
""""
alpha_carbons=CaPPBuilder()

```
chain1_carbons=alpha_carbons.build_peptides(chain1)
chain1_carbons=chain1_carbons[0].get_sequence()

chain2_carbons=alpha_carbons.build_peptides(chain2)
chain2_carbons=chain2_carbons[0].get_sequence()

# alignment=pairwise2.align.globalxx(chain1_carbons,chain2_carbons)
#
# alig_score=alignment[0][2]/max(len(chain1_carbons),len(chain2_carbons))

return align_chains(chain1_carbons, chain2_carbons)
```

#===

def superimpose_chains(ref_structure,alt_structure,threshold, options_verbose):
"""
Core function to firstly align chains from reference and alternative model.
Secondly, for those chains found to be similar, superimpose them and obtain
a dictionary with all the possible superimposition of the chains from the
two structures (if the superimposition is below a certain RMSD threshold)
"""
superimpositions={}
best_RMSD=""
ref_chains=[x for x in ref_structure.get_chains()]
alt_chains=[x for x in alt_structure.get_chains()]
sup=Superimposer() # Superimposer from Biopython

```
for ref_chain in ref_chains:
    for alt_chain in alt_chains:
        if align_chains_peptides(ref_chain,alt_chain) > 0.95: # for the similar chains
            ref_atoms, ref_molecule = alpha_carbons_retriever(ref_chain, options_verbose)
            alt_atoms, alt_molecule = alpha_carbons_retriever(alt_chain, options_verbose)
            sup.set_atoms(ref_atoms,alt_atoms)  # retrieve rotation and translation matrix
            RMSD=sup.rms                        # get RMSD for superimposition

            if RMSD < threshold:
                if not best_RMSD or RMSD < best_RMSD:
                    best_RMSD=RMSD
                superimpositions[(ref_chain.id,alt_chain.id)]=sup # add superimposition to dictionary

if bool(superimpositions) == True: #If we are able to superimpose any chain
    superimpositions=sorted(superimpositions.items(), key=lambda x:x[1].rms) #sort the superimpositions by RM
    return (superimpositions,best_RMSD)
```

#===

def create_ID(IDs_present):
"""
Create new ID to make sure it is a non-taken ID
Input: list of IDs already occupied
Return: new ID
"""

```
Up = list(string.ascii_uppercase)
Low = list(string.ascii_lowercase)
Dig = list(string.digits)
possibilities = set(Up+Low+Dig) # set of all acceptable IDs that are possible

if len(IDs_present)<62:
    possibilities.difference_update(set(IDs_present)) # update possibilities set by substracting taken ID
    return list(possibilities)[0]

elif len(IDs_present)>=62: # as soon as all possibilities from the set are taken
    for character in possibilities:
        for character2 in possibilities:
            ID = character + character2     # combine letters to createe new ID
            if ID not in IDs_present:       # test if new ID already taken
                return
            else:
                continue
```