STL Scripting in Blender for 3D Printing

Why script STL files in Blender?

You need STL or DXF files containing your layouts to use Nanoscribe GT2 at CNF. The Nanoscribe is a two-photon polymerization (2PP) tool that can print microscale 3D objects. Its workflow is similar to that of consumer 3D printers. There are several CAD options to create printer-ready files, one of which is Blender. Blender is a free, powerful and versatile 3D software for Windows, Mac and Linux with an inbuilt STL export plugin and a robust scripting environment.

Scripting is useful when the direct way of building your device in an editor is too tedious, for example because the device is composed of many similar but slightly different objects whose dimensions vary according to a parameter, or because modelling it requires a large number of steps.

You can download Blender from its official website. Knowledge of basic Python is necessary to use this guide. You should also know the basics of Blender: although we will go through all the important steps to get started with scripting, it is very useful to know how the software is normally used, and be able to modify a scripted layout by hand.

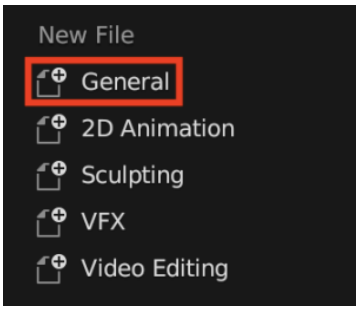
A very useful resource for this guide is Blender's API reference, which you may need to consult for your work. Reading the quick start guide is strongly recommended.

Before we start, it's worthwhile to consider Blender's geometry nodes as an alternative. A lot can be done with geometry nodes instead of scripting, which is useful if you already know how to use them or would rather learn them instead of scripting. Knowing geometry nodes is recommended either way, as they are very powerful and one of Blender's defining features.

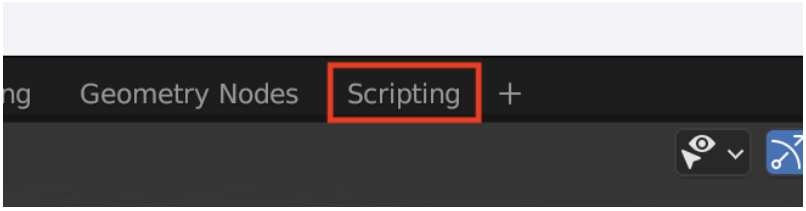
Preparations

This guide was written using blender 2.4.1.

First of all, open Blender and select "General" from the new file dialog (though in practice any environment will do).



Click on the default cube at the center of the screen, press `X` and click "Delete". Then click on the "Scripting" tab on the upper left.



The center window is the text editor, where we can type our code. Commands used for scripting and script files are available in this window's toolbar (the most useful are under "Text"). Keep in mind the following quirks of the system:

- Saving the Blender file (with extension `.blend`) saves the code in the text editor, but the code can also be saved separately to a script file (usually with extension `.py`). There are separate shortcuts to save the two (in Mac, they are `Cmd + S` and `Opt + S`, respectively). Code within Blender and in the external script files do not synchronize: you must save both separately and, if you edit the script externally, you must reload it in Blender by selecting "Text", then "Reload" twice (or using the assigned shortcut).
- Blender comes with its own inbuilt Python distribution with its own modules plus several standard ones, including all those we will need for this tutorial. While it is possible to install Python packages in it, or use Blender-specific modules outside of its environment, it is easiest to start with what is avalialble.

As a last note, Blender scripting logic can be difficult to follow at times, as it is built around the editor's logic (including the use of context) rather than a more straightforward programming paradigm.

Scripting in Blender

In all examples we will be working with meters as the base unit, because scaling can be easily performed later in the printing software and it sidesteps having to change to micrometers in Blender. Note that, even if your units are correctly set in Blender, the printing software may misinterpret them. Always check the size of your objects there before printing.

Creating and exporting a simple bar

A simple but useful shape is the "bar" (rectangular prism). To create a 3D model of a bar in Blender, we will

1. Calculate the positions of the corners at the base
2. Make lists of vertices and faces
3. Assign vertices and faces to a mesh

The reason for doing it this way is that the code can be changed easily to make structures out of arbitrary sequences of vertices, as we will see later.

Before we begin, we need to import Blender's `bpy` and `bmesh`, and we will also need `Vector` from `mathutils`.

```
import bpy
import bmesh
from mathutils import Vector
```

We start by defining the general parameters of our bar and calculating its corners, which we append to a list. We will create it at (0, 0, 0), resting on the XY plane, and move it around later.

```
### Definitions

dimensions = (2, 1, 1)        # Bar XYZ dimension
baseLocation = (1., -1., 2.)  # Bar base XYZ position
name = 'Bar'                  # Object name

### Calculate base corners

baseCorners = []
dx = dimensions[0]
dy = dimensions[1]
dz = dimensions[2]
baseCorners.append((-0.5*dx, -0.5*dy))
baseCorners.append((-0.5*dx, 0.5*dy))
baseCorners.append((0.5*dx, 0.5*dy))
baseCorners.append((0.5*dx, -0.5*dy))
```

Each face of the bar is simply a list of vertices. We can loop on the four corners and from each get the base vertices by using `z = 0`, which puts the bottom face on the XY plane, and the top vertices by using `z = dz`, which is the structure's `z` dimension.

After enough vertices have been appended, a new face is calculated at each loop, made up of the latest four. The order in which vertices are appended is important, because they are connected by Blender one after another as they are listed, forming the face's perimeter.

We must also remember to save some vertices to separate lists, for the top and bottom "cap" faces of the bar.

```
### Calculate vertices and faces

vertexList = []
topCapVertexList = []
bottomCapVertexList = []
faceList = []
for c in baseCorners:
    vertexList.append((c[0], c[1], 0))
    vertexList.append((c[0], c[1], dz))
    ### Calculate faces
    lVL = len(vertexList)
    if lVL > 3:
        f = (lVL - 1, lVL - 2, lVL - 4, lVL - 3)
        faceList.append(f)
    ### Append vertex indices for the cap faces
    topCapVertexList.append(lVL - 1)
    bottomCapVertexList.append(lVL - 2)
```

Not all of the faces have ended up in the list. We are still missing the last "wall", which connects to the first two vertices, and the two caps, for which we already have separate lists of vertices.

```
### Append remaining faces

f = (lVL - 1, lVL - 2, 0, 1)
faceList.append(f)
f = tuple(bottomCapVertexList)
faceList.append(f)
f = tuple(topCapVertexList)
faceList.append(f)
```

We can now make an object out of the vertices and faces. First, we create a mesh and assign it the name we defined above. Proper naming is important when creating many objects. For the sake of example, we set its `location` at `(0, 0, 0)` for now and later move it to its intended location with `translate`, but `location` can be used directly.

The object is linked to the currently active collection. If there are multiple collections in the Blender file, any can be selected (e.g. by clicking on it in the outliner) but this could also be done programmatically.

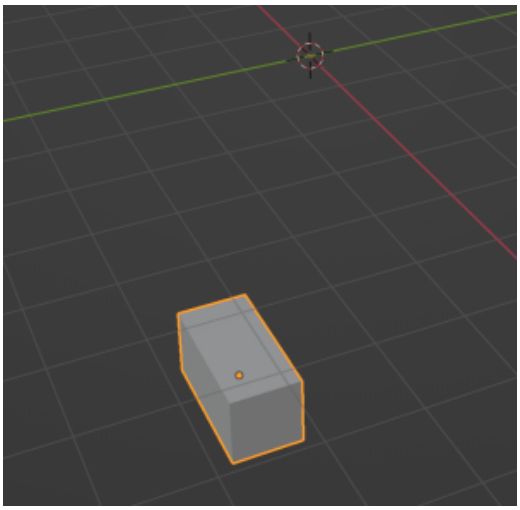
The rest of the commands make a mesh out of the Python lists of vertices and faces (no edges should be provided in this case) and a `bmesh` out of the mesh. The `validate` step is always recommended, as is recalculating the face normals.

```
### Create mesh

mesh = bpy.data.meshes.new(name)
ob = bpy.data.objects.new(name, mesh)
ob.location = Vector((0., 0., 0.))
bpy.context.collection.objects.link(ob)
mesh.from_pydata(vertexList,[],faceList)
mesh.validate(verbose=True)
bm = bmesh.new()
bm.from_mesh(mesh)
bmesh.ops.recalc_face_normals(bm, faces=bm.faces)
bmesh.ops.translate(bm, vec = baseLocation, verts = bm.verts)
bm.to_mesh(mesh)
mesh.update()
```

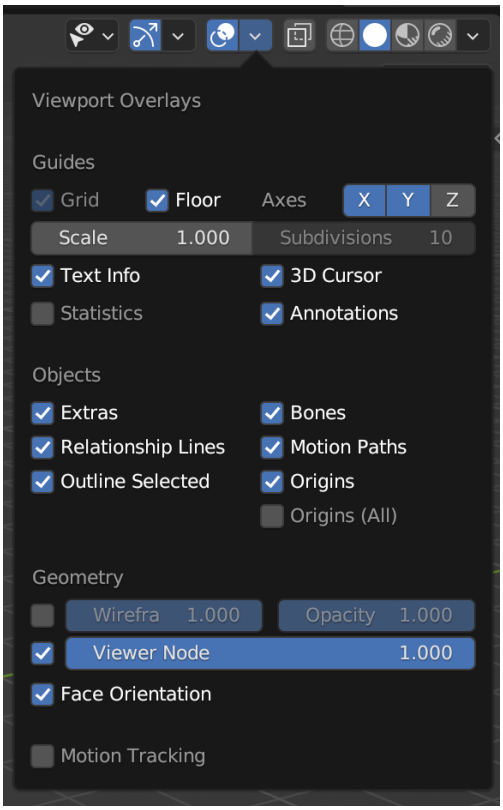
As a last step we can move the object's origin back to the center of the mesh, as the translation operations have moved it outside. This is not strictly necessary, but can help prevent confusion when starting with scripted objects and then modifying them manually in the editor.

```
### Origin to center of mesh

bpy.ops.object.select_all(action='DESELECT')
bpy.context.view_layer.objects.active = ob
ob.select_set(True)
bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```

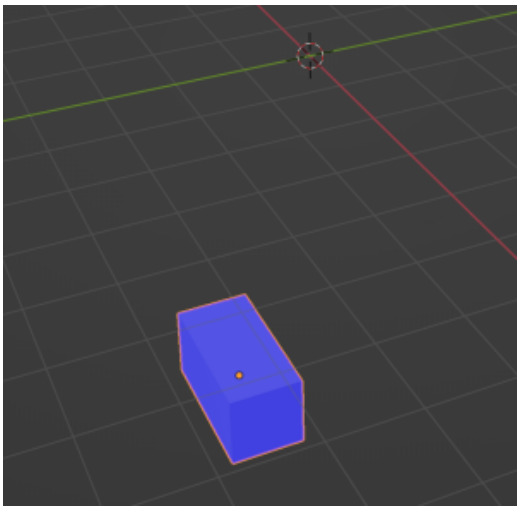
Running the entire script with the "Run Script" button at the top of the text editor will create the bar object and place it as defined.



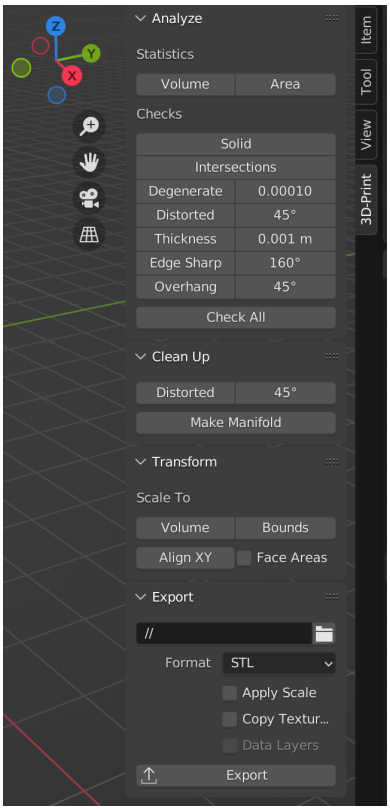
It's best to check that the face orientation is indeed correct. Go to the "Layout" tab and select "Face Orientation" in the "Viewport Overlays" dialog at the top right.



The structure should be entirely blue (red faces are in the reverse orientation).



In the "Edit" menu select "Preferences" tab, then the "Add-ons" tab. Search for and enable the "Mesh: 3D-Print Toolbox". It will enable a "3D-Print" tab at the top right when an object is selected.



It's a good idea to always use "Check All". A good object should reveal few problems, although overhang faces are common and must be placed at the bottom or supported in the printing software. Many issues can be corrected with the tools under "Clean Up", especially "Make Manifold", but sometimes they may fail or distort the geometry excessively. In this case, it's better to go back to the code and figure out the issue rather than risk printing.

Under "Export" at the bottom, the currently selected meshes can be saved in the STL format, which can be opened directly in the printing software.

This code can be easily extended to structures any shape by just providing a different list of vertices at the beginning, provided their walls are vertical. For more complex structures, the part which calculates the faces must also be modified. Several examples can be found in later sections.

Using classes to create a bar array

Let's build an array out of bars. It is convenient to take most of the code from the previous example and make a class out of it.

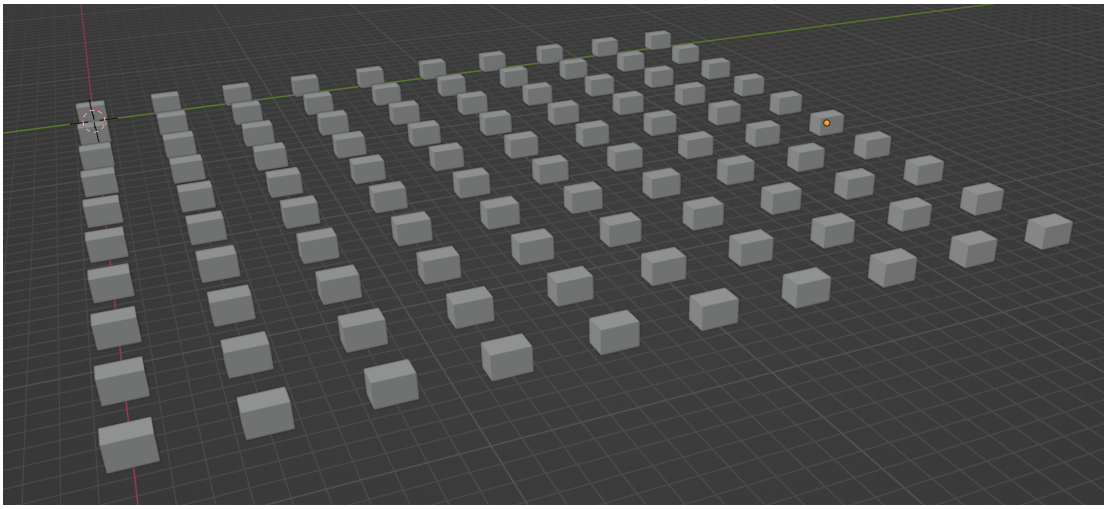
```
import bpy
import bmesh
from mathutils import Vector

class bar:
    '''Create a bar'''

    def create(self,
           dimensions = (2, 1, 1),      # Bar XYZ dimension
           baseLocation = (0., 0., 0.), # Bar base XYZ position
           name = 'Bar'):               # Object name
        '''Create bar mesh and object'''

        ### Calculate base corners
        baseCorners = []
        dx = dimensions[0]
        dy = dimensions[1]
        dz = dimensions[2]
        baseCorners.append((-0.5*dx, -0.5*dy))
        baseCorners.append((-0.5*dx, 0.5*dy))
        baseCorners.append((0.5*dx, 0.5*dy))
        baseCorners.append((0.5*dx, -0.5*dy))

        ### Calculate vertices and faces
        vertexList = []
        topCapVertexList = []
        bottomCapVertexList = []
        faceList = []
        for c in baseCorners:
            vertexList.append((c[0], c[1], 0))
            vertexList.append((c[0], c[1], dz))
            ### Calculate faces
            lVL = len(vertexList)
            if lVL > 3:
                f = (lVL - 1, lVL - 2, lVL - 4, lVL - 3)
                faceList.append(f)
            ### Append vertex indices for the cap faces
            topCapVertexList.append(lVL - 1)
            bottomCapVertexList.append(lVL - 2)

        ### Append remaining faces
        f = (lVL - 1, lVL - 2, 0, 1)
        faceList.append(f)
        f = tuple(bottomCapVertexList)
        faceList.append(f)
        f = tuple(topCapVertexList)
        faceList.append(f)

        ### Create mesh
        mesh = bpy.data.meshes.new(name)
        ob = bpy.data.objects.new(name, mesh)
        ob.location = Vector((baseLocation[0], baseLocation[1], baseLocation[2]))
        bpy.context.collection.objects.link(ob)
        mesh.from_pydata(vertexList,[],faceList)
        mesh.validate(verbose=True)
        bm = bmesh.new()
        bm.from_mesh(mesh)
        bmesh.ops.recalc_face_normals(bm, faces=bm.faces)
        bmesh.ops.translate(bm, vec = baseLocation, verts = bm.verts)
        bm.to_mesh(mesh)
        mesh.update()

        ### Origin to center of mesh
        bpy.ops.object.select_all(action='DESELECT')
        bpy.context.view_layer.objects.active = ob
        ob.select_set(True)
        bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```

We then define our array's basic parameters.

```
### Array definitions

N_X = 10      # Number of structures along x
N_Y = 10      # Number of structures along y
P_X = 2       # Array period along x
P_Y = 2       # Array period along y
W_X = 1       # Structure dimension along x
W_Y = 1.5     # Structure dimension along y
W_Z = 1       # Structure dimension along z
```

Finally, we simply run the class in a loop to create the array. Because Blender objects must have an unique names, we assign one to each bar based of its position. If we didn't, Blender would append a number to each one, but this method makes them easier to distinguish.

```
### Create array

for x in range(0, N_X):
    for y in range(0, N_Y):
        barName = 'Bar_{:04.0f}_{:04.0f}'.format(x, y)
        bar1 = bar()
        bar1.create(dimensions = (W_X, W_Y, W_Z),
                    baseLocation = [x * P_X, y * P_Y, 0],
                    name = barName)
```



This method can be applied to arrays of structures of any kind.

Cylinder with custom angular resolution

A cylinder is another very commonly used shape. We will also build it vertex by vertex, for which we need the `math` package.

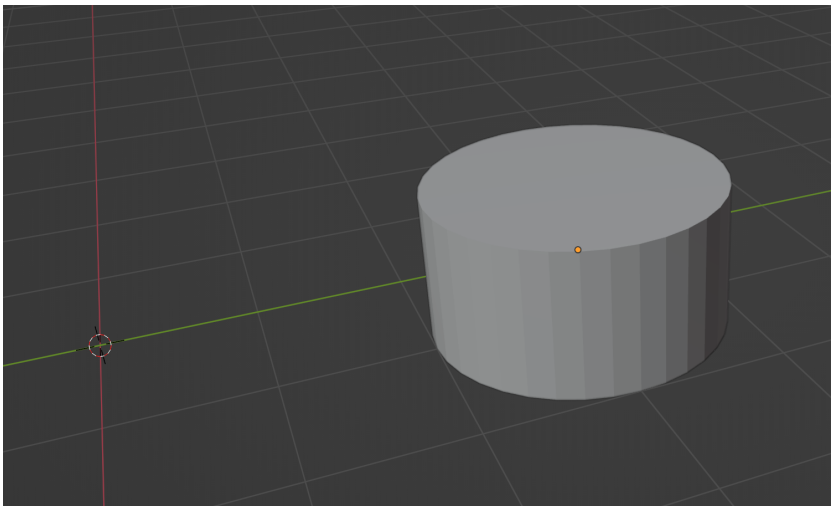
```
import bpy
import bmesh
import math
from mathutils import Vector
```

Our definitions are similar to those for the bar, but we add the choice of the number of points into which to subdivide the circle. This follows the default Blender behavior. With very small structures, you don't need all that many vertices to describe a curve, although you may want to add a few to be on the safe side.

```
baseLocation = (1., 3., 0.)    # Cylinder base XYZ position
bottomRadius = 1               # Cylinder bottom radius
circlePoints = 36              # Circle subdivision points
height = 1                     # Cylinder height
name = 'Cylinder'              # Object name
topRadius = 1                  # Cylinder top radius
```

From here onwards, the vertices can be calculated by basic trigonometry, and the rest of the code follows the bar example closely.

```
### Calculate circle subdivision angle

angle = (2 * math.pi / 360) * (360/circlePoints)

### Calculate vertices

vertexList = []
faceList = []
topCapVertexList = []
bottomCapVertexList = []
for u in range(0, int(circlePoints)):
    ### Outer lower vertex
    xol = bottomRadius * math.cos(u * angle)
    yol = bottomRadius * math.sin(u * angle)
    vol = (xol, yol, 0)
    vertexList.append(vol)
    ### Outer upper vertex
    xou = topRadius * math.cos(u * angle)
    you = topRadius * math.sin(u * angle)
    vou = (xou, you, height)
    vertexList.append(vou)
    ### Calculate faces
    lVL = len(vertexList)
    if lVL > 3:
        f = (lVL - 1, lVL - 2, lVL - 4, lVL - 3)
        faceList.append(f)
    ### Append vertex indices for the cap faces
    topCapVertexList.append(lVL - 1)
    bottomCapVertexList.append(lVL - 2)

### Append remaining faces

f = (lVL - 1, lVL - 2, 0, 1)
faceList.append(f)
f = tuple(bottomCapVertexList)
faceList.append(f)
f = tuple(topCapVertexList)
faceList.append(f)

### Create mesh

mesh = bpy.data.meshes.new(name)
ob = bpy.data.objects.new(name, mesh)
ob.location = Vector((0., 0., 0.))
bpy.context.collection.objects.link(ob)
mesh.from_pydata(vertexList,[],faceList)
mesh.validate(verbose=True)
bm = bmesh.new()
bm.from_mesh(mesh)
bmesh.ops.recalc_face_normals(bm, faces=bm.faces)
bmesh.ops.translate(bm, vec = baseLocation, verts = bm.verts)
bm.to_mesh(mesh)
mesh.update()

### Origin to center of mesh

bpy.ops.object.select_all(action='DESELECT')
bpy.context.view_layer.objects.active = ob
ob.select_set(True)
bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```



More flexibility along z with prisms

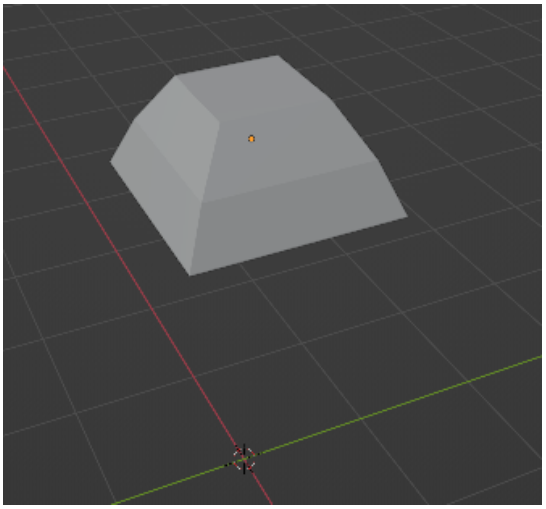
So far, our example structures have had vertical sidewalls, which is often limiting. However, our method of creating geometry vertex by vertex means we can simply provide lists of vertices directly to define an object. We can have a lists of lists in the format `[[], [], ... []]` where each of the inner lists is a sequence of vertices for a layer along `z`. When connected, they yield a "multilayer prism". Layers must all have the same number of points.

This time, for the sake of variety, we will start by defining a class for our prism and creating object via class instances, just as with the bar array example.

```
import bpy
import bmesh
from mathutils import Vector

class prism:
    '''Create a prism with arbitrary base vertices and number of layers'''

    def create(self,
        vertices = [[(-1, -1, 0), (-1, 1, 0), (1, 1, 0), (1, -1, 0)],
                    [(-.5, -.5, 1), (-.5, .5, 1), (.5, .5, 1), (.5, -.5, 1)]], # List of vertices, per layer
        baseLocation = (0., 0., 0.), # Prism base XYZ position
        name = 'Prism'):    # Object name
        '''Create prism mesh and object'''

        ### Calculate vertices
        vertexList = []
        faceList = []
        topCapVertexList = []
        bottomCapVertexList = []
        for li, layer in enumerate(vertices):
            for v in layer:
                vertexList.append(v)
                lVL = len(vertexList)
                if lVL > li * len(layer) + 1:
                    f = (lVL - 1, lVL - 2, lVL - len(layer) - 2, lVL - len(layer) - 1)
                    faceList.append(f)
                # Append vertex indices for the cap faces
                if li == len(vertices) - 1:
                    topCapVertexList.append(lVL - 1)
                elif li == 0:
                    bottomCapVertexList.append(lVL - 1)
            ### Append remaining faces
            f = (lVL - 1, lVL - len(layer), lVL - 2 * len(layer), lVL - len(layer) - 1)
            faceList.append(f)
        f = tuple(bottomCapVertexList)
        faceList.append(f)
        f = tuple(topCapVertexList)
        faceList.append(f)

        ### Create mesh
        mesh = bpy.data.meshes.new(name)
        ob = bpy.data.objects.new(name, mesh)
        ob.location = Vector((0., 0., 0.))
        bpy.context.collection.objects.link(ob)
        mesh.from_pydata(vertexList,[],faceList)
        mesh.validate(verbose=True)
        bm = bmesh.new()
        bm.from_mesh(mesh)
        bmesh.ops.recalc_face_normals(bm, faces=bm.faces)
        bmesh.ops.translate(bm, vec = baseLocation, verts = bm.verts)
        bm.to_mesh(mesh)
        mesh.update()

        ### Origin to center of mesh
        bpy.ops.object.select_all(action='DESELECT')
        bpy.context.view_layer.objects.active = ob
        ob.select_set(True)
        bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```

We test the class by creating an instance and feeding it a list of vertices. We use the class defaults, so specifying the vertices and the `vertices` keyword is redundant, but it serves as one more usage example.

```
prism1 = prism()
vertices = [[(-1, -1, 0), (-1, 1, 0), (1, 1, 0), (1, -1, 0)],
           [(-.8, -.8, .5), (-.8, .8, .5), (.8, .8, .5), (.8, -.8, .5)],
           [(-.5, -.5, 1), (-.5, .5, 1), (.5, .5, 1), (.5, -.5, 1)]]
prism1.create(baseLocation = [-2, 1, 1], vertices = vertices)
```



Useful classes

This section gathers code from the examples above in pre-made classes for ease of use. Also available is an "eraser" class to delete geometry programmatically. Feel free to copy them and paste them in your projects.

Bar

A simple bar.

```
import bpy
import bmesh
from mathutils import Vector

class bar:
    '''Create a bar'''

    def create(self,
            dimensions = (2, 1, 1),      # Bar XYZ dimension
            baseLocation = (0., 0., 0.), # Bar base XYZ position
            name = 'Bar'):               # Object name
        '''Create bar mesh and object'''

        ### Calculate base corners
        baseCorners = []
        dx = dimensions[0]
        dy = dimensions[1]
        dz = dimensions[2]
        baseCorners.append((-0.5*dx, -0.5*dy))
        baseCorners.append((-0.5*dx, 0.5*dy))
        baseCorners.append((0.5*dx, 0.5*dy))
        baseCorners.append((0.5*dx, -0.5*dy))

        ### Calculate vertices and faces
        vertexList = []
        topCapVertexList = []
        bottomCapVertexList = []
        faceList = []
        for c in baseCorners:
            vertexList.append((c[0], c[1], 0))
            vertexList.append((c[0], c[1], dz))
            ### Calculate faces
            lVL = len(vertexList)
            if lVL > 3:
                f = (lVL - 1, lVL - 2, lVL - 4, lVL - 3)
                faceList.append(f)
            ### Append vertex indices for the cap faces
            topCapVertexList.append(lVL - 1)
            bottomCapVertexList.append(lVL - 2)

        ### Append remaining faces
        f = (lVL - 1, lVL - 2, 0, 1)
        faceList.append(f)
        f = tuple(bottomCapVertexList)
        faceList.append(f)
        f = tuple(topCapVertexList)
        faceList.append(f)

        ### Create mesh
        mesh = bpy.data.meshes.new(name)
        ob = bpy.data.objects.new(name, mesh)
        ob.location = Vector((baseLocation[0], baseLocation[1], baseLocation[2]))
        bpy.context.collection.objects.link(ob)
        mesh.from_pydata(vertexList,[],faceList)
        mesh.validate(verbose=True)
        bm = bmesh.new()
        bm.from_mesh(mesh)
        bmesh.ops.recalc_face_normals(bm, faces=bm.faces)
        bmesh.ops.translate(bm, vec = baseLocation, verts = bm.verts)
        bm.to_mesh(mesh)
        mesh.update()

        ### Origin to center of mesh
        bpy.ops.object.select_all(action='DESELECT')
        bpy.context.view_layer.objects.active = ob
        ob.select_set(True)
        bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```

Example:

```
bar1 = bar()
bar1.create(baseLocation = [-1, -1, 1])
```

Cylinder

A simple cylinder.

```
class cylinder:
    '''Create a cylinder or conic section'''

    def create(self,
            baseLocation = (0., 0., 0.),    # Cylinder base XYZ position
            bottomRadius = 1,               # Cylinder bottom radius
            circlePoints = 36,              # Circle subdivision points
            height = 1,                     # Cylinder height
            name = 'Cylinder',                  # Object name
            topRadius = 1):                 # Cylinder top radius
        '''Create cylinder mesh and object'''

        ### Calculate circle subdivision angle
        angle = (2 * math.pi / 360) * (360/circlePoints)

        ### Calculate vertices
        vertexList = []
        faceList = []
        topCapVertexList = []
        bottomCapVertexList = []
        for u in range(0, int(circlePoints)):
            ### Outer lower vertex
            xol = bottomRadius * math.cos(u * angle)
            yol = bottomRadius * math.sin(u * angle)
            vol = (xol, yol, 0)
            vertexList.append(vol)
            ### Outer upper vertex
            xou = topRadius * math.cos(u * angle)
            you = topRadius * math.sin(u * angle)
            vou = (xou, you, height)
            vertexList.append(vou)
            ### Calculate faces
            lVL = len(vertexList)
            if lVL > 3:
                f = (lVL - 1, lVL - 2, lVL - 4, lVL - 3)
                faceList.append(f)
            ### Append vertex indices for the cap faces
            topCapVertexList.append(lVL - 1)
            bottomCapVertexList.append(lVL - 2)

        ### Append remaining faces
        f = (lVL - 1, lVL - 2, 0, 1)
        faceList.append(f)
        f = tuple(bottomCapVertexList)
        faceList.append(f)
        f = tuple(topCapVertexList)
        faceList.append(f)

        ### Create mesh
        mesh = bpy.data.meshes.new(name)
        ob = bpy.data.objects.new(name, mesh)
        ob.location = Vector((0., 0., 0.))
        bpy.context.collection.objects.link(ob)
        mesh.from_pydata(vertexList,[],faceList)
        mesh.validate(verbose=True)
        bm = bmesh.new()
        bm.from_mesh(mesh)
        bmesh.ops.recalc_face_normals(bm, faces=bm.faces)
        bmesh.ops.translate(bm, vec = baseLocation, verts = bm.verts)
        bm.to_mesh(mesh)
        mesh.update()

        ### Origin to center of mesh
        bpy.ops.object.select_all(action='DESELECT')
        bpy.context.view_layer.objects.active = ob
        ob.select_set(True)
        bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```

Example:

```
cyl1 = cylinder()
cyl1.create(bottomRadius = 1, height = 3)
```

Eraser

Should, in most instances, clean a `.blend` file of all objects, collections and meshes.

```
import bpy

class eraser:
    '''Remove Blender objects'''

    def delete_all_objects(self):
        '''Delete all objects in the scene'''
        for o in bpy.context.scene.objects:
            o.select_set(True)
            bpy.ops.object.delete()

    def delete_all_collections(self):
        '''Delete all collections'''
        for c in bpy.data.collections:
            collection = bpy.data.collections.get(c.name)
            bpy.data.collections.remove(collection)

    def delete_orphaned_meshes(self):
        '''Delete all orphaned meshes'''
        orphan_mesh = [m for m in bpy.data.meshes if not m.users]
        while orphan_mesh:
            bpy.data.meshes.remove(orphan_mesh.pop())
```

Example:

```
eraser1 = eraser()
eraser1.delete_all_objects()
eraser1.delete_all_collections()
eraser1.delete_orphaned_meshes()
```

Prism

A multi-layer prism.

```
import bpy
import bmesh
from mathutils import Vector

class prism:
    '''Create a prism with arbitrary base vertices and number of layers'''

    def create(self,
        vertices = [[(-1, -1, 0), (-1, 1, 0), (1, 1, 0), (1, -1, 0)],
                    [(-.5, -.5, 1), (-.5, .5, 1), (.5, .5, 1), (.5, -.5, 1)]], # List of vertices, per layer
        baseLocation = (0., 0., 0.), # Prism base XYZ position
        name = 'Prism'):    # Object name
        '''Create prism mesh and object'''

        ### Calculate vertices
        vertexList = []
        faceList = []
        topCapVertexList = []
        bottomCapVertexList = []
        for li, layer in enumerate(vertices):
            for v in layer:
                vertexList.append(v)
                lVL = len(vertexList)
                if lVL > li * len(layer) + 1:
                    f = (lVL - 1, lVL - 2, lVL - len(layer) - 2, lVL - len(layer) - 1)
                    faceList.append(f)
                # Append vertex indices for the cap faces
                if li == len(vertices) - 1:
                    topCapVertexList.append(lVL - 1)
                elif li == 0:
                    bottomCapVertexList.append(lVL - 1)
            ### Append remaining faces
            f = (lVL - 1, lVL - len(layer), lVL - 2 * len(layer), lVL - len(layer) - 1)
            faceList.append(f)
        f = tuple(bottomCapVertexList)
        faceList.append(f)
        f = tuple(topCapVertexList)
        faceList.append(f)

        ### Create mesh
        mesh = bpy.data.meshes.new(name)
        ob = bpy.data.objects.new(name, mesh)
        ob.location = Vector((0., 0., 0.))
        bpy.context.collection.objects.link(ob)
        mesh.from_pydata(vertexList,[],faceList)
        mesh.validate(verbose=True)
        bm = bmesh.new()
        bm.from_mesh(mesh)
        bmesh.ops.recalc_face_normals(bm, faces=bm.faces)
        bmesh.ops.translate(bm, vec = baseLocation, verts = bm.verts)
        bm.to_mesh(mesh)
        mesh.update()

        ### Origin to center of mesh
        bpy.ops.object.select_all(action='DESELECT')
        bpy.context.view_layer.objects.active = ob
        ob.select_set(True)
        bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='MEDIAN')
```

Example:

```
prism1 = prism()
vertices = [[(-1, -1, 0), (-1, 1, 0), (1, 1, 0), (1, -1, 0)],
           [(-.8, -.8, .5), (-.8, .8, .5), (.8, .8, .5), (.8, -.8, .5)],
           [(-.5, -.5, 1), (-.5, .5, 1), (.5, .5, 1), (.5, -.5, 1)]]
prism1.create(baseLocation = [-2, 1, 1], vertices = vertices)
```

Alternatives

Dedicated CAD software, if available, is always the best solution for creating precise geometries. Many can export to STL of DXF and may have scripting facilities, too. Alternatively, if your layout is scripted but in GDSII (for example by using our guide it can be converted to STL by using for example gds3extrude.