ground

Release 9.0.0

Azat Ibrakov

May 12, 2023

CONTENTS

Note: If object is not listed in documentation it should be considered as implementation detail that can change and should not be relied upon.

CHAPTER

ONE

BASE MODULE

class ground.base.**Location**(*value*)

Represents kinds of locations in which point can be relative to geometry.

EXTERIOR = 0

point lies in the exterior of the geometry

BOUNDARY = 1

point lies on the boundary of the geometry

INTERIOR = 2

point lies in the interior of the geometry

class ground.base.**Kind**(*value*)

Represents kinds of angles based on their degrees value in range `[0, 180]`.

OBTUSE = -1

`(90, 180]` degrees

RIGHT = 0

`90` degrees

ACUTE = 1

`[0, 90)` degrees

class ground.base.**Orientation**(*value*)

Represents kinds of angle orientations.

CLOCKWISE = -1

in the same direction as a clock's hands

COLLINEAR = 0

to the top and then to the bottom or vice versa

COUNTERCLOCKWISE = 1

opposite to clockwise

class ground.base.**Relation**(*value*)

Represents kinds of relations in which geometries can be. Order of members assumes that conditions for previous ones do not hold.

DISJOINT = 0

intersection is empty

TOUCH = 1

intersection is a strict subset of each of the geometries, has dimension less than at least of one of the geometries and if we traverse boundary of each of the geometries in any direction then boundary of the other geometry won't be on one of sides at each point of boundaries intersection

CROSS = 2

intersection is a strict subset of each of the geometries, has dimension less than at least of one of the geometries and if we traverse boundary of each of the geometries in any direction then boundary of the other geometry will be on both sides at some point of boundaries intersection

OVERLAP = 3

intersection is a strict subset of each of the geometries and has the same dimension as geometries

COVER = 4

interior of the geometry is a superset of the other

ENCLOSES = 5

boundary of the geometry contains at least one boundary point of the other, but not all, interior of the geometry contains other points of the other

COMPOSITE = 6

geometry is a strict superset of the other and interior/boundary of the geometry is a superset of interior/boundary of the other

EQUAL = 7

geometries are equal

COMPONENT = 8

geometry is a strict subset of the other and interior/boundary of the geometry is a subset of interior/boundary of the other

ENCLOSED = 9

at least one boundary point of the geometry lies on the boundary of the other, but not all, other points of the geometry lie in the interior of the other

WITHIN = 10

geometry is a subset of the interior of the other

class ground.base.**Mode**(*value*)

Represents possible context modes.

class ground.base.**Context**(**, box_cls: ~typing.Type[~ground.hints.Box] = <class 'ground.core.geometries.Box'>, contour_cls: ~typing.Type[~ground.hints.Contour] = <class 'ground.core.geometries.Contour'>, empty_cls: ~typing.Type[~ground.hints.Empty] = <class 'ground.core.geometries.Empty'>, mix_cls: ~typing.Type[~ground.hints.Mix] = <class 'ground.core.geometries.Mix'>, multipoint_cls: ~typing.Type[~ground.hints.Multipoint] = <class 'ground.core.geometries.Multipoint'>, multipolygon_cls: ~typing.Type[~ground.hints.Multipolygon] = <class 'ground.core.geometries.Multipolygon'>, multisegment_cls: ~typing.Type[~ground.hints.Multisegment] = <class 'ground.core.geometries.Multisegment'>, point_cls: ~typing.Type[~ground.hints.Point] = <class 'ground.core.geometries.Point'>, polygon_cls: ~typing.Type[~ground.hints.Polygon] = <class 'ground.core.geometries.Polygon'>, segment_cls: ~typing.Type[~ground.hints.Segment] = <class 'ground.core.geometries.Segment'>, mode: ~ground.base.Mode = Mode.EXACT, sqrt: ~typing.Callable[[~ground.core.hints.Scalar], ~ground.core.hints.Scalar] = <function sqrt>*)

Represents common language for computational geometry.

property angle_kind: Callable[[*Point*, *Point*, *Point*], *Kind*]

Returns function for computing angle kind.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> (context.angle_kind(Point(0, 0), Point(1, 0), Point(-1, 0))
...  is Kind.OBTUSE)
True
>>> (context.angle_kind(Point(0, 0), Point(1, 0), Point(0, 1))
...  is Kind.RIGHT)
True
>>> (context.angle_kind(Point(0, 0), Point(1, 0), Point(1, 0))
...  is Kind.ACUTE)
True
```

property angle_orientation: Callable[[*Point*, *Point*, *Point*], *Orientation*]

Returns function for computing angle orientation.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> (context.angle_orientation(Point(0, 0), Point(0, 1), Point(1, 0))
...  is Orientation.CLOCKWISE)
```

(continues on next page)

(continued from previous page)

```
True
>>> (context.angle_orientation(Point(0, 0), Point(1, 0), Point(1, 0))
...  is Orientation.COLLINEAR)
True
>>> (context.angle_orientation(Point(0, 0), Point(1, 0), Point(0, 1))
...  is Orientation.COUNTERCLOCKWISE)
True
```

property box_cls: Type[*Box*]

Returns type for boxes.

property box_point_squared_distance: Callable[[*Box*, *Point*], Scalar]

Returns squared Euclidean distance between box and a point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Box, Point = context.box_cls, context.point_cls
>>> context.box_point_squared_distance(Box(0, 1, 0, 1),
...                                    Point(1, 1)) == 0
True
>>> context.box_point_squared_distance(Box(0, 1, 0, 1),
...                                    Point(2, 1)) == 1
True
>>> context.box_point_squared_distance(Box(0, 1, 0, 1),
...                                    Point(2, 2)) == 2
True
```

property contour_cls: Type[*Contour*]

Returns type for contours.

property cross_product: Callable[[*Point*, *Point*, *Point*, *Point*], Scalar]

Returns cross product of the segments.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> context.cross_product(Point(0, 0), Point(0, 1), Point(0, 0),
...                       Point(1, 0)) == -1
True
>>> context.cross_product(Point(0, 0), Point(1, 0), Point(0, 0),
...                       Point(1, 0)) == 0
True
>>> context.cross_product(Point(0, 0), Point(1, 0), Point(0, 0),
...                       Point(0, 1)) == 1
True
```

property dot_product: Callable[[*Point*, *Point*, *Point*, *Point*], Scalar]

Returns dot product of the segments.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> context.dot_product(Point(0, 0), Point(1, 0), Point(0, 0),
...                     Point(-1, 0)) == -1
True
>>> context.dot_product(Point(0, 0), Point(1, 0), Point(0, 0),
...                     Point(0, 1)) == 0
True
>>> context.dot_product(Point(0, 0), Point(1, 0), Point(0, 0),
...                     Point(1, 0)) == 1
True
```

property empty: *Empty*

Returns an empty geometry.

property empty_cls: Type[*Empty*]

Returns type for empty geometries.

property mix_cls: Type[*Mix*]

Returns type for mixes.

property mode: *Mode*

Returns mode of the context.

property multipoint_cls: Type[*Multipoint*]

Returns type for multipoints.

property multipolygon_cls: Type[*Multipolygon*]

Returns type for multipolygons.

property multisegment_cls: Type[*Multisegment*]

Returns type for multisegments.

property origin: *Point*

Returns origin.

property point_cls: Type[*Point*]

Returns type for points.

property locate_point_in_point_point_point_circle: Callable[[*Point*, *Point*, *Point*, *Point*], *Location*]

Returns location of point in point-point-point circle.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> (context.locate_point_in_point_point_point_circle(
...      Point(1, 1), Point(0, 0), Point(2, 0), Point(0, 2))
...  is Location.INTERIOR)
True
>>> (context.locate_point_in_point_point_point_circle(
...      Point(2, 2), Point(0, 0), Point(2, 0), Point(0, 2))
...  is Location.BOUNDARY)
True
>>> (context.locate_point_in_point_point_point_circle(
...      Point(3, 3), Point(0, 0), Point(2, 0), Point(0, 2))
...  is Location.EXTERIOR)
True
```

property points_squared_distance: Callable[[*Point*, *Point*], Scalar]

Returns squared Euclidean distance between two points.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> context.points_squared_distance(Point(0, 0), Point(0, 0)) == 0
True
>>> context.points_squared_distance(Point(0, 0), Point(1, 0)) == 1
True
>>> context.points_squared_distance(Point(0, 1), Point(1, 0)) == 2
True
```

property polygon_cls: Type[*Polygon*]

Returns type for polygons.

property region_signed_area: Callable[[*Contour*[Scalar]], Scalar]

Returns signed area of the region given its contour.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> (context.region_signed_area(Contour([Point(0, 0), Point(1, 0),
...                                      Point(1, 1), Point(0, 1)]))
...  == 1)
True
>>> (context.region_signed_area(Contour([Point(0, 0), Point(0, 1),
...                                      Point(1, 1), Point(1, 0)]))
```

(continues on next page)

(continued from previous page)

```
...  == -1)
True
```

property segment_cls: Type[*Segment*]

Returns type for segments.

property sqrt: Callable[[Scalar], Scalar]

Returns function for computing square root.

box_segment_squared_distance(*box:* Box, *segment:* Segment) → Scalar

Returns squared Euclidean distance between box and a segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Box = context.box_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> context.box_segment_squared_distance(
...     Box(0, 1, 0, 1), Segment(Point(0, 0), Point(1, 1))) == 0
True
>>> context.box_segment_squared_distance(
...     Box(0, 1, 0, 1), Segment(Point(2, 0), Point(2, 1))) == 1
True
>>> context.box_segment_squared_distance(
...     Box(0, 1, 0, 1), Segment(Point(2, 2), Point(3, 2))) == 2
True
```

contour_box(*contour:* Contour) → *Box*

Constructs box from contour.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where `vertices_count = len(contour.vertices)`.

```
>>> context = get_context()
>>> Box, Contour, Point = (context.box_cls, context.contour_cls,
...                        context.point_cls)
>>> (context.contour_box(Contour([Point(0, 0), Point(1, 0),
...                               Point(1, 1), Point(0, 1)]))
...  == Box(0, 1, 0, 1))
True
```

contour_centroid(*contour:* Contour) → *Point*

Constructs centroid of a contour.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Contour, Point = context.contour_cls, context.point_cls
>>> (context.contour_centroid(Contour([Point(0, 0), Point(2, 0),
...                                    Point(2, 2), Point(0, 2)]))
...  == Point(1, 1))
True
```

contour_length(*contour:* Contour) → Scalar

Returns Euclidean length of a contour.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> context.contour_length(Contour([Point(0, 0), Point(3, 0),
...                                 Point(0, 4)])) == 12
True
>>> context.contour_length(Contour([Point(0, 0), Point(1, 0),
...                                 Point(1, 1), Point(0, 1)])) == 4
True
```

contour_segments(*contour:* Contour) → Sequence[*Segment*]

Constructs segments of a contour.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.contour_segments(Contour([Point(0, 0), Point(2, 0),
...                                    Point(2, 2), Point(0, 2)]))
...  == [Segment(Point(0, 2), Point(0, 0)),
...      Segment(Point(0, 0), Point(2, 0)),
...      Segment(Point(2, 0), Point(2, 2)),
...      Segment(Point(2, 2), Point(0, 2))])
True
```

contours_box(*contours: Sequence[*Contour*]*) → *Box*

Constructs box from contours.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where `vertices_count = sum(len(contour.vertices) for contour in contours)`.

```
>>> context = get_context()
>>> Box = context.box_cls
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> (context.contours_box([Contour([Point(0, 0), Point(1, 0),
...                                 Point(1, 1), Point(0, 1)]),
...                         Contour([Point(1, 1), Point(2, 1),
...                                  Point(2, 2), Point(1, 2)])])
...  == Box(0, 2, 0, 2))
True
```

is_region_convex(*contour:* Contour) → bool

Checks if region (given its contour) is convex.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> context.is_region_convex(Contour([Point(0, 0), Point(3, 0),
...                                   Point(1, 1), Point(0, 3)]))
False
>>> context.is_region_convex(Contour([Point(0, 0), Point(2, 0),
...                                   Point(2, 2), Point(0, 2)]))
True
```

merged_box(*first_box:* Box, *second_box:* Box) → *Box*

Merges two boxes.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Box = context.box_cls
>>> (context.merged_box(Box(0, 1, 0, 1), Box(1, 2, 1, 2))
...  == Box(0, 2, 0, 2))
True
```

multipoint_centroid(*multipoint:* Multipoint) → *Point*

Constructs centroid of a multipoint.

Time complexity:
O(len(multipoint.points))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> context.multipoint_centroid(
...     Multipoint([Point(0, 0), Point(2, 0), Point(2, 2),
...                 Point(0, 2)])) == Point(1, 1)
True
```

multipolygon_centroid(*multipolygon:* Multipolygon) → *Point*

Constructs centroid of a multipolygon.

Time complexity:
O(len(vertices_count))

Memory complexity:
O(1)

where `vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Multipolygon = context.multipolygon_cls
>>> (context.multipolygon_centroid(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(1, 1), Point(0, 1)]),
...                           []),
...                   Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(2, 2), Point(1, 2)]),
...                           [])]))
...  == Point(1, 1))
True
```

multisegment_centroid(*multisegment:* Multisegment) → *Point*

Constructs centroid of a multisegment.

Time complexity:
O(len(multisegment.segments))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> Multisegment = context.multisegment_cls
>>> (context.multisegment_centroid(
...      Multisegment([Segment(Point(0, 0), Point(2, 0)),
...                    Segment(Point(2, 0), Point(2, 2)),
...                    Segment(Point(0, 2), Point(2, 2)),
...                    Segment(Point(0, 0), Point(0, 2))]))
```

(continues on next page)

(continued from previous page)

```
...  == Point(1, 1))
True
```

multisegment_length(*multisegment:* Multisegment) → Scalar

Returns Euclidean length of a multisegment.

Time complexity:
`O(len(multisegment.segments))`

Memory complexity:
`O(1)`

```
>>> context = get_context()
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> context.multisegment_length(
...     Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(0, 0), Point(0, 1))])) == 2
True
>>> context.multisegment_length(
...     Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(0, 0), Point(3, 4))])) == 6
True
```

points_convex_hull(*points: Sequence[*Point*]*) → Sequence[*Point*]

Constructs convex hull of points.

Time complexity:
`O(points_count * log(points_count))`

Memory complexity:
`O(points_count)`

where `points_count = len(points)`.

```
>>> context = get_context()
>>> Point = context.point_cls
>>> (context.points_convex_hull([Point(0, 0), Point(2, 0), Point(2, 2),
...                              Point(0, 2)])
...  == [Point(0, 0), Point(2, 0), Point(2, 2), Point(0, 2)])
True
```

points_box(*points: Sequence[*Point*]*) → *Box*

Constructs box from points.

Time complexity:
`O(len(points))`

Memory complexity:
`O(1)`

```
>>> context = get_context()
>>> Box, Point = context.box_cls, context.point_cls
>>> (context.points_box([Point(0, 0), Point(2, 0), Point(2, 2),
```

(continues on next page)

(continued from previous page)

```
...                       Point(0, 2)])
...  == Box(0, 2, 0, 2))
True
```

polygon_box(*polygon:* Polygon) → *Box*

Constructs box from polygon.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where `vertices_count = len(polygon.border.vertices)`.

```
>>> context = get_context()
>>> Box, Contour, Point, Polygon = (context.box_cls,
...                                 context.contour_cls,
...                                 context.point_cls,
...                                 context.polygon_cls)
>>> context.polygon_box(
...     Polygon(Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                      Point(0, 1)]), [])) == Box(0, 1, 0, 1)
True
```

polygon_centroid(*polygon:* Polygon) → *Point*

Constructs centroid of a polygon.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where `vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> context.polygon_centroid(
...     Polygon(Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                      Point(0, 4)]),
...             [Contour([Point(1, 1), Point(1, 3), Point(3, 3),
...                       Point(3, 1)])])) == Point(2, 2)
True
```

polygons_box(*polygons: Sequence[*Polygon*]*) → *Box*

Constructs box from polygons.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where `vertices_count = sum(len(polygon.border.vertices) for polygon in polygons)`.

```
>>> context = get_context()
>>> Box, Contour, Point, Polygon = (context.box_cls,
...                                 context.contour_cls,
...                                 context.point_cls,
...                                 context.polygon_cls)
>>> context.polygons_box(
...     [Polygon(Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                       Point(0, 1)]), []),
...      Polygon(Contour([Point(1, 1), Point(2, 1), Point(2, 2),
...                       Point(1, 2)]), [])]) == Box(0, 2, 0, 2)
True
```

region_centroid(*contour:* Contour) → *Point*

Constructs centroid of a region given its contour.

Time complexity:
`O(len(contour.vertices))`

Memory complexity:
`O(1)`

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> (context.region_centroid(Contour([Point(0, 0), Point(2, 0),
...                                   Point(2, 2), Point(0, 2)]))
...  == Point(1, 1))
True
```

replace(*, *box_cls: Optional[Type[Box]] = None*, *contour_cls: Optional[Type[Contour]] = None*, *empty_cls: Optional[Type[Empty]] = None*, *mix_cls: Optional[Type[Mix]] = None*, *multipoint_cls: Optional[Type[Multipoint]] = None*, *multipolygon_cls: Optional[Type[Multipolygon]] = None*, *multisegment_cls: Optional[Type[Multisegment]] = None*, *point_cls: Optional[Type[Point]] = None*, *polygon_cls: Optional[Type[Polygon]] = None*, *segment_cls: Optional[Type[Segment]] = None*, *mode: Optional[Mode] = None*, *sqrt: Optional[Callable[[Scalar], Scalar]] = None*) → *Context*

Constructs context from the original one replacing given parameters.

Time complexity:
`O(1)`

Memory complexity:
`O(1)`

```
>>> context = get_context()
>>> robust_context = context.replace(mode=Mode.ROBUST)
>>> isinstance(robust_context, Context)
True
>>> robust_context.mode is Mode.ROBUST
True
```

rotate_contour(*contour:* Contour, *cosine: Scalar*, *sine: Scalar*, *center:* Point) → *Contour*

Returns contour rotated by given angle around given center.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(len(contour.vertices))

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> (context.rotate_contour(
...      Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), 1, 0,
...      Point(0, 1))
...  == Contour([Point(0, 0), Point(1, 0), Point(0, 1)]))
True
>>> (context.rotate_contour(
...      Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), 0, 1,
...      Point(0, 1))
...  == Contour([Point(1, 1), Point(1, 2), Point(0, 1)]))
True
```

rotate_contour_around_origin(*contour:* Contour, *cosine: Scalar*, *sine: Scalar*) → *Contour*

Returns contour rotated by given angle around origin.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(len(contour.vertices))

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> (context.rotate_contour_around_origin(
...      Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), 1, 0)
...  == Contour([Point(0, 0), Point(1, 0), Point(0, 1)]))
True
>>> (context.rotate_contour_around_origin(
...      Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), 0, 1)
...  == Contour([Point(0, 0), Point(0, 1), Point(-1, 0)]))
True
```

rotate_multipoint(*multipoint:* Multipoint, *cosine: Scalar*, *sine: Scalar*, *center:* Point) → *Multipoint*

Returns multipoint rotated by given angle around given center.

Time complexity:
O(len(multipoint.points))

Memory complexity:
O(len(multipoint.points))

```
>>> context = get_context()
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> (context.rotate_multipoint(Multipoint([Point(0, 0), Point(1, 0)]),
...                            1, 0, Point(0, 1))
...  == Multipoint([Point(0, 0), Point(1, 0)]))
```

(continues on next page)

(continued from previous page)

```
True
>>> (context.rotate_multipoint(Multipoint([Point(0, 0), Point(1, 0)]),
...                            0, 1, Point(0, 1))
...  == Multipoint([Point(1, 1), Point(1, 2)]))
True
```

rotate_multipoint_around_origin(*multipoint:* Multipoint, *cosine: Scalar*, *sine: Scalar*) → *Multipoint*

Returns multipoint rotated by given angle around origin.

Time complexity:
`O(len(multipoint.points))`

Memory complexity:
`O(len(multipoint.points))`

```
>>> context = get_context()
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> (context.rotate_multipoint_around_origin(
...      Multipoint([Point(0, 0), Point(1, 0)]), 1, 0)
...  == Multipoint([Point(0, 0), Point(1, 0)]))
True
>>> (context.rotate_multipoint_around_origin(
...      Multipoint([Point(0, 0), Point(1, 0)]), 0, 1)
...  == Multipoint([Point(0, 0), Point(0, 1)]))
True
```

rotate_multipolygon(*multipolygon:* Multipolygon, *cosine: Scalar*, *sine: Scalar*, *center:* Point) → *Multipolygon*

Returns multipolygon rotated by given angle around given center.

Time complexity:
`O(vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> (context.rotate_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), [])]),
...      1, 0, Point(0, 1))
...  == Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                    Point(0, 1)]), [])]))
True
>>> (context.rotate_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
```

(continues on next page)

(continued from previous page)

```
...                                         Point(0, 1)]), [])]),
...       0, 1, Point(0, 1))
...  == Multipolygon([Polygon(Contour([Point(1, 1), Point(1, 2),
...                                    Point(0, 1)]), [])]))
True
```

rotate_multipolygon_around_origin(*multipolygon:* Multipolygon, *cosine: Scalar*, *sine: Scalar*) → *Multipolygon*

Returns multipolygon rotated by given angle around origin.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where `vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> (context.rotate_multipolygon_around_origin(
...       Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                      Point(0, 1)]), [])]),
...       1, 0)
...  == Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                    Point(0, 1)]), [])]))
True
>>> (context.rotate_multipolygon_around_origin(
...       Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                      Point(0, 1)]), [])]),
...       0, 1)
...  == Multipolygon([Polygon(Contour([Point(0, 0), Point(0, 1),
...                                    Point(-1, 0)]), [])]))
True
```

rotate_multisegment(*multisegment:* Multisegment, *cosine: Scalar*, *sine: Scalar*, *center:* Point) → *Multisegment*

Returns multisegment rotated by given angle around given center.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.rotate_multisegment(
```

(continues on next page)

(continued from previous page)

```
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 1, 0,
...      Point(0, 1))
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(0, 0), Point(0, 1))]))
True
>>> (context.rotate_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 0, 1,
...      Point(0, 1))
...  == Multisegment([Segment(Point(1, 1), Point(1,2)),
...                   Segment(Point(1, 1), Point(0, 1))]))
True
```

rotate_multisegment_around_origin(*multisegment:* Multisegment, *cosine: Scalar*, *sine: Scalar*) → *Multisegment*

Returns multisegment rotated by given angle around origin.

Time complexity:

O(len(multisegment.segments))

Memory complexity:

O(len(multisegment.segments))

```
>>> context = get_context()
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.rotate_multisegment_around_origin(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 1, 0)
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(0, 0), Point(0, 1))]))
True
>>> (context.rotate_multisegment_around_origin(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 0, 1)
...  == Multisegment([Segment(Point(0, 0), Point(0, 1)),
...                   Segment(Point(0, 0), Point(-1, 0))]))
True
```

rotate_point(*point:* Point, *cosine: Scalar*, *sine: Scalar*, *center:* Point) → *Point*

Returns point rotated by given angle around given center.

Time complexity:

O(1)

Memory complexity:

O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> context.rotate_point(Point(1, 0), 1, 0, Point(0, 1)) == Point(1, 0)
True
```

(continues on next page)

(continued from previous page)

```
>>> context.rotate_point(Point(1, 0), 0, 1, Point(0, 1)) == Point(1, 2)
True
```

rotate_point_around_origin(*point:* Point, *cosine: Scalar*, *sine: Scalar*) → *Point*

Returns point rotated by given angle around origin.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> (context.rotate_point_around_origin(Point(1, 0), 1, 0)
...  == Point(1, 0))
True
>>> (context.rotate_point_around_origin(Point(1, 0), 0, 1)
...  == Point(0, 1))
True
```

rotate_polygon(*polygon:* Polygon, *cosine: Scalar*, *sine: Scalar*, *center:* Point) → *Polygon*

Returns polygon rotated by given angle around given center.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where `vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> (context.rotate_polygon(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      1, 0, Point(0, 1))
...  == Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []))
True
>>> (context.rotate_polygon(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      0, 1, Point(0, 1))
...  == Polygon(Contour([Point(1, 1), Point(1, 2), Point(0, 1)]), []))
True
```

rotate_polygon_around_origin(*polygon:* Polygon, *cosine: Scalar*, *sine: Scalar*) → *Polygon*

Returns polygon rotated by given angle around origin.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where `vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> (context.rotate_polygon_around_origin(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      1, 0)
...  == Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []))
True
>>> (context.rotate_polygon_around_origin(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      0, 1)
...  == Polygon(Contour([Point(0, 0), Point(0, 1), Point(-1, 0)]), []))
True
```

rotate_segment(*segment:* Segment, *cosine: Scalar*, *sine: Scalar*, *center:* Point) → *Segment*

Returns segment rotated by given angle around given center.

Time complexity:

`O(1)`

Memory complexity:

`O(1)`

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.rotate_segment(Segment(Point(0, 0), Point(1, 0)), 1, 0,
...                         Point(0, 1))
...  == Segment(Point(0, 0), Point(1, 0)))
True
>>> (context.rotate_segment(Segment(Point(0, 0), Point(1, 0)), 0, 1,
...                         Point(0, 1))
...  == Segment(Point(1, 1), Point(1, 2)))
True
```

rotate_segment_around_origin(*segment:* Segment, *cosine: Scalar*, *sine: Scalar*) → *Segment*

Returns segment rotated by given angle around origin.

Time complexity:

`O(1)`

Memory complexity:

`O(1)`

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.rotate_segment_around_origin(
...      Segment(Point(0, 0), Point(1, 0)), 1, 0)
...  == Segment(Point(0, 0), Point(1, 0)))
True
```

(continues on next page)

(continued from previous page)

```
>>> (context.rotate_segment_around_origin(
...      Segment(Point(0, 0), Point(1, 0)), 0, 1)
...  == Segment(Point(0, 0), Point(0, 1)))
True
```

scale_contour(*contour:* Contour, *factor_x: Scalar*, *factor_y: Scalar*) → Union[*Contour*, *Multipoint*, *Segment*]

Returns contour scaled by given factor.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(len(contour.vertices))

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.scale_contour(Contour([Point(0, 0), Point(1, 0),
...                                 Point(0, 1)]), 0, 0)
...  == Multipoint([Point(0, 0)]))
True
>>> (context.scale_contour(Contour([Point(0, 0), Point(1, 0),
...                                 Point(0, 1)]), 1, 0)
...  == Segment(Point(0, 0), Point(1, 0)))
True
>>> (context.scale_contour(Contour([Point(0, 0), Point(1, 0),
...                                 Point(0, 1)]), 0, 1)
...  == Segment(Point(0, 0), Point(0, 1)))
True
>>> (context.scale_contour(Contour([Point(0, 0), Point(1, 0),
...                                 Point(0, 1)]), 1, 1)
...  == Contour([Point(0, 0), Point(1, 0), Point(0, 1)]))
True
```

scale_multipoint(*multipoint:* Multipoint, *factor_x: Scalar*, *factor_y: Scalar*) → *Multipoint*

Returns multipoint scaled by given factor.

Time complexity:
O(len(multipoint.points))

Memory complexity:
O(len(multipoint.points))

```
>>> context = get_context()
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> (context.scale_multipoint(Multipoint([Point(0, 0), Point(1, 1)]),
...                           0, 0)
...  == Multipoint([Point(0, 0)]))
True
>>> (context.scale_multipoint(Multipoint([Point(0, 0), Point(1, 1)]),
```

(continues on next page)

(continued from previous page)

```
...                              1, 0)
...  == Multipoint([Point(0, 0), Point(1, 0)]))
True
>>> (context.scale_multipoint(Multipoint([Point(0, 0), Point(1, 1)]),
...                           0, 1)
...  == Multipoint([Point(0, 0), Point(0, 1)]))
True
>>> (context.scale_multipoint(Multipoint([Point(0, 0), Point(1, 1)]),
...                           1, 1)
...  == Multipoint([Point(0, 0), Point(1, 1)]))
True
```

scale_multipolygon(*multipolygon:* Multipolygon, *factor_x: Scalar*, *factor_y: Scalar*) → Union[*Multipoint*, *Multipolygon*, *Multisegment*]

Returns multipolygon scaled by given factor.

Time complexity:

`O(vertices_count)`

Memory complexity:

`O(vertices_count)`

where `vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipoint = context.multipoint_cls
>>> Multipolygon = context.multipolygon_cls
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Segment = context.segment_cls
>>> (context.scale_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 0, 0)
...  == Multipoint([Point(0, 0)]))
True
>>> (context.scale_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 1, 0)
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(1, 0), Point(2, 0))]))
True
>>> (context.scale_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 0, 1)
```

(continues on next page)

(continued from previous page)

```
...   == Multisegment([Segment(Point(0, 0), Point(0, 1)),
...                    Segment(Point(0, 1), Point(0, 2))]))
True
>>> (context.scale_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 1, 1)
...   == Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]))
True
```

scale_multisegment(*multisegment:* Multisegment, *factor_x: Scalar*, *factor_y: Scalar*) → Union[*Mix*, *Multipoint*, *Multisegment*]

Returns multisegment scaled by given factor.

Time complexity:
O(len(multisegment.segments))

Memory complexity:
O(len(multisegment.segments))

```
>>> context = get_context()
>>> EMPTY = context.empty
>>> Mix = context.mix_cls
>>> Multipoint = context.multipoint_cls
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.scale_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 0, 0)
...   == Multipoint([Point(0, 0)]))
True
>>> (context.scale_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 1, 0)
...   == Mix(Multipoint([Point(0, 0)]),
...          Segment(Point(0, 0), Point(1, 0)), EMPTY))
True
>>> (context.scale_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 0, 1)
...   == Mix(Multipoint([Point(0, 0)]),
...          Segment(Point(0, 0), Point(0, 1)), EMPTY))
True
>>> (context.scale_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 1, 1)
...   == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]))
```

(continues on next page)

(continued from previous page)

```
True
```

scale_point(*point:* Point, *factor_x: Scalar*, *factor_y: Scalar*) → *Point*

Returns point scaled by given factor.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> context.scale_point(Point(1, 1), 0, 0) == Point(0, 0)
True
>>> context.scale_point(Point(1, 1), 1, 0) == Point(1, 0)
True
>>> context.scale_point(Point(1, 1), 0, 1) == Point(0, 1)
True
>>> context.scale_point(Point(1, 1), 1, 1) == Point(1, 1)
True
```

scale_polygon(*polygon:* Polygon, *factor_x: Scalar*, *factor_y: Scalar*) → Union[*Multipoint*, *Polygon*, *Segment*]

Returns polygon scaled by given factor.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where `vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Segment = context.segment_cls
>>> (context.scale_polygon(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      0, 0)
...  == Multipoint([Point(0, 0)]))
True
>>> (context.scale_polygon(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      1, 0)
...  == Segment(Point(0, 0), Point(1, 0)))
True
>>> (context.scale_polygon(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      0, 1)
```

(continues on next page)

(continued from previous page)

```
...   == Segment(Point(0, 0), Point(0, 1)))
True
>>> (context.scale_polygon(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...      1, 1)
...   == Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []))
True
```

scale_segment(*segment:* Segment, *factor_x: Scalar*, *factor_y: Scalar*) → Union[*Multipoint*, *Segment*]

Returns segment scaled by given factor.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.scale_segment(Segment(Point(0, 0), Point(1, 1)), 0, 0)
...   == Multipoint([Point(0, 0)]))
True
>>> (context.scale_segment(Segment(Point(0, 0), Point(1, 1)), 1, 0)
...   == Segment(Point(0, 0), Point(1, 0)))
True
>>> (context.scale_segment(Segment(Point(0, 0), Point(1, 1)), 0, 1)
...   == Segment(Point(0, 0), Point(0, 1)))
True
>>> (context.scale_segment(Segment(Point(0, 0), Point(1, 1)), 1, 1)
...   == Segment(Point(0, 0), Point(1, 1)))
True
```

segment_box(*segment:* Segment) → *Box*

Constructs box from segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Box, Point, Segment = (context.box_cls, context.point_cls,
...                        context.segment_cls)
>>> (context.segment_box(Segment(Point(0, 1), Point(2, 3)))
...   == Box(0, 2, 1, 3))
True
```

segment_centroid(*segment:* Segment) → *Point*

Constructs centroid of a segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point, Segment = context.point_cls, context.segment_cls
>>> (context.segment_centroid(Segment(Point(0, 1), Point(2, 3)))
...  == Point(1, 2))
True
```

segment_contains_point(*segment:* Segment, *point:* Point) → bool

Checks if a segment contains given point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> context.segment_contains_point(Segment(Point(0, 0), Point(2, 0)),
...                                Point(0, 0))
True
>>> context.segment_contains_point(Segment(Point(0, 0), Point(2, 0)),
...                                Point(0, 2))
False
>>> context.segment_contains_point(Segment(Point(0, 0), Point(2, 0)),
...                                Point(1, 0))
True
>>> context.segment_contains_point(Segment(Point(0, 0), Point(2, 0)),
...                                Point(1, 1))
False
>>> context.segment_contains_point(Segment(Point(0, 0), Point(2, 0)),
...                                Point(2, 0))
True
>>> context.segment_contains_point(Segment(Point(0, 0), Point(2, 0)),
...                                Point(3, 0))
False
```

segment_length(*segment:* Segment) → Scalar

Returns Euclidean length of a segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> context.segment_length(Segment(Point(0, 0), Point(1, 0))) == 1
True
>>> context.segment_length(Segment(Point(0, 0), Point(0, 1))) == 1
```

(continues on next page)

(continued from previous page)

```
True
>>> context.segment_length(Segment(Point(0, 0), Point(3, 4))) == 5
True
```

segment_point_squared_distance(*segment:* Segment, *point:* Point) → Scalar

Returns squared Euclidean distance between segment and a point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> context.segment_point_squared_distance(
...     Segment(Point(0, 0), Point(1, 0)), Point(0, 0)) == 0
True
>>> context.segment_point_squared_distance(
...     Segment(Point(0, 0), Point(1, 0)), Point(0, 1)) == 1
True
>>> context.segment_point_squared_distance(
...     Segment(Point(0, 0), Point(1, 0)), Point(2, 1)) == 2
True
```

segments_box(*segments: Sequence[*Segment*]*) → *Box*

Constructs box from segments.

Time complexity:
O(len(segments))

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Box, Point, Segment = (context.box_cls, context.point_cls,
...                        context.segment_cls)
>>> (context.segments_box([Segment(Point(0, 0), Point(1, 1)),
...                        Segment(Point(1, 1), Point(2, 2))])
...  == Box(0, 2, 0, 2))
True
```

segments_intersection(*first:* Segment, *second:* Segment) → *Point*

Returns intersection point of two segments.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
```

(continues on next page)

(continued from previous page)

```
>>> (context.segments_intersection(Segment(Point(0, 0), Point(2, 0)),
...                                Segment(Point(0, 0), Point(0, 1)))
...  == Point(0, 0))
True
>>> (context.segments_intersection(Segment(Point(0, 0), Point(2, 0)),
...                                Segment(Point(1, 0), Point(1, 1)))
...  == Point(1, 0))
True
>>> (context.segments_intersection(Segment(Point(0, 0), Point(2, 0)),
...                                Segment(Point(2, 0), Point(3, 0)))
...  == Point(2, 0))
True
```

segments_relation(*test:* Segment, *goal:* Segment) → *Relation*

Returns relation between two segments.

Time complexity:

O(1)

Memory complexity:

O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.segments_relation(Segment(Point(0, 0), Point(2, 2)),
...                            Segment(Point(1, 0), Point(2, 0)))
...  is Relation.DISJOINT)
True
>>> (context.segments_relation(Segment(Point(0, 0), Point(2, 2)),
...                            Segment(Point(0, 0), Point(2, 0)))
...  is Relation.TOUCH)
True
>>> (context.segments_relation(Segment(Point(0, 0), Point(2, 2)),
...                            Segment(Point(2, 0), Point(0, 2)))
...  is Relation.CROSS)
True
>>> (context.segments_relation(Segment(Point(0, 0), Point(2, 2)),
...                           Segment(Point(0, 0), Point(1, 1)))
...  is Relation.COMPOSITE)
True
>>> (context.segments_relation(Segment(Point(0, 0), Point(2, 2)),
...                            Segment(Point(0, 0), Point(2, 2)))
...  is Relation.EQUAL)
True
>>> (context.segments_relation(Segment(Point(0, 0), Point(2, 2)),
...                            Segment(Point(0, 0), Point(3, 3)))
...  is Relation.COMPONENT)
True
>>> (context.segments_relation(Segment(Point(0, 0), Point(2, 2)),
...                            Segment(Point(1, 1), Point(3, 3)))
...  is Relation.OVERLAP)
True
```

segments_squared_distance(*first:* Segment, *second:* Segment) → Scalar

Returns squared Euclidean distance between two segments.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> context.segments_squared_distance(
...     Segment(Point(0, 0), Point(1, 0)),
...     Segment(Point(0, 0), Point(0, 1))) == 0
True
>>> context.segments_squared_distance(
...     Segment(Point(0, 0), Point(1, 0)),
...     Segment(Point(0, 1), Point(1, 1))) == 1
True
>>> context.segments_squared_distance(
...     Segment(Point(0, 0), Point(1, 0)),
...     Segment(Point(2, 1), Point(2, 2))) == 2
True
```

translate_contour(*contour:* Contour, *step_x: Scalar*, *step_y: Scalar*) → *Contour*

Returns contour translated by given step.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(len(contour.vertices))

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.translate_contour(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), 0, 0)
...  == Contour([Point(0, 0), Point(1, 0), Point(0, 1)]))
True
>>> (context.translate_contour(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), 1, 0)
...  == Contour([Point(1, 0), Point(2, 0), Point(1, 1)]))
True
>>> (context.translate_contour(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), 0, 1)
...  == Contour([Point(0, 1), Point(1, 1), Point(0, 2)]))
True
>>> (context.translate_contour(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), 1, 1)
...  == Contour([Point(1, 1), Point(2, 1), Point(1, 2)]))
True
```

translate_multipoint(*multipoint:* Multipoint, *step_x: Scalar*, *step_y: Scalar*) → *Multipoint*

Returns multipoint translated by given step.

Time complexity:
O(len(multipoint.points))

Memory complexity:
O(len(multipoint.points))

```
>>> context = get_context()
>>> Multipoint = context.multipoint_cls
>>> Point = context.point_cls
>>> (context.translate_multipoint(Multipoint([Point(0, 0),
...                                           Point(1, 0)]), 0, 0)
...  == Multipoint([Point(0, 0), Point(1, 0)]))
True
>>> (context.translate_multipoint(Multipoint([Point(0, 0),
...                                           Point(1, 0)]), 1, 0)
...  == Multipoint([Point(1, 0), Point(2, 0)]))
True
>>> (context.translate_multipoint(Multipoint([Point(0, 0),
...                                           Point(1, 0)]), 0, 1)
...  == Multipoint([Point(0, 1), Point(1, 1)]))
True
>>> (context.translate_multipoint(Multipoint([Point(0, 0),
...                                           Point(1, 0)]), 1, 1)
...  == Multipoint([Point(1, 1), Point(2, 1)]))
True
```

translate_multipolygon(*multipolygon:* Multipolygon, *step_x: Scalar*, *step_y: Scalar*) → *Multipolygon*

Returns multipolygon translated by given step.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where `vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> (context.translate_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 0, 0)
...  == Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                    Point(0, 1)]), []),
...                   Polygon(Contour([Point(1, 1), Point(2, 1),
...                                    Point(1, 2)]), [])]))
True
```

(continues on next page)

(continued from previous page)

```
>>> (context.translate_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 1, 0)
...  == Multipolygon([Polygon(Contour([Point(1, 0), Point(2, 0),
...                                    Point(1, 1)]), []),
...                   Polygon(Contour([Point(2, 1), Point(3, 1),
...                                    Point(2, 2)]), [])]))
True
>>> (context.translate_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 0, 1)
...  == Multipolygon([Polygon(Contour([Point(0, 1), Point(1, 1),
...                                    Point(0, 2)]), []),
...                   Polygon(Contour([Point(1, 2), Point(2, 2),
...                                    Point(1, 3)]), [])]))
True
>>> (context.translate_multipolygon(
...      Multipolygon([Polygon(Contour([Point(0, 0), Point(1, 0),
...                                     Point(0, 1)]), []),
...                    Polygon(Contour([Point(1, 1), Point(2, 1),
...                                     Point(1, 2)]), [])]), 1, 1)
...  == Multipolygon([Polygon(Contour([Point(1, 1), Point(2, 1),
...                                    Point(1, 2)]), []),
...                   Polygon(Contour([Point(2, 2), Point(3, 2),
...                                    Point(2, 3)]), [])]))
True
```

translate_multisegment(*multisegment:* Multisegment, *step_x: Scalar*, *step_y: Scalar*) → *Multisegment*

Returns multisegment translated by given step.

Time complexity:

O(len(multisegment.segments))

Memory complexity:

O(len(multisegment.segments))

```
>>> context = get_context()
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.translate_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 0, 0)
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(0, 0), Point(0, 1))]))
True
>>> (context.translate_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 1, 0)
```

(continues on next page)

(continued from previous page)

```
...   == Multisegment([Segment(Point(1, 0), Point(2, 0)),
...                    Segment(Point(1, 0), Point(1, 1))]))
True
>>> (context.translate_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 0, 1)
...   == Multisegment([Segment(Point(0, 1), Point(1, 1)),
...                    Segment(Point(0, 1), Point(0, 2))]))
True
>>> (context.translate_multisegment(
...      Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                    Segment(Point(0, 0), Point(0, 1))]), 1, 1)
...   == Multisegment([Segment(Point(1, 1), Point(2, 1)),
...                    Segment(Point(1, 1), Point(1, 2))]))
True
```

translate_point(*point:* Point, *step_x: Scalar*, *step_y: Scalar*) → *Point*

Returns point translated by given step.

Time complexity:

O(1)

Memory complexity:

O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> context.translate_point(Point(0, 0), 0, 0) == Point(0, 0)
True
>>> context.translate_point(Point(0, 0), 1, 0) == Point(1, 0)
True
>>> context.translate_point(Point(0, 0), 0, 1) == Point(0, 1)
True
>>> context.translate_point(Point(0, 0), 1, 1) == Point(1, 1)
True
```

translate_polygon(*polygon:* Polygon, *step_x: Scalar*, *step_y: Scalar*) → *Polygon*

Returns polygon translated by given step.

Time complexity:

O(vertices_count)

Memory complexity:

O(vertices_count)

where `vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

```
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> (context.translate_polygon(
...      Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
```

(continues on next page)

(continued from previous page)

```
...       0, 0)
...    == Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []))
True
>>> (context.translate_polygon(
...       Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...       1, 0)
...    == Polygon(Contour([Point(1, 0), Point(2, 0), Point(1, 1)]), []))
True
>>> (context.translate_polygon(
...       Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...       0, 1)
...    == Polygon(Contour([Point(0, 1), Point(1, 1), Point(0, 2)]), []))
True
>>> (context.translate_polygon(
...       Polygon(Contour([Point(0, 0), Point(1, 0), Point(0, 1)]), []),
...       1, 1)
...    == Polygon(Contour([Point(1, 1), Point(2, 1), Point(1, 2)]), []))
True
```

translate_segment(*segment:* Segment, *step_x: Scalar*, *step_y: Scalar*) → *Segment*

Returns segment translated by given step.

Time complexity:

O(1)

Memory complexity:

O(1)

```
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> (context.translate_segment(Segment(Point(0, 0), Point(1, 0)), 0, 0)
...    == Segment(Point(0, 0), Point(1, 0)))
True
>>> (context.translate_segment(Segment(Point(0, 0), Point(1, 0)), 1, 0)
...    == Segment(Point(1, 0), Point(2, 0)))
True
>>> (context.translate_segment(Segment(Point(0, 0), Point(1, 0)), 0, 1)
...    == Segment(Point(0, 1), Point(1, 1)))
True
>>> (context.translate_segment(Segment(Point(0, 0), Point(1, 0)), 1, 1)
...    == Segment(Point(1, 1), Point(2, 1)))
True
```

ground.base.**get_context**() → *Context*

Returns current context.

ground.base.**set_context**(*context:* Context) → None

Sets current context.

CHAPTER

TWO

HINTS MODULE

class ground.hints.**Box**(*min_x: Scalar*, *max_x: Scalar*, *min_y: Scalar*, *max_y: Scalar*)

Box is a limited closed region defined by axis-aligned rectangular contour.

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *min_x: Scalar*, *max_x: Scalar*, *min_y: Scalar*, *max_y: Scalar*) → *Box*

Constructs box given its coordinates limits.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property max_x: Scalar

Maximum x-coordinate of the box.

abstract property max_y: Scalar

Maximum y-coordinate of the box.

abstract property min_x: Scalar

Minimum x-coordinate of the box.

abstract property min_y: Scalar

Minimum y-coordinate of the box.

class ground.hints.**Contour**(*vertices: Sequence[*Point*]*)

Contour is a linear geometry that represents closed simple polyline defined by a sequence of points (called *contour's vertices*).

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *vertices: Sequence[*Point*]*) → *Contour*

Constructs contour given its vertices.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property vertices: Sequence[*Point*]

Vertices of the contour.

class ground.hints.**Empty**

Represents an empty set of points.

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*) → *Empty*

Constructs empty geometry.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

class ground.hints.**Mix**(*discrete: Union[*Empty, Multipoint*]*, *linear: Union[*Empty, Segment, Multisegment, Contour*]*, *shaped: Union[*Empty, Polygon, Multipolygon*]*)

Mix is a set of two or more non-empty geometries with different dimensions.

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *discrete: Union[*Empty, Multipoint*]*, *linear: Union[*Empty, Segment, Multisegment, Contour*]*, *shaped: Union[*Empty, Polygon, Multipolygon*]*) → *Mix*

Constructs mix given its components.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property discrete: Union[*Empty*, *Multipoint*]

Discrete component of the mix.

abstract property linear: Union[*Empty*, *Segment*, *Multisegment*, *Contour*]

Linear component of the mix.

abstract property shaped: Union[*Empty*, *Polygon*, *Multipolygon*]

Shaped component of the mix.

class ground.hints.**Multipoint**(*points: Sequence[*Point*]*)

Multipoint is a discrete geometry that represents non-empty set of unique points.

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *points: Sequence[*Point*]*) → *Multipoint*

Constructs multipoint given its points.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property points: Sequence[*Point*]

Points of the multipoint.

class ground.hints.**Multipolygon**(*polygons: Sequence[*Polygon*]*)

Multipolygon is a shaped geometry that represents set of two or more non-overlapping polygons intersecting only in discrete set of points.

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *polygons: Sequence[*Polygon*]*) → *Multipolygon*

Constructs multipolygon given its polygons.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property polygons: Sequence[*Polygon*]

Polygons of the multipolygon.

class ground.hints.**Multisegment**(*segments: Sequence[*Segment*]*)

Multisegment is a linear geometry that represents set of two or more non-crossing and non-overlapping segments.

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *segments: Sequence[*Segment*]*) → *Multisegment*

Constructs multisegment given its segments.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property segments: Sequence[*Segment*]

Segments of the multisegment.

class ground.hints.**Point**(*x: Scalar*, *y: Scalar*)

Point is a minimal element of the plane defined by pair of real numbers (called *point's coordinates*).

Points considered to be sorted lexicographically, with abscissas being compared first.

abstract __ge__(*other:* Point) → bool

Checks if the point is greater than or equal to the other.

abstract __gt__(*other:* Point) → bool

Checks if the point is greater than the other.

abstract __hash__() → int

Returns hash value of the point.

__init__(**args*, ***kwargs*)

abstract __le__(*other:* Point) → bool

Checks if the point is less than or equal to the other.

abstract __lt__(*other:* Point) → bool

Checks if the point is less than the other.

abstract static __new__(*cls*, *x: Scalar*, *y: Scalar*) → *Point*

Constructs point given its coordinates.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property x: Scalar

Abscissa of the point.

abstract property y: Scalar

Ordinate of the point.

class ground.hints.**Polygon**(*border:* Contour, *holes: Sequence[*Contour*]*)

Polygon is a shaped geometry that represents limited closed region defined by the pair of outer contour (called *polygon's border*) and possibly empty sequence of inner contours (called *polygon's holes*).

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *border:* Contour, *holes: Sequence[*Contour*]*) → *Polygon*

Constructs polygon given its border and holes.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property border: *Contour*

Border of the polygon.

abstract property holes: Sequence[*Contour*]

Holes of the polygon.

class ground.hints.**Segment**(*start:* Point, *end:* Point)

Segment (or **line segment**) is a linear geometry that represents a limited continuous part of the line containing more than one point defined by a pair of unequal points (called *segment's endpoints*).

__init__(**args*, ***kwargs*)

abstract static __new__(*cls*, *start:* Point, *end:* Point) → *Segment*

Constructs segment given its endpoints.

__subclasshook__()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

abstract property end: *Point*

End endpoint of the segment.

abstract property start: *Point*

Start endpoint of the segment.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

G

H

I

K

L

M

O

P

R

S

T

V

W

X

Y