Math Types & Utilities (mathutils)

This module provides access to math operations.

Note

Classes, methods and attributes that accept vectors also accept other numeric sequences, such as tuples, lists.

The `mathutils` module provides the following classes:

- `Color`,
- `Euler`,
- `Matrix`,
- `Quaternion`,
- `Vector`,

SUBMODULES

```
import mathutils
from math import radians

vec = mathutils.Vector((1.0, 2.0, 3.0))

mat_rot = mathutils.Matrix.Rotation(radians(90.0), 4, 'X')
mat_trans = mathutils.Matrix.Translation(vec)

mat = mat_trans @ mat_rot
mat.invert()

mat3 = mat.to_3x3()
quat1 = mat.to_quaternion()
quat2 = mat3.to_quaternion()

quat_diff = quat1.rotation_difference(quat2)

print(quat_diff.angle)
```

class mathutils.**Color(rgb)**

This object gives access to Colors in Blender.

Most colors returned by Blender APIs are in scene linear color space, as defined by the OpenColorIO configuration. The notable exception is user interface theming colors, which are in sRGB color space.

PARAMETERS:

rgb (*Sequence[float]*) – (red, green, blue) color values where (0, 0, 0) is black & (1, 1, 1) is white.

```
import mathutils

# color values are represented as RGB values from 0 - 1, this is blue
```

```
col = mathutils.Color((0.0, 0.0, 1.0))

# as well as r/g/b attribute access you can adjust them by h/s/v
col.s *= 0.5

# you can access its components by attribute or index
print("Color R:", col.r)
print("Color G:", col[1])
print("Color B:", col[-1])
print("Color HSV: {:.2f}, {:.2f}, {:.2f}".format(*col))


# components of an existing color can be set
col[:] = 0.0, 0.5, 1.0

# components of an existing color can use slice notation to get a tuple
print("Values: {:f}, {:f}, {:f}".format(*col))

# colors can be added and subtracted
col += mathutils.Color((0.25, 0.0, 0.0))

# Color can be multiplied, in this example color is scaled to 0-255
# can printed as integers
print("Color: {:d}, {:d}, {:d}".format(*(int(c) for c in (col * 255.0))))

# This example prints the color as hexadecimal
print("Hexadecimal: {:02x}{:02x}{:02x}".format(int(col.r * 255), int(col.g * 255), int(
```

copy()

Returns a copy of this color.

RETURNS:

A copy of the color.

RETURN TYPE:

`Color`

Note

use this to get a copy of a wrapped color with no reference to the original data.

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

from_aces_to_scene_linear()

Convert from ACES2065-1 linear to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

`Color`

from_rec709_linear_to_scene_linear()

Convert from Rec.709 linear color space to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

`Color`

from_scene_linear_to_aces()

Convert from scene linear to ACES2065-1 linear color space.

RETURNS:

A color in ACES2065-1 linear color space.

RETURN TYPE:

`Color`

from_scene_linear_to_rec709_linear()

Convert from scene linear to Rec.709 linear color space.

RETURNS:

A color in Rec.709 linear color space.

RETURN TYPE:

`Color`

from_scene_linear_to_srgb()

Convert from scene linear to sRGB color space.

RETURNS:

A color in sRGB color space.

RETURN TYPE:

`Color`

from_scene_linear_to_xyz_d65()

Convert from scene linear to CIE XYZ (Illuminant D65) color space.

RETURNS:

A color in XYZ color space.

RETURN TYPE:

`Color`

from_srgb_to_scene_linear()

Convert from sRGB to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

`Color`

from_xyz_d65_to_scene_linear()

Convert from CIE XYZ (Illuminant D65) to scene linear color space.

RETURNS:

A color in scene linear color space.

RETURN TYPE:

`Color`

b

Blue color channel.

TYPE:

float

g

Green color channel.

TYPE:

float

h

HSV Hue component in [0, 1].

TYPE:

float

hsv

HSV Values in [0, 1].

TYPE:

float triplet

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

owner

The item this is wrapping or None (read-only).

r

Red color channel.

TYPE:

float

s

HSV Saturation component in [0, 1].

TYPE:

float

v

HSV Value component in [0, 1].

TYPE:

float

class mathutils.**Euler(angles, order='XYZ')**

This object gives access to Eulers in Blender.

See also

Euler angles on Wikipedia.

PARAMETERS:

- **angles** (*Sequence[float]*) – (X, Y, Z) angles in radians.
- **order** (*str*) – Optional order of the angles, a permutation of `XYZ` .

```
import mathutils
import math

# create a new euler with default axis rotation order
eul = mathutils.Euler((0.0, math.radians(45.0), 0.0), 'XYZ')

# rotate the euler
eul.rotate_axis('Z', math.radians(10.0))

# you can access its components by attribute or index
print("Euler X", eul.x)
print("Euler Y", eul[1])
print("Euler Z", eul[-1])

# components of an existing euler can be set
eul[:] = 1.0, 2.0, 3.0

# components of an existing euler can use slice notation to get a tuple
print("Values: {:f}, {:f}, {:f}".format(*eul))

# the order can be set at any time too
eul.order = 'ZYX'

# eulers can be used to rotate vectors
vec = mathutils.Vector((0.0, 0.0, 1.0))
vec.rotate(eul)

# often its useful to convert the euler into a matrix so it can be used as
# transformations with more flexibility
mat_rot = eul.to_matrix()
mat_loc = mathutils.Matrix.Translation((2.0, 3.0, 4.0))
mat = mat_loc @ mat_rot.to_4x4()
```

copy()

Returns a copy of this euler.

RETURNS:

A copy of the euler.

RETURN TYPE:

`Euler`

> Note
>
> use this to get a copy of a wrapped euler with no reference to the original data.

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

make_compatible(other)

Make this euler compatible with another, so interpolating between them works as intended.

> Note
>
> the rotation order is not taken into account for this function.

rotate(other)

Rotates the euler by another mathutils value.

PARAMETERS:

other (`Euler` | `Quaternion` | `Matrix`) – rotation component of mathutils value

rotate_axis(axis, angle)

Rotates the euler a certain amount and returning a unique euler rotation (no 720 degree pitches).

PARAMETERS:

- **axis** (*str*) – single character in [‘X, ‘Y’, ‘Z’].
- **angle** (*float*) – angle in radians.

to_matrix()

Return a matrix representation of the euler.

RETURNS:

A 3x3 rotation matrix representation of the euler.

RETURN TYPE:

`Matrix`

to_quaternion()

Return a quaternion representation of the euler.

RETURNS:

Quaternion representation of the euler.

RETURN TYPE:

`Quaternion`

zero()

Set all values to zero.

is_frozen

True when this object has been frozen (read-only).

TYPE:
bool

is_valid

True when the owner of this data is valid.

TYPE:
bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:
bool

order

Euler rotation order.

TYPE:
str in [‘XYZ’, ‘XZY’, ‘YXZ’, ‘YZX’, ‘ZXY’, ‘ZYX’]

owner

The item this is wrapping or None (read-only).

x

Euler axis angle in radians.

TYPE:
float

y

Euler axis angle in radians.

TYPE:
float

z

Euler axis angle in radians.

TYPE:
float

class mathutils.**Matrix([rows])**

This object gives access to Matrices in Blender, supporting square and rectangular matrices from 2x2 up to 4x4.

PARAMETERS:
rows (*Sequence[Sequence[float]]*) – Sequence of rows. When omitted, a 4x4 identity matrix is constructed.

```
import mathutils
import math

# Create a location matrix.
mat_loc = mathutils.Matrix.Translation((2.0, 3.0, 4.0))

# Create an identity matrix.
```

```
mat_sca = mathutils.Matrix.Scale(0.5, 4, (0.0, 0.0, 1.0))

# Create a rotation matrix.
mat_rot = mathutils.Matrix.Rotation(math.radians(45.0), 4, 'X')

# Combine transformations.
mat_out = mat_loc @ mat_rot @ mat_sca
print(mat_out)

# Extract components back out of the matrix as two vectors and a quaternion.
loc, rot, sca = mat_out.decompose()
print(loc, rot, sca)

# Recombine extracted components.
mat_out2 = mathutils.Matrix.LocRotScale(loc, rot, sca)
print(mat_out2)

# It can also be useful to access components of a matrix directly.
mat = mathutils.Matrix()
mat[0][0], mat[1][0], mat[2][0] = 0.0, 1.0, 2.0

mat[0][0:3] = 0.0, 1.0, 2.0

# Each item in a matrix is a vector so vector utility functions can be used.
mat[0].xyz = 0.0, 1.0, 2.0
```

classmethod Diagonal(vector)

Create a diagonal (scaling) matrix using the values from the vector.

PARAMETERS:

vector (`Vector`) – The vector of values for the diagonal.

RETURNS:

A diagonal matrix.

RETURN TYPE:

`Matrix`

classmethod Identity(size)

Create an identity matrix.

PARAMETERS:

size (*int*) – The size of the identity matrix to construct [2, 4].

RETURNS:

A new identity matrix.

RETURN TYPE:

`Matrix`

classmethod LocRotScale(location, rotation, scale)

Create a matrix combining translation, rotation and scale, acting as the inverse of the decompose() method.

Any of the inputs may be replaced with None if not needed.

PARAMETERS:

- **location** (`Vector` | None) – The translation component.
- **rotation** (`Matrix` | `Quaternion` | `Euler` | None) – The rotation component as a 3x3 matrix, quaternion, euler or None for n

rotation.

- **scale** (`Vector` | None) – The scale component.

RETURNS:

Combined transformation as a 4x4 matrix.

RETURN TYPE:

`Matrix`

```
# Compute local object transformation matrix:
if obj.rotation_mode == 'QUATERNION':
    matrix = mathutils.Matrix.LocRotScale(obj.location, obj.rotation_quaternion, obj
else:
    matrix = mathutils.Matrix.LocRotScale(obj.location, obj.rotation_euler, obj.scal
```

classmethod OrthoProjection(axis, size)

Create a matrix to represent an orthographic projection.

PARAMETERS:

- **axis** (str | `Vector`) – Can be any of the following: [‘X’, ‘Y’, ‘XY’, ‘XZ’, ‘YZ’], where a single axis is for a 2D matrix. Or a vector fo an arbitrary axis
- **size** (*int*) – The size of the projection matrix to construct [2, 4].

RETURNS:

A new projection matrix.

RETURN TYPE:

`Matrix`

classmethod Rotation(angle, size, axis)

Create a matrix representing a rotation.

PARAMETERS:

- **angle** (*float*) – The angle of rotation desired, in radians.
- **size** (*int*) – The size of the rotation matrix to construct [2, 4].
- **axis** (str | `Vector`) – a string in [‘X’, ‘Y’, ‘Z’] or a 3D Vector Object (optional when size is 2).

RETURNS:

A new rotation matrix.

RETURN TYPE:

`Matrix`

classmethod Scale(factor, size, axis)

Create a matrix representing a scaling.

PARAMETERS:

- **factor** (*float*) – The factor of scaling to apply.
- **size** (*int*) – The size of the scale matrix to construct [2, 4].
- **axis** (`Vector`) – Direction to influence scale. (optional).

RETURNS:

A new scale matrix.

RETURN TYPE:

`Matrix`

classmethod Shear(plane, size, factor)

Create a matrix to represent an shear transformation.

PARAMETERS:

- **plane** (*str*) – Can be any of the following: [‘X’, ‘Y’, ‘XY’, ‘XZ’, ‘YZ’], where a single axis is for a 2D matrix only.
- **size** (*int*) – The size of the shear matrix to construct [2, 4].
- **factor** (*float* | *Sequence[float]*) – The factor of shear to apply. For a 2 *size* matrix use a single float. For a 3 or 4 *size* matrix pass a pa of floats corresponding with the *plane* axis.

RETURNS:

A new shear matrix.

RETURN TYPE:

`Matrix`

classmethod Translation(vector)

Create a matrix representing a translation.

PARAMETERS:

vector (`Vector`) – The translation vector.

RETURNS:

An identity matrix with a translation.

RETURN TYPE:

`Matrix`

adjugate()

Set the matrix to its adjugate.

RAISES:

ValueError – if the matrix cannot be adjugate.

See also

Adjugate matrix on Wikipedia.

adjugated()

Return an adjugated copy of the matrix.

RETURNS:

the adjugated matrix.

RETURN TYPE:

`Matrix`

RAISES:

ValueError – if the matrix cannot be adjugated

copy()

Returns a copy of this matrix.

RETURNS:

an instance of itself

RETURN TYPE:

`Matrix`

decompose()

Return the translation, rotation, and scale components of this matrix.

RETURNS:

RETURNS:

Tuple of translation, rotation, and scale.

RETURN TYPE:

tuple[`Vector` , `Quaternion` , `Vector`]

determinant()

Return the determinant of a matrix.

RETURNS:

Return the determinant of a matrix.

RETURN TYPE:

float

See also

Determinant on Wikipedia.

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

identity()

Set the matrix to the identity matrix.

Note

An object with a location and rotation of zero, and a scale of one will have an identity matrix.

See also

Identity matrix on Wikipedia.

invert(fallback=None)

Set the matrix to its inverse.

PARAMETERS:

fallback (`Matrix`) – Set the matrix to this value when the inverse cannot be calculated (instead of raising a `ValueError` exception).

See also

Inverse matrix on Wikipedia.

invert_safe()

Set the matrix to its inverse, will never error. If degenerated (e.g. zero scale on an axis), add some epsilon to its diagonal, to get an invertible one. If tweaked matrix is still degenerated, set to the identity matrix instead.

See also

Inverse Matrix on Wikipedia.

inverted(fallback=None)

Return an inverted copy of the matrix.

PARAMETERS

PARAMETERS:

fallback (*Any*) – return this when the inverse can't be calculated (instead of raising a `ValueError`).

RETURNS:

The inverted matrix or fallback when given.

RETURN TYPE:

`Matrix` | Any

inverted_safe()

Return an inverted copy of the matrix, will never error. If degenerated (e.g. zero scale on an axis), add some epsilon to its diagonal, to get an invertible one. If tweaked matrix is still degenerated, return the identity matrix instead.

RETURNS:

the inverted matrix.

RETURN TYPE:

`Matrix`

lerp(other, factor)

Returns the interpolation of two matrices. Uses polar decomposition, see "Matrix Animation and Polar Decomposition", Shoemake and Duff, 1992.

PARAMETERS:

- **other** (`Matrix`) – value to interpolate with.
- **factor** (*float*) – The interpolation value in [0.0, 1.0].

RETURNS:

The interpolated matrix.

RETURN TYPE:

`Matrix`

normalize()

Normalize each of the matrix columns.

Note

for 4x4 matrices, the 4th column (translation) is left untouched.

normalized()

Return a column normalized matrix

RETURNS:

a column normalized matrix

RETURN TYPE:

`Matrix`

Note

for 4x4 matrices, the 4th column (translation) is left untouched.

resize_4x4()

Resize the matrix to 4x4.

rotate(other)

Rotates the matrix by another mathutils value.

PARAMETERS:

other (`Euler` | `Quaternion` | `Matrix`) – rotation component of mathutils value

> Note
>
> If any of the columns are not unit length this may not have desired results.

to_2x2()

Return a 2x2 copy of this matrix.

RETURNS:

a new matrix.

RETURN TYPE:

`Matrix`

to_3x3()

Return a 3x3 copy of this matrix.

RETURNS:

a new matrix.

RETURN TYPE:

`Matrix`

to_4x4()

Return a 4x4 copy of this matrix.

RETURNS:

a new matrix.

RETURN TYPE:

`Matrix`

to_euler(order, euler_compat)

Return an Euler representation of the rotation matrix (3x3 or 4x4 matrix only).

PARAMETERS:

- **order** (*str*) – Optional rotation order argument in ['XYZ', 'XZY', 'YXZ', 'YZX', 'ZXY', 'ZYX'].
- **euler_compat** (`Euler`) – Optional euler argument the new euler will be made compatible with (no axis flipping between them). Usefu for converting a series of matrices to animation curves.

RETURNS:

Euler representation of the matrix.

RETURN TYPE:

`Euler`

to_quaternion()

Return a quaternion representation of the rotation matrix.

RETURNS:

Quaternion representation of the rotation matrix.

RETURN TYPE:

`Quaternion`

to_scale()

Return the scale part of a 3x3 or 4x4 matrix.

RETURNS:

Return the scale of a matrix.

RETURN TYPE:

`Vector`

Note

This method does not return a negative scale on any axis because it is not possible to obtain this data from the matrix alone.

to_translation()

Return the translation part of a 4 row matrix.

RETURNS:

Return the translation of a matrix.

RETURN TYPE:

`Vector`

transpose()

Set the matrix to its transpose.

See also

Transpose on Wikipedia.

transposed()

Return a new, transposed matrix.

RETURNS:

a transposed matrix

RETURN TYPE:

`Matrix`

zero()

Set all the matrix values to zero.

col

Access the matrix by columns, 3x3 and 4x4 only, (read-only).

TYPE:

Matrix Access

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_identity

True if this is an identity matrix (read-only).

TYPE:

bool

is_negative

True if this matrix results in a negative scale, 3x3 and 4x4 only, (read-only).

TYPE:

bool

bool

is_orthogonal

True if this matrix is orthogonal, 3x3 and 4x4 only, (read-only).

TYPE:

bool

is_orthogonal_axis_vectors

True if this matrix has got orthogonal axis vectors, 3x3 and 4x4 only, (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

median_scale

The average scale applied to each axis (read-only).

TYPE:

float

owner

The item this is wrapping or None (read-only).

row

Access the matrix by rows (default), (read-only).

TYPE:

Matrix Access

translation

The translation component of the matrix.

TYPE:

`Vector`

class mathutils.**Quaternion([seq[, angle]])**

This object gives access to Quaternions in Blender.

PARAMETERS:

- **seq** (`Vector`) – size 3 or 4
- **angle** (*float*) – rotation angle, in radians

The constructor takes arguments in various forms:

(), *no args*

Create an identity quaternion

(*wxyz*)

Create a quaternion from a `(w, x, y, z)` vector.

(*exponential_map*)

Create a quaternion from a 3d exponential map vector.

> See also
>
> `to_exponential_map()`

(*axis, angle*)

Create a quaternion representing a rotation of *angle* radians over *axis*.

> See also
>
> `to_axis_angle()`

```
import mathutils
import math

# a new rotation 90 degrees about the Y axis
quat_a = mathutils.Quaternion((0.7071068, 0.0, 0.7071068, 0.0))

# passing values to Quaternion's directly can be confusing so axis, angle
# is supported for initializing too
quat_b = mathutils.Quaternion((0.0, 1.0, 0.0), math.radians(90.0))

print("Check quaternions match", quat_a == quat_b)

# like matrices, quaternions can be multiplied to accumulate rotational values
quat_a = mathutils.Quaternion((0.0, 1.0, 0.0), math.radians(90.0))
quat_b = mathutils.Quaternion((0.0, 0.0, 1.0), math.radians(45.0))
quat_out = quat_a @ quat_b

# print the quat, euler degrees for mere mortals and (axis, angle)
print("Final Rotation:")
print(quat_out)
print("{:.2f}, {:.2f}, {:.2f}".format(*(math.degrees(a) for a in quat_out.to_euler())))
print("({:.2f}, {:.2f}, {:.2f}), {:.2f}".format(*quat_out.axis, math.degrees(quat_out.a

# multiple rotations can be interpolated using the exponential map
quat_c = mathutils.Quaternion((1.0, 0.0, 0.0), math.radians(15.0))
exp_avg = (quat_a.to_exponential_map() +
           quat_b.to_exponential_map() +
           quat_c.to_exponential_map()) / 3.0
quat_avg = mathutils.Quaternion(exp_avg)
print("Average rotation:")
print(quat_avg)
```

conjugate()

Set the quaternion to its conjugate (negate x, y, z).

conjugated()

Return a new conjugated quaternion.

RETURNS:

a new quaternion.

RETURN TYPE:

`Quaternion`

copy()

Returns a copy of this quaternion.

RETURNS:

A copy of the quaternion.

RETURN TYPE:

`Quaternion`

Note

use this to get a copy of a wrapped quaternion with no reference to the original data.

cross(other)

Return the cross product of this quaternion and another.

PARAMETERS:

other (`Quaternion`) – The other quaternion to perform the cross product with.

RETURNS:

The cross product.

RETURN TYPE:

`Quaternion`

dot(other)

Return the dot product of this quaternion and another.

PARAMETERS:

other (`Quaternion`) – The other quaternion to perform the dot product with.

RETURNS:

The dot product.

RETURN TYPE:

float

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

identity()

Set the quaternion to an identity quaternion.

invert()

Set the quaternion to its inverse.

inverted()

Return a new, inverted quaternion.

RETURNS:

the inverted value.

RETURN TYPE:

`Quaternion`

make_compatible(other)

Make this quaternion compatible with another, so interpolating between them works as intended.

negate()

Set the quaternion to its negative.

normalize()

Normalize the quaternion.

normalized()

Return a new normalized quaternion.

RETURNS:

a normalized copy.

RETURN TYPE:

`Quaternion`

rotate(other)

Rotates the quaternion by another mathutils value.

PARAMETERS:

other (`Euler` | `Quaternion` | `Matrix`) – rotation component of mathutils value

rotation_difference(other)

Returns a quaternion representing the rotational difference.

PARAMETERS:

other (`Quaternion`) – second quaternion.

RETURNS:

the rotational difference between the two quat rotations.

RETURN TYPE:

`Quaternion`

slerp(other, factor)

Returns the interpolation of two quaternions.

PARAMETERS:

- **other** (`Quaternion`) – value to interpolate with.
- **factor** (*float*) – The interpolation value in [0.0, 1.0].

RETURNS:

The interpolated rotation.

RETURN TYPE:

`Quaternion`

to_axis_angle()

Return the axis, angle representation of the quaternion.

RETURNS:

Axis, angle.

RETURN TYPE:

tuple[Vector , float]

to_euler(order, euler_compat)

Return Euler representation of the quaternion.

PARAMETERS:

- **order** (*str*) – Optional rotation order argument in [‘XYZ’, ‘XZY’, ‘YXZ’, ‘YZX’, ‘ZXY’, ‘ZYX’].
- **euler_compat** (Euler) – Optional euler argument the new euler will be made compatible with (no axis flipping between them). Usefu for converting a series of matrices to animation curves.

RETURNS:

Euler representation of the quaternion.

RETURN TYPE:

Euler

to_exponential_map()

Return the exponential map representation of the quaternion.

This representation consist of the rotation axis multiplied by the rotation angle. Such a representation is useful for interpolation between multipl orientations.

RETURNS:

exponential map.

RETURN TYPE:

Vector of size 3

To convert back to a quaternion, pass it to the Quaternion constructor.

to_matrix()

Return a matrix representation of the quaternion.

RETURNS:

A 3x3 rotation matrix representation of the quaternion.

RETURN TYPE:

Matrix

to_swing_twist(axis)

Split the rotation into a swing quaternion with the specified axis fixed at zero, and the remaining twist rotation angle.

PARAMETERS:

axis (*str*) – Twist axis as a string in [‘X’, ‘Y’, ‘Z’].

RETURNS:

Swing, twist angle.

RETURN TYPE:

tuple[Quaternion , float]

angle

Angle of the quaternion.

TYPE:

float

axis

Quaternion axis as a vector.

TYPE:

`Vector`

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

magnitude

Size of the quaternion (read-only).

TYPE:

float

owner

The item this is wrapping or None (read-only).

w

Quaternion axis value.

TYPE:

float

x

Quaternion axis value.

TYPE:

float

y

Quaternion axis value.

TYPE:

float

z

Quaternion axis value.

TYPE:

float

class mathutils.**Vector(seq)**

This object gives access to Vectors in Blender.

PARAMETERS:

seq (*Sequence[float]*) – Components of the vector, must be a sequence of at least two.

```
import mathutils

# zero length vector
vec = mathutils.Vector((0.0, 0.0, 1.0))

# unit length vector
vec_a = vec.normalized()

vec_b = mathutils.Vector((0.0, 1.0, 2.0))

vec2d = mathutils.Vector((1.0, 2.0))
vec3d = mathutils.Vector((1.0, 0.0, 0.0))
vec4d = vec_a.to_4d()

# Other `mathutils` types.
quat = mathutils.Quaternion()
matrix = mathutils.Matrix()

# Comparison operators can be done on Vector classes:

# (In)equality operators == and != test component values, e.g. 1,2,3 != 3,2,1
vec_a == vec_b
vec_a != vec_b

# Ordering operators >, >=, > and <= test vector length.
vec_a > vec_b
vec_a >= vec_b
vec_a < vec_b
vec_a <= vec_b


# Math can be performed on Vector classes
vec_a + vec_b
vec_a - vec_b
vec_a @ vec_b
vec_a * 10.0
matrix @ vec_a
quat @ vec_a
-vec_a


# You can access a vector object like a sequence
x = vec_a[0]
len(vec)
vec_a[:] = vec_b
vec_a[:] = 1.0, 2.0, 3.0
vec2d[:] = vec3d[:2]


# Vectors support 'swizzle' operations
# See https://en.wikipedia.org/wiki/Swizzling_(computer_graphics)
vec.xyz = vec.zyx
vec.xy = vec4d.zw
vec.xyz = vec4d.wzz
vec4d.wxyz = vec.yxyx
```

classmethod Fill(size, fill=0.0)

Create a vector of length size with all values set to fill.

PARAMETERS:

- **size** (*int*) – The length of the vector to be created.
- **fill** (*float*) – The value used to fill the vector.

classmethod Linspace(start, stop, size)

Create a vector of the specified size which is filled with linearly spaced values between start and stop values.

PARAMETERS:

- **start** (*int*) – The start of the range used to fill the vector.
- **stop** (*int*) – The end of the range used to fill the vector.
- **size** (*int*) – The size of the vector to be created.

classmethod Range(start, stop, step=1)

Create a filled with a range of values.

PARAMETERS:

- **start** (*int*) – The start of the range used to fill the vector.
- **stop** (*int*) – The end of the range used to fill the vector.
- **step** (*int*) – The step between successive values in the vector.

classmethod Repeat(vector, size)

Create a vector by repeating the values in vector until the required size is reached.

PARAMETERS:

- **vector** (`mathutils.Vector`) – The vector to draw values from.
- **size** (*int*) – The size of the vector to be created.

angle(other, fallback=None)

Return the angle between two vectors.

PARAMETERS:

- **other** (`Vector`) – another vector to compare the angle with
- **fallback** (*Any*) – return this when the angle can’t be calculated (zero length vector), (instead of raising a `ValueError`).

RETURNS:

angle in radians or fallback when given

RETURN TYPE:

float | Any

angle_signed(other, fallback=None)

Return the signed angle between two 2D vectors (clockwise is positive).

PARAMETERS:

- **other** (`Vector`) – another vector to compare the angle with
- **fallback** (*Any*) – return this when the angle can’t be calculated (zero length vector), (instead of raising a `ValueError`).

RETURNS:

angle in radians or fallback when given

RETURN TYPE:

float | Any

copy()

Returns a copy of this vector.

RETURNS:

A copy of the vector.

RETURN TYPE:

`Vector`

> Note
>
> use this to get a copy of a wrapped vector with no reference to the original data.

cross(other)

Return the cross product of this vector and another.

PARAMETERS:

other (`Vector`) – The other vector to perform the cross product with.

RETURNS:

The cross product as a vector or a float when 2D vectors are used.

RETURN TYPE:

`Vector` | float

> Note
>
> both vectors must be 2D or 3D

dot(other)

Return the dot product of this vector and another.

PARAMETERS:

other (`Vector`) – The other vector to perform the dot product with.

RETURNS:

The dot product.

RETURN TYPE:

float

freeze()

Make this object immutable.

After this the object can be hashed, used in dictionaries & sets.

RETURNS:

An instance of this object.

lerp(other, factor)

Returns the interpolation of two vectors.

PARAMETERS:

- **other** (`Vector`) – value to interpolate with.
- **factor** (*float*) – The interpolation value in [0.0, 1.0].

RETURNS:

The interpolated vector.

RETURN TYPE:

`Vector`

negate()

Set all values to their negative.

normalize()

Normalize the vector, making the length of the vector always 1.0.

Warning

Normalizing a vector where all values are zero has no effect.

Note

Normalize works for vectors of all sizes, however 4D Vectors w axis is left untouched.

normalized()

Return a new, normalized vector.

RETURNS:

a normalized copy of the vector

RETURN TYPE:

`Vector`

orthogonal()

Return a perpendicular vector.

RETURNS:

a new vector 90 degrees from this vector.

RETURN TYPE:

`Vector`

Note

the axis is undefined, only use when any orthogonal vector is acceptable.

project(other)

Return the projection of this vector onto the *other*.

PARAMETERS:

other (`Vector`) – second vector.

RETURNS:

the parallel projection vector

RETURN TYPE:

`Vector`

reflect(mirror)

Return the reflection vector from the *mirror* argument.

PARAMETERS:

mirror (`Vector`) – This vector could be a normal from the reflecting surface.

RETURNS:

The reflected vector matching the size of this vector.

RETURN TYPE:

`Vector`

resize(size=3)

Resize the vector to have size number of elements.

resize_2d()

Resize the vector to 2D (x, y).

resize_3d()

Resize the vector to 3D (x, y, z).

resize_4d()

Resize the vector to 4D (x, y, z, w).

resized(size=3)

Return a resized copy of the vector with size number of elements.

RETURNS:

a new vector

RETURN TYPE:

`Vector`

rotate(other)

Rotate the vector by a rotation value.

> Note
>
> 2D vectors are a special case that can only be rotated by a 2x2 matrix.

PARAMETERS:

other (`Euler` | `Quaternion` | `Matrix`) – rotation component of mathutils value

rotation_difference(other)

Returns a quaternion representing the rotational difference between this vector and another.

PARAMETERS:

other (`Vector`) – second vector.

RETURNS:

the rotational difference between the two vectors.

RETURN TYPE:

`Quaternion`

> Note
>
> 2D vectors raise an `AttributeError`.

slerp(other, factor, fallback=None)

Returns the interpolation of two non-zero vectors (spherical coordinates).

PARAMETERS:

- **other** (`Vector`) – value to interpolate with.
- **factor** (*float*) – The interpolation value typically in [0.0, 1.0].
- **fallback** (*Any*) – return this when the vector can’t be calculated (zero length vector or direct opposites), (instead of raising a `ValueError`).

RETURNS:

The interpolated vector.

RETURN TYPE:

`Vector`

to_2d()

Return a 2d copy of the vector.

RETURNS:

a new vector

RETURN TYPE:

`Vector`

to_3d()

Return a 3d copy of the vector.

RETURNS:

a new vector

RETURN TYPE:

`Vector`

to_4d()

Return a 4d copy of the vector.

RETURNS:

a new vector

RETURN TYPE:

`Vector`

to_track_quat(track, up)

Return a quaternion rotation from the vector and the track and up axis.

PARAMETERS:

- **track** (*str*) – Track axis in [‘X’, ‘Y’, ‘Z’, ‘-X’, ‘-Y’, ‘-Z’].
- **up** (*str*) – Up axis in [‘X’, ‘Y’, ‘Z’].

RETURNS:

rotation from the vector and the track and up axis.

RETURN TYPE:

`Quaternion`

to_tuple(precision=-1)

Return this vector as a tuple with.

PARAMETERS:

precision (*int*) – The number to round the value to in [-1, 21].

RETURNS:

the values of the vector rounded by *precision*

RETURN TYPE:

tuple[float, …]

zero()

Set all values to zero.

is_frozen

True when this object has been frozen (read-only).

TYPE:

bool

is_valid

True when the owner of this data is valid.

TYPE:

bool

is_wrapped

True when this object wraps external data (read-only).

TYPE:

bool

length

Vector Length.

TYPE:

float

length_squared

Vector length squared (v.dot(v)).

TYPE:

float

magnitude

Vector Length.

TYPE:

float

owner

The item this is wrapping or None (read-only).

w

Vector W axis (4D Vectors only).

TYPE:

float

ww

TYPE:

`Vector`

www

TYPE:

`Vector`

wwww

TYPE:

`Vector`

wwwx

TYPE:

`Vector`

wwwy

TYPE:

`Vector`

wwwz

TYPE:

`Vector`

wwx

TYPE:

`Vector`

wwxw

TYPE:

`Vector`

wwxx

TYPE:

`Vector`

wwxy

TYPE:

`Vector`

wwxz

TYPE:

`Vector`

wwy

TYPE:

`Vector`

wwyw

TYPE:

`Vector`

wwyx

TYPE:

`Vector`

wwyy

TYPE:

`Vector`

wwyz

TYPE:

`Vector`

-----

wwz

TYPE:

`Vector`

wwzw

TYPE:

`Vector`

wwzx

TYPE:

`Vector`

wwzy

TYPE:

`Vector`

wwzz

TYPE:

`Vector`

wx

TYPE:

`Vector`

wxw

TYPE:

`Vector`

wxww

TYPE:

`Vector`

wxwx

TYPE:

`Vector`

wxwy

TYPE:

`Vector`

wxwz

TYPE:

`Vector`

wxx

TYPE:

`Vector`

wxxw

TYPE:

`Vector`

wxxx

TYPE:

`Vector`

wxxy

TYPE:

`Vector`

wxxz

TYPE:

`Vector`

wxy

TYPE:

`Vector`

wxyw

TYPE:

`Vector`

wxyx

TYPE:

`Vector`

wxyy

TYPE:

`Vector`

wxyz

TYPE:

`Vector`

wxz

TYPE:

`Vector`

wxzw

TYPE:

`Vector`

wxzx

TYPE:

`Vector`

wxzy

TYPE:

`Vector`

wxzz

TYPE:

`Vector`

wy

TYPE:

`Vector`

vector

wyw

TYPE:

Vector

wyww

TYPE:

Vector

wywx

TYPE:

Vector

wywy

TYPE:

Vector

wywz

TYPE:

Vector

wyx

TYPE:

Vector

wyxw

TYPE:

Vector

wyxx

TYPE:

Vector

wyxy

TYPE:

Vector

wyxz

TYPE:

Vector

wyy

TYPE:

Vector

wyyw

TYPE:

Vector

wyyx

TYPE:

Vector

wyyy

TYPE:

`Vector`

wyyz

TYPE:

`Vector`

wyz

TYPE:

`Vector`

wyzw

TYPE:

`Vector`

wyzx

TYPE:

`Vector`

wyzy

TYPE:

`Vector`

wyzz

TYPE:

`Vector`

wz

TYPE:

`Vector`

wzw

TYPE:

`Vector`

wzww

TYPE:

`Vector`

wzwx

TYPE:

`Vector`

wzwy

TYPE:

`Vector`

wzwz

TYPE:

`Vector`

wzx

TYPE:

`Vector`

wzxw

TYPE:

`Vector`

wzxx

TYPE:

`Vector`

wzxy

TYPE:

`Vector`

wzxz

TYPE:

`Vector`

wzy

TYPE:

`Vector`

wzyw

TYPE:

`Vector`

wzyx

TYPE:

`Vector`

wzyy

TYPE:

`Vector`

wzyz

TYPE:

`Vector`

wzz

TYPE:

`Vector`

wzzw

TYPE:

`Vector`

wzzx

TYPE:

`Vector`

wzzy

TYPE:

`Vector`

wzzz

TYPE:

`Vector`

x

Vector X axis.

TYPE:

float

xw

TYPE:

`Vector`

xww

TYPE:

`Vector`

xwww

TYPE:

`Vector`

xwwx

TYPE:

`Vector`

xwwy

TYPE:

`Vector`

xwwz

TYPE:

`Vector`

xwx

TYPE:

`Vector`

xwxw

TYPE:

`Vector`

xwxx

TYPE:

`Vector`

xwxy

TYPE:

`Vector`

xwxz

TYPE:

TYPE:

Vector

xwy

TYPE:

Vector

xwyw

TYPE:

Vector

xwyx

TYPE:

Vector

xwyy

TYPE:

Vector

xwyz

TYPE:

Vector

xwz

TYPE:

Vector

xwzw

TYPE:

Vector

xwzx

TYPE:

Vector

xwzy

TYPE:

Vector

xwzz

TYPE:

Vector

xx

TYPE:

Vector

xxw

TYPE:

Vector

xxww

TYPE:

Vector

xxwx

TYPE:

`Vector`

xxwy

TYPE:

`Vector`

xxwz

TYPE:

`Vector`

xxx

TYPE:

`Vector`

xxxw

TYPE:

`Vector`

xxxx

TYPE:

`Vector`

xxxy

TYPE:

`Vector`

xxxz

TYPE:

`Vector`

xxy

TYPE:

`Vector`

xxyw

TYPE:

`Vector`

xxyx

TYPE:

`Vector`

xxyy

TYPE:

`Vector`

xxyz

TYPE:

`Vector`

xxz

TYPE:

`Vector`

xxzw

TYPE:

`Vector`

xxzx

TYPE:

`Vector`

xxzy

TYPE:

`Vector`

xxzz

TYPE:

`Vector`

xy

TYPE:

`Vector`

xyw

TYPE:

`Vector`

xyww

TYPE:

`Vector`

xywx

TYPE:

`Vector`

xywy

TYPE:

`Vector`

xywz

TYPE:

`Vector`

xyx

TYPE:

`Vector`

xyxw

TYPE:

`Vector`

xyxx

TYPE:

`Vector`

xyxy

TYPE:

`Vector`

xyxz

TYPE:

`Vector`

xyy

TYPE:

`Vector`

xyyw

TYPE:

`Vector`

xyyx

TYPE:

`Vector`

xyyy

TYPE:

`Vector`

xyyz

TYPE:

`Vector`

xyz

TYPE:

`Vector`

xyzw

TYPE:

`Vector`

xyzx

TYPE:

`Vector`

xyzy

TYPE:

`Vector`

xyzz

TYPE:

`Vector`

xz

TYPE:

`Vector`

xzw

TYPE:

`Vector`

xzww

TYPE:

`Vector`

xzwx

TYPE:

`Vector`

xzwy

TYPE:

`Vector`

xzwz

TYPE:

`Vector`

xzx

TYPE:

`Vector`

xzxw

TYPE:

`Vector`

xzxx

TYPE:

`Vector`

xzxy

TYPE:

`Vector`

xzxz

TYPE:

`Vector`

xzy

TYPE:

`Vector`

xzyw

TYPE:

`Vector`

xzyx

TYPE:

`Vector`

xzyy

TYPE:

`Vector`

xzyz

TYPE:

`Vector`

xzz

TYPE:

`Vector`

xzzw

TYPE:

`Vector`

xzzx

TYPE:

`Vector`

xzzy

TYPE:

`Vector`

xzzz

TYPE:

`Vector`

y

Vector Y axis.

TYPE:

float

yw

TYPE:

`Vector`

yww

TYPE:

`Vector`

ywww

TYPE:

`Vector`

ywwx

TYPE:

`Vector`

ywwy

TYPE:

`Vector`

ywwz

TYPE:

`Vector`

ywx

TYPE:

`Vector`

ywxw

TYPE:

`Vector`

ywxx

TYPE:

`Vector`

ywxy

TYPE:

`Vector`

ywxz

TYPE:

`Vector`

ywy

TYPE:

`Vector`

ywyw

TYPE:

`Vector`

ywyx

TYPE:

`Vector`

ywyy

TYPE:

`Vector`

ywyz

TYPE:

`Vector`

ywz

TYPE:

`Vector`

ywzw

TYPE:

`Vector`

ywzx

TYPE:

`Vector`

ywzy

TYPE:

`Vector`

ywzz

TYPE:

`Vector`

yx

TYPE:

`Vector`

yxw

TYPE:

`Vector`

yxww

TYPE:

`Vector`

yxwx

TYPE:

`Vector`

yxwy

TYPE:

`Vector`

yxwz

TYPE:

`Vector`

yxx

TYPE:

`Vector`

yxxw

TYPE:

`Vector`

yxxx

TYPE:

`Vector`

yxxy

TYPE:

`Vector`

yxxz

TYPE:

`Vector`

yxy

TYPE:

`Vector`

yxyw

TYPE:

`Vector`

yxyx

TYPE:

`Vector`

yxyy

TYPE:

`Vector`

yxyz

TYPE:

`Vector`

yxz

TYPE:

`Vector`

yxzw

TYPE:

`Vector`

yxzx

TYPE:

`Vector`

yxzy

TYPE:

`Vector`

yxzz

TYPE:

`Vector`

yy

TYPE:

`Vector`

yyw

TYPE:

`Vector`

yyww

TYPE:

`Vector`

yywx

TYPE:

`Vector`

yywy

TYPE:

`Vector`

yywz

TYPE:

`Vector`

yyx

TYPE:

`Vector`

yyxw

TYPE:

`Vector`

yyxx

TYPE:

`Vector`

yyxy

TYPE:

`Vector`

yyxz

TYPE:

`Vector`

yyy

TYPE:

`Vector`

yyyw

TYPE:

`Vector`

yyyx

TYPE:

`Vector`

yyyy

TYPE:

`Vector`

yyyz

TYPE:

`Vector`

yyz

TYPE:

`Vector`

yyzw

TYPE:

`Vector`

yyzx

TYPE:

`Vector`

yyzy

TYPE:

`Vector`

yyzz

TYPE:

`Vector`

yz

TYPE:

`Vector`

yzw

TYPE:

`Vector`

yzww

TYPE:

`Vector`

yzwx

TYPE:

`Vector`

yzwy

TYPE:

`Vector`

yzwz

TYPE:

`Vector`

yzx

TYPE:

`Vector`

yzxw

TYPE:

`Vector`

yzxx

TYPE:

`Vector`

yzxy

TYPE:

`Vector`

yzxz

TYPE:

`Vector`

yzy

TYPE:

`Vector`

yzyw

TYPE:

`Vector`

yzyx

TYPE:

`Vector`

yzyy

TYPE:

`Vector`

yzyz

TYPE:

`Vector`

yzz

TYPE:

`Vector`

yzzw

TYPE:

`Vector`

yzzx

TYPE:

`Vector`

yzzy

TYPE:

`Vector`

yzzz

TYPE:

`Vector`

z

Vector Z axis (3D Vectors only).

TYPE:

float

zw

TYPE:

Vector

zww

TYPE:

Vector

zwww

TYPE:

Vector

zwwx

TYPE:

Vector

zwwy

TYPE:

Vector

zwwz

TYPE:

Vector

zwx

TYPE:

Vector

zwxw

TYPE:

Vector

zwxx

TYPE:

Vector

zwxy

TYPE:

Vector

zwxz

TYPE:

Vector

zwy

TYPE:

Vector

zwyw

TYPE:

`Vector`

zwyx

TYPE:

`Vector`

zwyy

TYPE:

`Vector`

zwyz

TYPE:

`Vector`

zwz

TYPE:

`Vector`

zwzw

TYPE:

`Vector`

zwzx

TYPE:

`Vector`

zwzy

TYPE:

`Vector`

zwzz

TYPE:

`Vector`

zx

TYPE:

`Vector`

zxw

TYPE:

`Vector`

zxww

TYPE:

`Vector`

zxwx

TYPE:

`Vector`

zxwy

TYPE:

`vector`

zxwz

TYPE:

`Vector`

zxx

TYPE:

`Vector`

zxxw

TYPE:

`Vector`

zxxx

TYPE:

`Vector`

zxxy

TYPE:

`Vector`

zxxz

TYPE:

`Vector`

zxy

TYPE:

`Vector`

zxyw

TYPE:

`Vector`

zxyx

TYPE:

`Vector`

zxyy

TYPE:

`Vector`

zxyz

TYPE:

`Vector`

zxz

TYPE:

`Vector`

zxzw

TYPE:

`Vector`

zxzx

TYPE:

`Vector`

zxzy

TYPE:

`Vector`

zxzz

TYPE:

`Vector`

zy

TYPE:

`Vector`

zyw

TYPE:

`Vector`

zyww

TYPE:

`Vector`

zywx

TYPE:

`Vector`

zywy

TYPE:

`Vector`

zywz

TYPE:

`Vector`

zyx

TYPE:

`Vector`

zyxw

TYPE:

`Vector`

zyxx

TYPE:

`Vector`

zyxy

TYPE:

`Vector`

zyxz

TYPE:

`Vector`

zyy

TYPE:

`Vector`

zyyw

TYPE:

`Vector`

zyyx

TYPE:

`Vector`

zyyy

TYPE:

`Vector`

zyyz

TYPE:

`Vector`

zyz

TYPE:

`Vector`

zyzw

TYPE:

`Vector`

zyzx

TYPE:

`Vector`

zyzy

TYPE:

`Vector`

zyzz

TYPE:

`Vector`

zz

TYPE:

`Vector`

zzw

TYPE:

`Vector`

zzww

TYPE:

`Vector`

zzwx

TYPE:

`Vector`

zzwy

TYPE:

`Vector`

zzwz

TYPE:

`Vector`

zzx

TYPE:

`Vector`

zzxw

TYPE:

`Vector`

zzxx

TYPE:

`Vector`

zzxy

TYPE:

`Vector`

zzxz

TYPE:

`Vector`

zzy

TYPE:

`Vector`

zzyw

TYPE:

`Vector`

zzyx

TYPE:

`Vector`

zzyy

TYPE:

`Vector`

zzyz

TYPE:

`Vector`

zzz

TYPE:

`Vector`

zzzw

TYPE:

`Vector`

zzzx

TYPE:

`Vector`

zzzy

TYPE:

`Vector`

zzzz

TYPE:

`Vector`