- **segments** – count of Bèzier-curve segments, at least one segment for each quarter (pi/2), 1 for as few as possible.

New in version 0.13.

ezdxf.math.**cubic_bezier_interpolation**(*points: Iterable[Vertex]*) → List[Bezier4P]
Returns an interpolation curve for given data *points* as multiple cubic Bézier-curves. Returns n-1 cubic Bézier-curves for n given data points, curve i goes from point[i] to point[i+1].

Parameters **points** – data points

New in version 0.13.

Transformation Classes

OCS Class

class ezdxf.math.**OCS**(*extrusion: Vertex = Vec3(0.0, 0.0, 1.0)*)
Establish an *OCS* for a given extrusion vector.

Parameters **extrusion** – extrusion vector.

ux
x-axis unit vector

uy
y-axis unit vector

uz
z-axis unit vector

from_wcs(*point: Vertex*) → Vertex
Returns OCS vector for WCS *point*.

points_from_wcs(*points: Iterable[Vertex]*) → Iterable[Vertex]
Returns iterable of OCS vectors from WCS *points*.

to_wcs(*point: Vertex*) → Vertex
Returns WCS vector for OCS *point*.

points_to_wcs(*points: Iterable[Vertex]*) → Iterable[Vertex]
Returns iterable of WCS vectors for OCS *points*.

render_axis(*layout: BaseLayout, length: float = 1, colors: Tuple[int, int, int] = (1, 3, 5)*)
Render axis as 3D lines into a *layout*.

UCS Class

class ezdxf.math.**UCS**(*origin: Vertex = (0, 0, 0), ux: Vertex = None, uy: Vertex = None, uz: Vertex = None*)
Establish an user coordinate system (*UCS*). The UCS is defined by the origin and two unit vectors for the x-, y- or z-axis, all axis in *WCS*. The missing axis is the cross product of the given axis.

If x- and y-axis are `None`: ux = `(1, 0, 0)`, uy = `(0, 1, 0)`, uz = `(0, 0, 1)`.

Unit vectors don't have to be normalized, normalization is done at initialization, this is also the reason why scaling gets lost by copying or rotating.

Parameters

- **ux** – defines the UCS x-axis as vector in *WCS*
- **uy** – defines the UCS y-axis as vector in *WCS*
- **uz** – defines the UCS z-axis as vector in *WCS*

ux
x-axis unit vector

uy
y-axis unit vector

uz
z-axis unit vector

is_cartesian
Returns `True` if cartesian coordinate system.

copy() → UCS
Returns a copy of this UCS.

to_wcs(*point: ezdxf.math._vector.Vec3*) → ezdxf.math._vector.Vec3
Returns WCS point for UCS *point*.

points_to_wcs(*points: Iterable[Vec3]*) → Iterable[ezdxf.math._vector.Vec3]
Returns iterable of WCS vectors for UCS *points*.

direction_to_wcs(*vector: ezdxf.math._vector.Vec3*) → ezdxf.math._vector.Vec3
Returns WCS direction for UCS *vector* without origin adjustment.

from_wcs(*point: ezdxf.math._vector.Vec3*) → ezdxf.math._vector.Vec3
Returns UCS point for WCS *point*.

points_from_wcs(*points: Iterable[Vec3]*) → Iterable[ezdxf.math._vector.Vec3]
Returns iterable of UCS vectors from WCS *points*.

direction_from_wcs(*vector: ezdxf.math._vector.Vec3*) → ezdxf.math._vector.Vec3
Returns UCS vector for WCS *vector* without origin adjustment.

to_ocs(*point: ezdxf.math._vector.Vec3*) → ezdxf.math._vector.Vec3
Returns OCS vector for UCS *point*.

The *OCS* is defined by the z-axis of the *UCS*.

points_to_ocs(*points: Iterable[Vec3]*) → Iterable[ezdxf.math._vector.Vec3]
Returns iterable of OCS vectors for UCS *points*.

The *OCS* is defined by the z-axis of the *UCS*.

Parameters **points** – iterable of UCS vertices

to_ocs_angle_deg(*angle: float*) → float
Transforms *angle* from current UCS to the parent coordinate system (most likely the WCS) including the transformation to the OCS established by the extrusion vector *UCS.uz*.

Parameters **angle** – in UCS in degrees

transform(*m: Matrix44*) → UCS
General inplace transformation interface, returns *self* (floating interface).

Parameters **m** – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

New in version 0.14.

rotate (*axis: Vertex*, *angle:float*) → UCS
Returns a new rotated UCS, with the same origin as the source UCS. The rotation vector is located in the origin and has *WCS* coordinates e.g. (0, 0, 1) is the WCS z-axis as rotation vector.

Parameters

- **axis** – arbitrary rotation axis as vector in *WCS*
- **angle** – rotation angle in radians

rotate_local_x (*angle:float*) → UCS
Returns a new rotated UCS, rotation axis is the local x-axis.

Parameters **angle** – rotation angle in radians

rotate_local_y (*angle:float*) → UCS
Returns a new rotated UCS, rotation axis is the local y-axis.

Parameters **angle** – rotation angle in radians

rotate_local_z (*angle:float*) → UCS
Returns a new rotated UCS, rotation axis is the local z-axis.

Parameters **angle** – rotation angle in radians

shift (*delta: Vertex*) → UCS
Shifts current UCS by *delta* vector and returns *self*.

Parameters **delta** – shifting vector

moveto (*location: Vertex*) → UCS
Place current UCS at new origin *location* and returns *self*.

Parameters **location** – new origin in WCS

static from_x_axis_and_point_in_xy (*origin: Vertex*, *axis: Vertex*, *point: Vertex*) → UCS
Returns an new *UCS* defined by the origin, the x-axis vector and an arbitrary point in the xy-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – x-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in *WCS*

static from_x_axis_and_point_in_xz (*origin: Vertex*, *axis: Vertex*, *point: Vertex*) → UCS
Returns an new *UCS* defined by the origin, the x-axis vector and an arbitrary point in the xz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – x-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in *WCS*

static from_y_axis_and_point_in_xy (*origin: Vertex*, *axis: Vertex*, *point: Vertex*) → UCS
Returns an new *UCS* defined by the origin, the y-axis vector and an arbitrary point in the xy-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – y-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in *WCS*

static from_y_axis_and_point_in_yz (*origin: Vertex*, *axis: Vertex*, *point: Vertex*) → UCS
Returns an new *UCS* defined by the origin, the y-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – y-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in *WCS*

static from_z_axis_and_point_in_xz (*origin: Vertex*, *axis: Vertex*, *point: Vertex*) → UCS
Returns an new *UCS* defined by the origin, the z-axis vector and an arbitrary point in the xz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – z-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in *WCS*

static from_z_axis_and_point_in_yz (*origin: Vertex*, *axis: Vertex*, *point: Vertex*) → UCS
Returns an new *UCS* defined by the origin, the z-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in *WCS*
- **axis** – z-axis vector as (x, y, z) tuple in *WCS*
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in *WCS*

render_axis (*layout: BaseLayout*, *length: float = 1*, *colors: Tuple[int, int, int] = (1, 3, 5)*)
Render axis as 3D lines into a *layout*.

Matrix44

class ezdxf.math.**Matrix44** (**args*)
This is a pure Python implementation for 4x4 transformation matrices, to avoid dependency to big numerical packages like `numpy`, before binary wheels, installation of these packages wasn't always easy on Windows.

The utility functions for constructing transformations and transforming vectors and points assumes that vectors are stored as row vectors, meaning when multiplied, transformations are applied left to right (e.g. vAB transforms v by A then by B).

Matrix44 initialization:

- `Matrix44()` returns the identity matrix.
- `Matrix44(values)` values is an iterable with the 16 components of the matrix.
- `Matrix44(row1, row2, row3, row4)` four rows, each row with four values.

__repr__() → str
Returns the representation string of the matrix: `Matrix44((col0, col1, col2, col3), (...), (...), (...))`

get_row (*row: int*) → Tuple[float, ...]
Get row as list of of four float values.

Parameters **row** – row index [0 .. 3]

set_row (*row: int*, *values: Sequence[float]*) → None
Sets the values in a row.

Parameters

- **row** – row index [0 .. 3]
- **values** – iterable of four row values

get_col (*col: int*) → Tuple[float, ...]
Returns a column as a tuple of four floats.

Parameters **col** – column index [0 .. 3]

set_col (*col: int, values: Sequence[float]*)
Sets the values in a column.

Parameters

- **col** – column index [0 .. 3]
- **values** – iterable of four column values

copy () → Matrix44
Returns a copy of same type.

__copy__ () → Matrix44
Returns a copy of same type.

classmethod scale (*sx: float*, *sy: float = None*, *sz: float = None*) → Matrix44
Returns a scaling transformation matrix. If *sy* is `None`, *sy* = *sx*, and if *sz* is `None` *sz* = *sx*.

classmethod translate (*dx: float*, *dy: float*, *dz: float*) → Matrix44
Returns a translation matrix for translation vector (dx, dy, dz).

classmethod x_rotate (*angle: float*) → Matrix44
Returns a rotation matrix about the x-axis.

Parameters **angle** – rotation angle in radians

classmethod y_rotate (*angle: float*) → Matrix44
Returns a rotation matrix about the y-axis.

Parameters **angle** – rotation angle in radians

classmethod z_rotate (*angle: float*) → Matrix44
Returns a rotation matrix about the z-axis.

Parameters **angle** – rotation angle in radians

classmethod axis_rotate (*axis: Vertex*, *angle: float*) → Matrix44
Returns a rotation matrix about an arbitrary *axis*.

Parameters

- **axis** – rotation axis as `(x, y, z)` tuple or *Vec3* object
- **angle** – rotation angle in radians

classmethod xyz_rotate (*angle_x: float*, *angle_y: float*, *angle_z: float*) → Matrix44
Returns a rotation matrix for rotation about each axis.

Parameters

- **angle_x** – rotation angle about x-axis in radians
- **angle_y** – rotation angle about y-axis in radians
- **angle_z** – rotation angle about z-axis in radians

classmethod perspective_projection(*left: float*, *right: float*, *top: float*, *bottom: float*, *near: float*, *far: float*) → Matrix44

Returns a matrix for a 2D projection.

Parameters

- **left** – Coordinate of left of screen
- **right** – Coordinate of right of screen
- **top** – Coordinate of the top of the screen
- **bottom** – Coordinate of the bottom of the screen
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

classmethod perspective_projection_fov(*fov: float*, *aspect: float*, *near: float*, *far: float*) → Matrix44

Returns a matrix for a 2D projection.

Parameters

- **fov** – The field of view (in radians)
- **aspect** – The aspect ratio of the screen (width / height)
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

static chain(**matrices: Iterable[Matrix44]*) → Matrix44

Compose a transformation matrix from one or more *matrices*.

static ucs(*ux: Vertex*, *uy: Vertex*, *uz: Vertex*) → Matrix44

Returns a matrix for coordinate transformation from WCS to UCS. For transformation from UCS to WCS, transpose the returned matrix.

Parameters

- **ux** – x-axis for UCS as unit vector
- **uy** – y-axis for UCS as unit vector
- **uz** – z-axis for UCS as unit vector
- **origin** – UCS origin as location vector

__hash__()

Return hash(self).

__getitem__(*index: Tuple[int, int]*)

Get (row, column) element.

__setitem__(*index: Tuple[int, int]*, *value: float*)

Set (row, column) element.

__iter__() → Iterable[float]

Iterates over all matrix values.

rows() → Iterable[Tuple[float, ...]]

Iterate over rows as 4-tuples.

columns() → Iterable[Tuple[float, ...]]

Iterate over columns as 4-tuples.

__mul__(*other: Matrix44*) → Matrix44
Returns a new matrix as result of the matrix multiplication with another matrix.

__imul__(*other: Matrix44*) → Matrix44
Inplace multiplication with another matrix.

transform(*vector: Vertex*) → ezdxf.math._vector.Vec3
Returns a transformed vertex.

transform_direction(*vector: Vertex*, *normalize=False*) → ezdxf.math._vector.Vec3
Returns a transformed direction vector without translation.

transform_vertices(*vectors: Iterable[Vertex]*) → Iterable[ezdxf.math._vector.Vec3]
Returns an iterable of transformed vertices.

transform_directions(*vectors: Iterable[Vertex]*, *normalize=False*) → Iterable[ezdxf.math._vector.Vec3]
Returns an iterable of transformed direction vectors without translation.

transpose() → None
Swaps the rows for columns inplace.

determinant() → float
Returns determinant.

inverse() → None
Calculates the inverse of the matrix.

Raises ZeroDivisionError – if matrix has no inverse.

Construction Tools

Vec3

class ezdxf.math.**Vec3**(**args*)
This is an immutable universal 3D vector object. This class is optimized for universality not for speed. Immutable means you can't change (x, y, z) components after initialization:

```
v1 = Vec3(1, 2, 3)
v2 = v1
v2.z = 7  # this is not possible, raises AttributeError
v2 = Vec3(v2.x, v2.y, 7)  # this creates a new Vec3() object
assert v1.z == 3  # and v1 remains unchanged
```

Vec3 initialization:

- Vec3(), returns Vec3(0, 0, 0)
- Vec3((x, y)), returns Vec3(x, y, 0)
- Vec3((x, y, z)), returns Vec3(x, y, z)
- Vec3(x, y), returns Vec3(x, y, 0)
- Vec3(x, y, z), returns Vec3(x, y, z)

Addition, subtraction, scalar multiplication and scalar division left and right handed are supported:

```
v = Vec3(1, 2, 3)
v + (1, 2, 3) == Vec3(2, 4, 6)
(1, 2, 3) + v == Vec3(2, 4, 6)
```

(continues on next page)

(continued from previous page)

```
v - (1, 2, 3) == Vec3(0, 0, 0)
(1, 2, 3) - v == Vec3(0, 0, 0)
v * 3 == Vec3(3, 6, 9)
3 * v == Vec3(3, 6, 9)
Vec3(3, 6, 9) / 3 == Vec3(1, 2, 3)
-Vec3(1, 2, 3) == (-1, -2, -3)
```

Comparison between vectors and vectors or tuples is supported:

```
Vec3(1, 2, 3) < Vec3 (2, 2, 2)
(1, 2, 3) < tuple(Vec3(2, 2, 2))  # conversion necessary
Vec3(1, 2, 3) == (1, 2, 3)

bool(Vec3(1, 2, 3)) is True
bool(Vec3(0, 0, 0)) is False
```

x
x-axis value

y
y-axis value

z
z-axis value

xy
Vec3 as `(x, y, 0)`, projected on the xy-plane.

xyz
Vec3 as `(x, y, z)` tuple.

vec2
Real 2D vector as *Vec2* object.

magnitude
Length of vector.

magnitude_xy
Length of vector in the xy-plane.

magnitude_square
Square length of vector.

is_null
`True` for `Vec3(0, 0, 0)`.

angle
Angle between vector and x-axis in the xy-plane in radians.

angle_deg
Returns angle of vector and x-axis in the xy-plane in degrees.

spatial_angle
Spatial angle between vector and x-axis in radians.

spatial_angle_deg
Spatial angle between vector and x-axis in degrees.

__str__() → str
Return `'(x, y, z)'` as string.

__repr__() → str
Return 'Vec3(x, y, z)' as string.

__len__() → int
Returns always 3.

__hash__() → int
Returns hash value of vector, enables the usage of vector as key in set and dict.

copy() → Vec3
Returns a copy of vector as *Vec3* object.

__copy__() → Vec3
Returns a copy of vector as *Vec3* object.

__deepcopy__(*memodict: dict*) → Vec3
copy.deepcopy() support.

__getitem__(*index: int*) → float
Support for indexing:

- v[0] is v.x
- v[1] is v.y
- v[2] is v.z

__iter__() → Iterable[float]
Returns iterable of x-, y- and z-axis.

__abs__() → float
Returns length (magnitude) of vector.

replace(*x: float = None*, *y: float = None*, *z: float = None*) → Vec3
Returns a copy of vector with replaced x-, y- and/or z-axis.

classmethod generate(*items: Iterable[Vertex]*) → Iterable[Vec3]
Returns an iterable of *Vec3* objects.

classmethod list(*items: Iterable[Vertex]*) → List[Vec3]
Returns a list of *Vec3* objects.

classmethod tuple(*items: Iterable[Vertex]*) → Sequence[Vec3]
Returns a tuple of *Vec3* objects.

classmethod from_angle(*angle: float*, *length: float = 1.*) → Vec3
Returns a *Vec3* object from *angle* in radians in the xy-plane, z-axis = 0.

classmethod from_deg_angle(*angle: float*, *length: float = 1.*) → Vec3
Returns a *Vec3* object from *angle* in degrees in the xy-plane, z-axis = 0.

orthogonal(*ccw: bool = True*) → Vec3
Returns orthogonal 2D vector, z-axis is unchanged.

Parameters **ccw** – counter clockwise if True else clockwise

lerp(*other: Vertex*, *factor=.5*) → Vec3
Returns linear interpolation between *self* and *other*.

Parameters

- **other** – end point as *Vec3* compatible object
- **factor** – interpolation factor (0 = self, 1 = other, 0.5 = mid point)

is_parallel (*other: Vec3*, *abs_tolr=1e-12*) → bool
Returns `True` if *self* and *other* are parallel to vectors.

project (*other: Vertex*) → Vec3
Returns projected vector of *other* onto *self*.

normalize (*length: float = 1.*) → Vec3
Returns normalized vector, optional scaled by *length*.

reversed () → Vec3
Returns negated vector (*-self*).

isclose (*other: Vertex*, *abs_tol: float = 1e-12*) → bool
Returns `True` if *self* is close to *other*. Uses `math.isclose()` to compare all axis.

__neg__ () → Vec3
Returns negated vector (*-self*).

__bool__ () → bool
Returns `True` if vector is not `(0, 0, 0)`.

__eq__ (*other: Vertex*) → bool
Equal operator.

Parameters **other** – *Vec3* compatible object

__lt__ (*other: Vertex*) → bool
Lower than operator.

Parameters **other** – *Vec3* compatible object

__add__ (*other: Vertex*) → Vec3
Add *Vec3* operator: *self* + *other*.

__radd__ (*other: Vertex*) → Vec3
RAdd *Vec3* operator: *other* + *self*.

__sub__ (*other: Vertex*) → Vec3
Sub *Vec3* operator: *self* - *other*.

__rsub__ (*other: Vertex*) → Vec3
RSub *Vec3* operator: *other* - *self*.

__mul__ (*other: float*) → Vec3
Scalar Mul operator: *self* * *other*.

__rmul__ (*other: float*) → Vec3
Scalar RMul operator: *other* * *self*.

__truediv__ (*other: float*) → Vec3
Scalar Div operator: *self* / *other*.

dot (*other: Vertex*) → float
Dot operator: *self* . *other*

Parameters **other** – *Vec3* compatible object

cross (*other: Vertex*) → Vec3
Dot operator: *self* x *other*

Parameters **other** – *Vec3* compatible object

distance (*other: Vertex*) → float
Returns distance between *self* and *other* vector.

angle_about (*base: Vec3*, *target: Vec3*) → float
Returns counter clockwise angle in radians about *self* from *base* to *target* when projected onto the plane defined by *self* as the normal vector.

Parameters

- **base** – base vector, defines angle 0
- **target** – target vector

angle_between (*other: Vertex*) → float
Returns angle between *self* and *other* in radians. +angle is counter clockwise orientation.

Parameters **other** – *Vec3* compatible object

rotate (*angle: float*) → Vec3
Returns vector rotated about *angle* around the z-axis.

Parameters **angle** – angle in radians

rotate_deg (*angle: float*) → Vec3
Returns vector rotated about *angle* around the z-axis.

Parameters **angle** – angle in degrees

static sum (*items: Iterable[Vertex]*) → Vec3
Add all vectors in *items*.

ezdxf.math.**X_AXIS**
`Vec3(1, 0, 0)`

ezdxf.math.**Y_AXIS**
`Vec3(0, 1, 0)`

ezdxf.math.**Z_AXIS**
`Vec3(0, 0, 1)`

ezdxf.math.**NULLVEC**
`Vec3(0, 0, 0)`

Vec2

class ezdxf.math.**Vec2** (*v: Any*, *y: float = None*)
Vec2 represents a special 2D vector `(x, y)`. The *Vec2* class is optimized for speed and not immutable, `iadd()`, `isub()`, `imul()` and `idiv()` modifies the vector itself, the *Vec3* class returns a new object.

Vec2 initialization accepts float-tuples `(x, y[, z])`, two floats or any object providing `x` and `y` attributes like *Vec2* and *Vec3* objects.

Parameters

- **v** – vector object with `x` and `y` attributes/properties or a sequence of float `[x, y, ...]` or x-axis as float if argument *y* is not `None`
- **y** – second float for `Vec2(x, y)`

Vec2 implements a subset of *Vec3*.

Plane

class ezdxf.math.**Plane**(*normal: Vec3*, *distance: float*)

Represents a plane in 3D space as normal vector and the perpendicular distance from origin.

normal

Normal vector of the plane.

distance_from_origin

The (perpendicular) distance of the plane from origin (0, 0, 0).

vector

Returns the location vector.

classmethod from_3p(*a: Vec3*, *b: Vec3*, *c: Vec3*) → Plane

Returns a new plane from 3 points in space.

classmethod from_vector(*vector*) → Plane

Returns a new plane from a location vector.

copy() → Plane

Returns a copy of the plane.

signed_distance_to(*v: Vec3*) → float

Returns signed distance of vertex *v* to plane, if distance is > 0, *v* is in 'front' of plane, in direction of the normal vector, if distance is < 0, *v* is at the 'back' of the plane, in the opposite direction of the normal vector.

distance_to(*v: Vec3*) → float

Returns absolute (unsigned) distance of vertex *v* to plane.

is_coplanar_vertex(*v: Vec3*, *abs_tol=1e-9*) → bool

Returns `True` if vertex *v* is coplanar, distance from plane to vertex *v* is 0.

is_coplanar_plane(*p: Plane*, *abs_tol=1e-9*) → bool

Returns `True` if plane *p* is coplanar, normal vectors in same or opposite direction.

BoundingBox

class ezdxf.math.**BoundingBox**(*vertices: Iterable[Vertex] = None*)

3D bounding box.

Parameters **vertices** – iterable of `(x, y, z)` tuples or *Vec3* objects

extmin

"lower left" corner of bounding box

extmax

"upper right" corner of bounding box

center

Returns center of bounding box.

extend(*vertices: Iterable[Vertex]*) → None

Extend bounds by *vertices*.

Parameters **vertices** – iterable of `(x, y, z)` tuples or *Vec3* objects

has_data

Returns `True` if data is available

inside (*vertex: Vertex*) → bool
Returns `True` if *vertex* is inside bounding box.

size
Returns size of bounding box.

BoundingBox2d

class ezdxf.math.**BoundingBox2d** (*vertices: Iterable[Vertex] = None*)
Optimized 2D bounding box.

Parameters **vertices** – iterable of `(x, y[, z])` tuples or *Vec3* objects

extmin
"lower left" corner of bounding box

extmax
"upper right" corner of bounding box

center
Returns center of bounding box.

extend (*vertices: Iterable[Vertex]*) → None
Extend bounds by *vertices*.

Parameters **vertices** – iterable of `(x, y[, z])` tuples or *Vec3* objects

has_data
Returns `True` if data is available

inside (*vertex: Vertex*) → bool
Returns `True` if *vertex* is inside bounding box.

size
Returns size of bounding box.

ConstructionRay

class ezdxf.math.**ConstructionRay** (*p1: Vertex, p2: Vertex = None, angle: float = None*)
Infinite 2D construction ray as immutable object.

Parameters

- **p1** – definition point 1
- **p2** – ray direction as 2nd point or `None`
- **angle** – ray direction as angle in radians or `None`

location
Location vector as *Vec2*.

direction
Direction vector as *Vec2*.

slope
Slope of ray or `None` if vertical.

angle
Angle between x-axis and ray in radians.

angle_deg
Angle between x-axis and ray in degrees.

is_vertical
`True` if ray is vertical (parallel to y-axis).

is_horizontal
`True` if ray is horizontal (parallel to x-axis).

__str__()
Return str(self).

is_parallel(*self*, *other: ConstructionRay*) → bool
Returns `True` if rays are parallel.

intersect(*other: ConstructionRay*) → Vec2
Returns the intersection point as `(x, y)` tuple of *self* and *other*.

Raises `ParallelRaysError` – if rays are parallel

orthogonal(*location: 'Vertex'*) → ConstructionRay
Returns orthogonal ray at *location*.

bisectrix(*other: ConstructionRay*) → ConstructionRay:
Bisectrix between *self* and *other*.

yof(*x: float*) → float
Returns y-value of ray for *x* location.

Raises `ArithmeticError` – for vertical rays

xof(*y: float*) → float
Returns x-value of ray for *y* location.

Raises `ArithmeticError` – for horizontal rays

ConstructionLine

class `ezdxf.math.`**ConstructionLine**(*start: Vertex*, *end: Vertex*)
2D ConstructionLine is similar to *ConstructionRay*, but has a start- and endpoint. The direction of line goes from start- to endpoint, "left of line" is always in relation to this line direction.

Parameters

- **start** – start point of line as *Vec2* compatible object
- **end** – end point of line as *Vec2* compatible object

start
start point as *Vec2*

end
end point as *Vec2*

bounding_box
bounding box of line as *BoundingBox2d* object.

ray
collinear *ConstructionRay*.

is_vertical
`True` if line is vertical.

__str__()
Return str(self).

translate(*dx: float*, *dy: float*) → None
Move line about *dx* in x-axis and about *dy* in y-axis.

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis

length() → float
Returns length of line.

midpoint() → Vec2
Returns mid point of line.

inside_bounding_box(*point: Vertex*) → bool
Returns `True` if *point* is inside of line bounding box.

intersect(*other: ConstructionLine*, *abs_tol:float=1e-10*) → Optional[Vec2]
Returns the intersection point of to lines or `None` if they have no intersection point.

Parameters

- **other** – other *ConstructionLine*
- **abs_tol** – tolerance for distance check

has_intersection(*other: ConstructionLine*, *abs_tol:float=1e-10*) → bool
Returns `True` if has intersection with *other* line.

is_point_left_of_line(*point: Vertex*, *colinear=False*) → bool
Returns `True` if *point* is left of construction line in relation to the line direction from start to end.

If *colinear* is `True`, a colinear point is also left of the line.

ConstructionCircle

class ezdxf.math.**ConstructionCircle**(*center: Vertex*, *radius: float = 1.0*)
Circle construction tool.

Parameters

- **center** – center point as *Vec2* compatible object
- **radius** – circle radius > *0*

center
center point as *Vec2*

radius
radius as float

bounding_box
2D bounding box of circle as *BoundingBox2d* object.

static from_3p(*p1: Vertex*, *p2: Vertex*, *p3: Vertex*) → ConstructionCircle
Creates a circle from three points, all points have to be compatible to *Vec2* class.

__str__() → str
Returns string representation of circle "ConstructionCircle(center, radius)".