gon

Release 6.0.0

Azat Ibrakov

Jun 27, 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

INTERFACES

class gon.base.**Geometry**(**args*, ***kwds*)

abstract distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between geometric objects.

abstract rotate(*angle: Angle[Scalar]*, *point: Optional[Point[Scalar]] = None*) → _T

Rotates geometric object by given angle around given point.

abstract scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → *Geometry*[Scalar]

Scales geometric object by given factor.

abstract translate(*step_x: Scalar*, *step_y: Scalar*) → _T

Translates geometric object by given step.

abstract validate() → None

Checks geometric object's constraints and raises error if any violation was found.

class gon.base.**Compound**(**args*, ***kwds*)

abstract __and__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the geometry with the other geometry.

abstract __contains__(*point: Point[Scalar]*) → bool

Checks if the geometry contains the point.

abstract __ge__(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is a superset of the other.

abstract __gt__(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is a strict superset of the other.

abstract __le__(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is a subset of the other.

abstract __lt__(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is a strict subset of the other.

abstract __or__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the geometry with the other geometry.

abstract __sub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the geometry with the other geometry.

abstract __xor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the geometry with the other geometry.

abstract property centroid: Point[Scalar]

Returns centroid of the geometry.

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

abstract locate(*point: Point[Scalar]*) → *Location*

Finds location of point relative to the geometry.

abstract relate(*other:* Compound*[Scalar]*) → *Relation*

Finds relation between geometric objects.

class gon.base.**Indexable**(**args*, ***kwds*)

abstract index() → None

Pre-processes geometry to potentially improve queries.

class gon.base.**Linear**(**args*, ***kwds*)

abstract property length: Scalar

Returns length of the geometry.

class gon.base.**Shaped**(**args*, ***kwds*)

abstract property area: Scalar

Returns area of the geometry.

abstract property perimeter: Scalar

Returns perimeter of the geometry.

CHAPTER

TWO

PRIMITIVE GEOMETRIES

class gon.base.**Point**(*x: Scalar*, *y: Scalar*)

__eq__(*other: Point[Scalar]*) → bool

Checks if the point is equal to the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> Point(0, 0) == Point(0, 0)
True
>>> Point(0, 0) == Point(0, 1)
False
>>> Point(0, 0) == Point(1, 1)
False
>>> Point(0, 0) == Point(1, 0)
False
```

__hash__() → int

Returns hash value of the point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> hash(Point(0, 0)) == hash(Point(0, 0))
True
```

__init__(*x: Scalar*, *y: Scalar*) → None

Initializes point.

Time complexity:
O(1)

Memory complexity:
O(1)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other: Point[Scalar]*) → bool

Checks if the point is less than or equal to the other. Compares points lexicographically, `x` coordinates first.

Time complexity:
`O(1)`

Memory complexity:
`O(1)`

Reference:
https://en.wikipedia.org/wiki/Lexicographical_order

```
>>> from gon.base import Point
>>> Point(0, 0) <= Point(0, 0)
True
>>> Point(0, 0) <= Point(0, 1)
True
>>> Point(0, 0) <= Point(1, 1)
True
>>> Point(0, 0) <= Point(1, 0)
True
```

__lt__(*other: Point[Scalar]*) → bool

Checks if the point is less than the other. Compares points lexicographically, `x` coordinates first.

Time complexity:
`O(1)`

Memory complexity:
`O(1)`

Reference:
https://en.wikipedia.org/wiki/Lexicographical_order

```
>>> from gon.base import Point
>>> Point(0, 0) < Point(0, 0)
False
>>> Point(0, 0) < Point(0, 1)
True
>>> Point(0, 0) < Point(1, 1)
True
>>> Point(0, 0) < Point(1, 0)
True
```

static __new__(*cls*, **args*, ***kwds*)

__repr__() → str

Return repr(self).

distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between the point and the other geometry.

Time complexity:
`O(1)`

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> point = Point(1, 0)
>>> point.distance_to(point) == 0
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Point[Scalar]

Rotates the point by given angle around given point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle, Point
>>> point = Point(1, 0)
>>> point.rotate(Angle(1, 0)) == point
True
>>> point.rotate(Angle(0, 1), Point(1, 1)) == Point(2, 1)
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → Point[Scalar]

Scales the point by given factor.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> point = Point(1, 0)
>>> point.scale(1) == point.scale(1, 2) == point
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Point[Scalar]

Translates the point by given step.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> Point(1, 0).translate(1, 2) == Point(2, 2)
True
```

validate() → None

Checks if coordinates are finite.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> Point(0, 0).validate()
```

property x: Scalar

Returns abscissa of the point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> Point(1, 0).x == 1
True
```

property y: Scalar

Returns ordinate of the point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point
>>> Point(1, 0).y == 0
True
```

CHAPTER

THREE

DEGENERATE GEOMETRIES

gon.base.**EMPTY = Empty()**

Empty geometry instance, equivalent of empty set.

class gon.base.**Empty**

__and__(*other:* Compound) → *Compound*

Returns intersection of the empty geometry with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY & EMPTY is EMPTY
True
```

__contains__(*point: Point*) → bool

Checks if the empty geometry contains the point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY, Point
>>> Point(0, 0) in EMPTY
False
```

__eq__(*other: Empty*) → bool

Checks if empty geometries are equal.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY == EMPTY
True
```

__ge__(*other:* Compound) → bool

Checks if the empty geometry is a superset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY >= EMPTY
True
```

__gt__(*other:* Compound) → bool

Checks if the empty geometry is a strict superset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY > EMPTY
False
```

__hash__() → int

Returns hash value of the empty geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> hash(EMPTY) == hash(EMPTY)
True
```

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other:* Compound) → bool

Checks if the empty geometry is a subset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY <= EMPTY
True
```

__lt__(*other:* Compound) → bool

Checks if the empty geometry is a strict subset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY < EMPTY
False
```

static __new__(*cls*) → Empty

Returns empty geometry instance.

Based on singleton pattern.

Time complexity:
O(1)

Memory complexity:
O(1)

Reference:
https://en.wikipedia.org/wiki/Singleton_pattern

__or__(*other:* Compound) → *Compound*

Returns union of the empty geometry with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY | EMPTY is EMPTY
True
```

__rand__(*other:* Compound) → *Compound*

Returns intersection of the empty geometry with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY & EMPTY is EMPTY
True
```

__repr__() → str

Return repr(self).

__ror__(*other:* Compound) → *Compound*

Returns union of the empty geometry with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY | EMPTY is EMPTY
True
```

__rsub__(*other:* Compound) → *Compound*

Returns difference of the other geometry with the empty geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

__rxor__(*other:* Compound) → *Compound*

Returns symmetric difference of the empty geometry with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY ^ EMPTY is EMPTY
True
```

__sub__(*other:* Compound) → *Compound*

Returns difference of the empty geometry with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY - EMPTY is EMPTY
True
```

__xor__(*other:* Compound) → *Compound*

Returns symmetric difference of the empty geometry with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY ^ EMPTY is EMPTY
True
```

property centroid: NoReturn

Returns centroid of the geometry.

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry) → NoReturn

Returns distance between geometric objects.

locate(*point: Point*) → *Location*

Finds location of the point relative to the empty geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY, Point
>>> EMPTY.locate(Point(0, 0)) is Location.EXTERIOR
True
```

relate(*other:* Compound) → *Relation*

Finds relation between the empty geometry and the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY.relate(EMPTY) is Relation.DISJOINT
True
```

rotate(*angle: Angle*, *point: Optional[Point] = None*) → Empty

Rotates the empty geometry by given angle around given point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY, Angle, Point
>>> (EMPTY.rotate(Angle(1, 0))
...  is EMPTY.rotate(Angle(0, 1), Point(1, 1))
...  is EMPTY)
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → Empty

Scales the empty geometry by given factor.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY.scale(1) is EMPTY.scale(1, 2) is EMPTY
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Empty

Translates the empty geometry by given step.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY
>>> EMPTY.translate(1, 2) is EMPTY
True
```

validate() → None

Checks if the empty geometry is valid.

Time complexity:
O(1)

Memory complexity:
O(1)

CHAPTER

FOUR

DISCRETE GEOMETRIES

class gon.base.**Multipoint**(*points: Sequence[Point[Scalar]]*)

__and__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the multipoint with the other geometry.

Time complexity:
`O(points_count)`

Memory complexity:
`O(points_count)`

where `points_count = len(self.points)`.

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint & multipoint == multipoint
True
```

__contains__(*point: Point[Scalar]*) → bool

Checks if the multipoint contains the point.

Time complexity:
`O(1)` expected, `O(len(self.points))` worst

Memory complexity:
`O(1)`

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> all(point in multipoint for point in multipoint.points)
True
```

__eq__(*other: Multipoint[Scalar]*) → bool

Checks if multipoints are equal.

Time complexity:
`O(min(len(self.points), len(other.points)))`

Memory complexity:
`O(1)`

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint == multipoint
```

(continues on next page)

(continued from previous page)

```
True
>>> multipoint == Multipoint([Point(0, 0), Point(1, 0), Point(1, 1),
...                           Point(0, 1)])
False
>>> multipoint == Multipoint([Point(1, 0), Point(0, 0), Point(0, 1)])
True
```

__ge__(*other:* Compound*[Scalar]*) → bool

Checks if the multipoint is a superset of the other geometry.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint >= multipoint
True
>>> multipoint >= Multipoint([Point(0, 0), Point(1, 0), Point(1, 1),
...                           Point(0, 1)])
False
>>> multipoint >= Multipoint([Point(1, 0), Point(0, 0), Point(0, 1)])
True
```

__gt__(*other:* Compound*[Scalar]*) → bool

Checks if the multipoint is a strict superset of the other geometry.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint > multipoint
False
>>> multipoint > Multipoint([Point(0, 0), Point(1, 0), Point(1, 1),
...                          Point(0, 1)])
False
>>> multipoint > Multipoint([Point(1, 0), Point(0, 0), Point(0, 1)])
False
```

__hash__() → int

Returns hash value of the multipoint.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> hash(multipoint) == hash(multipoint)
True
```

__init__(*points: Sequence[Point[Scalar]]*) → None

Initializes multipoint.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where `points_count = len(points)`.

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other:* Compound*[Scalar]*) → bool

Checks if the multipoint is a subset of the other geometry.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint <= multipoint
True
>>> multipoint <= Multipoint([Point(0, 0), Point(1, 0), Point(1, 1),
...                           Point(0, 1)])
True
>>> multipoint <= Multipoint([Point(1, 0), Point(0, 0), Point(0, 1)])
True
```

__lt__(*other:* Compound*[Scalar]*) → bool

Checks if the multipoint is a strict subset of the other geometry.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint < multipoint
False
>>> multipoint < Multipoint([Point(0, 0), Point(1, 0), Point(1, 1),
...                          Point(0, 1)])
True
>>> multipoint < Multipoint([Point(1, 0), Point(0, 0), Point(0, 1)])
False
```

static __new__(*cls*, **args*, ***kwds*)

__or__(*other:* Compound[*Scalar*]) → *Compound*[Scalar]

Returns union of the multipoint with the other geometry.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where points_count = len(self.points).

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint | multipoint == multipoint
True
```

__rand__(*other:* Compound[*Scalar*]) → *Compound*[Scalar]

Returns intersection of the multipoint with the other geometry.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where points_count = len(self.points).

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint & multipoint == multipoint
True
```

__repr__() → str

Return repr(self).

__sub__(*other:* Compound[*Scalar*]) → *Compound*[Scalar]

Returns difference of the multipoint with the other geometry.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where points_count = len(self.points).

```
>>> from gon.base import EMPTY, Multipoint, Point
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint - multipoint is EMPTY
True
```

__xor__(*other:* Compound[*Scalar*]) → *Compound*[Scalar]

Returns symmetric difference of the multipoint with the other geometry.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where points_count = len(self.points).

```
>>> from gon.base import EMPTY, Multipoint, Point
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint ^ multipoint is EMPTY
True
```

property centroid: Point[Scalar]

Returns centroid of the multipoint.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where points_count = len(self.points).

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(3, 0), Point(0, 3)])
>>> multipoint.centroid == Point(1, 1)
True
```

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between the multipoint and the other geometry.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint.distance_to(multipoint) == 0
True
```

index() → None

Pre-processes the multipoint to potentially improve queries.

Time complexity:
O(points_count * log points_count)

Memory complexity:
O(points_count)

where points_count = len(self.points).

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint.index()
```

locate(*point: Point[Scalar]*) → *Location*

Finds location of the point relative to the multipoint.

Time complexity:
`O(1)` expected, `O(len(self.points))` worst

Memory complexity:
`O(1)`

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> all(multipoint.locate(point) is Location.BOUNDARY
...     for point in multipoint.points)
True
```

property points: Sequence[Point[Scalar]]

Returns points of the multipoint.

Time complexity:
`O(points_count)`

Memory complexity:
`O(points_count)`

where `points_count = len(self.points)`.

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint.points == [Point(0, 0), Point(1, 0), Point(0, 1)]
True
```

relate(*other:* Compound*[Scalar]*) → *Relation*

Finds relation between the multipoint and the other geometry.

Time complexity:
`O(points_count)`

Memory complexity:
`O(points_count)`

where `points_count = len(self.points)`.

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint.relate(multipoint) is Relation.EQUAL
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Multipoint[Scalar]

Rotates geometric object by given angle around given point.

Time complexity:
`O(points_count)`

Memory complexity:
`O(points_count)`

where `points_count = len(self.points)`.

```
>>> from gon.base import Angle, Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint.rotate(Angle(1, 0)) == multipoint
True
>>> (multipoint.rotate(Angle(0, 1), Point(1, 1))
...  == Multipoint([Point(2, 0), Point(2, 1), Point(1, 0)]))
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → Multipoint[Scalar]

Scales the multipoint by given factor.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where `points_count = len(self.points)`.

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint.scale(1) == multipoint
True
>>> (multipoint.scale(1, 2)
...  == Multipoint([Point(0, 0), Point(1, 0), Point(0, 2)]))
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Multipoint[Scalar]

Translates the multipoint by given step.

Time complexity:
O(points_count)

Memory complexity:
O(points_count)

where `points_count = len(self.points)`.

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> (multipoint.translate(1, 2)
...  == Multipoint([Point(1, 2), Point(2, 2), Point(1, 3)]))
True
```

validate() → None

Checks if the multipoint is valid.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multipoint, Point
>>> multipoint = Multipoint([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> multipoint.validate()
```

CHAPTER

FIVE

LINEAR GEOMETRIES

class gon.base.**Segment**(*start: Point[Scalar]*, *end: Point[Scalar]*)

__and__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the segment with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment & segment == segment
True
```

__contains__(*point: Point[Scalar]*) → bool

Checks if the segment contains the point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.start in segment
True
>>> segment.end in segment
True
```

__eq__(*other: Segment[Scalar]*) → bool

Checks if the segment is equal to the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
```

(continues on next page)

(continued from previous page)

```
>>> segment == segment
True
>>> segment == Segment(Point(2, 0), Point(0, 0))
True
>>> segment == Segment(Point(0, 0), Point(1, 0))
False
>>> segment == Segment(Point(0, 0), Point(0, 2))
False
```

__ge__(*other:* Compound[*Scalar*]) → bool

Checks if the segment is a superset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment >= segment
True
>>> segment >= Segment(Point(2, 0), Point(0, 0))
True
>>> segment >= Segment(Point(0, 0), Point(1, 0))
True
>>> segment >= Segment(Point(0, 0), Point(0, 2))
False
```

__gt__(*other:* Compound[*Scalar*]) → bool

Checks if the segment is a strict superset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment > segment
False
>>> segment > Segment(Point(2, 0), Point(0, 0))
False
>>> segment > Segment(Point(0, 0), Point(1, 0))
True
>>> segment > Segment(Point(0, 0), Point(0, 2))
False
```

__hash__() → int

Returns hash value of the segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> hash(segment) == hash(segment)
True
>>> hash(segment) == hash(Segment(Point(2, 0), Point(0, 0)))
True
```

__init__(*start: Point[Scalar]*, *end: Point[Scalar]*) → None

Initializes segment.

Time complexity:
O(1)

Memory complexity:
O(1)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other:* Compound*[Scalar]*) → bool

Checks if the segment is a subset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment <= segment
True
>>> segment <= Segment(Point(2, 0), Point(0, 0))
True
>>> segment <= Segment(Point(0, 0), Point(1, 0))
False
>>> segment <= Segment(Point(0, 0), Point(0, 2))
False
```

__lt__(*other:* Compound*[Scalar]*) → bool

Checks if the segment is a strict subset of the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment < segment
False
```

(continues on next page)

(continued from previous page)

```
>>> segment < Segment(Point(2, 0), Point(0, 0))
False
>>> segment < Segment(Point(0, 0), Point(1, 0))
False
>>> segment < Segment(Point(0, 0), Point(0, 2))
False
```

static __new__(*cls*, **args*, ***kwds*)

__or__(*other:* Compound) → *Compound*

Returns union of the segment with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment | segment == segment
True
```

__rand__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the segment with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment & segment == segment
True
```

__repr__() → str

Return repr(self).

__ror__(*other:* Compound) → *Compound*

Returns union of the segment with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment | segment == segment
True
```

__rxor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the segment with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY, Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment ^ segment is EMPTY
True
```

__sub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the segment with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY, Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment - segment is EMPTY
True
```

__xor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the segment with the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import EMPTY, Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment ^ segment is EMPTY
True
```

property centroid: Point[Scalar]

Returns centroid of the segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.centroid == Point(1, 0)
True
```

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between the segment and the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.distance_to(segment) == 0
True
```

property end: Point[Scalar]

Returns end of the segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.end == Point(2, 0)
True
```

property is_horizontal: bool

Checks if the segment is horizontal.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.is_horizontal
True
```

property is_vertical: bool

Checks if the segment is vertical.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.is_vertical
False
```

property length: Scalar

Returns length of the segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.length == 2
True
```

locate(*point: Point[Scalar]*) → *Location*

Finds location of the point relative to the segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.locate(segment.start) is Location.BOUNDARY
True
>>> segment.locate(segment.end) is Location.BOUNDARY
True
```

relate(*other:* Compound*[Scalar]*) → *Relation*

Finds relation between the segment and the other geometry.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.relate(segment) is Relation.EQUAL
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Segment[Scalar]

Rotates the segment by given angle around given point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle, Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.rotate(Angle(1, 0)) == segment
True
```

(continues on next page)

(continued from previous page)

```
>>> (segment.rotate(Angle(0, 1), Point(1, 1))
...  == Segment(Point(2, 0), Point(2, 2)))
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → *Compound*[Scalar]

Scales the segment by given factor.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.scale(1) == segment.scale(1, 2) == segment
True
```

property start: Point[Scalar]

Returns start of the segment.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.start == Point(0, 0)
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Segment[Scalar]

Translates the segment by given step.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.translate(1, 2) == Segment(Point(1, 2), Point(3, 2))
True
```

validate() → None

Checks if endpoints are valid and unequal.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Segment
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> segment.validate()
```

class gon.base.**Multisegment**(*segments: Sequence[Segment]*)

__and__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the multisegment with the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment & multisegment == multisegment
True
```

__contains__(*point: Point[Scalar]*) → bool

Checks if the multisegment contains the point.

Time complexity:
`O(log segments_count)` expected after indexing, `O(segments_count)` worst after indexing or without it

Memory complexity:
`O(1)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> all(segment.start in multisegment and segment.end in multisegment
...     for segment in multisegment.segments)
True
```

__eq__(*other: Multisegment[Scalar]*) → bool

Checks if multisegments are equal.

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

Memory complexity:
`O(1)`

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment == multisegment
True
>>> multisegment == Multisegment([Segment(Point(0, 0), Point(1, 0)),
```

(continues on next page)

(continued from previous page)

```
...                                 Segment(Point(0, 1), Point(1, 1)),
...                                 Segment(Point(0, 0), Point(1, 1))])
False
>>> multisegment == Multisegment([Segment(Point(0, 1), Point(1, 1)),
...                                 Segment(Point(0, 0), Point(1, 0))])
True
```

__ge__(*other:* Compound*[Scalar]*) → bool

Checks if the multisegment is a superset of the other geometry.

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

Memory complexity:
    `O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment >= multisegment
True
>>> multisegment >= Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                               Segment(Point(0, 1), Point(1, 1)),
...                               Segment(Point(0, 0), Point(1, 1))])
False
>>> multisegment >= Multisegment([Segment(Point(0, 1), Point(1, 1)),
...                               Segment(Point(0, 0), Point(1, 0))])
True
```

__gt__(*other:* Compound*[Scalar]*) → bool

Checks if the multisegment is a strict superset of the other geometry.

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

Memory complexity:
    `O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment > multisegment
False
>>> multisegment > Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1)),
...                              Segment(Point(0, 0), Point(1, 1))])
False
>>> multisegment > Multisegment([Segment(Point(0, 1), Point(1, 1)),
...                              Segment(Point(0, 0), Point(1, 0))])
False
```

__hash__() → int

Returns hash value of the multisegment.

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

Memory complexity:
O(1)

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> hash(multisegment) == hash(multisegment)
True
```

__init__(*segments: Sequence[Segment]*) → None

Initializes multisegment.

Time complexity:
O(segments_count)

Memory complexity:
O(segments_count)

where segments_count = len(segments).

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other:* Compound*[Scalar]*) → bool

Checks if the multisegment is a subset of the other geometry.

Time complexity:
O(segments_count * log segments_count)

Memory complexity:
O(segments_count)

where segments_count = len(self.segments).

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment <= multisegment
True
>>> multisegment <= Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                               Segment(Point(0, 1), Point(1, 1)),
...                               Segment(Point(0, 0), Point(1, 1))])
True
>>> multisegment <= Multisegment([Segment(Point(0, 1), Point(1, 1)),
...                               Segment(Point(0, 0), Point(1, 0))])
True
```

__lt__(*other:* Compound*[Scalar]*) → bool

Checks if the multisegment is a strict subset of the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment < multisegment
False
>>> multisegment < Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1)),
...                              Segment(Point(0, 0), Point(1, 1))])
True
>>> multisegment < Multisegment([Segment(Point(0, 1), Point(1, 1)),
...                              Segment(Point(0, 0), Point(1, 0))])
False
```

static __new__(*cls*, **args*, ***kwds*)

__or__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the multisegment with the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment | multisegment == multisegment
True
```

__rand__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the multisegment with the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment & multisegment == multisegment
True
```

__repr__() → str

Return repr(self).

__ror__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the multisegment with the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment | multisegment == multisegment
True
```

__rsub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the other geometry with the multisegment.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

__rxor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the multisegment with the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import EMPTY, Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment ^ multisegment is EMPTY
True
```

__sub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the multisegment with the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import EMPTY, Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment - multisegment is EMPTY
True
```

__xor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the multisegment with the other geometry.

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

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import EMPTY, Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment ^ multisegment is EMPTY
True
```

property centroid: Point[Scalar]

Returns centroid of the multisegment.

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

Memory complexity:
`O(1)`

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(2, 0)),
...                              Segment(Point(0, 2), Point(2, 2))])
>>> multisegment.centroid == Point(1, 1)
True
```

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between the multisegment and the other geometry.

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

Memory complexity:
`O(1)`

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.distance_to(multisegment) == 0
True
```

index() → None

Pre-processes the multisegment to potentially improve queries.

Time complexity:
`O(segments_count * log segments_count)` expected, `O(segments_count ** 2)` worst

Memory complexity:
`O(segments_count)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.index()
```

property length: Scalar

Returns length of the multisegment.

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

Memory complexity:
`O(1)`

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.length == 2
True
```

locate(*point: Point[Scalar]*) → *Location*

Finds location of the point relative to the multisegment.

Time complexity:
`O(log segments_count)` expected after indexing, `O(segments_count)` worst after indexing or without it

Memory complexity:
`O(1)`

where `segments_count = len(self.segments)`.

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> all(multisegment.locate(segment.start)
...     is multisegment.locate(segment.end)
...     is Location.BOUNDARY
...     for segment in multisegment.segments)
True
```

relate(*other:* Compound*[Scalar]*) → *Relation*

Finds relation between the multisegment and the other geometry.

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

Memory complexity:
O(segments_count)

where segments_count = len(self.segments).

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.relate(multisegment) is Relation.EQUAL
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Multisegment[Scalar]

Rotates the multisegment by given angle around given point.

Time complexity:
O(segments_count)

Memory complexity:
O(segments_count)

where segments_count = len(self.segments).

```
>>> from gon.base import Angle, Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.rotate(Angle(1, 0)) == multisegment
True
>>> (multisegment.rotate(Angle(0, 1), Point(1, 1))
...  == Multisegment([Segment(Point(2, 0), Point(2, 1)),
...                   Segment(Point(1, 0), Point(1, 1))]))
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → *Compound*[Scalar]

Scales the multisegment by given factor.

Time complexity:
O(segments_count)

Memory complexity:
O(segments_count)

where segments_count = len(self.segments).

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.scale(1) == multisegment
True
>>> (multisegment.scale(1, 2)
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(0, 2), Point(1, 2))]))
True
```

property segments: Sequence[Segment[Scalar]]

Returns segments of the multisegment.

Time complexity:
O(segments_count)

Memory complexity:
O(segments_count)

where segments_count = len(self.segments).

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.segments == [Segment(Point(0, 0), Point(1, 0)),
...                           Segment(Point(0, 1), Point(1, 1))]
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Multisegment[Scalar]

Translates the multisegment by given step.

Time complexity:
O(segments_count)

Memory complexity:
O(segments_count)

where segments_count = len(self.segments).

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> (multisegment.translate(1, 2)
...  == Multisegment([Segment(Point(1, 2), Point(2, 2)),
...                   Segment(Point(1, 3), Point(2, 3))]))
True
```

validate() → None

Checks if the multisegment is valid.

Time complexity:
O(segments_count * log segments_count)

Memory complexity:
O(segments_count)

where segments_count = len(self.segments).

```
>>> from gon.base import Multisegment, Point, Segment
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(0, 1), Point(1, 1))])
>>> multisegment.validate()
```

class gon.base.**Contour**(*vertices: Sequence[Point[Scalar]]*)

__and__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the contour with the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Multisegment, Point, Segment
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> (contour & contour
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(1, 0), Point(0, 1)),
...                   Segment(Point(0, 1), Point(0, 0))]))
True
```

__contains__(*point: Point[Scalar]*) → bool

Checks if the contour contains the point.

Time complexity:
`O(log vertices_count)` expected after indexing, `O(vertices_count)` worst after indexing or without it

Memory complexity:
`O(1)`

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> all(vertex in contour for vertex in contour.vertices)
True
```

__eq__(*other: Contour[Scalar]*) → bool

Checks if contours are equal.

Time complexity:
`O(min(len(self.vertices), len(other.vertices)))`

Memory complexity:
`O(1)`

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour == contour
True
>>> contour == Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                     Point(0, 1)])
False
>>> contour == Contour([Point(1, 0), Point(0, 0), Point(0, 1)])
True
```

__ge__(*other:* Compound*[Scalar]*) → bool

Checks if the contour is a superset of the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour >= contour
True
>>> contour >= Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                     Point(0, 1)])
False
>>> contour >= Contour([Point(1, 0), Point(0, 0), Point(0, 1)])
True
```

__gt__(*other:* Compound*[Scalar]*) → bool

Checks if the contour is a strict superset of the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour > contour
False
>>> contour > Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                    Point(0, 1)])
False
>>> contour > Contour([Point(1, 0), Point(0, 0), Point(0, 1)])
False
```

__hash__() → int

Returns hash value of the contour.

Time complexity:
`O(vertices_count)`

Memory complexity:
`O(1)` if contour is counterclockwise and starts from the bottom leftmost vertex, `O(vertices_count)` otherwise

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> hash(contour) == hash(contour)
True
```

__init__(*vertices: Sequence[Point[Scalar]]*) → None

Initializes contour.

Time complexity:
`O(vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(vertices)`.

classmethod **__init_subclass__**(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other:* Compound*[Scalar]*) → bool

Checks if the contour is a subset of the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour <= contour
True
>>> contour <= Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                     Point(0, 1)])
False
>>> contour <= Contour([Point(1, 0), Point(0, 0), Point(0, 1)])
True
```

__lt__(*other:* Compound*[Scalar]*) → bool

Checks if the contour is a strict subset of the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour < contour
False
>>> contour < Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                    Point(0, 1)])
False
>>> contour < Contour([Point(1, 0), Point(0, 0), Point(0, 1)])
False
```

static **__new__**(*cls*, **args*, ***kwds*)

__or__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the contour with the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import Contour, Multisegment, Point, Segment
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> (contour | contour
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(1, 0), Point(0, 1)),
...                   Segment(Point(0, 1), Point(0, 0))]))
True
```

__rand__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the contour with the other geometry.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(vertices_count)
```

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

```
>>> from gon.base import Contour, Multisegment, Point, Segment
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> (contour & contour
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(1, 0), Point(0, 1)),
...                   Segment(Point(0, 1), Point(0, 0))]))
True
```

__repr__() → str

Return repr(self).

__ror__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the contour with the other geometry.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(vertices_count)
```

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

```
>>> from gon.base import Contour, Multisegment, Point, Segment
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> (contour | contour
...  == Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                   Segment(Point(1, 0), Point(0, 1)),
...                   Segment(Point(0, 1), Point(0, 0))]))
True
```

__rsub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the other geometry with the contour.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(vertices_count)
```

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

__rxor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the contour with the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import EMPTY, Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour ^ contour is EMPTY
True
```

__sub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the contour with the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import EMPTY, Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour - contour is EMPTY
True
```

__xor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the contour with the other geometry.

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

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import EMPTY, Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour ^ contour is EMPTY
True
```

property centroid: Point[Scalar]

Returns centroid of the contour.

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

Memory complexity:
`O(1)`

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(2, 0), Point(2, 2),
...                    Point(0, 2)])
>>> contour.centroid == Point(1, 1)
True
```

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between the contour and the other geometry.

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

Memory complexity:
`O(1)`

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.distance_to(contour) == 0
True
```

index() → None

Pre-processes the contour to potentially improve queries.

Time complexity:
`O(vertices_count * log vertices_count)` expected, `O(vertices_count ** 2)` worst

Memory complexity:
`O(vertices_count)`

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.index()
```

property length: Scalar

Returns length of the contour.

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

Memory complexity:
`O(1)`

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(1, 1),
...                    Point(0, 1)])
>>> contour.length == 4
True
```

locate(*point: Point[Scalar]*) → *Location*

Finds location of the point relative to the contour.

Time complexity:
O(log vertices_count) expected after indexing, O(vertices_count) worst after indexing or without it

Memory complexity:
O(1)

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Location, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> all(contour.locate(vertex) is Location.BOUNDARY
...     for vertex in contour.vertices)
True
```

property orientation: *Orientation*

Returns orientation of the contour.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Contour, Orientation, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.orientation is Orientation.COUNTERCLOCKWISE
True
```

relate(*other:* Compound*[Scalar]*) → *Relation*

Finds relation between the contour and the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Point, Relation
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.relate(contour) is Relation.EQUAL
True
```

reverse() → Contour[Scalar]

Returns the reversed contour.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
```

(continues on next page)

(continued from previous page)

```
>>> contour.reverse().reverse() == contour
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Contour[Scalar]

Rotates the contour by given angle around given point.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

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

```
>>> from gon.base import Angle, Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.rotate(Angle(1, 0)) == contour
True
>>> (contour.rotate(Angle(0, 1), Point(1, 1))
...  == Contour([Point(2, 0), Point(2, 1), Point(1, 0)]))
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → *Compound*[Scalar]

Scales the contour by given factor.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

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

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.scale(1) == contour
True
>>> (contour.scale(1, 2)
...  == Contour([Point(0, 0), Point(1, 0), Point(0, 2)]))
True
```

property segments: Sequence[Segment[Scalar]]

Returns segments of the contour.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

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

```
>>> from gon.base import Contour, Point, Segment
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.segments == [Segment(Point(0, 1), Point(0, 0)),
...                      Segment(Point(0, 0), Point(1, 0)),
```

(continues on next page)

(continued from previous page)

```
...                         Segment(Point(1, 0), Point(0, 1))]
True
```

to_clockwise() → Contour[Scalar]

Returns the clockwise contour.

Time complexity:
O(1) if clockwise already, O(vertices_count) – otherwise

Memory complexity:
O(1) if clockwise already, O(vertices_count) – otherwise

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Orientation, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.to_clockwise().orientation is Orientation.CLOCKWISE
True
```

to_counterclockwise() → Contour[Scalar]

Returns the counterclockwise contour.

Time complexity:
O(1) if counterclockwise already, O(vertices_count) – otherwise

Memory complexity:
O(1) if counterclockwise already, O(vertices_count) – otherwise

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Orientation, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> (contour.to_counterclockwise().orientation
...  is Orientation.COUNTERCLOCKWISE)
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Contour[Scalar]

Translates the contour by given step.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> (contour.translate(1, 2)
...  == Contour([Point(1, 2), Point(2, 2), Point(1, 3)]))
True
```

validate() → None

Checks if the contour is valid.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.validate()
```

property vertices: Sequence[Point[Scalar]]

Returns vertices of the contour.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where vertices_count = len(self.vertices).

```
>>> from gon.base import Contour, Point
>>> contour = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> contour.vertices == [Point(0, 0), Point(1, 0), Point(0, 1)]
True
```

CHAPTER

SIX

SHAPED GEOMETRIES

class gon.base.**Polygon**(*border: Contour[Scalar]*, *holes: Optional[Sequence[Contour[Scalar]]] = None*)

__and__(*other:* Compound) → *Compound*

Returns intersection of the polygon with the other geometry.

Time complexity:

`O(vertices_count * log vertices_count)`

Memory complexity:

`O(vertices_count)`

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon & polygon == polygon
True
```

__contains__(*point: Point*) → bool

Checks if the polygon contains the point.

Time complexity:

`O(log vertices_count)` expected after indexing, `O(vertices_count)` worst after indexing or without it

Memory complexity:

`O(1)`

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
```

(continues on next page)

(continued from previous page)

```
...                                Point(4, 2)])])
>>> Point(0, 0) in polygon
True
>>> Point(1, 1) in polygon
True
>>> Point(2, 2) in polygon
True
>>> Point(3, 3) in polygon
False
>>> Point(4, 3) in polygon
True
>>> Point(5, 2) in polygon
True
>>> Point(6, 1) in polygon
True
>>> Point(7, 0) in polygon
False
```

__eq__(*other: Polygon*) → bool

Checks if polygons are equal.

Time complexity:

```
O(vertices_count)
```

Memory complexity:

```
O(1)
```

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])])
>>> polygon == polygon
True
```

__ge__(*other:* Compound) → bool

Checks if the polygon is a superset of the other geometry.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(1)
```

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon >= polygon
True
```

__gt__(*other:* Compound) → bool

Checks if the polygon is a strict superset of the other geometry.

Time complexity:

O(vertices_count * log vertices_count)

Memory complexity:

O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon > polygon
False
```

__hash__() → int

Returns hash value of the polygon.

Time complexity:

O(vertices_count)

Memory complexity:

O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> hash(polygon) == hash(polygon)
True
```

__init__(*border: Contour[Scalar]*, *holes: Optional[Sequence[Contour[Scalar]]] = None*) → None

Initializes polygon.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where

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

classmethod **__init_subclass__**(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other:* Compound) → bool

Checks if the polygon is a subset of the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon <= polygon
True
```

__lt__(*other:* Compound) → bool

Checks if the polygon is a strict subset of the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
```

(continues on next page)

(continued from previous page)

```
>>> polygon < polygon
False
```

static __new__(*cls*, **args*, ***kwds*)

__or__(*other:* Compound) → *Compound*

Returns union of the polygon with the other geometry.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(vertices_count)
```

where

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

```
>>> from gon.base import Multipolygon
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon | polygon == polygon
True
```

__rand__(*other:* Compound) → *Compound*

Returns intersection of the polygon with the other geometry.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(vertices_count)
```

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon & polygon == polygon
True
```

__repr__() → str

Return repr(self).

__ror__(*other:* Compound) → *Compound*

Returns union of the polygon with the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import Multipolygon
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon | polygon == polygon
True
```

__rsub__(*other:* Compound) → *Compound*

Returns difference of the other geometry with the polygon.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where

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

__rxor__(*other:* Compound) → *Compound*

Returns symmetric difference of the polygon with the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import EMPTY, Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon ^ polygon is EMPTY
True
```

__sub__(*other:* Compound) → *Compound*

Returns difference of the polygon with the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import EMPTY, Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])])
>>> polygon - polygon is EMPTY
True
```

__xor__(*other:* Compound) → *Compound*

Returns symmetric difference of the polygon with the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import EMPTY, Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])])
>>> polygon ^ polygon is EMPTY
True
```

property area: Scalar

Returns area of the polygon.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.area == 32
True
```

property border: Contour

Returns border of the polygon.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.border == Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)])
True
```

property centroid: Point

Returns centroid of the polygon.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.centroid == Point(3, 3)
True
```

property convex_hull: Polygon

Returns convex hull of the polygon.

Time complexity:
O(border_vertices_count) if convex already, O(border_vertices_count * log border_vertices_count) – otherwise

Memory complexity:
O(1) if convex already, O(border_vertices_count) – otherwise

where `border_vertices_count = len(self.border.vertices)`.

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.convex_hull == Polygon(polygon.border, [])
True
```

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry) → Scalar

Returns distance between the polygon and the other geometry.

Time complexity:
`O(vertices_count)`

Memory complexity:
`O(1)`

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.distance_to(polygon) == 0
True
```

property edges: Sequence[Segment]

Returns edges of the polygon.

Time complexity:
`O(vertices_count)`

Memory complexity:
`O(vertices_count)`

where

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

```
>>> from gon.base import Contour, Point, Polygon, Segment
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.edges == [Segment(Point(0, 6), Point(0, 0)),
...                   Segment(Point(0, 0), Point(6, 0)),
```

(continues on next page)

(continued from previous page)

```
...                     Segment(Point(6, 0), Point(6, 6)),
...                     Segment(Point(6, 6), Point(0, 6)),
...                     Segment(Point(4, 2), Point(2, 2)),
...                     Segment(Point(2, 2), Point(2, 4)),
...                     Segment(Point(2, 4), Point(4, 4)),
...                     Segment(Point(4, 4), Point(4, 2))]
True
```

property holes: Sequence[Contour]

Returns holes of the polygon.

Time complexity:
`O(holes_count)`

Memory complexity:
`O(holes_count)`

where `holes_count = len(self.holes)`.

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.holes == [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                            Point(4, 2)])]
True
```

index() → None

Pre-processes the polygon to potentially improve queries.

Time complexity:
`O(vertices_count * log vertices_count)` expected, `O(vertices_count ** 2)` worst

Memory complexity:
`O(vertices_count)`

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.index()
```

property is_convex: bool

Checks if the polygon is convex.

Time complexity:
`O(len(self.border.vertices))`

Memory complexity:
O(1)

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.is_convex
False
>>> polygon.convex_hull.is_convex
True
```

locate(*point: Point*) → *Location*

Finds location of the point relative to the polygon.

Time complexity:
O(log vertices_count) expected after indexing, O(vertices_count) worst after indexing or without it

Memory complexity:
O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.locate(Point(0, 0)) is Location.BOUNDARY
True
>>> polygon.locate(Point(1, 1)) is Location.INTERIOR
True
>>> polygon.locate(Point(2, 2)) is Location.BOUNDARY
True
>>> polygon.locate(Point(3, 3)) is Location.EXTERIOR
True
>>> polygon.locate(Point(4, 3)) is Location.BOUNDARY
True
>>> polygon.locate(Point(5, 2)) is Location.INTERIOR
True
>>> polygon.locate(Point(6, 1)) is Location.BOUNDARY
True
>>> polygon.locate(Point(7, 0)) is Location.EXTERIOR
True
```

property perimeter: Scalar

Returns perimeter of the polygon.

Time complexity:
O(vertices_count)

Memory complexity:
O(1)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.perimeter == 32
True
```

relate(*other:* Compound) → *Relation*

Finds relation between the polygon and the other geometry.

Time complexity:
O(vertices_count * log vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.relate(polygon) is Relation.EQUAL
True
```

rotate(*angle: Angle*, *point: Optional[Point] = None*) → Polygon

Rotates the polygon by given angle around given point.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import Angle, Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
```

(continues on next page)

(continued from previous page)

```
...                                Point(4, 2)])])
>>> polygon.rotate(Angle(1, 0)) == polygon
True
>>> (polygon.rotate(Angle(0, 1), Point(1, 1))
...  == Polygon(Contour([Point(2, 0), Point(2, 6), Point(-4, 6),
...                      Point(-4, 0)]),
...             [Contour([Point(0, 2), Point(-2, 2), Point(-2, 4),
...                       Point(0, 4)])]))
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → Polygon

Scales the polygon by given factor.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])])
>>> polygon.scale(1) == polygon
True
>>> (polygon.scale(1, 2)
...  == Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 12),
...                      Point(0, 12)]),
...             [Contour([Point(2, 4), Point(2, 8), Point(4, 8),
...                       Point(4, 4)])]))
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Polygon[Scalar]

Translates the polygon by given step.

Time complexity:
O(vertices_count)

Memory complexity:
O(vertices_count)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
```

(continues on next page)

(continued from previous page)

```
...                               Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])])
>>> (polygon.translate(1, 2)
...  == Polygon(Contour([Point(1, 2), Point(7, 2), Point(7, 8),
...                      Point(1, 8)]),
...             [Contour([Point(3, 4), Point(3, 6), Point(5, 6),
...                       Point(5, 4)])]))
True
```

triangulate() → *Triangulation*

Returns triangulation of the polygon.

Time complexity:

O(vertices_count ** 2)

Memory complexity:

O(vertices_count)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])])
>>> triangulation = polygon.triangulate()
>>> (triangulation.triangles()
...  == [Contour([Point(4, 4), Point(6, 0), Point(6, 6)]),
...      Contour([Point(4, 2), Point(6, 0), Point(4, 4)]),
...      Contour([Point(0, 6), Point(4, 4), Point(6, 6)]),
...      Contour([Point(0, 0), Point(2, 2), Point(0, 6)]),
...      Contour([Point(0, 0), Point(6, 0), Point(4, 2)]),
...      Contour([Point(0, 6), Point(2, 4), Point(4, 4)]),
...      Contour([Point(0, 6), Point(2, 2), Point(2, 4)]),
...      Contour([Point(0, 0), Point(4, 2), Point(2, 2)])])
True
```

validate() → None

Checks if the polygon is valid.

Time complexity:

O(vertices_count * log (vertices_count))

Memory complexity:

O(vertices_count)

where

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

```
>>> from gon.base import Contour, Point, Polygon
>>> polygon = Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])])
>>> polygon.validate()
```

class gon.base.**Multipolygon**(*polygons: Sequence[Polygon[Scalar]]*)

__and__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the multipolygon with the other geometry.

Time complexity:

`O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon & multipolygon == multipolygon
True
```

__contains__(*point: Point[Scalar]*) → bool

Checks if the multipolygon contains the point.

Time complexity:

`O(log vertices_count)` expected after indexing, `O(vertices_count)` worst after indexing or without it

Memory complexity:

`O(1)`

where

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

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> Point(0, 0) in multipolygon
True
>>> Point(1, 1) in multipolygon
True
>>> Point(2, 2) in multipolygon
True
>>> Point(3, 3) in multipolygon
False
>>> Point(4, 5) in multipolygon
True
>>> Point(5, 6) in multipolygon
True
>>> Point(6, 7) in multipolygon
True
>>> Point(7, 7) in multipolygon
False
```

__eq__(*other: Multipolygon[Scalar]*) → bool

Checks if multipolygons are equal.

Time complexity:

```
O(len(self.polygons))
```

Memory complexity:

```
O(1)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon == multipolygon
True
```

__ge__(*other:* Compound*[Scalar]*) → bool

Checks if the multipolygon is a superset of the other geometry.

Time complexity:

```
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon >= multipolygon
True
```

__gt__(*other:* Compound*[Scalar]*) → bool

Checks if the multipolygon is a strict superset of the other geometry.

Time complexity:
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon > multipolygon
False
```

__hash__() → int

Returns hash value of the polygon.

Time complexity:
O(len(self.polygons))

Memory complexity:
O(1)

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> hash(multipolygon) == hash(multipolygon)
True
```

__init__(*polygons: Sequence[Polygon[Scalar]]*) → None

Initializes multipolygon.

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 self.polygons)
```

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other:* Compound*[Scalar]*) → bool

Checks if the multipolygon is a subset of the other geometry.

Time complexity:
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon <= multipolygon
True
```

__lt__(*other:* Compound*[Scalar]*) → bool

Checks if the multipolygon is a strict subset of the other geometry.

Time complexity:

```
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon < multipolygon
False
```

static __new__(*cls*, **args*, ***kwds*)

__or__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the multipolygon with the other geometry.

Time complexity:

```
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon | multipolygon == multipolygon
True
```

__rand__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the multipolygon with the other geometry.

Time complexity:

```
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon & multipolygon == multipolygon
True
```

__repr__() → str

Return repr(self).

__ror__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the multipolygon with the other geometry.

Time complexity:

```
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon | multipolygon == multipolygon
True
```

__rsub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the other geometry with the multipolygon.

Time complexity:
O(vertices_count * log 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 self.polygons)
```

__rxor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the multipolygon with the other geometry.

Time complexity:
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import EMPTY, Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon ^ multipolygon is EMPTY
True
```

__sub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the multipolygon with the other geometry.

Time complexity:

```
O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import EMPTY, Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon - multipolygon is EMPTY
True
```

__xor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the multipolygon with the other geometry.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(vertices_count)
```

where

```
vertices_count = sum(len(polygon.border.vertices)
                     + sum(len(hole.vertices)
```

(continues on next page)

(continued from previous page)

```
                                for hole in polygon.holes)
                            for polygon in self.polygons)
```

```
>>> from gon.base import EMPTY, Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon ^ multipolygon is EMPTY
True
```

property area: Scalar

Returns area of the multipolygon.

Time complexity:
O(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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.area == 128
True
```

property centroid: Point[Scalar]

Returns centroid of the multipolygon.

Time complexity:
O(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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.centroid == Point(7, 7)
True
```

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between the multipolygon and the other geometry.

Time complexity:
`O(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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.distance_to(multipolygon) == 0
True
```

index() → None

Pre-processes the multipolygon to potentially improve queries.

Time complexity:
`O(vertices_count * log vertices_count)` expected, `O(vertices_count ** 2)` worst

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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.index()
```

locate(*point: Point[Scalar]*) → *Location*

Finds location of the point relative to the multipolygon.

Time complexity:
`O(log vertices_count)` expected after indexing, `O(vertices_count)` worst after indexing or without it

Memory complexity:
`O(1)`

where

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

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.locate(Point(0, 0)) is Location.BOUNDARY
True
>>> multipolygon.locate(Point(1, 1)) is Location.INTERIOR
True
>>> multipolygon.locate(Point(2, 2)) is Location.BOUNDARY
True
```

(continues on next page)

(continued from previous page)

```
>>> multipolygon.locate(Point(3, 3)) is Location.EXTERIOR
True
>>> multipolygon.locate(Point(4, 5)) is Location.BOUNDARY
True
>>> multipolygon.locate(Point(5, 6)) is Location.INTERIOR
True
>>> multipolygon.locate(Point(6, 7)) is Location.BOUNDARY
True
>>> multipolygon.locate(Point(7, 7)) is Location.EXTERIOR
True
```

property perimeter: Scalar

Returns perimeter of the multipolygon.

Time complexity:

`O(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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.perimeter == 128
True
```

property polygons: Sequence[Polygon[Scalar]]

Returns polygons of the multipolygon.

Time complexity:

`O(polygons_count)`

Memory complexity:

`O(polygons_count)`

where `polygons_count = len(self.polygons)`.

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
```

(continues on next page)

(continued from previous page)

```
...                   [Contour([Point(2, 2), Point(2, 12),
...                             Point(12, 12), Point(12, 2)])]),
...           Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                            Point(4, 10)]),
...                   [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                             Point(8, 6)])])])
>>> (multipolygon.polygons
...  == [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                       Point(0, 14)]),
...              [Contour([Point(2, 2), Point(2, 12), Point(12, 12),
...                        Point(12, 2)])]),
...      Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                       Point(4, 10)]),
...              [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                        Point(8, 6)])])])
True
```

relate(*other:* Compound*[Scalar]*) → *Relation*

Finds relation between the multipolygon and the other geometry.

Time complexity:

O(vertices_count * log 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.relate(multipolygon) is Relation.EQUAL
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Multipolygon[Scalar]

Rotates the multipolygon by given angle around given point.

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 self.polygons)
```

```
>>> from gon.base import Angle, Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.rotate(Angle(1, 0)) == multipolygon
True
>>> (multipolygon.rotate(Angle(0, 1), Point(1, 1))
...  == Multipolygon([
...          Polygon(Contour([Point(2, 0), Point(2, 14),
...                           Point(-12, 14), Point(-12, 0)]),
...                  [Contour([Point(0, 2), Point(-10, 2),
...                            Point(-10, 12), Point(0, 12)])]),
...          Polygon(Contour([Point(-2, 4), Point(-2, 10),
...                           Point(-8, 10), Point(-8, 4)]),
...                  [Contour([Point(-4, 6), Point(-6, 6),
...                            Point(-6, 8), Point(-4, 8)])])]))
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → *Compound*[Scalar]

Scales the multipolygon 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
```

(continues on next page)

(continued from previous page)

```
...                             Point(4, 10)]),
...                     [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                               Point(8, 6)])])])
>>> multipolygon.scale(1) == multipolygon
True
>>> (multipolygon.scale(1, 2)
...  == Multipolygon([
...         Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 28),
...                          Point(0, 28)]),
...                 [Contour([Point(2, 4), Point(2, 24),
...                           Point(12, 24), Point(12, 4)])]),
...         Polygon(Contour([Point(4, 8), Point(10, 8), Point(10, 20),
...                          Point(4, 20)]),
...                 [Contour([Point(6, 12), Point(6, 16),
...                           Point(8, 16), Point(8, 12)])])]))
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Multipolygon[Scalar]

Translates the multipolygon 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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> (multipolygon.translate(1, 2)
...  == Multipolygon([
...         Polygon(Contour([Point(1, 2), Point(15, 2), Point(15, 16),
...                          Point(1, 16)]),
...                 [Contour([Point(3, 4), Point(3, 14),
...                           Point(13, 14), Point(13, 4)])]),
...         Polygon(Contour([Point(5, 6), Point(11, 6), Point(11, 12),
...                          Point(5, 12)]),
...                 [Contour([Point(7, 8), Point(7, 10), Point(9, 10),
...                           Point(9, 8)])])]))
```

(continues on next page)

(continued from previous page)

```
True
```

validate() → None

Checks if the multipolygon is valid.

Time complexity:

O(vertices_count * log (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 self.polygons)
```

```
>>> from gon.base import Contour, Multipolygon, Point, Polygon
>>> multipolygon = Multipolygon(
...         [Polygon(Contour([Point(0, 0), Point(14, 0), Point(14, 14),
...                           Point(0, 14)]),
...                  [Contour([Point(2, 2), Point(2, 12),
...                            Point(12, 12), Point(12, 2)])]),
...          Polygon(Contour([Point(4, 4), Point(10, 4), Point(10, 10),
...                           Point(4, 10)]),
...                  [Contour([Point(6, 6), Point(6, 8), Point(8, 8),
...                            Point(8, 6)])])])
>>> multipolygon.validate()
```

CHAPTER

SEVEN

MIXED GEOMETRIES

class gon.base.**Mix**(*discrete: Union[Empty, Multipoint[Scalar]]*, *linear: Union[Empty,* Linear*[Scalar]]*, *shaped: Union[Empty,* Shaped*[Scalar]]*)

__and__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the mix with the other geometry.

Time complexity:

```
O(elements_count * log elements_count)
```

Memory complexity:

```
O(elements_count)
```

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
```

(continues on next page)

(continued from previous page)

```
>>> mix & mix == mix
True
```

__contains__(*point: Point[Scalar]*) → bool

Checks if the mix contains the point.

Time complexity:

`O(log elements_count)` expected after indexing, `O(elements_count)` worst after indexing or without it

Memory complexity:

`O(1)`

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> Point(0, 0) in mix
True
>>> Point(1, 1) in mix
True
>>> Point(2, 2) in mix
True
>>> Point(3, 3) in mix
True
>>> Point(4, 3) in mix
True
>>> Point(5, 2) in mix
```

(continues on next page)

(continued from previous page)

```
True
>>> Point(6, 1) in mix
True
>>> Point(7, 0) in mix
False
```

__eq__(*other: Mix[Scalar]*) → bool

Checks if mixes are equal.

Time complexity:

O(elements_count)

Memory complexity:

O(1)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix == mix
True
```

__ge__(*other:* Compound*[Scalar]*) → bool

Checks if the mix is a superset of the other geometry.

Time complexity:

O(elements_count * log elements_count)

Memory complexity:

O(1)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix >= mix
True
```

__gt__(*other:* Compound*[Scalar]*) → bool

Checks if the mix is a strict superset of the other geometry.

Time complexity:

`O(elements_count * log elements_count)`

Memory complexity:

`O(1)`

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
```

(continues on next page)

(continued from previous page)

```
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix > mix
False
```

__hash__() → int

Returns hash value of the mix.

Time complexity:

O(components_size)

Memory complexity:

O(1)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
```

(continues on next page)

(continued from previous page)

```
...                             Point(0, 6)]),
...                    [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                              Point(4, 2)])]))
>>> hash(mix) == hash(mix)
True
```

__init__(*discrete: Union[Empty, Multipoint[Scalar]]*, *linear: Union[Empty, Linear[Scalar]]*, *shaped: Union[Empty, Shaped[Scalar]]*) → None

Initializes mix.

Time complexity:
O(1)

Memory complexity:
O(1)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__le__(*other: Compound[Scalar]*) → bool

Checks if the mix is a subset of the other geometry.

Time complexity:
O(elements_count * log elements_count)

Memory complexity:
O(1)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
```

(continues on next page)

(continued from previous page)

```
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix <= mix
True
```

__lt__(*other:* Compound*[Scalar]*) → bool

Checks if the mix is a strict subset of the other geometry.

Time complexity:

```
O(elements_count * log elements_count)
```

Memory complexity:

```
O(1)
```

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix < mix
False
```

static __new__(*cls*, **args*, ***kwds*)

__or__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the mix with the other geometry.

Time complexity:

```
O(elements_count * log elements_count)
```

Memory complexity:
O(elements_count)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix | mix == mix
True
```

__rand__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns intersection of the mix with the other geometry.

Time complexity:
O(elements_count * log elements_count)

Memory complexity:
O(elements_count)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
```

(continues on next page)

(continued from previous page)

```
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix & mix == mix
True
```

__repr__() → str

Return repr(self).

__ror__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns union of the mix with the other geometry.

Time complexity:

O(elements_count * log elements_count)

Memory complexity:

O(elements_count)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix | mix == mix
True
```

__rsub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the other geometry with the mix.

Time complexity:

```
O(elements_count * log elements_count)
```

Memory complexity:

```
O(1)
```

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

__rxor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the mix with the other geometry.

Time complexity:

```
O(elements_count * log elements_count)
```

Memory complexity:

```
O(elements_count)
```

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
```

(continues on next page)

(continued from previous page)

```
                          + sum(len(hole.vertices)
                                for hole in polygon.holes)
                          for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (EMPTY, Contour, Mix, Multipoint, Point,
...                       Polygon, Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix ^ mix is EMPTY
True
```

__sub__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns difference of the mix with the other geometry.

Time complexity:

`O(elements_count * log elements_count)`

Memory complexity:

`O(1)`

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                          + sum(len(hole.vertices)
                                for hole in polygon.holes)
                          for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
```

(continues on next page)

(continued from previous page)

```
                    if isinstance(self.linear, Multipolygon)
                    else [self.shaped]))
```

```
>>> from gon.base import (EMPTY, Contour, Mix, Multipoint, Point,
...                       Polygon, Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix - mix is EMPTY
True
```

__xor__(*other:* Compound*[Scalar]*) → *Compound*[Scalar]

Returns symmetric difference of the mix with the other geometry.

Time complexity:

```
O(elements_count * log elements_count)
```

Memory complexity:

```
O(elements_count)
```

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (EMPTY, Contour, Mix, Multipoint, Point,
...                       Polygon, Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
```

(continues on next page)

(continued from previous page)

```
>>> mix ^ mix is EMPTY
True
```

property centroid: Point[Scalar]

Returns centroid of the mix.

Time complexity:
O(elements_count)

Memory complexity:
O(1)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix.centroid == Point(3, 3)
True
```

property discrete: Union[Empty, Multipoint[Scalar]]

Returns disrete component of the mix.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix.discrete == Multipoint([Point(3, 3)])
True
```

disjoint(*other:* Compound*[Scalar]*) → bool

Checks if the geometry is disjoint from the other.

distance_to(*other:* Geometry*[Scalar]*) → Scalar

Returns distance between the mix and the other geometry.

Time complexity:

`O(elements_count)`

Memory complexity:

`O(1)`

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix.distance_to(mix) == 0
True
```

index() → None

Pre-processes the mix to potentially improve queries.

Time complexity:
`O(elements_count * log elements_count)` expected, `O(elements_count ** 2)` worst

Memory complexity:
`O(elements_count)`

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix.index()
```

property linear: Union[Empty, *Linear*[Scalar]]

Returns linear component of the mix.

Time complexity:
`O(1)`

Memory complexity:
`O(1)`

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
```

(continues on next page)

(continued from previous page)

```
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix.linear == Segment(Point(6, 6), Point(6, 8))
True
```

`locate`(*point: Point[Scalar]*) → *Location*

Finds location of the point relative to the mix.

Time complexity:

`O(log elements_count)` expected after indexing, `O(elements_count)` worst after indexing or without it

Memory complexity:

`O(1)`

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix.locate(Point(0, 0)) is Location.BOUNDARY
True
>>> mix.locate(Point(1, 1)) is Location.INTERIOR
True
>>> mix.locate(Point(2, 2)) is Location.BOUNDARY
True
>>> mix.locate(Point(3, 3)) is Location.BOUNDARY
True
>>> mix.locate(Point(4, 3)) is Location.BOUNDARY
```

(continues on next page)

(continued from previous page)

```
True
>>> mix.locate(Point(5, 2)) is Location.INTERIOR
True
>>> mix.locate(Point(6, 1)) is Location.BOUNDARY
True
>>> mix.locate(Point(7, 0)) is Location.EXTERIOR
True
```

relate(*other:* Compound*[Scalar]*) → *Relation*

Finds relation between the mix and the other geometry.

Time complexity:

```
O(elements_count * log elements_count)
```

Memory complexity:

```
O(elements_count)
```

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> mix.relate(mix) is Relation.EQUAL
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Mix[Scalar]

Rotates the mix by given angle around given point.

Time complexity:

```
O(elements_count)
```

Memory complexity:
O(elements_count)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Angle, Contour, Mix, Multipoint, Point,
...                       Polygon, Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix.rotate(Angle(1, 0)) == mix
True
>>> (mix.rotate(Angle(0, 1), Point(1, 1))
...  == Mix(Multipoint([Point(-1, 3)]),
...         Segment(Point(-4, 6), Point(-6, 6)),
...         Polygon(Contour([Point(2, 0), Point(2, 6), Point(-4, 6),
...                          Point(-4, 0)]),
...                 [Contour([Point(0, 2), Point(-2, 2), Point(-2, 4),
...                           Point(0, 4)])])))
True
```

scale(*factor_x: Scalar*, *factor_y: Optional[Scalar] = None*) → *Compound*[Scalar]

Scales the mix by given factor.

Time complexity:
O(elements_count)

Memory complexity:
O(elements_count)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                         + sum(len(hole.vertices)
                               for hole in polygon.holes)
                         for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix.scale(1) == mix
True
>>> (mix.scale(1, 2)
...  == Mix(Multipoint([Point(3, 6)]),
...         Segment(Point(6, 12), Point(6, 16)),
...         Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 12),
...                          Point(0, 12)]),
...                 [Contour([Point(2, 4), Point(2, 8), Point(4, 8),
...                           Point(4, 4)])])))
True
```

property shaped: Union[Empty, *Shaped*[Scalar]]

Returns shaped component of the mix.

Time complexity:

O(1)

Memory complexity:

O(1)

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
```

(continues on next page)

(continued from previous page)

```
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> from gon.base import Contour
>>> mix.shaped == Polygon(Contour([Point(0, 0), Point(6, 0),
...                                Point(6, 6), Point(0, 6)]),
...                       [Contour([Point(2, 2), Point(2, 4),
...                                 Point(4, 4), Point(4, 2)])])
True
```

translate(*step_x: Scalar*, *step_y: Scalar*) → Mix[Scalar]

Translates the mix by given step.

Time complexity:
O(elements_count)

Memory complexity:
O(elements_count)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                   [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                             Point(4, 2)])]))
>>> (mix.translate(1, 2)
...  == Mix(Multipoint([Point(4, 5)]),
...         Segment(Point(7, 8), Point(7, 10)),
...         Polygon(Contour([Point(1, 2), Point(7, 2), Point(7, 8),
...                          Point(1, 8)]),
...                 [Contour([Point(3, 4), Point(3, 6), Point(5, 6),
```

(continues on next page)

(continued from previous page)

```
...                             Point(5, 4)])])))
True
```

validate() → None

Checks if the mix is valid.

Time complexity:
O(elements_count * log elements_count)

Memory complexity:
O(elements_count)

where

```
elements_count = discrete_size + linear_size + shaped_vertices_count
discrete_size = len(points)
linear_size = len(segments)
shaped_vertices_count = (sum(len(polygon.border.vertices)
                             + sum(len(hole.vertices)
                                   for hole in polygon.holes)
                             for polygon in polygons)
points = [] if self.discrete is EMPTY else self.discrete.points
segments = ([]
            if self.linear is EMPTY
            else ([self.linear]
                  if isinstance(self.linear, Segment)
                  else self.linear.segments))
polygons = ([]
            if self.shaped is EMPTY
            else (self.shaped.polygons
                  if isinstance(self.linear, Multipolygon)
                  else [self.shaped]))
```

```
>>> from gon.base import (Contour, Mix, Multipoint, Point, Polygon,
...                       Segment)
>>> mix = Mix(Multipoint([Point(3, 3)]),
...           Segment(Point(6, 6), Point(6, 8)),
...           Polygon(Contour([Point(0, 0), Point(6, 0), Point(6, 6),
...                            Point(0, 6)]),
...                     [Contour([Point(2, 2), Point(2, 4), Point(4, 4),
...                               Point(4, 2)])]))
>>> mix.validate()
```

CHAPTER

EIGHT

HELPER GEOMETRIC OBJECTS

class gon.base.**Angle**(*cosine: Scalar*, *sine: Scalar*)

__add__(*other: Angle[Scalar]*) → Angle[Scalar]

Returns sum of the angle with the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> angle + Angle(1, 0) == angle
True
```

__bool__() → bool

Checks that the angle is non-zero.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> bool(angle)
True
```

__eq__(*other: Angle*) → bool

Checks if the angle is equal to the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> angle == angle
True
```

__hash__() → int

Returns hash value of the angle.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> hash(angle) == hash(angle)
True
```

__init__(*cosine: Scalar*, *sine: Scalar*) → None

Initializes angle.

Time complexity:
O(1)

Memory complexity:
O(1)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__neg__() → Angle[Scalar]

Returns the angle negated.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> -angle == Angle(0, -1)
True
```

static __new__(*cls*, **args*, ***kwds*)

__pos__() → Angle[Scalar]

Returns the angle positive.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> +angle == angle
True
```

__repr__() → str

Return repr(self).

__sub__(*other: Angle[Scalar]*) → Angle[Scalar]

Returns difference of the angle with the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> angle - Angle(1, 0) == angle
True
```

property cosine: Scalar

Returns cosine of the angle.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> angle.cosine == 0
True
```

classmethod from_sides(*vertex: Point[Scalar]*, *first_ray_point: Point[Scalar]*, *second_ray_point: Point[Scalar]*) → Angle[Scalar]

Constructs angle from sides.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle, Point
>>> angle = Angle.from_sides(Point(0, 0), Point(1, 0), Point(0, 1))
>>> angle == Angle(0, 1)
True
```

property kind: Kind

Returns kind of the angle.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle, Kind
>>> angle = Angle(0, 1)
```

(continues on next page)

(continued from previous page)

```
>>> angle.kind is Kind.RIGHT
True
```

property orientation: *Orientation*

Returns orientation of the angle.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle, Orientation
>>> angle = Angle(0, 1)
>>> angle.orientation is Orientation.COUNTERCLOCKWISE
True
```

property sine: Scalar

Returns sine of the angle.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> angle.sine == 1
True
```

validate() → None

Checks if the angle is valid.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle
>>> angle = Angle(0, 1)
>>> angle.validate()
```

class gon.base.**Vector**(*start: Point[Scalar]*, *end: Point[Scalar]*)

__add__(*other: Vector[Scalar]*) → Vector[Scalar]

Returns sum of the vector with the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector + Vector(Point(0, 0), Point(0, 0)) == vector
True
```

__bool__() → bool

Checks that the vector is non-zero.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> bool(vector)
True
```

__eq__(*other: Vector[Scalar]*) → bool

Checks if the vector is equal to the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector == vector
True
```

__hash__() → int

Returns hash value of the vector.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> hash(vector) == hash(vector)
True
```

__init__(*start: Point[Scalar]*, *end: Point[Scalar]*) → None

Initializes vector.

Time complexity:
O(1)

Memory complexity:
O(1)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__mul__(*factor: Scalar*) → Vector[Scalar]

Scales the vector by given factor.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector * 1 == vector
True
```

__neg__() → Vector[Scalar]

Returns the vector negated.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> -vector == Vector(Point(2, 0), Point(0, 0))
True
```

static __new__(*cls*, **args*, ***kwds*)

__pos__() → Vector[Scalar]

Returns the vector positive.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> +vector == vector
True
```

__repr__() → str

Return repr(self).

__rmul__(*factor: Scalar*) → Vector[Scalar]

Scales the vector by given factor.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector * 1 == vector
True
```

__sub__(*other: Vector[Scalar]*) → Vector

Returns difference of the vector with the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector - Vector(Point(0, 0), Point(0, 0)) == vector
True
```

cross(*other: Vector[Scalar]*) → Scalar

Returns cross product of the vector with the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.cross(vector) == 0
True
```

dot(*other: Vector[Scalar]*) → Scalar

Returns dot product of the vector with the other.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.dot(vector) == 4
True
```

property end: Point[Scalar]

Returns end of the vector.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.end == Point(2, 0)
True
```

classmethod from_position(*end: Point[Scalar]*) → Vector[Scalar]

Constructs position vector.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector.from_position(Point(2, 0))
>>> vector == Vector(Point(0, 0), Point(2, 0))
True
```

kind_of(*point: Point[Scalar]*) → Kind

Returns kind of angle formed by the vector and given point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Kind, Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.kind_of(vector.end) is Kind.ACUTE
True
```

property length: Scalar

Returns length of the vector.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.length == 2
True
```

orientation_of(*point: Point[Scalar]*) → *Orientation*

Returns orientation of angle formed by the vector and given point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Orientation, Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.orientation_of(vector.end) is Orientation.COLLINEAR
True
```

rotate(*angle: Angle*, *point: Optional[Point[Scalar]] = None*) → Vector[Scalar]

Rotates the vector by given angle around given point.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Angle, Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.rotate(Angle(1, 0)) == vector
True
>>> (vector.rotate(Angle(0, 1), Point(1, 1))
...  == Vector(Point(2, 0), Point(2, 2)))
True
```

property start: Point[Scalar]

Returns start of the vector.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.start == Point(0, 0)
True
```

validate() → None

Checks if vector is finite.

Time complexity:
O(1)

Memory complexity:
O(1)

```
>>> from gon.base import Point, Vector
>>> vector = Vector(Point(0, 0), Point(2, 0))
>>> vector.validate()
```

CHAPTER

NINE

ENUMERATIONS

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

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

BOUNDARY = 1

point lies on the boundary of the geometry

EXTERIOR = 0

point lies in the exterior of the geometry

INTERIOR = 2

point lies in the interior of the geometry

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

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

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

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

COVER = 4

interior of the geometry is a superset of the other

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

DISJOINT = 0

intersection is empty

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

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

EQUAL = 7

geometries are equal

OVERLAP = 3

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

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

WITHIN = 10

geometry is a subset of the interior of the other

CHAPTER

TEN

GRAPHS

class gon.base.**Triangulation**(*left_side: QuadEdge*, *right_side: QuadEdge*, *context: Context*)

Represents triangulation.

classmethod constrained_delaunay(*polygon: Polygon*, ***, *extra_constraints: Sequence[Segment] = ()*, *extra_points: Sequence[Point] = ()*, *context: Context*) → *Triangulation*

Constructs constrained Delaunay triangulation of given polygon (with potentially extra points and constraints).

Based on

- divide-and-conquer algorithm by L. Guibas & J. Stolfi for generating Delaunay triangulation,
- algorithm by S. W. Sloan for adding constraints to Delaunay triangulation,
- clipping algorithm by F. Martinez et al. for deleting in-hole triangles.

Time complexity:
`O(vertices_count * log vertices_count)` for convex polygons without extra constraints, `O(vertices_count ** 2)` otherwise

Memory complexity:
`O(vertices_count)`

where

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

Reference:
http://www.sccg.sk/~samuelcik/dgs/quad_edge.pdf https://www.newcastle.edu.au/__data/assets/pdf_file/0019/22519/23_A-fast-algortithm-for-generating-constrained-Delaunay-triangulations.pdf https://doi.org/10.1016/j.advengsoft.2013.04.004 http://www4.ujaen.es/~fmartin/bool_op.html

Parameters

- **polygon** – target polygon.
- **extra_points** – additional points to be presented in the triangulation.
- **extra_constraints** – additional constraints to be presented in the triangulation.
- **context** – geometric context.

Returns
triangulation of the border, holes & extra points considering constraints.

classmethod delaunay(*points: Sequence[Point]*, ***, *context: Context*) → *Triangulation*

Constructs Delaunay triangulation of given points.

Based on divide-and-conquer algorithm by L. Guibas & J. Stolfi.

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

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

Reference:
http://www.sccg.sk/~samuelcik/dgs/quad_edge.pdf

Parameters

- **points** – 3 or more points to triangulate.
- **context** – geometric context.

Returns
triangulation of the points.

delete(*edge: QuadEdge*) → None

Deletes given edge from the triangulation.

triangles() → List[Contour]

Returns triangles of the triangulation.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

V

W

X

Y