List of Examples

3.10.1 Simple Rectangular Plate

Just about the simplest possible example, a rectangular box

```
result = cadquery.Workplane("front").box(2.0, 2.0, 0.5)
```

Api References

- *Workplane()* !
- *Workplane.box()* !

3.10.2 Plate with Hole

A rectangular box, but with a hole added.

">Z" selects the top most face of the resulting box. The hole is located in the center because the default origin of a working plane is the projected origin of the last Workplane, the last Workplane having origin at (0,0,0) the projection is at the center of the face. The default hole depth is through the entire part.

```
# The dimensions of the box. These can be modified rather than changing the
# object's code directly.
length = 80.0
height = 60.0
thickness = 10.0
center_hole_dia = 22.0

# Create a box based on the dimensions above and add a 22mm center hole
result = (
    cq.Workplane("XY")
    .box(length, height, thickness)
    .faces(">Z")
    .workplane()
    .hole(center_hole_dia)
)
```

Api References

- *Workplane.hole()* !
- *Workplane.box()*
- *Workplane.box()*

3.10.3 An extruded prismatic solid

Build a prismatic solid using extrusion. After a drawing operation, the center of the previous object is placed on the stack, and is the reference for the next operation. So in this case, the rect() is drawn centered on the previously draw circle.

By default, rectangles and circles are centered around the previous working point.

```
result = cq.Workplane("front").circle(2.0).rect(0.5, 0.75).extrude(0.5)
```

Api References

- *Workplane.circle()*!
- *Workplane.rect()*!
- *Workplane.extrude()*!
- *Workplane()*

3.10.4 Building Profiles using lines and arcs

Sometimes you need to build complex profiles using lines and arcs. This example builds a prismatic solid from 2D operations.

2D operations maintain a current point, which is initially at the origin. Use close() to finish a closed curve.

```
result = (
    cq.Workplane("front")
    .lineTo(2.0, 0)
    .lineTo(2.0, 1.0)
    .threePointArc((1.0, 1.5), (0.0, 1.0))
    .close()
    .extrude(0.25)
)
```

Api References

- *Workplane.threePointArc()*!
- *Workplane.lineTo()*!
- *Workplane.extrude()*
- *Workplane()*

3.10.5 Moving The Current working point

In this example, a closed profile is required, with some interior features as well.

This example also demonstrates using multiple lines of code instead of longer chained commands, though of course in this case it was possible to do it in one long line as well.

A new work plane center can be established at any point.

```
result = cq.Workplane("front").circle(
    3.0
)  # current point is the center of the circle, at (0, 0)
result = result.center(1.5, 0.0).rect(0.5, 0.5)  # new work center is (1.5, 0.0)

result = result.center(-1.5, 1.5).circle(0.25)  # new work center is (0.0, 1.5).
# The new center is specified relative to the previous center, not global coordinates!

result = result.extrude(0.25)
```

Api References

- *Workplane.center()*!
- *Workplane()*
- *Workplane.circle()*
- *Workplane.rect()*
- *Workplane.extrude()*

3.10.6 Using Point Lists

Sometimes you need to create a number of features at various locations, and using *Workplane.center()* is too cumbersome.

You can use a list of points to construct multiple objects at once. Most construction methods, like *Workplane.circle()* and *Workplane.rect()*, will operate on multiple points if they are on the stack

```
r = cq.Workplane("front").circle(2.0)  # make base
r = r.pushPoints(
    [(1.5, 0), (0, 1.5), (-1.5, 0), (0, -1.5)]
)  # now four points are on the stack
r = r.circle(0.25)  # circle will operate on all four points
result = r.extrude(0.125)  # make prism
```

Api References

- *Workplane.pushPoints()*!
- *Workplane()*
- *Workplane.circle()*
- *Workplane.extrude()*

3.10.7 Polygons

You can create polygons for each stack point if you would like. Useful in 3d printers whose firmware does not correct for small hole sizes.

```
result = (
    cq.Workplane("front")
    .box(3.0, 4.0, 0.25)
    .pushPoints([(0, 0.75), (0, -0.75)])
    .polygon(6, 1.0)
    .cutThruAll()
)
```

Api References

- *Workplane.polygon()*!
- *Workplane.pushPoints()*
- *Workplane.box()*

3.10.8 Polylines

Workplane.polyline() allows creating a shape from a large number of chained points connected by lines.

This example uses a polyline to create one half of an i-beam shape, which is mirrored to create the final profile.

```
(L, H, W, t) = (100.0, 20.0, 20.0, 1.0)
pts = [
    (0, H / 2.0),
    (W / 2.0, H / 2.0),
    (W / 2.0, (H / 2.0 - t)),
    (t / 2.0, (H / 2.0 - t)),
    (t / 2.0, (t - H / 2.0)),
    (W / 2.0, (t - H / 2.0)),
    (W / 2.0, H / -2.0),
    (0, H / -2.0),
]
result = cq.Workplane("front").polyline(pts).mirrorY().extrude(L)
```

Api References

- *Workplane.polyline()* !
- *Workplane()*
- *Workplane.mirrorY()*
- *Workplane.extrude()*

3.10.9 Defining an Edge with a Spline

This example defines a side using a spline curve through a collection of points. Useful when you have an edge that needs a complex profile

```
s = cq.Workplane("XY")
sPnts = [
    (2.75, 1.5),
    (2.5, 1.75),
    (2.0, 1.5),
    (1.5, 1.0),
    (1.0, 1.25),
    (0.5, 1.0),
    (0, 1.0),
]
r = s.lineTo(3.0, 0).lineTo(3.0, 1.0).spline(sPnts, includeCurrent=True).close()
result = r.extrude(0.5)
```

Api References

- *Workplane.spline()* !
- *Workplane()*
- *Workplane.close()*
- *Workplane.lineTo()*
- *Workplane.extrude()*

3.10.10 Mirroring Symmetric Geometry

You can mirror 2D geometry when your shape is symmetric. In this example we also introduce horizontal and vertical lines, which make for slightly easier coding.

```
r = cq.Workplane("front").hLine(1.0)  # 1.0 is the distance, not coordinate
r = (
    r.vLine(0.5).hLine(-0.25).vLine(-0.25).hLineTo(0.0)
)  # hLineTo allows using xCoordinate not distance
result = r.mirrorY().extrude(0.25)  # mirror the geometry and extrude
```

Api References

- *Workplane.hLine()*!
- *Workplane.vLine()*!
- *Workplane.hLineTo()*!
- *Workplane.mirrorY()*!
- *Workplane.mirrorX()*!
- *Workplane()*
- *Workplane.extrude()*

3.10.11 Mirroring 3D Objects

```
result0 = (
    cadquery.Workplane("XY")
    .moveTo(10, 0)
    .lineTo(5, 0)
    .threePointArc((3.9393, 0.4393), (3.5, 1.5))
    .threePointArc((3.0607, 2.5607), (2, 3))
    .lineTo(1.5, 3)
    .threePointArc((0.4393, 3.4393), (0, 4.5))
    .lineTo(0, 13.5)
    .threePointArc((0.4393, 14.5607), (1.5, 15))
    .lineTo(28, 15)
    .lineTo(28, 13.5)
    .lineTo(24, 13.5)
    .lineTo(24, 11.5)
    .lineTo(27, 11.5)
    .lineTo(27, 10)
    .lineTo(22, 10)
    .lineTo(22, 13.2)
    .lineTo(14.5, 13.2)
    .lineTo(14.5, 10)
    .lineTo(12.5, 10)
    .lineTo(12.5, 13.2)
    .lineTo(5.5, 13.2)
    .lineTo(5.5, 2)
    .threePointArc((5.793, 1.293), (6.5, 1))
    .lineTo(10, 1)
    .close()
)
result = result0.extrude(100)
```

(continues on next page)

(continued from previous page)

```
result = result.rotate((0, 0, 0), (1, 0, 0), 90)

result = result.translate(result.val().BoundingBox().center.multiply(-1))

mirXY_neg = result.mirror(mirrorPlane="XY", basePointVector=(0, 0, -30))
mirXY_pos = result.mirror(mirrorPlane="XY", basePointVector=(0, 0, 30))
mirZY_neg = result.mirror(mirrorPlane="ZY", basePointVector=(-30, 0, 0))
mirZY_pos = result.mirror(mirrorPlane="ZY", basePointVector=(30, 0, 0))

result = result.union(mirXY_neg).union(mirXY_pos).union(mirZY_neg).union(mirZY_pos)
```

Api References

- *Workplane.moveTo()*
- *Workplane.lineTo()*
- *Workplane.threePointArc()*
- *Workplane.extrude()*
- *Workplane.mirror()*
- *Workplane.union()*
- *Workplane.rotate()*

3.10.12 Mirroring From Faces

This example shows how you can mirror about a selected face. It also shows how the resulting mirrored object can be unioned immediately with the referenced mirror geometry.

```
result = cq.Workplane("XY").line(0, 1).line(1, 0).line(0, -0.5).close().extrude(1)

result = result.mirror(result.faces(">X"), union=True)
```

Api References

- *Workplane.line()*
- *Workplane.close()*
- *Workplane.extrude()*
- *Workplane.faces()*
- *Workplane.mirror()*
- *Workplane.union()*

3.10.13 Creating Workplanes on Faces

This example shows how to locate a new workplane on the face of a previously created feature.

Note: Using workplanes in this way are a key feature of CadQuery. Unlike a typical 3d scripting language, using work planes frees you from tracking the position of various features in variables, and allows the model to adjust itself with removing redundant dimensions

---

The *Workplane.faces()* method allows you to select the faces of a resulting solid. It accepts a selector string or object, that allows you to target a single face, and make a workplane oriented on that face.

Keep in mind that by default the origin of a new workplane is calculated by forming a plane from the selected face and projecting the previous origin onto that plane. This behaviour can be changed through the centerOption argument of

Workplane.workplane().

```
result = cq.Workplane("front").box(2, 3, 0.5)  # make a basic prism
result = (
    result.faces(">Z").workplane().hole(0.5)
)  # find the top-most face and make a hole
```

Api References

- *Workplane.faces()* !
- *StringSyntaxSelector()* !
- *Selectors Reference* !
- *Workplane.workplane()*
- *Workplane.box()*
- *Workplane()*

3.10.14 Locating a Workplane on a vertex

Normally, the *Workplane.workplane()* method requires a face to be selected. But if a vertex is selected **immediately after a face**, *Workplane.workplane()* with the centerOption argument set to CenterOfMass will locate the workplane on the face, with the origin at the vertex instead of at the center of the face

The example also introduces *Workplane.cutThruAll()*, which makes a cut through the entire part, no matter how deep the part is.

```
result = cq.Workplane("front").box(3, 2, 0.5)  # make a basic prism
result = (
    result.faces(">Z").vertices("<XY").workplane(centerOption="CenterOfMass")
)  # select the lower left vertex and make a workplane
result = result.circle(1.0).cutThruAll()  # cut the corner out
```

Api References

- *Workplane.cutThruAll()* !
- *Selectors Reference* !
- *Workplane.vertices()* !
- *Workplane.box()*
- *Workplane()*
- *StringSyntaxSelector()* !

3.10.15 Offset Workplanes

Workplanes do not have to lie exactly on a face. When you make a workplane, you can define it at an offset from an existing face.

This example uses an offset workplane to make a compound object, which is perfectly valid!

```
result = cq.Workplane("front").box(3, 2, 0.5)  # make a basic prism
result = result.faces("<X").workplane(
    offset=0.75
)  # workplane is offset from the object surface
result = result.circle(1.0).extrude(0.5)  # disc
```

Api References

- *Workplane.extrude()*
- *Selectors Reference* !
- *Workplane.box()*
- *Workplane()*

3.10.16 Copying Workplanes

An existing CQ object can copy a workplane from another CQ object.

```
result = (
    cq.Workplane("front")
    .circle(1)
    .extrude(10)  # make a cylinder
    # We want to make a second cylinder perpendicular to the first,
    # but we have no face to base the workplane off
    .copyWorkplane(
        # create a temporary object with the required workplane
        cq.Workplane("right", origin=(-5, 0, 0))
    )
    .circle(1)
    .extrude(10)
)
```

API References

- *Workplane.copyWorkplane()* !
- *Workplane.circle()*
- *Workplane.extrude()*
- *Workplane()*

3.10.17 Rotated Workplanes

You can create a rotated work plane by specifying angles of rotation relative to another workplane

```
result = (
    cq.Workplane("front")
    .box(4.0, 4.0, 0.25)
    .faces(">Z")
    .workplane()
    .transformed(offset=cq.Vector(0, -1.5, 1.0), rotate=cq.Vector(60, 0, 0))
    .rect(1.5, 1.5, forConstruction=True)
    .vertices()
    .hole(0.25)
)
```

Api References

- *Workplane.transformed()* !
- *Workplane.box()*
- *Workplane.rect()*
- *Workplane.faces()*

3.10.18 Using construction Geometry

You can draw shapes to use the vertices as points to locate other features. Features that are used to locate other features, rather than to create them, are called `Construction Geometry`

In the example below, a rectangle is drawn, and its vertices are used to locate a set of holes.

```
result = (
    cq.Workplane("front")
    .box(2, 2, 0.5)
    .faces(">Z")
    .workplane()
    .rect(1.5, 1.5, forConstruction=True)
    .vertices()
    .hole(0.125)
)
```

Api References

- *Workplane.rect()* (forConstruction=True)
- *Selectors Reference*
- *Workplane.workplane()*
- *Workplane.box()*
- *Workplane.hole()*
- *Workplane()*

3.10.19 Shelling To Create Thin features

Shelling converts a solid object into a shell of uniform thickness.

To shell an object and 'hollow out' the inside pass a negative thickness parameter to the *Workplane.shell()* method of a shape.

```
result = cq.Workplane("front").box(2, 2, 2).shell(-0.1)
```

A positive thickness parameter wraps an object with filleted outside edges and the original object will be the 'hollowed out' portion.

```
result = cq.Workplane("front").box(2, 2, 2).shell(0.1)
```

Use face selectors to select a face to be removed from the resulting hollow shape.

```
result = cq.Workplane("front").box(2, 2, 2).faces("+Z").shell(0.1)
```

Multiple faces can be removed using more complex selectors.

```
result = cq.Workplane("front").box(2, 2, 2).faces("+Z or -X or +X").shell(0.1)
```

Api References

- *Workplane.shell()* !
- *Selectors Reference*
- *Workplane.box()*
- *Workplane.faces()*
- *Workplane()*

3.10.20 Making Lofts

A loft is a solid swept through a set of wires. This example creates lofted section between a rectangle and a circular section.

```
result = (
    cq.Workplane("front")
    .box(4.0, 4.0, 0.25)
    .faces(">Z")
    .circle(1.5)
    .workplane(offset=3.0)
    .rect(0.75, 0.5)
    .loft(combine=True)
)
```

Api References

- *Workplane.loft()* !
- *Workplane.box()*
- *Workplane.faces()*
- *Workplane.circle()*
- *Workplane.rect()*

3.10.21 Extruding until a given face

Sometimes you will want to extrude a wire until a given face that can be not planar or where you might not know easily the distance you have to extrude to. In such cases you can use *next*, *last* or even give a *Face* object for the *until* argument of *extrude()*.

```
result = (
    cq.Workplane(origin=(20, 0, 0))
    .circle(2)
    .revolve(180, (-20, 0, 0), (-20, -1, 0))
    .center(-20, 0)
    .workplane()
    .rect(20, 4)
    .extrude("next")
)
```

The same behaviour is available with *cutBlind()* and as you can see it is also possible to work on several *Wire* objects at a time (the same is true for *extrude()*).

```
skyscrapers_locations = [(-16, 1), (-8, 0), (7, 0.2), (17, -1.2)]
angles = iter([15, 0, -8, 10])
skyscrapers = (
    cq.Workplane()
    .pushPoints(skyscrapers_locations)
    .eachpoint(
        lambda loc: (
            cq.Workplane()
            .rect(5, 16)
            .workplane(offset=10)
            .ellipse(3, 8)
```

(continues on next page)

(continued from previous page)

```
            .workplane(offset=10)
            .slot2D(20, 5, 90)
            .loft()
            .rotateAboutCenter((0, 0, 1), next(angles))
            .val()
            .located(loc)
        )
    )
)

result = (
    skyscrapers.transformed((0, -90, 0))
    .moveTo(15, 0)
    .rect(3, 3, forConstruction=True)
    .vertices()
    .circle(1)
    .cutBlind("last")
)
```

Here is a typical situation where extruding and cuting until a given surface is very handy. It allows us to extrude or cut until a curved surface without overlapping issues.

```
import cadquery as cq

sphere = cq.Workplane().sphere(5)
base = cq.Workplane(origin=(0, 0, -2)).box(12, 12, 10).cut(sphere).edges("|Z").fillet(2)
sphere_face = base.faces(">>X[2] and (not |Z) and (not |Y)").val()
base = base.faces("<Z").workplane().circle(2).extrude(10)

shaft = cq.Workplane().sphere(4.5).circle(1.5).extrude(20)

spherical_joint = (
    base.union(shaft)
    .faces(">X")
    .workplane(centerOption="CenterOfMass")
    .move(0, 4)
    .slot2D(10, 2, 90)
    .cutBlind(sphere_face)
    .workplane(offset=10)
    .move(0, 2)
    .circle(0.9)
    .extrude("next")
)

result = spherical_joint
```

Warning: If the wire you want to extrude cannot be fully projected on the target surface, the result will be unpredictable. Furthermore, the algorithm in charge of finding the candidate faces does its search by counting all the faces intersected by a line created from your wire center along your extrusion direction. So make sure your wire can be projected on your target face to avoid unexpected behaviour.

Api References

- *Workplane.cutBlind()*!
- *Workplane.rect()*
- *Workplane.ellipse()*
- *Workplane.workplane()*
- *Workplane.slot2D()*
- *Workplane.loft()*
- *Workplane.rotateAboutCenter()*
- *Workplane.transformed()*
- *Workplane.moveTo()*
- *Workplane.circle()*

3.10.22 Making Counter-bored and Counter-sunk Holes

Counterbored and countersunk holes are so common that CadQuery creates macros to create them in a single step.

Similar to *Workplane.hole()*, these functions operate on a list of points as well as a single point.

```
result = (
    cq.Workplane(cq.Plane.XY())
    .box(4, 2, 0.5)
    .faces(">Z")
    .workplane()
    .rect(3.5, 1.5, forConstruction=True)
    .vertices()
    .cboreHole(0.125, 0.25, 0.125, depth=None)
)
```

Api References

- *Workplane.cboreHole()*!
- *Workplane.cskHole()*!
- *Workplane.box()*
- *Workplane.rect()*
- *Workplane.workplane()*
- *Workplane.vertices()*
- *Workplane.faces()*
- *Workplane()*

3.10.23 Offsetting wires in 2D

Two dimensional wires can be transformed with *Workplane.offset2D()*. They can be offset inwards or outwards, and with different techniques for extending the corners.

```
original = cq.Workplane().polygon(5, 10).extrude(0.1).translate((0, 0, 2))
arc = cq.Workplane().polygon(5, 10).offset2D(1, "arc").extrude(0.1).translate((0, 0, 1))
intersection = cq.Workplane().polygon(5, 10).offset2D(1, "intersection").extrude(0.1)
result = original.add(arc).add(intersection)
```

Using the forConstruction argument you can do the common task of offsetting a series of bolt holes from the outline of an object. Here is the counterbore example from above but with the bolt holes offset from the edges.

```
result = (
    cq.Workplane()
    .box(4, 2, 0.5)
    .faces(">Z")
    .edges()
```

(continues on next page)

(continued from previous page)

```
    .toPending()
    .offset2D(-0.25, forConstruction=True)
    .vertices()
    .cboreHole(0.125, 0.25, 0.125, depth=None)
)
```

Note that *Workplane.edges()* is for selecting objects. It does not add the selected edges to pending edges in the modelling context, because this would result in your next extrusion including everything you had only selected in addition to the lines you had drawn. To specify you want these edges to be used in *Workplane.offset2D()*, you call *Workplane.toPending()* to explicitly put them in the list of pending edges.

Api References

- *Workplane.offset2D()* !
- *Workplane.cboreHole()*
- *Workplane.cskHole()*
- *Workplane.box()*
- *Workplane.polygon()*
- *Workplane.workplane()*
- *Workplane.vertices()*
- *Workplane.edges()*
- *Workplane.faces()*
- *Workplane()*

3.10.24 Rounding Corners with Fillet

Filleting is done by selecting the edges of a solid, and using the fillet function.

Here we fillet all of the edges of a simple plate.

```
result = cq.Workplane("XY").box(3, 3, 0.5).edges("|Z").fillet(0.125)
```

Api References

- *Workplane.fillet()* !
- *Workplane.box()*
- *Workplane.edges()*
- *Workplane()*

3.10.25 Tagging objects

The *Workplane.tag()* method can be used to tag a particular object in the chain with a string, so that it can be referred to later in the chain.

The *Workplane.workplaneFromTagged()* method applies *Workplane.copyWorkplane()* to a tagged object. For example, when extruding two different solids from a surface, after the first solid is extruded it can become difficult to reselect the original surface with CadQuery's other selectors.

```
result = (
    cq.Workplane("XY")
    # create and tag the base workplane
    .box(10, 10, 10)
    .faces(">Z")
    .workplane()
```

(continues on next page)

(continued from previous page)

```
    .tag("baseplane")
    # extrude a cylinder
    .center(-3, 0)
    .circle(1)
    .extrude(3)
    # to reselect the base workplane, simply
    .workplaneFromTagged("baseplane")
    # extrude a second cylinder
    .center(3, 0)
    .circle(1)
    .extrude(2)
)
```

Tags can also be used with most selectors, including *Workplane.vertices()*, *Workplane.faces()*, *Workplane.edges()*, *Workplane.wires()*, *Workplane.shells()*, *Workplane.solids()* and *Workplane.compounds()*.

```
result = (
    cq.Workplane("XY")
    # create a triangular prism and tag it
    .polygon(3, 5)
    .extrude(4)
    .tag("prism")
    # create a sphere that obscures the prism
    .sphere(10)
    # create features based on the prism's faces
    .faces("<X", tag="prism")
    .workplane()
    .circle(1)
    .cutThruAll()
    .faces(">X", tag="prism")
    .faces(">Y")
    .workplane()
    .circle(1)
    .cutThruAll()
)
```

Api References

- *Workplane.tag()*!
- Workplane.getTagged()!
- *Workplane.workplaneFromTagged()*!
- *Workplane.extrude()*
- *Workplane.cutThruAll()*
- *Workplane.circle()*
- *Workplane.faces()*
- *Workplane()*

3.10.26 A Parametric Bearing Pillow Block

Combining a few basic functions, its possible to make a very good parametric bearing pillow block, with just a few lines of code.

```
(length, height, bearing_diam, thickness, padding) = (30.0, 40.0, 22.0, 10.0, 8.0)

result = (
    cq.Workplane("XY")
    .box(length, height, thickness)
    .faces(">Z")
    .workplane()
    .hole(bearing_diam)
    .faces(">Z")
    .workplane()
    .rect(length - padding, height - padding, forConstruction=True)
    .vertices()
    .cboreHole(2.4, 4.4, 2.1)
)
```

3.10.27 Splitting an Object

You can split an object using a workplane, and retain either or both halves

```
c = cq.Workplane("XY").box(1, 1, 1).faces(">Z").workplane().circle(0.25).cutThruAll()

# now cut it in half sideways
result = c.faces(">Y").workplane(-0.5).split(keepTop=True)
```

Api References

- *Workplane.split()*!
- *Workplane.box()*
- *Workplane.circle()*
- *Workplane.cutThruAll()*
- *Workplane.workplane()*
- *Workplane()*

3.10.28 The Classic OCC Bottle

CadQuery is based on the OpenCascade.org (OCC) modeling Kernel. Those who are familiar with OCC know about the famous 'bottle' example. The bottle example in the OCCT online documentation.

A pythonOCC version is listed here.

Of course one difference between this sample and the OCC version is the length. This sample is one of the longer ones at 13 lines, but that's very short compared to the pythonOCC version, which is 10x longer!

```
(L, w, t) = (20.0, 6.0, 3.0)
s = cq.Workplane("XY")

# Draw half the profile of the bottle and extrude it
p = (
```

(continues on next page)

(continued from previous page)

```
    s.center(-L / 2.0, 0)
    .vLine(w / 2.0)
    .threePointArc((L / 2.0, w / 2.0 + t), (L, w / 2.0))
    .vLine(-w / 2.0)
    .mirrorX()
    .extrude(30.0, True)
)

# Make the neck
p = p.faces(">Z").workplane(centerOption="CenterOfMass").circle(3.0).extrude(2.0, True)

# Make a shell
result = p.faces(">Z").shell(0.3)
```

Api References

- *Workplane.extrude()*
- *Workplane.mirrorX()*
- *Workplane.threePointArc()*
- *Workplane.workplane()*
- *Workplane.vertices()*
- *Workplane.vLine()*
- *Workplane.faces()*
- *Workplane()*

3.10.29 A Parametric Enclosure

```
# parameter definitions
p_outerWidth = 100.0  # Outer width of box enclosure
p_outerLength = 150.0  # Outer length of box enclosure
p_outerHeight = 50.0  # Outer height of box enclosure

p_thickness = 3.0  # Thickness of the box walls
p_sideRadius = 10.0  # Radius for the curves around the sides of the box
p_topAndBottomRadius = (
    2.0  # Radius for the curves on the top and bottom edges of the box
)

p_screwpostInset = 12.0  # How far in from the edges the screw posts should be place.
p_screwpostID = 4.0  # Inner Diameter of the screw post holes, should be roughly screw␣
↪diameter not including threads
p_screwpostOD = 10.0  # Outer Diameter of the screw posts.\nDetermines overall thickness␣
↪of the posts

p_boreDiameter = 8.0  # Diameter of the counterbore hole, if any
p_boreDepth = 1.0  # Depth of the counterbore hole, if
p_countersinkDiameter = 0.0  # Outer diameter of countersink. Should roughly match the␣
↪outer diameter of the screw head
p_countersinkAngle = 90.0  # Countersink angle (complete angle between opposite sides,␣
↪not from center to one side)
p_flipLid = True  # Whether to place the lid with the top facing down or not.
p_lipHeight = 1.0  # Height of lip on the underside of the lid.\nSits inside the box␣
↪body for a snug fit.
```

(continues on next page)

(continued from previous page)

```
# outer shell
oshell = (
    cq.Workplane("XY")
    .rect(p_outerWidth, p_outerLength)
    .extrude(p_outerHeight + p_lipHeight)
)

# weird geometry happens if we make the fillets in the wrong order
if p_sideRadius > p_topAndBottomRadius:
    oshell = oshell.edges("|Z").fillet(p_sideRadius)
    oshell = oshell.edges("#Z").fillet(p_topAndBottomRadius)
else:
    oshell = oshell.edges("#Z").fillet(p_topAndBottomRadius)
    oshell = oshell.edges("|Z").fillet(p_sideRadius)

# inner shell
ishell = (
    oshell.faces("<Z")
    .workplane(p_thickness, True)
    .rect((p_outerWidth - 2.0 * p_thickness), (p_outerLength - 2.0 * p_thickness))
    .extrude(
        (p_outerHeight - 2.0 * p_thickness), False
    )  # set combine false to produce just the new boss
)
ishell = ishell.edges("|Z").fillet(p_sideRadius - p_thickness)

# make the box outer box
box = oshell.cut(ishell)

# make the screw posts
POSTWIDTH = p_outerWidth - 2.0 * p_screwpostInset
POSTLENGTH = p_outerLength - 2.0 * p_screwpostInset

box = (
    box.faces(">Z")
    .workplane(-p_thickness)
    .rect(POSTWIDTH, POSTLENGTH, forConstruction=True)
    .vertices()
    .circle(p_screwpostOD / 2.0)
    .circle(p_screwpostID / 2.0)
    .extrude(-1.0 * (p_outerHeight + p_lipHeight - p_thickness), True)
)

# split lid into top and bottom parts
(lid, bottom) = (
    box.faces(">Z")
    .workplane(-p_thickness - p_lipHeight)
    .split(keepTop=True, keepBottom=True)
    .all()
)  # splits into two solids
```

(continues on next page)

(continued from previous page)

```
# translate the lid, and subtract the bottom from it to produce the lid inset
lowerLid = lid.translate((0, 0, -p_lipHeight))
cutlip = lowerLid.cut(bottom).translate(
    (p_outerWidth + p_thickness, 0, p_thickness - p_outerHeight + p_lipHeight)
)

# compute centers for screw holes
topOfLidCenters = (
    cutlip.faces(">Z")
    .workplane(centerOption="CenterOfMass")
    .rect(POSTWIDTH, POSTLENGTH, forConstruction=True)
    .vertices()
)

# add holes of the desired type
if p_boreDiameter > 0 and p_boreDepth > 0:
    topOfLid = topOfLidCenters.cboreHole(
        p_screwpostID, p_boreDiameter, p_boreDepth, 2.0 * p_thickness
    )
elif p_countersinkDiameter > 0 and p_countersinkAngle > 0:
    topOfLid = topOfLidCenters.cskHole(
        p_screwpostID, p_countersinkDiameter, p_countersinkAngle, 2.0 * p_thickness
    )
else:
    topOfLid = topOfLidCenters.hole(p_screwpostID, 2.0 * p_thickness)

# flip lid upside down if desired
if p_flipLid:
    topOfLid = topOfLid.rotateAboutCenter((1, 0, 0), 180)

# return the combined result
result = topOfLid.union(bottom)
```

Api References

- *Workplane.circle()*
- *Workplane.rect()*
- *Workplane.extrude()*
- *Workplane.box()*
- *Workplane.all()*
- *Workplane.faces()*
- *Workplane.vertices()*
- *Workplane.edges()*
- *Workplane.workplane()*
- *Workplane.fillet()*
- *Workplane.cut()*
- *Workplane.union()*
- *Workplane.rotateAboutCenter()*
- *Workplane.cboreHole()*
- *Workplane.cskHole()*
- *Workplane.hole()*

3.10.30 Lego Brick

This script will produce any size regular rectangular Lego(TM) brick. Its only tricky because of the logic regarding the underside of the brick.

```
#####
# Inputs
######
lbumps = 6  # number of bumps long
wbumps = 2  # number of bumps wide
thin = True  # True for thin, False for thick

#
# Lego Brick Constants-- these make a Lego brick a Lego :)
#
pitch = 8.0
clearance = 0.1
bumpDiam = 4.8
bumpHeight = 1.8
if thin:
    height = 3.2
else:
    height = 9.6

t = (pitch - (2 * clearance) - bumpDiam) / 2.0
postDiam = pitch - t  # works out to 6.5
total_length = lbumps * pitch - 2.0 * clearance
total_width = wbumps * pitch - 2.0 * clearance

# make the base
s = cq.Workplane("XY").box(total_length, total_width, height)

# shell inwards not outwards
s = s.faces("<Z").shell(-1.0 * t)

# make the bumps on the top
s = (
    s.faces(">Z")
    .workplane()
    .rarray(pitch, pitch, lbumps, wbumps, True)
    .circle(bumpDiam / 2.0)
    .extrude(bumpHeight)
)

# add posts on the bottom. posts are different diameter depending on geometry
# solid studs for 1 bump, tubes for multiple, none for 1x1
tmp = s.faces("<Z").workplane(invert=True)

if lbumps > 1 and wbumps > 1:
    tmp = (
        tmp.rarray(pitch, pitch, lbumps - 1, wbumps - 1, center=True)
        .circle(postDiam / 2.0)
        .circle(bumpDiam / 2.0)
```

(continues on next page)

(continued from previous page)

```
        .extrude(height - t)
    )
elif lbumps > 1:
    tmp = (
        tmp.rarray(pitch, pitch, lbumps - 1, 1, center=True)
        .circle(t)
        .extrude(height - t)
    )
elif wbumps > 1:
    tmp = (
        tmp.rarray(pitch, pitch, 1, wbumps - 1, center=True)
        .circle(t)
        .extrude(height - t)
    )
else:
    tmp = s
```

3.10.31 Braille Example

```
from collections import namedtuple


# text_lines is a list of text lines.
# Braille (converted with braille-converter:
# https://github.com/jpaugh/braille-converter.git).
text_lines = ["          "]
# See http://www.tiresias.org/research/reports/braille_cell.htm for examples
# of braille cell geometry.
horizontal_interdot = 2.5
vertical_interdot = 2.5
horizontal_intercell = 6
vertical_interline = 10
dot_height = 0.5
dot_diameter = 1.3

base_thickness = 1.5

# End of configuration.
BrailleCellGeometry = namedtuple(
    "BrailleCellGeometry",
    (
        "horizontal_interdot",
        "vertical_interdot",
        "intercell",
        "interline",
        "dot_height",
        "dot_diameter",
    ),
)
```

(continues on next page)

(continued from previous page)

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __len__(self):
        return 2

    def __getitem__(self, index):
        return (self.x, self.y)[index]

    def __str__(self):
        return "({}, {})".format(self.x, self.y)


def brailleToPoints(text, cell_geometry):
    # Unicode bit pattern (cf. https://en.wikipedia.org/wiki/Braille_Patterns).
    mask1 = 0b00000001
    mask2 = 0b00000010
    mask3 = 0b00000100
    mask4 = 0b00001000
    mask5 = 0b00010000
    mask6 = 0b00100000
    mask7 = 0b01000000
    mask8 = 0b10000000
    masks = (mask1, mask2, mask3, mask4, mask5, mask6, mask7, mask8)

    # Corresponding dot position
    w = cell_geometry.horizontal_interdot
    h = cell_geometry.vertical_interdot
    pos1 = Point(0, 2 * h)
    pos2 = Point(0, h)
    pos3 = Point(0, 0)
    pos4 = Point(w, 2 * h)
    pos5 = Point(w, h)
    pos6 = Point(w, 0)
    pos7 = Point(0, -h)
    pos8 = Point(w, -h)
    pos = (pos1, pos2, pos3, pos4, pos5, pos6, pos7, pos8)

    # Braille blank pattern (u'\u2800').
    blank = ""
    points = []
    # Position of dot1 along the x-axis (horizontal).
    character_origin = 0
    for c in text:
        for m, p in zip(masks, pos):
            delta_to_blank = ord(c) - ord(blank)
```

(continues on next page)

(continued from previous page)

```
            if m & delta_to_blank:
                points.append(p + Point(character_origin, 0))
        character_origin += cell_geometry.intercell
    return points


def get_plate_height(text_lines, cell_geometry):
    # cell_geometry.vertical_interdot is also used as space between base
    # borders and characters.
    return (
        2 * cell_geometry.vertical_interdot
        + 2 * cell_geometry.vertical_interdot
        + (len(text_lines) - 1) * cell_geometry.interline
    )


def get_plate_width(text_lines, cell_geometry):
    # cell_geometry.horizontal_interdot is also used as space between base
    # borders and characters.
    max_len = max([len(t) for t in text_lines])
    return (
        2 * cell_geometry.horizontal_interdot
        + cell_geometry.horizontal_interdot
        + (max_len - 1) * cell_geometry.intercell
    )


def get_cylinder_radius(cell_geometry):
    """Return the radius the cylinder should have
    The cylinder have the same radius as the half-sphere make the dots (the
    hidden and the shown part of the dots).
    The radius is such that the spherical cap with diameter
    cell_geometry.dot_diameter has a height of cell_geometry.dot_height.
    """
    h = cell_geometry.dot_height
    r = cell_geometry.dot_diameter / 2
    return (r**2 + h**2) / 2 / h


def get_base_plate_thickness(plate_thickness, cell_geometry):
    """Return the height on which the half spheres will sit"""
    return (
        plate_thickness + get_cylinder_radius(cell_geometry) - cell_geometry.dot_height
    )


def make_base(text_lines, cell_geometry, plate_thickness):
    base_width = get_plate_width(text_lines, cell_geometry)
    base_height = get_plate_height(text_lines, cell_geometry)
    base_thickness = get_base_plate_thickness(plate_thickness, cell_geometry)
    base = cq.Workplane("XY").box(
        base_width, base_height, base_thickness, centered=False
```

(continues on next page)

(continued from previous page)

```
    )
    return base


def make_embossed_plate(text_lines, cell_geometry):
    """Make an embossed plate with dots as spherical caps
    Method:
        - make a thin plate on which sit cylinders
        - fillet the upper edge of the cylinders so to get pseudo half-spheres
        - make the union with a thicker plate so that only the sphere caps stay
          "visible".
    """
    base = make_base(text_lines, cell_geometry, base_thickness)

    dot_pos = []
    base_width = get_plate_width(text_lines, cell_geometry)
    base_height = get_plate_height(text_lines, cell_geometry)
    y = base_height - 3 * cell_geometry.vertical_interdot
    line_start_pos = Point(cell_geometry.horizontal_interdot, y)
    for text in text_lines:
        dots = brailleToPoints(text, cell_geometry)
        dots = [p + line_start_pos for p in dots]
        dot_pos += dots
        line_start_pos += Point(0, -cell_geometry.interline)

    r = get_cylinder_radius(cell_geometry)
    base = (
        base.faces(">Z")
        .vertices("<XY")
        .workplane()
        .pushPoints(dot_pos)
        .circle(r)
        .extrude(r)
    )
    # Make a fillet almost the same radius to get a pseudo spherical cap.
    base = base.faces(">Z").edges().fillet(r - 0.001)
    hidding_box = cq.Workplane("XY").box(
        base_width, base_height, base_thickness, centered=False
    )
    result = hidding_box.union(base)
    return result


_cell_geometry = BrailleCellGeometry(
    horizontal_interdot,
    vertical_interdot,
    horizontal_intercell,
    vertical_interline,
    dot_height,
    dot_diameter,
)
```

(continues on next page)

(continued from previous page)

```
if base_thickness < get_cylinder_radius(_cell_geometry):
    raise ValueError("Base thickness should be at least {}".format(dot_height))

result = make_embossed_plate(text_lines, _cell_geometry)
```

3.10.32 Panel With Various Connector Holes

```
# The dimensions of the model. These can be modified rather than changing the
# object's code directly.
width = 400
height = 500
thickness = 2

# Create a plate with two polygons cut through it
result = cq.Workplane("front").box(width, height, thickness)

h_sep = 60
for idx in range(4):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(157, 210 - idx * h_sep)
        .moveTo(-23.5, 0)
        .circle(1.6)
        .moveTo(23.5, 0)
        .circle(1.6)
        .moveTo(-17.038896, -5.7)
        .threePointArc((-19.44306, -4.70416), (-20.438896, -2.3))
        .lineTo(-21.25, 2.3)
        .threePointArc((-20.25416, 4.70416), (-17.85, 5.7))
        .lineTo(17.85, 5.7)
        .threePointArc((20.25416, 4.70416), (21.25, 2.3))
        .lineTo(20.438896, -2.3)
        .threePointArc((19.44306, -4.70416), (17.038896, -5.7))
        .close()
        .cutThruAll()
    )

for idx in range(4):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(157, -30 - idx * h_sep)
        .moveTo(-16.65, 0)
        .circle(1.6)
        .moveTo(16.65, 0)
        .circle(1.6)
        .moveTo(-10.1889, -5.7)
        .threePointArc((-12.59306, -4.70416), (-13.5889, -2.3))
        .lineTo(-14.4, 2.3)
        .threePointArc((-13.40416, 4.70416), (-11, 5.7))
        .lineTo(11, 5.7)
```

(continues on next page)

(continued from previous page)

```
        .threePointArc((13.40416, 4.70416), (14.4, 2.3))
        .lineTo(13.5889, -2.3)
        .threePointArc((12.59306, -4.70416), (10.1889, -5.7))
        .close()
        .cutThruAll()
    )

h_sep4DB9 = 30
for idx in range(8):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(91, 225 - idx * h_sep4DB9)
        .moveTo(-12.5, 0)
        .circle(1.6)
        .moveTo(12.5, 0)
        .circle(1.6)
        .moveTo(-6.038896, -5.7)
        .threePointArc((-8.44306, -4.70416), (-9.438896, -2.3))
        .lineTo(-10.25, 2.3)
        .threePointArc((-9.25416, 4.70416), (-6.85, 5.7))
        .lineTo(6.85, 5.7)
        .threePointArc((9.25416, 4.70416), (10.25, 2.3))
        .lineTo(9.438896, -2.3)
        .threePointArc((8.44306, -4.70416), (6.038896, -5.7))
        .close()
        .cutThruAll()
    )

for idx in range(4):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(25, 210 - idx * h_sep)
        .moveTo(-23.5, 0)
        .circle(1.6)
        .moveTo(23.5, 0)
        .circle(1.6)
        .moveTo(-17.038896, -5.7)
        .threePointArc((-19.44306, -4.70416), (-20.438896, -2.3))
        .lineTo(-21.25, 2.3)
        .threePointArc((-20.25416, 4.70416), (-17.85, 5.7))
        .lineTo(17.85, 5.7)
        .threePointArc((20.25416, 4.70416), (21.25, 2.3))
        .lineTo(20.438896, -2.3)
        .threePointArc((19.44306, -4.70416), (17.038896, -5.7))
        .close()
        .cutThruAll()
    )

for idx in range(4):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(25, -30 - idx * h_sep)
```

(continues on next page)

(continued from previous page)

```
        .moveTo(-16.65, 0)
        .circle(1.6)
        .moveTo(16.65, 0)
        .circle(1.6)
        .moveTo(-10.1889, -5.7)
        .threePointArc((-12.59306, -4.70416), (-13.5889, -2.3))
        .lineTo(-14.4, 2.3)
        .threePointArc((-13.40416, 4.70416), (-11, 5.7))
        .lineTo(11, 5.7)
        .threePointArc((13.40416, 4.70416), (14.4, 2.3))
        .lineTo(13.5889, -2.3)
        .threePointArc((12.59306, -4.70416), (10.1889, -5.7))
        .close()
        .cutThruAll()
    )

for idx in range(8):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(-41, 225 - idx * h_sep4DB9)
        .moveTo(-12.5, 0)
        .circle(1.6)
        .moveTo(12.5, 0)
        .circle(1.6)
        .moveTo(-6.038896, -5.7)
        .threePointArc((-8.44306, -4.70416), (-9.438896, -2.3))
        .lineTo(-10.25, 2.3)
        .threePointArc((-9.25416, 4.70416), (-6.85, 5.7))
        .lineTo(6.85, 5.7)
        .threePointArc((9.25416, 4.70416), (10.25, 2.3))
        .lineTo(9.438896, -2.3)
        .threePointArc((8.44306, -4.70416), (6.038896, -5.7))
        .close()
        .cutThruAll()
    )

for idx in range(4):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(-107, 210 - idx * h_sep)
        .moveTo(-23.5, 0)
        .circle(1.6)
        .moveTo(23.5, 0)
        .circle(1.6)
        .moveTo(-17.038896, -5.7)
        .threePointArc((-19.44306, -4.70416), (-20.438896, -2.3))
        .lineTo(-21.25, 2.3)
        .threePointArc((-20.25416, 4.70416), (-17.85, 5.7))
        .lineTo(17.85, 5.7)
        .threePointArc((20.25416, 4.70416), (21.25, 2.3))
        .lineTo(20.438896, -2.3)
        .threePointArc((19.44306, -4.70416), (17.038896, -5.7))
```

(continues on next page)

(continued from previous page)

```
        .close()
        .cutThruAll()
    )

for idx in range(4):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(-107, -30 - idx * h_sep)
        .circle(14)
        .rect(24.7487, 24.7487, forConstruction=True)
        .vertices()
        .hole(3.2)
        .cutThruAll()
    )

for idx in range(8):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(-173, 225 - idx * h_sep4DB9)
        .moveTo(-12.5, 0)
        .circle(1.6)
        .moveTo(12.5, 0)
        .circle(1.6)
        .moveTo(-6.038896, -5.7)
        .threePointArc((-8.44306, -4.70416), (-9.438896, -2.3))
        .lineTo(-10.25, 2.3)
        .threePointArc((-9.25416, 4.70416), (-6.85, 5.7))
        .lineTo(6.85, 5.7)
        .threePointArc((9.25416, 4.70416), (10.25, 2.3))
        .lineTo(9.438896, -2.3)
        .threePointArc((8.44306, -4.70416), (6.038896, -5.7))
        .close()
        .cutThruAll()
    )

for idx in range(4):
    result = (
        result.workplane(offset=1, centerOption="CenterOfBoundBox")
        .center(-173, -30 - idx * h_sep)
        .moveTo(-2.9176, -5.3)
        .threePointArc((-6.05, 0), (-2.9176, 5.3))
        .lineTo(2.9176, 5.3)
        .threePointArc((6.05, 0), (2.9176, -5.3))
        .close()
        .cutThruAll()
    )
```

3.10.33 Cycloidal gear

You can define complex geometries using the parametricCurve functionality. This specific examples generates a helical cycloidal gear.

```
import cadquery as cq
from math import sin, cos, pi, floor


# define the generating function
def hypocycloid(t, r1, r2):
    return (
        (r1 - r2) * cos(t) + r2 * cos(r1 / r2 * t - t),
        (r1 - r2) * sin(t) + r2 * sin(-(r1 / r2 * t - t)),
    )


def epicycloid(t, r1, r2):
    return (
        (r1 + r2) * cos(t) - r2 * cos(r1 / r2 * t + t),
        (r1 + r2) * sin(t) - r2 * sin(r1 / r2 * t + t),
    )


def gear(t, r1=4, r2=1):
    if (-1) ** (1 + floor(t / 2 / pi * (r1 / r2))) < 0:
        return epicycloid(t, r1, r2)
    else:
        return hypocycloid(t, r1, r2)


# create the gear profile and extrude it
result = (
    cq.Workplane("XY")
    .parametricCurve(lambda t: gear(t * 2 * pi, 6, 1))
    .twistExtrude(15, 90)
    .faces(">Z")
    .workplane()
    .circle(2)
    .cutThruAll()
)
```

3.11 API Reference

The CadQuery API is made up of 4 main objects:

- **Sketch** – Construct 2D sketches
- **Workplane** – Wraps a topological entity and provides a 2D modelling context.
- **Selector** – Filter and select things
- **Assembly** – Combine objects into assemblies.