Data Mining the PDB
Session 7

Robert Palmere

2021

Topics:

- Acquire data from the PDB
- Handling of PDB data
- User-defined functions for analysis

This sessions serves as an example of how one might perform data mining of the Protein Data Bank (PDB) using Python. We will look at methods for handling textual data from the PDB and write user defined functions for analysis. For this example, we will look at which residues are most solvent exposed across the span of proteins related to membranes.

The first order of business is to write a series of functions for parsing a PDB file. In certain cases, the coordinate values may not be space delimited. For example,

```
... -140.405 107.301
... -153.534-108.134
```

Because of this, a function is written to check for this and separate the two values despite not being space delimited.

```
def floatcustom(s: str):
  try:
      return float(s)
  except:
    indices = []
    floats_ = []
    for index, char in enumerate(s):
        if char == '-':
          indices.append(index)
    if len(indices) > 1 and ' ' not in s:
        for n, i in enumerate(indices):
          if n >= 1:
             floats_.append(float(s[indices[n-1]:indices[n]]))
        floats_.append(float(s[indices[-1]:]))
    return floats_
```

Using a try / except here, we check if the string can be casted to a float. If not, then the negative sign will be the criterion for parsing the columns. Here we also use the new syntax of Python3 where we can indicate the type expected by the function for clarity. When using this function, it may append a sublist to the list of columns we wish to parse from the PDB text. We write a function to check that all sublists containing columns are flattened into a single list of each column value per row.

```
def checkflattened(l: list):
   flattened_ = []
   contains_sublist = False
   for i in l:
         if isinstance(i, list):
            contains_sublist = True
            for j in i:
               flattened_.append(j)
      else:
            flattened_.append(i)
   if contains_sublist:
      del flattened_[-1]
   return flattened_

test_list = ['1', ['2', '3'], '4']
checkflattened(test_list) # returns ['1', '2', '3', '4']
```

Lastly, we implement a function for the complete parsing of a PDB file. The function returns both a dictionary of information provided by the PDB as well as the data from the columns of the PDB file as a separate dictionary. Note that *checkflattened* and *floatcustom* are only required for the 'xyz' portion of the PDB file data. Using the additions to the syntax in Python3, we can also indicate the expected return type of the function using type hints in the function prototype.

```
   def pdbParser(pdbtext: str) -> dict:
      info_ = {
              'HEADER' : None,
              'TITLE' : None,
              'SOURCE' : [],
              'REMARKS' : [],
              }
      protein_ = {'ATOM' : [], 'HETATM' : []}
      listofstrings = pdbtext.split('\n')
      listofstrings = [' '.join(substring.split()) for substring in
          listofstrings]
      for substring in listofstrings:
          split = substring.split(' ')
          if split[0] in tuple(info_.keys()):
              if split[0] == 'HEADER' or split[0] == 'TITLE':
                  info_[split[0]] = split[1:]
              else:
                  info_[split[0]].append(split[1:])
          elif split[0] in protein_.keys():
              xyz = checkflattened([floatcustom(i) for i in
                  split[6:9]])
              if len(xyz) != 3:
                  return None
              p_ = {
                  'serial' : split[1],
                  'name' : split[2],
                  'resname' : split[3],
                  'chain' : split[4],
                  'resid' : split[5],
                  'xyz' : checkflattened([floatcustom(i) for i in
                     split[6:9]])
                  }
              protein_[split[0]].append(p_)
      return info_, protein_
```

The *pypdb* library is imported so that a function can be written for querying the PDB database without downloading the files to our local system using the *Query* class. We will find, however, that although this method works for a small set of PDBs, the querying takes too long to perform on all the requested PDBs. There are a couple of ways to increase the speed of this analysis:

```
1. Parallelization
2. Vectorize (use NumPy were applicable)
3. Cythonize
4. Change to a mid-level language
5. Profile and modify
```

In our case, we will use Amarel for computations and download a subset of the PDBs to our current working directory. Before we write the function to acquiring data from these PDBs, we will write the function for obtaining solvent exposed residues. This function takes in the coordinates of the PDB (ignoring vdW radii) and renders a ranking of residues which are "solvent exposed". This is not the most accurate method, but is used for demonstration purposes.

```
    def getSolventExposed(protein, threshold=5):
       c = np.stack(np.asarray([np.asarray(atom['xyz']) for atom in
           protein['ATOM']]), axis=0)
       resname_list = np.asarray([atom['resname'] for atom in
           protein['ATOM']])
       center = (np.average(np.asarray(c)[:, 0]),
           np.average(np.asarray(c)[:, 1]),
           np.average(np.asarray(c)[:, 2]))
       distance = lambda c_tuple : np.sqrt((c_tuple[0]-center[0])**2 +
           (c_tuple[1]-center[1])**2 + (c_tuple[2]-center[2])**2)
       distances = np.asarray([distance(coord) for n, coord in
           enumerate(c)])
       indices = np.argsort(distances)[::-1] # sort from highest to
           lowest distance - return the indices
       resname_sorted = resname_list[indices]
       distances_sorted = distances[indices]
       percent = 0
       top_residues = []
       for r in resname_sorted:
          percent += 1/len(resname_sorted) * 100
          top_residues.append(r)
             if percent >= threshold:
                break
       d_ = {i:top_residues.count(i) for i in top_residues}
       return d_
```

Now we can write the full function for acquiring data on "solvent exposed" residues. Additional annotations are provided in the associated Jupter notebook.

```
 def getData(key: str, function, index=None, plot=False) -> dict:
       dataDict = {}
       try:
          key = str(key)
       except:
          raise ValueError('Key must be able to be cast as a string
              literal.')
       if plot == True:
          fig = plt.figure()
          ax = plt.axes(projection='3d')
       pdb_entries = os.listdir(key)
       if index != None and isinstance(index, int):
          file = os.getcwd() + '/' + key + '/' + pdb_entries[index]
          pdb = ''.join(open(file, 'r').read())
          if pdbParser(pdb) == None:
                raise TypeError('pdbParser returned None for the
                    indexed PDB -- try another PDB index.')
          info, protein = pdbParser(pdb)
          coords = [atom['xyz'] for atom in protein['ATOM']]
          if plot == True:
               c = np.asarray(coords)
               ax.plot(c[:, 0], c[:, 1], c[:, 2], lw=0.5)
          plt.show()
          fDict = function(protein)
          addDictKeys(dataDict, fDict)
       elif index == None:
          for n, i in enumerate(pdb_entries):
               file = os.getcwd() + '/' + key + '/' + i
               pdb = ''.join(open(file, 'r').read())
               if pdbParser(pdb) == None:
                     continue
               info, protein = pdbParser(pdb)
               fDict = function(protein)
               addDictKeys(dataDict, fDict)
       else:
          raise ValueError('index keyword must be an integer.')
       return dataDict
```

The above function accepts user-defined analysis functions (passed as *function*). To show the convenience of this, a simple function to determine the net charge of a protein is provided and then passed to *getData()*. The Jupyter notebook also has additional information regarding rudimentary benchmarking.