

---

# **ground**

***Release 9.0.0***

**Azat Ibrakov**

**May 12, 2023**



# CONTENTS

|          |                            |           |
|----------|----------------------------|-----------|
| <b>1</b> | <b>base module</b>         | <b>3</b>  |
| <b>2</b> | <b>hints module</b>        | <b>35</b> |
|          | <b>Python Module Index</b> | <b>39</b> |
|          | <b>Index</b>               | <b>41</b> |



---

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

---



## 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:**

$O(\text{len}(\text{contour.vertices}))$

**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)]))
...  == 0)
True
```

(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`)  $\rightarrow$  `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`)  $\rightarrow$  `Box`

Constructs box from contour.

**Time complexity:**

$O(\text{vertices\_count})$

**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`)  $\rightarrow$  `Point`

Constructs centroid of a contour.

**Time complexity:**

$O(\text{len}(\text{contour.vertices}))$

**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:** $O(\text{len}(\text{contour.vertices}))$ **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:** $O(\text{len}(\text{contour.vertices}))$ **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:** $O(\text{vertices\_count})$

**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:** $O(\text{len}(\text{contour.vertices}))$ **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:** $O(\text{len}(\text{multipoint.points}))$ **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:**

$O(\text{len}(\text{vertices\_count}))$

**Memory complexity:**

$O(1)$

where  $\text{vertices\_count} = \text{sum}(\text{len}(\text{polygon.border.vertices}) + \text{sum}(\text{len}(\text{hole.vertices})$   
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:**

$O(\text{len}(\text{multisegment.segments}))$

**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:**

$O(\text{len}(\text{multisegment.segments}))$

**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:**

$O(\text{points\_count} * \log(\text{points\_count}))$

**Memory complexity:**

$O(\text{points\_count})$

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:**

$O(\text{len}(\text{points}))$

**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:**

$O(\text{vertices\_count})$

**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:**

$O(\text{vertices\_count})$

**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:**

$O(\text{vertices\_count})$

**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:**

$O(\text{len}(\text{contour.vertices}))$

**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:** $O(\text{len}(\text{contour.vertices}))$ **Memory complexity:** $O(\text{len}(\text{contour.vertices}))$ 

```

>>> 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:** $O(\text{len}(\text{contour.vertices}))$ **Memory complexity:** $O(\text{len}(\text{contour.vertices}))$ 

```

>>> 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:** $O(\text{len}(\text{multipoint.points}))$ **Memory complexity:** $O(\text{len}(\text{multipoint.points}))$ 

```

>>> 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:**

$O(\text{len}(\text{multipoint.points}))$

**Memory complexity:**

$O(\text{len}(\text{multipoint.points}))$

```

>>> 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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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:**

$O(\text{len}(\text{multisegment.segments}))$

**Memory complexity:**

$O(\text{len}(\text{multisegment.segments}))$

```

>>> 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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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:**

$O(\text{len}(\text{contour.vertices}))$

**Memory complexity:**

$O(\text{len}(\text{contour.vertices}))$

```

>>> 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:**

$O(\text{len}(\text{multipoint.points}))$

**Memory complexity:**

$O(\text{len}(\text{multipoint.points}))$

```

>>> 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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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)]), [])[0, 1])
... == 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])
... == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...     Segment(Point(1, 0), Point(2, 0))]))
True

```

(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:**

$O(\text{len}(\text{multisegment.segments}))$

**Memory complexity:**

$O(\text{len}(\text{multisegment.segments}))$

```

>>> 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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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:**

$O(\text{len}(\text{segments}))$

**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:**

$O(\text{len}(\text{contour.vertices}))$

**Memory complexity:**

$O(\text{len}(\text{contour.vertices}))$

```
>>> 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:**

$O(\text{len}(\text{multipoint.points}))$

**Memory complexity:**

$O(\text{len}(\text{multipoint.points}))$

```
>>> 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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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:**

$O(\text{len}(\text{multisegment.segments}))$

**Memory complexity:**

$O(\text{len}(\text{multisegment.segments}))$

```

>>> 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:**

$O(\text{vertices\_count})$

**Memory complexity:**

$O(\text{vertices\_count})$

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.

## 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

### g

`ground.base`, [3](#)  
`ground.hints`, [35](#)



## Symbols

\_\_ge\_\_() (ground.hints.Point method), 37  
 \_\_gt\_\_() (ground.hints.Point method), 37  
 \_\_hash\_\_() (ground.hints.Point method), 37  
 \_\_init\_\_() (ground.hints.Box method), 35  
 \_\_init\_\_() (ground.hints.Contour method), 35  
 \_\_init\_\_() (ground.hints.Empty method), 36  
 \_\_init\_\_() (ground.hints.Mix method), 36  
 \_\_init\_\_() (ground.hints.Multipoint method), 36  
 \_\_init\_\_() (ground.hints.Multipolygon method), 37  
 \_\_init\_\_() (ground.hints.Multisegment method), 37  
 \_\_init\_\_() (ground.hints.Point method), 37  
 \_\_init\_\_() (ground.hints.Polygon method), 38  
 \_\_init\_\_() (ground.hints.Segment method), 38  
 \_\_le\_\_() (ground.hints.Point method), 37  
 \_\_lt\_\_() (ground.hints.Point method), 37  
 \_\_new\_\_() (ground.hints.Box static method), 35  
 \_\_new\_\_() (ground.hints.Contour static method), 35  
 \_\_new\_\_() (ground.hints.Empty static method), 36  
 \_\_new\_\_() (ground.hints.Mix static method), 36  
 \_\_new\_\_() (ground.hints.Multipoint static method), 36  
 \_\_new\_\_() (ground.hints.Multipolygon static method), 37  
 \_\_new\_\_() (ground.hints.Multisegment static method), 37  
 \_\_new\_\_() (ground.hints.Point static method), 37  
 \_\_new\_\_() (ground.hints.Polygon static method), 38  
 \_\_new\_\_() (ground.hints.Segment static method), 38  
 \_\_subclasshook\_\_() (ground.hints.Box method), 35  
 \_\_subclasshook\_\_() (ground.hints.Contour method), 35  
 \_\_subclasshook\_\_() (ground.hints.Empty method), 36  
 \_\_subclasshook\_\_() (ground.hints.Mix method), 36  
 \_\_subclasshook\_\_() (ground.hints.Multipoint method), 36  
 \_\_subclasshook\_\_() (ground.hints.Multipolygon method), 37  
 \_\_subclasshook\_\_() (ground.hints.Multisegment method), 37  
 \_\_subclasshook\_\_() (ground.hints.Point method), 38  
 \_\_subclasshook\_\_() (ground.hints.Polygon method), 38

\_\_subclasshook\_\_() (ground.hints.Segment method), 38

## A

ACUTE (ground.base.Kind attribute), 3  
 angle\_kind (ground.base.Context property), 5  
 angle\_orientation (ground.base.Context property), 5

## B

border (ground.hints.Polygon property), 38  
 BOUNDARY (ground.base.Location attribute), 3  
 Box (class in ground.hints), 35  
 box\_cls (ground.base.Context property), 6  
 box\_point\_squared\_distance (ground.base.Context property), 6  
 box\_segment\_squared\_distance() (ground.base.Context method), 9

## C

CLOCKWISE (ground.base.Orientation attribute), 3  
 COLLINEAR (ground.base.Orientation attribute), 3  
 COMPONENT (ground.base.Relation attribute), 4  
 COMPOSITE (ground.base.Relation attribute), 4  
 Context (class in ground.base), 4  
 Contour (class in ground.hints), 35  
 contour\_box() (ground.base.Context method), 9  
 contour\_centroid() (ground.base.Context method), 9  
 contour\_cls (ground.base.Context property), 6  
 contour\_length() (ground.base.Context method), 10  
 contour\_segments() (ground.base.Context method), 10  
 contours\_box() (ground.base.Context method), 10  
 COUNTERCLOCKWISE (ground.base.Orientation attribute), 3  
 COVER (ground.base.Relation attribute), 4  
 CROSS (ground.base.Relation attribute), 4  
 cross\_product (ground.base.Context property), 6

## D

discrete (ground.hints.Mix property), 36  
 DISJOINT (ground.base.Relation attribute), 3  
 dot\_product (ground.base.Context property), 6

## E

`Empty` (class in `ground.hints`), 35  
`empty` (`ground.base.Context` property), 7  
`empty_cls` (`ground.base.Context` property), 7  
`ENCLOSED` (`ground.base.Relation` attribute), 4  
`ENCLOSES` (`ground.base.Relation` attribute), 4  
`end` (`ground.hints.Segment` property), 38  
`EQUAL` (`ground.base.Relation` attribute), 4  
`EXTERIOR` (`ground.base.Location` attribute), 3

## G

`get_context()` (in module `ground.base`), 34  
`ground.base`  
    module, 3  
`ground.hints`  
    module, 35

## H

`holes` (`ground.hints.Polygon` property), 38

## I

`INTERIOR` (`ground.base.Location` attribute), 3  
`is_region_convex()` (`ground.base.Context` method),  
    11

## K

`Kind` (class in `ground.base`), 3

## L

`linear` (`ground.hints.Mix` property), 36  
`locate_point_in_point_point_point_circle`  
    (`ground.base.Context` property), 7  
`Location` (class in `ground.base`), 3

## M

`max_x` (`ground.hints.Box` property), 35  
`max_y` (`ground.hints.Box` property), 35  
`merged_box()` (`ground.base.Context` method), 11  
`min_x` (`ground.hints.Box` property), 35  
`min_y` (`ground.hints.Box` property), 35  
`Mix` (class in `ground.hints`), 36  
`mix_cls` (`ground.base.Context` property), 7  
`Mode` (class in `ground.base`), 4  
`mode` (`ground.base.Context` property), 7  
`module`  
    `ground.base`, 3  
    `ground.hints`, 35  
`Multipoint` (class in `ground.hints`), 36  
`multipoint_centroid()` (`ground.base.Context`  
    method), 11  
`multipoint_cls` (`ground.base.Context` property), 7  
`Multipolygon` (class in `ground.hints`), 36

`multipolygon_centroid()` (`ground.base.Context`  
    method), 12  
`multipolygon_cls` (`ground.base.Context` property), 7  
`Multisegment` (class in `ground.hints`), 37  
`multisegment_centroid()` (`ground.base.Context`  
    method), 12  
`multisegment_cls` (`ground.base.Context` property), 7  
`multisegment_length()` (`ground.base.Context`  
    method), 13

## O

`OBTUSE` (`ground.base.Kind` attribute), 3  
`Orientation` (class in `ground.base`), 3  
`origin` (`ground.base.Context` property), 7  
`OVERLAP` (`ground.base.Relation` attribute), 4

## P

`Point` (class in `ground.hints`), 37  
`point_cls` (`ground.base.Context` property), 7  
`points` (`ground.hints.Multipoint` property), 36  
`points_box()` (`ground.base.Context` method), 13  
`points_convex_hull()` (`ground.base.Context`  
    method), 13  
`points_squared_distance` (`ground.base.Context`  
    property), 8  
`Polygon` (class in `ground.hints`), 38  
`polygon_box()` (`ground.base.Context` method), 14  
`polygon_centroid()` (`ground.base.Context` method),  
    14  
`polygon_cls` (`ground.base.Context` property), 8  
`polygons` (`ground.hints.Multipolygon` property), 37  
`polygons_box()` (`ground.base.Context` method), 14

## R

`region_centroid()` (`ground.base.Context` method), 15  
`region_signed_area` (`ground.base.Context` property),  
    8  
`Relation` (class in `ground.base`), 3  
`replace()` (`ground.base.Context` method), 15  
`RIGHT` (`ground.base.Kind` attribute), 3  
`rotate_contour()` (`ground.base.Context` method), 15  
`rotate_contour_around_origin()`  
    (`ground.base.Context` method), 16  
`rotate_multipoint()` (`ground.base.Context` method),  
    16  
`rotate_multipoint_around_origin()`  
    (`ground.base.Context` method), 17  
`rotate_multipolygon()` (`ground.base.Context`  
    method), 17  
`rotate_multipolygon_around_origin()`  
    (`ground.base.Context` method), 18  
`rotate_multisegment()` (`ground.base.Context`  
    method), 18

`rotate_multisegment_around_origin()`  
     (*ground.base.Context method*), 19  
`rotate_point()` (*ground.base.Context method*), 19  
`rotate_point_around_origin()`  
     (*ground.base.Context method*), 20  
`rotate_polygon()` (*ground.base.Context method*), 20  
`rotate_polygon_around_origin()`  
     (*ground.base.Context method*), 20  
`rotate_segment()` (*ground.base.Context method*), 21  
`rotate_segment_around_origin()`  
     (*ground.base.Context method*), 21

## S

`scale_contour()` (*ground.base.Context method*), 22  
`scale_multipoint()` (*ground.base.Context method*), 22  
`scale_multipolygon()` (*ground.base.Context method*), 23  
`scale_multisegment()` (*ground.base.Context method*), 24  
`scale_point()` (*ground.base.Context method*), 25  
`scale_polygon()` (*ground.base.Context method*), 25  
`scale_segment()` (*ground.base.Context method*), 26  
`Segment` (*class in ground.hints*), 38  
`segment_box()` (*ground.base.Context method*), 26  
`segment_centroid()` (*ground.base.Context method*), 26  
`segment_cls` (*ground.base.Context property*), 9  
`segment_contains_point()` (*ground.base.Context method*), 27  
`segment_length()` (*ground.base.Context method*), 27  
`segment_point_squared_distance()`  
     (*ground.base.Context method*), 28  
`segments` (*ground.hints.Multisegment property*), 37  
`segments_box()` (*ground.base.Context method*), 28  
`segments_intersection()` (*ground.base.Context method*), 28  
`segments_relation()` (*ground.base.Context method*), 29  
`segments_squared_distance()` (*ground.base.Context method*), 29  
`set_context()` (*in module ground.base*), 34  
`shaped` (*ground.hints.Mix property*), 36  
`sqrt` (*ground.base.Context property*), 9  
`start` (*ground.hints.Segment property*), 38

## T

`TOUCH` (*ground.base.Relation attribute*), 3  
`translate_contour()` (*ground.base.Context method*), 30  
`translate_multipoint()` (*ground.base.Context method*), 30  
`translate_multipolygon()` (*ground.base.Context method*), 31

`translate_multisegment()` (*ground.base.Context method*), 32  
`translate_point()` (*ground.base.Context method*), 33  
`translate_polygon()` (*ground.base.Context method*), 33  
`translate_segment()` (*ground.base.Context method*), 34

## V

`vertices` (*ground.hints.Contour property*), 35

## W

`WITHIN` (*ground.base.Relation attribute*), 4

## X

`x` (*ground.hints.Point property*), 38

## Y

`y` (*ground.hints.Point property*), 38