

```
1 #
2 # Turtle3D.py
3 #
4 # Implementation of a 3D Turtle in Python.
5 # Author: Eric G. Suchanek, PhD
6 # A part of the program Proteus, https://github.com/suchanek/proteus,
7 # a program for the manipulation and analysis of macromolecules
8 # Based on the C implementation originally authored by Eric G. Suchanek PhD
9 1990
10 #
11 import numpy
12 import math
13
14 from Bio.PDB.vectors import Vector
15 from Bio.PDB.vectors import calc_angle, calc_dihedral
16
17 from proteusPy.proteusGlobals import *
18
19 _DOWN_ = -1
20 _UP_ = 1
21 _ORIENTATION_INIT = -1
22
23 """
24     Return a (left multiplying) matrix that rotates p onto q.
25
26     :param p: moving vector
27     :type p: L{Vector}
28
29     :param q: fixed vector
30     :type q: L{Vector}
31
32     :return: rotation matrix that rotates p onto q
33     :rtype: 3x3 Numeric array
34
35     Examples
36     -----
37     >>> from Bio.PDB.vectors import rotmat
38     >>> p, q = Vector(1, 2, 3), Vector(2, 3, 5)
39     >>> r = rotmat(p, q)
40     >>> print(q)
41     <Vector 2.00, 3.00, 5.00>
42     >>> print(p)
43     <Vector 1.00, 2.00, 3.00>
44     >>> p.left_multiply(r)
45     <Vector 1.21, 1.82, 3.03>
46
47     """
```

```
48
49 # Class Definition begins
50
51 class Turtle3D:
52     """3D Turtle."""
53
54     def __init__(self, name="3D_Turtle"):
55         # internal rep is arrays
56         self._position = numpy.array((0.0, 0.0, 0.0), "d")
57         self._heading = numpy.array((1.0, 0.0, 0.0), "d")
58         self._left = numpy.array((0.0, 1.0, 0.0), "d")
59         self._up = numpy.array((0.0, 0.0, 1.0), "d")
60
61         # expose these as Vectors
62         self.position = Vector(0.0, 0.0, 0.0)
63         self.heading = Vector(1.0, 0.0, 0.0)
64         self.left = Vector(0.0, 1.0, 0.0)
65         self.up = Vector(0.0, 0.0, 1.0)
66
67         self._name = name
68         self._pen = _UP_
69         self._orientation = _ORIENTATION_INIT # will be set to 1 or 2 later
70
71         when used for residue building
72         self._recording = False
73         self._tape = []
74
75     def copy_coords(self, source):
76         """
77         Copy the Position, Heading, Left and Up coordinate system from the
78         input source into self.
79         Argument: source: Turtle3D
80         Returns: None
81         """
82
83         # copy the Arrays
84         self._position = source._position.copy()
85         self._heading = source._heading.copy()
86         self._left = source._left.copy()
87         self._up = source._up.copy()
88
89         # copy the Vectors - create new ones from the source arrays
90         self.position = Vector(source._position)
91         self.heading = Vector(source._heading)
92         self.left = Vector(source._left)
93         self.up = Vector(source._up)
94
95         self._orientation = source._orientation
```

```
94     def reset(self):
95         """
96         Reset the Turtle to be at the Origin, with correct Heading, Left
... and Up vectors.
97         Arguments: None
98         Returns: None
99         """
100         self._position = numpy.array((0.0, 0.0, 0.0), "d")
101         self._heading = numpy.array((1.0, 0.0, 0.0), "d")
102         self._left = numpy.array((0.0, 1.0, 0.0), "d")
103         self._up = numpy.array((0.0, 0.0, 1.0), "d")
104
105         # expose these as Vectors
106         self.position = Vector(0.0, 0.0, 0.0)
107         self.heading = Vector(1.0, 0.0, 0.0)
108         self.left = Vector(0.0, 1.0, 0.0)
109         self.up = Vector(0.0, 0.0, 1.0)
110
111         self._pen = _UP_
112         self._orientation = -1
113         self._recording = False
114         self._tape = []
115
116     def Orientation(self):
117         return self._orientation
118
119     def set_orientation(self, orientation):
120         assert orientation == ORIENT_BACKBONE or orientation ==
... ORIENT_SIDECHAIN, f'Orientation must be {ORIENT_BACKBONE} or
... {ORIENT_SIDECHAIN}'
121         self._orientation = orientation
122
123     def Pen(self):
124         if self._pen == _UP_:
125             return('UP')
126         else:
127             return('Down')
128
129     def PenUp(self):
130         self._pen = _UP_
131
132     def PenDown(self):
133         self._pen = _DOWN_
134
135     def Recording(self):
136         return self._recording
137
138     def RecordOn(self):
```

```
139         self._recording = True
140
141     def RecordOff(self):
142         self._recording = False
143
144     def ResetTape(self):
145         self._tape = []
146
147     def setPosition(self, x, y=None, z=None):
148         """
149         Set the Turtle's Position.
150         X is either a list or tuple.
151         """
152         if y is None and z is None:
153             # Array, list, tuple...
154             if len(x) != 3:
155                 raise ValueError("Turtle3D: x is not a list/tuple/array of
... 3 numbers")
156             self._position = numpy.array(x, "d")
157         else:
158             # Three numbers
159             self._position = numpy.array((x, y, z), "d")
160
161         self.position = Vector(self._position)
162         return
163
164     def getPosition(self) -> numpy.array:
165         """
166         Get the Turtle's Position.
167         Return: Turtle's position (Array)
168         """
169         return(self._position)
170
171     def getVPosition(self) -> Vector:
172         """
173         Get the Turtle's Position.
174         Return: Turtle's position (Array)
175         """
176         return(self._position)
177
178     def getName(self):
179         """
180         Get the Turtle's Position.
181         """
182         return(self._name)
183
184     def setName(self, name):
185         """
```

```
186     Set the Turtle'Name.
187
188     """
189     self._name = name
190
191     def move(self, distance):
192         """
193         Move the Turtle distance, in direction of Heading
194         """
195         self._position = self._position + self._heading * distance
196
197     def roll(self, angle):
198         """
199         Roll the Turtle about the heading vector angle degrees
200         """
201
202         ang = angle * math.pi / 180.0
203         cosang = numpy.cos(ang)
204         sinang = numpy.sin(ang)
205
206         self._up[0] = cosang * self._up[0] - sinang * self._left[0]
207         self._up[1] = cosang * self._up[1] - sinang * self._left[1]
208         self._up[2] = cosang * self._up[2] - sinang * self._left[2]
209
210         self._left[0] = cosang * self._left[0] + sinang * self._up[0]
211         self._left[1] = cosang * self._left[1] + sinang * self._up[1]
212         self._left[2] = cosang * self._left[2] + sinang * self._up[2]
213
214     def yaw(self, angle):
215         """
216         Yaw the Turtle about the up vector (180 - angle) degrees. This is
217         used when building molecules
218         """
219
220         ang = ((180 - angle) * math.pi) / 180.0
221         cosang = numpy.cos(ang)
222         sinang = numpy.sin(ang)
223
224         self._heading[0] = cosang * self._heading[0] + sinang *
225 self._left[0]
226         self._heading[1] = cosang * self._heading[1] + sinang *
227 self._left[1]
228         self._heading[2] = cosang * self._heading[2] + sinang *
229 self._left[2]
230
231         self._left[0] = cosang * self._left[0] - sinang * self._heading[0]
232         self._left[1] = cosang * self._left[1] - sinang * self._heading[1]
233         self._left[2] = cosang * self._left[2] - sinang * self._heading[2]
```

```
230
231     def turn(self, angle):
232         """
233         Turn the Turtle about the up vector angle degrees.
234         """
235
236         ang = (angle * math.pi) / 180.0
237
238         cosang = numpy.cos(ang)
239         sinang = numpy.sin(ang)
240
241         self._heading[0] = cosang * self._heading[0] + sinang *
242 self._left[0]
243         self._heading[1] = cosang * self._heading[1] + sinang *
244 self._left[1]
245         self._heading[2] = cosang * self._heading[2] + sinang *
246 self._left[2]
247
248         self._left[0] = cosang * self._left[0] - sinang * self._heading[0]
249         self._left[1] = cosang * self._left[1] - sinang * self._heading[1]
250         self._left[2] = cosang * self._left[2] - sinang * self._heading[2]
251
252     def pitch(self, angle):
253         """
254         pitch the Turtle about the left vector angle degrees
255         """
256
257         ang = angle * math.pi / 180.0
258         cosang = numpy.cos(ang)
259         sinang = numpy.sin(ang)
260
261         self._heading[0] = self._heading[0] * cosang - self._up[0] * sinang
262         self._heading[1] = self._heading[1] * cosang - self._up[1] * sinang
263         self._heading[2] = self._heading[2] * cosang - self._up[2] * sinang
264
265         self._up[0] = self._up[0] * cosang + self._heading[0] * sinang
266         self._up[1] = self._up[1] * cosang + self._heading[1] * sinang
267         self._up[2] = self._up[2] * cosang + self._heading[2] * sinang
268
269     def _setHeading(self, x, y=None, z=None):
270         """Set the Turtle's Heading.
271         x is either a list or tuple.
272         """
273         if y is None and z is None:
274             # Array, list, tuple...
275             if len(x) != 3:
276                 raise ValueError("Turtle3D: x is not a list/tuple/array of
277 3 numbers")
```

```
274         self._heading = numpy.array(x, "d")
275     else:
276         # Three numbers
277         self._heading = numpy.array((x, y, z), "d")
278     self.heading = Vector(self.heading)
279     return
280
281     def _setLeft(self, x, y=None, z=None):
282         """Set the Turtle's Left.
283         x is either a list or tuple.
284         """
285         if y is None and z is None:
286             # Array, list, tuple...
287             if len(x) != 3:
288                 raise ValueError("Turtle3D: x is not a list/tuple/array of
... 3 numbers")
289             self._left = numpy.array(x, "d")
290         else:
291             # Three numbers
292             self._left = numpy.array((x, y, z), "d")
293         self.left = Vector(self._left)
294         return
295
296     def _setUp(self, x, y=None, z=None):
297         """Set the Turtle's Up.
298         x is either a list or tuple.
299         """
300         if y is None and z is None:
301             # Array, list, tuple...
302             if len(x) != 3:
303                 raise ValueError("Turtle3D: x is not a list/tuple/array of
... 3 numbers")
304             self._up = numpy.array(x, "d")
305         else:
306             # Three numbers
307             self._up = numpy.array((x, y, z), "d")
308         self.up = Vector(self._up)
309         return
310
311     def unit(self, v):
312         norm = numpy.linalg.norm(v)
313         if norm == 0:
314             return v
315         return v / norm
316
317     def orient(self, position: numpy.array, heading: numpy.array, left:
... numpy.array):
318         """
```

```
319     p3 Orients the turtle with Position at p1, Heading at p2 and Left at
... p3
320
321     Arguments:
322         position
323         """
324
325     self._position = position
326
327     temp = heading - position
328     self._heading = self.unit(temp)
329     self.heading = Vector(self._heading)
330
331     temp = left - position
332     self._left = self.unit(temp)
333
334     temp = numpy.cross(self._heading, self._left)
335     self._up = self.unit(temp)
336     self.up = Vector(self._up)
337
338     # fix left to be orthogonal
339     temp = numpy.cross(self._up, self._heading)
340     self._left = self.unit(temp)
341     self.left = Vector(self._left)
342     return
343
344     def orient_at_residue(self, chain, resnumb, orientation):
345         """
346         Orient the turtle at the specified residue from the input Chain in
347         either orientation 1 or 2.
348
349         Arguments:
350             turtle: input Turtle3D
351             chain: list of Residues in the model, eg: chain = model['A']
352             resnumb: residue number
353             orientation:
354                 1 - at Ca heading towards Cb with N at the left
355                 2 - at Ca heading towards C with N at the left
356         Returns: None. Turtle internal state is modified
357         """
358
359         assert self._orientation == 1 or self._orientation == 2,
... f'orient_at_residue() requires Turtle3D to be #1 or #2'
359
360         residue = chain[resnumb]
361         assert residue is not None, f'get_backbone_from_sidechain()
... requires valid residue number'
362
363         # 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         _n = n.copy()
399         _ca = ca.copy()
400         _cb = cb.copy()
401         _c = c.copy()
402
403         if orientation == ORIENT_SIDECHAIN:
404             self.orient(_ca, _cb, _n)
405             self.set_orientation(ORIENT_SIDECHAIN)
406         elif orientation == ORIENT_BACKBONE:
407             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     p1 = self._position[0] + self._heading[0] * local[0] +
453     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
```