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

706 statements | 365 run | 341 missing | 0 excluded

```python
from functools import reduce
import copy
import itertools as it
import operator as op
import os
import random
import sys

from colour import Color
import numpy as np

import manimlib.constants as consts
from manimlib.constants import *
from manimlib.container.container import Container
from manimlib.utils.color import color_gradient
from manimlib.utils.color import interpolate_color
from manimlib.utils.iterables import list_update
from manimlib.utils.iterables import remove_list_redundancies
from manimlib.utils.paths import straight_path
from manimlib.utils.simple_functions import get_parameters
from manimlib.utils.space_ops import angle_of_vector
from manimlib.utils.space_ops import get_norm
from manimlib.utils.space_ops import rotation_matrix


# TODO: Explain array_attrs

class Mobject(Container):
    """
    Mathematical Object
    """
    CONFIG = {
        "color": WHITE,
        "name": None,
        "dim": 3,
        "target": None,
    }

    def __init__(self, **kwargs):
        Container.__init__(self, **kwargs)
        self.submobjects = []
        self.color = Color(self.color)
        if self.name is None:
            self.name = self.__class__.__name__
        self.updaters = []
        self.updating_suspended = False
        self.reset_points()
        self.generate_points()
        self.init_colors()

    def __str__(self):
        return str(self.name)

    def reset_points(self):
        self.points = np.zeros((0, self.dim))

    def init_colors(self):
        # For subclasses
        pass

    def generate_points(self):
        # Typically implemented in subclass, unless purposefully left blank
        pass

    def add(self, *mobjects):
        if self in mobjects:
            raise Exception("Mobject cannot contain self")
        self.submobjects = list_update(self.submobjects, mobjects)
        return self

    def add_to_back(self, *mobjects):
        self.remove(*mobjects)
        self.submobjects = list(mobjects) + self.submobjects
        return self

    def remove(self, *mobjects):
        for mobject in mobjects:
            if mobject in self.submobjects:
                self.submobjects.remove(mobject)
        return self

    def get_array_attrs(self):
        return ["points"]

    def digest_mobject_attrs(self):
        """
        Ensures all attributes which are mobjects are included
        in the submobjects list.
        """
        mobject_attrs = [x for x in list(self.__dict__.values()) if isinstance(x, Mobject)]
        self.submobjects = list_update(self.submobjects, mobject_attrs)
        return self

    def apply_over_attr_arrays(self, func):
        for attr in self.get_array_attrs():
            setattr(self, attr, func(getattr(self, attr)))
        return self

    # Displaying

    def get_image(self, camera=None):
        if camera is None:
            from manimlib.camera.camera import Camera
            camera = Camera()
        camera.capture_mobject(self)
        return camera.get_image()

    def show(self, camera=None):
        self.get_image(camera=camera).show()

    def save_image(self, name=None):
        self.get_image().save(
            os.path.join(consts.VIDEO_DIR, (name or str(self)) + ".png")
        )

    def copy(self):
        # TODO, either justify reason for shallow copy, or
        # remove this redundancy everywhere
        # return self.deepcopy()

        copy_mobject = copy.copy(self)
        copy_mobject.points = np.array(self.points)
        copy_mobject.submobjects = [
            submob.copy() for submob in self.submobjects
        ]
        copy_mobject.updaters = list(self.updaters)
        family = self.get_family()
        for attr, value in list(self.__dict__.items()):
            if isinstance(value, Mobject) and value in family and value is not self:
                setattr(copy_mobject, attr, value.copy())
            if isinstance(value, np.ndarray):
                setattr(copy_mobject, attr, np.array(value))
        return copy_mobject

    def deepcopy(self):
        return copy.deepcopy(self)

    def generate_target(self, use_deepcopy=False):
        self.target = None  # Prevent exponential explosion
        if use_deepcopy:
            self.target = self.deepcopy()
        else:
            self.target = self.copy()
        return self.target

    # Updating

    def update(self, dt=0, recursive=True):
        if self.updating_suspended:
            return self
        for updater in self.updaters:
            parameters = get_parameters(updater)
            if "dt" in parameters:
                updater(self, dt)
            else:
                updater(self)
        if recursive:
            for submob in self.submobjects:
                submob.update(dt, recursive)
        return self

    def get_time_based_updaters(self):
        return [
            updater for updater in self.updaters
            if "dt" in get_parameters(updater)
        ]

    def has_time_based_updater(self):
        for updater in self.updaters:
            if "dt" in get_parameters(updater):
                return True
        return False

    def get_updaters(self):
        return self.updaters

    def get_family_updaters(self):
        return list(it.chain(*[
            sm.get_updaters()
            for sm in self.get_family()
        ]))

    def add_updater(self, update_function, index=None, call_updater=True):
        if index is None:
            self.updaters.append(update_function)
        else:
            self.updaters.insert(index, update_function)
        if call_updater:
            self.update(0)
        return self

    def remove_updater(self, update_function):
        while update_function in self.updaters:
            self.updaters.remove(update_function)
        return self

    def clear_updaters(self, recursive=True):
        self.updaters = []
        if recursive:
            for submob in self.submobjects:
                submob.clear_updaters()
        return self

    def match_updaters(self, mobject):
        self.clear_updaters()
        for updater in mobject.get_updaters():
            self.add_updater(updater)
        return self

    def suspend_updating(self, recursive=True):
        self.updating_suspended = True
        if recursive:
            for submob in self.submobjects:
                submob.suspend_updating(recursive)
        return self

    def resume_updating(self, recursive=True):
        self.updating_suspended = False
        if recursive:
            for submob in self.submobjects:
                submob.resume_updating(recursive)
        self.update(dt=0, recursive=recursive)
        return self

    # Transforming operations

    def apply_to_family(self, func):
        for mob in self.family_members_with_points():
            func(mob)

    def shift(self, *vectors):
        total_vector = reduce(op.add, vectors)
        for mob in self.family_members_with_points():
            mob.points = mob.points.astype('float')
            mob.points += total_vector
        return self

    def scale(self, scale_factor, **kwargs):
        """
        Default behavior is to scale about the center of the mobject.
        The argument about_edge can be a vector, indicating which side of
        the mobject to scale about, e.g., mob.scale(about_edge = RIGHT)
        scales about mob.get_right().

        Otherwise, if about_point is given a value, scaling is done with
        respect to that point.
        """
        self.apply_points_function_about_point(
            lambda points: scale_factor * points, **kwargs
        )
        return self

    def rotate_about_origin(self, angle, axis=OUT, axes=[]):
        return self.rotate(angle, axis, about_point=ORIGIN)

    def rotate(self, angle, axis=OUT, **kwargs):
        rot_matrix = rotation_matrix(angle, axis)
        self.apply_points_function_about_point(
            lambda points: np.dot(points, rot_matrix.T),
            **kwargs
        )
        return self

    def flip(self, axis=UP, **kwargs):
        return self.rotate(TAU / 2, axis, **kwargs)

    def stretch(self, factor, dim, **kwargs):
        def func(points):
            points[:, dim] *= factor
            return points
        self.apply_points_function_about_point(func, **kwargs)
        return self

    def apply_function(self, function, **kwargs):
        # Default to applying matrix about the origin, not mobjects center
        if len(kwargs) == 0:
            kwargs["about_point"] = ORIGIN
        self.apply_points_function_about_point(
            lambda points: np.apply_along_axis(function, 1, points),
            **kwargs
        )
        return self

    def apply_function_to_position(self, function):
        self.move_to(function(self.get_center()))
```

```python
        return self

    def apply_function_to_submobject_positions(self, function):
        for submob in self.submobjects:
            submob.apply_function_to_position(function)
        return self

    def apply_matrix(self, matrix, **kwargs):
        # Default to applying matrix about the origin, not mobjects center
        if ("about_point" not in kwargs) and ("about_edge" not in kwargs):
            kwargs["about_point"] = ORIGIN
        full_matrix = np.identity(self.dim)
        matrix = np.array(matrix)
        full_matrix[:matrix.shape[0], :matrix.shape[1]] = matrix
        self.apply_points_function_about_point(
            lambda points: np.dot(points, full_matrix.T),
            **kwargs
        )
        return self

    def apply_complex_function(self, function, **kwargs):
        def R3_func(point):
            x, y, z = point
            xy_complex = function(complex(x, y))
            return [
                xy_complex.real,
                xy_complex.imag,
                z
            ]
        return self.apply_function(R3_func)

    def wag(self, direction=RIGHT, axis=DOWN, wag_factor=1.0):
        for mob in self.family_members_with_points():
            alphas = np.dot(mob.points, np.transpose(axis))
            alphas -= min(alphas)
            alphas /= max(alphas)
            alphas = alphas**wag_factor
            mob.points += np.dot(
                alphas.reshape((len(alphas), 1)),
                np.array(direction).reshape((1, mob.dim))
            )
        return self

    def reverse_points(self):
        for mob in self.family_members_with_points():
            mob.apply_over_attr_arrays(
                lambda arr: np.array(list(reversed(arr)))
            )
        return self

    def repeat(self, count):
        """
        This can make transition animations nicer
        """
        def repeat_array(array):
            return reduce(
                lambda a1, a2: np.append(a1, a2, axis=0),
                [array] * count
            )
        for mob in self.family_members_with_points():
            mob.apply_over_attr_arrays(repeat_array)
        return self

    # In place operations.
    # Note, much of these are now redundant with default behavior of
    # above methods

    def apply_points_function_about_point(self, func, about_point=None, about_edge=None):
        if about_point is None:
            if about_edge is None:
                about_edge = ORIGIN
            about_point = self.get_critical_point(about_edge)
        for mob in self.family_members_with_points():
            mob.points -= about_point
            mob.points = func(mob.points)
            mob.points += about_point
        return self

    def rotate_in_place(self, angle, axis=OUT):
        # redundant with default behavior of rotate now.
        return self.rotate(angle, axis=axis)

    def scale_in_place(self, scale_factor, **kwargs):
        # Redundant with default behavior of scale now.
        return self.scale(scale_factor, **kwargs)

    def scale_about_point(self, scale_factor, point):
        # Redundant with default behavior of scale now.
        return self.scale(scale_factor, about_point=point)

    def pose_at_angle(self, **kwargs):
        self.rotate(TAU / 14, RIGHT + UP, **kwargs)
        return self

    # Positioning methods

    def center(self):
        self.shift(-self.get_center())
        return self

    def align_on_border(self, direction, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
        """
        Direction just needs to be a vector pointing towards side or
        corner in the 2d plane.
        """
        target_point = np.sign(direction) * (FRAME_X_RADIUS, FRAME_Y_RADIUS, 0)
        point_to_align = self.get_critical_point(direction)
        shift_val = target_point - point_to_align - buff * np.array(direction)
        shift_val = shift_val * abs(np.sign(direction))
        self.shift(shift_val)
        return self

    def to_corner(self, corner=LEFT + DOWN, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
        return self.align_on_border(corner, buff)

    def to_edge(self, edge=LEFT, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
        return self.align_on_border(edge, buff)

    def next_to(self, mobject_or_point,
                direction=RIGHT,
                buff=DEFAULT_MOBJECT_TO_MOBJECT_BUFFER,
                aligned_edge=ORIGIN,
                submobject_to_align=None,
                index_of_submobject_to_align=None,
                coor_mask=np.array([1, 1, 1]),
                ):
        if isinstance(mobject_or_point, Mobject):
            mob = mobject_or_point
            if index_of_submobject_to_align is not None:
                target_aligner = mob[index_of_submobject_to_align]
            else:
                target_aligner = mob
            target_point = target_aligner.get_critical_point(
                aligned_edge + direction
            )
        else:
            target_point = mobject_or_point
        if submobject_to_align is not None:
            aligner = submobject_to_align
        elif index_of_submobject_to_align is not None:
            aligner = self[index_of_submobject_to_align]
        else:
            aligner = self
        point_to_align = aligner.get_critical_point(aligned_edge - direction)
        self.shift((target_point - point_to_align +
                    buff * direction) * coor_mask)
        return self

    def shift_onto_screen(self, **kwargs):
        space_lengths = [FRAME_X_RADIUS, FRAME_Y_RADIUS]
        for vect in UP, DOWN, LEFT, RIGHT:
            dim = np.argmax(np.abs(vect))
            buff = kwargs.get("buff", DEFAULT_MOBJECT_TO_EDGE_BUFFER)
            max_val = space_lengths[dim] - buff
            edge_center = self.get_edge_center(vect)
            if np.dot(edge_center, vect) > max_val:
                self.to_edge(vect, **kwargs)
        return self

    def is_off_screen(self):
        if self.get_left()[0] > FRAME_X_RADIUS:
            return True
        if self.get_right()[0] < -FRAME_X_RADIUS:
            return True
        if self.get_bottom()[1] > FRAME_Y_RADIUS:
            return True
        if self.get_top()[1] < -FRAME_Y_RADIUS:
            return True
        return False

    def stretch_about_point(self, factor, dim, point):
        return self.stretch(factor, dim, about_point=point)

    def stretch_in_place(self, factor, dim):
        # Now redundant with stretch
        return self.stretch(factor, dim)

    def rescale_to_fit(self, length, dim, stretch=False, **kwargs):
        old_length = self.length_over_dim(dim)
        if old_length == 0:
            return self
        if stretch:
            self.stretch(length / old_length, dim, **kwargs)
        else:
            self.scale(length / old_length, **kwargs)
        return self

    def stretch_to_fit_width(self, width, **kwargs):
        return self.rescale_to_fit(width, 0, stretch=True, **kwargs)

    def stretch_to_fit_height(self, height, **kwargs):
        return self.rescale_to_fit(height, 1, stretch=True, **kwargs)

    def stretch_to_fit_depth(self, depth, **kwargs):
        return self.rescale_to_fit(depth, 1, stretch=True, **kwargs)

    def set_width(self, width, stretch=False, **kwargs):
        return self.rescale_to_fit(width, 0, stretch=stretch, **kwargs)

    def set_height(self, height, stretch=False, **kwargs):
        return self.rescale_to_fit(height, 1, stretch=stretch, **kwargs)

    def set_depth(self, depth, stretch=False, **kwargs):
        return self.rescale_to_fit(depth, 2, stretch=stretch, **kwargs)

    def set_coord(self, value, dim, direction=ORIGIN):
        curr = self.get_coord(dim, direction)
        shift_vect = np.zeros(self.dim)
        shift_vect[dim] = value - curr
        self.shift(shift_vect)
        return self

    def set_x(self, x, direction=ORIGIN):
        return self.set_coord(x, 0, direction)

    def set_y(self, y, direction=ORIGIN):
        return self.set_coord(y, 1, direction)

    def set_z(self, z, direction=ORIGIN):
```

```python
        return self.set_coord(z, 2, direction)

    def space_out_submobjects(self, factor=1.5, **kwargs):
        self.scale(factor, **kwargs)
        for submob in self.submobjects:
            submob.scale(1. / factor)
        return self

    def move_to(self, point_or_mobject, aligned_edge=ORIGIN,
                coor_mask=np.array([1, 1, 1])):
        if isinstance(point_or_mobject, Mobject):
            target = point_or_mobject.get_critical_point(aligned_edge)
        else:
            target = point_or_mobject
        point_to_align = self.get_critical_point(aligned_edge)
        self.shift((target - point_to_align) * coor_mask)
        return self

    def replace(self, mobject, dim_to_match=0, stretch=False):
        if not mobject.get_num_points() and not mobject.submobjects:
            raise Warning("Attempting to replace mobject with no points")
            return self
        if stretch:
            self.stretch_to_fit_width(mobject.get_width())
            self.stretch_to_fit_height(mobject.get_height())
        else:
            self.rescale_to_fit(
                mobject.length_over_dim(dim_to_match),
                dim_to_match,
                stretch=False
            )
        self.shift(mobject.get_center() - self.get_center())
        return self

    def surround(self, mobject,
                 dim_to_match=0,
                 stretch=False,
                 buff=MED_SMALL_BUFF):
        self.replace(mobject, dim_to_match, stretch)
        length = mobject.length_over_dim(dim_to_match)
        self.scale_in_place((length + buff) / length)
        return self

    def put_start_and_end_on(self, start, end):
        curr_start, curr_end = self.get_start_and_end()
        curr_vect = curr_end - curr_start
        if np.all(curr_vect == 0):
            raise Exception("Cannot position endpoints of closed loop")
        target_vect = end - start
        self.scale(
            get_norm(target_vect) / get_norm(curr_vect),
            about_point=curr_start,
        )
        self.rotate(
            angle_of_vector(target_vect) -
            angle_of_vector(curr_vect),
            about_point=curr_start
        )
        self.shift(start - curr_start)
        return self

    # Background rectangle
    def add_background_rectangle(self, color=BLACK, opacity=0.75, **kwargs):
        # TODO, this does not behave well when the mobject has points,
        # since it gets displayed on top
        from manimlib.mobject.shape_matchers import BackgroundRectangle
        self.background_rectangle = BackgroundRectangle(
            self, color=color,
            fill_opacity=opacity,
            **kwargs
        )
        self.add_to_back(self.background_rectangle)
        return self

    def add_background_rectangle_to_submobjects(self, **kwargs):
        for submobject in self.submobjects:
            submobject.add_background_rectangle(**kwargs)
        return self

    def add_background_rectangle_to_family_members_with_points(self, **kwargs):
        for mob in self.family_members_with_points():
            mob.add_background_rectangle(**kwargs)
        return self

    # Color functions

    def set_color(self, color=YELLOW_C, family=True):
        """
        Condition is function which takes in one arguments, (x, y, z).
        Here it just recurses to submobjects, but in subclasses this
        should be further implemented based on the the inner workings
        of color
        """
        if family:
            for submob in self.submobjects:
                submob.set_color(color, family=family)
        self.color = color
        return self

    def set_color_by_gradient(self, *colors):
        self.set_submobject_colors_by_gradient(*colors)
        return self

    def set_colors_by_radial_gradient(self, center=None, radius=1, inner_color=WHITE, outer_color=BLACK):
        self.set_submobject_colors_by_radial_gradient(
            center, radius, inner_color, outer_color)
        return self

    def set_submobject_colors_by_gradient(self, *colors):
        if len(colors) == 0:
            raise Exception("Need at least one color")
        elif len(colors) == 1:
            return self.set_color(*colors)

        mobs = self.family_members_with_points()
        new_colors = color_gradient(colors, len(mobs))

        for mob, color in zip(mobs, new_colors):
            mob.set_color(color, family=False)
        return self

    def set_submobject_colors_by_radial_gradient(self, center=None, radius=1, inner_color=WHITE, outer_color=BLACK)
        if center is None:
            center = self.get_center()

        for mob in self.family_members_with_points():
            t = get_norm(mob.get_center() - center) / radius
            t = min(t, 1)
            mob_color = interpolate_color(inner_color, outer_color, t)
            mob.set_color(mob_color, family=False)

        return self

    def to_original_color(self):
        self.set_color(self.color)
        return self

    def fade_to(self, color, alpha, family=True):
        if self.get_num_points() > 0:
            new_color = interpolate_color(
                self.get_color(), color, alpha
            )
            self.set_color(new_color, family=False)
        if family:
            for submob in self.submobjects:
                submob.fade_to(color, alpha)
        return self

    def fade(self, darkness=0.5, family=True):
        if family:
            for submob in self.submobjects:
                submob.fade(darkness, family)
        return self

    def get_color(self):
        return self.color

    ##

    def save_state(self, use_deepcopy=False):
        if hasattr(self, "saved_state"):
            # Prevent exponential growth of data
            self.saved_state = None
        if use_deepcopy:
            self.saved_state = self.deepcopy()
        else:
            self.saved_state = self.copy()
        return self

    def restore(self):
        if not hasattr(self, "saved_state") or self.save_state is None:
            raise Exception("Trying to restore without having saved")
        self.become(self.saved_state)
        return self

    ##

    def reduce_across_dimension(self, points_func, reduce_func, dim):
        points = self.get_all_points()
        if points is None or len(points) == 0:
            # Note, this default means things like empty VGroups
            # will appear to have a center at [0, 0, 0]
            return 0
        values = points_func(points[:, dim])
        return reduce_func(values)

    def nonempty_submobjects(self):
        return [
            submob for submob in self.submobjects
            if len(submob.submobjects) != 0 or len(submob.points) != 0
        ]

    def get_merged_array(self, array_attr):
        result = getattr(self, array_attr)
        for submob in self.submobjects:
            result = np.append(
                result, submob.get_merged_array(array_attr),
                axis=0
            )
            submob.get_merged_array(array_attr)
        return result

    def get_all_points(self):
        return self.get_merged_array("points")

    # Getters

    def get_points_defining_boundary(self):
        return self.get_all_points()

    def get_num_points(self):
        return len(self.points)

    def get_extremum_along_dim(self, points=None, dim=0, key=0):
        if points is None:
            points = self.get_points_defining_boundary()
        values = points[:, dim]
        if key < 0:
            return np.min(values)
        elif key == 0:
            return (np.min(values) + np.max(values)) / 2
        else:
            return np.max(values)

    def get_critical_point(self, direction):
        """
        Picture a box bounding the mobject.  Such a box has
        9 'critical points': 4 corners, 4 edge center, the
        center.  This returns one of them.
        """
        result = np.zeros(self.dim)
        all_points = self.get_points_defining_boundary()
        if len(all_points) == 0:
            return result
        for dim in range(self.dim):
            result[dim] = self.get_extremum_along_dim(
                all_points, dim=dim, key=direction[dim]
            )
        return result

    # Pseudonyms for more general get_critical_point method

    def get_edge_center(self, direction):
        return self.get_critical_point(direction)

    def get_corner(self, direction):
        return self.get_critical_point(direction)

    def get_center(self):
        return self.get_critical_point(np.zeros(self.dim))

    def get_center_of_mass(self):
        return np.apply_along_axis(np.mean, 0, self.get_all_points())

    def get_boundary_point(self, direction):
        all_points = self.get_points_defining_boundary()
        index = np.argmax(np.dot(all_points, np.array(direction).T))
        return all_points[index]

    def get_top(self):
        return self.get_edge_center(UP)

    def get_bottom(self):
        return self.get_edge_center(DOWN)

    def get_right(self):
        return self.get_edge_center(RIGHT)

    def get_left(self):
        return self.get_edge_center(LEFT)

    def get_zenith(self):
        return self.get_edge_center(OUT)

    def get_nadir(self):
        return self.get_edge_center(IN)

    def length_over_dim(self, dim):
        return (
            self.reduce_across_dimension(np.max, np.max, dim) -
            self.reduce_across_dimension(np.min, np.min, dim)
        )

    def get_width(self):
        return self.length_over_dim(0)

    def get_height(self):
        return self.length_over_dim(1)

    def get_depth(self):
        return self.length_over_dim(2)

```

```python
    def get_coord(self, dim, direction=ORIGIN):
        """
        Meant to generalize get_x, get_y, get_z
        """
        return self.get_extremum_along_dim(
            dim=dim, key=direction[dim]
        )

    def get_x(self, direction=ORIGIN):
        return self.get_coord(0, direction)

    def get_y(self, direction=ORIGIN):
        return self.get_coord(1, direction)

    def get_z(self, direction=ORIGIN):
        return self.get_coord(2, direction)

    def get_start(self):
        self.throw_error_if_no_points()
        return np.array(self.points[0])

    def get_end(self):
        self.throw_error_if_no_points()
        return np.array(self.points[-1])

    def get_start_and_end(self):
        return self.get_start(), self.get_end()

    def point_from_proportion(self, alpha):
        raise Exception("Not implemented")

    def get_pieces(self, n_pieces):
        template = self.copy()
        template.submobjects = []
        alphas = np.linspace(0, 1, n_pieces + 1)
        return Group(*[
            template.copy().pointwise_become_partial(
                self, a1, a2
            )
            for a1, a2 in zip(alphas[:-1], alphas[1:])
        ])

    def get_z_index_reference_point(self):
        # TODO, better place to define default z_index_group?
        z_index_group = getattr(self, "z_index_group", self)
        return z_index_group.get_center()

    def has_points(self):
        return len(self.points) > 0

    def has_no_points(self):
        return not self.has_points()

    # Match other mobject properties

    def match_color(self, mobject):
        return self.set_color(mobject.get_color())

    def match_dim_size(self, mobject, dim, **kwargs):
        return self.rescale_to_fit(
            mobject.length_over_dim(dim), dim,
            **kwargs
        )

    def match_width(self, mobject, **kwargs):
        return self.match_dim_size(mobject, 0, **kwargs)

    def match_height(self, mobject, **kwargs):
        return self.match_dim_size(mobject, 1, **kwargs)

    def match_depth(self, mobject, **kwargs):
        return self.match_dim_size(mobject, 2, **kwargs)

    def match_coord(self, mobject, dim, direction=ORIGIN):
        return self.set_coord(
            mobject.get_coord(dim, direction),
            dim=dim,
            direction=direction,
        )

    def match_x(self, mobject, direction=ORIGIN):
        return self.match_coord(mobject, 0, direction)

    def match_y(self, mobject, direction=ORIGIN):
        return self.match_coord(mobject, 1, direction)

    def match_z(self, mobject, direction=ORIGIN):
        return self.match_coord(mobject, 2, direction)

    def align_to(self, mobject_or_point, direction=ORIGIN, alignment_vect=UP):
        """
        Examples:
        mob1.align_to(mob2, UP) moves mob1 vertically so that its
        top edge lines ups with mob2's top edge.

        mob1.align_to(mob2, alignment_vect = RIGHT) moves mob1
        horizontally so that it's center is directly above/below
        the center of mob2
        """
        if isinstance(mobject_or_point, Mobject):
            point = mobject_or_point.get_critical_point(direction)
        else:
            point = mobject_or_point

        for dim in range(self.dim):
            if direction[dim] != 0:
                self.set_coord(point[dim], dim, direction)
        return self

    # Family matters

    def __getitem__(self, value):
        self_list = self.split()
        if isinstance(value, slice):
            GroupClass = self.get_group_class()
            return GroupClass(*self_list.__getitem__(value))
        return self_list.__getitem__(value)

    def __iter__(self):
        return iter(self.split())

    def __len__(self):
        return len(self.split())

    def get_group_class(self):
        return Group

    def split(self):
        result = [self] if len(self.points) > 0 else []
        return result + self.submobjects

    def get_family(self):
        sub_families = list(map(Mobject.get_family, self.submobjects))
        all_mobjects = [self] + list(it.chain(*sub_families))
        return remove_list_redundancies(all_mobjects)

    def family_members_with_points(self):
        return [m for m in self.get_family() if m.get_num_points() > 0]

    def arrange(self, direction=RIGHT, center=True, **kwargs):
        for m1, m2 in zip(self.submobjects, self.submobjects[1:]):
            m2.next_to(m1, direction, **kwargs)
        if center:
            self.center()
        return self

    def arrange_in_grid(self, n_rows=None, n_cols=None, **kwargs):
        submobs = self.submobjects
        if n_rows is None and n_cols is None:
            n_cols = int(np.sqrt(len(submobs)))

        if n_rows is not None:
            v1 = RIGHT
            v2 = DOWN
            n = len(submobs) // n_rows
        elif n_cols is not None:
            v1 = DOWN
            v2 = RIGHT
            n = len(submobs) // n_cols
        Group(*[
            Group(*submobs[i:i + n]).arrange(v1, **kwargs)
            for i in range(0, len(submobs), n)
        ]).arrange(v2, **kwargs)
        return self

    def sort(self, point_to_num_func=lambda p: p[0], submob_func=None):
        if submob_func is None:
            submob_func = lambda m: point_to_num_func(m.get_center())
        self.submobjects.sort(key=submob_func)
        return self

    def shuffle(self, recursive=False):
        if recursive:
            for submob in self.submobjects:
                submob.shuffle(recursive=True)
        random.shuffle(self.submobjects)

    # Just here to keep from breaking old scenes.
    def arrange_submobjects(self, *args, **kwargs):
        return self.arrange(*args, **kwargs)

    def sort_submobjects(self, *args, **kwargs):
        return self.sort(*args, **kwargs)

    def shuffle_submobjects(self, *args, **kwargs):
        return self.shuffle(*args, **kwargs)

    # Alignment
    def align_data(self, mobject):
        self.null_point_align(mobject)
        self.align_submobjects(mobject)
        self.align_points(mobject)
        # Recurse
        for m1, m2 in zip(self.submobjects, mobject.submobjects):
            m1.align_data(m2)

    def get_point_mobject(self, center=None):
        """
        The simplest mobject to be transformed to or from self.
        Should by a point of the appropriate type
        """
        message = "get_point_mobject not implemented for {}"
        raise Exception(message.format(self.__class__.__name__))

    def align_points(self, mobject):
        count1 = self.get_num_points()
        count2 = mobject.get_num_points()
        if count1 < count2:
            self.align_points_with_larger(mobject)
        elif count2 < count1:
            mobject.align_points_with_larger(self)
        return self

    def align_points_with_larger(self, larger_mobject):
        raise Exception("Not implemented")

    def align_submobjects(self, mobject):
        mob1 = self
        mob2 = mobject
        n1 = len(mob1.submobjects)
        n2 = len(mob2.submobjects)
        mob1.add_n_more_submobjects(max(0, n2 - n1))
        mob2.add_n_more_submobjects(max(0, n1 - n2))
        return self

    def null_point_align(self, mobject):
        """
        If a mobject with points is being aligned to
        one without, treat both as groups, and push
        the one with points into its own submobjects
        list.
        """
        for m1, m2 in (self, mobject), (mobject, self):
            if m1.has_no_points() and m2.has_points():
                m2.push_self_into_submobjects()
        return self

    def push_self_into_submobjects(self):
        copy = self.copy()
        copy.submobjects = []
        self.reset_points()
        self.add(copy)
        return self

    def add_n_more_submobjects(self, n):
        if n == 0:
            return

        curr = len(self.submobjects)
        if curr == 0:
            # If empty, simply add n point mobjects
            self.submobjects = [
                self.get_point_mobject()
                for k in range(n)
            ]
            return

        target = curr + n
        # TODO, factor this out to utils so as to reuse
        # with VMobject.insert_n_curves
        repeat_indices = (np.arange(target) * curr) // target
        split_factors = [
            sum(repeat_indices == i)
            for i in range(curr)
        ]
        new_submobs = []
        for submob, sf in zip(self.submobjects, split_factors):
            new_submobs.append(submob)
            for k in range(1, sf):
                new_submobs.append(
                    submob.copy().fade(1)
                )
        self.submobjects = new_submobs
        return self

    def repeat_submobject(self, submob):
        return submob.copy()

    def interpolate(self, mobject1, mobject2,
                    alpha, path_func=straight_path):
        """
        Turns self into an interpolation between mobject1
        and mobject2.
        """
        self.points = path_func(
            mobject1.points, mobject2.points, alpha
        )
        self.interpolate_color(mobject1, mobject2, alpha)
        return self

    def interpolate_color(self, mobject1, mobject2, alpha):
        pass  # To implement in subclass

    def become_partial(self, mobject, a, b):
        """
        Set points in such a way as to become only
        part of mobject.
        Inputs 0 <= a < b <= 1 determine what portion
        of mobject to become.
        """
        pass  # To implement in subclasses

        # TODO, color?

    def pointwise_become_partial(self, mobject, a, b):
        pass  # To implement in subclass

    def become(self, mobject, copy_submobjects=True):
        """
        Edit points, colors and submobjects to be idential
        to another mobject
        """
        self.align_data(mobject)
        for sm1, sm2 in zip(self.get_family(), mobject.get_family()):
            sm1.points = np.array(sm2.points)
            sm1.interpolate_color(sm1, sm2, 1)
        return self

    # Errors
    def throw_error_if_no_points(self):
        if self.has_no_points():
            message = "Cannot call Mobject.{} " +\
                      "for a Mobject with no points"
            caller_name = sys._getframe(1).f_code.co_name
            raise Exception(message.format(caller_name))


class Group(Mobject):
    def __init__(self, *mobjects, **kwargs):
        if not all([isinstance(m, Mobject) for m in mobjects]):
            raise Exception("All submobjects must be of type Mobject")
        Mobject.__init__(self, **kwargs)
        self.add(*mobjects)
```

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