Numpy stl Documentation

Release 2.7.0

Rick van Hattem

Jun 25, 2018

Contents

Contents:

CHAPTER 1

numpy-stl

Simple library to make working with STL files (and 3D objects in general) fast and easy.

Due to all operations heavily relying on *numpy* this is one of the fastest STL editing libraries for Python available.

1.1 Links

- The source: https://github.com/WoLpH/numpy-stl
- Project page: https://pypi.python.org/pypi/numpy-stl
- Reporting bugs: https://github.com/WoLpH/numpy-stl/issues
- Documentation: http://numpy-stl.readthedocs.org/en/latest/
- My blog: https://wol.ph/

1.2 Requirements for installing:

- numpy any recent version
- python-utils version 1.6 or greater

1.3 Installation:

pip install numpy-stl

1.4 Initial usage:

- *stl2bin your_ascii_stl_file.stl new_binary_stl_file.stl*
- *stl2ascii your_binary_stl_file.stl new_ascii_stl_file.stl*
- *stl your_ascii_stl_file.stl new_binary_stl_file.stl*

1.5 Contributing:

Contributions are always welcome. Please view the guidelines to get started: https://github.com/WoLpH/numpy-stl/blob/develop/CONTRIBUTING.rst

1.6 Quickstart

```
import numpy
from stl import mesh

# Using an existing stl file:
your_mesh = mesh.Mesh.from_file('some_file.stl')

# Or creating a new mesh (make sure not to overwrite the `mesh` import by
# naming it `mesh`):
VERTICE_COUNT = 100
data = numpy.zeros(VERTICE_COUNT, dtype=mesh.Mesh.dtype)
your_mesh = mesh.Mesh(data, remove_empty_areas=False)

# The mesh normals (calculated automatically)
your_mesh.normals
# The mesh vectors
your_mesh.v0, your_mesh.v1, your_mesh.v2
# Accessing individual points (concatenation of v0, v1 and v2 in triplets)
assert (your_mesh.points[0][0:3] == your_mesh.v0[0]).all()
assert (your_mesh.points[0][3:6] == your_mesh.v1[0]).all()
assert (your_mesh.points[0][6:9] == your_mesh.v2[0]).all()
assert (your_mesh.points[1][0:3] == your_mesh.v0[1]).all()

your_mesh.save('new_stl_file.stl')
```

1.7 Plotting using matplotlib is equally easy:

```
from stl import mesh
from mpl_toolkits import mplot3d
from matplotlib import pyplot

# Create a new plot
figure = pyplot.figure()
axes = mplot3d.Axes3D(figure)

# Load the STL files and add the vectors to the plot
```

(continues on next page)

(continued from previous page)

```
your_mesh = mesh.Mesh.from_file('tests/stl_binary/HalfDonut.stl')
axes.add_collection3d(mplot3d.art3d.Poly3DCollection(your_mesh.vectors))

# Auto scale to the mesh size
scale = your_mesh.points.flatten(-1)
axes.auto_scale_xyz(scale, scale, scale)

# Show the plot to the screen
pyplot.show()
```

1.8 Modifying Mesh objects

```
from stl import mesh
import math
import numpy

# Create 3 faces of a cube
data = numpy.zeros(6, dtype=mesh.Mesh.dtype)

# Top of the cube
data['vectors'][0] = numpy.array([[0, 1, 1],
                                  [1, 0, 1],
                                  [0, 0, 1]])
data['vectors'][1] = numpy.array([[1, 0, 1],
                                  [0, 1, 1],
                                  [1, 1, 1]])
# Right face
data['vectors'][2] = numpy.array([[1, 0, 0],
                                  [1, 0, 1],
                                  [1, 1, 0]])
data['vectors'][3] = numpy.array([[1, 1, 1],
                                  [1, 0, 1],
                                  [1, 1, 0]])
# Left face
data['vectors'][4] = numpy.array([[0, 0, 0],
                                  [1, 0, 0],
                                  [1, 0, 1]])
data['vectors'][5] = numpy.array([[0, 0, 0],
                                  [0, 0, 1],
                                  [1, 0, 1]])

# Since the cube faces are from 0 to 1 we can move it to the middle by
# substracting .5
data['vectors'] -= .5

# Generate 4 different meshes so we can rotate them later
meshes = [mesh.Mesh(data.copy()) for _ in range(4)]

# Rotate 90 degrees over the Y axis
meshes[0].rotate([0.0, 0.5, 0.0], math.radians(90))

# Translate 2 points over the X axis
meshes[1].x += 2
```

(continues on next page)

(continued from previous page)

```
# Rotate 90 degrees over the X axis
meshes[2].rotate([0.5, 0.0, 0.0], math.radians(90))
# Translate 2 points over the X and Y points
meshes[2].x += 2
meshes[2].y += 2

# Rotate 90 degrees over the X and Y axis
meshes[3].rotate([0.5, 0.0, 0.0], math.radians(90))
meshes[3].rotate([0.0, 0.5, 0.0], math.radians(90))
# Translate 2 points over the Y axis
meshes[3].y += 2


# Optionally render the rotated cube faces
from matplotlib import pyplot
from mpl_toolkits import mplot3d

# Create a new plot
figure = pyplot.figure()
axes = mplot3d.Axes3D(figure)

# Render the cube faces
for m in meshes:
    axes.add_collection3d(mplot3d.art3d.Poly3DCollection(m.vectors))

# Auto scale to the mesh size
scale = numpy.concatenate([m.points for m in meshes]).flatten(-1)
axes.auto_scale_xyz(scale, scale, scale)

# Show the plot to the screen
pyplot.show()
```

1.9 Extending Mesh objects

```
from stl import mesh
import math
import numpy

# Create 3 faces of a cube
data = numpy.zeros(6, dtype=mesh.Mesh.dtype)

# Top of the cube
data['vectors'][0] = numpy.array([[0, 1, 1],
                                  [1, 0, 1],
                                  [0, 0, 1]])
data['vectors'][1] = numpy.array([[1, 0, 1],
                                  [0, 1, 1],
                                  [1, 1, 1]])
# Right face
data['vectors'][2] = numpy.array([[1, 0, 0],
                                  [1, 0, 1],
                                  [1, 1, 0]])
data['vectors'][3] = numpy.array([[1, 1, 1],
                                  [1, 0, 1],
```

(continues on next page)

(continued from previous page)

```
                                  [1, 1, 0]])
# Left face
data['vectors'][4] = numpy.array([[0, 0, 0],
                                  [1, 0, 0],
                                  [1, 0, 1]])
data['vectors'][5] = numpy.array([[0, 0, 0],
                                  [0, 0, 1],
                                  [1, 0, 1]])

# Since the cube faces are from 0 to 1 we can move it to the middle by
# substracting .5
data['vectors'] -= .5

cube_back = mesh.Mesh(data.copy())
cube_front = mesh.Mesh(data.copy())

# Rotate 90 degrees over the X axis followed by the Y axis followed by the
# X axis
cube_back.rotate([0.5, 0.0, 0.0], math.radians(90))
cube_back.rotate([0.0, 0.5, 0.0], math.radians(90))
cube_back.rotate([0.5, 0.0, 0.0], math.radians(90))

cube = mesh.Mesh(numpy.concatenate([
    cube_back.data.copy(),
    cube_front.data.copy(),
]))

# Optionally render the rotated cube faces
from matplotlib import pyplot
from mpl_toolkits import mplot3d

# Create a new plot
figure = pyplot.figure()
axes = mplot3d.Axes3D(figure)

# Render the cube
axes.add_collection3d(mplot3d.art3d.Poly3DCollection(cube.vectors))

# Auto scale to the mesh size
scale = cube_back.points.flatten(-1)
axes.auto_scale_xyz(scale, scale, scale)

# Show the plot to the screen
pyplot.show()
```

1.10 Creating Mesh objects from a list of vertices and faces

```
import numpy as np
from stl import mesh

# Define the 8 vertices of the cube
vertices = np.array([\
    [-1, -1, -1],
    [+1, -1, -1],
```

(continues on next page)

(continued from previous page)

```
    [+1, +1, -1],
    [-1, +1, -1],
    [-1, -1, +1],
    [+1, -1, +1],
    [+1, +1, +1],
    [-1, +1, +1]])
# Define the 12 triangles composing the cube
faces = np.array([\
    [0,3,1],
    [1,3,2],
    [0,4,7],
    [0,7,3],
    [4,5,6],
    [4,6,7],
    [5,1,2],
    [5,2,6],
    [2,3,6],
    [3,7,6],
    [0,1,5],
    [0,5,4]])

# Create the mesh
cube = mesh.Mesh(np.zeros(faces.shape[0], dtype=mesh.Mesh.dtype))
for i, f in enumerate(faces):
    for j in range(3):
        cube.vectors[i][j] = vertices[f[j],:]

# Write the mesh to file "cube.stl"
cube.save('cube.stl')
```

1.11 Evaluating Mesh properties (Volume, Center of gravity, Inertia)

```
import numpy as np
from stl import mesh

# Using an existing closed stl file:
your_mesh = mesh.Mesh.from_file('some_file.stl')

volume, cog, inertia = your_mesh.get_mass_properties()
print("Volume                                  = {0}".format(volume))
print("Position of the center of gravity (COG) = {0}".format(cog))
print("Inertia matrix at expressed at the COG  = {0}".format(inertia[0,:]))
print("                                          {0}".format(inertia[1,:]))
print("                                          {0}".format(inertia[2,:]))
```

1.12 Combining multiple STL files

```
import math
import stl
from stl import mesh
import numpy
```

(continues on next page)

(continued from previous page)

```
# find the max dimensions, so we can know the bounding box, getting the height,
# width, length (because these are the step size)...
def find_mins_maxs(obj):
    minx = maxx = miny = maxy = minz = maxz = None
    for p in obj.points:
        # p contains (x, y, z)
        if minx is None:
            minx = p[stl.Dimension.X]
            maxx = p[stl.Dimension.X]
            miny = p[stl.Dimension.Y]
            maxy = p[stl.Dimension.Y]
            minz = p[stl.Dimension.Z]
            maxz = p[stl.Dimension.Z]
        else:
            maxx = max(p[stl.Dimension.X], maxx)
            minx = min(p[stl.Dimension.X], minx)
            maxy = max(p[stl.Dimension.Y], maxy)
            miny = min(p[stl.Dimension.Y], miny)
            maxz = max(p[stl.Dimension.Z], maxz)
            minz = min(p[stl.Dimension.Z], minz)
    return minx, maxx, miny, maxy, minz, maxz


def translate(_solid, step, padding, multiplier, axis):
    if 'x' == axis:
        items = 0, 3, 6
    elif 'y' == axis:
        items = 1, 4, 7
    elif 'z' == axis:
        items = 2, 5, 8
    else:
        raise RuntimeError('Unknown axis %r, expected x, y or z' % axis)

    # _solid.points.shape == [:, ((x, y, z), (x, y, z), (x, y, z))]
    _solid.points[:, items] += (step * multiplier) + (padding * multiplier)


def copy_obj(obj, dims, num_rows, num_cols, num_layers):
    w, l, h = dims
    copies = []
    for layer in range(num_layers):
        for row in range(num_rows):
            for col in range(num_cols):
                # skip the position where original being copied is
                if row == 0 and col == 0 and layer == 0:
                    continue
                _copy = mesh.Mesh(obj.data.copy())
                # pad the space between objects by 10% of the dimension being
                # translated
                if col != 0:
                    translate(_copy, w, w / 10., col, 'x')
                if row != 0:
                    translate(_copy, l, l / 10., row, 'y')
                if layer != 0:
                    translate(_copy, h, h / 10., layer, 'z')
```

(continues on next page)

(continued from previous page)

```
                copies.append(_copy)
    return copies

# Using an existing stl file:
main_body = mesh.Mesh.from_file('ball_and_socket_simplified_-_main_body.stl')

# rotate along Y
main_body.rotate([0.0, 0.5, 0.0], math.radians(90))

minx, maxx, miny, maxy, minz, maxz = find_mins_maxs(main_body)
w1 = maxx - minx
l1 = maxy - miny
h1 = maxz - minz
copies = copy_obj(main_body, (w1, l1, h1), 2, 2, 1)

# I wanted to add another related STL to the final STL
twist_lock = mesh.Mesh.from_file('ball_and_socket_simplified_-_twist_lock.stl')
minx, maxx, miny, maxy, minz, maxz = find_mins_maxs(twist_lock)
w2 = maxx - minx
l2 = maxy - miny
h2 = maxz - minz
translate(twist_lock, w1, w1 / 10., 3, 'x')
copies2 = copy_obj(twist_lock, (w2, l2, h2), 2, 2, 1)
combined = mesh.Mesh(numpy.concatenate([main_body.data, twist_lock.data] +
                                    [copy.data for copy in copies] +
                                    [copy.data for copy in copies2]))

combined.save('combined.stl', mode=stl.Mode.ASCII)  # save as ASCII
```

CHAPTER 2

tests and examples

2.1 tests.stl_corruption module

```
from __future__ import print_function
import numpy
import pytest
import struct

from stl import mesh

_STL_FILE = '''
solid test.stl
facet normal -0.014565 0.073223 -0.002897
  outer loop
    vertex 0.399344 0.461940 1.044090
    vertex 0.500000 0.500000 1.500000
    vertex 0.576120 0.500000 1.117320
  endloop
endfacet
endsolid test.stl
'''.lstrip()


def test_valid_ascii(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('w+') as fh:
        fh.write(_STL_FILE)
        fh.seek(0)
        mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)


def test_ascii_with_missing_name(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('w+') as fh:
```

(continues on next page)

(continued from previous page)

```
        # Split the file into lines
        lines = _STL_FILE.splitlines()

        # Remove everything except solid
        lines[0] = lines[0].split()[0]

        # Join the lines to test files that start with solid without space
        fh.write('\n'.join(lines))
        fh.seek(0)
        mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)


def test_ascii_with_blank_lines(tmpdir, speedups):
    _stl_file = '''
    solid test.stl


      facet normal -0.014565 0.073223 -0.002897

        outer loop

          vertex 0.399344 0.461940 1.044090
          vertex 0.500000 0.500000 1.500000

          vertex 0.576120 0.500000 1.117320

        endloop

      endfacet

    endsolid test.stl
    '''.lstrip()

    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('w+') as fh:
        fh.write(_stl_file)
        fh.seek(0)
        mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)


def test_incomplete_ascii_file(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('w+') as fh:
        fh.write('solid some_file.stl')
        fh.seek(0)
        with pytest.raises(AssertionError):
            mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)

    for offset in (-20, 82, 100):
        with tmp_file.open('w+') as fh:
            fh.write(_STL_FILE[:-offset])
            fh.seek(0)
            with pytest.raises(AssertionError):
                mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)


def test_corrupt_ascii_file(tmpdir, speedups):
```

(continues on next page)

(continued from previous page)

```
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('w+') as fh:
        fh.write(_STL_FILE)
        fh.seek(40)
        print('####\n' * 100, file=fh)
        fh.seek(0)
        with pytest.raises(AssertionError):
            mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)

    with tmp_file.open('w+') as fh:
        fh.write(_STL_FILE)
        fh.seek(40)
        print(' ' * 100, file=fh)
        fh.seek(80)
        fh.write(struct.pack('<i', 10).decode('utf-8'))
        fh.seek(0)
        with pytest.raises(AssertionError):
            mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)


def test_corrupt_binary_file(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('w+') as fh:
        fh.write('#########\n' * 8)
        fh.write('#\0\0\0')
        fh.seek(0)
        mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)

    with tmp_file.open('w+') as fh:
        fh.write('#########\n' * 9)
        fh.seek(0)
        with pytest.raises(AssertionError):
            mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)

    with tmp_file.open('w+') as fh:
        fh.write('#########\n' * 8)
        fh.write('#\0\0\0')
        fh.seek(0)
        fh.write('solid test.stl')
        fh.seek(0)
        mesh.Mesh.from_file(str(tmp_file), fh=fh, speedups=speedups)


def test_duplicate_polygons():
    data = numpy.zeros(3, dtype=mesh.Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [1, 0, 0],
                                      [0, 1, 1.]])
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [2, 0, 0],
                                      [0, 2, 1.]])
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [3, 0, 0],
                                      [0, 3, 1.]])

    assert not mesh.Mesh(data, remove_empty_areas=False).check()
```

2.2 tests.test_commandline module

```
import sys

from stl import main


def test_main(ascii_file, binary_file, tmpdir, speedups):
    original_argv = sys.argv[:]
    args_pre = ['stl']
    args_post = [str(tmpdir.join('output.stl'))]

    if not speedups:
        args_pre.append('-s')

    try:
        sys.argv[:] = args_pre + [ascii_file] + args_post
        main.main()
        sys.argv[:] = args_pre + ['-r', ascii_file] + args_post
        main.main()
        sys.argv[:] = args_pre + ['-a', binary_file] + args_post
        main.main()
        sys.argv[:] = args_pre + ['-b', ascii_file] + args_post
        main.main()
    finally:
        sys.argv[:] = original_argv


def test_args(ascii_file, tmpdir):
    parser = main._get_parser('')

    def _get_name(*args):
        return main._get_name(parser.parse_args(list(map(str, args))))

    assert _get_name('--name', 'foobar') == 'foobar'
    assert _get_name('-', tmpdir.join('binary.stl')).endswith('binary.stl')
    assert _get_name(ascii_file, '-').endswith('HalfDonut.stl')
    assert _get_name('-', '-')


def test_ascii(binary_file, tmpdir, speedups):
    original_argv = sys.argv[:]
    try:
        sys.argv[:] = [
            'stl',
            '-s' if not speedups else '',
            binary_file,
            str(tmpdir.join('ascii.stl')),
        ]
        try:
            main.to_ascii()
        except SystemExit:
            pass
    finally:
        sys.argv[:] = original_argv
```

(continues on next page)

(continued from previous page)

```
def test_binary(ascii_file, tmpdir, speedups):
    original_argv = sys.argv[:]
    try:
        sys.argv[:] = [
            'stl',
            '-s' if not speedups else '',
            ascii_file,
            str(tmpdir.join('binary.stl')),
        ]
        try:
            main.to_binary()
        except SystemExit:
            pass
    finally:
        sys.argv[:] = original_argv
```

2.3 tests.test_convert module

```
# import os
import pytest
import tempfile

from stl import stl


def _test_conversion(from_, to, mode, speedups):

    for name in from_.listdir():
        source_file = from_.join(name)
        expected_file = to.join(name)
        if not expected_file.exists():
            continue

        mesh = stl.StlMesh(source_file, speedups=speedups)
        with open(str(expected_file), 'rb') as expected_fh:
            expected = expected_fh.read()
            # For binary files, skip the header
            if mode is stl.BINARY:
                expected = expected[80:]

            with tempfile.TemporaryFile() as dest_fh:
                mesh.save(name, dest_fh, mode)
                # Go back to the beginning to read
                dest_fh.seek(0)
                dest = dest_fh.read()
                # For binary files, skip the header
                if mode is stl.BINARY:
                    dest = dest[80:]

                assert dest.strip() == expected.strip()


def test_ascii_to_binary(ascii_path, binary_path, speedups):
    _test_conversion(ascii_path, binary_path, mode=stl.BINARY,
```

(continues on next page)

(continued from previous page)

```
                    speedups=speedups)


def test_binary_to_ascii(ascii_path, binary_path, speedups):
    _test_conversion(binary_path, ascii_path, mode=stl.ASCII,
                     speedups=speedups)


def test_stl_mesh(ascii_file, tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')

    mesh = stl.StlMesh(ascii_file, speedups=speedups)
    with pytest.raises(ValueError):
        mesh.save(filename=str(tmp_file), mode='test')

    mesh.save(str(tmp_file))
    mesh.save(str(tmp_file), update_normals=False)
```

2.4 tests.test_mesh module

```
import numpy

from stl.mesh import Mesh
from stl.base import BaseMesh
from stl.base import RemoveDuplicates

from . import utils


def test_units_1d():
    data = numpy.zeros(1, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [1, 0, 0],
                                      [2, 0, 0]])

    mesh = Mesh(data, remove_empty_areas=False)
    mesh.update_units()

    assert mesh.areas == 0
    utils.array_equals(mesh.normals, [0, 0, 0])
    utils.array_equals(mesh.units, [0, 0, 0])


def test_units_2d():
    data = numpy.zeros(2, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [1, 0, 0],
                                      [0, 1, 0]])
    data['vectors'][1] = numpy.array([[1, 0, 0],
                                      [0, 1, 0],
                                      [1, 1, 0]])

    mesh = Mesh(data, remove_empty_areas=False)
    mesh.update_units()
```

(continues on next page)

(continued from previous page)

```
    assert numpy.allclose(mesh.areas, [.5, .5])
    assert numpy.allclose(mesh.normals, [[0, 0, 1.], [0, 0, -1.]])
    assert numpy.allclose(mesh.units, [[0, 0, 1], [0, 0, -1]])


def test_units_3d():
    data = numpy.zeros(1, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [1, 0, 0],
                                      [0, 1, 1.]])

    mesh = Mesh(data, remove_empty_areas=False)
    mesh.update_units()

    assert (mesh.areas - 2 ** .5) < 0.0001
    assert numpy.allclose(mesh.normals, [0, -1, 1])

    units = mesh.units[0]
    assert units[0] == 0
    # Due to floating point errors
    assert (units[1] + .5 * 2 ** .5) < 0.0001
    assert (units[2] - .5 * 2 ** .5) < 0.0001


def test_duplicate_polygons():
    data = numpy.zeros(6, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[1, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][1] = numpy.array([[2, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][2] = numpy.array([[0, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][3] = numpy.array([[2, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][4] = numpy.array([[1, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][5] = numpy.array([[0, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])

    mesh = Mesh(data)
    assert mesh.data.size == 6

    mesh = Mesh(data, remove_duplicate_polygons=0)
    assert mesh.data.size == 6

    mesh = Mesh(data, remove_duplicate_polygons=False)
    assert mesh.data.size == 6

    mesh = Mesh(data, remove_duplicate_polygons=None)
    assert mesh.data.size == 6
```

(continues on next page)

(continued from previous page)

```
    mesh = Mesh(data, remove_duplicate_polygons=RemoveDuplicates.NONE)
    assert mesh.data.size == 6

    mesh = Mesh(data, remove_duplicate_polygons=RemoveDuplicates.SINGLE)
    assert mesh.data.size == 3

    mesh = Mesh(data, remove_duplicate_polygons=True)
    assert mesh.data.size == 3

    assert numpy.allclose(mesh.vectors[0], numpy.array([[1, 0, 0],
                                                        [0, 0, 0],
                                                        [0, 0, 0]]))
    assert numpy.allclose(mesh.vectors[1], numpy.array([[2, 0, 0],
                                                        [0, 0, 0],
                                                        [0, 0, 0]]))
    assert numpy.allclose(mesh.vectors[2], numpy.array([[0, 0, 0],
                                                        [0, 0, 0],
                                                        [0, 0, 0]]))

    mesh = Mesh(data, remove_duplicate_polygons=RemoveDuplicates.ALL)
    assert mesh.data.size == 3

    assert numpy.allclose(mesh.vectors[0], numpy.array([[1, 0, 0],
                                                        [0, 0, 0],
                                                        [0, 0, 0]]))
    assert numpy.allclose(mesh.vectors[1], numpy.array([[2, 0, 0],
                                                        [0, 0, 0],
                                                        [0, 0, 0]]))
    assert numpy.allclose(mesh.vectors[2], numpy.array([[0, 0, 0],
                                                        [0, 0, 0],
                                                        [0, 0, 0]]))


def test_remove_all_duplicate_polygons():
    data = numpy.zeros(5, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][1] = numpy.array([[1, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][2] = numpy.array([[2, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][3] = numpy.array([[3, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])
    data['vectors'][4] = numpy.array([[3, 0, 0],
                                      [0, 0, 0],
                                      [0, 0, 0]])

    mesh = Mesh(data, remove_duplicate_polygons=False)
    assert mesh.data.size == 5
    Mesh.remove_duplicate_polygons(mesh.data, RemoveDuplicates.NONE)

    mesh = Mesh(data, remove_duplicate_polygons=RemoveDuplicates.ALL)
```

(continues on next page)

(continued from previous page)

```
    assert mesh.data.size == 3

    assert (mesh.vectors[0] == numpy.array([[0, 0, 0],
                                            [0, 0, 0],
                                            [0, 0, 0]])).all()
    assert (mesh.vectors[1] == numpy.array([[1, 0, 0],
                                            [0, 0, 0],
                                            [0, 0, 0]])).all()
    assert (mesh.vectors[2] == numpy.array([[2, 0, 0],
                                            [0, 0, 0],
                                            [0, 0, 0]])).all()


def test_empty_areas():
    data = numpy.zeros(3, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 0, 0],
                                      [1, 0, 0],
                                      [0, 1, 0]])
    data['vectors'][1] = numpy.array([[1, 0, 0],
                                      [0, 1, 0],
                                      [1, 0, 0]])
    data['vectors'][2] = numpy.array([[1, 0, 0],
                                      [0, 1, 0],
                                      [1, 0, 0]])

    mesh = Mesh(data, remove_empty_areas=False)
    assert mesh.data.size == 3

    mesh = Mesh(data, remove_empty_areas=True)
    assert mesh.data.size == 1


def test_base_mesh():
    data = numpy.zeros(10, dtype=BaseMesh.dtype)
    mesh = BaseMesh(data, remove_empty_areas=False)
    # Increment vector 0 item 0
    mesh.v0[0] += 1
    mesh.v1[0] += 2

    # Check item 0 (contains v0, v1 and v2)
    assert (mesh[0] == numpy.array(
        [1., 1., 1., 2., 2., 2., 0., 0., 0.], dtype=numpy.float32)
    ).all()
    assert (mesh.vectors[0] == numpy.array([
            [1., 1., 1.],
            [2., 2., 2.],
            [0., 0., 0.]], dtype=numpy.float32)).all()
    assert (mesh.v0[0] == numpy.array([1., 1., 1.], dtype=numpy.float32)).all()
    assert (mesh.points[0] == numpy.array(
        [1., 1., 1., 2., 2., 2., 0., 0., 0.], dtype=numpy.float32)
    ).all()
    assert (
        mesh.x[0] == numpy.array([1., 2., 0.], dtype=numpy.float32)).all()

    mesh[0] = 3
    assert (mesh[0] == numpy.array(
        [3., 3., 3., 3., 3., 3., 3., 3., 3.], dtype=numpy.float32)
```

(continues on next page)

(continued from previous page)

```
    ).all()

    assert len(mesh) == len(list(mesh))
    assert (mesh.min_ < mesh.max_).all()
    mesh.update_normals()
    assert mesh.units.sum() == 0.0
    mesh.v0[:] = mesh.v1[:] = mesh.v2[:] = 0
    assert mesh.points.sum() == 0.0
```

2.5 tests.test_multiple module

```
from stl import mesh
from stl.utils import b

_STL_FILE = b('''
solid test.stl
facet normal -0.014565 0.073223 -0.002897
  outer loop
    vertex 0.399344 0.461940 1.044090
    vertex 0.500000 0.500000 1.500000
    vertex 0.576120 0.500000 1.117320
  endloop
endfacet
endsolid test.stl
'''.lstrip())


def test_single_stl(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('wb+') as fh:
        fh.write(_STL_FILE)
        fh.seek(0)
        for m in mesh.Mesh.from_multi_file(
                str(tmp_file), fh=fh, speedups=speedups):
            pass


def test_multiple_stl(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('wb+') as fh:
        for _ in range(10):
            fh.write(_STL_FILE)
        fh.seek(0)
        for i, m in enumerate(mesh.Mesh.from_multi_file(
                str(tmp_file), fh=fh, speedups=speedups)):
            assert m.name == b'test.stl'

        assert i == 9


def test_single_stl_file(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('wb+') as fh:
        fh.write(_STL_FILE)
```

(continues on next page)

(continued from previous page)

```
        fh.seek(0)
        for m in mesh.Mesh.from_multi_file(
                str(tmp_file), speedups=speedups):
            pass


def test_multiple_stl_file(tmpdir, speedups):
    tmp_file = tmpdir.join('tmp.stl')
    with tmp_file.open('wb+') as fh:
        for _ in range(10):
            fh.write(_STL_FILE)

        fh.seek(0)
        for i, m in enumerate(mesh.Mesh.from_multi_file(
                str(tmp_file), speedups=speedups)):
            assert m.name == b'test.stl'

        assert i == 9
```

2.6 tests.test_rotate module

```
import math
import numpy
import pytest

from stl.mesh import Mesh

from . import utils


def test_rotation():
    # Create 6 faces of a cube
    data = numpy.zeros(6, dtype=Mesh.dtype)

    # Top of the cube
    data['vectors'][0] = numpy.array([[0, 1, 1],
                                      [1, 0, 1],
                                      [0, 0, 1]])
    data['vectors'][1] = numpy.array([[1, 0, 1],
                                      [0, 1, 1],
                                      [1, 1, 1]])
    # Right face
    data['vectors'][2] = numpy.array([[1, 0, 0],
                                      [1, 0, 1],
                                      [1, 1, 0]])
    data['vectors'][3] = numpy.array([[1, 1, 1],
                                      [1, 0, 1],
                                      [1, 1, 0]])
    # Left face
    data['vectors'][4] = numpy.array([[0, 0, 0],
                                      [1, 0, 0],
                                      [1, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
    data['vectors'][5] = numpy.array([[0, 0, 0],
                                      [0, 0, 1],
                                      [1, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)

    # Since the cube faces are from 0 to 1 we can move it to the middle by
    # substracting .5
    data['vectors'] -= .5

    # Rotate 90 degrees over the X axis followed by the Y axis followed by the
    # X axis
    mesh.rotate([0.5, 0.0, 0.0], math.radians(90))
    mesh.rotate([0.0, 0.5, 0.0], math.radians(90))
    mesh.rotate([0.5, 0.0, 0.0], math.radians(90))

    # Since the cube faces are from 0 to 1 we can move it to the middle by
    # substracting .5
    data['vectors'] += .5

    # We use a slightly higher absolute tolerance here, for ppc64le
    # https://github.com/WoLpH/numpy-stl/issues/78
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[1, 0, 0], [0, 1, 0], [0, 0, 0]],
        [[0, 1, 0], [1, 0, 0], [1, 1, 0]],
        [[0, 1, 1], [0, 1, 0], [1, 1, 1]],
        [[1, 1, 0], [0, 1, 0], [1, 1, 1]],
        [[0, 0, 1], [0, 1, 1], [0, 1, 0]],
        [[0, 0, 1], [0, 0, 0], [0, 1, 0]],
    ]), atol=1e-07)


def test_rotation_over_point():
    # Create a single face
    data = numpy.zeros(1, dtype=Mesh.dtype)

    data['vectors'][0] = numpy.array([[1, 0, 0],
                                      [0, 1, 0],
                                      [0, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)

    mesh.rotate([1, 0, 0], math.radians(180), point=[1, 2, 3])
    utils.array_equals(
        mesh.vectors,
        numpy.array([[[1., 4., 6.],
                      [0., 3., 6.],
                      [0., 4., 5.]]]))

    mesh.rotate([1, 0, 0], math.radians(-180), point=[1, 2, 3])
    utils.array_equals(
        mesh.vectors,
        numpy.array([[[1, 0, 0],
                      [0, 1, 0],
                      [0, 0, 1]]]))

    mesh.rotate([1, 0, 0], math.radians(180), point=0.0)
```

(continues on next page)

(continued from previous page)

```
    utils.array_equals(
        mesh.vectors,
        numpy.array([[[1., 0., -0.],
                      [0., -1., -0.],
                      [0., 0., -1.]]]))

    with pytest.raises(TypeError):
        mesh.rotate([1, 0, 0], math.radians(180), point='x')


def test_double_rotation():
    # Create a single face
    data = numpy.zeros(1, dtype=Mesh.dtype)

    data['vectors'][0] = numpy.array([[1, 0, 0],
                                      [0, 1, 0],
                                      [0, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)

    rotation_matrix = mesh.rotation_matrix([1, 0, 0], math.radians(180))
    combined_rotation_matrix = numpy.dot(rotation_matrix, rotation_matrix)

    mesh.rotate_using_matrix(combined_rotation_matrix)
    utils.array_equals(
        mesh.vectors,
        numpy.array([[[1., 0., 0.],
                      [0., 1., 0.],
                      [0., 0., 1.]]]))


def test_no_rotation():
    # Create a single face
    data = numpy.zeros(1, dtype=Mesh.dtype)

    data['vectors'][0] = numpy.array([[0, 1, 1],
                                      [1, 0, 1],
                                      [0, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)

    # Rotate by 0 degrees
    mesh.rotate([0.5, 0.0, 0.0], math.radians(0))
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))

    # Use a zero rotation matrix
    mesh.rotate([0.0, 0.0, 0.0], math.radians(90))
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))


def test_no_translation():
    # Create a single face
    data = numpy.zeros(1, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 1, 1],
                                      [1, 0, 1],
```

(continues on next page)

(continued from previous page)

```
                                        [0, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))

    # Translate mesh with a zero vector
    mesh.translate([0.0, 0.0, 0.0])
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))


def test_translation():
    # Create a single face
    data = numpy.zeros(1, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 1, 1],
                                      [1, 0, 1],
                                      [0, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))

    # Translate mesh with vector [1, 2, 3]
    mesh.translate([1.0, 2.0, 3.0])
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[1, 3, 4], [2, 2, 4], [1, 2, 4]]]))


def test_no_transformation():
    # Create a single face
    data = numpy.zeros(1, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 1, 1],
                                      [1, 0, 1],
                                      [0, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))

    # Transform mesh with identity matrix
    mesh.transform(numpy.eye(4))
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))
    assert numpy.allclose(mesh.areas, 0.5)


def test_transformation():
    # Create a single face
    data = numpy.zeros(1, dtype=Mesh.dtype)
    data['vectors'][0] = numpy.array([[0, 1, 1],
                                      [1, 0, 1],
                                      [0, 0, 1]])

    mesh = Mesh(data, remove_empty_areas=False)
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 1, 1], [1, 0, 1], [0, 0, 1]]]))
```

(continues on next page)

(continued from previous page)

```
    # Transform mesh with identity matrix
    tr = numpy.zeros((4, 4))
    tr[0:3, 0:3] = Mesh.rotation_matrix([0, 0, 1], 0.5 * numpy.pi)
    tr[0:3, 3] = [1, 2, 3]
    mesh.transform(tr)
    assert numpy.allclose(mesh.vectors, numpy.array([
        [[0, 2, 4], [1, 3, 4], [1, 2, 4]]]))
    assert numpy.allclose(mesh.areas, 0.5)
```

CHAPTER 3

stl package

3.1 stl.Mesh

class stl.**Mesh**(*data*, *calculate_normals=True*, *remove_empty_areas=False*, *remove_duplicate_polygons=<RemoveDuplicates.NONE: 0>*, *name=u''*, *speedups=True*, ***kwargs*)

Bases: `stl.stl.BaseStl`

areas

Mesh areas

attr

check()

classmethod debug(*msg*, **args*, ***kwargs*)

Log a message with severity 'DEBUG' on the root logger.

`dtype = dtype([('normals', '<f4', (3,)), ('vectors', '<f4', (3, 3)), ('attr', '<u2', (`

classmethod error(*msg*, **args*, ***kwargs*)

Log a message with severity 'ERROR' on the root logger.

classmethod exception(*msg*, **args*, ***kwargs*)

Log a message with severity 'ERROR' on the root logger, with exception information.

classmethod from_file(*filename*, *calculate_normals=True*, *fh=None*, *mode=<Mode.AUTOMATIC: 0>*, *speedups=True*, ***kwargs*)

Load a mesh from a STL file

Parameters

- **filename** (`str`) – The file to load
- **calculate_normals** (`bool`) – Whether to update the normals
- **fh** (`file`) – The file handle to open
- ****kwargs** (`dict`) – The same as for `stl.mesh.Mesh`

classmethod from_multi_file(*filename*, *calculate_normals=True*, *fh=None*, *mode=<Mode.ASCII: 1>*, *speedups=True*, ***kwargs*)

Load multiple meshes from a STL file

Note: mode is hardcoded to ascii since binary stl files do not support the multi format

Parameters

- **filename** (*str*) – The file to load
- **calculate_normals** (*bool*) – Whether to update the normals
- **fh** (*file*) – The file handle to open
- ****kwargs** (*dict*) – The same as for *stl.mesh.Mesh*

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_mass_properties()

Evaluate and return a tuple with the following elements:

- the volume
- the position of the center of gravity (COG)
- the inertia matrix expressed at the COG

Documentation can be found here: http://www.geometrictools.com/Documentation/PolyhedralMassProperties.pdf

classmethod info(*msg*, **args*, ***kwargs*)

Log a message with severity 'INFO' on the root logger.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

classmethod load(*fh*, *mode=<Mode.AUTOMATIC: 0>*, *speedups=True*)

Load Mesh from STL file

Automatically detects binary versus ascii STL files.

Parameters

- **fh** (*file*) – The file handle to open
- **mode** (*int*) – Automatically detect the filetype or force binary

classmethod log(*lvl*, *msg*, **args*, ***kwargs*)

Log 'msg % args' with the integer severity 'level' on the root logger.

logger = <logging.Logger object>

max_

Mesh maximum value

min_

Mesh minimum value

normals

points

classmethod remove_duplicate_polygons(*data*, *value=<RemoveDuplicates.SINGLE: 1>*)

classmethod remove_empty_areas(*data*)

rotate(*axis*, *theta=0*, *point=None*)

Rotate the matrix over the given axis by the given theta (angle)

Uses the `rotation_matrix()` in the background.

Note: Note that the *point* was accidentaly inverted with the old version of the code. To get the old and incorrect behaviour simply pass *-point* instead of *point* or *-numpy.array(point)* if you're passing along an array.

Parameters

- **axis** (*numpy.array*) – Axis to rotate over (x, y, z)
- **theta** (*float*) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.
- **point** (*numpy.array*) – Rotation point so manual translation is not required

rotate_using_matrix(*rotation_matrix*, *point=None*)

classmethod rotation_matrix(*axis*, *theta*)

Generate a rotation matrix to Rotate the matrix over the given axis by the given theta (angle)

Uses the Euler-Rodrigues formula for fast rotations.

Parameters

- **axis** (*numpy.array*) – Axis to rotate over (x, y, z)
- **theta** (*float*) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.

save(*filename*, *fh=None*, *mode=<Mode.AUTOMATIC: 0>*, *update_normals=True*)

Save the STL to a (binary) file

If mode is `AUTOMATIC` an `ASCII` file will be written if the output is a TTY and a `BINARY` file otherwise.

Parameters

- **filename** (*str*) – The file to load
- **fh** (*file*) – The file handle to open
- **mode** (*int*) – The mode to write, default is `AUTOMATIC`.
- **update_normals** (*bool*) – Whether to update the normals

transform(*matrix*)

Transform the mesh with a rotation and a translation stored in a single 4x4 matrix

Parameters matrix (*numpy.array*) – Transform matrix with shape (4, 4), where matrix[0:3, 0:3] represents the rotation part of the transformation matrix[0:3, 3] represents the translation part of the transformation

translate(*translation*)

Translate the mesh in the three directions

Parameters translation (*numpy.array*) – Translation vector (x, y, z)

units
Mesh unit vectors

update_areas()

update_max()

update_min()

update_normals()
Update the normals for all points

update_units()

v0

v1

v2

values() → list of D's values

vectors

classmethod warning(*msg*, **args*, ***kwargs*)
Log a message with severity 'WARNING' on the root logger.

x

y

z

3.2 stl.main module

stl.main.**main**()

stl.main.**to_ascii**()

stl.main.**to_binary**()

3.3 stl.base module

stl.base.**AREA_SIZE_THRESHOLD = 0**
When removing empty areas, remove areas that are smaller than this

class stl.base.**BaseMesh**(*data*, *calculate_normals=True*, *remove_empty_areas=False*, *remove_duplicate_polygons=<RemoveDuplicates.NONE: 0>*, *name=u''*, *speedups=True*, ***kwargs*)
Bases: python_utils.logger.Logged, _abcoll.Mapping

Mesh object with easy access to the vectors through v0, v1 and v2. The normals, areas, min, max and units are calculated automatically.

Parameters

- **data** (*numpy.array*) – The data for this mesh
- **calculate_normals** (*bool*) – Whether to calculate the normals
- **remove_empty_areas** (*bool*) – Whether to remove triangles with 0 area (due to rounding errors for example)

Variables

- **name** (*str*) – Name of the solid, only exists in ASCII files
- **data** (*numpy.array*) – Data as *BaseMesh.dtype()*
- ***points*** (*numpy.array*) – All points (Nx9)
- ***normals*** (*numpy.array*) – Normals for this mesh, calculated automatically by default (Nx3)
- ***vectors*** (*numpy.array*) – Vectors in the mesh (Nx3x3)
- ***attr*** (*numpy.array*) – Attributes per vector (used by binary STL)
- ***x*** (*numpy.array*) – Points on the X axis by vertex (Nx3)
- ***y*** (*numpy.array*) – Points on the Y axis by vertex (Nx3)
- ***z*** (*numpy.array*) – Points on the Z axis by vertex (Nx3)
- ***v0*** (*numpy.array*) – Points in vector 0 (Nx3)
- ***v1*** (*numpy.array*) – Points in vector 1 (Nx3)
- ***v2*** (*numpy.array*) – Points in vector 2 (Nx3)

```
>>> data = numpy.zeros(10, dtype=BaseMesh.dtype)
>>> mesh = BaseMesh(data, remove_empty_areas=False)
>>> # Increment vector 0 item 0
>>> mesh.v0[0] += 1
>>> mesh.v1[0] += 2
```

```
>>> # Check item 0 (contains v0, v1 and v2)
>>> assert numpy.array_equal(
...     mesh[0],
...     numpy.array([1., 1., 1., 2., 2., 2., 0., 0., 0.]))
>>> assert numpy.array_equal(
... mesh.vectors[0],
... numpy.array([[1., 1., 1.],
...     [2., 2., 2.],
...     [0., 0., 0.]]))
>>> assert numpy.array_equal(
...     mesh.v0[0],
...     numpy.array([1., 1., 1.]))
>>> assert numpy.array_equal(
...     mesh.points[0],
...     numpy.array([1., 1., 1., 2., 2., 2., 0., 0., 0.]))
>>> assert numpy.array_equal(
...     mesh.data[0],
...     numpy.array((
...             [0., 0., 0.],
...             [[1., 1., 1.], [2., 2., 2.], [0., 0., 0.]],
...             [0]),
...         dtype=BaseMesh.dtype))
>>> assert numpy.array_equal(mesh.x[0], numpy.array([1., 2., 0.]))
```

```
>>> mesh[0] = 3
>>> assert numpy.array_equal(
...     mesh[0],
...     numpy.array([3., 3., 3., 3., 3., 3., 3., 3., 3.]))
```

```
>>> len(mesh) == len(list(mesh))
True
>>> (mesh.min_ < mesh.max_).all()
True
>>> mesh.update_normals()
>>> mesh.units.sum()
0.0
>>> mesh.v0[:] = mesh.v1[:] = mesh.v2[:] = 0
>>> mesh.points.sum()
0.0
```

```
>>> mesh.v0 = mesh.v1 = mesh.v2 = 0
>>> mesh.x = mesh.y = mesh.z = 0
```

```
>>> mesh.attr = 1
>>> (mesh.attr == 1).all()
True
```

```
>>> mesh.normals = 2
>>> (mesh.normals == 2).all()
True
```

```
>>> mesh.vectors = 3
>>> (mesh.vectors == 3).all()
True
```

```
>>> mesh.points = 4
>>> (mesh.points == 4).all()
True
```

areas
Mesh areas

attr

check()

classmethod debug(*msg*, **args*, ***kwargs*)
Log a message with severity 'DEBUG' on the root logger.

dtype = dtype([('normals', '<f4', (3,)), ('vectors', '<f4', (3, 3)), ('attr', '<u2', (

- normals: `numpy.float32()`, *(3,)*
- vectors: `numpy.float32()`, *(3, 3)*
- attr: `numpy.uint16()`, *(1,)*

classmethod error(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger.

classmethod exception(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger, with exception information.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_mass_properties()

Evaluate and return a tuple with the following elements:

- the volume
- the position of the center of gravity (COG)
- the inertia matrix expressed at the COG

Documentation can be found here: http://www.geometrictools.com/Documentation/PolyhedralMassProperties.pdf

classmethod info(*msg*, **args*, ***kwargs*)
Log a message with severity 'INFO' on the root logger.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

classmethod log(*lvl*, *msg*, **args*, ***kwargs*)
Log 'msg % args' with the integer severity 'level' on the root logger.

logger = <logging.Logger object>

max_
Mesh maximum value

min_
Mesh minimum value

normals

points

classmethod remove_duplicate_polygons(*data*, *value=<RemoveDuplicates.SINGLE: 1>*)

classmethod remove_empty_areas(*data*)

rotate(*axis*, *theta=0*, *point=None*)
Rotate the matrix over the given axis by the given theta (angle)

Uses the `rotation_matrix()` in the background.

Note: Note that the *point* was accidentaly inverted with the old version of the code. To get the old and incorrect behaviour simply pass *-point* instead of *point* or *-numpy.array(point)* if you're passing along an array.

Parameters

- **axis** (`numpy.array`) – Axis to rotate over (x, y, z)
- **theta** (`float`) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.
- **point** (`numpy.array`) – Rotation point so manual translation is not required

rotate_using_matrix(*rotation_matrix*, *point=None*)

classmethod rotation_matrix(*axis*, *theta*)
Generate a rotation matrix to Rotate the matrix over the given axis by the given theta (angle)

Uses the Euler-Rodrigues formula for fast rotations.

Parameters

- **axis** (*numpy.array*) – Axis to rotate over (x, y, z)
- **theta** (*float*) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.

transform(*matrix*)
Transform the mesh with a rotation and a translation stored in a single 4x4 matrix

Parameters matrix (*numpy.array*) – Transform matrix with shape (4, 4), where matrix[0:3, 0:3] represents the rotation part of the transformation matrix[0:3, 3] represents the translation part of the transformation

translate(*translation*)
Translate the mesh in the three directions

Parameters translation (*numpy.array*) – Translation vector (x, y, z)

units
Mesh unit vectors

update_areas()

update_max()

update_min()

update_normals()
Update the normals for all points

update_units()

v0

v1

v2

values() → list of D's values

vectors

classmethod warning(*msg*, **args*, ***kwargs*)
Log a message with severity 'WARNING' on the root logger.

x

y

z

stl.base.**DIMENSIONS = 3**
Dimensions used in a vector

class stl.base.**Dimension**
Bases: `enum.IntEnum`

X = 0
X index (for example, *mesh.v0[0][X]*)

Y = 1
Y index (for example, *mesh.v0[0][Y]*)

Z = 2
Z index (for example, *mesh.v0[0][Z]*)

class stl.base.**RemoveDuplicates**
Bases: `enum.Enum`

Choose whether to remove no duplicates, leave only a single of the duplicates or remove all duplicates (leaving holes).

ALL = 2

NONE = 0

SINGLE = 1

stl.base.**VECTORS = 3**
Vectors in a point

stl.base.**logged**(*class_*)

3.4 stl.mesh module

class stl.mesh.**Mesh**(*data*, *calculate_normals=True*, *remove_empty_areas=False*, *remove_duplicate_polygons=<RemoveDuplicates.NONE: 0>*, *name=u''*, *speedups=True*, ***kwargs*)
Bases: `stl.stl.BaseStl`

areas
Mesh areas

attr

check()

classmethod debug(*msg*, **args*, ***kwargs*)
Log a message with severity 'DEBUG' on the root logger.

dtype = dtype([('normals', '<f4', (3,)), ('vectors', '<f4', (3, 3)), ('attr', '<u2', (

classmethod error(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger.

classmethod exception(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger, with exception information.

classmethod from_file(*filename*, *calculate_normals=True*, *fh=None*, *mode=<Mode.AUTOMATIC: 0>*, *speedups=True*, ***kwargs*)
Load a mesh from a STL file

Parameters

- **filename** (`str`) – The file to load
- **calculate_normals** (`bool`) – Whether to update the normals
- **fh** (`file`) – The file handle to open
- ****kwargs** (`dict`) – The same as for `stl.mesh.Mesh`

classmethod from_multi_file(*filename*, *calculate_normals=True*, *fh=None*, *mode=<Mode.ASCII: 1>*, *speedups=True*, ***kwargs*)
Load multiple meshes from a STL file

Note: mode is hardcoded to ascii since binary stl files do not support the multi format

Parameters

- **filename** (*str*) – The file to load
- **calculate_normals** (*bool*) – Whether to update the normals
- **fh** (*file*) – The file handle to open
- ****kwargs** (*dict*) – The same as for *stl.mesh.Mesh*

get (*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_mass_properties ()

Evaluate and return a tuple with the following elements:

- the volume
- the position of the center of gravity (COG)
- the inertia matrix expressed at the COG

Documentation can be found here: http://www.geometrictools.com/Documentation/PolyhedralMassProperties.pdf

classmethod info (*msg*, **args*, ***kwargs*)
Log a message with severity 'INFO' on the root logger.

items () → list of D's (key, value) pairs, as 2-tuples

iteritems () → an iterator over the (key, value) items of D

iterkeys () → an iterator over the keys of D

itervalues () → an iterator over the values of D

keys () → list of D's keys

classmethod load (*fh*, *mode=<Mode.AUTOMATIC: 0>*, *speedups=True*)
Load Mesh from STL file

Automatically detects binary versus ascii STL files.

Parameters

- **fh** (*file*) – The file handle to open
- **mode** (*int*) – Automatically detect the filetype or force binary

classmethod log (*lvl*, *msg*, **args*, ***kwargs*)
Log 'msg % args' with the integer severity 'level' on the root logger.

logger = <logging.Logger object>

max_
Mesh maximum value

min_
Mesh minimum value

normals

points

classmethod remove_duplicate_polygons (*data*, *value=<RemoveDuplicates.SINGLE: 1>*)

classmethod remove_empty_areas (*data*)

rotate (*axis*, *theta=0*, *point=None*)
Rotate the matrix over the given axis by the given theta (angle)

Uses the `rotation_matrix()` in the background.

Note: Note that the *point* was accidentaly inverted with the old version of the code. To get the old and incorrect behaviour simply pass *-point* instead of *point* or *-numpy.array(point)* if you're passing along an array.

Parameters

- **axis** (*numpy.array*) – Axis to rotate over (x, y, z)
- **theta** (*float*) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.
- **point** (*numpy.array*) – Rotation point so manual translation is not required

rotate_using_matrix (*rotation_matrix*, *point=None*)

classmethod rotation_matrix (*axis*, *theta*)
Generate a rotation matrix to Rotate the matrix over the given axis by the given theta (angle)

Uses the Euler-Rodrigues formula for fast rotations.

Parameters

- **axis** (*numpy.array*) – Axis to rotate over (x, y, z)
- **theta** (*float*) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.

save (*filename*, *fh=None*, *mode=<Mode.AUTOMATIC: 0>*, *update_normals=True*)
Save the STL to a (binary) file

If mode is `AUTOMATIC` an `ASCII` file will be written if the output is a TTY and a `BINARY` file otherwise.

Parameters

- **filename** (*str*) – The file to load
- **fh** (*file*) – The file handle to open
- **mode** (*int*) – The mode to write, default is `AUTOMATIC`.
- **update_normals** (*bool*) – Whether to update the normals

transform (*matrix*)
Transform the mesh with a rotation and a translation stored in a single 4x4 matrix

Parameters **matrix** (*numpy.array*) – Transform matrix with shape (4, 4), where matrix[0:3, 0:3] represents the rotation part of the transformation matrix[0:3, 3] represents the translation part of the transformation

translate (*translation*)
Translate the mesh in the three directions

Parameters **translation** (*numpy.array*) – Translation vector (x, y, z)

units
Mesh unit vectors

update_areas ()

update_max()

update_min()

update_normals()
Update the normals for all points

update_units()

v0

v1

v2

values() → list of D's values

vectors

classmethod warning(*msg*, **args*, ***kwargs*)
Log a message with severity 'WARNING' on the root logger.

x

y

z

3.5 stl.stl module

stl.stl.**BUFFER_SIZE = 4096**
Amount of bytes to read while using buffered reading

class stl.stl.**BaseStl**(*data*, *calculate_normals=True*, *remove_empty_areas=False*, *remove_duplicate_polygons=<RemoveDuplicates.NONE: 0>*, *name=u"*, *speedups=True*, ***kwargs*)
Bases: `stl.base.BaseMesh`

areas
Mesh areas

attr

check()

classmethod debug(*msg*, **args*, ***kwargs*)
Log a message with severity 'DEBUG' on the root logger.

dtype = dtype([('normals', '<f4', (3,)), ('vectors', '<f4', (3, 3)), ('attr', '<u2', (

classmethod error(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger.

classmethod exception(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' on the root logger, with exception information.

classmethod from_file(*filename*, *calculate_normals=True*, *fh=None*, *mode=<Mode.AUTOMATIC: 0>*, *speedups=True*, ***kwargs*)
Load a mesh from a STL file

Parameters

- **filename** (*str*) – The file to load
- **calculate_normals** (*bool*) – Whether to update the normals

- **fh** (*file*) – The file handle to open
- ****kwargs** (*dict*) – The same as for *stl.mesh.Mesh*

classmethod from_multi_file(*filename*, *calculate_normals=True*, *fh=None*, *mode=<Mode.ASCII: 1>*, *speedups=True*, ***kwargs*)

Load multiple meshes from a STL file

Note: mode is hardcoded to ascii since binary stl files do not support the multi format

Parameters

- **filename** (*str*) – The file to load
- **calculate_normals** (*bool*) – Whether to update the normals
- **fh** (*file*) – The file handle to open
- ****kwargs** (*dict*) – The same as for *stl.mesh.Mesh*

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_mass_properties()

Evaluate and return a tuple with the following elements:

- the volume
- the position of the center of gravity (COG)
- the inertia matrix expressed at the COG

Documentation can be found here: http://www.geometrictools.com/Documentation/PolyhedralMassProperties.pdf

classmethod info(*msg*, **args*, ***kwargs*)

Log a message with severity 'INFO' on the root logger.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

classmethod load(*fh*, *mode=<Mode.AUTOMATIC: 0>*, *speedups=True*)

Load Mesh from STL file

Automatically detects binary versus ascii STL files.

Parameters

- **fh** (*file*) – The file handle to open
- **mode** (*int*) – Automatically detect the filetype or force binary

classmethod log(*lvl*, *msg*, **args*, ***kwargs*)

Log 'msg % args' with the integer severity 'level' on the root logger.

logger = <logging.Logger object>

max_

Mesh maximum value

min_

Mesh minimum value

normals

points

classmethod remove_duplicate_polygons (*data*, *value=<RemoveDuplicates.SINGLE: 1>*)

classmethod remove_empty_areas (*data*)

rotate (*axis*, *theta=0*, *point=None*)

Rotate the matrix over the given axis by the given theta (angle)

Uses the `rotation_matrix()` in the background.

Note: Note that the *point* was accidentaly inverted with the old version of the code. To get the old and incorrect behaviour simply pass *-point* instead of *point* or *-numpy.array(point)* if you're passing along an array.

Parameters

- **axis** (`numpy.array`) – Axis to rotate over (x, y, z)
- **theta** (`float`) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.
- **point** (`numpy.array`) – Rotation point so manual translation is not required

rotate_using_matrix (*rotation_matrix*, *point=None*)

classmethod rotation_matrix (*axis*, *theta*)

Generate a rotation matrix to Rotate the matrix over the given axis by the given theta (angle)

Uses the Euler-Rodrigues formula for fast rotations.

Parameters

- **axis** (`numpy.array`) – Axis to rotate over (x, y, z)
- **theta** (`float`) – Rotation angle in radians, use *math.radians* to convert degrees to radians if needed.

save (*filename*, *fh=None*, *mode=<Mode.AUTOMATIC: 0>*, *update_normals=True*)

Save the STL to a (binary) file

If mode is `AUTOMATIC` an `ASCII` file will be written if the output is a TTY and a `BINARY` file otherwise.

Parameters

- **filename** (`str`) – The file to load
- **fh** (`file`) – The file handle to open
- **mode** (`int`) – The mode to write, default is `AUTOMATIC`.
- **update_normals** (`bool`) – Whether to update the normals

transform (*matrix*)

Transform the mesh with a rotation and a translation stored in a single 4x4 matrix

Parameters matrix (`numpy.array`) – Transform matrix with shape (4, 4), where matrix[0:3, 0:3] represents the rotation part of the transformation matrix[0:3, 3] represents the translation part of the transformation

translate (*translation*)

Translate the mesh in the three directions

Parameters **translation** (*numpy.array*) – Translation vector (x, y, z)

units
Mesh unit vectors

update_areas()

update_max()

update_min()

update_normals()
Update the normals for all points

update_units()

v0

v1

v2

values() → list of D's values

vectors

classmethod warning(*msg*, **args*, ***kwargs*)
Log a message with severity 'WARNING' on the root logger.

x

y

z

stl.stl.**COUNT_SIZE = 4**
The amount of bytes in the count field

stl.stl.**HEADER_SIZE = 80**
The amount of bytes in the header field

stl.stl.**MAX_COUNT = 100000000.0**
The maximum amount of triangles we can read from binary files

class stl.stl.**Mode**
Bases: enum.IntEnum

ASCII = 1
Force writing ASCII

AUTOMATIC = 0
Automatically detect whether the output is a TTY, if so, write ASCII otherwise write BINARY

BINARY = 2
Force writing BINARY

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

K

L

M

N

P

R

S

T

U

V

W

X

Y

Z