```
#
# Turtle3D.py
#
# Implementation of a 3D Turtle in Python.
# Author: Eric G. Suchanek, PhD
# A part of the program Proteus, https://github.com/suchanek/proteus,
# a program for the manipulation and analysis of macromolecules
# Based on the C implementation originally authored by Eric G. Suchanek PhD
1990
#

import numpy
import math

from Bio.PDB.vectors import Vector
from Bio.PDB.vectors import calc_angle, calc_dihedral

from  proteusPy.proteusGlobals import *

_DOWN_ = -1
_UP_ = 1
_ORIENTATION_INIT = -1

"""
    Return a (left multiplying) matrix that rotates p onto q.

    :param p: moving vector
    :type p: L{Vector}

    :param q: fixed vector
    :type q: L{Vector}

    :return: rotation matrix that rotates p onto q
    :rtype: 3x3 Numeric array

    Examples
    --------
    >>> from Bio.PDB.vectors import rotmat
    >>> p, q = Vector(1, 2, 3), Vector(2, 3, 5)
    >>> r = rotmat(p, q)
    >>> print(q)
    <Vector 2.00, 3.00, 5.00>
    >>> print(p)
    <Vector 1.00, 2.00, 3.00>
    >>> p.left_multiply(r)
    <Vector 1.21, 1.82, 3.03>

    """
```



```


# Class Definition begins

class Turtle3D:
    """3D Turtle."""

    def __init__(self, name="3D_Turtle"):
        # internal rep is arrays
        self._position = numpy.array((0.0, 0.0, 0.0), "d")
        self._heading = numpy.array((1.0, 0.0, 0.0), "d")
        self._left = numpy.array((0.0, 1.0, 0.0), "d")
        self._up = numpy.array((0.0, 0.0, 1.0), "d")

        # expose these as Vectors
        self.position = Vector(0.0, 0.0, 0.0)
        self.heading = Vector(1.0, 0.0, 0.0)
        self.left = Vector(0.0, 1.0, 0.0)
        self.up = Vector(0.0, 0.0, 1.0)

        self._name = name
        self._pen = _UP_
        self._orientation = _ORIENTATION_INIT # will be set to 1 or 2 later
when used for residue building
        self._recording = False
        self._tape = []

    def copy_coords(self, source):
        '''
        Copy the Position, Heading, Left and Up coordinate system from the
input source into self.
        Argument: source: Turtle3D
        Returns: None
        '''

        # copy the Arrays
        self._position = source._position.copy()
        self._heading = source._heading.copy()
        self._left = source._left.copy()
        self._up = source._up.copy()

        # copy the Vectors - create new ones from the source arrays
        self.position = Vector(source._position)
        self.heading = Vector(source._heading)
        self.left = Vector(source._left)
        self.up = Vector(source._up)

        self._orientation = source._orientation

```

```
    def reset(self):
        """
        Reset the Turtle to be at the Origin, with correct Heading, Left
and Up vectors.
        Arguments: None
        Returns: None
        """
        self._position = numpy.array((0.0, 0.0, 0.0), "d")
        self._heading = numpy.array((1.0, 0.0, 0.0), "d")
        self._left = numpy.array((0.0, 1.0, 0.0), "d")
        self._up = numpy.array((0.0, 0.0, 1.0), "d")

        # expose these as Vectors
        self.position = Vector(0.0, 0.0, 0.0)
        self.heading = Vector(1.0, 0.0, 0.0)
        self.left = Vector(0.0, 1.0, 0.0)
        self.up = Vector(0.0, 0.0, 1.0)

        self._pen = _UP_
        self._orientation = -1
        self._recording = False
        self._tape = []

    def Orientation(self):
        return self._orientation

    def set_orientation(self, orientation):
        assert orientation == ORIENT_BACKBONE or orientation ==
ORIENT_SIDECHAIN, f'Orientation must be {ORIENT_BACKBONE} or
{ORIENT_SIDECHAIN}'
        self._orientation = orientation

    def Pen(self):
        if self._pen == _UP_:
            return('UP')
        else:
            return('Down')

    def PenUp(self):
        self._pen = _UP_

    def PenDown(self):
        self._pen = _DOWN_

    def Recording(self):
        return self._recording

    def RecordOn(self):
```



```
        self._recording = True

    def RecordOff(self):
        self._recording = False

    def ResetTape(self):
        self._tape = []

    def setPosition(self, x, y=None, z=None):
        """
        Set the Turtle's Position.
            X is either a list or tuple.
        """
        if y is None and z is None:
            # Array, list, tuple...
            if len(x) != 3:
                raise ValueError("Turtle3D: x is not a list/tuple/array of
3 numbers")
            self._position = numpy.array(x, "d")
        else:
            # Three numbers
            self._position = numpy.array((x, y, z), "d")

        self.position = Vector(self._position)
        return

    def getPosition(self) -> numpy.array:
        """
        Get the Turtle's Position.
        Return: Turtle's position (Array)
        """
        return(self._position)

    def getVPosition(self) -> Vector:
        """
        Get the Turtle's Position.
        Return: Turtle's position (Array)
        """
        return(self._position)

    def getName(self):
        """
        Get the Turtle's Position.
        """
        return(self._name)

    def setName(self, name):
        """
```

```
        Set the Turtle'Name.

        """
        self._name = name

    def move(self, distance):
        """
        Move the Turtle distance, in direction of Heading
        """
        self._position = self._position + self._heading * distance

    def roll(self, angle):
        """
        Roll the Turtle about the heading vector angle degrees
        """

        ang = angle * math.pi / 180.0
        cosang = numpy.cos(ang)
        sinang = numpy.sin(ang)

        self._up[0] = cosang * self._up[0] - sinang * self._left[0]
        self._up[1] = cosang * self._up[1] - sinang * self._left[1]
        self._up[2] = cosang * self._up[2]- sinang * self._left[2]

        self._left[0] = cosang * self._left[0] + sinang * self._up[0]
        self._left[1] = cosang * self._left[1] + sinang * self._up[1]
        self._left[2] = cosang * self._left[2] + sinang * self._up[2]

    def yaw(self, angle):
        """
        Yaw the Turtle about the up vector (180 - angle) degrees. This is
used when building molecules
        """

        ang = ((180 - angle) * math.pi) / 180.0
        cosang = numpy.cos(ang)
        sinang = numpy.sin(ang)

        self._heading[0] = cosang * self._heading[0] + sinang *
self._left[0]
        self._heading[1] = cosang * self._heading[1] + sinang *
self._left[1]
        self._heading[2] = cosang * self._heading[2] + sinang *
self._left[2]

        self._left[0] = cosang * self._left[0] - sinang * self._heading[0]
        self._left[1] = cosang * self._left[1] - sinang * self._heading[1]
        self._left[2] = cosang * self._left[2] - sinang * self._heading[2]
```



```

    def turn(self, angle):
        """
        Turn the Turtle about the up vector angle degrees.
        """

        ang = (angle * math.pi) / 180.0

        cosang = numpy.cos(ang)
        sinang = numpy.sin(ang)

        self._heading[0] = cosang * self._heading[0] + sinang *
self._left[0]
        self._heading[1] = cosang * self._heading[1] + sinang *
self._left[1]
        self._heading[2] = cosang * self._heading[2] + sinang *
self._left[2]

        self._left[0] = cosang * self._left[0] - sinang * self._heading[0]
        self._left[1] = cosang * self._left[1] - sinang * self._heading[1]
        self._left[2] = cosang * self._left[2] - sinang * self._heading[2]

    def pitch(self, angle):
        """
        pitch the Turtle about the left vector angle degrees
        """

        ang = angle * math.pi / 180.0
        cosang = numpy.cos(ang)
        sinang = numpy.sin(ang)

        self._heading[0] = self._heading[0] * cosang - self._up[0] * sinang
        self._heading[1] = self._heading[1] * cosang - self._up[1] * sinang
        self._heading[2] = self._heading[2] * cosang - self._up[2] * sinang

        self._up[0] = self._up[0] * cosang + self._heading[0] * sinang
        self._up[1] = self._up[1] * cosang + self._heading[1] * sinang
        self._up[2] = self._up[2] * cosang + self._heading[2] * sinang

    def _setHeading(self, x, y=None, z=None):
        """Set the Turtle's Heading.
            x is either a list or tuple.
        """
        if y is None and z is None:
            # Array, list, tuple...
            if len(x) != 3:
                raise ValueError("Turtle3D: x is not a list/tuple/array of
3 numbers")
```

```
            self._heading = numpy.array(x, "d")
        else:
            # Three numbers
            self._heading = numpy.array((x, y, z), "d")
        self.heading = Vector(self.heading)
        return

    def _setLeft(self, x, y=None, z=None):
        """Set the Turtle's Left.
            x is either a list or tuple.
        """
        if y is None and z is None:
            # Array, list, tuple...
            if len(x) != 3:
                raise ValueError("Turtle3D: x is not a list/tuple/array of
3 numbers")
            self._left = numpy.array(x, "d")
        else:
            # Three numbers
            self._left = numpy.array((x, y, z), "d")
        self.left = Vector(self._left)
        return

    def _setUp(self, x, y=None, z=None):
        """Set the Turtle's Up.
            x is either a list or tuple.
        """
        if y is None and z is None:
            # Array, list, tuple...
            if len(x) != 3:
                raise ValueError("Turtle3D: x is not a list/tuple/array of
3 numbers")
            self._up = numpy.array(x, "d")
        else:
            # Three numbers
            self._up = numpy.array((x, y, z), "d")
        self.up = Vector(self.up)
        return

    def unit(self, v):
        norm = numpy.linalg.norm(v)
        if norm == 0:
            return v
        return v / norm

    def orient(self, position: numpy.array, heading: numpy.array, left:
numpy.array):
        """
```



```
        Orients the turtle with Position at p1, Heading at p2 and Left at
p3
        Arguments:
            position
        """

        self._position = position

        temp = heading - position
        self._heading = self.unit(temp)
        self.heading = Vector(self._heading)

        temp = left - position
        self._left = self.unit(temp)

        temp = numpy.cross(self._heading, self._left)
        self._up = self.unit(temp)
        self.up = Vector(self._up)

        # fix left to be orthogonal
        temp = numpy.cross(self._up, self._heading)
        self._left = self.unit(temp)
        self.left = Vector(self._left)
        return

    def orient_at_residue(self, chain, resnumb, orientation):
        '''
        Orient the turtle at the specified residue from the input Chain in
        either orientation 1 or 2.

        Arguments:
            turtle: input Turtle3D
            chain: list of Residues in the model, eg: chain = model['A']
            resnumb: residue number
            orientation:
            1 - at Ca heading towards Cb with N at the left
            2 - at Ca heading towards C with N at the left
        Returns: None. Turtle internal state is modified
        '''

        assert self._orientation == 1 or self._orientation == 2,
f'orient_at_residue() requires Turtle3D to be #1 or #2'

        residue = chain[resnumb]
        assert residue is not None, f'get_backbone_from_sidechain()
requires valid residue number'

        # by this point I'm pretty confident I have coordinates
```

```
364         # we pull the actual numpy.array from the coordinates since that's
    … what the
365         # Turtle3D expects
366
367         n = residue['N'].get_vector().get_array()
368         ca = residue['CA'].get_vector().get_array()
369         cb = residue['CB'].get_vector().get_array()
370         c = residue['C'].get_vector().get_array()
371
372         if orientation == ORIENT_SIDECHAIN:
373             self.orient(ca, cb, n)
374             self.set_orientation(ORIENT_SIDECHAIN)
375         elif orientation == ORIENT_BACKBONE:
376             self.orient(ca, c, n)
377             self.set_orientation(ORIENT_BACKBONE)
378         return
379
380     def orient_from_backbone(self, n: numpy.array, ca: numpy.array, cb:
    … numpy.array, c: numpy.array, orientation):
381         '''
382         Orient the turtle at the specified residue from the input Chain in
383         either orientation 1 or 2.
384
385         Arguments:
386             turtle: input Turtle3D object
387             n: position of n atom
388             ca: position of ca atom
389             c: position of c atom
390             orientation:
391             1 - at Ca heading towards Cb with N at the left
392             2 - at Ca heading towards C with N at the left
393         Returns: None. Turtle internal state is modified
394         '''
395
396         assert orientation == 1 or orientation == 2, f'orient_at_residue()
    … requires Turtle3D to be #1 or #2'
397
398
399         _n = n.copy()
400         _ca = ca.copy()
401         _cb = cb.copy()
402         _c = c.copy()
403
404         if orientation == ORIENT_SIDECHAIN:
405             self.orient(_ca, _cb, _n)
406             self.set_orientation(ORIENT_SIDECHAIN)
407         elif orientation == ORIENT_BACKBONE:
408             self.orient(_ca, _c, _n)
```



```
409             self.set_orientation(ORIENT_BACKBONE)
410         return
411
412     def to_local(self, global_vec) -> numpy.array:
413         """
414         Returns the Turtle-centered local coordinates for input Global
    … vector (3d)
415         """
416
417         newpos = global_vec - self._position
418         dp1 = numpy.dot(self._heading, newpos)
419         dp2 = numpy.dot(self._left, newpos)
420         dp3 = numpy.dot(self._up, newpos)
421
422         result = numpy.array((dp1, dp2, dp3), "d")
423         return result
424
425     def to_localVec(self, global_vec) -> Vector:
426         """
427         Returns the Turtle-centered local coordinates for input Global
    … vector (3d)
428         """
429
430         newpos = global_vec - self._position
431         dp1 = numpy.dot(self._heading, newpos)
432         dp2 = numpy.dot(self._left, newpos)
433         dp3 = numpy.dot(self._up, newpos)
434
435         return Vector(dp1, dp2, dp3)
436
437     def to_global(self, local) -> numpy.array:
438         """
439         Returns the global coordinates for input local vector (3d)
440         """
441
442         p1 = self._position[0] + self._heading[0] * local[0] +
    … self._left[0] * local[1] + self._up[0] * local[2]
443         p2 = self._position[1] + self._heading[1] * local[0] +
    … self._left[1] * local[1] + self._up[1] * local[2]
444         p3 = self._position[2] + self._heading[2] * local[0] +
    … self._left[2] * local[1] * self._up[2] * local[2]
445
446         return numpy.array((p1, p2, p3), "d")
447
448     def to_globalVec(self, local) -> Vector:
449         """
450         Returns the global coordinates for input local vector (3d)
451         """
```

```
452
453         p1 = self._position[0] + self._heading[0] * local[0] +
    … self._left[0] * local[1] + self._up[0] * local[2]
454         p2 = self._position[1] + self._heading[1] * local[0] +
    … self._left[1] * local[1] + self._up[1] * local[2]
455         p3 = self._position[2] + self._heading[2] * local[0] +
    … self._left[2] * local[1] * self._up[2] * local[2]
456
457         return Vector(p1, p2, p3)
458
459     def __repr__(self):
460         """Return Turtle 3D coordinates."""
461         return f"<Turtle: {self._name}\n Position: {self._position},\n
    … Heading: {self._heading} \n Left: {self._left} \n Up: {self._up}\n
    … Orientation: {self._orientation}\n Pen: {self.Pen()} \n Recording:
    … {self._recording}>"
462
463     def bbone_to_schain(self):
464         '''
465         Function requires turtle to be in orientation #2 (at alpha carbon,
466         headed towards carbonyl, with nitrogen on left) and converts to
    … orientation #1
467         (at alpha c, headed to beta carbon, with nitrogen on left.
468
469         Arguments:
470             turtle: Turtle3D object in orientation #2
471
472         Returns: modified Turtle3D
473         '''
474
475         assert self._orientation == 2, f'bbone_to_schain() requires
    … Turtle3D to be in orientation #2'
476
477         self.roll(240.0)
478         self.pitch(180.0)
479         self.yaw(110.0)
480         self.roll(240.0)
481         self.set_orientation(1) # sets the orientation flag
482
483
484     def schain_to_bbone(self):
485         '''
486         Function requires turtle to be in orientation #1 (at alpha c,
    … headed to beta carbon, with nitrogen on left)
487         and converts to orientation #2 (at alpha carbon, headed towards
    … carbonyl, with nitrogen on left).
488
489         Arguments:
```



```
490             None
491         Returns: modified Turtle3D
492         '''
493
494         assert self._orientation == 1, f'schain_to_bbone() requires
    … Turtle3D to be in orientation #1'
495
496         self.pitch(180.0)
497         self.roll(240.0)
498         self.yaw(110.0)
499         self.roll(120.0)
500         self.set_orientation(2) # sets the orientation flag
501         return
502
503 # End of file
504
```