

Coverage for manimlib/mobject/mobject.py : 52%



706 statements 365 run 341 missing 0 excluded

```

1  from functools import reduce
2  import copy
3  import itertools as it
4  import operator as op
5  import os
6  import random
7  import sys
8
9  from colour import Color
10 import numpy as np
11
12 import manimlib.constants as consts
13 from manimlib.constants import *
14 from manimlib.container.container import Container
15 from manimlib.utils.color import color_gradient
16 from manimlib.utils.color import interpolate_color
17 from manimlib.utils.iterables import list_update
18 from manimlib.utils.iterables import remove_list_redundancies
19 from manimlib.utils.paths import straight_path
20 from manimlib.utils.simple_functions import get_parameters
21 from manimlib.utils.space_ops import angle_of_vector
22 from manimlib.utils.space_ops import get_norm
23 from manimlib.utils.space_ops import rotation_matrix
24
25
26 # TODO: Explain array_attrs
27
28 class Mobject(Container):
29     """
30     Mathematical Object
31     """
32     CONFIG = {
33         "color": WHITE,
34         "name": None,
35         "dim": 3,
36         "target": None,
37     }
38
39     def __init__(self, **kwargs):
40         Container.__init__(self, **kwargs)
41         self.submobjects = []
42         self.color = Color(self.color)
43         if self.name is None:
44             self.name = self.__class__.__name__
45         self.updaters = []
46         self.updating_suspended = False
47         self.reset_points()
48         self.generate_points()
49         self.init_colors()
50
51     def __str__(self):
52         return str(self.name)
53
54     def reset_points(self):
55         self.points = np.zeros((0, self.dim))
56
57     def init_colors(self):
58         # For subclasses
59         pass
60
61     def generate_points(self):
62         # Typically implemented in subclass, unless purposefully left blank
63         pass
64
65     def add(self, *mobjects):
66         if self in mobjects:

```

```

67         raise Exception("Mobject cannot contain self")
68     self.submobjects = list_update(self.submobjects, mobjects)
69     return self
70
71     def add_to_back(self, *mobjects):
72         self.remove(*mobjects)
73         self.submobjects = list(mobjects) + self.submobjects
74         return self
75
76     def remove(self, *mobjects):
77         for mobject in mobjects:
78             if mobject in self.submobjects:
79                 self.submobjects.remove(mobject)
80         return self
81
82     def get_array_attrs(self):
83         return ["points"]
84
85     def digest_mobject_attrs(self):
86         """
87         Ensures all attributes which are mobjects are included
88         in the submobjects list.
89         """
90         mobject_attrs = [x for x in list(self.__dict__.values()) if isinstance(x, Mobject)]
91         self.submobjects = list_update(self.submobjects, mobject_attrs)
92         return self
93
94     def apply_over_attr_arrays(self, func):
95         for attr in self.get_array_attrs():
96             setattr(self, attr, func(getattr(self, attr)))
97         return self
98
99     # Displaying
100
101     def get_image(self, camera=None):
102         if camera is None:
103             from manimlib.camera.camera import Camera
104             camera = Camera()
105             camera.capture_mobject(self)
106             return camera.get_image()
107
108     def show(self, camera=None):
109         self.get_image(camera=camera).show()
110
111     def save_image(self, name=None):
112         self.get_image().save(
113             os.path.join(consts.VIDEO_DIR, (name or str(self)) + ".png")
114         )
115
116     def copy(self):
117         # TODO, either justify reason for shallow copy, or
118         # remove this redundancy everywhere
119         # return self.deepcopy()
120
121         copy_mobject = copy.copy(self)
122         copy_mobject.points = np.array(self.points)
123         copy_mobject.submobjects = [
124             submob.copy() for submob in self.submobjects
125         ]
126         copy_mobject.updaters = list(self.updaters)
127         family = self.get_family()
128         for attr, value in list(self.__dict__.items()):
129             if isinstance(value, Mobject) and value in family and value is not self:
130                 setattr(copy_mobject, attr, value.copy())
131             if isinstance(value, np.ndarray):
132                 setattr(copy_mobject, attr, np.array(value))
133         return copy_mobject
134
135     def deepcopy(self):
136         return copy.deepcopy(self)
137
138     def generate_target(self, use_deepcopy=False):
139         self.target = None # Prevent exponential explosion

```

```

140         if use_deepcopy:
141             self.target = self.deepcopy()
142         else:
143             self.target = self.copy()
144         return self.target
145
146     # Updating
147
148     def update(self, dt=0, recursive=True):
149         if self.updating_suspended:
150             return self
151         for updater in self.updaters:
152             parameters = get_parameters(updater)
153             if "dt" in parameters:
154                 updater(self, dt)
155             else:
156                 updater(self)
157         if recursive:
158             for submob in self.submobjects:
159                 submob.update(dt, recursive)
160         return self
161
162     def get_time_based_updaters(self):
163         return [
164             updater for updater in self.updaters
165             if "dt" in get_parameters(updater)
166         ]
167
168     def has_time_based_updater(self):
169         for updater in self.updaters:
170             if "dt" in get_parameters(updater):
171                 return True
172         return False
173
174     def get_updaters(self):
175         return self.updaters
176
177     def get_family_updaters(self):
178         return list(it.chain(*[
179             sm.get_updaters()
180             for sm in self.get_family()
181         ]))
182
183     def add_updater(self, update_function, index=None, call_updater=True):
184         if index is None:
185             self.updaters.append(update_function)
186         else:
187             self.updaters.insert(index, update_function)
188         if call_updater:
189             self.update(0)
190         return self
191
192     def remove_updater(self, update_function):
193         while update_function in self.updaters:
194             self.updaters.remove(update_function)
195         return self
196
197     def clear_updaters(self, recursive=True):
198         self.updaters = []
199         if recursive:
200             for submob in self.submobjects:
201                 submob.clear_updaters()
202         return self
203
204     def match_updaters(self, mobject):
205         self.clear_updaters()
206         for updater in mobject.get_updaters():
207             self.add_updater(updater)
208         return self
209
210     def suspend_updating(self, recursive=True):
211         self.updating_suspended = True
212         if recursive:

```

```

213         for submob in self.submobjects:
214             submob.suspend_updating(recursive)
215         return self
216
217     def resume_updating(self, recursive=True):
218         self.updating_suspended = False
219         if recursive:
220             for submob in self.submobjects:
221                 submob.resume_updating(recursive)
222         self.update(dt=0, recursive=recursive)
223         return self
224
225     # Transforming operations
226
227     def apply_to_family(self, func):
228         for mob in self.family_members_with_points():
229             func(mob)
230
231     def shift(self, *vectors):
232         total_vector = reduce(op.add, vectors)
233         for mob in self.family_members_with_points():
234             mob.points = mob.points.astype('float')
235             mob.points += total_vector
236         return self
237
238     def scale(self, scale_factor, **kwargs):
239         """
240         Default behavior is to scale about the center of the mobject.
241         The argument about_edge can be a vector, indicating which side of
242         the mobject to scale about, e.g., mob.scale(about_edge = RIGHT)
243         scales about mob.get_right().
244
245         Otherwise, if about_point is given a value, scaling is done with
246         respect to that point.
247         """
248         self.apply_points_function_about_point(
249             lambda points: scale_factor * points, **kwargs
250         )
251         return self
252
253     def rotate_about_origin(self, angle, axis=OUT, axes=[]):
254         return self.rotate(angle, axis, about_point=ORIGIN)
255
256     def rotate(self, angle, axis=OUT, **kwargs):
257         rot_matrix = rotation_matrix(angle, axis)
258         self.apply_points_function_about_point(
259             lambda points: np.dot(points, rot_matrix.T),
260             **kwargs
261         )
262         return self
263
264     def flip(self, axis=UP, **kwargs):
265         return self.rotate(TAU / 2, axis, **kwargs)
266
267     def stretch(self, factor, dim, **kwargs):
268         def func(points):
269             points[:, dim] *= factor
270             return points
271         self.apply_points_function_about_point(func, **kwargs)
272         return self
273
274     def apply_function(self, function, **kwargs):
275         # Default to applying matrix about the origin, not mobjects center
276         if len(kwargs) == 0:
277             kwargs["about_point"] = ORIGIN
278         self.apply_points_function_about_point(
279             lambda points: np.apply_along_axis(function, 1, points),
280             **kwargs
281         )
282         return self
283
284     def apply_function_to_position(self, function):
285         self.move_to(function(self.get_center()))

```

```

286         return self
287
288     def apply_function_to_submobject_positions(self, function):
289         for submob in self.submobjects:
290             submob.apply_function_to_position(function)
291         return self
292
293     def apply_matrix(self, matrix, **kwargs):
294         # Default to applying matrix about the origin, not mobjects center
295         if ("about_point" not in kwargs) and ("about_edge" not in kwargs):
296             kwargs["about_point"] = ORIGIN
297         full_matrix = np.identity(self.dim)
298         matrix = np.array(matrix)
299         full_matrix[:matrix.shape[0], :matrix.shape[1]] = matrix
300         self.apply_points_function_about_point(
301             lambda points: np.dot(points, full_matrix.T),
302             **kwargs
303         )
304         return self
305
306     def apply_complex_function(self, function, **kwargs):
307         def R3_func(point):
308             x, y, z = point
309             xy_complex = function(complex(x, y))
310             return [
311                 xy_complex.real,
312                 xy_complex.imag,
313                 z
314             ]
315         return self.apply_function(R3_func)
316
317     def wag(self, direction=RIGHT, axis=DOWN, wag_factor=1.0):
318         for mob in self.family_members_with_points():
319             alphas = np.dot(mob.points, np.transpose(axis))
320             alphas -= min(alphas)
321             alphas /= max(alphas)
322             alphas = alphas*wag_factor
323             mob.points += np.dot(
324                 alphas.reshape((len(alphas), 1)),
325                 np.array(direction).reshape((1, mob.dim))
326             )
327         return self
328
329     def reverse_points(self):
330         for mob in self.family_members_with_points():
331             mob.apply_over_attr_arrays(
332                 lambda arr: np.array(list(reversed(arr)))
333             )
334         return self
335
336     def repeat(self, count):
337         """
338         This can make transition animations nicer
339         """
340         def repeat_array(array):
341             return reduce(
342                 lambda a1, a2: np.append(a1, a2, axis=0),
343                 [array] * count
344             )
345         for mob in self.family_members_with_points():
346             mob.apply_over_attr_arrays(repeat_array)
347         return self
348
349         # In place operations.
350         # Note, much of these are now redundant with default behavior of
351         # above methods
352
353     def apply_points_function_about_point(self, func, about_point=None, about_edge=None):
354         if about_point is None:
355             if about_edge is None:
356                 about_edge = ORIGIN
357             about_point = self.get_critical_point(about_edge)
358         for mob in self.family_members_with_points():

```

```

359         mob.points -= about_point
360         mob.points = func(mob.points)
361         mob.points += about_point
362     return self
363
364     def rotate_in_place(self, angle, axis=OUT):
365         # redundant with default behavior of rotate now.
366         return self.rotate(angle, axis=axis)
367
368     def scale_in_place(self, scale_factor, **kwargs):
369         # Redundant with default behavior of scale now.
370         return self.scale(scale_factor, **kwargs)
371
372     def scale_about_point(self, scale_factor, point):
373         # Redundant with default behavior of scale now.
374         return self.scale(scale_factor, about_point=point)
375
376     def pose_at_angle(self, **kwargs):
377         self.rotate(TAU / 14, RIGHT + UP, **kwargs)
378         return self
379
380     # Positioning methods
381
382     def center(self):
383         self.shift(-self.get_center())
384         return self
385
386     def align_on_border(self, direction, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
387         """
388         Direction just needs to be a vector pointing towards side or
389         corner in the 2d plane.
390         """
391         target_point = np.sign(direction) * (FRAME_X_RADIUS, FRAME_Y_RADIUS, 0)
392         point_to_align = self.get_critical_point(direction)
393         shift_val = target_point - point_to_align - buff * np.array(direction)
394         shift_val = shift_val * abs(np.sign(direction))
395         self.shift(shift_val)
396         return self
397
398     def to_corner(self, corner=LEFT + DOWN, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
399         return self.align_on_border(corner, buff)
400
401     def to_edge(self, edge=LEFT, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
402         return self.align_on_border(edge, buff)
403
404     def next_to(self, mobject_or_point,
405                 direction=RIGHT,
406                 buff=DEFAULT_MOBJECT_TO_MOBJECT_BUFFER,
407                 aligned_edge=ORIGIN,
408                 submobject_to_align=None,
409                 index_of_submobject_to_align=None,
410                 coor_mask=np.array([1, 1, 1]),
411                 ):
412         if isinstance(mobject_or_point, Mobject):
413             mob = mobject_or_point
414             if index_of_submobject_to_align is not None:
415                 target_aligner = mob[index_of_submobject_to_align]
416             else:
417                 target_aligner = mob
418             target_point = target_aligner.get_critical_point(
419                 aligned_edge + direction
420             )
421         else:
422             target_point = mobject_or_point
423         if submobject_to_align is not None:
424             aligner = submobject_to_align
425         elif index_of_submobject_to_align is not None:
426             aligner = self[index_of_submobject_to_align]
427         else:
428             aligner = self
429         point_to_align = aligner.get_critical_point(aligned_edge - direction)
430         self.shift((target_point - point_to_align +
431                   buff * direction) * coor_mask)

```

```

432         return self
433
434     def shift_onto_screen(self, **kwargs):
435         space_lengths = [FRAME_X_RADIUS, FRAME_Y_RADIUS]
436         for vect in UP, DOWN, LEFT, RIGHT:
437             dim = np.argmax(np.abs(vect))
438             buff = kwargs.get("buff", DEFAULT_MOBJECT_TO_EDGE_BUFFER)
439             max_val = space_lengths[dim] - buff
440             edge_center = self.get_edge_center(vect)
441             if np.dot(edge_center, vect) > max_val:
442                 self.to_edge(vect, **kwargs)
443         return self
444
445     def is_off_screen(self):
446         if self.get_left()[0] > FRAME_X_RADIUS:
447             return True
448         if self.get_right()[0] < -FRAME_X_RADIUS:
449             return True
450         if self.get_bottom()[1] > FRAME_Y_RADIUS:
451             return True
452         if self.get_top()[1] < -FRAME_Y_RADIUS:
453             return True
454         return False
455
456     def stretch_about_point(self, factor, dim, point):
457         return self.stretch(factor, dim, about_point=point)
458
459     def stretch_in_place(self, factor, dim):
460         # Now redundant with stretch
461         return self.stretch(factor, dim)
462
463     def rescale_to_fit(self, length, dim, stretch=False, **kwargs):
464         old_length = self.length_over_dim(dim)
465         if old_length == 0:
466             return self
467         if stretch:
468             self.stretch(length / old_length, dim, **kwargs)
469         else:
470             self.scale(length / old_length, **kwargs)
471         return self
472
473     def stretch_to_fit_width(self, width, **kwargs):
474         return self.rescale_to_fit(width, 0, stretch=True, **kwargs)
475
476     def stretch_to_fit_height(self, height, **kwargs):
477         return self.rescale_to_fit(height, 1, stretch=True, **kwargs)
478
479     def stretch_to_fit_depth(self, depth, **kwargs):
480         return self.rescale_to_fit(depth, 2, stretch=True, **kwargs)
481
482     def set_width(self, width, stretch=False, **kwargs):
483         return self.rescale_to_fit(width, 0, stretch=stretch, **kwargs)
484
485     def set_height(self, height, stretch=False, **kwargs):
486         return self.rescale_to_fit(height, 1, stretch=stretch, **kwargs)
487
488     def set_depth(self, depth, stretch=False, **kwargs):
489         return self.rescale_to_fit(depth, 2, stretch=stretch, **kwargs)
490
491     def set_coord(self, value, dim, direction=ORIGIN):
492         curr = self.get_coord(dim, direction)
493         shift_vect = np.zeros(self.dim)
494         shift_vect[dim] = value - curr
495         self.shift(shift_vect)
496         return self
497
498     def set_x(self, x, direction=ORIGIN):
499         return self.set_coord(x, 0, direction)
500
501     def set_y(self, y, direction=ORIGIN):
502         return self.set_coord(y, 1, direction)
503
504     def set_z(self, z, direction=ORIGIN):

```

```

505         return self.set_coord(z, 2, direction)
506
507     def space_out_subobjects(self, factor=1.5, **kwargs):
508         self.scale(factor, **kwargs)
509         for submob in self.submobjects:
510             submob.scale(1. / factor)
511         return self
512
513     def move_to(self, point_or_mobject, aligned_edge=ORIGIN,
514                 coor_mask=np.array([1, 1, 1])):
515         if isinstance(point_or_mobject, Mobject):
516             target = point_or_mobject.get_critical_point(aligned_edge)
517         else:
518             target = point_or_mobject
519         point_to_align = self.get_critical_point(aligned_edge)
520         self.shift((target - point_to_align) * coor_mask)
521         return self
522
523     def replace(self, mobject, dim_to_match=0, stretch=False):
524         if not mobject.get_num_points() and not mobject.submobjects:
525             raise Warning("Attempting to replace mobject with no points")
526         return self
527         if stretch:
528             self.stretch_to_fit_width(mobject.get_width())
529             self.stretch_to_fit_height(mobject.get_height())
530         else:
531             self.rescale_to_fit(
532                 mobject.length_over_dim(dim_to_match),
533                 dim_to_match,
534                 stretch=False
535             )
536         self.shift(mobject.get_center() - self.get_center())
537         return self
538
539     def surround(self, mobject,
540                 dim_to_match=0,
541                 stretch=False,
542                 buff=MED_SMALL_BUFF):
543         self.replace(mobject, dim_to_match, stretch)
544         length = mobject.length_over_dim(dim_to_match)
545         self.scale_in_place((length + buff) / length)
546         return self
547
548     def put_start_and_end_on(self, start, end):
549         curr_start, curr_end = self.get_start_and_end()
550         curr_vect = curr_end - curr_start
551         if np.all(curr_vect == 0):
552             raise Exception("Cannot position endpoints of closed loop")
553         target_vect = end - start
554         self.scale(
555             get_norm(target_vect) / get_norm(curr_vect),
556             about_point=curr_start,
557         )
558         self.rotate(
559             angle_of_vector(target_vect) -
560             angle_of_vector(curr_vect),
561             about_point=curr_start
562         )
563         self.shift(start - curr_start)
564         return self
565
566     # Background rectangle
567     def add_background_rectangle(self, color=BLACK, opacity=0.75, **kwargs):
568         # TODO, this does not behave well when the mobject has points,
569         # since it gets displayed on top
570         from manimlib.mobject.shape_matchers import BackgroundRectangle
571         self.background_rectangle = BackgroundRectangle(
572             self, color=color,
573             fill_opacity=opacity,
574             **kwargs
575         )
576         self.add_to_back(self.background_rectangle)
577         return self

```



```

578
579 def add_background_rectangle_to_submobjects(self, **kwargs):
580     for submobject in self.submobjects:
581         submobject.add_background_rectangle(**kwargs)
582     return self
583
584 def add_background_rectangle_to_family_members_with_points(self, **kwargs):
585     for mob in self.family_members_with_points():
586         mob.add_background_rectangle(**kwargs)
587     return self
588
589 # Color functions
590
591 def set_color(self, color=YELLOW_C, family=True):
592     """
593     Condition is function which takes in one arguments, (x, y, z).
594     Here it just recurses to submobjects, but in subclasses this
595     should be further implemented based on the the inner workings
596     of color
597     """
598     if family:
599         for submob in self.submobjects:
600             submob.set_color(color, family=family)
601     self.color = color
602     return self
603
604 def set_color_by_gradient(self, *colors):
605     self.set_submobject_colors_by_gradient(*colors)
606     return self
607
608 def set_colors_by_radial_gradient(self, center=None, radius=1, inner_color=WHITE, outer_color=BLACK):
609     self.set_submobject_colors_by_radial_gradient(
610         center, radius, inner_color, outer_color)
611     return self
612
613 def set_submobject_colors_by_gradient(self, *colors):
614     if len(colors) == 0:
615         raise Exception("Need at least one color")
616     elif len(colors) == 1:
617         return self.set_color(*colors)
618
619     mobs = self.family_members_with_points()
620     new_colors = color_gradient(colors, len(mobs))
621
622     for mob, color in zip(mobs, new_colors):
623         mob.set_color(color, family=False)
624     return self
625
626 def set_submobject_colors_by_radial_gradient(self, center=None, radius=1, inner_color=WHITE, outer_color=BLACK):
627     if center is None:
628         center = self.get_center()
629
630     for mob in self.family_members_with_points():
631         t = get_norm(mob.get_center() - center) / radius
632         t = min(t, 1)
633         mob_color = interpolate_color(inner_color, outer_color, t)
634         mob.set_color(mob_color, family=False)
635
636     return self
637
638 def to_original_color(self):
639     self.set_color(self.color)
640     return self
641
642 def fade_to(self, color, alpha, family=True):
643     if self.get_num_points() > 0:
644         new_color = interpolate_color(
645             self.get_color(), color, alpha
646         )
647         self.set_color(new_color, family=False)
648     if family:
649         for submob in self.submobjects:
650             submob.fade_to(color, alpha)

```

```

651         return self
652
653     def fade(self, darkness=0.5, family=True):
654         if family:
655             for submob in self.submobjects:
656                 submob.fade(darkness, family)
657         return self
658
659     def get_color(self):
660         return self.color
661
662     ##
663
664     def save_state(self, use_deepcopy=False):
665         if hasattr(self, "saved_state"):
666             # Prevent exponential growth of data
667             self.saved_state = None
668         if use_deepcopy:
669             self.saved_state = self.deepcopy()
670         else:
671             self.saved_state = self.copy()
672         return self
673
674     def restore(self):
675         if not hasattr(self, "saved_state") or self.saved_state is None:
676             raise Exception("Trying to restore without having saved")
677         self.become(self.saved_state)
678         return self
679
680     ##
681
682     def reduce_across_dimension(self, points_func, reduce_func, dim):
683         points = self.get_all_points()
684         if points is None or len(points) == 0:
685             # Note, this default means things like empty VGroups
686             # will appear to have a center at [0, 0, 0]
687             return 0
688         values = points_func(points[:, dim])
689         return reduce_func(values)
690
691     def nonempty_submobjects(self):
692         return [
693             submob for submob in self.submobjects
694             if len(submob.submobjects) != 0 or len(submob.points) != 0
695         ]
696
697     def get_merged_array(self, array_attr):
698         result = getattr(self, array_attr)
699         for submob in self.submobjects:
700             result = np.append(
701                 result, submob.get_merged_array(array_attr),
702                 axis=0
703             )
704         return submob.get_merged_array(array_attr)
705         return result
706
707     def get_all_points(self):
708         return self.get_merged_array("points")
709
710     # Getters
711
712     def get_points_defining_boundary(self):
713         return self.get_all_points()
714
715     def get_num_points(self):
716         return len(self.points)
717
718     def get_extremum_along_dim(self, points=None, dim=0, key=0):
719         if points is None:
720             points = self.get_points_defining_boundary()
721         values = points[:, dim]
722         if key < 0:
723             return np.min(values)

```

```

724         elif key == 0:
725             return (np.min(values) + np.max(values)) / 2
726         else:
727             return np.max(values)
728
729     def get_critical_point(self, direction):
730         """
731         Picture a box bounding the mobject. Such a box has
732         9 'critical points': 4 corners, 4 edge center, the
733         center. This returns one of them.
734         """
735         result = np.zeros(self.dim)
736         all_points = self.get_points_defining_boundary()
737         if len(all_points) == 0:
738             return result
739         for dim in range(self.dim):
740             result[dim] = self.get_extremum_along_dim(
741                 all_points, dim=dim, key=direction[dim]
742             )
743         return result
744
745     # Pseudonyms for more general get_critical_point method
746
747     def get_edge_center(self, direction):
748         return self.get_critical_point(direction)
749
750     def get_corner(self, direction):
751         return self.get_critical_point(direction)
752
753     def get_center(self):
754         return self.get_critical_point(np.zeros(self.dim))
755
756     def get_center_of_mass(self):
757         return np.apply_along_axis(np.mean, 0, self.get_all_points())
758
759     def get_boundary_point(self, direction):
760         all_points = self.get_points_defining_boundary()
761         index = np.argmax(np.dot(all_points, np.array(direction).T))
762         return all_points[index]
763
764     def get_top(self):
765         return self.get_edge_center(UP)
766
767     def get_bottom(self):
768         return self.get_edge_center(DOWN)
769
770     def get_right(self):
771         return self.get_edge_center(RIGHT)
772
773     def get_left(self):
774         return self.get_edge_center(LEFT)
775
776     def get_zenith(self):
777         return self.get_edge_center(OUT)
778
779     def get_nadir(self):
780         return self.get_edge_center(IN)
781
782     def length_over_dim(self, dim):
783         return (
784             self.reduce_across_dimension(np.max, np.max, dim) -
785             self.reduce_across_dimension(np.min, np.min, dim)
786         )
787
788     def get_width(self):
789         return self.length_over_dim(0)
790
791     def get_height(self):
792         return self.length_over_dim(1)
793
794     def get_depth(self):
795         return self.length_over_dim(2)
796

```

```

797 | def get_coord(self, dim, direction=ORIGIN):
798 |     """
799 |     Meant to generalize get_x, get_y, get_z
800 |     """
801 |     return self.get_extremum_along_dim(
802 |         dim=dim, key=direction[dim]
803 |     )
804 |
805 | def get_x(self, direction=ORIGIN):
806 |     return self.get_coord(0, direction)
807 |
808 | def get_y(self, direction=ORIGIN):
809 |     return self.get_coord(1, direction)
810 |
811 | def get_z(self, direction=ORIGIN):
812 |     return self.get_coord(2, direction)
813 |
814 | def get_start(self):
815 |     self.throw_error_if_no_points()
816 |     return np.array(self.points[0])
817 |
818 | def get_end(self):
819 |     self.throw_error_if_no_points()
820 |     return np.array(self.points[-1])
821 |
822 | def get_start_and_end(self):
823 |     return self.get_start(), self.get_end()
824 |
825 | def point_from_proportion(self, alpha):
826 |     raise Exception("Not implemented")
827 |
828 | def get_pieces(self, n_pieces):
829 |     template = self.copy()
830 |     template.submobjects = []
831 |     alphas = np.linspace(0, 1, n_pieces + 1)
832 |     return Group(*[
833 |         template.copy().pointwise_become_partial(
834 |             self, a1, a2
835 |         )
836 |         for a1, a2 in zip(alphas[:-1], alphas[1:])
837 |     ])
838 |
839 | def get_z_index_reference_point(self):
840 |     # TODO, better place to define default z_index_group?
841 |     z_index_group = getattr(self, "z_index_group", self)
842 |     return z_index_group.get_center()
843 |
844 | def has_points(self):
845 |     return len(self.points) > 0
846 |
847 | def has_no_points(self):
848 |     return not self.has_points()
849 |
850 | # Match other mobject properties
851 |
852 | def match_color(self, mobject):
853 |     return self.set_color(mobject.get_color())
854 |
855 | def match_dim_size(self, mobject, dim, **kwargs):
856 |     return self.rescale_to_fit(
857 |         mobject.length_over_dim(dim), dim,
858 |         **kwargs
859 |     )
860 |
861 | def match_width(self, mobject, **kwargs):
862 |     return self.match_dim_size(mobject, 0, **kwargs)
863 |
864 | def match_height(self, mobject, **kwargs):
865 |     return self.match_dim_size(mobject, 1, **kwargs)
866 |
867 | def match_depth(self, mobject, **kwargs):
868 |     return self.match_dim_size(mobject, 2, **kwargs)
869 |

```

```

870     def match_coord(self, mobject, dim, direction=ORIGIN):
871         return self.set_coord(
872             mobject.get_coord(dim, direction),
873             dim=dim,
874             direction=direction,
875         )
876
877     def match_x(self, mobject, direction=ORIGIN):
878         return self.match_coord(mobject, 0, direction)
879
880     def match_y(self, mobject, direction=ORIGIN):
881         return self.match_coord(mobject, 1, direction)
882
883     def match_z(self, mobject, direction=ORIGIN):
884         return self.match_coord(mobject, 2, direction)
885
886     def align_to(self, mobject_or_point, direction=ORIGIN, alignment_vect=UP):
887         """
888         Examples:
889         mob1.align_to(mob2, UP) moves mob1 vertically so that its
890         top edge lines ups with mob2's top edge.
891
892         mob1.align_to(mob2, alignment_vect = RIGHT) moves mob1
893         horizontally so that it's center is directly above/below
894         the center of mob2
895         """
896         if isinstance(mobject_or_point, Mobject):
897             point = mobject_or_point.get_critical_point(direction)
898         else:
899             point = mobject_or_point
900
901         for dim in range(self.dim):
902             if direction[dim] != 0:
903                 self.set_coord(point[dim], dim, direction)
904         return self
905
906     # Family matters
907
908     def __getitem__(self, value):
909         self_list = self.split()
910         if isinstance(value, slice):
911             GroupClass = self.get_group_class()
912             return GroupClass(*self_list.__getitem__(value))
913         return self_list.__getitem__(value)
914
915     def __iter__(self):
916         return iter(self.split())
917
918     def __len__(self):
919         return len(self.split())
920
921     def get_group_class(self):
922         return Group
923
924     def split(self):
925         result = [self] if len(self.points) > 0 else []
926         return result + self.submobjects
927
928     def get_family(self):
929         sub_families = list(map(Mobject.get_family, self.submobjects))
930         all_mobjects = [self] + list(it.chain(*sub_families))
931         return remove_list_redundancies(all_mobjects)
932
933     def family_members_with_points(self):
934         return [m for m in self.get_family() if m.get_num_points() > 0]
935
936     def arrange(self, direction=RIGHT, center=True, **kwargs):
937         for m1, m2 in zip(self.submobjects, self.submobjects[1:]):
938             m2.next_to(m1, direction, **kwargs)
939         if center:
940             self.center()
941         return self
942

```

```

943     def arrange_in_grid(self, n_rows=None, n_cols=None, **kwargs):
944         submobs = self.submobjects
945         if n_rows is None and n_cols is None:
946             n_cols = int(np.sqrt(len(submobs)))
947
948         if n_rows is not None:
949             v1 = RIGHT
950             v2 = DOWN
951             n = len(submobs) // n_rows
952         elif n_cols is not None:
953             v1 = DOWN
954             v2 = RIGHT
955             n = len(submobs) // n_cols
956         Group(*[
957             Group(*submobs[i:i + n]).arrange(v1, **kwargs)
958             for i in range(0, len(submobs), n)
959         ]).arrange(v2, **kwargs)
960         return self
961
962     def sort(self, point_to_num_func=lambda p: p[0], submob_func=None):
963         if submob_func is None:
964             submob_func = lambda m: point_to_num_func(m.get_center())
965         self.submobjects.sort(key=submob_func)
966         return self
967
968     def shuffle(self, recursive=False):
969         if recursive:
970             for submob in self.submobjects:
971                 submob.shuffle(recursive=True)
972             random.shuffle(self.submobjects)
973
974     # Just here to keep from breaking old scenes.
975     def arrange_submobjects(self, *args, **kwargs):
976         return self.arrange(*args, **kwargs)
977
978     def sort_submobjects(self, *args, **kwargs):
979         return self.sort(*args, **kwargs)
980
981     def shuffle_submobjects(self, *args, **kwargs):
982         return self.shuffle(*args, **kwargs)
983
984     # Alignment
985     def align_data(self, mobject):
986         self.null_point_align(mobject)
987         self.align_submobjects(mobject)
988         self.align_points(mobject)
989         # Recurse
990         for m1, m2 in zip(self.submobjects, mobject.submobjects):
991             m1.align_data(m2)
992
993     def get_point_mobject(self, center=None):
994         """
995         The simplest mobject to be transformed to or from self.
996         Should be a point of the appropriate type
997         """
998         message = "get_point_mobject not implemented for {}"
999         raise Exception(message.format(self.__class__.__name__))
1000
1001     def align_points(self, mobject):
1002         count1 = self.get_num_points()
1003         count2 = mobject.get_num_points()
1004         if count1 < count2:
1005             self.align_points_with_larger(mobject)
1006         elif count2 < count1:
1007             mobject.align_points_with_larger(self)
1008         return self
1009
1010     def align_points_with_larger(self, larger_mobject):
1011         raise Exception("Not implemented")
1012
1013     def align_submobjects(self, mobject):
1014         mob1 = self
1015         mob2 = mobject

```

```

1016         n1 = len(mob1.submobjects)
1017         n2 = len(mob2.submobjects)
1018         mob1.add_n_more_submobjects(max(0, n2 - n1))
1019         mob2.add_n_more_submobjects(max(0, n1 - n2))
1020         return self
1021
1022     def null_point_align(self, mobject):
1023         """
1024         If a mobject with points is being aligned to
1025         one without, treat both as groups, and push
1026         the one with points into its own submobjects
1027         list.
1028         """
1029         for m1, m2 in (self, mobject), (mobject, self):
1030             if m1.has_no_points() and m2.has_points():
1031                 m2.push_self_into_submobjects()
1032         return self
1033
1034     def push_self_into_submobjects(self):
1035         copy = self.copy()
1036         copy.submobjects = []
1037         self.reset_points()
1038         self.add(copy)
1039         return self
1040
1041     def add_n_more_submobjects(self, n):
1042         if n == 0:
1043             return
1044
1045         curr = len(self.submobjects)
1046         if curr == 0:
1047             # If empty, simply add n point mobjects
1048             self.submobjects = [
1049                 self.get_point_mobject()
1050                 for k in range(n)
1051             ]
1052             return
1053
1054         target = curr + n
1055         # TODO, factor this out to utils so as to reuse
1056         # with VObject.insert_n_curves
1057         repeat_indices = (np.arange(target) * curr) // target
1058         split_factors = [
1059             sum(repeat_indices == i)
1060             for i in range(curr)
1061         ]
1062         new_submobs = []
1063         for submob, sf in zip(self.submobjects, split_factors):
1064             new_submobs.append(submob)
1065             for k in range(1, sf):
1066                 new_submobs.append(
1067                     submob.copy().fade(1)
1068                 )
1069         self.submobjects = new_submobs
1070         return self
1071
1072     def repeat_submobject(self, submob):
1073         return submob.copy()
1074
1075     def interpolate(self, mobject1, mobject2,
1076                     alpha, path_func=straight_path):
1077         """
1078         Turns self into an interpolation between mobject1
1079         and mobject2.
1080         """
1081         self.points = path_func(
1082             mobject1.points, mobject2.points, alpha
1083         )
1084         self.interpolate_color(mobject1, mobject2, alpha)
1085         return self
1086
1087     def interpolate_color(self, mobject1, mobject2, alpha):
1088         pass # To implement in subclass

```

```

1089
1090 | def become_partial(self, mobject, a, b):
1091 |     """
1092 |     Set points in such a way as to become only
1093 |     part of mobject.
1094 |     Inputs 0 <= a < b <= 1 determine what portion
1095 |     of mobject to become.
1096 |     """
1097 |     pass # To implement in subclasses
1098
1099 |     # TODO, color?
1100
1101 | def pointwise_become_partial(self, mobject, a, b):
1102 |     pass # To implement in subclass
1103
1104 | def become(self, mobject, copy_submobjects=True):
1105 |     """
1106 |     Edit points, colors and submobjects to be identical
1107 |     to another mobject
1108 |     """
1109 |     self.align_data(mobject)
1110 |     for sm1, sm2 in zip(self.get_family(), mobject.get_family()):
1111 |         sm1.points = np.array(sm2.points)
1112 |         sm1.interpolate_color(sm1, sm2, 1)
1113 |     return self
1114
1115 | # Errors
1116 | def throw_error_if_no_points(self):
1117 |     if self.has_no_points():
1118 |         message = "Cannot call Mobject.{0} " + \
1119 |             "for a Mobject with no points"
1120 |         caller_name = sys._getframe(1).f_code.co_name
1121 |         raise Exception(message.format(caller_name))
1122
1123
1124 | class Group(Mobject):
1125 |     def __init__(self, *mobjects, **kwargs):
1126 |         if not all([isinstance(m, Mobject) for m in mobjects]):
1127 |             raise Exception("All submobjects must be of type Mobject")
1128 |         Mobject.__init__(self, **kwargs)
1129 |         self.add(*mobjects)

```

« index coverage.py v5.0.3, created at 2020-05-17 11:47