CadQuery Documentation

Release 2.4.0

David Cowden

Jan 15, 2024

CONTENTS

CadQuery is an intuitive, easy-to-use Python library for building parametric 3D CAD models. It has several goals:

- Build models with scripts that are as close as possible to how you'd describe the object to a human, using a standard, already established programming language
- Create parametric models that can be very easily customized by end users
- Output high quality CAD formats like STEP, AMF and 3MF in addition to traditional STL
- Provide a non-proprietary, plain text model format that can be edited and executed with only a web browser

CHAPTER

ONE

SEE CADQUERY IN ACTION

This Getting Started Video will show you what CadQuery can do. Please note that the video has not been updated for CadQuery 2 and still shows CadQuery use within FreeCAD.

CHAPTER

TWO

QUICK LINKS

- *QuickStart*
- CadQuery CheatSheet
- *API Reference*

CHAPTER

THREE

TABLE OF CONTENTS

3.1 Introduction

3.1.1 What is CadQuery

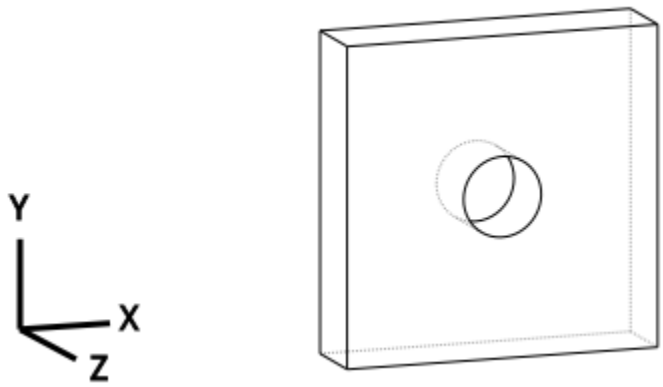
CadQuery is an intuitive, easy-to-use Python library for building parametric 3D CAD models. It has several goals:

- Build models with scripts that are as close as possible to how you'd describe the object to a human, using a standard, already established programming language
- Create parametric models that can be very easily customized by end users
- Output high quality CAD formats like STEP, AMF and 3MF in addition to traditional STL
- Provide a non-proprietary, plain text model format that can be edited and executed with only a web browser

CadQuery 2 is based on OCP, which is a set of Python bindings for the open-source OpenCascade modelling kernel.

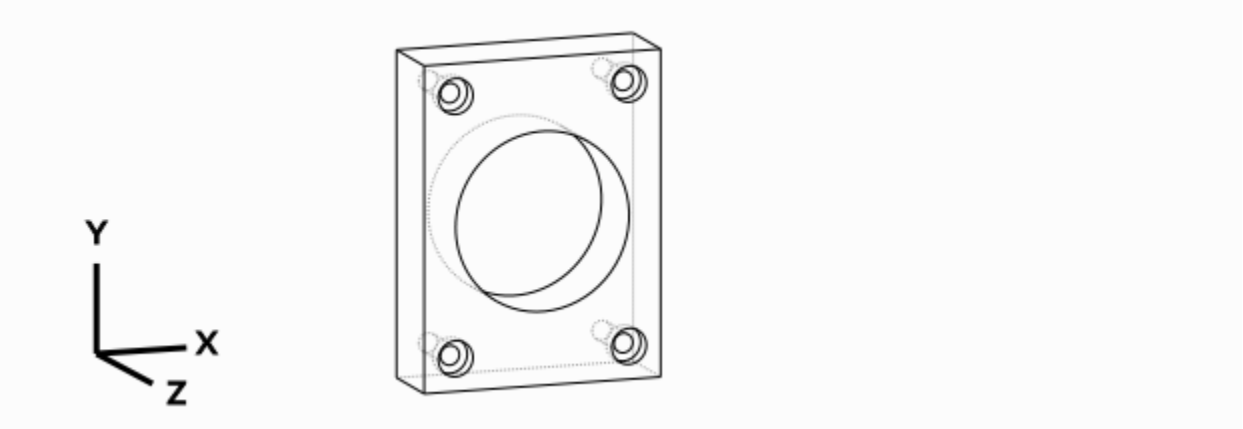
Using CadQuery, you can build fully parametric models with a very small amount of code. For example, this simple script produces a flat plate with a hole in the middle:

```
thickness = 0.5
width = 2.0
result = Workplane("front").box(width, width, thickness).faces(">Z").hole(thickness)
```



That's a bit of a dixie-cup example. But it is pretty similar to a more useful part: a parametric pillow block for a standard 608-size ball bearing:

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

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



Lots more examples are available in the *Examples*

3.1.2 CadQuery is a library, GUIs are separate

CadQuery is a library, that's intentionally designed to be usable as a GUI-less library. This enables its use in a variety of engineering and scientific applications that create 3D models programmatically.

If you'd like a GUI, you have a couple of options:

- The Qt-based GUI CQ-editor
- As a Jupyter extension jupyter-cadquery

3.1.3 Why CadQuery instead of OpenSCAD?

Like OpenSCAD, CadQuery is an open-source, script based, parametric model generator. But CadQuery has several key advantages:

1. **The scripts use a standard programming language**, Python, and thus can benefit from the associated infrastructure. This includes many standard libraries and IDEs
2. **More powerful CAD kernel** OpenCascade is much more powerful than CGAL. Features supported natively by OCC include NURBS, splines, surface sewing, STL repair, STEP import/export, and other complex operations, in addition to the standard CSG operations supported by CGAL

3. **Ability to import/export STEP and DXF** We think the ability to begin with a STEP model, created in a CAD package, and then add parametric features is key. This is possible in OpenSCAD using STL, but STL is a lossy format
4. **Less Code and easier scripting** CadQuery scripts require less code to create most objects, because it is possible to locate features based on the position of other features, workplanes, vertices, etc.
5. **Better Performance** CadQuery scripts can build STL, STEP, AMF and 3MF faster than OpenSCAD.

3.1.4 Where does the name CadQuery come from?

CadQuery is inspired by jQuery, a popular framework that revolutionized web development involving JavaScript.

CadQuery is for 3D CAD what jQuery is for JavaScript. If you are familiar with how jQuery works, you will probably recognize several jQuery features that CadQuery uses:

- A fluent API to create clean, easy to read code
- Ability to use the library along side other Python libraries
- Clear and complete documentation, with plenty of samples.

3.2 Installing CadQuery

To install both Cadquery and CQ-Editor together with a single installer see the instructions below *Adding a Nicer GUI via CQ-editor*.

CadQuery may be installed with either conda or pip. The conda installation method is the better tested and more mature option.

3.2.1 Install via conda

Begin by installing the conda package manager. If conda is already installed skip to *conda*.

Install the Conda Package Manager

In principle, any Conda distribution will work, but it is probably best to install Mambaforge to a local directory and to avoid running *conda init*. After performing a local directory installation, Mambaforge can be activated via the [scripts,bin]/activate scripts. This will help avoid polluting and breaking the local Python installation.

Mambaforge is a minimal installer that sets *conda-forge* as the default channel for package installation and provides mamba. You can swap almost all commands between conda & mamba.

In Linux/MacOS, the local directory installation method looks something like this:

```
# Install to ~/mambaforge
curl -L -o mambaforge.sh "https://github.com/conda-forge/miniforge/releases/latest/
↪download/Mambaforge-$(uname)-$(uname -m).sh"
bash mambaforge.sh -b -p $HOME/mambaforge

# Activate
source $HOME/mambaforge/bin/activate
```

On Windows, download the installer and double click it on the file browser or install non-interactively as follows:

```
:: Install to %USERPROFILE%\Mambaforge
curl -L -o mambaforge.exe https://github.com/conda-forge/miniforge/releases/latest/
↪download/Mambaforge-Windows-x86_64.exe
start /wait "" mambaforge.exe /InstallationType=JustMe /RegisterPython=0 /NoRegistry=1 /
↪NoScripts=1 /S /D=%USERPROFILE%\Mambaforge

:: Activate
cmd /K ""%USERPROFILE%/Mambaforge/Scripts/activate.bat" "%USERPROFILE%/Mambaforge""
```

It might be worthwhile to consider using `/NoScripts=0` to have an activation shortcut added to the start menu.

After conda installation, create and activate a new conda environment to prepare for cadquery installation.

conda

`mamba install` is recommended over `conda install` for faster and less memory intensive cadquery installation.

Install the latest released version of cadquery:

```
conda create -n cq
conda activate cq
mamba install cadquery
```

or install a given version of cadquery[1]:

```
conda create -n cq231
conda activate cq231
mamba install cadquery=2.3.1
```

or install the latest dev version:

```
conda create -n cqdev
conda activate cqdev
mamba install -c cadquery cadquery=master
```

Add the *conda-forge* channel explicitly to the install command if needed (not using a miniforge based conda distribution).

3.2.2 Install via pip

CadQuery has a complex set of dependencies including OCP, which is our set of bindings to the OpenCASCADE CAD kernel. OCP is distributed as binary wheels for Linux, MacOS and Windows. However, there are some limitations. Only Python 3.8 through 3.10 are currently supported, and some older Linux distributions such as Ubuntu 18.04 are not supported. If the pip installation method does not work for your system, you can try the conda installation method.

It is highly recommended that a virtual environment is used when installing CadQuery, although it is not strictly required. Installing CadQuery via pip requires an up-to-date version of pip, which can be obtained with the following command line (or a slight variation thereof).:

[1] Older releases may not be compatible with the latest OCP/OCCT version. In that case, specify the version of the dependency explicitly.

```
mamba install cadquery=2.2.0 ocp=7.7.0.*
```

```
python3 -m pip install --upgrade pip
```

Once a current version of pip is installed, CadQuery can be installed using the following command line.:

```
pip install cadquery
```

It is also possible to install the very latest changes directly from CadQuery's GitHub repository, with the understanding that sometimes breaking changes can occur. To install from the git repository, run the following command line.:

```
pip install git+https://github.com/CadQuery/cadquery.git
```

You should now have a working CadQuery installation, but developers or users who want to use CadQuery with IPython/Jupyter or to set up a developer environment can read the rest of this section.

If you are installing CadQuery to use with IPython/Jupyter, you may want to run the following command line to install the extra dependencies.:

```
pip install cadquery[ipython]
```

If you want to create a developer setup to contribute to CadQuery, the following command line will install all the development dependencies that are needed.:

```
pip install cadquery[dev]
```

3.2.3 Adding a Nicer GUI via CQ-editor

If you prefer to have a GUI available, your best option is to use CQ-editor.

You can download the newest build here. Install and run the *run.sh* (Linux/MacOS) or *run.bat* (Windows) script in the root CQ-editor directory. The CQ-editor window should launch.

Linux/MacOS

1. Download the installer (.sh script matching OS and platform).
2. Select the script in the file browser and make executable. Choose **Properties** from the context menu and select **Permissions**, **Allow executing file as a program** (or similar, this step varies depending on OS and window manager).
3. Select the script in the file browser and choose **Run as Program** (or similar).

   Follow the prompts to accept the license and optionally change the installation location.

   The default installation location is `/home/<username>/cq-editor`.
4. Launch the **run.sh** script from the file brower (again make executable first and then run as program).

To install from command line, download the installer using curl or wget or your favorite program and run the script.:

```
curl -LO https://github.com/CadQuery/CQ-editor/releases/download/nightly/CQ-editor-
↪master-Linux-x86_64.sh
sh CQ-editor-master-Linux-x86_64.sh
```

To run from command.:

```
$HOME/cq-editor/run.sh
```

Windows

1. Download the installer (.exe) and double click it on the file browser.

   Follow the prompts to accept the license and optionally change the installation location.

   The default installation location is `C:\Users\<username>\cq-editor`.

2. Launch the **run.bat** script from the file brower (select **Open**).

To run from command line, activate the environment, then run cq-editor:

```
C:\Users\<username>\cq-editor\run.bat
```

Installing extra packages

mamba, and *pip* are bundled with the CQ-editor installer and available for package installation.

First activate the environment, then call mamba or pip to install additional packages.

On windows.:

```
C:\Users\<username>\cq-editor\Scripts\activate
mamba install <packagename>
```

On Linux/MacOS.:

```
source $HOME/cq-editor/bin/activate
mamba install <packagename>
```

3.2.4 Adding CQ-editor to an Existing Environment

You can install CQ-editor into a conda environment or Python virtual environment using conda (mamba) or pip.

Example cq-editor installation with conda (this installs both cadquery and cq-editor):

```
conda create -n cqdev
conda activate cqdev
mamba install -c cadquery cq-editor=master
```

Example cq-editor installation with pip:

```
pip install PyQt5 spyder pyqtgraph logbook
pip install git+https://github.com/CadQuery/CQ-editor.git
```

3.2.5 Jupyter

Viewing models in Jupyter is another good option for a GUI. Models are rendered in the browser.

The cadquery library works out-of-the-box with Jupyter. First install cadquery, then install JupyterLab in the same conda or Python venv.:

conda

```
mamba install jupyterlab
```

pip

```
pip install jupyterlab
```

Start JupyterLab:

```
jupyter lab
```

JupyterLab will open automatically in your browser. Create a Notebook to interactively edit/view CadQuery models.

Call `display` to show the model.:

```
display(<Workplane, Shape, or Assembly object>)
```

3.2.6 Test Your Installation

If all has gone well, you can open a command line/prompt, and type:

```
$ python
$ import cadquery
$ cadquery.Workplane('XY').box(1,2,3).toSvg()
```

You should see raw SVG output displayed on the command line if the CadQuery installation was successful.

Note:

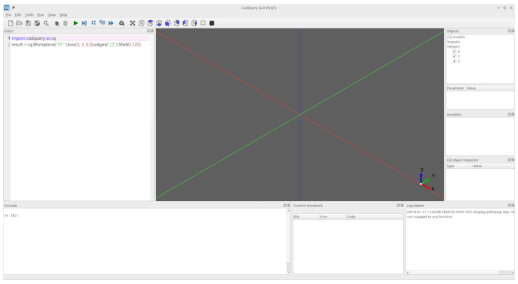
3.3 QuickStart

Want a quick glimpse of what CadQuery can do? This quickstart will demonstrate the basics of CadQuery using a simple example

3.3.1 Prerequisites: CadQuery and CQ-editor installation

If you have not already done so, follow the *Installing CadQuery*, to install CadQuery and CQ-editor.

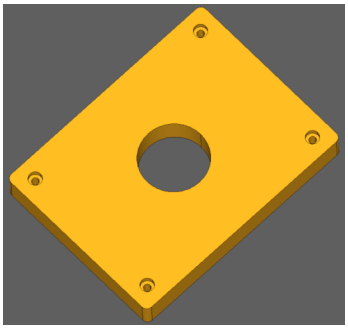
After installation, run CQ-editor:



Find the CadQuery code editor, on the left side. You'll see that we start out with the script for a simple block.

3.3.2 What we'll accomplish

We will build a fully parametric bearing pillow block in this quickstart. Our finished object will look like this:



We would like our block to have these features:

1. It should be sized to hold a single 608 ('skate') bearing, in the center of the block.
2. It should have counter-bored holes for M2 socket head cap screws at the corners.
3. The length and width of the block should be configurable by the user to any reasonable size.

A human would describe this as:

> "A rectangular block 80mm x 60mm x 10mm , with counter-bored holes for M2 socket head cap screws at the corners, and a circular pocket 22mm in diameter in the middle for a bearing."

Human descriptions are very elegant, right? Hopefully our finished script will not be too much more complex than this human-oriented description.

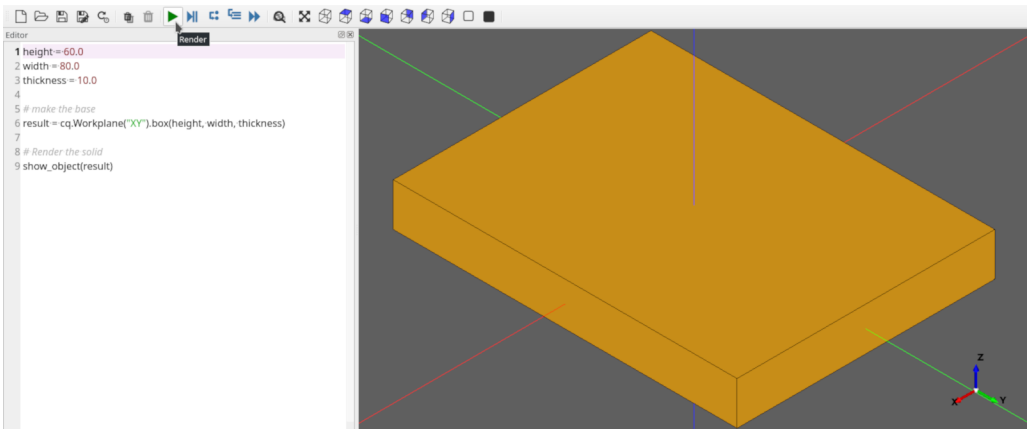
Let's see how we do.

3.3.3 Start With A single, simple Plate

Let's start with a simple model that makes nothing but a rectangular block, but with place-holders for the dimensions. Paste this into the code editor:

```
height = 60.0
width = 80.0
thickness = 10.0

# make the base
result = cq.Workplane("XY").box(height, width, thickness)

# Render the solid
show_object(result)
```

Press the green Render button in the toolbar to run the script. You should see our base object.



Nothing special, but its a start!

3.3.4 Add the Holes

Our pillow block needs to have a 22mm diameter hole in the center to hold the bearing.

This modification will do the trick:

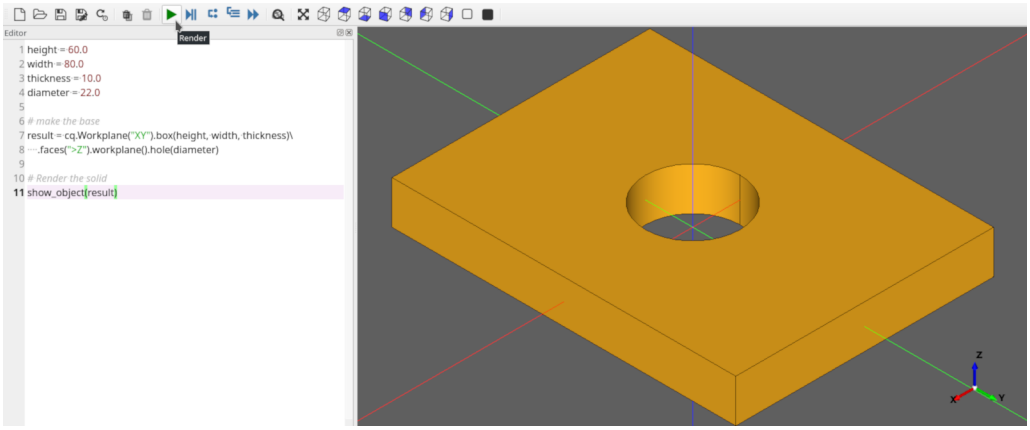
```
height = 60.0
width = 80.0
thickness = 10.0
diameter = 22.0

# make the base
result = (
    cq.Workplane("XY")
    .box(height, width, thickness)
    .faces(">Z")
    .workplane()
    .hole(diameter)
)

```

(continues on next page)

(continued from previous page)

```
15 # Render the solid
16 show_object(result)
```

Rebuild your model by clicking the Render button. Your block should look like this:



The code is pretty compact, let's step through it.

Line 4 adds a new parameter, diameter, for the diameter of the hole

Lines 10-12, we're adding the hole. *cadquery.Workplane.faces()* selects the top-most face in the Z direction, and then *cadquery.Workplane.workplane()* begins a new workplane located on this face. The center of this workplane is located at the center of mass of the shape, which in this case is the center of the plate. Finally, *cadquery.Workplane.hole()* drills a hole through the part, 22mm in diameter.

Note: Don't worry about the CadQuery syntax now.. you can learn all about it in the *API Reference* later.

3.3.5 More Holes

Ok, that hole was not too hard, but what about the counter-bored holes in the corners?

An M2 Socket head cap screw has these dimensions:

- **Head Diameter** : 3.8 mm
- **Head height** : 2.0 mm
- **Clearance Hole** : 2.4 mm
- **CounterBore diameter** : 4.4 mm

The centers of these holes should be 6mm from the edges of the block. And, we want the block to work correctly even when the block is re-sized by the user.

Don't tell me we'll have to repeat the steps above 8 times to get counter-bored holes? Good news!– we can get the job done with just a few lines of code. Here's the code we need:

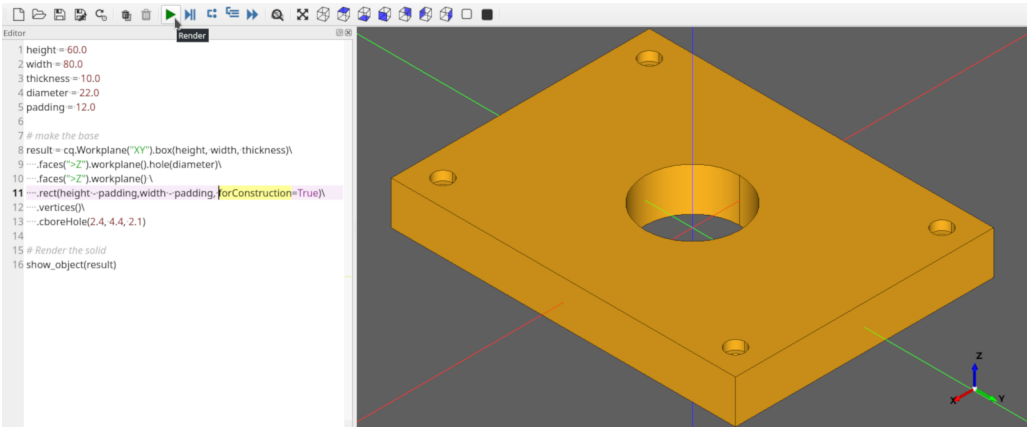
```
1 height = 60.0
2 width = 80.0
3 thickness = 10.0
4 diameter = 22.0
5 padding = 12.0
```

(continues on next page)

(continued from previous page)

```
6
7   # make the base
8   result = (
9       cq.Workplane("XY")
10      .box(height, width, thickness)
11      .faces(">Z")
12      .workplane()
13      .hole(diameter)
14      .faces(">Z")
15      .workplane()
16      .rect(height - padding, width - padding, forConstruction=True)
17      .vertices()
18      .cboreHole(2.4, 4.4, 2.1)
19  )
20  # Render the solid
21  show_object(result)
```

After clicking the Render button to re-execute the model, you should see something like this:



There is quite a bit going on here, so let's break it down a bit.

Line 5 creates a new padding parameter that decides how far the holes are from the edges of the plate.

Lines 11-12 selects the top-most face of the block, and creates a workplane on the top of that face, which we'll use to define the centers of the holes in the corners.

There are a couple of things to note about this line:

1. The *cadquery.Workplane.rect()* function draws a rectangle. **forConstruction=True** tells CadQuery that this rectangle will not form a part of the solid, but we are just using it to help define some other geometry.
2. Unless you specify otherwise, a rectangle is drawn with its center on the current workplane center– in this case, the center of the top face of the block. So this rectangle will be centered on the face.

Line 16 draws a rectangle 12mm smaller than the overall length and width of the block, which we will use to locate the corner holes. We'll use the vertices (corners) of this rectangle to locate the holes. The rectangle's center is at the center of the workplane, which in this case coincides with the center of the bearing hole.

Line 17 selects the vertices of the rectangle, which we will use for the centers of the holes. The *cadquery.Workplane.vertices()* function selects the corners of the rectangle.

Line 18 uses the cboreHole function to draw the holes. The *cadquery.Workplane.cboreHole()* function is a handy CadQuery function that makes a counterbored hole. Like most other CadQuery functions, it operates on the values on

the stack. In this case, since we selected the four vertices before calling the function, the function operates on each of the four points– which results in a counterbore hole at each of the rectangle corners.

3.3.6 Filleting

Almost done. Let's just round the corners of the block a bit. That's easy, we just need to select the edges and then fillet them:

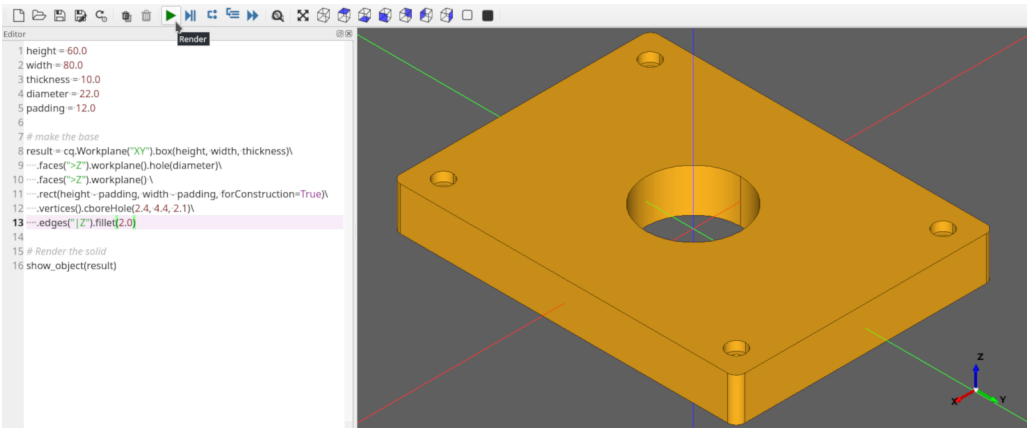
We can do that using the preset dictionaries in the parameter definition:

```
1  height = 60.0
2  width = 80.0
3  thickness = 10.0
4  diameter = 22.0
5  padding = 12.0
6
7  # make the base
8  result = (
9      cq.Workplane("XY")
10     .box(height, width, thickness)
11     .faces(">Z")
12     .workplane()
13     .hole(diameter)
14     .faces(">Z")
15     .workplane()
16     .rect(height - padding, width - padding, forConstruction=True)
17     .vertices()
18     .cboreHole(2.4, 4.4, 2.1)
19     .edges("|Z")
20     .fillet(2.0)
21 )
22
23 # Render the solid
24 show_object(result)
```

Line 20 fillets the edges using the *cadquery.Workplane.fillet()* method.

To grab the right edges, the *cadquery.Workplane.edges()* selects all of the edges that are parallel to the Z axis ("|Z"),

The finished product looks like this:



3.3.7 Exporting

If you want to fabricate a physical object you need to export the result to STL or DXF. Additionally, exporting as STEP for post-processing in another CAD tool is also possible.

This can be easily accomplished using the `cadquery.exporters.export()` function:

```
height = 60.0
width = 80.0
thickness = 10.0
diameter = 22.0
padding = 12.0

# make the base
result = (
    cq.Workplane("XY")
    .box(height, width, thickness)
    .faces(">Z")
    .workplane()
    .hole(diameter)
    .faces(">Z")
    .workplane()
    .rect(height - padding, width - padding, forConstruction=True)
    .vertices()
    .cboreHole(2.4, 4.4, 2.1)
    .edges("|Z")
    .fillet(2.0)
)

# Render the solid
show_object(result)

# Export
cq.exporters.export(result, "result.stl")
cq.exporters.export(result.section(), "result.dxf")
cq.exporters.export(result, "result.step")
```

3.3.8 Done!

You just made a parametric, model that can generate pretty much any bearing pillow block with <30 lines of code.

3.3.9 Want to learn more?

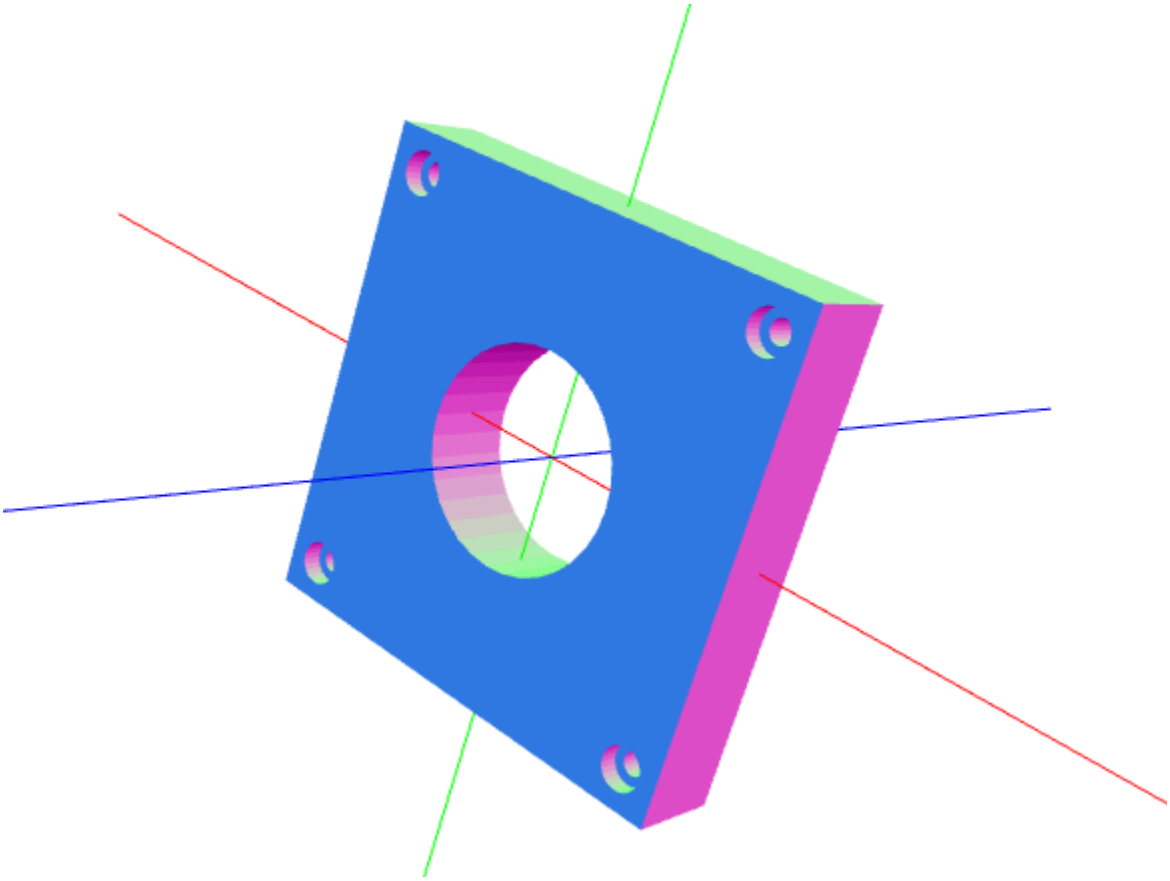
- The *Examples* contains lots of examples demonstrating cadquery features
- The *API Reference* is a good overview of language features grouped by function
- The *CadQuery Class Summary* is the hard-core listing of all functions available.

3.4 Design Principles

3.4.1 Principle 1: Intuitive Construction

CadQuery aims to make building models using python scripting easy and intuitive. CadQuery strives to allow scripts to read roughly as a human would describe an object verbally.

For example, consider this object:



A human would describe this as:

> "A block 80mm square x 30mm thick , with countersunk holes for M2 socket head cap screws at the corners, and a circular pocket 22mm in diameter in the middle for a bearing"

The goal is to have the CadQuery script that produces this object be as close as possible to the English phrase a human would use.

3.4.2 Principle 2: Capture Design Intent

The features that are **not** part of the part description above are just as important as those that are. For example, most humans will assume that:

- The countersunk holes are spaced a uniform distance from the edges
- The circular pocket is in the center of the block, no matter how big the block is

If you have experience with 3D CAD systems, you also know that there is a key design intent built into this object. After the base block is created, how the hole is located is key. If it is located from one edge, changing the block size will have a different effect than if the hole is located from the center.

Many scripting languages do not provide a way to capture design intent– because they require that you always work in global coordinates. CadQuery is different– you can locate features relative to others in a relative way– preserving the design intent just like a human would when creating a drawing or building an object.

In fact, though many people know how to use 3D CAD systems, few understand how important the way that an object is built impact its maintainability and resiliency to design changes.

3.4.3 Principle 3: Plugins as first class citizens

Any system for building 3D models will evolve to contain an immense number of libraries and feature builders. It is important that these can be seamlessly included into the core and used alongside the built in libraries. Plugins should be easy to install and familiar to use.

3.4.4 Principle 4: CAD models as source code makes sense

It is surprising that the world of 3D CAD is primarily dominated by systems that create opaque binary files. Just like the world of software, CAD models are very complex.

CAD models have many things in common with software, and would benefit greatly from the use of tools that are standard in the software industry, such as:

1. Easily re-using features between objects
2. Storing objects using version control systems
3. Computing the differences between objects by using source control tools
4. Share objects on the Internet
5. Automate testing and generation by allowing objects to be built from within libraries

CadQuery is designed to make 3D content creation easy enough that the above benefits can be attained without more work than using existing ‘opaque’, ‘point and click’ solutions.

3.5 Concepts

3.5.1 3D BREP Topology Concepts

Before talking about CadQuery, it makes sense to talk a little about 3D CAD topology. CadQuery is based upon the OpenCascade kernel, which uses Boundary Representations (BREP) for objects. This just means that objects are defined by their enclosing surfaces.

When working in a BREP system, these fundamental constructs exist to define a shape (working up the food chain):

vertex
a single point in space

edge
a connection between two or more vertices along a particular path (called a curve)

wire
a collection of edges that are connected together.

face
a set of edges or wires that enclose a surface

shell
a collection of faces that are connected together along some of their edges

solid
a shell that has a closed interior

compound
a collection of solids

When using CadQuery, all of these objects are created, hopefully with the least possible work. In the actual CAD kernel, there is another set of Geometrical constructs involved as well. For example, an arc-shaped edge will hold a reference to an underlying curve that is a full circle, and each linear edge holds underneath it the equation for a line. CadQuery shields you from these constructs.

3.5.2 Workplane class

The Workplane class contains the currently selected objects (a list of Shapes, Vectors or Locations in the `objects` attribute), the modelling context (in the `ctx` attribute), and CadQuery's fluent api methods. It is the main class that users will instantiate.

See *API Reference* to learn more.

3.5.3 Workplanes

Most CAD programs use the concept of Workplanes. If you have experience with other CAD programs you will probably feel comfortable with CadQuery's Workplanes, but if you don't have experience then they are an essential concept to understand.

Workplanes represent a plane in space, from which other features can be located. They have a center point and a local coordinate system. Most methods that create an object do so relative to the current workplane.

Usually the first workplane created is the "XY" plane, also known as the "front" plane. Once a solid is defined the most common way to create a workplane is to select a face on the solid that you intend to modify and create a new workplane relative to it. You can also create new workplanes in anywhere in world coordinate system, or relative to other planes using offsets or rotations.

The most powerful feature of workplanes is that they allow you to work in 2D space in the coordinate system of the workplane, and then CadQuery will transform these points from the workplane coordinate system to the world coordinate system so your 3D features are located where you intended. This makes scripts much easier to create and maintain.

See *cadquery.Workplane* to learn more.

3.5.4 2D Construction

Once you create a workplane, you can work in 2D, and then later use the features you create to make 3D objects. You'll find all of the 2D constructs you expect – circles, lines, arcs, mirroring, points, etc.

See *2D Operations* to learn more.

3.5.5 3D Construction

You can construct 3D primitives such as boxes, wedges, cylinders and spheres directly. You can also sweep, extrude, and loft 2D geometry to form 3D features. Of course the basic primitive operations are also available.

See *3D Operations* to learn more.

3.5.6 Selectors

Selectors allow you to select one or more features, in order to define new features. As an example, you might extrude a box, and then select the top face as the location for a new feature. Or, you might extrude a box, and then select all of the vertical edges so that you can apply a fillet to them.

You can select Vertices, Edges, Faces, Solids, and Wires using selectors.

Think of selectors as the equivalent of your hand and mouse, if you were to build an object using a conventional CAD system.

See *Selectors* to learn more.

3.5.7 Construction Geometry

Construction geometry are features that are not part of the object, but are only defined to aid in building the object. A common example might be to define a rectangle, and then use the corners to define the location of a set of holes.

Most CadQuery construction methods provide a `forConstruction` keyword, which creates a feature that will only be used to locate other features.

3.5.8 The Stack

As you work in CadQuery, each operation returns a new Workplane object with the result of that operations. Each Workplane object has a list of objects, and a reference to its parent.

You can always go backwards to older operations by removing the current object from the stack. For example:

```
Workplane(someObject).faces(">Z").first().vertices()
```

returns a CadQuery object that contains all of the vertices on the highest face of someObject. But you can always move backwards in the stack to get the face as well:

```
Workplane(someObject).faces(">Z").first().vertices().end()
```

You can browse stack access methods here: *Stack and Selector Methods*.

3.5.9 Chaining

All Workplane methods return another Workplane object, so that you can chain the methods together fluently. Use the core Workplane methods to get at the objects that were created.

Each time a new Workplane object is produced during these chained calls, it has a `parent` attribute that points to the Workplane object that created it. Several CadQuery methods search this parent chain, for example when searching for the context solid. You can also give a Workplane object a tag, and further down your chain of calls you can refer back to this particular object using its tag.

3.5.10 The Context Solid

Most of the time, you are building a single object, and adding features to that single object. CadQuery watches your operations, and defines the first solid object created as the 'context solid'. After that, any features you create are automatically combined (unless you specify otherwise) with that solid. This happens even if the solid was created a long way up in the stack. For example:

```
Workplane("XY").box(1, 2, 3).faces(">Z").circle(0.25).extrude(1)
```

Will create a 1x2x3 box, with a cylindrical boss extending from the top face. It was not necessary to manually combine the cylinder created by extruding the circle with the box, because the default behavior for extrude is to combine the result with the context solid. The hole() method works similarly – CadQuery presumes that you want to subtract the hole from the context solid.

If you want to avoid this, you can specify `combine=False`, and CadQuery will create the solid separately.

3.5.11 Iteration

CAD models often have repeated geometry, and its really annoying to resort to for loops to construct features. Many CadQuery methods operate automatically on each element on the stack, so that you don't have to write loops. For example, this:

```
Workplane("XY").box(1, 2, 3).faces(">Z").vertices().circle(0.5)
```

Will actually create 4 circles, because `vertices()` selects 4 vertices of a rectangular face, and the `circle()` method iterates on each member of the stack.

This is really useful to remember when you author your own plugins. *cadquery.Workplane.each()* is useful for this purpose.

3.5.12 CadQuery API layers

Once you start to dive a bit more into CadQuery, you may find yourself a bit confused juggling between different types of objects the CadQuery APIs can return. This chapter aims to give an explanation on this topic and to provide background on the underlying implementation and kernel layers so you can leverage more of CadQuery functionality.

CadQuery is composed of 3 different API, which are implemented on top of each other.

1. The Fluent API
2. The Direct API
3. The OCCT API

The Fluent API

What we call the fluent API is what you work with when you first start using CadQuery, the *Workplane* class and all its methods defines the Fluent API. This is the API you will use and see most of the time, it's fairly easy to use and it simplifies a lot of things for you. A classic example could be :

```
part = Workplane("XY").box(1, 2, 3).faces(">Z").vertices().circle(0.5).cutThruAll()
```

Here we create a *Workplane* object on which we subsequently call several methods to create our part. A general way of thinking about the Fluent API is to consider the *Workplane* as your part object and all it's methods as operations that will affect your part. Often you will start with an empty *Workplane*, then add more features by calling *Workplane* methods.

This hierarchical structure of operations modifying a part is well seen with the traditional code style used in CadQuery code. Code written with the CadQuery fluent API will often look like this :

```
part = Workplane("XY").box(1, 2, 3).faces(">Z").vertices().circle(0.5).cutThruAll()
```

Or like this :

```
part = Workplane("XY")
part = part.box(1, 2, 3)
part = part.faces(">Z")
part = part.vertices()
part = part.circle(0.5)
part = part.cutThruAll()
```

Note: While the first code style is what people default to, it's important to note that when you write your code like this it's equivalent as writting it on a single line. It's then more difficult to debug as you cannot visualize each operation step by step, which is a functionality that is provided by the CQ-Editor debugger for example.

The Direct API

While the fluent API exposes much functionality, you may find scenarios that require extra flexibility or require working with lower level objects.

The direct API is the API that is called by the fluent API under the hood. The 9 topological classes and their methods compose the direct API. These classes actually wrap the equivalent Open CASCADE Technology (OCCT) classes. The 9 topological classes are :

1. *Shape*
2. *Compound*
3. CompSolid
4. *Solid*
5. *Shell*
6. *Face*
7. *Wire*
8. *Edge*
9. *Vertex*

Each class has its own methods to create and/or edit shapes of their respective type. As already explained in *Concepts* there is also some kind of hierarchy in the topological classes. A Wire is made of several edges which are themselves made of several vertices. This means you can create geometry from the bottom up and have a lot of control over it.

For example we can create a circular face like so

```
circle_wire = Wire.makeCircle(10, Vector(0, 0, 0), Vector(0, 0, 1))
circular_face = Face.makeFromWires(circle_wire, [])
```

Note: In CadQuery (and OCCT) all the topological classes are shapes, the *Shape* class is the most abstract topological class. The topological class inherits `Mixin3D` or `Mixin1D` which provide aditional methods that are shared between the classes that inherits them.

The direct API as its name suggests doesn't provide a parent/children data structure, instead each method call directly returns an object of the specified topological type. It is more verbose than the fluent API and more tedious to work with, but as it offers more flexibility (you can work with faces, which is something you can't do in the fluent API) it is sometimes more convenient than the fluent API.

The OCCT API

Finally we are discussing about the OCCT API. The OCCT API is the lowest level of CadQuery. The direct API is built upon the OCCT API, where the OCCT API in CadQuery is available through OCP. OCP are the Python bindings of the OCCT C++ libraries CadQuery uses. This means you have access to (almost) all the OCCT C++ libraries in Python and in CadQuery. Working with the OCCT API will give you the maximum flexibility and control over you designs, it is however very verbose and difficult to use. You will need to have a strong knowledge of the different C++ libraries to be able to achieve what you want. To obtain this knowledge the most obvious ways are :

1. Read the direct API source code, since it is build upon the OCCT API it is full of example usage.
2. Go through the C++ documentation

Note: The general way of importing a specific class of the OCCT API is

```
from OCP.thePackageName import theClassName
```

For example if you want to use the class BRepPrimAPI_MakeBox. You will go by the following

```
from OCP.BRepPrimAPI import BRepPrimAPI_MakeBox
```

The package name of any class is written at the top of the documentation page. Often it's written in the class name itself as a prefix.

Going back and forth between the APIs

While the 3 APIs provide 3 different layer of complexity and functionality you can mix the 3 layers as you wish. Below is presented the different ways you can interact with the different API layers.

Fluent API <=> Direct API

Here are all the possibilities you have to get an object from the Direct API (i.e a topological object).

You can end the Fluent API call chain and get the last object on the stack with *Workplane.val()* alternatively you can get all the objects with *Workplane.vals()*

```
>>> box = Workplane().box(10, 5, 5)
>>> print(type(box))
<class cadquery.cq.Workplane>

>>> box = Workplane().box(10, 5, 5).val()
>>> print(type(box))
<class cadquery.occ_impl.shapes.Solid>
```

If you are only interested in getting the context solid of your Workplane, you can use *Workplane.findSolid()*:

```
>>> part = Workplane().box(10,5,5).circle(3).val()
>>> print(type(part))
<class cadquery.cq.Wire>

>>> part = Workplane().box(10,5,5).circle(3).findSolid()
>>> print(type(part))
<class cadquery.occ_impl.shapes.Compound>
# The return type of findSolid is either a Solid or a Compound object
```

If you want to go the other way around i.e using objects from the topological API in the Fluent API here are your options :

You can pass a topological object as a base object to the *Workplane* object.

```
solid_box = Solid.makeBox(10, 10, 10)
part = Workplane(obj=solid_box)
# And you can continue your modelling in the fluent API
part = part.faces(">Z").circle(1).extrude(10)
```

You can add a topological object as a new operation/step in the Fluent API call chain with *Workplane.newObject()*

```
circle_wire = Wire.makeCircle(1, Vector(0, 0, 0), Vector(0, 0, 1))
box = Workplane().box(10, 10, 10).newObject([circle_wire])
# And you can continue modelling
box = (
    box.toPending().cutThruAll()
)  # notice the call to `toPending` that is needed if you want to use it in a subsequent␣
↪operation
```

Direct API <=> OCCT API

Every object of the Direct API stores its OCCT equivalent object in its `wrapped` attribute.:

```
>>> box = Solid.makeBox(10,5,5)
>>> print(type(box))
<class cadquery.occ_impl.shapes.Solid>

>>> box = Solid.makeBox(10,5,5).wrapped
>>> print(type(box))
<class OCP.TopoDS.TopoDS_Solid>
```

If you want to cast an OCCT object into a Direct API one you can just pass it as a parameter of the intended class:

```
>>> occt_box = BRepPrimAPI_MakeBox(5,5,5).Solid()
>>> print(type(occt_box))
<class OCP.TopoDS.TopoDS_Solid>

>>> direct_api_box = Solid(occt_box)
>>> print(type(direct_api_box))
<class cadquery.occ_impl.shapes.Solid>
```

Note: You can cast into the direct API the types found here

3.5.13 Multimethods

CadQuery uses Multimethod to allow a call to a method to be dispatched depending on the types of the arguments. An example is *arc()*, where `a_sketch.arc((1, 2), (2, 3))` would be dispatched to one method but `a_sketch.arc((1, 2), (2, 3), (3, 4))` would be dispatched to a different method. For multimethods to work, you should not use keyword arguments to specify positional parameters. For example, you **should not** write `a_sketch.arc(p1=(1, 2), p2=(2, 3), p3=(3, 4))`, instead you should use the previous example. Note CadQuery makes an attempt to fall back on the first registered multimethod in the event of a dispatch error, but it is still best practice to not use keyword arguments to specify positional arguments in CadQuery.

3.5.14 An Introspective Example

Note: If you are just beginning with CadQuery then you can leave this example for later. If you have some experience with creating CadQuery models and now you want to read the CadQuery source to better understand what your code does, then it is recommended you read this example first.

To demonstrate the above concepts, we can define more detailed string representations for the *Workplane*, *Plane* and `CQContext` classes and patch them in:

```
import cadquery as cq


def tidy_repr(obj):
    """Shortens a default repr string"""
```

(continues on next page)

(continued from previous page)

```
    return repr(obj).split(".")[-1].rstrip(">")


def _ctx_str(self):
    return (
        tidy_repr(self)
        + ":\n"
        + f"    pendingWires: {self.pendingWires}\n"
        + f"    pendingEdges: {self.pendingEdges}\n"
        + f"    tags: {self.tags}"
    )


cq.cq.CQContext.__str__ = _ctx_str


def _plane_str(self):
    return (
        tidy_repr(self)
        + ":\n"
        + f"    origin: {self.origin.toTuple()}\n"
        + f"    z direction: {self.zDir.toTuple()}"
    )


cq.occ_impl.geom.Plane.__str__ = _plane_str


def _wp_str(self):
    out = tidy_repr(self) + ":\n"
    out += f"  parent: {tidy_repr(self.parent)}\n" if self.parent else "  no parent\n"
    out += f"  plane: {self.plane}\n"
    out += f"  objects: {self.objects}\n"
    out += f"  modelling context: {self.ctx}"
    return out


cq.Workplane.__str__ = _wp_str
```

Now we can make a simple part and examine the *Workplane* and `CQContext` objects at each step. The final part looks like:

```
part = (
    cq.Workplane()
    .box(1, 1, 1)
    .tag("base")
    .wires(">Z")
    .toPending()
    .translate((0.1, 0.1, 1.0))
    .toPending()
    .loft()
    .faces(">>X", tag="base")
```

(continues on next page)

(continued from previous page)

```
    .workplane(centerOption="CenterOfMass")
    .circle(0.2)
    .extrude(1)
)
```

Note: Some of the modelling process for this part is a bit contrived and not a great example of fluent CadQuery techniques.

The start of our chain of calls is:

```
part = cq.Workplane()
print(part)
```

Which produces the output:

```
Workplane object at 0x2760:
  no parent
  plane: Plane object at 0x2850:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: []
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {}
```

This is simply an empty *Workplane*. Being the first *Workplane* in the chain, it does not have a parent. The `plane` attribute contains a *Plane* object that describes the XY plane.

Now we create a simple box. To keep things short, the `print(part)` line will not be shown for the rest of these code blocks:

```
part = part.box(1, 1, 1)
```

Which produces the output:

```
Workplane object at 0xaa90:
  parent: Workplane object at 0x2760
  plane: Plane object at 0x3850:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Solid object at 0xbbe0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {}
```

The first thing to note is that this is a different *Workplane* object to the previous one, and in the `parent` attribute of this *Workplane* is our previous *Workplane*. Returning a new instance of *Workplane* is the normal behaviour of most *Workplane* methods (with some exceptions, as will be shown below) and this is how the *chaining* concept is implemented.

Secondly, the modelling context object is the same as the one in the previous *Workplane*, and this one modelling

context at 0x2730 will be shared between every *Workplane* object in this chain. If we instantiate a new *Workplane* with `part2 = cq.Workplane()`, then this `part2` would have a different instance of the `CQContext` attached to it.

Thirdly, in our objects list is a single *Solid* object, which is the box we just created.

Often when creating models you will find yourself wanting to refer back to a specific *Workplane* object, perhaps because it is easier to select the feature you want in this earlier state, or because you want to reuse a plane. Tags offer a way to refer back to a previous *Workplane*. We can tag the *Workplane* that contains this basic box now:

```
part = part.tag("base")
```

The string representation of `part` is now:

```
Workplane object at 0xaa90:
  parent: Workplane object at 0x2760
  plane: Plane object at 0x3850:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Solid object at 0xbbe0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

The `tags` attribute of the modelling context is simply a dict associating the string name given by the *tag()* method to the *Workplane*. Methods such as *workplaneFromTagged()* and selection methods like *edges()* can operate on a tagged *Workplane*. Note that unlike the `part = part.box(1, 1, 1)` step where we went from `Workplane object at 0x2760` to `Workplane object at 0xaa90`, the *tag()* method has returned the same object at `0xaa90`. This is unusual for a *Workplane* method.

The next step is:

```
part = part.faces(">>Z")
```

The output is:

```
Workplane object at 0x8c40:
  parent: Workplane object at 0xaa90
  plane: Plane object at 0xac40:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Face object at 0x3c10>]
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

Our selection method has taken the *Solid* from the `objects` list of the previous *Workplane*, found the face with it's center furthest in the Z direction, and placed that face into the `objects` attribute. The *Solid* representing the box we are modelling is gone, and when a *Workplane* method needs to access that solid it searches through the parent chain for the nearest solid. This action can also be done by a user through the *findSolid()* method.

Now we want to select the boundary of this *Face* (a *Wire*), so we use:

```
part = part.wires()
```

The output is now:

```
Workplane object at 0x6880:
  parent: Workplane object at 0x8c40
  plane: Plane object at 0x38b0:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Wire object at 0xaca0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

Modelling operations take their wires and edges from the modelling context's pending lists. In order to use the *loft()* command further down the chain, we need to push this wire to the modelling context with:

```
part = part.toPending()
```

Now we have:

```
Workplane object at 0x6880:
  parent: Workplane object at 0x8c40
  plane: Plane object at 0x38b0:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Wire object at 0xaca0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: [<cadquery.occ_impl.shapes.Wire object at 0xaca0>]
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

The *Wire* object that was only in the `objects` attribute before is now also in the modelling context's `pendingWires`. The *toPending()* method is also another of the unusual methods that return the same *Workplane* object instead of a new one.

To set up the other side of the *loft()* command further down the chain, we translate the wire in `objects` by calling:

```
part = part.translate((0.1, 0.1, 1.0))
```

Now the string representation of `part` looks like:

```
Workplane object at 0x3a00:
  parent: Workplane object at 0x6880
  plane: Plane object at 0xac70:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Wire object at 0x35e0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: [<cadquery.occ_impl.shapes.Wire object at 0xaca0>]
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

It may look similar to the previous step, but the *Wire* object in `objects` is different. To get this wire into the pending wires list, again we use:

```
part = part.toPending()
```

The result:

```
Workplane object at 0x3a00:
  parent: Workplane object at 0x6880
  plane: Plane object at 0xac70:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Wire object at 0x35e0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: [<cadquery.occ_impl.shapes.Wire object at 0xaca0>, <cadquery.occ_impl.
↪shapes.Wire object at 0x7f5c7f5c35e0>]
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

The modelling context's `pendingWires` attribute now contains the two wires we want to loft between, and we simply call:

```
part = part.loft()
```

After the loft operation, our Workplane looks quite different:

```
Workplane object at 0x32b0:
  parent: Workplane object at 0x3a00
  plane: Plane object at 0x3d60:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Compound object at 0xad30>]
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

In the `cq.Workplane.objects` attribute we now have one Compound object and the modelling context's `pendingWires` has been cleared by *loft()*.

Note: To inspect the *Compound* object further you can use *val()* or *findSolid()* to get at the *Compound* object, then use *cadquery.Shape.Solids()* to return a list of the *Solid* objects contained in the *Compound*, which in this example will be a single *Solid* object. For example:

```
>>> a_compound = part.findSolid()
>>> a_list_of_solids = a_compound.Solids()
>>> len(a_list_of_solids)
1
```

Now we will create a small cylinder protruding from a face on the original box. We need to set up a workplane to draw a circle on, so firstly we will select the correct face:

```
part = part.faces(">>X", tag="base")
```

Which results in:

```
Workplane object at 0x3f10:
  parent: Workplane object at 0x32b0
```

(continues on next page)

(continued from previous page)

```
  plane: Plane object at 0xefa0:
    origin: (0.0, 0.0, 0.0)
    z direction: (0.0, 0.0, 1.0)
  objects: [<cadquery.occ_impl.shapes.Face object at 0x3af0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

We have the desired *Face* in the `objects` attribute, but the `plane` has not changed yet. To create the new plane we use the *Workplane.workplane()* method:

```
part = part.workplane()
```

Now:

```
Workplane object at 0xe700:
  parent: Workplane object at 0x3f10
  plane: Plane object at 0xe730:
    origin: (0.5, 0.0, 0.0)
    z direction: (1.0, 0.0, 0.0)
  objects: []
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

The `objects` list has been cleared and the *Plane* object has a local Z direction in the global X direction. Since the base of the plane is the side of the box, the origin is offset in the X direction.

Onto this plane we can draw a circle:

```
part = part.circle(0.2)
```

Now:

```
Workplane object at 0xe790:
  parent: Workplane object at 0xe700
  plane: Plane object at 0xaf40:
    origin: (0.5, 0.0, 0.0)
    z direction: (1.0, 0.0, 0.0)
  objects: [<cadquery.occ_impl.shapes.Wire object at 0xe610>]
  modelling context: CQContext object at 0x2730:
    pendingWires: [<cadquery.occ_impl.shapes.Wire object at 0xe610>]
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

The *circle()* method - like all 2D drawing methods - has placed the circle into both the `objects` attribute (where it will be cleared during the next modelling step), and the modelling context's pending wires (where it will persist until used by another *Workplane* method).

The next step is to extrude this circle and create a cylindrical protrusion:

```
part = part.extrude(1, clean=False)
```

Now:

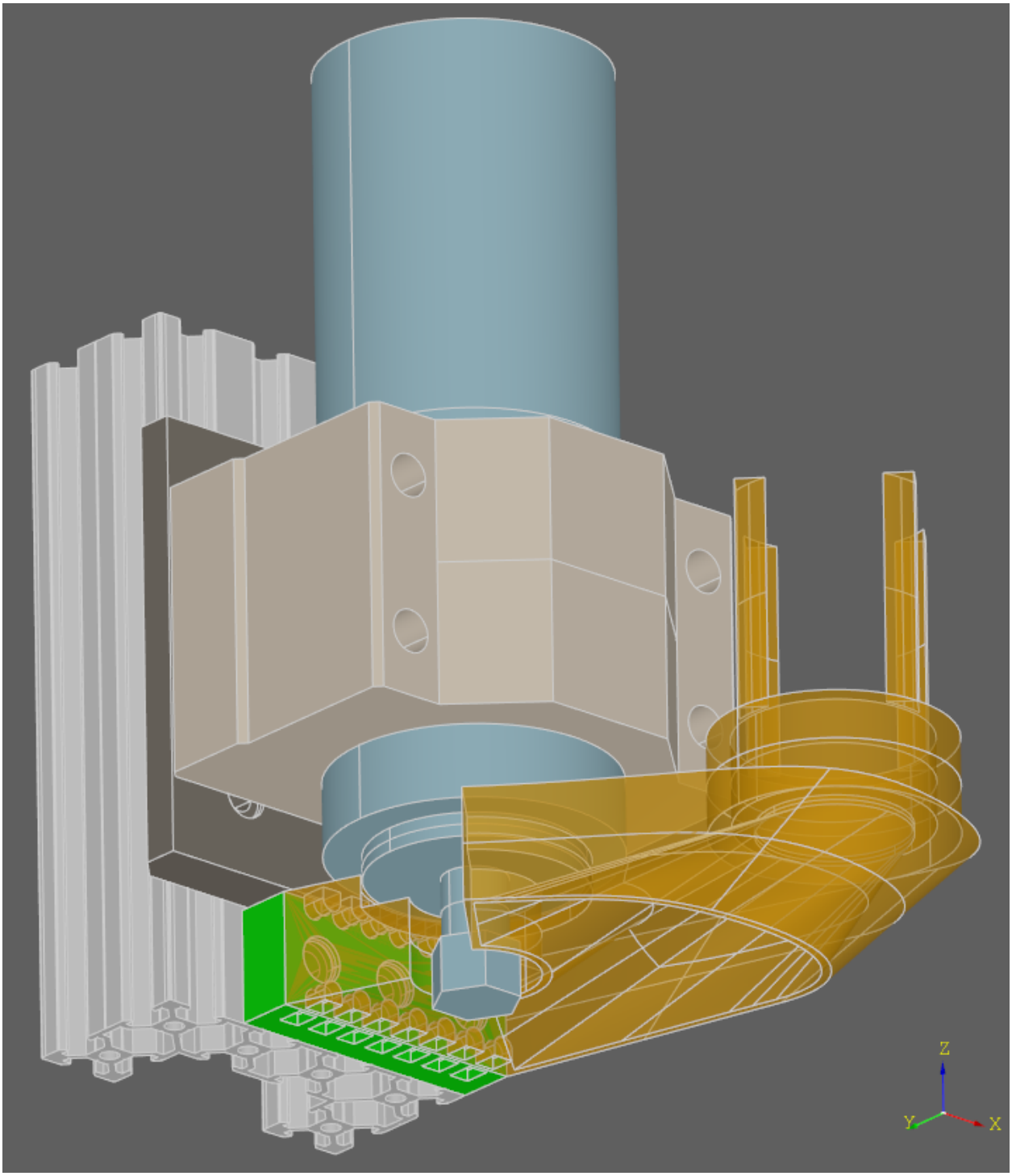
```
Workplane object at 0xafd0:
  parent: Workplane object at 0xe790
  plane: Plane object at 0x3e80:
    origin: (0.5, 0.0, 0.0)
    z direction: (1.0, 0.0, 0.0)
  objects: [<cadquery.occ_impl.shapes.Compound object at 0xaaf0>]
  modelling context: CQContext object at 0x2730:
    pendingWires: []
    pendingEdges: []
    tags: {'base': <cadquery.cq.Workplane object at 0xaa90>}
```

The *extrude()* method has cleared all the pending wires and edges. The `objects` attribute contains the final *Compound* object that is shown in the 3D view above.

Note: The *extrude()* has an argument for `clean` which defaults to `True`. This extrudes the pending wires (creating a new *Workplane* object), then runs the *clean()* method to refine the result, creating another *Workplane*. If you were to run the example with the default `clean=True` then you would see an intermediate *Workplane* object in `parent` rather than the object from the previous step.

3.5.15 Assemblies

Simple models can be combined into complex, possibly nested, assemblies.



A simple example could look as follows:

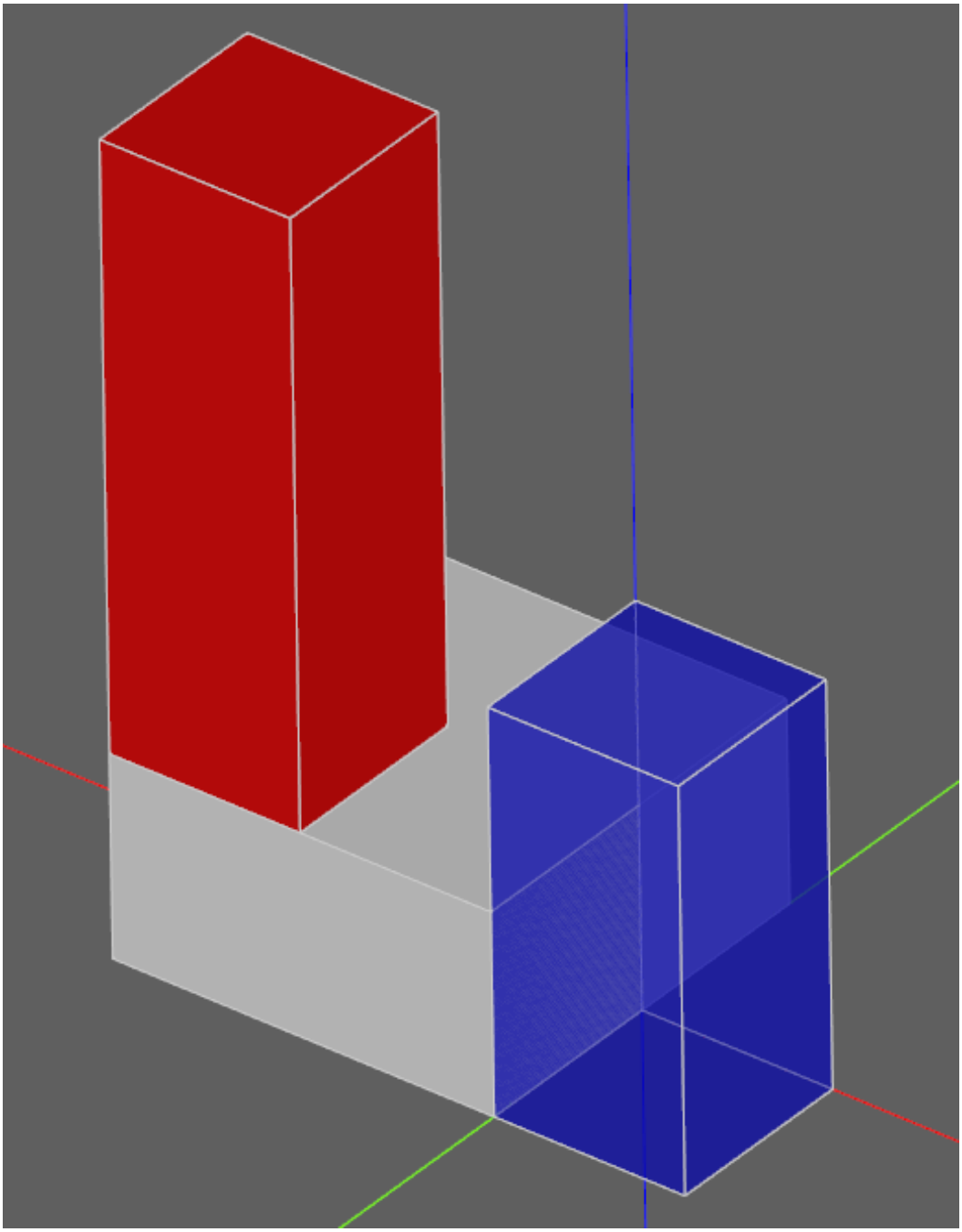
```
from cadquery import *

w = 10
d = 10
h = 10
```

(continues on next page)

(continued from previous page)

```
part1 = Workplane().box(2 * w, 2 * d, h)
part2 = Workplane().box(w, d, 2 * h)
part3 = Workplane().box(w, d, 3 * h)

assy = (
    Assembly(part1, loc=Location(Vector(-w, 0, h / 2)))
    .add(
        part2, loc=Location(Vector(1.5 * w, -0.5 * d, h / 2)), color=Color(0, 0, 1, 0.5)
    )
    .add(part3, loc=Location(Vector(-0.5 * w, -0.5 * d, 2 * h)), color=Color("red"))
)
```

Resulting in:



Note that the locations of the children parts are defined with respect to their parents - in the above example `part3` will be located at (-5,-5,20) in the global coordinate system. Assemblies with different colors can be created this way and exported to STEP or the native OCCT xml format.

You can browse assembly related methods here: *Assemblies*.

3.5.16 Assemblies with constraints

Sometimes it is not desirable to define the component positions explicitly but rather use constraints to obtain a fully parametric assembly. This can be achieved in the following way:

```
from cadquery import *

w = 10
d = 10
h = 10

part1 = Workplane().box(2 * w, 2 * d, h)
part2 = Workplane().box(w, d, 2 * h)
part3 = Workplane().box(w, d, 3 * h)

assy = (
    Assembly(part1, name="part1", loc=Location(Vector(-w, 0, h / 2)))
    .add(part2, name="part2", color=Color(0, 0, 1, 0.5))
    .add(part3, name="part3", color=Color("red"))
    .constrain("part1@faces@>Z", "part3@faces@<Z", "Axis")
    .constrain("part1@faces@>Z", "part2@faces@<Z", "Axis")
    .constrain("part1@faces@>Y", "part3@faces@<Y", "Axis")
    .constrain("part1@faces@>Y", "part2@faces@<Y", "Axis")
    .constrain("part1@vertices@>(-1,-1,1)", "part3@vertices@>(-1,-1,-1)", "Point")
    .constrain("part1@vertices@>(1,-1,-1)", "part2@vertices@>(-1,-1,-1)", "Point")
    .solve()
)
```

This code results in identical object as one from the previous section. The added benefit is that with changing parameters `w`, `d`, `h` the final locations will be calculated automatically. It is admittedly dense and can be made clearer using tags. Tags can be directly referenced when constructing the constraints:

```
from cadquery import *

w = 10
d = 10
h = 10

part1 = Workplane().box(2 * w, 2 * d, h)
part2 = Workplane().box(w, d, 2 * h)
part3 = Workplane().box(w, d, 3 * h)

part1.faces(">Z").edges("<X").vertices("<Y").tag("pt1")
part1.faces(">X").edges("<Z").vertices("<Y").tag("pt2")
part3.faces("<Z").edges("<X").vertices("<Y").tag("pt1")
part2.faces("<X").edges("<Z").vertices("<Y").tag("pt2")

assy1 = (
    Assembly(part1, name="part1", loc=Location(Vector(-w, 0, h / 2)))
    .add(part2, name="part2", color=Color(0, 0, 1, 0.5))
    .add(part3, name="part3", color=Color("red"))
    .constrain("part1@faces@>Z", "part3@faces@<Z", "Axis")
    .constrain("part1@faces@>Z", "part2@faces@<Z", "Axis")
    .constrain("part1@faces@>Y", "part3@faces@<Y", "Axis")
```

(continues on next page)

(continued from previous page)

```
    .constrain("part1@faces@>Y", "part2@faces@<Y", "Axis")
    .constrain("part1?pt1", "part3?pt1", "Point")
    .constrain("part1?pt2", "part2?pt2", "Point")
    .solve()
)
```

The following constraints are currently implemented:

Axis
: two normal vectors are anti-coincident or the angle (in radians) between them is equal to the specified value. Can be defined for all entities with consistent normal vector - planar faces, wires and edges.

Point
: two points are coincident or separated by a specified distance. Can be defined for all entities, center of mass is used for lines, faces, solids and the vertex position for vertices.

Plane
: combination of :Axis: and :Point: constraints.

For a more elaborate assembly example see *Assemblies*.

3.6 Sketch

3.6.1 Sketch tutorial

The purpose of this section is to demonstrate how to construct sketches using different approaches.

Face-based API

The main approach for constructing sketches is based on constructing faces and combining them using boolean operations.

```
import cadquery as cq

result = (
    cq.Sketch()
    .trapezoid(4, 3, 90)
    .vertices()
    .circle(0.5, mode="s")
    .reset()
    .vertices()
    .fillet(0.25)
    .reset()
    .rarray(0.6, 1, 5, 1)
    .slot(1.5, 0.4, mode="s", angle=90)
)
```

Note that selectors are implemented, but selection has to be explicitly reset. Sketch class does not implement history and all modifications happen in-place.

Modes

Every operation from the face API accepts a mode parameter to define how to combine the created object with existing ones. It can be fused (`mode='a'`), cut (`mode='s'`), intersected (`mode='i'`) or just stored for construction (`mode='c'`). In the last case, it is mandatory to specify a `tag` in order to be able to refer to the object later on. By default faces are fused together. Note the usage of the subtractive and additive modes in the example above. The additional two are demonstrated below.

```
result = (
    cq.Sketch()
    .rect(1, 2, mode="c", tag="base")
    .vertices(tag="base")
    .circle(0.7)
    .reset()
    .edges("|Y", tag="base")
    .ellipse(1.2, 1, mode="i")
    .reset()
    .rect(2, 2, mode="i")
    .clean()
)
```

Edge-based API

If needed, one can construct sketches by placing individual edges.

```
import cadquery as cq

result = (
    cq.Sketch()
    .segment((0.0, 0), (0.0, 2.0))
    .segment((2.0, 0))
    .close()
    .arc((0.6, 0.6), 0.4, 0.0, 360.0)
    .assemble(tag="face")
    .edges("%LINE", tag="face")
    .vertices()
    .chamfer(0.2)
)
```

Once the construction is finished it has to be converted to the face-based representation using *assemble()*. Afterwards, face based operations can be applied.

Convex hull

Warning: The Convex Hull feature is currently experimental.

For certain special use-cases convex hull can be constructed from straight segments and circles.

```
result = (
    cq.Sketch()
```

(continues on next page)

(continued from previous page)

```
    .arc((0, 0), 1.0, 0.0, 360.0)
    .arc((1, 1.5), 0.5, 0.0, 360.0)
    .segment((0.0, 2), (-1, 3.0))
    .hull()
)
```

Constraint-based sketches

Warning: The 2D Sketch constraints and solver is currently experimental.

Finally, if desired, geometric constraints can be used to construct sketches. So far only line segments and arcs can be used in such a use case.

```
import cadquery as cq

result = (
    cq.Sketch()
    .segment((0, 0), (0, 3.0), "s1")
    .arc((0.0, 3.0), (1.5, 1.5), (0.0, 0.0), "a1")
    .constrain("s1", "Fixed", None)
    .constrain("s1", "a1", "Coincident", None)
    .constrain("a1", "s1", "Coincident", None)
    .constrain("s1", "a1", "Angle", 45)
    .solve()
    .assemble()
)
```

Following constraints are implemented. Arguments are passed in as one tuple in *constrain()*. In this table, *0..1* refers to a float between 0 and 1 where 0 would create a constraint relative to the start of the element, and 1 the end.

Name	Arity	Entities	Arguments	Description
FixedPoint	1	All	*None* for arc center or *0..1* for point on segment/arc	Specified point is fixed
Coincident	2	All	None	Specified points coincide
Angle	2	All	*angle*	Angle between the tangents of the two entities is fixed
Length	1	All	*length*	Specified entity has fixed length
Distance	2	All	*None or 0..1, None or 0..1, distance*	Distance between two points is fixed
Radius	1	Arc	*radius*	Specified entity has a fixed radius
Orientation	1	Segment	*x,y*	Specified entity is parallel to *(x,y)*
ArcAngle	1	Arc	*angle*	Specified entity is fixed angular span

3.6.2 Workplane integration

Once created, a sketch can be used to construct various features on a workplane. Supported operations include *extrude()*, *twistExtrude()*, *revolve()*, *sweep()*, *cutBlind()*, *cutThruAll()* and *loft()*.

Sketches can be created as separate entities and reused, but also created ad-hoc in one fluent chain of calls as shown below.

Note that the sketch is placed on all locations that are on the top of the stack.

Constructing sketches in-place can be accomplished as follows.

```
import cadquery as cq

result = (
    cq.Workplane()
    .box(5, 5, 1)
    .faces(">Z")
    .sketch()
    .regularPolygon(2, 3, tag="outer")
    .regularPolygon(1.5, 3, mode="s")
    .vertices(tag="outer")
    .fillet(0.2)
    .finalize()
    .extrude(0.5)
)
```

Sketch API is available after the *sketch()* call and original *workplane*.

When multiple elements are selected before constructing the sketch, multiple sketches will be created.

```
import cadquery as cq

result = (
    cq.Workplane()
    .box(5, 5, 1)
    .faces(">Z")
    .workplane()
    .rarray(2, 2, 2, 2)
    .rect(1.5, 1.5)
    .extrude(0.5)
    .faces(">Z")
    .sketch()
    .circle(0.4)
    .wires()
    .distribute(6)
    .circle(0.1, mode="a")
    .clean()
    .finalize()
    .cutBlind(-0.5, taper=10)
)
```

Sometimes it is desired to reuse existing sketches and place them as-is on a workplane.

```
import cadquery as cq
```

(continues on next page)

(continued from previous page)

```
s = cq.Sketch().trapezoid(3, 1, 110).vertices().fillet(0.2)

result = (
    cq.Workplane()
    .box(5, 5, 5)
    .faces(">X")
    .workplane()
    .transformed((0, 0, -90))
    .placeSketch(s)
    .cutThruAll()
)
```

Reusing of existing sketches is needed when using *loft()*.

```
from cadquery import Workplane, Sketch, Vector, Location

s1 = Sketch().trapezoid(3, 1, 110).vertices().fillet(0.2)

s2 = Sketch().rect(2, 1).vertices().fillet(0.2)

result = Workplane().placeSketch(s1, s2.moved(Location(Vector(0, 0, 3)))).loft()
```

When lofting only outer wires are taken into account and inner wires are silently ignored. Note that only sketches on the top of stack are considered for the current operation (i.e. there are no pending sketches), so when lofting or sweeping all relevant sketches have to be added in one *placeSketch* call.

3.7 Assemblies

3.7.1 Assembly tutorial

The purpose of this section is to demonstrate how to use the assembly and constraints functionality to build a realistic model. It will be a enclosure door assembly made out of 20x20 v-slot profiles.

Defining parameters

We want to start with defining the model parameters to allow for easy dimension changes later:

```
import cadquery as cq

# Parameters
H = 400
W = 200
D = 350

PROFILE = cq.importers.importDXF("vslot-2020_1.dxf").wires()

SLOT_D = 5
PANEL_T = 3

HANDLE_D = 20
```

(continues on next page)

(continued from previous page)

```
HANDLE_L = 50
HANDLE_W = 4
```

It is interesting to note that the v-slot profile is imported from a DXF file. This way it is very easy to change to other aluminum extrusion type, e.g. Item or Bosch. Vendors usually provide DXF files.

Defining reusable components

Next we want to define functions generating the assembly components based on the specified parameters.

```
def make_vslot(l):
    return PROFILE.toPending().extrude(l)


def make_connector():
    rv = (
        cq.Workplane()
        .box(20, 20, 20)
        .faces("<X")
        .workplane()
        .cboreHole(6, 15, 18)
        .faces("<Z")
        .workplane(centerOption="CenterOfMass")
        .cboreHole(6, 15, 18)
    )

    # tag mating faces
    rv.faces(">X").tag("X").end()
    rv.faces(">Z").tag("Z").end()

    return rv


def make_panel(w, h, t, cutout):
    rv = (
        cq.Workplane("XZ")
        .rect(w, h)
        .extrude(t)
        .faces(">Y")
        .vertices()
        .rect(2 * cutout, 2 * cutout)
        .cutThruAll()
        .faces("<Y")
        .workplane()
        .pushPoints([(-w / 3, HANDLE_L / 2), (-w / 3, -HANDLE_L / 2)])
        .hole(3)
    )

    # tag mating edges
    rv.faces(">Y").edges("%CIRCLE").edges(">Z").tag("hole1")
    rv.faces(">Y").edges("%CIRCLE").edges("<Z").tag("hole2")
```

(continues on next page)

(continued from previous page)

```
    return rv


def make_handle(w, h, r):
    pts = ((0, 0), (w, 0), (w, h), (0, h))

    path = cq.Workplane().polyline(pts)

    rv = (
        cq.Workplane("YZ")
        .rect(r, r)
        .sweep(path, transition="round")
        .tag("solid")
        .faces("<X")
        .workplane()
        .faces("<X", tag="solid")
        .hole(r / 1.5)
    )

    # tag mating faces
    rv.faces("<X").faces(">Y").tag("mate1")
    rv.faces("<X").faces("<Y").tag("mate2")

    return rv
```

Initial assembly

Next we want to instantiate all the components and add them to the assembly.

```
# define the elements
door = (
    cq.Assembly()
    .add(make_vslot(H), name="left")
    .add(make_vslot(H), name="right")
    .add(make_vslot(W), name="top")
    .add(make_vslot(W), name="bottom")
    .add(make_connector(), name="con_tl", color=cq.Color("black"))
    .add(make_connector(), name="con_tr", color=cq.Color("black"))
    .add(make_connector(), name="con_bl", color=cq.Color("black"))
    .add(make_connector(), name="con_br", color=cq.Color("black"))
    .add(
        make_panel(W + SLOT_D, H + SLOT_D, PANEL_T, SLOT_D),
        name="panel",
        color=cq.Color(0, 0, 1, 0.2),
    )
    .add(
        make_handle(HANDLE_D, HANDLE_L, HANDLE_W),
        name="handle",
        color=cq.Color("yellow"),
    )
```

(continues on next page)

(continued from previous page)

```
)
```

Constraints definition

Then we want to define all the constraints

```
# define the constraints
(
    door
    # left profile
    .constrain("left@faces@<Z", "con_bl?Z", "Plane")
    .constrain("left@faces@<X", "con_bl?X", "Axis")
    .constrain("left@faces@>Z", "con_tl?Z", "Plane")
    .constrain("left@faces@<X", "con_tl?X", "Axis")
    # top
    .constrain("top@faces@<Z", "con_tl?X", "Plane")
    .constrain("top@faces@<Y", "con_tl@faces@>Y", "Axis")
    # bottom
    .constrain("bottom@faces@<Y", "con_bl@faces@>Y", "Axis")
    .constrain("bottom@faces@>Z", "con_bl?X", "Plane")
    # right connectors
    .constrain("top@faces@>Z", "con_tr@faces@>X", "Plane")
    .constrain("bottom@faces@<Z", "con_br@faces@>X", "Plane")
    .constrain("left@faces@>Z", "con_tr?Z", "Axis")
    .constrain("left@faces@<Z", "con_br?Z", "Axis")
    # right profile
    .constrain("right@faces@>Z", "con_tr@faces@>Z", "Plane")
    .constrain("right@faces@<X", "left@faces@<X", "Axis")
    # panel
    .constrain("left@faces@>X[-4]", "panel@faces@<X", "Plane")
    .constrain("left@faces@>Z", "panel@faces@>Z", "Axis")
    # handle
    .constrain("panel?hole1", "handle?mate1", "Plane")
    .constrain("panel?hole2", "handle?mate2", "Point")
)
```

Should you need to do something unusual that is not possible with the string based selectors (e.g. use *cadquery.selectors.BoxSelector* or a user-defined selector class), it is possible to pass *cadquery.Shape* objects to the *cadquery.Assembly.constrain()* method directly. For example, the above

```
.constrain("part1@faces@>Z", "part3@faces@<Z", "Axis")
```

is equivalent to

```
.constrain("part1", part1.faces(">z").val(), "part3", part3.faces("<Z").val(), "Axis")
```

This method requires a *cadquery.Shape* object, so remember to use the *cadquery.Workplane.val()* method to pass a single *cadquery.Shape* and not the whole *cadquery.Workplane* object.

Final result

Below is the complete code including the final solve step.

```
import cadquery as cq

# Parameters
H = 400
W = 200
D = 350

PROFILE = cq.importers.importDXF("vslot-2020_1.dxf").wires()

SLOT_D = 6
PANEL_T = 3

HANDLE_D = 20
HANDLE_L = 50
HANDLE_W = 4


def make_vslot(l):
    return PROFILE.toPending().extrude(l)


def make_connector():
    rv = (
        cq.Workplane()
        .box(20, 20, 20)
        .faces("<X")
        .workplane()
        .cboreHole(6, 15, 18)
        .faces("<Z")
        .workplane(centerOption="CenterOfMass")
        .cboreHole(6, 15, 18)
    )

    # tag mating faces
    rv.faces(">X").tag("X").end()
    rv.faces(">Z").tag("Z").end()

    return rv


def make_panel(w, h, t, cutout):
    rv = (
        cq.Workplane("XZ")
        .rect(w, h)
        .extrude(t)
        .faces(">Y")
        .vertices()
        .rect(2 * cutout, 2 * cutout)
        .cutThruAll()
```

(continues on next page)

(continued from previous page)

```
        .faces("<Y")
        .workplane()
        .pushPoints([(-w / 3, HANDLE_L / 2), (-w / 3, -HANDLE_L / 2)])
        .hole(3)
    )

    # tag mating edges
    rv.faces(">Y").edges("%CIRCLE").edges(">Z").tag("hole1")
    rv.faces(">Y").edges("%CIRCLE").edges("<Z").tag("hole2")

    return rv


def make_handle(w, h, r):
    pts = ((0, 0), (w, 0), (w, h), (0, h))

    path = cq.Workplane().polyline(pts)

    rv = (
        cq.Workplane("YZ")
        .rect(r, r)
        .sweep(path, transition="round")
        .tag("solid")
        .faces("<X")
        .workplane()
        .faces("<X", tag="solid")
        .hole(r / 1.5)
    )

    # tag mating faces
    rv.faces("<X").faces(">Y").tag("mate1")
    rv.faces("<X").faces("<Y").tag("mate2")

    return rv


# define the elements
door = (
    cq.Assembly()
    .add(make_vslot(H), name="left")
    .add(make_vslot(H), name="right")
    .add(make_vslot(W), name="top")
    .add(make_vslot(W), name="bottom")
    .add(make_connector(), name="con_tl", color=cq.Color("black"))
    .add(make_connector(), name="con_tr", color=cq.Color("black"))
    .add(make_connector(), name="con_bl", color=cq.Color("black"))
    .add(make_connector(), name="con_br", color=cq.Color("black"))
    .add(
        make_panel(W + 2 * SLOT_D, H + 2 * SLOT_D, PANEL_T, SLOT_D),
        name="panel",
        color=cq.Color(0, 0, 1, 0.2),
    )
```

(continues on next page)

(continued from previous page)

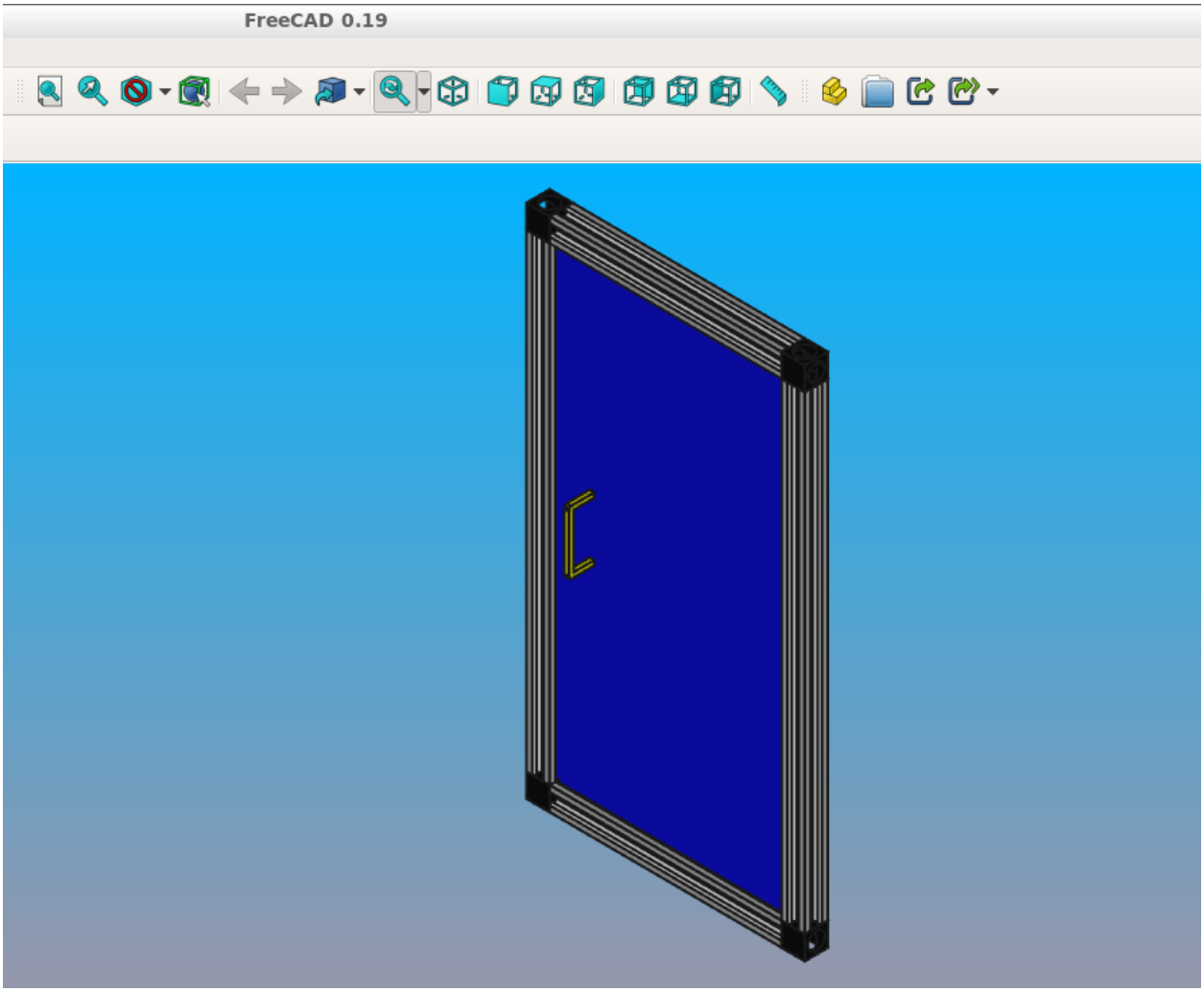
```
    .add(
        make_handle(HANDLE_D, HANDLE_L, HANDLE_W),
        name="handle",
        color=cq.Color("yellow"),
    )
)

# define the constraints
(
    door
    # left profile
    .constrain("left@faces@<Z", "con_bl?Z", "Plane")
    .constrain("left@faces@<X", "con_bl?X", "Axis")
    .constrain("left@faces@>Z", "con_tl?Z", "Plane")
    .constrain("left@faces@<X", "con_tl?X", "Axis")
    # top
    .constrain("top@faces@<Z", "con_tl?X", "Plane")
    .constrain("top@faces@<Y", "con_tl@faces@>Y", "Axis")
    # bottom
    .constrain("bottom@faces@<Y", "con_bl@faces@>Y", "Axis")
    .constrain("bottom@faces@>Z", "con_bl?X", "Plane")
    # right connectors
    .constrain("top@faces@>Z", "con_tr@faces@>X", "Plane")
    .constrain("bottom@faces@<Z", "con_br@faces@>X", "Plane")
    .constrain("left@faces@>Z", "con_tr?Z", "Axis")
    .constrain("left@faces@<Z", "con_br?Z", "Axis")
    # right profile
    .constrain("right@faces@>Z", "con_tr@faces@>Z", "Plane")
    .constrain("right@faces@<X", "left@faces@<X", "Axis")
    # panel
    .constrain("left@faces@>X[-4]", "panel@faces@<X", "Plane")
    .constrain("left@faces@>Z", "panel@faces@>Z", "Axis")
    # handle
    .constrain("panel?hole1", "handle?mate1", "Plane")
    .constrain("panel?hole2", "handle?mate2", "Point")
)

# solve
door.solve()

show_object(door, name="door")
```

Data export

The resulting assembly can be exported as a STEP file or in a internal OCCT XML format.

STEP can be loaded in all CAD tool, e.g. in FreeCAD and the XML be used in other applications using OCCT.

```
door.save("door.step")
door.save("door.xml")
```



3.7.2 Object locations

Objects can be added to an assembly with initial locations supplied, such as:

```
import cadquery as cq

cone = cq.Solid.makeCone(1, 0, 2)

assy = cq.Assembly()
assy.add(
    cone,
```

(continues on next page)

(continued from previous page)

```
    loc=cq.Location((0, 0, 0), (1, 0, 0), 180),
    name="cone0",
    color=cq.Color("green"),
)
assy.add(cone, name="cone1", color=cq.Color("blue"))

show_object(assy)
```

As an alternative to the user calculating locations, constraints and the method *solve()* can be used to position objects in an assembly.

If initial locations and the method *solve()* are used the solver will overwrite these initial locations with it's solution, however initial locations can still affect the final solution. In an underconstrained system the solver may not move an object if it does not contribute to the cost function, or if multiple solutions exist (ie. multiple instances where the cost function is at a minimum) initial locations can cause the solver to converge on one particular solution. For very complicated assemblies setting approximately correct initial locations can also reduce the computational time required.

3.7.3 Constraints

Constraints are often a better representation of the real world relationship the user wants to model than directly supplying locations. In the above example the real world relationship is that the bottom face of each cone should touch, which can be modelled with a Plane constraint. When the user provides explicit locations (instead of constraints) then they are also responsible for updating them when, for example, the location of `cone1` changes.

When at least one constraint is supplied and the method *solve()* is run, an optimization problem is set up. Each constraint provides a cost function that depends on the position and orientation (represented by a *Location*) of the two objects specified when creating the constraint. The solver varies the location of the assembly's children and attempts to minimize the sum of all cost functions. Hence by reading the formulae of the cost functions below, you can understand exactly what each constraint does.

Point

The Point constraint is a frequently used constraint that minimizes the distance between two points. Some example uses are centering faces or aligning verticies, but it is also useful with dummy vertices to create offsets between two parts.

The cost function is:

$$(param - |\vec{c_1} - \vec{c_2}|)^2$$

Where:

- $param$ is the parameter of the constraint, which defaults to 0,
- $\vec{c_i}$ is the center of the ith object, and
- $|\vec{v}|$ is the modulus of $\vec{v}$, ie. the length of $\vec{v}$.

When creating a Point constraint, the `param` argument can be used to specify a desired offset between the two centers. This offset does not have a direction associated with it, if you want to specify an offset in a specific direction then you should use a dummy *Vertex*.

The Point constraint uses the *Center()* to find the center of the argument. Hence it will work with all subclasses of *Shape*.

```
import cadquery as cq

# Use the Point constraint to position boxes relative to an arc
line = cq.Edge.makeCircle(radius=10, angle1=0, angle2=90)
box = cq.Workplane().box(1, 1, 1)

assy = cq.Assembly()
assy.add(line, name="line")

# position the red box on the center of the arc
assy.add(box, name="box0", color=cq.Color("red"))
assy.constrain("line", "box0", "Point")

# position the green box at a normalized distance of 0.8 along the arc
position0 = line.positionAt(0.8)
assy.add(box, name="box1", color=cq.Color("green"))
assy.constrain(
    "line",
    cq.Vertex.makeVertex(*position0.toTuple()),
    "box1",
    box.val(),
    "Point",
)

# position the orange box 2 units in any direction from the green box
assy.add(box, name="box2", color=cq.Color("orange"))
assy.constrain(
    "line",
    cq.Vertex.makeVertex(*position0.toTuple()),
    "box2",
    box.val(),
    "Point",
    param=2,
)

# position the blue box offset 2 units in the x direction from the green box
position1 = position0 + cq.Vector(2, 0, 0)
assy.add(box, name="box3", color=cq.Color("blue"))
assy.constrain(
    "line",
    cq.Vertex.makeVertex(*position1.toTuple()),
    "box3",
    box.val(),
    "Point",
)

assy.solve()
show_object(assy)
```

Axis

The Axis constraint minimizes the angle between two vectors. It is frequently used to align faces and control the rotation of an object.

The cost function is:

$$(k_{dir} \times (param - \vec{d_1} \angle \vec{d_2})^2$$

Where:

- k_{dir} is a scaling factor for directional constraints,
- $param$ is the parameter of the constraint, which defaults to 180 degrees,
- $\vec{d_i}$ is the direction created from the ith object argument as described below, and
- $\vec{d_1} \angle \vec{d_2}$ is the angle between $\vec{d_1}$ and $\vec{d_2}$.

The argument `param` defaults to 180 degrees, which sets the two directions opposite to each other. This represents what is often called a "mate" relationship, where the external faces of two objects touch.

```
import cadquery as cq

cone = cq.Solid.makeCone(1, 0, 2)

assy = cq.Assembly()
assy.add(cone, name="cone0", color=cq.Color("green"))
assy.add(cone, name="cone1", color=cq.Color("blue"))
assy.constrain("cone0@faces@<Z", "cone1@faces@<Z", "Axis")

assy.solve()
show_object(assy)
```

If the `param` argument is set to zero, then the two objects will point in the same direction. This is often used when one object goes through another, such as a pin going into a hole in a plate:

```
import cadquery as cq

plate = cq.Workplane().box(10, 10, 1).faces(">Z").workplane().hole(2)
cone = cq.Solid.makeCone(0.8, 0, 4)

assy = cq.Assembly()
assy.add(plate, name="plate", color=cq.Color("green"))
assy.add(cone, name="cone", color=cq.Color("blue"))
# place the center of the flat face of the cone in the center of the upper face of the␣
↪plate
assy.constrain("plate@faces@>Z", "cone@faces@<Z", "Point")

# set both the flat face of the cone and the upper face of the plate to point in the␣
↪same direction
assy.constrain("plate@faces@>Z", "cone@faces@<Z", "Axis", param=0)

assy.solve()
show_object(assy)
```

In creating an Axis constraint, a direction vector is extracted in one of three different ways, depending on the object's type.

***Face*:**
Using *normalAt()*

***Edge* and *geomType()* is "CIRCLE":**
Using `normal()`

***Edge* and *geomType()* is not "CIRCLE":**
Using `tangentAt()`

Using any other type of object will raise a `ValueError`. By far the most common use case is to define an Axis constraint from a *Face*.

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

# Create a sinusoidal surface:
surf = cq.Workplane().parametricSurface(
    lambda u, v: (u, v, 5 * sin(pi * u / 10) * cos(pi * v / 10)),
    N=40,
    start=0,
    stop=20,
)

# Create a cone with a small, flat tip:
cone = (
    cq.Workplane()
    .add(cq.Solid.makeCone(1, 0.1, 2))
    # tag the tip for easy reference in the constraint:
    .faces(">Z")
    .tag("tip")
    .end()
)

assy = cq.Assembly()
assy.add(surf, name="surf", color=cq.Color("lightgray"))
assy.add(cone, name="cone", color=cq.Color("green"))
# set the Face on the tip of the cone to point in
# the opposite direction of the center of the surface:
assy.constrain("surf", "cone?tip", "Axis")
# to make the example clearer, move the cone to the center of the face:
assy.constrain("surf", "cone?tip", "Point")
assy.solve()

show_object(assy)
```

Plane

The Plane constraint is simply a combination of both the Point and Axis constraints. It is a convenient shortcut for a commonly used combination of constraints. It can be used to shorten the previous example from the two constraints to just one:

```
assy = cq.Assembly()
assy.add(surf, name="surf", color=cq.Color("lightgray"))
assy.add(cone, name="cone", color=cq.Color("green"))
-# set the Face on the tip of the cone to point in
-# the opposite direction of the center of the surface:
-assy.constrain("surf", "cone?tip", "Axis")
-# to make the example clearer, move the cone to the center of the face:
-assy.constrain("surf", "cone?tip", "Point")
+assy.constrain("surf", "cone?tip", "Plane")
assy.solve()

show_object(assy)
```

The result of this code is identical to the above two constraint example.

For the cost function of Plane, please see the Point and Axis sections. The `param` argument is applied to Axis and should be left as the default value for a "mate" style constraint (two surfaces touching) or can be set to `0` for a through surface constraint (see description in the Axis constraint section).

PointInPlane

PointInPlane positions the center of the first object within the plane defined by the second object. The cost function is:

$$\text{dist}(\vec{c}, p_{\text{offset}})^2$$

Where:

- $\vec{c}$ is the center of the first argument,
- p_{offset} is a plane created from the second object, offset in the plane's normal direction by `param`, and
- $\text{dist}(\vec{a}, b)$ is the distance between point $\vec{a}$ and plane b.

```
import cadquery as cq

# Create an L-shaped object:
bracket = (
    cq.Workplane("YZ")
    .hLine(1)
    .vLine(0.1)
    .hLineTo(0.2)
    .vLineTo(1)
    .hLineTo(0)
    .close()
    .extrude(1)
    # tag some faces for easy reference:
    .faces(">Y[1]")
    .tag("inner_vert")
    .end()
```

(continues on next page)

(continued from previous page)

```
    .faces(">Z[1]")
    .tag("inner_horiz")
    .end()
)

box = cq.Workplane().box(0.5, 0.5, 0.5)

assy = cq.Assembly()
assy.add(bracket, name="bracket", color=cq.Color("gray"))
assy.add(box, name="box", color=cq.Color("green"))

# lock bracket orientation:
assy.constrain("bracket@faces@>Z", "box@faces@>Z", "Axis", param=0)
assy.constrain("bracket@faces@>X", "box@faces@>X", "Axis", param=0)

# constrain the bottom of the box to be on the plane defined by inner_horiz:
assy.constrain("box@faces@<Z", "bracket?inner_horiz", "PointInPlane")
# constrain the side of the box to be 0.2 units from the plane defined by inner_vert
assy.constrain("box@faces@<Y", "bracket?inner_vert", "PointInPlane", param=0.2)
# constrain the end of the box to be 0.1 units inside the end of the bracket
assy.constrain("box@faces@>X", "bracket@faces@>X", "PointInPlane", param=-0.1)

assy.solve()
show_object(assy)
```

PointOnLine

PointOnLine positions the center of the first object on the line defined by the second object. The cost function is:

$$(param - \mathrm{dist}(\vec{c}, l))^2$$

Where:

- $\vec{c}$ is the center of the first argument,
- l is a line created from the second object
- $param$ is the parameter of the constraint, which defaults to 0,
- $\mathrm{dist}(\vec{a}, b)$ is the distance between point $\vec{a}$ and line b.

```
import cadquery as cq

b1 = cq.Workplane().box(1, 1, 1)
b2 = cq.Workplane().sphere(0.15)

assy = (
    cq.Assembly()
    .add(b1, name="b1")
    .add(b2, loc=cq.Location((0, 0, 4)), name="b2", color=cq.Color("red"))
)

# fix the position of b1
```

(continues on next page)

(continued from previous page)

```
assy.constrain("b1", "Fixed")
# b2 on one of the edges of b1
assy.constrain("b2", "b1@edges@>>Z and >>Y", "PointOnLine")
# b2 on another of the edges of b1
assy.constrain("b2", "b1@edges@>>Z and >>X", "PointOnLine")
# effectively b2 will be constrained to be on the intersection of the two edges

assy.solve()
show_object(assy)
```

FixedPoint

FixedPoint fixes the position of the given argument to be equal to the given point specified via the parameter of the constraint. This constraint locks all translational degrees of freedom of the argument. The cost function is:

$$\|\vec{c} - \vec{param}\|^2$$

Where:

- $\vec{c}$ is the center of the argument,
- $param$ is the parameter of the constraint - tuple specifying the target position.

```
import cadquery as cq

b1 = cq.Workplane().box(1, 1, 1)
b2 = cq.Workplane().sphere(0.15)

assy = (
    cq.Assembly()
    .add(b1, name="b1")
    .add(b2, loc=cq.Location((0, 0, 4)), name="b2", color=cq.Color("red"))
    .add(b1, loc=cq.Location((-2, 0, 0)), name="b3", color=cq.Color("red"))
)

pnt = (0.5, 0.5, 0.5)

# fix the position of b1
assy.constrain("b1", "Fixed")
# fix b2 center at point
assy.constrain("b2", "FixedPoint", pnt)
# fix b3 vertex position at point
assy.constrain("b3@vertices@<X and <Y and <Z", "FixedPoint", pnt)

assy.solve()
show_object(assy)
```

FixedRotation

FixedRotation fixes the rotation of the given argument to be equal to the value specified via the parameter of the constraint.

This constraint locks all rotational degrees of freedom of the argument. The cost function is:

$$\left\| \vec{R} - \vec{param} \right\|^2$$

Where:

- $\vec{R}$ vector of the rotation angles of the rotation applied to the argument,
- $param$ is the parameter of the constraint - tuple specifying the target rotation.

```
import cadquery as cq

b1 = cq.Workplane().box(1, 1, 1)
b2 = cq.Workplane().rect(0.1, 0.1).extrude(1, taper=-15)

assy = (
    cq.Assembly()
    .add(b1, name="b1")
    .add(b2, loc=cq.Location((0, 0, 4)), name="b2", color=cq.Color("red"))
)

# fix the position of b1
assy.constrain("b1", "Fixed")
# fix b2 bottom face position (but not rotation)
assy.constrain("b2@faces@<Z", "FixedPoint", (0, 0, 0.5))
# fix b2 rotational degrees of freedom too
assy.constrain("b2", "FixedRotation", (45, 0, 45))

assy.solve()
show_object(assy)
```

FixedAxis

FixedAxis fixes the orientation of the given argument's normal or tangent to be equal to the orientation of the vector specified via the parameter of the constraint. This constraint locks two rotational degrees of freedom of the argument. The cost function is:

$$(\vec{a} \angle \vec{param})^2$$

Where:

- $\vec{a}$ normal or tangent vector of the argument,
- $param$ is the parameter of the constraint - tuple specifying the target direction.

```
import cadquery as cq

b1 = cq.Workplane().box(1, 1, 1)
b2 = cq.Workplane().rect(0.1, 0.1).extrude(1, taper=-15)
```

(continues on next page)

(continued from previous page)

```
assy = (
    cq.Assembly()
    .add(b1, name="b1")
    .add(b2, loc=cq.Location((0, 0, 4)), name="b2", color=cq.Color("red"))
)

# fix the position of b1
assy.constrain("b1", "Fixed")
# fix b2 bottom face position (but not rotation)
assy.constrain("b2@faces@<Z", "FixedPoint", (0, 0, 0.5))
# fix b2 some rotational degrees of freedom too
assy.constrain("b2@faces@>Z", "FixedAxis", (1, 0, 2))

assy.solve()
show_object(assy)
```

3.7.4 Assembly colors

Aside from RGBA values, the *Color* class can be instantiated from a text name. Valid names are listed along with a color sample below:

3.8 CadQuery Scripts and Object Output

CadQuery scripts are pure Python scripts, that may follow a few conventions.

If you are using CadQuery as a library, there are no constraints.

If you are using CadQuery scripts inside of a CadQuery execution environment like CQ-editor, there are a few conventions you need to be aware of:

- cadquery is usually imported as 'cq' at the top of a script
- to return an object to the execution environment (like CQ-editor) for rendering, you need to call the show_object() method

Each script generally has three sections:

- Variable Assignments and metadata definitions
- CadQuery and other Python code
- object export or rendering, via the show_object() function

see the *The CadQuery Gateway Interface* section for more details.

3.9 Examples

The examples on this page can help you learn how to build objects with CadQuery.

They are organized from simple to complex, so working through them in order is the best way to absorb them.

Each example lists the API elements used in the example for easy reference. Items introduced in the example are marked with a **!**

Note: We strongly recommend installing CQ-editor, so that you can work along with these examples interactively. See *Installing CadQuery* for more info.

If you do, make sure to take these steps so that they work:

1. import cadquery as cq
2. add the line `show_object(result)` at the end. The samples below are autogenerated, but they use a different syntax than the models on the website need to be.

List of Examples

- *Extruding until a given face*
- *Making Counter-bored and Counter-sunk Holes*
- *Offsetting wires in 2D*
- *Rounding Corners with Fillet*
- *Tagging objects*
- *A Parametric Bearing Pillow Block*
- *Splitting an Object*
- *The Classic OCC Bottle*
- *A Parametric Enclosure*
- *Lego Brick*
- *Braille Example*
- *Panel With Various Connector Holes*
- *Cycloidal gear*

3.9.1 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()* !

3.9.2 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)
```

(continues on next page)

(continued from previous page)

```
    .faces(">Z")
    .workplane()
    .hole(center_hole_dia)
)
```

Api References

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

3.9.3 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()*

3.9.4 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()*

3.9.5 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()*

3.9.6 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()*

3.9.7 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()*

3.9.8 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()*

3.9.9 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()*

3.9.10 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()*

3.9.11 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)

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()*

3.9.12 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()*

3.9.13 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()*

3.9.14 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()* !

3.9.15 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()*

3.9.16 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,
```

(continues on next page)

(continued from previous page)

```
    # 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()*

3.9.17 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()*

3.9.18 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()
```

(continues on next page)

(continued from previous page)

```
    .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()*

3.9.19 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()*

3.9.20 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()*

3.9.21 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()*

3.9.22 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()*

3.9.23 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()*

3.9.24 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()*

3.9.25 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()*

3.9.26 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)
)
```

3.9.27 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()*

3.9.28 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()*

3.9.29 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()*

3.9.30 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
```

3.9.31 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)
```

3.9.32 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()
    )
```

3.9.33 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()
)
```

3.10 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.

This page lists methods of these objects grouped by **functional area**

See also:

This page lists api methods grouped by functional area. Use *CadQuery Class Summary* to see methods alphabetically by class.

3.10.1 Sketch initialization

Creating new sketches.

Sketch(parent, locs)	2D sketch.
Sketch.importDXF(filename[, tol, exclude, ...])	Import a DXF file and construct face(s)
Workplane.sketch()	Initialize and return a sketch
Sketch.finalize()	Finish sketch construction and return the parent.
Sketch.copy()	Create a partial copy of the sketch.
Sketch.located(loc)	Create a partial copy of the sketch with a new location.
Sketch.moved(loc)	Create a partial copy of the sketch with moved _faces.

3.10.2 Sketch selection

Selecting, tagging and manipulating elements.

Sketch.tag(tag)	Tag current selection.
Sketch.select(*tags)	Select based on tags.
Sketch.reset()	Reset current selection.
Sketch.delete()	Delete selected object.
Sketch.faces([s, tag])	Select faces.
Sketch.edges([s, tag])	Select edges.
Sketch.vertices([s, tag])	Select vertices.

3.10.3 Sketching with faces

Sketching using the face-based API.

Sketch.face(b[, angle, mode, tag, ...])	Construct a face from a wire or edges.
Sketch.rect(w, h[, angle, mode, tag])	Construct a rectangular face.
Sketch.circle(r[, mode, tag])	Construct a circular face.
Sketch.ellipse(a1, a2[, angle, mode, tag])	Construct an elliptical face.
Sketch.trapezoid(w, h, a1[, a2, angle, ...])	Construct a trapezoidal face.
Sketch.slot(w, h[, angle, mode, tag])	Construct a slot-shaped face.
Sketch.regularPolygon(r, n[, angle, mode, tag])	Construct a regular polygonal face.
Sketch.polygon(pts[, angle, mode, tag])	Construct a polygonal face.
Sketch.rarray(xs, ys, nx, ny)	Generate a rectangular array of locations.
Sketch.parray(r, a1, da, n[, rotate])	Generate a polar array of locations.
Sketch.distribute(n[, start, stop, rotate])	Distribute locations along selected edges or wires.
Sketch.each(callback[, mode, tag, ...])	Apply a callback on all applicable entities.
Sketch.push(locs[, tag])	Set current selection to given locations or points.
Sketch.hull([mode, tag])	Generate a convex hull from current selection or all objects.
Sketch.offset(d[, mode, tag])	Offset selected wires or edges.
Sketch.fillet(d)	Add a fillet based on current selection.
Sketch.chamfer(d)	Add a chamfer based on current selection.
Sketch.clean()	Remove internal wires.

3.10.4 Sketching with edges and constraints

Sketching using the edge-based API.

Sketch.edge(val[, tag, forConstruction])	Add an edge to the sketch.
Sketch.segment(...)	Construct a segment.
Sketch.arc(...)	Construct an arc.
Sketch.spline(...)	Construct a spline edge.
Sketch.close([tag])	Connect last edge to the first one.
Sketch.assemble([mode, tag])	Assemble edges into faces.
Sketch.constrain(...)	Add a constraint.
Sketch.solve()	Solve current constraints and update edge positions.

3.10.5 Initialization

Creating new workplanes and object chains

Workplane(, obj=None))	Defines a coordinate system in space, in which 2D coordinates can be used.

3.10.6 2D Operations

Creating 2D constructs that can be used to create 3D features.

All 2D operations require a **Workplane** object to be created.

Workplane.center(x, y)	Shift local coordinates to the specified location.
Workplane.lineTo(x, y[, forConstruction])	Make a line from the current point to the provided point
Workplane.line(xDist, yDist[, forConstruction])	Make a line from the current point to the provided point, using dimensions relative to the current point
Workplane.vLine(distance[, forConstruction])	Make a vertical line from the current point the provided distance
Workplane.vLineTo(yCoord[, forConstruction])	Make a vertical line from the current point to the provided y coordinate.
Workplane.hLine(distance[, forConstruction])	Make a horizontal line from the current point the provided distance
Workplane.hLineTo(xCoord[, forConstruction])	Make a horizontal line from the current point to the provided x coordinate.
Workplane.polarLine(distance, angle[, ...])	Make a line of the given length, at the given angle from the current point
Workplane.polarLineTo(distance, angle[, ...])	Make a line from the current point to the given polar coordinates
Workplane.moveTo([x, y])	Move to the specified point, without drawing.
Workplane.move([xDist, yDist])	Move the specified distance from the current point, without drawing.
Workplane.spline(listOfXYTuple[, tangents, ...])	Create a spline interpolated through the provided points (2D or 3D).
Workplane.parametricCurve(func[, N, start, ...])	Create a spline curve approximating the provided function.
Workplane.parametricSurface(func[, N, ...])	Create a spline surface approximating the provided function.
Workplane.threePointArc(point1, point2[, ...])	Draw an arc from the current point, through point1, and ending at point2
Workplane.sagittaArc(endPoint, sag[, ...])	Draw an arc from the current point to endPoint with an arc defined by the sag (sagitta).
Workplane.radiusArc(endPoint, radius[, ...])	Draw an arc from the current point to endPoint with an arc defined by the radius.
Workplane.tangentArcPoint(endpoint[, ...])	Draw an arc as a tangent from the end of the current edge to endpoint.
Workplane.mirrorY()	Mirror entities around the y axis of the workplane plane.
Workplane.mirrorX()	Mirror entities around the x axis of the workplane plane.
Workplane.wire([forConstruction])	Returns a CQ object with all pending edges connected into a wire.
Workplane.rect(xLen, yLen[, centered, ...])	Make a rectangle for each item on the stack.
Workplane.circle(radius[, forConstruction])	Make a circle for each item on the stack.
Workplane.ellipse(x_radius, y_radius[, ...])	Make an ellipse for each item on the stack.
Workplane.ellipseArc(x_radius, y_radius[, ...])	Draw an elliptical arc with x and y radiuses either with start point at current point or or current point being the center of the arc
Workplane.polyline(listOfXYTuple[, ...])	Create a polyline from a list of points
Workplane.close()	End construction, and attempt to build a closed wire.

continues on next page

Table 1 – continued from previous page

Workplane.rarray(xSpacing, ySpacing, xCount, ...)	Creates an array of points and pushes them onto the stack.
Workplane.polarArray(radius, startAngle, ...)	Creates a polar array of points and pushes them onto the stack.
Workplane.slot2D(length, diameter[, angle])	Creates a rounded slot for each point on the stack.
Workplane.offset2D(d[, kind, forConstruction])	Creates a 2D offset wire.
Workplane.placeSketch(*sketches)	Place the provided sketch(es) based on the current items on the stack.

3.10.7 3D Operations

Some 3D operations also require an active 2D workplane, but some do not.

3D operations that require a 2D workplane to be active:

Workplane.cboreHole(diameter, cboreDiameter, ...)	Makes a counterbored hole for each item on the stack.
Workplane.cskHole(diameter, cskDiameter, ...)	Makes a countersunk hole for each item on the stack.
Workplane.hole(diameter[, depth, clean])	Makes a hole for each item on the stack.
Workplane.extrude(until[, combine, clean, ...])	Use all un-extruded wires in the parent chain to create a prismatic solid.
Workplane.cut(toCut[, clean, tol])	Cuts the provided solid from the current solid, IE, perform a solid subtraction.
Workplane.cutBlind(until[, clean, both, taper])	Use all un-extruded wires in the parent chain to create a prismatic cut from existing solid.
Workplane.cutThruAll([clean, taper])	Use all un-extruded wires in the parent chain to create a prismatic cut from existing solid.
Workplane.box(length, width, height[, ...])	Return a 3d box with specified dimensions for each object on the stack.
Workplane.sphere(radius[, direct, angle1, ...])	Returns a 3D sphere with the specified radius for each point on the stack.
Workplane.wedge(dx, dy, dz, xmin, zmin, ...)	Returns a 3D wedge with the specified dimensions for each point on the stack.
Workplane.cylinder(height, radius[, direct, ...])	Returns a cylinder with the specified radius and height for each point on the stack
Workplane.union([toUnion, clean, glue, tol])	Unions all of the items on the stack of toUnion with the current solid.
Workplane.combine([clean, glue, tol])	Attempts to combine all of the items on the stack into a single item.
Workplane.intersect(toIntersect[, clean, tol])	Intersects the provided solid from the current solid.
Workplane.loft([ruled, combine, clean])	Make a lofted solid, through the set of wires.
Workplane.sweep(path[, multisection, ...])	Use all un-extruded wires in the parent chain to create a swept solid.
Workplane.twistExtrude(distance, angleDegrees)	Extrudes a wire in the direction normal to the plane, but also twists by the specified angle over the length of the extrusion.
Workplane.revolve([angleDegrees, axisStart, ...])	Use all un-revolved wires in the parent chain to create a solid.
Workplane.text(txt, fontsize, distance[, ...])	Returns a 3D text.

3D operations that do NOT require a 2D workplane to be active:

Workplane.shell(thickness[, kind])	Remove the selected faces to create a shell of the specified thickness.
Workplane.fillet(radius)	Fillets a solid on the selected edges.
Workplane.chamfer(length[, length2])	Chamfers a solid on the selected edges.
Workplane.split()	Splits a solid on the stack into two parts, optionally keeping the separate parts.
Workplane.rotate(axisStartPoint, ...)	Returns a copy of all of the items on the stack rotated through and angle around the axis of rotation.
Workplane.rotateAboutCenter(axisEndPoint, ...)	Rotates all items on the stack by the specified angle, about the specified axis
Workplane.translate(vec)	Returns a copy of all of the items on the stack moved by the specified translation vector.
Workplane.mirror([mirrorPlane, ...])	Mirror a single CQ object.

3.10.8 File Management and Export

Workplane.toSvg([opts])	Returns svg text that represents the first item on the stack.
Workplane.exportSvg(fileName)	Exports the first item on the stack as an SVG file

importers.importStep(fileName)	Accepts a file name and loads the STEP file into a cadquery Workplane
importers.importDXF(filename[, tol, ...])	Loads a DXF file into a Workplane.
exporters.export(w, fname[, exportType, ...])	Export Workplane or Shape to file.
occ_impl.exporters.dxf.DxfDocument([...])	Create DXF document from CadQuery objects.

3.10.9 Iteration Methods

Methods that allow iteration over the stack or objects

Workplane.each(callback[, ...])	Runs the provided function on each value in the stack, and collects the return values into a new CQ object.
Workplane.eachpoint(callback[, ...])	Same as each(), except each item on the stack is converted into a point before it is passed into the callback function.

3.10.10 Stack and Selector Methods

CadQuery methods that operate on the stack

Workplane.all()	Return a list of all CQ objects on the stack.
Workplane.size()	Return the number of objects currently on the stack
Workplane.vals()	get the values in the current list
Workplane.add()	Adds an object or a list of objects to the stack
Workplane.val()	Return the first value on the stack.
Workplane.first()	Return the first item on the stack
Workplane.item(i)	Return the ith item on the stack.
Workplane.last()	Return the last item on the stack.
Workplane.end([n])	Return the nth parent of this CQ element
Workplane.vertices([selector, tag])	Select the vertices of objects on the stack, optionally filtering the selection.
Workplane.faces([selector, tag])	Select the faces of objects on the stack, optionally filtering the selection.
Workplane.edges([selector, tag])	Select the edges of objects on the stack, optionally filtering the selection.
Workplane.wires([selector, tag])	Select the wires of objects on the stack, optionally filtering the selection.
Workplane.solids([selector, tag])	Select the solids of objects on the stack, optionally filtering the selection.
Workplane.shells([selector, tag])	Select the shells of objects on the stack, optionally filtering the selection.
Workplane.compounds([selector, tag])	Select compounds on the stack, optionally filtering the selection.

3.10.11 Selectors

Objects that filter and select CAD objects. Selectors are used to select existing geometry as a basis for further operations.

NearestToPointSelector(pnt)	Selects object nearest the provided point.
BoxSelector(point0, point1[, boundingbox])	Selects objects inside the 3D box defined by 2 points.
BaseDirSelector(vector[, tolerance])	A selector that handles selection on the basis of a single direction vector.
ParallelDirSelector(vector[, tolerance])	Selects objects parallel with the provided direction.
DirectionSelector(vector[, tolerance])	Selects objects aligned with the provided direction.
DirectionNthSelector(vector, n[, ...])	Filters for objects parallel (or normal) to the specified direction then returns the Nth one.
LengthNthSelector(n[, directionMax, tolerance])	Select the object(s) with the Nth length
AreaNthSelector(n[, directionMax, tolerance])	Selects the object(s) with Nth area
RadiusNthSelector(n[, directionMax, tolerance])	Select the object with the Nth radius.
PerpendicularDirSelector(vector[, tolerance])	Selects objects perpendicular with the provided direction.
TypeSelector(typeString)	Selects objects having the prescribed geometry type.
DirectionMinMaxSelector(vector[, ...])	Selects objects closest or farthest in the specified direction.
CenterNthSelector(vector, n[, directionMax, ...])	Sorts objects into a list with order determined by the distance of their center projected onto the specified direction.
BinarySelector(left, right)	Base class for selectors that operates with two other selectors.
AndSelector(left, right)	Intersection selector.
SumSelector(left, right)	Union selector.
SubtractSelector(left, right)	Difference selector.
InverseSelector(selector)	Inverts the selection of given selector.
StringSyntaxSelector(selectorString)	Filter lists objects using a simple string syntax.

3.10.12 Assemblies

Workplane and Shape objects can be connected together into assemblies

Assembly([obj, loc, name, color, metadata])	Nested assembly of Workplane and Shape objects defining their relative positions.
Assembly.add()	Add a subassembly to the current assembly.
Assembly.save(path[, exportType, mode, ...])	Save assembly to a file.
Assembly.constrain()	Define a new constraint.
Assembly.solve([verbosity])	Solve the constraints.
Constraint	alias of `ConstraintSpec`
Color()	Wrapper for the OCCT color object Quantity_ColorRGBA.

3.11 Selectors Reference

CadQuery selector strings allow filtering various types of object lists. Most commonly, Edges, Faces, and Vertices are used, but all objects types can be filtered.

Object lists are created by using the following methods, which each collect a type of shape:

- *cadquery.Workplane.vertices()*
- *cadquery.Workplane.edges()*
- *cadquery.Workplane.faces()*
- *cadquery.Workplane.shells()*
- *cadquery.Workplane.solids()*

Each of these methods accepts either a Selector object or a string. String selectors are simply shortcuts for using the full object equivalents. If you pass one of the string patterns in, CadQuery will automatically use the associated selector object.

Note: String selectors are simply shortcuts to concrete selector classes, which you can use or extend. For a full description of how each selector class works, see *CadQuery Class Summary*.

If you find that the built-in selectors are not sufficient, you can easily plug in your own. See *Extending CadQuery* to see how.

3.11.1 Combining Selectors

Selectors can be combined logically, currently defined operators include **and**, **or**, **not** and **exc[ept]** (set difference). For example:

```
result = cq.Workplane("XY").box(2, 2, 2).edges("|Z and >Y").chamfer(0.2)
```

Much more complex expressions are possible as well:

```
result = (
    cq.Workplane("XY")
    .box(2, 2, 2)
    .faces(">Z")
    .shell(-0.2)
    .faces(">Z")
    .edges("not(<X or >X or <Y or >Y)")
    .chamfer(0.1)
)
```

3.11.2 Filtering Faces

All types of string selectors work on faces. In most cases, the selector refers to the direction of the **normal vector** of the face.

Warning: If a face is not planar, selectors are evaluated at the center of mass of the face. This can lead to results that are quite unexpected.

The axis used in the listing below are for illustration: any axis would work similarly in each case.

Selector	Selects	Selector Class
+Z	Faces with normal in +z direction	*cadquery.DirectionSelector*
|Z	Faces with normal parallel to z dir	*cadquery.ParallelDirSelector*
-X	Faces with normal in neg x direction	*cadquery.DirectionSelector*
#Z	Faces with normal orthogonal to z dir	*cadquery.PerpendicularDirSelector*
%Plane	Faces of type plane	*cadquery.TypeSelector*
>Y	Face farthest in the positive y dir	*cadquery.DirectionMinMaxSelector*
<Y	Face farthest in the negative y dir	*cadquery.DirectionMinMaxSelector*
>Y[-2]	2nd farthest Face **normal** to the y dir	cadquery.DirectionNthSelector
<Y[0]	1st closest Face **normal** to the y dir	cadquery.DirectionNthSelector
>>Y[-2]	2nd farthest Face in the y dir	cadquery.CenterNthSelector
<<Y[0]	1st closest Face in the y dir	cadquery.CenterNthSelector

3.11.3 Filtering Edges

The selector usually refers to the **direction** of the edge.

Warning: Non-linear edges are not selected for any string selectors except type (%) and center (>>). Non-linear edges are never returned when these filters are applied.

The axis used in the listing below are for illustration: any axis would work similarly in each case.

Selector	Selects	Selector Class
+Z	Edges aligned in the Z direction	*cadquery.DirectionSelector*
|Z	Edges parallel to z direction	*cadquery.ParallelDirSelector*
-X	Edges aligned in neg x direction	*cadquery.DirectionSelector*
#Z	Edges perpendicular to z direction	*cadquery.PerpendicularDirSelector*
%Line	Edges of type line	*cadquery.TypeSelector*
>Y	Edges farthest in the positive y dir	*cadquery.DirectionMinMaxSelector*
<Y	Edges farthest in the negative y dir	*cadquery.DirectionMinMaxSelector*
>Y[1]	2nd closest **parallel** edge in the positive y dir	cadquery.DirectionNthSelector
<Y[-2]	2nd farthest **parallel** edge in the negative y dir	cadquery.DirectionNthSelector
>>Y[-2]	2nd farthest edge in the y dir	cadquery.CenterNthSelector
<<Y[0]	1st closest edge in the y dir	cadquery.CenterNthSelector

3.11.4 Filtering Vertices

Only a few of the filter types apply to vertices. The location of the vertex is the subject of the filter.

Selector	Selects	Selector Class
>Y	Vertices farthest in the positive y dir	*cadquery.DirectionMinMaxSelector*
<Y	Vertices farthest in the negative y dir	*cadquery.DirectionMinMaxSelector*
>>Y[-2]	2nd farthest vertex in the y dir	cadquery.CenterNthSelector
<<Y[0]	1st closest vertex in the y dir	cadquery.CenterNthSelector

3.11.5 User-defined Directions

It is possible to use user defined vectors as a basis for the selectors. For example:

```
result = cq.Workplane("XY").box(10, 10, 10)

# chamfer only one edge
result = result.edges(">(-1, 1, 0)").chamfer(1)
```

3.11.6 Topological Selectors

Is is also possible to use topological relations to select objects. Currently the following methods are supported:

- *cadquery.Workplane.ancestors()*
- *cadquery.Workplane.siblings()*

Ancestors allows to select all objects containing currently selected object.

```
result = cq.Workplane("XY").box(10, 10, 10).faces(">Z").edges("<Y")

result = result.ancestors("Face")
```

Siblings allows to select all objects of the same type as selection that are connected via the specfied kind of elements.

```
result = cq.Workplane("XY").box(10, 10, 10).faces(">Z")

result = result.siblings("Edge")
```

3.11.7 Using selectors with Shape and Sketch objects

It is possible to use selectors with *cadquery.Shape* and *cadquery.Sketch* objects. This includes chaining and combining.

```
box = cq.Solid.makeBox(1,2,3)

# select top and bottom wires
result = box.faces(">Z or <Z").wires()
```

3.12 CadQuery Class Summary

This page documents all of the methods and functions of the CadQuery classes, organized alphabetically.

See also:

For a listing organized by functional area, see the *API Reference*

3.12.1 Core Classes

Sketch(parent, locs)	2D sketch.
Workplane(, obj=None))	Defines a coordinate system in space, in which 2D coordinates can be used.
Assembly([obj, loc, name, color, metadata])	Nested assembly of Workplane and Shape objects defining their relative positions.
Constraint	alias of `ConstraintSpec`

3.12.2 Topological Classes

Shape(obj)	Represents a shape in the system.
Vertex(obj[, forConstruction])	A Single Point in Space
Edge(obj)	A trimmed curve that represents the border of a face
cadquery.occ_impl.shapes.Mixin1D()	
Wire(obj)	A series of connected, ordered Edges, that typically bounds a Face
Face(obj)	a bounded surface that represents part of the boundary of a solid
Shell(obj)	the outer boundary of a surface
cadquery.occ_impl.shapes.Mixin3D()	
Solid(obj)	a single solid
Compound(obj)	a collection of disconnected solids

3.12.3 Geometry Classes

Vector()	Create a 3-dimensional vector
Matrix()	A 3d , 4x4 transformation matrix.
Plane(origin[, xDir, normal])	A 2D coordinate system in space
Location()	Location in 3D space.

3.12.4 Selector Classes

Selector()	Filters a list of objects.
NearestToPointSelector(pnt)	Selects object nearest the provided point.
BoxSelector(point0, point1[, boundingbox])	Selects objects inside the 3D box defined by 2 points.
BaseDirSelector(vector[, tolerance])	A selector that handles selection on the basis of a single direction vector.
ParallelDirSelector(vector[, tolerance])	Selects objects parallel with the provided direction.
DirectionSelector(vector[, tolerance])	Selects objects aligned with the provided direction.
PerpendicularDirSelector(vector[, tolerance])	Selects objects perpendicular with the provided direction.
TypeSelector(typeString)	Selects objects having the prescribed geometry type.
RadiusNthSelector(n[, directionMax, tolerance])	Select the object with the Nth radius.
CenterNthSelector(vector, n[, directionMax, ...])	Sorts objects into a list with order determined by the distance of their center projected onto the specified direction.
DirectionMinMaxSelector(vector[, ...])	Selects objects closest or farthest in the specified direction.
DirectionNthSelector(vector, n[, ...])	Filters for objects parallel (or normal) to the specified direction then returns the Nth one.
LengthNthSelector(n[, directionMax, tolerance])	Select the object(s) with the Nth length
AreaNthSelector(n[, directionMax, tolerance])	Selects the object(s) with Nth area
BinarySelector(left, right)	Base class for selectors that operates with two other selectors.
AndSelector(left, right)	Intersection selector.
SumSelector(left, right)	Union selector.
SubtractSelector(left, right)	Difference selector.
InverseSelector(selector)	Inverts the selection of given selector.
StringSyntaxSelector(selectorString)	Filter lists objects using a simple string syntax.

3.12.5 Class Details

class cadquery.**Assembly**(*obj: Optional[Union[*Shape*,* Workplane*]] = None*, *loc: Optional[*Location*] = None*, *name: Optional[str] = None*, *color: Optional[*Color*] = None*, *metadata: Optional[Dict[str, Any]] = None*)

Bases: `object`

Nested assembly of Workplane and Shape objects defining their relative positions.

Parameters

- **obj** (*Optional[Union[*Shape*,* Workplane*]]*) –
- **loc** (Location) –
- **name** (*str*) –
- **color** (*Optional[*Color*]*) –
- **metadata** (*Dict[str, Any]*) –

__init__(*obj: Optional[Union[*Shape*,* Workplane*]] = None*, *loc: Optional[*Location*] = None*, *name: Optional[str] = None*, *color: Optional[*Color*] = None*, *metadata: Optional[Dict[str, Any]] = None*)

construct an assembly

Parameters

- **obj** (*Optional[Union[Shape, Workplane]]*) – root object of the assembly (default: None)
- **loc** (*Optional[Location]*) – location of the root object (default: None, interpreted as identity transformation)
- **name** (*Optional[str]*) – unique name of the root object (default: None, resulting in an UUID being generated)
- **color** (*Optional[Color]*) – color of the added object (default: None)
- **metadata** (*Optional[Dict[str, Any]]*) – a store for user-defined metadata (default: None)

Returns
An Assembly object.

To create an empty assembly use:

```
assy = Assembly(None)
```

To create one constraint a root object:

```
b = Workplane().box(1, 1, 1)
assy = Assembly(b, Location(Vector(0, 0, 1)), name="root")
```

__iter__(*loc: Optional[Location] = None*, *name: Optional[str] = None*, *color: Optional[Color] = None*) → Iterator[Tuple[*Shape*, str, *Location*, Optional[*Color*]]]

Assembly iterator yielding shapes, names, locations and colors.

Parameters

- **loc** (*Optional[Location]*) –
- **name** (*Optional[str]*) –
- **color** (*Optional[Color]*) –

Return type
Iterator[*Tuple*[Shape, str, Location, *Optional*[Color]]]

__weakref__

list of weak references to the object (if defined)

add(*obj: Assembly*, *loc: Optional[Location] = None*, *name: Optional[str] = None*, *color: Optional[Color] = None*) → *Assembly*

add(*obj: Optional[Union[Shape, Workplane]]*, *loc: Optional[Location] = None*, *name: Optional[str] = None*, *color: Optional[Color] = None*, *metadata: Optional[Dict[str, Any]] = None*) → *Assembly*

Add a subassembly to the current assembly.

constrain(*q1: str*, *q2: str*, *kind: Literal['Plane', 'Point', 'Axis', 'PointInPlane', 'Fixed', 'FixedPoint', 'FixedAxis', 'PointOnLine', 'FixedRotation']*, *param: Any = None*) → *Assembly*

constrain(*q1: str*, *kind: Literal['Plane', 'Point', 'Axis', 'PointInPlane', 'Fixed', 'FixedPoint', 'FixedAxis', 'PointOnLine', 'FixedRotation']*, *param: Any = None*) → *Assembly*

constrain(*id1: str*, *s1: Shape*, *id2: str*, *s2: Shape*, *kind: Literal['Plane', 'Point', 'Axis', 'PointInPlane', 'Fixed', 'FixedPoint', 'FixedAxis', 'PointOnLine', 'FixedRotation']*, *param: Any = None*) → *Assembly*

constrain(*id1: str*, *s1:* Shape, *kind: Literal['Plane', 'Point', 'Axis', 'PointInPlane', 'Fixed', 'FixedPoint', 'FixedAxis', 'PointOnLine', 'FixedRotation']*, *param: Any = None*) → *Assembly*

Define a new constraint.

save(*path: str*, *exportType: Optional[Literal['STEP', 'XML', 'GLTF', 'VTKJS', 'VRML', 'STL']] = None*, *mode: Literal['default', 'fused'] = 'default'*, *tolerance: float = 0.1*, *angularTolerance: float = 0.1*, ***kwargs*) → *Assembly*

Save assembly to a file.

Parameters

- **path** (*str*) – Path and filename for writing.
- **exportType** (*Optional[Literal['STEP', 'XML', 'GLTF', 'VTKJS', 'VRML', 'STL']]*) – export format (default: None, results in format being inferred form the path)
- **mode** (*Literal['default', 'fused']*) – STEP only - See *exportAssembly()*.
- **tolerance** (*float*) – the deflection tolerance, in model units. Only used for glTF, VRML. Default 0.1.
- **angularTolerance** (*float*) – the angular tolerance, in radians. Only used for glTF, VRML. Default 0.1.
- ****kwargs** – Additional keyword arguments. Only used for STEP, glTF and STL. See *exportAssembly()*.
- **ascii** (*bool*) – STL only - Sets whether or not STL export should be text or binary

Return type
Assembly

property shapes: List[*Shape*]

List of Shape objects in the .obj field

solve(*verbosity: int = 0*) → *Assembly*

Solve the constraints.

Parameters
verbosity (*int*) –

Return type
Assembly

toCompound() → *Compound*

Returns a Compound made from this Assembly (including all children) with the current Locations applied. Usually this method would only be used after solving.

Return type
Compound

traverse() → Iterator[Tuple[str, *Assembly*]]

Yield (name, child) pairs in a bottom-up manner

Return type
Iterator[*Tuple*[str, Assembly]]

class cadquery.**BoundBox**(*bb: Bnd_Box*)

Bases: object

A BoundingBox for an object or set of objects. Wraps the OCP one

Parameters
bb (*Bnd_Box*) –

__init__(*bb: Bnd_Box*) → None

Parameters
bb (*Bnd_Box*) –

Return type
None

__weakref__

list of weak references to the object (if defined)

add(*obj: Union[Tuple[float, float, float], Vector, BoundBox], tol: Optional[float] = None*) → *BoundBox*

Returns a modified (expanded) bounding box

obj can be one of several things:

1. a 3-tuple corresponding to x,y, and z amounts to add
2. a vector, containing the x,y,z values to add
3. another bounding box, where a new box will be created that encloses both.

This bounding box is not changed.

Parameters

- **obj** (*Union[Tuple[float, float, float], Vector, BoundBox]*) –
- **tol** (*Optional[float]*) –

Return type
BoundBox

enlarge(*tol: float*) → *BoundBox*

Returns a modified (expanded) bounding box, expanded in all directions by the tolerance value.

This means that the minimum values of its X, Y and Z intervals of the bounding box are reduced by the absolute value of tol, while the maximum values are increased by the same amount.

Parameters
tol (*float*) –

Return type
BoundBox

static findOutsideBox2D(*bb1:* BoundBox, *bb2:* BoundBox) → Optional[*BoundBox*]

Compares bounding boxes

Compares bounding boxes. Returns none if neither is inside the other. Returns the outer one if either is outside the other.

BoundBox.isInside works in 3d, but this is a 2d bounding box, so it doesn't work correctly plus, there was all kinds of rounding error in the built-in implementation i do not understand.

Parameters

- **bb1** (BoundBox) –
- **bb2** (BoundBox) –

Return type
Optional[BoundBox]

isInside(*b2:* BoundBox) → bool

Is the provided bounding box inside this one?

Parameters
b2 (BoundBox) –

Return type
bool

cadquery.**CQ**

alias of *Workplane*

class cadquery.**Color**(*name: str*)

class cadquery.**Color**(*r: float*, *g: float*, *b: float*, *a: float = 0*)

class cadquery.**Color**

Bases: object

Wrapper for the OCCT color object Quantity_ColorRGBA.

__init__(*name: str*)

__init__(*r: float*, *g: float*, *b: float*, *a: float = 0*)

__init__()

__weakref__

list of weak references to the object (if defined)

toTuple() → Tuple[float, float, float, float]

Convert Color to RGB tuple.

Return type
Tuple[float, float, float, float]

class cadquery.**Compound**(*obj: TopoDS_Shape*)

Bases: *Shape*, *Mixin3D*

a collection of disconnected solids

Parameters
obj (*TopoDS_Shape*) –

__bool__() → bool

Check if empty.

Return type
bool

ancestors(*shape:* Shape, *kind: Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) → *Compound*

Iterate over ancestors, i.e. shapes of same kind within shape that contain elements of self.

Parameters

- **shape** (Shape) –
- **kind** (*Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) –

Return type
Compound

cut(**toCut: Shape, tol: Optional[float] = None*) → *Compound*

Remove the positional arguments from this Shape.

Parameters

- **tol** (*Optional[float]*) – Fuzzy mode tolerance
- **toCut** (Shape) –

Return type
Compound

fuse(**toFuse: Shape, glue: bool = False, tol: Optional[float] = None*) → *Compound*

Fuse shapes together

Parameters

- **toFuse** (Shape) –
- **glue** (*bool*) –
- **tol** (*Optional[float]*) –

Return type
Compound

intersect(**toIntersect: Shape, tol: Optional[float] = None*) → *Compound*

Intersection of the positional arguments and this Shape.

Parameters

- **tol** (*Optional[float]*) – Fuzzy mode tolerance
- **toIntersect** (Shape) –

Return type
Compound

classmethod makeCompound(*listOfShapes: Iterable[Shape]*) → *Compound*

Create a compound out of a list of shapes

Parameters
listOfShapes (*Iterable[Shape]*) –

Return type
Compound

classmethod makeText(*text: str, size: float, height: float, font: str = 'Arial', fontPath: Optional[str] = None, kind: Literal['regular', 'bold', 'italic'] = 'regular', halign: Literal['center', 'left', 'right'] = 'center', valign: Literal['center', 'top', 'bottom'] = 'center', position: Plane = Plane(origin=(0.0, 0.0, 0.0), xDir=(1.0, 0.0, 0.0), normal=(0.0, 0.0, 1.0))*) → *Shape*

Create a 3D text

Parameters

- **text** (*str*) –
- **size** (*float*) –
- **height** (*float*) –
- **font** (*str*) –
- **fontPath** (*Optional[str]*) –

- **kind** (*Literal['regular', 'bold', 'italic']*) –
- **halign** (*Literal['center', 'left', 'right']*) –
- **valign** (*Literal['center', 'top', 'bottom']*) –
- **position** (Plane) –

Return type
Shape

remove(*shape:* Shape)

Remove the specified shape.

Parameters
shape (Shape) –

siblings(*shape:* Shape, *kind: Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*, *level: int = 1*) → *Compound*

Iterate over siblings, i.e. shapes within shape that share subshapes of kind with the elements of self.

Parameters

- **shape** (Shape) –
- **kind** (*Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) –
- **level** (*int*) –

Return type
Compound

cadquery.**Constraint**

alias of ConstraintSpec

class cadquery.**DirectionMinMaxSelector**(*vector:* Vector, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Bases: *CenterNthSelector*

Selects objects closest or farthest in the specified direction.

Applicability:
All object types. for a vertex, its point is used. for all other kinds of objects, the center of mass of the object is used.

You can use the string shortcuts >(X|Y|Z) or <(X|Y|Z) if you want to select based on a cardinal direction.

For example this:

```
CQ(aCube).faces(DirectionMinMaxSelector((0, 0, 1), True))
```

Means to select the face having the center of mass farthest in the positive z direction, and is the same as:

```
CQ(aCube).faces(">Z")
```

Parameters

- **vector** (Vector) –
- **directionMax** (*bool*) –
- **tolerance** (*float*) –

__init__(*vector:* Vector, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Parameters

- **vector** (Vector) –
- **directionMax** (*bool*) –
- **tolerance** (*float*) –

class cadquery.**DirectionSelector**(*vector:* Vector, *tolerance: float = 0.0001*)

Bases: *BaseDirSelector*

Selects objects aligned with the provided direction.

Applicability:
Linear Edges Planar Faces

Use the string syntax shortcut +/-(X|Y|Z) if you want to select based on a cardinal direction.

Example:

```
CQ(aCube).faces(DirectionSelector((0, 0, 1)))
```

selects faces with the normal in the z direction, and is equivalent to:

```
CQ(aCube).faces("+Z")
```

Parameters

- **vector** (Vector) –
- **tolerance** (*float*) –

test(*vec:* Vector) → bool

Test a specified vector. Subclasses override to provide other implementations

Parameters
vec (Vector) –

Return type
bool

class cadquery.**Edge**(*obj: TopoDS_Shape*)

Bases: *Shape*, *Mixin1D*

A trimmed curve that represents the border of a face

Parameters
obj (*TopoDS_Shape*) –

arcCenter() → *Vector*

Center of an underlying circle or ellipse geometry.

Return type
Vector

close() → Union[*Edge*, *Wire*]

Close an Edge

Return type
Union[Edge, Wire]

classmethod makeEllipse(*x_radius: float, y_radius: float, pnt: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 1.0), xdir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (1.0, 0.0, 0.0), angle1: float = 360.0, angle2: float = 360.0, sense: ~typing.Literal[-1, 1] = 1*) → *Edge*

Makes an Ellipse centered at the provided point, having normal in the provided direction.

Parameters

- **cls** –
- **x_radius** (*float*) – x radius of the ellipse (along the x-axis of plane the ellipse should lie in)
- **y_radius** (*float*) – y radius of the ellipse (along the y-axis of plane the ellipse should lie in)
- **pnt** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – vector representing the center of the ellipse
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – vector representing the direction of the plane the ellipse should lie in
- **angle1** (*float*) – start angle of arc
- **angle2** (*float*) – end angle of arc (angle2 == angle1 return closed ellipse = default)
- **sense** (*Literal[-1, 1]*) – clockwise (-1) or counter clockwise (1)
- **xdir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –

Returns
an Edge

Return type
Edge

classmethod makeLine(*v1: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], v2: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → *Edge*

Create a line between two points

Parameters

- **v1** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – Vector that represents the first point

- **v2** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – Vector that represents the second point

Returns
A linear edge between the two provided points

Return type
Edge

classmethod makeSpline(*listOfVector: List[Vector]*, *tangents: Optional[Sequence[Vector]] = None*, *periodic: bool = False*, *parameters: Optional[Sequence[float]] = None*, *scale: bool = True*, *tol: float = 1e-06*) → *Edge*

Interpolate a spline through the provided points.

Parameters

- **listOfVector** (*List[Vector]*) – a list of Vectors that represent the points
- **tangents** (*Optional[Sequence[Vector]]*) – tuple of Vectors specifying start and finish tangent
- **periodic** (*bool*) – creation of periodic curves
- **parameters** (*Optional[Sequence[float]]*) – the value of the parameter at each interpolation point. (The interpolated curve is represented as a vector-valued function of a scalar parameter.) If periodic == True, then len(parameters) must be len(intepolation points) + 1, otherwise len(parameters) must be equal to len(interpolation points).
- **scale** (*bool*) – whether to scale the specified tangent vectors before interpolating. Each tangent is scaled, so it's length is equal to the derivative of the Lagrange interpolated curve. I.e., set this to True, if you want to use only the direction of the tangent vectors specified by `tangents`, but not their magnitude.
- **tol** (*float*) – tolerance of the algorithm (consult OCC documentation). Used to check that the specified points are not too close to each other, and that tangent vectors are not too short. (In either case interpolation may fail.)

Returns
an Edge

Return type
Edge

classmethod makeSplineApprox(*listOfVector: List[Vector]*, *tol: float = 0.001*, *smoothing: Optional[Tuple[float, float, float]] = None*, *minDeg: int = 1*, *maxDeg: int = 6*) → *Edge*

Approximate a spline through the provided points.

Parameters

- **listOfVector** (*List[Vector]*) – a list of Vectors that represent the points
- **tol** (*float*) – tolerance of the algorithm (consult OCC documentation).
- **smoothing** (*Optional[Tuple[float, float, float]]*) – optional tuple of 3 weights use for variational smoothing (default: None)
- **minDeg** (*int*) – minimum spline degree. Enforced only when smothing is None (default: 1)
- **maxDeg** (*int*) – maximum spline degree (default: 6)

Returns
an Edge

Return type
Edge

classmethod makeTangentArc(*v1: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *v2: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *v3: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → *Edge*

Makes a tangent arc from point v1, in the direction of v2 and ends at v3.

Parameters

- **cls** –
- **v1** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – start vector
- **v2** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – tangent vector
- **v3** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – end vector

Returns
an edge

Return type
Edge

classmethod makeThreePointArc(*v1: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *v2: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *v3: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → *Edge*

Makes a three point arc through the provided points

Parameters

- **cls** –
- **v1** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – start vector
- **v2** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – middle vector
- **v3** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – end vector

Returns
an edge object through the three points

Return type
Edge

class cadquery.**Face**(*obj: TopoDS_Shape*)

Bases: *Shape*

a bounded surface that represents part of the boundary of a solid

Parameters
obj (*TopoDS_Shape*) –

Center() → *Vector*

Returns
The point of the center of mass of this Shape

Return type
Vector

chamfer2D(*d: float*, *vertices: Iterable[*Vertex*]*) → *Face*

Apply 2D chamfer to a face

Parameters

- **d** (*float*) –
- **vertices** (*Iterable[Vertex]*) –

Return type
Face

fillet2D(*radius: float*, *vertices: Iterable[*Vertex*]*) → *Face*

Apply 2D fillet to a face

Parameters

- **radius** (*float*) –
- **vertices** (*Iterable[Vertex]*) –

Return type
Face

classmethod makeFromWires(*outerWire:* Wire, *innerWires: List[*Wire*] = []*) → *Face*

Makes a planar face from one or more wires

Parameters

- **outerWire** (*Wire*) –
- **innerWires** (*List[Wire]*) –

Return type
Face

classmethod makeNSidedSurface(*edges: ~typing.Iterable[~typing.Union[~cadquery.occ_impl.shapes.Edge, ~cadquery.occ_impl.shapes.Wire]], constraints: ~typing.Iterable[~typing.Union[~cadquery.occ_impl.shapes.Edge, ~cadquery.occ_impl.shapes.Wire, ~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]], ~OCP.gp.gp_Pnt]], continuity: ~OCP.GeomAbs.GeomAbs_Shape = <GeomAbs_Shape.GeomAbs_C0: 0>, degree: int = 3, nbPtsOnCur: int = 15, nbIter: int = 2, anisotropy: bool = False, tol2d: float = 1e-05, tol3d: float = 0.0001, tolAng: float = 0.01, tolCurv: float = 0.1, maxDeg: int = 8, maxSegments: int = 9*) → *Face*

Returns a surface enclosed by a closed polygon defined by 'edges' and 'constraints'.

Parameters

- **edges** (*list of edges or wires*) – edges
- **constraints** (*list of points or edges*) – constraints
- **continuity** (*GeomAbs_Shape*) – OCC.Core.GeomAbs continuity condition
- **degree** (*int*) – >=2
- **nbPtsOnCur** (*int*) – number of points on curve >= 15
- **nbIter** (*int*) – number of iterations >= 2
- **anisotropy** (*bool*) – bool Anisotropy
- **tol2d** (*float*) – 2D tolerance >0
- **tol3d** (*float*) – 3D tolerance >0
- **tolAng** (*float*) – angular tolerance
- **tolCurv** (*float*) – tolerance for curvature >0
- **maxDeg** (*int*) – highest polynomial degree >= 2
- **maxSegments** (*int*) – greatest number of segments >= 2

Return type
Face

classmethod makeRuledSurface(*edgeOrWire1:* Edge, *edgeOrWire2:* Edge) → *Face*

classmethod makeRuledSurface(*edgeOrWire1:* Wire, *edgeOrWire2:* Wire) → *Face*

makeRuledSurface(Edge|Wire,Edge|Wire) – Make a ruled surface Create a ruled surface out of two edges or wires. If wires are used then these must have the same number of edges

classmethod makeSplineApprox(*points: List[List[*Vector*]], tol: float = 0.01, smoothing: Optional[Tuple[float, float, float]] = None, minDeg: int = 1, maxDeg: int = 3*) → *Face*

Approximate a spline surface through the provided points.

Parameters

- **points** (*List[List[*Vector*]]*) – a 2D list of Vectors that represent the points
- **tol** (*float*) – tolerance of the algorithm (consult OCC documentation).

- **smoothing** (*Optional[Tuple[float, float, float]]*) – optional tuple of 3 weights use for variational smoothing (default: None)
- **minDeg** (*int*) – minimum spline degree. Enforced only when smothing is None (default: 1)
- **maxDeg** (*int*) – maximum spline degree (default: 6)

Return type
Face

normalAt(*locationVector: Optional[*Vector*] = None*) → *Vector*

Computes the normal vector at the desired location on the face.

Returns
a vector representing the direction

Parameters
locationVector (*a vector that lies on the surface.*) – the location to compute the normal at. If none, the center of the face is used.

Return type
Vector

thicken(*thickness: float*) → *Solid*

Return a thickened face

Parameters
thickness (*float*) –

Return type
Solid

toArcs(*tolerance: float = 0.001*) → *Face*

Approximate planar face with arcs and straight line segments.

Parameters
tolerance (*float*) – Approximation tolerance.

Return type
Face

toPln() → gp_Pln

Convert this face to a gp_Pln.

Note the Location of the resulting plane may not equal the center of this face, however the resulting plane will still contain the center of this face.

Return type
gp_Pln

class cadquery.**Location**

class cadquery.**Location**(*t: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*)

class cadquery.**Location**(*t:* Plane)

class cadquery.**Location**(*t:* Plane*, v: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*)

class cadquery.**Location**(*t: TopLoc_Location*)

class cadquery.**Location**(*t: gp_Trsf*)

class cadquery.**Location**(*t: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], ax: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], angle: float*)

Bases: object

Location in 3D space. Depending on usage can be absolute or relative.

This class wraps the TopLoc_Location class from OCCT. It can be used to move Shape objects in both relative and absolute manner. It is the preferred type to locate objects in CQ.

__init__() → None

__init__(*t: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → None

__init__(*t:* Plane) → None

__init__(*t:* Plane, *v: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → None

__init__(*t: TopLoc_Location*) → None

__init__(*t: gp_Trsf*) → None

__init__(*t: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], ax: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], angle: float*) → None

__weakref__

list of weak references to the object (if defined)

toTuple() → Tuple[Tuple[float, float, float], Tuple[float, float, float]]

Convert the location to a translation, rotation tuple.

Return type
Tuple[*Tuple*[float, float, float], *Tuple*[float, float, float]]

class cadquery.**Matrix**

class cadquery.**Matrix**(*matrix: Union[gp_GTrsf, gp_Trsf]*)

class cadquery.**Matrix**(*matrix: Sequence[Sequence[float]]*)

Bases: object

A 3d , 4x4 transformation matrix.

Used to move geometry in space.

The provided "matrix" parameter may be None, a gp_GTrsf, or a nested list of values.

If given a nested list, it is expected to be of the form:

[[m11, m12, m13, m14],
[m21, m22, m23, m24], [m31, m32, m33, m34]]

A fourth row may be given, but it is expected to be: [0.0, 0.0, 0.0, 1.0] since this is a transform matrix.

__getitem__(*rc: Tuple[int, int]*) → float

Provide Matrix[r, c] syntax for accessing individual values. The row and column parameters start at zero, which is consistent with most python libraries, but is counter to gp_GTrsf(), which is 1-indexed.

Parameters
rc (*Tuple[int, int]*) –

Return type
float

__init__() → None

__init__(*matrix: Union[gp_GTrsf, gp_Trsf]*) → None

__init__(*matrix: Sequence[Sequence[float]]*) → None

__repr__() → str

Generate a valid python expression representing this Matrix

Return type
str

__weakref__

list of weak references to the object (if defined)

transposed_list() → Sequence[float]

Needed by the cqparts gltf exporter

Return type
Sequence[float]

class cadquery.**NearestToPointSelector**(*pnt*)

Bases: *Selector*

Selects object nearest the provided point.

If the object is a vertex or point, the distance is used. For other kinds of shapes, the center of mass is used to to compute which is closest.

Applicability: All Types of Shapes

Example:

```
CQ(aCube).vertices(NearestToPointSelector((0, 1, 0)))
```

returns the vertex of the unit cube closest to the point x=0,y=1,z=0

__init__(*pnt*)

filter(*objectList: Sequence[Shape]*)

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

class cadquery.**ParallelDirSelector**(*vector:* Vector, *tolerance: float = 0.0001*)

Bases: *BaseDirSelector*

Selects objects parallel with the provided direction.

Applicability:
Linear Edges Planar Faces

Use the string syntax shortcut |(X|Y|Z) if you want to select based on a cardinal direction.

Example:

```
CQ(aCube).faces(ParallelDirSelector((0, 0, 1)))
```

selects faces with the normal parallel to the z direction, and is equivalent to:

```
CQ(aCube).faces("|Z")
```

Parameters

- **vector** (Vector) –
- **tolerance** (*float*) –

test(*vec:* Vector) → bool

Test a specified vector. Subclasses override to provide other implementations

Parameters
vec (Vector) –

Return type
bool

class cadquery.**PerpendicularDirSelector**(*vector:* Vector, *tolerance: float = 0.0001*)

Bases: *BaseDirSelector*

Selects objects perpendicular with the provided direction.

Applicability:
Linear Edges Planar Faces

Use the string syntax shortcut #(X|Y|Z) if you want to select based on a cardinal direction.

Example:

```
CQ(aCube).faces(PerpendicularDirSelector((0, 0, 1)))
```

selects faces with the normal perpendicular to the z direction, and is equivalent to:

```
CQ(aCube).faces("#Z")
```

Parameters

- **vector** (Vector) –
- **tolerance** (*float*) –

test(*vec:* Vector) → bool

Test a specified vector. Subclasses override to provide other implementations

Parameters
vec (Vector) –

Return type
bool

class cadquery.**Plane**(*origin: Union[Tuple[float, float, float], Vector], xDir: Optional[Union[Tuple[float, float, float], Vector]] = None, normal: Union[Tuple[float, float, float], Vector] = (0, 0, 1)*)

Bases: object

A 2D coordinate system in space

A 2D coordinate system in space, with the x-y axes on the plane, and a particular point as the origin.

A plane allows the use of 2D coordinates, which are later converted to global, 3d coordinates when the operations are complete.

Frequently, it is not necessary to create work planes, as they can be created automatically from faces.

Parameters

- **origin** (*Union[Tuple[float, float, float], Vector]*) –
- **xDir** (*Vector*) –
- **normal** (*Union[Tuple[float, float, float], Vector]*) –

__eq__(*other*)

Return self==value.

__hash__ = None

__init__(*origin: Union[Tuple[float, float, float], Vector], xDir: Optional[Union[Tuple[float, float, float], Vector]] = None, normal: Union[Tuple[float, float, float], Vector] = (0, 0, 1)*)

Create a Plane with an arbitrary orientation

Parameters

- **origin** (*Union[Tuple[float, float, float], Vector]*) – the origin in global coordinates
- **xDir** (*Optional[Union[Tuple[float, float, float], Vector]]*) – an optional vector representing the xDirection.
- **normal** (*Union[Tuple[float, float, float], Vector]*) – the normal direction for the plane

Raises

ValueError – if the specified xDir is not orthogonal to the provided normal

__ne__(*other*)

Return self!=value.

__repr__()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

classmethod named(*stdName: str, origin=(0, 0, 0)*) → *Plane*

Create a predefined Plane based on the conventional names.

Parameters

- **stdName** (*string*) – one of (XY|YZ|ZX|XZ|YX|ZY|front|back|left|right|top|bottom)
- **origin** (*3-tuple of the origin of the new plane, in global coordinates.*) – the desired origin, specified in global coordinates

Return type

Plane

Available named planes are as follows. Direction references refer to the global directions.

Name	xDir	yDir	zDir
XY	+x	+y	+z
YZ	+y	+z	+x
ZX	+z	+x	+y
XZ	+x	+z	-y
YX	+y	+x	-z
ZY	+z	+y	-x
front	+x	+y	+z
back	-x	+y	-z
left	+z	+y	-x
right	-z	+y	+x
top	+x	-z	+y
bottom	+x	+z	-y

rotated(*rotate=(0, 0, 0)*)

Returns a copy of this plane, rotated about the specified axes

Since the z axis is always normal the plane, rotating around Z will always produce a plane that is parallel to this one.

The origin of the workplane is unaffected by the rotation.

Rotations are done in order x, y, z. If you need a different order, manually chain together multiple rotate() commands.

Parameters
rotate – Vector [xDegrees, yDegrees, zDegrees]

Returns
a copy of this plane rotated as requested.

setOrigin2d(*x*, *y*)

Set a new origin in the plane itself

Set a new origin in the plane itself. The plane's orientation and xDrection are unaffected.

Parameters

- **x** (*float*) – offset in the x direction
- **y** (*float*) – offset in the y direction

Returns
void

The new coordinates are specified in terms of the current 2D system. As an example:

p = Plane.XY() p.setOrigin2d(2, 2) p.setOrigin2d(2, 2)

results in a plane with its origin at (x, y) = (4, 4) in global coordinates. Both operations were relative to local coordinates of the plane.

toLocalCoords(*obj*)

Project the provided coordinates onto this plane

Parameters
obj – an object or vector to convert

Returns
an object of the same type, but converted to local coordinates

Most of the time, the z-coordinate returned will be zero, because most operations based on a plane are all 2D. Occasionally, though, 3D points outside of the current plane are transformed. One such example is *Workplane.box()*, where 3D corners of a box are transformed to orient the box in space correctly.

toWorldCoords(*tuplePoint*) → *Vector*

Convert a point in local coordinates to global coordinates

Parameters
tuplePoint (*a 2 or three tuple of float. The third value is taken to be zero if not supplied.*) – point in local coordinates to convert.

Returns
a Vector in global coordinates

Return type
Vector

class cadquery.**Selector**

Bases: object

Filters a list of objects.

Filters must provide a single method that filters objects.

__weakref__

list of weak references to the object (if defined)

filter(*objectList: Sequence[Shape]*) → List[Shape]

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

Return type
List[*Shape*]

class cadquery.**Shape**(*obj: TopoDS_Shape*)

Bases: object

Represents a shape in the system. Wraps TopoDS_Shape.

Parameters
obj (*TopoDS_Shape*) –

Area() → float

Returns
The surface area of all faces in this Shape

Return type
float

BoundingBox(*tolerance: Optional[float] = None*) → *BoundBox*

Create a bounding box for this Shape.

Parameters
tolerance (*Optional[float]*) – Tolerance value passed to *BoundBox*

Returns
A *BoundBox* object for this Shape

Return type
BoundBox

Center() → *Vector*

Returns
The point of the center of mass of this Shape

Return type
Vector

CenterOfBoundBox(*tolerance: Optional[float] = None*) → *Vector*

Parameters
tolerance (*Optional[float]*) – Tolerance passed to the *BoundingBox()* method

Returns
Center of the bounding box of this shape

Return type
Vector

Closed() → bool

Returns
The closedness flag

Return type
bool

static CombinedCenter(*objects: Iterable[Shape]*) → *Vector*

Calculates the center of mass of multiple objects.

Parameters
objects (*Iterable[Shape]*) – A list of objects with mass

Return type
Vector

static CombinedCenterOfBoundBox(*objects: List[Shape]*) → *Vector*

Calculates the center of a bounding box of multiple objects.

Parameters
objects (*List[Shape]*) – A list of objects

Return type
Vector

CompSolids() → List[CompSolid]

Returns
All the compsolids in this Shape

Return type
List[*CompSolid*]

Compounds() → List[*Compound*]

Returns
All the compounds in this Shape

Return type
List[Compound]

Edges() → List[*Edge*]

Returns
All the edges in this Shape

Return type
List[Edge]

Faces() → List[*Face*]

Returns
All the faces in this Shape

Return type
List[Face]

Shells() → List[*Shell*]

Returns
All the shells in this Shape

Return type
List[Shell]

Solids() → List[*Solid*]

Returns
All the solids in this Shape

Return type
List[Solid]

Vertices() → List[*Vertex*]

Returns
All the vertices in this Shape

Return type
List[Vertex]

Volume() → float

Returns
The volume of this Shape

Return type
float

Wires() → List[*Wire*]

Returns
All the wires in this Shape

Return type
List[Wire]

__eq__(*other*) → bool
Return self==value.

Return type
bool

__hash__() → int

Return hash(self).

Return type
int

__init__(*obj: TopoDS_Shape*)

Parameters
obj (*TopoDS_Shape*) –

__iter__() → Iterator[*Shape*]

Iterate over subshapes.

Return type
Iterator[Shape]

__weakref__

list of weak references to the object (if defined)

ancestors(*shape:* Shape, *kind: Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) → *Compound*

Iterate over ancestors, i.e. shapes of same kind within shape that contain self.

Parameters

- **shape** (Shape) –
- **kind** (*Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) –

Return type
Compound

classmethod cast(*obj: TopoDS_Shape*, *forConstruction: bool = False*) → *Shape*

Returns the right type of wrapper, given a OCCT object

Parameters

- **obj** (*TopoDS_Shape*) –
- **forConstruction** (*bool*) –

Return type
Shape

static centerOfMass(*obj:* Shape) → *Vector*

Calculates the center of 'mass' of an object.

Parameters
obj (Shape) – Compute the center of mass of this object

Return type
Vector

clean() → T

Experimental clean using ShapeUpgrade

Parameters
self (*T*) –

Return type
T

static computeMass(*obj:* Shape) → float

Calculates the 'mass' of an object.

Parameters
obj (Shape) – Compute the mass of this object

Return type
float

copy(*mesh: bool = False*) → T

Creates a new object that is a copy of this object.

Parameters

- **self** (*T*) –
- **mesh** (*bool*) – should I copy the triangulation too (default: False)

Returns
a copy of the object

Return type
T

cut(**toCut:* Shape, *tol: Optional[float] = None*) → *Shape*

Remove the positional arguments from this Shape.

Parameters

- **tol** (*Optional[float]*) – Fuzzy mode tolerance
- **toCut** (Shape) –

Return type
Shape

distance(*other:* Shape) → float

Minimal distance between two shapes

Parameters
other (Shape) –

Return type
float

distances(**others:* Shape) → Iterator[float]

Minimal distances to between self and other shapes

Parameters
others (Shape) –

Return type
Iterator[float]

edges(*selector: Optional[Union[*Selector*, str]] = None*) → *Shape*

Select edges.

Parameters
selector (*Optional[Union[Selector, str]]*) –

Return type
Shape

exportBrep(*f: Union[str, BytesIO]*) → bool

Export this shape to a BREP file

Parameters
f (*Union[str, BytesIO]*) –

Return type
bool

exportStep(*fileName: str*, ***kwargs*) → IFSelect_ReturnStatus

Export this shape to a STEP file.

kwargs is used to provide optional keyword arguments to configure the exporter.

Parameters

- **fileName** (*str*) – Path and filename for writing.
- **write_pcurves** (*bool*) – Enable or disable writing parametric curves to the STEP file. Default True.

  If False, writes STEP file without pcurves. This decreases the size of the resulting STEP file.
- **precision_mode** (*int*) – Controls the uncertainty value for STEP entities. Specify -1, 0, or 1. Default 0. See OCCT documentation.

Return type
IFSelect_ReturnStatus

exportStl(*fileName: str*, *tolerance: float = 0.001*, *angularTolerance: float = 0.1*, *ascii: bool = False*, *relative: bool = True*, *parallel: bool = True*) → bool

Exports a shape to a specified STL file.

Parameters

- **fileName** (*str*) – The path and file name to write the STL output to.
- **tolerance** (*float*) – A linear deflection setting which limits the distance between a curve and its tessellation. Setting this value too low will result in large meshes that can consume computing resources. Setting the value too high can result in meshes with a level of detail that is too low. Default is 1e-3, which is a good starting point for a range of cases.
- **angularTolerance** (*float*) – Angular deflection setting which limits the angle between subsequent segments in a polyline. Default is 0.1.
- **ascii** (*bool*) – Export the file as ASCII (True) or binary (False) STL format. Default is binary.
- **relative** (*bool*) – If True, tolerance will be scaled by the size of the edge being meshed. Default is True. Setting this value to True may cause large features to become faceted, or small features dense.
- **parallel** (*bool*) – If True, OCCT will use parallel processing to mesh the shape. Default is True.

Return type
bool

faces(*selector: Optional[Union[*Selector*, str]] = None*) → *Shape*

Select faces.

Parameters
selector (*Optional[Union[Selector, str]]*) –

Return type
Shape

facesIntersectedByLine(*point: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], axis: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], tol: float = 0.0001, direction: Optional[Literal['AlongAxis', 'Opposite']] = None*)

Computes the intersections between the provided line and the faces of this Shape

Parameters

- **point** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – Base point for defining a line
- **axis** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – Axis on which the line rests
- **tol** (*float*) – Intersection tolerance
- **direction** (*Optional[Literal['AlongAxis', 'Opposite']]*) – Valid values: "AlongAxis", "Opposite"; If specified, will ignore all faces that are not in the specified direction including the face where the point lies if it is the case

Returns
A list of intersected faces sorted by distance from point

fix() → T

Try to fix shape if not valid

Parameters
self (*T*) –

Return type
T

fuse(**toFuse: Shape, glue: bool = False, tol: Optional[float] = None*) → *Shape*

Fuse the positional arguments with this Shape.

Parameters

- **glue** (*bool*) – Sets the glue option for the algorithm, which allows increasing performance of the intersection of the input shapes
- **tol** (*Optional[float]*) – Fuzzy mode tolerance
- **toFuse** (Shape) –

Return type
Shape

geomType() → Literal['Vertex', 'Wire', 'Shell', 'Solid', 'Compound', 'PLANE', 'CYLINDER', 'CONE', 'SPHERE', 'TORUS', 'BEZIER', 'BSPLINE', 'REVOLUTION', 'EXTRUSION', 'OFFSET', 'OTHER', 'LINE', 'CIRCLE', 'ELLIPSE', 'HYPERBOLA', 'PARABOLA']

Gets the underlying geometry type.

Implementations can return any values desired, but the values the user uses in type filters should correspond to these.

As an example, if a user does:

```
CQ(object).faces("%mytype")
```

The expectation is that the geomType attribute will return 'mytype'

The return values depend on the type of the shape:

Vertex: always 'Vertex'
Edge: LINE, CIRCLE, ELLIPSE, HYPERBOLA, PARABOLA, BEZIER, BSPLINE, OFFSET, OTHER
Face: PLANE, CYLINDER, CONE, SPHERE, TORUS, BEZIER, BSPLINE, REVOLUTION, EXTRUSION, OFFSET, OTHER
Solid: 'Solid'
Shell: 'Shell'
Compound: 'Compound'
Wire: 'Wire'

Returns
A string according to the geometry type

Return type
Literal['Vertex', 'Wire', 'Shell', 'Solid', 'Compound', 'PLANE', 'CYLINDER', 'CONE', 'SPHERE', 'TORUS', 'BEZIER', 'BSPLINE', 'REVOLUTION', 'EXTRUSION', 'OFFSET', 'OTHER', 'LINE', 'CIRCLE', 'ELLIPSE', 'HYPERBOLA', 'PARABOLA']

hashCode() → int

Returns a hashed value denoting this shape. It is computed from the TShape and the Location. The Orientation is not used.

Return type
int

classmethod importBrep(*f: Union[str, BytesIO]*) → *Shape*

Import shape from a BREP file

Parameters
f (*Union[str, BytesIO]*) –

Return type
Shape

intersect(**toIntersect:* Shape, *tol: Optional[float] = None*) → *Shape*

Intersection of the positional arguments and this Shape.

Parameters

- **tol** (*Optional[float]*) – Fuzzy mode tolerance
- **toIntersect** (Shape) –

Return type
Shape

isEqual(*other:* Shape) → bool

Returns True if two shapes are equal, i.e. if they share the same TShape with the same Locations and Orientations. Also see *isSame()*.

Parameters
other (Shape) –

Return type
bool

isNull() → bool

Returns true if this shape is null. In other words, it references no underlying shape with the potential to be given a location and an orientation.

Return type
bool

isSame(*other:* Shape) → bool

Returns True if other and this shape are same, i.e. if they share the same TShape with the same Locations. Orientations may differ. Also see *isEqual()*

Parameters
other (Shape) –

Return type
bool

isValid() → bool

Returns True if no defect is detected on the shape S or any of its subshapes. See the OCCT docs on BRepCheck_Analyzer::IsValid for a full description of what is checked.

Return type
bool

locate(*loc:* Location) → T

Apply a location in absolute sense to self

Parameters

- **self** (*T*) –
- **loc** (Location) –

Return type
T

located(*loc:* Location) → T

Apply a location in absolute sense to a copy of self

Parameters

- **self** (*T*) –
- **loc** (Location) –

Return type
T

location() → *Location*

Return the current location

Return type
Location

static matrixOfInertia(*obj:* Shape) → List[List[float]]

Calculates the matrix of inertia of an object. :param obj: Compute the matrix of inertia of this object

Parameters
obj (Shape) –

Return type
List[*List*[float]]

mesh(*tolerance: float*, *angularTolerance: float = 0.1*)

Generate triangulation if none exists.

Parameters

- **tolerance** (*float*) –
- **angularTolerance** (*float*) –

mirror(*mirrorPlane: Union[Literal['XY', 'YX', 'XZ', 'ZX', 'YZ', 'ZY'], Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]] = 'XY'*, *basePointVector: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]] = (0, 0, 0)*) → *Shape*

Applies a mirror transform to this Shape. Does not duplicate objects about the plane.

Parameters

- **mirrorPlane** (*Union[Literal['XY', 'YX', 'XZ', 'ZX', 'YZ', 'ZY'], Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – The direction of the plane to mirror about - one of 'XY', 'XZ' or 'YZ'
- **basePointVector** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – The origin of the plane to mirror about

Returns
The mirrored shape

Return type
Shape

move(*loc:* Location) → T

Apply a location in relative sense (i.e. update current location) to self

Parameters

- **self** (*T*) –
- **loc** (Location) –

Return type
T

moved(*loc:* Location) → T

Apply a location in relative sense (i.e. update current location) to a copy of self

Parameters

- **self** (*T*) –
- **loc** (Location) –

Return type
T

rotate(*startVector: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *endVector: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *angleDegrees: float*) → T

Rotates a shape around an axis.

Parameters

- **self** (*T*) –
- **startVector** (*either a 3-tuple or a Vector*) – start point of rotation axis
- **endVector** (*either a 3-tuple or a Vector*) – end point of rotation axis
- **angleDegrees** (*float*) – angle to rotate, in degrees

Returns
a copy of the shape, rotated

Return type
T

scale(*factor: float*) → *Shape*

Scales this shape through a transformation.

Parameters
factor (*float*) –

Return type
Shape

shells(*selector: Optional[Union[Selector, str]] = None*) → *Shape*

Select shells.

Parameters
selector (*Optional[Union[Selector, str]]*) –

Return type
Shape

siblings(*shape: Shape*, *kind: Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*, *level: int = 1*) → *Compound*

Iterate over siblings, i.e. shapes within shape that share subshapes of kind with self.

Parameters

- **shape** (Shape) –
- **kind** (*Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) –
- **level** (*int*) –

Return type
Compound

solids(*selector: Optional[Union[Selector, str]] = None*) → *Shape*

Select solids.

Parameters
selector (*Optional[Union[Selector, str]]*) –

Return type
Shape

split(**splitters:* Shape) → *Shape*

Split this shape with the positional arguments.

Parameters
splitters (Shape) –

Return type
Shape

toSplines(*degree: int = 3*, *tolerance: float = 0.001*, *nurbs: bool = False*) → T

Approximate shape with b-splines of the specified degree.

Parameters

- **self** (*T*) –
- **degree** (*int*) – Maximum degree.
- **tolerance** (*float*) – Approximation tolerance.
- **nurbs** (*bool*) – Use rational splines.

Return type
T

toVtkPolyData(*tolerance: Optional[float] = None*, *angularTolerance: Optional[float] = None*, *normals: bool = False*) → vtkPolyData

Convert shape to vtkPolyData

Parameters

- **tolerance** (*Optional[float]*) –
- **angularTolerance** (*Optional[float]*) –
- **normals** (*bool*) –

Return type
vtkPolyData

transformGeometry(*tMatrix:* Matrix) → *Shape*

Transforms this shape by tMatrix.

WARNING: transformGeometry will sometimes convert lines and circles to splines, but it also has the ability to handle skew and stretching transformations.

If your transformation is only translation and rotation, it is safer to use *transformShape()*, which doesn't change the underlying type of the geometry, but cannot handle skew transformations.

Parameters
tMatrix (Matrix) – The transformation matrix

Returns
a copy of the object, but with geometry transformed instead of just rotated.

Return type
Shape

transformShape(*tMatrix:* Matrix) → *Shape*

Transforms this Shape by tMatrix. Also see *transformGeometry()*.

Parameters
tMatrix (Matrix) – The transformation matrix

Returns
a copy of the object, transformed by the provided matrix, with all objects keeping their type

Return type
Shape

translate(*vector: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → T

Translates this shape through a transformation.

Parameters

- **self** (*T*) –
- **vector** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –

Return type
T

vertices(*selector: Optional[Union[Selector, str]] = None*) → *Shape*

Select vertices.

Parameters
selector (*Optional[Union[Selector, str]]*) –

Return type
Shape

wires(*selector: Optional[Union[Selector, str]] = None*) → *Shape*

Select wires.

Parameters
selector (*Optional[Union[Selector, str]]*) –

Return type
Shape

class cadquery.**Shell**(*obj: TopoDS_Shape*)

Bases: *Shape*

the outer boundary of a surface

Parameters
obj (*TopoDS_Shape*) –

classmethod makeShell(*listOfFaces: Iterable[Face]*) → *Shell*

Makes a shell from faces.

Parameters
listOfFaces (*Iterable[Face]*) –

Return type
Shell

class cadquery.**Sketch**(*parent: ~typing.Any = None*, *locs: ~typing.Iterable[~cadquery.occ_impl.geom.Location] = (<cadquery.occ_impl.geom.Location object>,)*)

Bases: object

2D sketch. Supports faces, edges and edges with constraints based construction.

Parameters

- **parent** (*Any*) –
- **locs** (*List[Location]*) –

__init__(*parent: ~typing.Any = None*, *locs: ~typing.Iterable[~cadquery.occ_impl.geom.Location] = (<cadquery.occ_impl.geom.Location object>,)*)

Construct an empty sketch.

Parameters

- **self** (*T*) –
- **parent** (*Any*) –
- **locs** (*Iterable[Location]*) –

__iter__() → Iterator[*Face*]

Iterate over faces-locations combinations.

Return type
Iterator[Face]

__weakref__

list of weak references to the object (if defined)

arc(*p1: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *p2: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *p3: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

arc(*p2: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *p3: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

arc(*c: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *r: Union[int, float]*, *a: Union[int, float]*, *da: Union[int, float]*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

Construct an arc.

Parameters

- **self** (*T*) –
- **p1** (*Union[Vector, Tuple[Union[int, float], Union[int, float]]]*) –
- **p2** (*Union[Vector, Tuple[Union[int, float], Union[int, float]]]*) –
- **p3** (*Union[Vector, Tuple[Union[int, float], Union[int, float]]]*) –
- **tag** (*Optional[str]*) –
- **forConstruction** (*bool*) –

Return type
T

assemble(*mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Assemble edges into faces.

Parameters

- **self** (*T*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

chamfer(*d: Union[int, float]*) → T

Add a chamfer based on current selection.

Parameters

- **self** (*T*) –
- **d** (*Union[int, float]*) –

Return type
T

circle(*r: Union[int, float]*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Construct a circular face.

Parameters

- **self** (*T*) –
- **r** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

clean() → T

Remove internal wires.

Parameters
self (*T*) –

Return type
T

close(*tag: Optional[str] = None*) → T

Connect last edge to the first one.

Parameters

- **self** (*T*) –
- **tag** (*Optional[str]*) –

Return type
T

constrain(*tag: str*, *constraint: Literal['Fixed', 'FixedPoint', 'Coincident', 'Angle', 'Length', 'Distance', 'Radius', 'Orientation', 'ArcAngle']*, *arg: Any*) → T

constrain(*tag1: str*, *tag2: str*, *constraint: Literal['Fixed', 'FixedPoint', 'Coincident', 'Angle', 'Length', 'Distance', 'Radius', 'Orientation', 'ArcAngle']*, *arg: Any*) → T

Add a constraint.

Parameters

- **self** (*T*) –
- **tag** (*str*) –
- **constraint** (*Literal['Fixed', 'FixedPoint', 'Coincident', 'Angle', 'Length', 'Distance', 'Radius', 'Orientation', 'ArcAngle']*) –
- **arg** (*Any*) –

Return type
T

copy() → T

Create a partial copy of the sketch.

Parameters
self (*T*) –

Return type
T

delete() → T

Delete selected object.

Parameters
self (*T*) –

Return type
T

distribute(*n: int*, *start: Union[int, float] = 0*, *stop: Union[int, float] = 1*, *rotate: bool = True*) → T

Distribute locations along selected edges or wires.

Parameters

- **self** (*T*) –
- **n** (*int*) –
- **start** (*Union[int, float]*) –
- **stop** (*Union[int, float]*) –
- **rotate** (*bool*) –

Return type
T

each(*callback: Callable[[Location], Union[Face, Sketch, Compound]]*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*, *ignore_selection: bool = False*) → T

Apply a callback on all applicable entities.

Parameters

- **self** (*T*) –
- **callback** (*Callable[[Location], Union[Face, Sketch, Compound]]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –
- **ignore_selection** (*bool*) –

Return type
T

edge(*val: Edge*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

Add an edge to the sketch.

Parameters

- **self** (*T*) –
- **val** (*Edge*) –

- **tag** (*Optional[str]*) –
- **forConstruction** (*bool*) –

Return type
T

edges(*s: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select edges.

Parameters

- **self** (*T*) –
- **s** (*Optional[Union[str, Selector]]*) –
- **tag** (*Optional[str]*) –

Return type
T

ellipse(*a1: Union[int, float]*, *a2: Union[int, float]*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Construct an elliptical face.

Parameters

- **self** (*T*) –
- **a1** (*Union[int, float]*) –
- **a2** (*Union[int, float]*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

face(*b: Union[Wire, Iterable[Edge], Compound, T]*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*, *ignore_selection: bool = False*) → T

Construct a face from a wire or edges.

Parameters

- **self** (*T*) –
- **b** (*Union[Wire, Iterable[Edge], Compound, T]*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –
- **ignore_selection** (*bool*) –

Return type
T

faces(*s: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select faces.

Parameters

- **self** (*T*) –
- **s** (*Optional[Union[str, Selector]]*) –
- **tag** (*Optional[str]*) –

Return type
T

fillet(*d: Union[int, float]*) → T

Add a fillet based on current selection.

Parameters

- **self** (*T*) –
- **d** (*Union[int, float]*) –

Return type
T

finalize() → Any

Finish sketch construction and return the parent.

Return type
Any

hull(*mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Generate a convex hull from current selection or all objects.

Parameters

- **self** (*T*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

importDXF(*filename: str*, *tol: float = 1e-06*, *exclude: List[str] = []*, *include: List[str] = []*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Import a DXF file and construct face(s)

Parameters

- **self** (*T*) –
- **filename** (*str*) –
- **tol** (*float*) –
- **exclude** (*List[str]*) –
- **include** (*List[str]*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

located(*loc: Location*) → T

Create a partial copy of the sketch with a new location.

Parameters

- **self** (*T*) –
- **loc** (*Location*) –

Return type
T

moved(*loc: Location*) → T

Create a partial copy of the sketch with moved _faces.

Parameters

- **self** (*T*) –
- **loc** (*Location*) –

Return type
T

offset(*d: Union[int, float]*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Offset selected wires or edges.

Parameters

- **self** (*T*) –
- **d** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

parray(*r: Union[int, float]*, *a1: Union[int, float]*, *da: Union[int, float]*, *n: int*, *rotate: bool = True*) → T

Generate a polar array of locations.

Parameters

- **self** (*T*) –
- **r** (*Union[int, float]*) –
- **a1** (*Union[int, float]*) –
- **da** (*Union[int, float]*) –
- **n** (*int*) –
- **rotate** (*bool*) –

Return type
T

polygon(*pts: Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]]]]*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Construct a polygonal face.

Parameters

- **self** (*T*) –

- **pts** (*Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]]]]*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

push(*locs: Iterable[Union[Location, Vector, Tuple[Union[int, float], Union[int, float]]]]*, *tag: Optional[str] = None*) → T

Set current selection to given locations or points.

Parameters

- **self** (*T*) –
- **locs** (*Iterable[Union[Location, Vector, Tuple[Union[int, float], Union[int, float]]]]*) –
- **tag** (*Optional[str]*) –

Return type
T

rarray(*xs: Union[int, float]*, *ys: Union[int, float]*, *nx: int*, *ny: int*) → T

Generate a rectangular array of locations.

Parameters

- **self** (*T*) –
- **xs** (*Union[int, float]*) –
- **ys** (*Union[int, float]*) –
- **nx** (*int*) –
- **ny** (*int*) –

Return type
T

rect(*w: Union[int, float]*, *h: Union[int, float]*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Construct a rectangular face.

Parameters

- **self** (*T*) –
- **w** (*Union[int, float]*) –
- **h** (*Union[int, float]*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

regularPolygon(*r: Union[int, float]*, *n: int*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Construct a regular polygonal face.

Parameters

- **self** (*T*) –
- **r** (*Union[int, float]*) –
- **n** (*int*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

reset() → T

Reset current selection.

Parameters
self (*T*) –

Return type
T

segment(*l: Union[int, float]*, *a: Union[int, float]*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

segment(*p2: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

segment(*p1: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *p2: Union[Vector, Tuple[Union[int, float], Union[int, float]]]*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

Construct a segment.

Parameters

- **self** (*T*) –
- **p1** (*Union[Vector, Tuple[Union[int, float], Union[int, float]]]*) –
- **p2** (*Union[Vector, Tuple[Union[int, float], Union[int, float]]]*) –
- **tag** (*Optional[str]*) –
- **forConstruction** (*bool*) –

Return type
T

select(**tags: str*) → T

Select based on tags.

Parameters

- **self** (*T*) –
- **tags** (*str*) –

Return type
T

slot(*w: Union[int, float]*, *h: Union[int, float]*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Construct a slot-shaped face.

Parameters

- **self** (*T*) –
- **w** (*Union[int, float]*) –
- **h** (*Union[int, float]*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

solve() → T

Solve current constraints and update edge positions.

Parameters
self (*T*) –

Return type
T

spline(*pts: Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]]]]*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

spline(*pts: Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]]]]*, *tangents: Optional[Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]]]]]*, *periodic: bool*, *tag: Optional[str] = None*, *forConstruction: bool = False*) → T

Construct a spline edge.

Parameters

- **self** (*T*) –
- **pts** (*Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]]]]*) –
- **tangents** (*Optional[Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]]]]]*) –
- **periodic** (*bool*) –
- **tag** (*Optional[str]*) –
- **forConstruction** (*bool*) –

Return type
T

tag(*tag: str*) → T

Tag current selection.

Parameters

- **self** (*T*) –
- **tag** (*str*) –

Return type
T

trapezoid(*w: Union[int, float]*, *h: Union[int, float]*, *a1: Union[int, float]*, *a2: Optional[float] = None*, *angle: Union[int, float] = 0*, *mode: Literal['a', 's', 'i', 'c'] = 'a'*, *tag: Optional[str] = None*) → T

Construct a trapezoidal face.

Parameters

- **self** (*T*) –
- **w** (*Union[int, float]*) –
- **h** (*Union[int, float]*) –
- **a1** (*Union[int, float]*) –
- **a2** (*Optional[float]*) –
- **angle** (*Union[int, float]*) –
- **mode** (*Literal['a', 's', 'i', 'c']*) –
- **tag** (*Optional[str]*) –

Return type
T

val() → Union[*Shape*, *Location*]

Return the first selected item or Location().

Parameters
self (*T*) –

Return type
Union[Shape, Location]

vals() → List[Union[*Shape*, *Location*]]

Return the list of selected items.

Parameters
self (*T*) –

Return type
List[*Union*[Shape, Location]]

vertices(*s: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select vertices.

Parameters

- **self** (*T*) –
- **s** (*Optional[Union[str, Selector]]*) –
- **tag** (*Optional[str]*) –

Return type
T

wires(*s: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select wires.

Parameters

- **self** (*T*) –

- **s** (*Optional[Union[str, Selector]]*) –
- **tag** (*Optional[str]*) –

Return type
T

class cadquery.**Solid**(*obj: TopoDS_Shape*)

Bases: *Shape*, *Mixin3D*

a single solid

Parameters
obj (*TopoDS_Shape*) –

classmethod **extrudeLinear**(*face:* Face, *vecNormal: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *taper: Union[float, int] = 0*) → *Solid*

classmethod **extrudeLinear**(*outerWire:* Wire, *innerWires: List[*Wire*]*, *vecNormal: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *taper: Union[float, int] = 0*) → *Solid*

Attempt to extrude the list of wires into a prismatic solid in the provided direction

Parameters

- **outerWire** (Wire) – the outermost wire
- **innerWires** (*List[Wire]*) – a list of inner wires
- **vecNormal** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – a vector along which to extrude the wires
- **taper** (*Union[float, int]*) – taper angle, default=0

Returns
a Solid object

Return type
Solid

The wires must not intersect

Extruding wires is very non-trivial. Nested wires imply very different geometry, and there are many geometries that are invalid. In general, the following conditions must be met:

- all wires must be closed
- there cannot be any intersecting or self-intersecting wires
- wires must be listed from outside in
- more than one levels of nesting is not supported reliably

This method will attempt to sort the wires, but there is much work remaining to make this method reliable.

classmethod **extrudeLinearWithRotation**(*outerWire:* Wire, *innerWires: List[*Wire*]*, *vecCenter: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *vecNormal: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *angleDegrees: Union[float, int]*) → *Solid*

classmethod extrudeLinearWithRotation(*face: Face*, *vecCenter: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *vecNormal: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *angleDegrees: Union[float, int]*) → *Solid*

Creates a 'twisted prism' by extruding, while simultaneously rotating around the extrusion vector.

Though the signature may appear to be similar enough to extrudeLinear to merit combining them, the construction methods used here are different enough that they should be separate.

At a high level, the steps followed are:

(1) accept a set of wires

(2) create another set of wires like this one, but which are transformed and rotated

(3) create a ruledSurface between the sets of wires

(4) create a shell and compute the resulting object

Parameters

- **outerWire** (*Wire*) – the outermost wire
- **innerWires** (*List[Wire]*) – a list of inner wires
- **vecCenter** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – the center point about which to rotate. the axis of rotation is defined by vecNormal, located at vecCenter.
- **vecNormal** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – a vector along which to extrude the wires
- **angleDegrees** (*Union[float, int]*) – the angle to rotate through while extruding

Returns
a Solid object

Return type
Solid

classmethod interpPlate(*surf_edges*, *surf_pts*, *thickness*, *degree=3*, *nbPtsOnCur=15*, *nbIter=2*, *anisotropy=False*, *tol2d=1e-05*, *tol3d=0.0001*, *tolAng=0.01*, *tolCurv=0.1*, *maxDeg=8*, *maxSegments=9*) → Union[*Solid*, *Face*]

Returns a plate surface that is 'thickness' thick, enclosed by 'surf_edge_pts' points, and going through 'surf_pts' points.

Parameters

- **surf_edges** – list of [x,y,z] float ordered coordinates or list of ordered or unordered wires
- **surf_pts** – list of [x,y,z] float coordinates (uses only edges if [])
- **thickness** – thickness may be negative or positive depending on direction, (returns 2D surface if 0)
- **degree** – >=2
- **nbPtsOnCur** – number of points on curve >= 15

- **nbIter** – number of iterations >= 2
- **anisotropy** – bool Anisotropy
- **tol2d** – 2D tolerance >0
- **tol3d** – 3D tolerance >0
- **tolAng** – angular tolerance
- **tolCurv** – tolerance for curvature >0
- **maxDeg** – highest polynomial degree >= 2
- **maxSegments** – greatest number of segments >= 2

Return type
Union[Solid, Face]

static isSolid(*obj:* Shape) → bool

Returns true if the object is a solid, false otherwise

Parameters
obj (Shape) –

Return type
bool

classmethod makeBox(*length,width,height,[pnt,dir]) -- Make a box located in pnt with the dimensions (length,width,height*)

By default pnt=Vector(0,0,0) and dir=Vector(0,0,1)

Parameters

- **length** (*float*) –
- **width** (*float*) –
- **height** (*float*) –
- **pnt** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –

Return type
Solid

classmethod makeCone(*radius1: float, radius2: float, height: float, pnt: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 1.0), angleDegrees: float = 360*) → *Solid*

Make a cone with given radii and height By default pnt=Vector(0,0,0), dir=Vector(0,0,1) and angle=360

Parameters

- **radius1** (*float*) –
- **radius2** (*float*) –

- **height** (*float*) –
- **pnt** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **angleDegrees** (*float*) –

Return type
Solid

classmethod makeCylinder(*radius: float, height: float, pnt: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 1.0), angleDegrees: float = 360*) → *Solid*

makeCylinder(radius,height,[pnt,dir,angle]) – Make a cylinder with a given radius and height By default pnt=Vector(0,0,0),dir=Vector(0,0,1) and angle=360

Parameters

- **radius** (*float*) –
- **height** (*float*) –
- **pnt** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **angleDegrees** (*float*) –

Return type
Solid

classmethod makeLoft(*listOfWire: List[Wire], ruled: bool = False*) → *Solid*

makes a loft from a list of wires The wires will be converted into faces when possible– it is presumed that nobody ever actually wants to make an infinitely thin shell for a real FreeCADPart.

Parameters

- **listOfWire** (*List[Wire]*) –
- **ruled** (*bool*) –

Return type
Solid

classmethod makeSolid(*shell:* Shell) → *Solid*

Makes a solid from a single shell.

Parameters
shell (Shell) –

Return type
Solid

classmethod makeSphere(*radius: float, pnt: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 1.0), angleDegrees1: float = 0, angleDegrees2: float = 90, angleDegrees3: float = 360*) → *Shape*

Make a sphere with a given radius By default pnt=Vector(0,0,0), dir=Vector(0,0,1), angle1=0, angle2=90 and angle3=360

Parameters

- **radius** (*float*) –
- **pnt** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **angleDegrees1** (*float*) –
- **angleDegrees2** (*float*) –
- **angleDegrees3** (*float*) –

Return type
Shape

classmethod makeTorus(*radius1: float, radius2: float, pnt: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 1.0), angleDegrees1: float = 0, angleDegrees2: float = 360*) → *Solid*

makeTorus(radius1,radius2,[pnt,dir,angle1,angle2,angle]) – Make a torus with a given radii and angles By default pnt=Vector(0,0,0),dir=Vector(0,0,1),angle1=0 ,angle1=360 and angle=360

Parameters

- **radius1** (*float*) –
- **radius2** (*float*) –
- **pnt** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **angleDegrees1** (*float*) –
- **angleDegrees2** (*float*) –

Return type
Solid

classmethod makeWedge(*dx: float, dy: float, dz: float, xmin: float, zmin: float, xmax: float, zmax: float, pnt: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 1.0)*) → *Solid*

Make a wedge located in pnt By default pnt=Vector(0,0,0) and dir=Vector(0,0,1)

Parameters

- **dx** (*float*) –
- **dy** (*float*) –
- **dz** (*float*) –
- **xmin** (*float*) –
- **zmin** (*float*) –
- **xmax** (*float*) –
- **zmax** (*float*) –
- **pnt** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –

Return type
Solid

classmethod revolve(*outerWire: Wire, innerWires: List[Wire], angleDegrees: Union[float, int], axisStart: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], axisEnd: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → *Solid*

classmethod revolve(*face: Face, angleDegrees: Union[float, int], axisStart: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]], axisEnd: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → *Solid*

Attempt to revolve the list of wires into a solid in the provided direction

Parameters

- **outerWire** (*Wire*) – the outermost wire
- **innerWires** (*List[Wire]*) – a list of inner wires
- **angleDegrees** (*float, anything less than 360 degrees will leave the shape open*) – the angle to revolve through.
- **axisStart** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – the start point of the axis of rotation

- **axisEnd** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – the end point of the axis of rotation

Returns
a Solid object

Return type
Solid

The wires must not intersect

- all wires must be closed
- there cannot be any intersecting or self-intersecting wires
- wires must be listed from outside in
- more than one levels of nesting is not supported reliably
- the wire(s) that you're revolving cannot be centered

This method will attempt to sort the wires, but there is much work remaining to make this method reliable.

classmethod sweep(*outerWire: Wire*, *innerWires: List[Wire]*, *path: Union[Wire, Edge]*, *makeSolid: bool = True*, *isFrenet: bool = False*, *mode: Optional[Union[Vector, Wire, Edge]] = None*, *transitionMode: Literal['transformed', 'round', 'right'] = 'transformed'*) → *Shape*

classmethod sweep(*face: Face*, *path: Union[Wire, Edge]*, *makeSolid: bool = True*, *isFrenet: bool = False*, *mode: Optional[Union[Vector, Wire, Edge]] = None*, *transitionMode: Literal['transformed', 'round', 'right'] = 'transformed'*) → *Shape*

Attempt to sweep the list of wires into a prismatic solid along the provided path

Parameters

- **outerWire** (*Wire*) – the outermost wire
- **innerWires** (*List[Wire]*) – a list of inner wires
- **path** (*Union[Wire, Edge]*) – The wire to sweep the face resulting from the wires over
- **makeSolid** (*bool*) – return Solid or Shell (default True)
- **isFrenet** (*bool*) – Frenet mode (default False)
- **mode** (*Optional[Union[Vector, Wire, Edge]]*) – additional sweep mode parameters
- **transitionMode** (*Literal['transformed', 'round', 'right']*) – handling of profile orientation at C1 path discontinuities. Possible values are {'transformed','round', 'right'} (default: 'right').

Returns
a Solid object

Return type
Shape

classmethod sweep_multi(*profiles: Iterable[Union[Wire, Face]]*, *path: Union[Wire, Edge]*, *makeSolid: bool = True*, *isFrenet: bool = False*, *mode: Optional[Union[Vector, Wire, Edge]] = None*) → *Solid*

Multi section sweep. Only single outer profile per section is allowed.

Parameters

- **profiles** (*Iterable[Union[Wire, Face]]*) – list of profiles
- **path** (*Union[Wire, Edge]*) – The wire to sweep the face resulting from the wires over
- **mode** (*Optional[Union[Vector, Wire, Edge]]*) – additional sweep mode parameters.
- **makeSolid** (*bool*) –
- **isFrenet** (*bool*) –

Returns
a Solid object

Return type
Solid

class cadquery.**StringSyntaxSelector**(*selectorString*)

Bases: *Selector*

Filter lists objects using a simple string syntax. All of the filters available in the string syntax are also available (usually with more functionality) through the creation of full-fledged selector objects. see *Selector* and its subclasses

Filtering works differently depending on the type of object list being filtered.

Parameters
selectorString – A two-part selector string, [selector][axis]

Returns
objects that match the specified selector

Modifiers are `('|','+','-','<','>','%')`

|
parallel to (same as *ParallelDirSelector*). Can return multiple objects.

#
perpendicular to (same as *PerpendicularDirSelector*)

+
positive direction (same as *DirectionSelector*)

-
negative direction (same as *DirectionSelector*)

>
maximize (same as *DirectionMinMaxSelector* with directionMax=True)

<
minimize (same as *DirectionMinMaxSelector* with directionMax=False)

%
curve/surface type (same as *TypeSelector*)

axisStrings are: `X,Y,Z,XY,YZ,XZ` or `(x,y,z)` which defines an arbitrary direction

It is possible to combine simple selectors together using logical operations. The following operations are supported

and
Logical AND, e.g. >X and >Y

or
Logical OR, e.g. |X or |Y

not
Logical NOT, e.g. not #XY

exc(ept)
Set difference (equivalent to AND NOT): |X exc >Z

Finally, it is also possible to use even more complex expressions with nesting and arbitrary number of terms, e.g.

(not >X[0] and #XY) or >XY[0]

Selectors are a complex topic: see *Selectors Reference* for more information

__init__(*selectorString*)

Feed the input string through the parser and construct an relevant complex selector object

filter(*objectList: Sequence[Shape]*)

Filter give object list through th already constructed complex selector object

Parameters
objectList (*Sequence[Shape]*) –

class cadquery.**TypeSelector**(*typeString: str*)

Bases: *Selector*

Selects objects having the prescribed geometry type.

Applicability:
Faces: PLANE, CYLINDER, CONE, SPHERE, TORUS, BEZIER, BSPLINE, REVOLUTION, EXTRUSION, OFFSET, OTHER Edges: LINE, CIRCLE, ELLIPSE, HYPERBOLA, PARABOLA, BEZIER, BSPLINE, OFFSET, OTHER

You can use the string selector syntax. For example this:

```
CQ(aCube).faces(TypeSelector("PLANE"))
```

will select 6 faces, and is equivalent to:

```
CQ(aCube).faces("%PLANE")
```

Parameters
typeString (*str*) –

__init__(*typeString: str*)

Parameters
typeString (*str*) –

filter(*objectList: Sequence[Shape]*) → List[Shape]

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

Return type
List[*Shape*]

class cadquery.**Vector**(*x: float*, *y: float*, *z: float*)

class cadquery.**Vector**(*x: float*, *y: float*)
class cadquery.**Vector**(*v:* Vector)
class cadquery.**Vector**(*v: Sequence[float]*)
class cadquery.**Vector**(*v: Union[gp_Vec, gp_Pnt, gp_Dir, gp_XYZ]*)
class cadquery.**Vector**

Bases: object

Create a 3-dimensional vector

Parameters
args – a 3D vector, with x-y-z parts.

you can either provide:

- nothing (in which case the null vector is return)
- a gp_Vec
- a vector (in which case it is copied)
- a 3-tuple
- a 2-tuple (z assumed to be 0)
- three float values: x, y, and z
- two float values: x,y

Center() → *Vector*

Return the vector itself

The center of myself is myself. Provided so that vectors, vertices, and other shapes all support a common interface, when Center() is requested for all objects on the stack.

Return type
Vector

__eq__(*other:* Vector) → bool

Return self==value.

Parameters
other (Vector) –

Return type
bool

__hash__ = None

__init__(*x: float*, *y: float*, *z: float*) → None
__init__(*x: float*, *y: float*) → None
__init__(*v:* Vector) → None
__init__(*v: Sequence[float]*) → None
__init__(*v: Union[gp_Vec, gp_Pnt, gp_Dir, gp_XYZ]*) → None
__init__() → None

__repr__() → str

Return repr(self).

Return type
str

__str__() → str

Return str(self).

Return type
str

__weakref__

list of weak references to the object (if defined)

multiply(*scale: float*) → *Vector*

Return a copy multiplied by the provided scalar

Parameters
scale (*float*) –

Return type
Vector

normalized() → *Vector*

Return a normalized version of this vector

Return type
Vector

projectToLine(*line:* Vector) → *Vector*

Returns a new vector equal to the projection of this Vector onto the line represented by Vector <line>

Parameters

- **args** – Vector
- **line** (Vector) –

Return type
Vector

Returns the projected vector.

projectToPlane(*plane:* Plane) → *Vector*

Vector is projected onto the plane provided as input.

Parameters

- **args** – Plane object
- **plane** (Plane) –

Return type
Vector

Returns the projected vector.

class cadquery.**Vertex**(*obj: TopoDS_Shape*, *forConstruction: bool = False*)

Bases: *Shape*

A Single Point in Space

Parameters

- **obj** (*TopoDS_Shape*) –
- **forConstruction** (*bool*) –

Center() → *Vector*

The center of a vertex is itself!

Return type
Vector

__init__(*obj: TopoDS_Shape*, *forConstruction: bool = False*)

Create a vertex

Parameters

- **obj** (*TopoDS_Shape*) –
- **forConstruction** (*bool*) –

class cadquery.**Wire**(*obj: TopoDS_Shape*)

Bases: *Shape*, *Mixin1D*

A series of connected, ordered Edges, that typically bounds a Face

Parameters
obj (*TopoDS_Shape*) –

__iter__() → Iterator[*Edge*]

Iterate over edges in an ordered way.

Return type
Iterator[Edge]

classmethod assembleEdges(*listOfEdges: Iterable[*Edge*]*) → *Wire*

Attempts to build a wire that consists of the edges in the provided list

Parameters

- **cls** –
- **listOfEdges** (*Iterable[*Edge*]*) – a list of Edge objects. The edges are not to be consecutive.

Returns
a wire with the edges assembled

Return type
Wire

BRepBuilderAPI_MakeWire::Error() values:

- BRepBuilderAPI_WireDone = 0
- BRepBuilderAPI_EmptyWire = 1
- BRepBuilderAPI_DisconnectedWire = 2
- BRepBuilderAPI_NonManifoldWire = 3

chamfer2D(*d: float*, *vertices: Iterable[*Vertex*]*) → *Wire*

Apply 2D chamfer to a wire

Parameters

- **d** (*float*) –
- **vertices** (*Iterable[*Vertex*]*) –

Return type
Wire

close() → *Wire*

Close a Wire

Return type
Wire

classmethod combine(*listOfWires: Iterable[Union[*Wire*,* Edge*]]*, *tol: float = 1e-09*) → List[*Wire*]

Attempt to combine a list of wires and edges into a new wire.

Parameters

- **cls** –
- **listOfWires** (*Iterable[Union[*Wire*,* Edge*]]*) –
- **tol** (*float*) – default 1e-9

Returns
List[Wire]

Return type
List[Wire]

fillet2D(*radius: float*, *vertices: Iterable[*Vertex*]*) → *Wire*

Apply 2D fillet to a wire

Parameters

- **radius** (*float*) –
- **vertices** (*Iterable[*Vertex*]*) –

Return type
Wire

classmethod makeCircle(*radius: float*, *center: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *normal: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) → *Wire*

Makes a Circle centered at the provided point, having normal in the provided direction

Parameters

- **radius** (*float*) – floating point radius of the circle, must be > 0
- **center** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – vector representing the center of the circle
- **normal** (*Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – vector representing the direction of the plane the circle should lie in

Return type
Wire

classmethod makeEllipse(*x_radius: float*, *y_radius: float*, *center: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *normal: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *xDir: Union[*Vector*, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *angle1: float = 360.0*, *angle2: float = 360.0*, *rotation_angle: float = 0.0*, *closed: bool = True*) → *Wire*

Makes an Ellipse centered at the provided point, having normal in the provided direction

Parameters

- **x_radius** (*float*) – floating point major radius of the ellipse (x-axis), must be > 0
- **y_radius** (*float*) – floating point minor radius of the ellipse (y-axis), must be > 0
- **center** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – vector representing the center of the circle
- **normal** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – vector representing the direction of the plane the circle should lie in
- **angle1** (*float*) – start angle of arc
- **angle2** (*float*) – end angle of arc
- **rotation_angle** (*float*) – angle to rotate the created ellipse / arc
- **xDir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **closed** (*bool*) –

Return type
Wire

classmethod makeHelix(*pitch: float, height: float, radius: float, center: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~cadquery.occ_impl.geom.Vector, ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float]], ~typing.Tuple[~typing.Union[int, float], ~typing.Union[int, float], ~typing.Union[int, float]]] = Vector: (0.0, 0.0, 1.0), angle: float = 360.0, lefthand: bool = False*) → *Wire*

Make a helix with a given pitch, height and radius By default a cylindrical surface is used to create the helix. If the fourth parameter is set (the apex given in degree) a conical surface is used instead'

Parameters

- **pitch** (*float*) –
- **height** (*float*) –
- **radius** (*float*) –
- **center** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **dir** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **angle** (*float*) –
- **lefthand** (*bool*) –

Return type
Wire

classmethod makePolygon(*listOfVertices: Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]]*, *forConstruction: bool = False*, *close: bool = False*) → *Wire*

Construct a polygonal wire from points.

Parameters

- **listOfVertices** (*Iterable[Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]]*) –
- **forConstruction** (*bool*) –
- **close** (*bool*) –

Return type
Wire

offset2D(*d: float*, *kind: Literal['arc', 'intersection', 'tangent'] = 'arc'*) → List[*Wire*]

Offsets a planar wire

Parameters

- **d** (*float*) –
- **kind** (*Literal['arc', 'intersection', 'tangent']*) –

Return type
List[Wire]

stitch(*other:* Wire) → *Wire*

Attempt to stitch wires

Parameters
other (*Wire*) –

Return type
Wire

class cadquery.**Workplane**(*obj: Union[Vector, Location, Shape, Sketch]*)

class cadquery.**Workplane**(*inPlane: Union[Plane, str] = 'XY'*, *origin: Union[Tuple[float, float], Tuple[float, float, float], Vector] = (0, 0, 0)*, *obj: Optional[Union[Vector, Location, Shape, Sketch]] = None*)

Bases: object

Defines a coordinate system in space, in which 2D coordinates can be used.

Parameters

- **plane** (*a Plane object, or a string in (XY|YZ|XZ|front|back|top|bottom|left|right)*) – the plane in which the workplane will be done
- **origin** (*a 3-tuple in global coordinates, or None to default to the origin*) – the desired origin of the new workplane
- **obj** (*a CAD primitive, or None to use the centerpoint of the plane as the initial stack value.*) – an object to use initially for the stack

Raises
ValueError if the provided plane is not a plane, a valid named workplane

Returns
A Workplane object, with coordinate system matching the supplied plane.

The most common use is:

```
s = Workplane("XY")
```

After creation, the stack contains a single point, the origin of the underlying plane, and the *current point* is on the origin.

Note: You can also create workplanes on the surface of existing faces using *workplane()*

__add__(*toUnion: Union[*Workplane, Solid, Compound*]*) → T

Syntactic sugar for union.

Notice that `r = a + b` is equivalent to `r = a.union(b)` and `r = a | b`.

Parameters

- **self** (*T*) –
- **toUnion** (*Union[*Workplane, Solid, Compound*]*) –

Return type
T

__and__(*toUnion: Union[*Workplane, Solid, Compound*]*) → T

Syntactic sugar for intersect.

Notice that `r = a & b` is equivalent to `r = a.intersect(b)`.

Example:

```
Box = Workplane("XY").box(1, 1, 1, centered=(False, False, False))
Sphere = Workplane("XY").sphere(1)
result = Box & Sphere
```

Parameters

- **self** (*T*) –
- **toUnion** (*Union[*Workplane, Solid, Compound*]*) –

Return type
T

__init__(*obj: Union[*Vector, Location, Shape, Sketch*]*) → None

__init__(*inPlane: Union[*Plane, *str] = 'XY'*, *origin: Union[Tuple[float, float], Tuple[float, float, float],* Vector*] = (0, 0, 0)*, *obj: Optional[Union[*Vector, Location, Shape, Sketch*]] = None*) → None

make a workplane from a particular plane

Parameters

- **inPlane** (*a Plane object, or a string in (XY|YZ|XZ|front|back|top|bottom|left|right)*) – the plane in which the workplane will be done
- **origin** (*a 3-tuple in global coordinates, or None to default to the origin*) – the desired origin of the new workplane
- **obj** (*a CAD primitive, or None to use the centerpoint of the plane as the initial stack value.*) – an object to use initially for the stack

Raises
ValueError if the provided plane is not a plane, or one of XY|YZ|XZ

Returns
A Workplane object, with coordinate system matching the supplied plane.

The most common use is:

```
s = Workplane("XY")
```

After creation, the stack contains a single point, the origin of the underlying plane, and the *current point* is on the origin.

__or__(*toUnion: Union[*Workplane, Solid, Compound*]*) → T

Syntactic sugar for union.

Notice that `r = a | b` is equivalent to `r = a.union(b)` and `r = a + b`.

Example:

```
Box = Workplane("XY").box(1, 1, 1, centered=(False, False, False))
Sphere = Workplane("XY").sphere(1)
result = Box | Sphere
```

Parameters

- **self** (*T*) –
- **toUnion** (*Union[*Workplane, Solid, Compound*]*) –

Return type
T

__sub__(*toUnion: Union[*Workplane, Solid, Compound*]*) → T

Syntactic sugar for cut.

Notice that `r = a - b` is equivalent to `r = a.cut(b)`.

Example:

```
Box = Workplane("XY").box(1, 1, 1, centered=(False, False, False))
Sphere = Workplane("XY").sphere(1)
result = Box - Sphere
```

Parameters

- **self** (*T*) –
- **toUnion** (*Union[*Workplane, Solid, Compound*]*) –

Return type
T

__weakref__

list of weak references to the object (if defined)

add(*obj:* Workplane) → T

add(*obj: Union[*Vector, Location, Shape, Sketch*]*) → T

add(*obj: Iterable[Union[*Vector, Location, Shape, Sketch*]]*) → T

Adds an object or a list of objects to the stack

Parameters
obj (*a Workplane, CAD primitive, or list of CAD primitives*) – an object to add

Returns
a Workplane with the requested operation performed

If a Workplane object, the values of that object's stack are added. If a list of cad primitives, they are all added. If a single CAD primitive then it is added.

Used in rare cases when you need to combine the results of several CQ results into a single Workplane object.

all() → List[T]

Return a list of all CQ objects on the stack.

useful when you need to operate on the elements individually.

Contrast with vals, which returns the underlying objects for all of the items on the stack

Parameters
self (*T*) –

Return type
List[*T*]

ancestors(*kind: Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*, *tag: Optional[str] = None*) → T

Select topological ancestors.

Parameters

- **self** (*T*) –
- **kind** (*Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) – kind of ancestor, e.g. "Face" or "Edge"
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a Workplane object whose stack contains selected ancestors.

Return type
T

box(*length: float*, *width: float*, *height: float*, *centered: Union[bool, Tuple[bool, bool, bool]] = True*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*) → T

Return a 3d box with specified dimensions for each object on the stack.

Parameters

- **self** (*T*) –
- **length** (*float*) – box size in X direction
- **width** (*float*) – box size in Y direction
- **height** (*float*) – box size in Z direction
- **centered** (*Union[bool, Tuple[bool, bool, bool]]*) – If True, the box will be centered around the reference point. If False, the corner of the box will be on the reference

point and it will extend in the positive x, y and z directions. Can also use a 3-tuple to specify centering along each axis.

- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – should the results be combined with other solids on the stack (and each other)?
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Return type
T

One box is created for each item on the current stack. If no items are on the stack, one box using the current workplane center is created.

If combine is true, the result will be a single object on the stack. If a solid was found in the chain, the result is that solid with all boxes produced fused onto it otherwise, the result is the combination of all the produced boxes.

If combine is false, the result will be a list of the boxes produced.

Most often boxes form the basis for a part:

```
# make a single box with lower left corner at origin
s = Workplane().box(1, 2, 3, centered=False)
```

But sometimes it is useful to create an array of them:

```
# create 4 small square bumps on a larger base plate:
s = (
    Workplane()
    .box(4, 4, 0.5)
    .faces(">Z")
    .workplane()
    .rect(3, 3, forConstruction=True)
    .vertices()
    .box(0.25, 0.25, 0.25, combine=True)
)
```

cboreHole(*diameter: float*, *cboreDiameter: float*, *cboreDepth: float*, *depth: Optional[float] = None*, *clean: bool = True*) → T

Makes a counterbored hole for each item on the stack.

Parameters

- **self** (*T*) –
- **diameter** (*float*) – the diameter of the hole
- **cboreDiameter** (*float*) – the diameter of the cbore, must be greater than hole diameter
- **cboreDepth** (*float > 0*) – depth of the counterbore
- **depth** (*float > 0 or None to drill thru the entire part*) – the depth of the hole
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Return type
T

The surface of the hole is at the current workplane plane.

One hole is created for each item on the stack. A very common use case is to use a construction rectangle to define the centers of a set of holes, like so:

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

This sample creates a plate with a set of holes at the corners.

Plugin Note: this is one example of the power of plugins. Counterbored holes are quite time consuming to create, but are quite easily defined by users.

see *cskHole()* to make countersinks instead of counterbores

center(*x: float*, *y: float*) → T

Shift local coordinates to the specified location.

The location is specified in terms of local coordinates.

Parameters

- **self** (*T*) –
- **x** (*float*) – the new x location
- **y** (*float*) – the new y location

Returns
the Workplane object, with the center adjusted.

Return type
T

The current point is set to the new center. This method is useful to adjust the center point after it has been created automatically on a face, but not where you'd like it to be.

In this example, we adjust the workplane center to be at the corner of a cube, instead of the center of a face, which is the default:

```
# this workplane is centered at x=0.5,y=0.5, the center of the upper face
s = Workplane().box(1, 1, 1).faces(">Z").workplane()

s = s.center(-0.5, -0.5)  # move the center to the corner
t = s.circle(0.25).extrude(0.2)
assert t.faces().size() == 9  # a cube with a cylindrical nub at the top right␣
↪corner
```

The result is a cube with a round boss on the corner

chamfer(*length: float*, *length2: Optional[float] = None*) → T

Chamfers a solid on the selected edges.

The edges on the stack are chamfered. The solid to which the edges belong must be in the parent chain of the selected edges.

Optional parameter *length2* can be supplied with a different value than *length* for a chamfer that is shorter on one side longer on the other side.

Parameters

- **self** (*T*) –
- **length** (*float*) – the length of the chamfer, must be greater than zero
- **length2** (*Optional[float]*) – optional parameter for asymmetrical chamfer

Raises

- **ValueError** – if at least one edge is not selected
- **ValueError** – if the solid containing the edge is not in the chain

Returns
CQ object with the resulting solid selected.

Return type
T

This example will create a unit cube, with the top edges chamfered:

```
s = Workplane("XY").box(1, 1, 1).faces("+Z").chamfer(0.1)
```

This example will create chamfers longer on the sides:

```
s = Workplane("XY").box(1, 1, 1).faces("+Z").chamfer(0.2, 0.1)
```

circle(*radius: float*, *forConstruction: bool = False*) → T

Make a circle for each item on the stack.

Parameters

- **self** (*T*) –
- **radius** (*float*) – radius of the circle
- **forConstruction** (*true if the wires are for reference, false if they are creating part geometry*) – should the new wires be reference geometry only?

Returns
a new CQ object with the created wires on the stack

Return type
T

A common use case is to use a for-construction rectangle to define the centers of a hole pattern:

```
s = Workplane().rect(4.0, 4.0, forConstruction=True).vertices().circle(0.25)
```

Creates 4 circles at the corners of a square centered on the origin. Another common case is to use successive circle() calls to create concentric circles. This works because the center of a circle is its reference point:

```
s = Workplane().circle(2.0).circle(1.0)
```

Creates two concentric circles, which when extruded will form a ring.

Future Enhancements:
better way to handle forConstruction project points not in the workplane plane onto the workplane plane

clean() → T

Cleans the current solid by removing unwanted edges from the faces.

Normally you don't have to call this function. It is automatically called after each related operation. You can disable this behavior with *clean=False* parameter if method has any. In some cases this can improve performance drastically but is generally dis-advised since it may break some operations such as fillet.

Note that in some cases where lots of solid operations are chained, *clean()* may actually improve performance since the shape is 'simplified' at each step and thus next operation is easier.

Also note that, due to limitation of the underlying engine, *clean* may fail to produce a clean output in some cases such as spherical faces.

Parameters
self (*T*) –

Return type
T

close() → T

End construction, and attempt to build a closed wire.

Returns
a CQ object with a completed wire on the stack, if possible.

Parameters
self (*T*) –

Return type
T

After 2D (or 3D) drafting with methods such as lineTo, threePointArc, tangentArcPoint and polyline, it is necessary to convert the edges produced by these into one or more wires.

When a set of edges is closed, CadQuery assumes it is safe to build the group of edges into a wire. This example builds a simple triangular prism:

```
s = Workplane().lineTo(1, 0).lineTo(1, 1).close().extrude(0.2)
```

combine(*clean: bool = True*, *glue: bool = False*, *tol: Optional[float] = None*) → T

Attempts to combine all of the items on the stack into a single item.

WARNING: all of the items must be of the same type!

Parameters

- **self** (*T*) –
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **glue** (*bool*) – use a faster gluing mode for non-overlapping shapes (default False)
- **tol** (*Optional[float]*) – tolerance value for fuzzy bool operation mode (default None)

Raises
ValueError if there are no items on the stack, or if they cannot be combined

Returns
a CQ object with the resulting object selected

Return type
T

combineSolids(*otherCQToCombine: Optional[*Workplane*] = None*) → *Workplane*

!!!DEPRECATED!!! use union() Combines all solids on the current stack, and any context object, together into a single object.

After the operation, the returned solid is also the context solid.

Parameters
otherCQToCombine (*Optional[Workplane]*) – another CadQuery to combine.

Returns
a CQ object with the resulting combined solid on the stack.

Return type
Workplane

Most of the time, both objects will contain a single solid, which is combined and returned on the stack of the new object.

compounds(*selector: Optional[Union[str,* Selector*]] = None*, *tag: Optional[str] = None*) → T

Select compounds on the stack, optionally filtering the selection. If there are multiple objects on the stack, they are collected and a list of all the distinct compounds is returned.

Parameters

- **self** (*T*) –
- **selector** (*Optional[Union[str, Selector]]*) – optional Selector object, or string selector expression (see *StringSyntaxSelector*)
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a CQ object whose stack contains all of the *distinct* compounds of *all* objects on the current stack, filtered by the provided selector.

Return type
T

A compound contains multiple CAD primitives that resulted from a single operation, such as a union, cut, split, or fillet. Compounds can contain multiple edges, wires, or solids.

consolidateWires() → T

Attempt to consolidate wires on the stack into a single. If possible, a new object with the results are returned. if not possible, the wires remain separated

Parameters
self (*T*) –

Return type
T

copyWorkplane(*obj: T*) → T

Copies the workplane from obj.

Parameters
obj (*a CQ object*) – an object to copy the workplane from

Returns
a CQ object with obj's workplane

Return type
T

cskHole(*diameter: float*, *cskDiameter: float*, *cskAngle: float*, *depth: Optional[float] = None*, *clean: bool = True*) → T

Makes a countersunk hole for each item on the stack.

Parameters

- **self** (*T*) –
- **diameter** (*float > 0*) – the diameter of the hole
- **cskDiameter** (*float*) – the diameter of the countersink, must be greater than hole diameter
- **cskAngle** (*float > 0*) – angle of the countersink, in degrees (82 is common)
- **depth** (*float > 0 or None to drill thru the entire part.*) – the depth of the hole
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Return type
T

The surface of the hole is at the current workplane.

One hole is created for each item on the stack. A very common use case is to use a construction rectangle to define the centers of a set of holes, like so:

```
s = (
    Workplane()
    .box(2, 4, 0.5)
    .faces(">Z")
    .workplane()
    .rect(1.5, 3.5, forConstruction=True)
    .vertices()
    .cskHole(0.125, 0.25, 82, depth=None)
)
```

This sample creates a plate with a set of holes at the corners.

Plugin Note: this is one example of the power of plugins. CounterSunk holes are quite time consuming to create, but are quite easily defined by users.

see *cboreHole()* to make counterbores instead of countersinks

cut(*toCut: Union[Workplane, Solid, Compound]*, *clean: bool = True*, *tol: Optional[float] = None*) → T

Cuts the provided solid from the current solid, IE, perform a solid subtraction.

Parameters

- **self** (*T*) –
- **toCut** (*Union[Workplane, Solid, Compound]*) – a solid object, or a Workplane object having a solid
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **tol** (*Optional[float]*) – tolerance value for fuzzy bool operation mode (default None)

Raises
ValueError – if there is no solid to subtract from in the chain

Returns
a Workplane object with the resulting object selected

Return type
T

cutBlind(*until: Union[float, Literal['next', 'last'], Face], clean: bool = True, both: bool = False, taper: Optional[float] = None*) → T

Use all un-extruded wires in the parent chain to create a prismatic cut from existing solid.

Specify either a distance value, or one of "next", "last" to indicate a face to cut to.

Similar to extrude, except that a solid in the parent chain is required to remove material from. cutBlind always removes material from a part.

Parameters

- **self** (*T*) –
- **until** (*Union[float, Literal['next', 'last'], Face]*) – The distance to cut to, normal to the workplane plane. When a negative float is passed the cut extends this far in the opposite direction to the normal of the plane (i.e in the solid). The string "next" cuts until the next face orthogonal to the wire normal. "last" cuts to the last face. If an object of type Face is passed, then the cut will extend until this face.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **both** (*bool*) – cut in both directions symmetrically
- **taper** (*Optional[float]*) – angle for optional tapered extrusion

Raises
ValueError – if there is no solid to subtract from in the chain

Returns
a CQ object with the resulting object selected

Return type
T

see *cutThruAll()* to cut material from the entire part

cutEach(*fcn: Callable[[Location], Shape], useLocalCoords: bool = False, clean: bool = True*) → T

Evaluates the provided function at each point on the stack (ie, eachpoint) and then cuts the result from the context solid.

Parameters

- **self** (*T*) –
- **fcn** (*Callable[[Location], Shape]*) – a function suitable for use in the eachpoint method: ie, that accepts a vector
- **useLocalCoords** (*bool*) – same as for *eachpoint()*
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Raises
ValueError – if no solids or compounds are found in the stack or parent chain

Returns
a CQ object that contains the resulting solid

Return type
T

cutThruAll(*clean: bool = True*, *taper: float = 0*) → T

Use all un-extruded wires in the parent chain to create a prismatic cut from existing solid. Cuts through all material in both normal directions of workplane.

Similar to extrude, except that a solid in the parent chain is required to remove material from. cutThruAll always removes material from a part.

Parameters

- **self** (*T*) –
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **taper** (*float*) –

Raises

- **ValueError** – if there is no solid to subtract from in the chain
- **ValueError** – if there are no pending wires to cut with

Returns
a CQ object with the resulting object selected

Return type
T

see *cutBlind()* to cut material to a limited depth

cylinder(*height: float*, *radius: float*, *direct: ~cadquery.occ_impl.geom.Vector = Vector: (0.0, 0.0, 1.0)*, *angle: float = 360*, *centered: ~typing.Union[bool, ~typing.Tuple[bool, bool, bool]] = True*, *combine: ~typing.Union[bool, ~typing.Literal['cut', 'a', 's']] = True*, *clean: bool = True*) → T

Returns a cylinder with the specified radius and height for each point on the stack

Parameters

- **self** (*T*) –
- **height** (*float*) – The height of the cylinder
- **radius** (*float*) – The radius of the cylinder
- **direct** (*A three-tuple*) – The direction axis for the creation of the cylinder
- **angle** (*float > 0*) – The angle to sweep the cylinder arc through
- **centered** (*Union[bool, Tuple[bool, bool, bool]]*) – If True, the cylinder will be centered around the reference point. If False, the corner of a bounding box around the cylinder will be on the reference point and it will extend in the positive x, y and z directions. Can also use a 3-tuple to specify centering along each axis.
- **combine** (*true to combine shapes, false otherwise*) – Whether the results should be combined with other solids on the stack (and each other)
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Returns
A cylinder object for each point on the stack

Return type
T

One cylinder is created for each item on the current stack. If no items are on the stack, one cylinder is created using the current workplane center.

If combine is true, the result will be a single object on the stack. If a solid was found in the chain, the result is that solid with all cylinders produced fused onto it otherwise, the result is the combination of all the produced cylinders.

If combine is false, the result will be a list of the cylinders produced.

each(*callback: Callable[[Union[Vector, Location, Shape, Sketch]], Shape]*, *useLocalCoordinates: bool = False*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*) → T

Runs the provided function on each value in the stack, and collects the return values into a new CQ object.

Special note: a newly created workplane always has its center point as its only stack item

Parameters

- **self** (*T*) –
- **callBackFunction** – the function to call for each item on the current stack.
- **useLocalCoordinates** (*bool*) – should values be converted from local coordinates first?
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or "a" to combine the resulting solid with parent solids if found, "cut" or "s" to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **callback** (*Callable[[Union[Vector, Location, Shape, Sketch]], Shape]*) –

Return type
T

The callback function must accept one argument, which is the item on the stack, and return one object, which is collected. If the function returns None, nothing is added to the stack. The object passed into the callBackFunction is potentially transformed to local coordinates, if useLocalCoordinates is true

useLocalCoordinates is very useful for plugin developers.

If false, the callback function is assumed to be working in global coordinates. Objects created are added as-is, and objects passed into the function are sent in using global coordinates

If true, the calling function is assumed to be working in local coordinates. Objects are transformed to local coordinates before they are passed into the callback method, and result objects are transformed to global coordinates after they are returned.

This allows plugin developers to create objects in local coordinates, without worrying about the fact that the working plane is different than the global coordinate system.

TODO: wrapper object for Wire will clean up forConstruction flag everywhere

eachpoint(*callback: Callable[[Location], Shape]*, *useLocalCoordinates: bool = False*, *combine: Union[bool, Literal['cut', 'a', 's']] = False*, *clean: bool = True*) → T

Same as each(), except each item on the stack is converted into a point before it is passed into the callback function.

Returns
CadQuery object which contains a list of vectors (points) on its stack.

Parameters

- **self** (*T*) –
- **useLocalCoordinates** (*bool*) – should points be in local or global coordinates

- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or "a" to combine the resulting solid with parent solids if found, "cut" or "s" to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **callback** (*Callable[[Location], Shape]*) –

Return type
T

The resulting object has a point on the stack for each object on the original stack. Vertices and points remain a point. Faces, Wires, Solids, Edges, and Shells are converted to a point by using their center of mass.

If the stack has zero length, a single point is returned, which is the center of the current workplane/coordinate system

edges(*selector: Optional[Union[str,* Selector*]] = None*, *tag: Optional[str] = None*) → T

Select the edges of objects on the stack, optionally filtering the selection. If there are multiple objects on the stack, the edges of all objects are collected and a list of all the distinct edges is returned.

Parameters

- **self** (*T*) –
- **selector** (*Optional[Union[str,* Selector*]]*) – optional Selector object, or string selector expression (see *StringSyntaxSelector*)
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a CQ object whose stack contains all of the *distinct* edges of *all* objects on the current stack, filtered by the provided selector.

Return type
T

If there are no edges for any objects on the current stack, an empty CQ object is returned

The typical use is to select the edges of a single object on the stack. For example:

```
Workplane().box(1, 1, 1).faces("+Z").edges().size()
```

returns 4, because the topmost face of a cube will contain four edges. Similarly:

```
Workplane().box(1, 1, 1).edges().size()
```

returns 12, because a cube has a total of 12 edges, And:

```
Workplane().box(1, 1, 1).edges("|Z").size()
```

returns 4, because a cube has 4 edges parallel to the z direction

ellipse(*x_radius: float*, *y_radius: float*, *rotation_angle: float = 0.0*, *forConstruction: bool = False*) → T

Make an ellipse for each item on the stack.

Parameters

- **self** (*T*) –
- **x_radius** (*float*) – x radius of the ellipse (x-axis of plane the ellipse should lie in)
- **y_radius** (*float*) – y radius of the ellipse (y-axis of plane the ellipse should lie in)

- **rotation_angle** (*float*) – angle to rotate the ellipse
- **forConstruction** (*true if the wires are for reference, false if they are creating part geometry*) – should the new wires be reference geometry only?

Returns
a new CQ object with the created wires on the stack

Return type
T

NOTE Due to a bug in opencascade (https://tracker.dev.opencascade.org/view.php?id=31290) the center of mass (equals center for next shape) is shifted. To create concentric ellipses use:

```
Workplane("XY").center(10, 20).ellipse(100, 10).center(0, 0).ellipse(50, 5)
```

ellipseArc(*x_radius: float*, *y_radius: float*, *angle1: float = 360*, *angle2: float = 360*, *rotation_angle: float = 0.0*, *sense: Literal[- 1, 1] = 1*, *forConstruction: bool = False*, *startAtCurrent: bool = True*, *makeWire: bool = False*) → T

Draw an elliptical arc with x and y radiuses either with start point at current point or or current point being the center of the arc

Parameters

- **self** (*T*) –
- **x_radius** (*float*) – x radius of the ellipse (along the x-axis of plane the ellipse should lie in)
- **y_radius** (*float*) – y radius of the ellipse (along the y-axis of plane the ellipse should lie in)
- **angle1** (*float*) – start angle of arc
- **angle2** (*float*) – end angle of arc (angle2 == angle1 return closed ellipse = default)
- **rotation_angle** (*float*) – angle to rotate the created ellipse / arc
- **sense** (*Literal[-1, 1]*) – clockwise (-1) or counter clockwise (1)
- **startAtCurrent** (*bool*) – True: start point of arc is moved to current point; False: center of arc is on current point
- **makeWire** (*bool*) – convert the resulting arc edge to a wire
- **forConstruction** (*bool*) –

Return type
T

end(*n: int = 1*) → *Workplane*

Return the nth parent of this CQ element

Parameters
n (*int*) – number of ancestor to return (default: 1)

Return type
a CQ object

Raises
ValueError if there are no more parents in the chain.

For example:

```
CQ(obj).faces("+Z").vertices().end()
```

will return the same as:

```
CQ(obj).faces("+Z")
```

exportSvg(*fileName: str*) → None

Exports the first item on the stack as an SVG file

For testing purposes mainly.

Parameters
fileName (*str*) – the filename to export, absolute path to the file

Return type
None

extrude(*until: Union[float, Literal['next', 'last'], Face]*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*, *both: bool = False*, *taper: Optional[float] = None*) → T

Use all un-extruded wires in the parent chain to create a prismatic solid.

Parameters

- **self** (*T*) –
- **until** (*Union[float, Literal['next', 'last'], Face]*) – The distance to extrude, normal to the workplane plane. When a float is passed, the extrusion extends this far and a negative value is in the opposite direction to the normal of the plane. The string "next" extrudes until the next face orthogonal to the wire normal. "last" extrudes to the last face. If a object of type Face is passed then the extrusion will extend until this face. **Note that the Workplane must contain a Solid for extruding to a given face.**
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or "a" to combine the resulting solid with parent solids if found, "cut" or "s" to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **both** (*bool*) – extrude in both directions symmetrically
- **taper** (*Optional[float]*) – angle for optional tapered extrusion

Returns
a CQ object with the resulting solid selected.

Return type
T

The returned object is always a CQ object, and depends on whether combine is True, and whether a context solid is already defined:

- **if combine is False, the new value is pushed onto the stack. Note that when extruding**
  until a specified face, combine can not be False
- **if combine is true, the value is combined with the context solid if it exists,**
  and the resulting solid becomes the new context solid.

faces(*selector: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select the faces of objects on the stack, optionally filtering the selection. If there are multiple objects on the stack, the faces of all objects are collected and a list of all the distinct faces is returned.

Parameters

- **self** (*T*) –
- **selector** (*Optional[Union[str, Selector]]*) – optional Selector object, or string selector expression (see *StringSyntaxSelector*)
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a CQ object whose stack contains all of the *distinct* faces of *all* objects on the current stack, filtered by the provided selector.

Return type
T

If there are no faces for any objects on the current stack, an empty CQ object is returned.

The typical use is to select the faces of a single object on the stack. For example:

```
Workplane().box(1, 1, 1).faces("+Z").size()
```

returns 1, because a cube has one face with a normal in the +Z direction. Similarly:

```
Workplane().box(1, 1, 1).faces().size()
```

returns 6, because a cube has a total of 6 faces, And:

```
Workplane().box(1, 1, 1).faces("|Z").size()
```

returns 2, because a cube has 2 faces having normals parallel to the z direction

fillet(*radius: float*) → T

Fillets a solid on the selected edges.

The edges on the stack are filleted. The solid to which the edges belong must be in the parent chain of the selected edges.

Parameters

- **self** (*T*) –
- **radius** (*float*) – the radius of the fillet, must be > zero

Raises

- **ValueError** – if at least one edge is not selected
- **ValueError** – if the solid containing the edge is not in the chain

Returns
CQ object with the resulting solid selected.

Return type
T

This example will create a unit cube, with the top edges filleted:

```
s = Workplane().box(1, 1, 1).faces("+Z").edges().fillet(0.1)
```

findFace(*searchStack: bool = True*, *searchParents: bool = True*) → *Face*

Finds the first face object in the chain, searching from the current node backwards through parents until one is found.

Parameters

- **searchStack** (*bool*) – should objects on the stack be searched first.
- **searchParents** (*bool*) – should parents be searched?

Returns
A face or None if no face is found.

Return type
Face

findSolid(*searchStack: bool = True*, *searchParents: bool = True*) → Union[*Solid*, *Compound*]

Finds the first solid object in the chain, searching from the current node backwards through parents until one is found.

Parameters

- **searchStack** (*bool*) – should objects on the stack be searched first?
- **searchParents** (*bool*) – should parents be searched?

Raises
ValueError – if no solid is found

Return type
Union[Solid, Compound]

This function is very important for chains that are modifying a single parent object, most often a solid.

Most of the time, a chain defines or selects a solid, and then modifies it using workplanes or other operations.

Plugin Developers should make use of this method to find the solid that should be modified, if the plugin implements a unary operation, or if the operation will automatically merge its results with an object already on the stack.

first() → T

Return the first item on the stack

Returns
the first item on the stack.

Return type
a CQ object

Parameters
self (*T*) –

hLine(*distance: float*, *forConstruction: bool = False*) → T

Make a horizontal line from the current point the provided distance

Parameters

- **self** (*T*) –
- **distance** (*float*) –

(x) distance from current point

- **forConstruction** (*bool*) –

Returns
the Workplane object with the current point at the end of the new line

Return type
T

hLineTo(*xCoord: float*, *forConstruction: bool = False*) → T

Make a horizontal line from the current point to the provided x coordinate.

Useful if it is more convenient to specify the end location rather than distance, as in *hLine()*

Parameters

- **self** (*T*) –
- **xCoord** (*float*) – x coordinate for the end of the line
- **forConstruction** (*bool*) –

Returns
the Workplane object with the current point at the end of the new line

Return type
T

hole(*diameter: float*, *depth: Optional[float] = None*, *clean: bool = True*) → T

Makes a hole for each item on the stack.

Parameters

- **self** (*T*) –
- **diameter** (*float*) – the diameter of the hole
- **depth** (*float > 0 or None to drill thru the entire part.*) – the depth of the hole
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Return type
T

The surface of the hole is at the current workplane.

One hole is created for each item on the stack. A very common use case is to use a construction rectangle to define the centers of a set of holes, like so:

```
s = (
    Workplane()
    .box(2, 4, 0.5)
    .faces(">Z")
    .workplane()
    .rect(1.5, 3.5, forConstruction=True)
    .vertices()
    .hole(0.125, 0.25, 82, depth=None)
)
```

This sample creates a plate with a set of holes at the corners.

Plugin Note: this is one example of the power of plugins. CounterSunk holes are quite time consuming to create, but are quite easily defined by users.

see *cboreHole()* and *cskHole()* to make counterbores or countersinks

interpPlate(*surf_edges: Union[Sequence[Union[Tuple[float, float], Tuple[float, float, float], Vector]], Sequence[Union[Edge, Wire]], Workplane]*, *surf_pts: Sequence[Union[Tuple[float, float], Tuple[float, float, float], Vector]] = []*, *thickness: float = 0*, *combine: Union[bool, Literal['cut', 'a', 's']] = False*, *clean: bool = True*, *degree: int = 3*, *nbPtsOnCur: int = 15*, *nbIter: int = 2*, *anisotropy: bool = False*, *tol2d: float = 1e-05*, *tol3d: float = 0.0001*, *tolAng: float = 0.01*, *tolCurv: float = 0.1*, *maxDeg: int = 8*, *maxSegments: int = 9*) → T

Returns a plate surface that is 'thickness' thick, enclosed by 'surf_edge_pts' points, and going through 'surf_pts' points. Using pushPoints directly with interpPlate and combine=True, can be very resource intensive depending on the complexity of the shape. In this case set combine=False.

Parameters

- **self** (*T*) –
- **surf_edges** (*Union[Sequence[Union[Tuple[float, float], Tuple[float, float, float], Vector]], Sequence[Union[Edge, Wire]], Workplane]*) – list of [x,y,z] ordered coordinates or list of ordered or unordered edges, wires
- **surf_pts** (*Sequence[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – list of points (uses only edges if [])
- **thickness** (*float*) – value may be negative or positive depending on thickening direction (2D surface if 0)
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – should the results be combined with other solids on the stack (and each other)?
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **degree** (*int*) – >= 2
- **nbPtsOnCur** (*int*) – number of points on curve >= 15
- **nbIter** (*int*) – number of iterations >= 2
- **anisotropy** (*bool*) – = bool Anisotropy
- **tol2d** (*float*) – 2D tolerance
- **tol3d** (*float*) – 3D tolerance
- **tolAng** (*float*) – angular tolerance
- **tolCurv** (*float*) – tolerance for curvature
- **maxDeg** (*int*) – highest polynomial degree >= 2
- **maxSegments** (*int*) – greatest number of segments >= 2

Return type
T

intersect(*toIntersect: Union[*Workplane*, *Solid*, *Compound*]*, *clean: bool = True*, *tol: Optional[float] = None*) → T

Intersects the provided solid from the current solid.

Parameters

- **self** (*T*) –
- **toIntersect** (*Union[Workplane, Solid, Compound]*) – a solid object, or a Workplane object having a solid
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **tol** (*Optional[float]*) – tolerance value for fuzzy bool operation mode (default None)

Raises
ValueError – if there is no solid to intersect with in the chain

Returns
a Workplane object with the resulting object selected

Return type
T

`item`(*i: int*) → T

Return the ith item on the stack.

Return type
a CQ object

Parameters

- **self** (*T*) –
- **i** (*int*) –

`largestDimension`() → float

Finds the largest dimension in the stack.

Used internally to create thru features, this is how you can compute how long or wide a feature must be to make sure to cut through all of the material

Raises
ValueError – if no solids or compounds are found

Returns
A value representing the largest dimension of the first solid on the stack

Return type
float

`last`() → T

Return the last item on the stack.

Return type
a CQ object

Parameters
self (*T*) –

`line`(*xDist: float*, *yDist: float*, *forConstruction: bool = False*) → T

Make a line from the current point to the provided point, using dimensions relative to the current point

Parameters

- **self** (*T*) –
- **xDist** (*float*) – x distance from current point
- **yDist** (*float*) – y distance from current point
- **forConstruction** (*bool*) –

Returns
the workplane object with the current point at the end of the new line

Return type
T

see *lineTo()* if you want to use absolute coordinates to make a line instead.

`lineTo`(*x: float*, *y: float*, *forConstruction: bool = False*) → T

Make a line from the current point to the provided point

Parameters

- **self** (*T*) –
- **x** (*float*) – the x point, in workplane plane coordinates
- **y** (*float*) – the y point, in workplane plane coordinates
- **forConstruction** (*bool*) –

Returns
the Workplane object with the current point at the end of the new line

Return type
T

See *line()* if you want to use relative dimensions to make a line instead.

loft(*ruled: bool = False*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*) → T

Make a lofted solid, through the set of wires.

Parameters

- **self** (*T*) –
- **ruled** (*bool*) – When set to *True* connects each section linearly and without continuity
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or "a" to combine the resulting solid with parent solids if found, "cut" or "s" to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Returns
a Workplane object containing the created loft

Return type
T

mirror(*mirrorPlane: Union[Literal['XY', 'YX', 'XZ', 'ZX', 'YZ', 'ZY'], Tuple[float, float], Tuple[float, float, float], Vector, Face, Workplane] = 'XY'*, *basePointVector: Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]] = None*, *union: bool = False*) → T

Mirror a single CQ object.

Parameters

- **self** (*T*) –
- **mirrorPlane** (*string, one of "XY", "YX", "XZ", "ZX", "YZ", "ZY" the planes or the normal vector of the plane eg (1,0,0) or a Face object*) – the plane to mirror about
- **basePointVector** (*Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – the base point to mirror about (this is overwritten if a Face is passed)
- **union** (*bool*) – If true will perform a union operation on the mirrored object

Return type
T

mirrorX() → T

Mirror entities around the x axis of the workplane plane.

Returns
a new object with any free edges consolidated into as few wires as possible.

Parameters
self (*T*) –

Return type
T

All free edges are collected into a wire, and then the wire is mirrored, and finally joined into a new wire

Typically used to make creating wires with symmetry easier.

mirrorY() → T

Mirror entities around the y axis of the workplane plane.

Returns
a new object with any free edges consolidated into as few wires as possible.

Parameters
self (*T*) –

Return type
T

All free edges are collected into a wire, and then the wire is mirrored, and finally joined into a new wire

Typically used to make creating wires with symmetry easier. This line of code:

```
s = Workplane().lineTo(2, 2).threePointArc((3, 1), (2, 0)).mirrorX().extrude(0.
↪25)
```

Produces a flat, heart shaped object

move(*xDist: float = 0*, *yDist: float = 0*) → T

Move the specified distance from the current point, without drawing.

Parameters

- **self** (*T*) –
- **xDist** (*float, or none for zero*) – desired x distance, in local coordinates
- **yDist** (*float, or none for zero.*) – desired y distance, in local coordinates

Return type
T

Not to be confused with *center()*, which moves the center of the entire workplane, this method only moves the current point (and therefore does not affect objects already drawn).

See *moveTo()* to do the same thing but using absolute coordinates

moveTo(*x: float = 0*, *y: float = 0*) → T

Move to the specified point, without drawing.

Parameters

- **self** (*T*) –
- **x** (*float, or none for zero*) – desired x location, in local coordinates
- **y** (*float, or none for zero.*) – desired y location, in local coordinates

Return type
T

Not to be confused with *center()*, which moves the center of the entire workplane, this method only moves the current point (and therefore does not affect objects already drawn).

See *move()* to do the same thing but using relative dimensions

newObject(*objlist: Iterable[Union[*Vector, Location, Shape, Sketch*]]*) → T

Create a new workplane object from this one.

Overrides CQ.newObject, and should be used by extensions, plugins, and subclasses to create new objects.

Parameters

- **self** (*T*) –
- **objlist** (*a list of CAD primitives*) – new objects to put on the stack

Returns
a new Workplane object with the current workplane as a parent.

Return type
T

offset2D(*d: float*, *kind: Literal['arc', 'intersection', 'tangent'] = 'arc'*, *forConstruction: bool = False*) → T

Creates a 2D offset wire.

Parameters

- **self** (*T*) –
- **d** (*float*) – thickness. Negative thickness denotes offset to inside.
- **kind** (*Literal['arc', 'intersection', 'tangent']*) – offset kind. Use "arc" for rounded and "intersection" for sharp edges (default: "arc")
- **forConstruction** (*bool*) – Should the result be added to pending wires?

Returns
CQ object with resulting wire(s).

Return type
T

parametricCurve(*func: Callable[[float], Union[Tuple[float, float], Tuple[float, float, float],* Vector*]]*, *N: int = 400*, *start: float = 0*, *stop: float = 1*, *tol: float = 1e-06*, *minDeg: int = 1*, *maxDeg: int = 6*, *smoothing: Optional[Tuple[float, float, float]] = (1, 1, 1)*, *makeWire: bool = True*) → T

Create a spline curve approximating the provided function.

Parameters

- **self** (*T*) –
- **func** (*float --> (float,float,float)*) – function f(t) that will generate (x,y,z) pairs
- **N** (*int*) – number of points for discretization
- **start** (*float*) – starting value of the parameter t
- **stop** (*float*) – final value of the parameter t
- **tol** (*float*) – tolerance of the algorithm (default: 1e-6)
- **minDeg** (*int*) – minimum spline degree (default: 1)
- **maxDeg** (*int*) – maximum spline degree (default: 6)
- **smoothing** (*Optional[Tuple[float, float, float]]*) – optional parameters for the variational smoothing algorithm (default: (1,1,1))

- **makeWire** (*bool*) – convert the resulting spline edge to a wire

Returns
a Workplane object with the current point unchanged

Return type
T

parametricSurface(*func: Callable[[float, float], Union[Tuple[float, float], Tuple[float, float, float], Vector]]*, *N: int = 20*, *start: float = 0*, *stop: float = 1*, *tol: float = 0.01*, *minDeg: int = 1*, *maxDeg: int = 6*, *smoothing: Optional[Tuple[float, float, float]] = (1, 1, 1)*) → T

Create a spline surface approximating the provided function.

Parameters

- **self** (*T*) –
- **func** (*(float,float) --> (float,float,float)*) – function f(u,v) that will generate (x,y,z) pairs
- **N** (*int*) – number of points for discretization in one direction
- **start** (*float*) – starting value of the parameters u,v
- **stop** (*float*) – final value of the parameters u,v
- **tol** (*float*) – tolerance used by the approximation algorithm (default: 1e-3)
- **minDeg** (*int*) – minimum spline degree (default: 1)
- **maxDeg** (*int*) – maximum spline degree (default: 3)
- **smoothing** (*Optional[Tuple[float, float, float]]*) – optional parameters for the variational smoothing algorithm (default: (1,1,1))

Returns
a Workplane object with the current point unchanged

Return type
T

This method might be unstable and may require tuning of the tol parameter.

placeSketch(**sketches: Sketch*) → T

Place the provided sketch(es) based on the current items on the stack.

Returns
Workplane object with the sketch added.

Parameters

- **self** (*T*) –
- **sketches** (Sketch) –

Return type
T

polarArray(*radius: float*, *startAngle: float*, *angle: float*, *count: int*, *fill: bool = True*, *rotate: bool = True*) → T

Creates a polar array of points and pushes them onto the stack. The zero degree reference angle is located along the local X-axis.

Parameters

- **self** (*T*) –

- **radius** (*float*) – Radius of the array.
- **startAngle** (*float*) – Starting angle (degrees) of array. Zero degrees is situated along the local X-axis.
- **angle** (*float*) – The angle (degrees) to fill with elements. A positive value will fill in the counter-clockwise direction. If fill is False, angle is the angle between elements.
- **count** (*int*) – Number of elements in array. (count >= 1)
- **fill** (*bool*) – Interpret the angle as total if True (default: True).
- **rotate** (*bool*) – Rotate every item (default: True).

Return type
T

polarLine(*distance: float*, *angle: float*, *forConstruction: bool = False*) → T

Make a line of the given length, at the given angle from the current point

Parameters

- **self** (*T*) –
- **distance** (*float*) – distance of the end of the line from the current point
- **angle** (*float*) – angle of the vector to the end of the line with the x-axis
- **forConstruction** (*bool*) –

Returns
the Workplane object with the current point at the end of the new line

Return type
T

polarLineTo(*distance: float*, *angle: float*, *forConstruction: bool = False*) → T

Make a line from the current point to the given polar coordinates

Useful if it is more convenient to specify the end location rather than the distance and angle from the current point

Parameters

- **self** (*T*) –
- **distance** (*float*) – distance of the end of the line from the origin
- **angle** (*float*) – angle of the vector to the end of the line with the x-axis
- **forConstruction** (*bool*) –

Returns
the Workplane object with the current point at the end of the new line

Return type
T

polygon(*nSides: int*, *diameter: float*, *forConstruction: bool = False*, *circumscribed: bool = False*) → T

Make a polygon for each item on the stack.

By default, each polygon is created by inscribing it in a circle of the specified diameter, such that the first vertex is oriented in the x direction. Alternatively, each polygon can be created by circumscribing it around a circle of the specified diameter, such that the midpoint of the first edge is oriented in the x direction. Circumscribed polygons are thus rotated by pi/nSides radians relative to the inscribed polygon.

This ensures the extent of the polygon along the positive x-axis is always known. This has the advantage of not requiring additional formulae for purposes such as tiling on the x-axis (at least for even sided polygons).

Parameters

- **self** (*T*) –
- **nSides** (*int*) – number of sides, must be >= 3
- **diameter** (*float*) – the diameter of the circle for constructing the polygon
- **circumscribed** (*true to create the polygon by circumscribing it about a circle, false to create the polygon by inscribing it in a circle*) – circumscribe the polygon about a circle
- **forConstruction** (*bool*) –

Returns
a polygon wire

Return type
T

polyline(*listOfXYTuple: Sequence[Union[Tuple[float, float], Tuple[float, float, float],* Vector*]]*, *forConstruction: bool = False*, *includeCurrent: bool = False*) → T

Create a polyline from a list of points

Parameters

- **self** (*T*) –
- **listOfXYTuple** (*Sequence[Union[Tuple[float, float], Tuple[float, float, float],* Vector*]]*) – a list of points in Workplane coordinates (2D or 3D)
- **forConstruction** (*true if the edges are for reference, false if they are for creating geometry part geometry*) – whether or not the edges are used for reference
- **includeCurrent** (*bool*) – use current point as a starting point of the polyline

Returns
a new CQ object with a list of edges on the stack

Return type
T

NOTE most commonly, the resulting wire should be closed.

pushPoints(*pntList: Iterable[Union[Tuple[float, float], Tuple[float, float, float],* Vector, Location*]]*) → T

Pushes a list of points onto the stack as vertices. The points are in the 2D coordinate space of the workplane face

Parameters

- **self** (*T*) –
- **pntList** (list of 2-tuples, in *local* coordinates) – a list of points to push onto the stack

Returns
a new workplane with the desired points on the stack.

Return type
T

A common use is to provide a list of points for a subsequent operation, such as creating circles or holes. This example creates a cube, and then drills three holes through it, based on three points:

```
s = (
    Workplane()
    .box(1, 1, 1)
    .faces(">Z")
    .workplane()
    .pushPoints([(-0.3, 0.3), (0.3, 0.3), (0, 0)])
)
body = s.circle(0.05).cutThruAll()
```

Here the circle function operates on all three points, and is then extruded to create three holes. See *circle()* for how it works.

radiusArc(*endPoint: Union[Tuple[float, float], Tuple[float, float, float],* Vector*]*, *radius: float*, *forConstruction: bool = False*) → T

Draw an arc from the current point to endPoint with an arc defined by the radius.

Parameters

- **self** (*T*) –
- **endPoint** (*2-tuple, in workplane coordinates*) – end point for the arc
- **radius** (*float, the radius of the arc between start point and end point.*) – the radius of the arc
- **forConstruction** (*bool*) –

Returns
a workplane with the current point at the end of the arc

Return type
T

Given that a closed contour is drawn clockwise; A positive radius means convex arc and negative radius means concave arc.

rarray(*xSpacing: float*, *ySpacing: float*, *xCount: int*, *yCount: int*, *center: Union[bool, Tuple[bool, bool]] = True*) → T

Creates an array of points and pushes them onto the stack. If you want to position the array at another point, create another workplane that is shifted to the position you would like to use as a reference

Parameters

- **self** (*T*) –
- **xSpacing** (*float*) – spacing between points in the x direction (must be > 0)
- **ySpacing** (*float*) – spacing between points in the y direction (must be > 0)
- **xCount** (*int*) – number of points (> 0)
- **yCount** (*int*) – number of points (> 0)
- **center** (*Union[bool, Tuple[bool, bool]]*) – If True, the array will be centered around the workplane center. If False, the lower corner will be on the reference point and the array will extend in the positive x and y directions. Can also use a 2-tuple to specify centering along each axis.

Return type
T

rect(*xLen: float*, *yLen: float*, *centered: Union[bool, Tuple[bool, bool]] = True*, *forConstruction: bool = False*) → T

Make a rectangle for each item on the stack.

Parameters

- **self** (*T*) –
- **xLen** (*float*) – length in the x direction (in workplane coordinates)
- **yLen** (*float*) – length in the y direction (in workplane coordinates)
- **centered** (*Union[bool, Tuple[bool, bool]]*) – If True, the rectangle will be centered around the reference point. If False, the corner of the rectangle will be on the reference point and it will extend in the positive x and y directions. Can also use a 2-tuple to specify centering along each axis.
- **forConstruction** (*true if the wires are for reference, false if they are creating part geometry*) – should the new wires be reference geometry only?

Returns
a new CQ object with the created wires on the stack

Return type
T

A common use case is to use a for-construction rectangle to define the centers of a hole pattern:

```
s = Workplane().rect(4.0, 4.0, forConstruction=True).vertices().circle(0.25)
```

Creates 4 circles at the corners of a square centered on the origin.

Negative values for xLen and yLen are permitted, although they only have an effect when centered is False.

Future Enhancements:

- project points not in the workplane plane onto the workplane plane

revolve(*angleDegrees: float = 360.0*, *axisStart: Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]] = None*, *axisEnd: Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]] = None*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*) → T

Use all un-revolved wires in the parent chain to create a solid.

Parameters

- **self** (*T*) –
- **angleDegrees** (*float, anything less than 360 degrees will leave the shape open*) – the angle to revolve through.
- **axisStart** (*Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – the start point of the axis of rotation
- **axisEnd** (*Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – the end point of the axis of rotation
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or “a” to combine the resulting solid with parent solids if found, “cut” or “s” to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Returns
a CQ object with the resulting solid selected.

Return type
T

The returned object is always a CQ object, and depends on whether combine is True, and whether a context solid is already defined:

- if combine is False, the new value is pushed onto the stack.
- if combine is true, the value is combined with the context solid if it exists, and the resulting solid becomes the new context solid.

Note: Keep in mind that *axisStart* and *axisEnd* are defined relative to the current Workplane center position. So if for example you want to revolve a circle centered at (10,0,0) around the Y axis, be sure to either `move()` (or `moveTo()`) the current Workplane position or specify *axisStart* and *axisEnd* with the correct vector position. In this example (0,0,0), (0,1,0) as axis coords would fail.

---

rotate(*axisStartPoint: Union[Tuple[float, float], Tuple[float, float, float], Vector]*, *axisEndPoint: Union[Tuple[float, float], Tuple[float, float, float], Vector]*, *angleDegrees: float*) → T

Returns a copy of all of the items on the stack rotated through and angle around the axis of rotation.

Parameters

- **self** (*T*) –
- **axisStartPoint** (*a 3-tuple of floats*) – The first point of the axis of rotation
- **axisEndPoint** (*a 3-tuple of floats*) – The second point of the axis of rotation
- **angleDegrees** (*float*) – the rotation angle, in degrees

Returns
a CQ object

Return type
T

rotateAboutCenter(*axisEndPoint: Union[Tuple[float, float], Tuple[float, float, float], Vector]*, *angleDegrees: float*) → T

Rotates all items on the stack by the specified angle, about the specified axis

The center of rotation is a vector starting at the center of the object on the stack, and ended at the specified point.

Parameters

- **self** (*T*) –
- **axisEndPoint** (*a three-tuple in global coordinates*) – the second point of axis of rotation
- **angleDegrees** (*float*) – the rotation angle, in degrees

Returns
a CQ object, with all items rotated.

Return type
T

WARNING: This version returns the same CQ object instead of a new one– the old object is not accessible.

Future Enhancements:

- A version of this method that returns a transformed copy, rather than modifying the originals

- This method doesn't expose a very good interface, because the axis of rotation could be inconsistent between multiple objects. This is because the beginning of the axis is variable, while the end is fixed. This is fine when operating on one object, but is not cool for multiple.

sagittaArc(*endPoint: Union[Tuple[float, float], Tuple[float, float, float], Vector]*, *sag: float*, *forConstruction: bool = False*) → T

Draw an arc from the current point to endPoint with an arc defined by the sag (sagitta).

Parameters

- **self** (*T*) –
- **endPoint** (*2-tuple, in workplane coordinates*) – end point for the arc
- **sag** (*float, perpendicular distance from arc center to arc baseline.*) – the sagitta of the arc
- **forConstruction** (*bool*) –

Returns
a workplane with the current point at the end of the arc

Return type
T

The sagitta is the distance from the center of the arc to the arc base. Given that a closed contour is drawn clockwise; A positive sagitta means convex arc and negative sagitta means concave arc. See https://en.wikipedia.org/wiki/Sagitta_(geometry) for more information.

section(*height: float = 0.0*) → T

Slices current solid at the given height.

Parameters

- **self** (*T*) –
- **height** (*float*) – height to slice at (default: 0)

Raises
ValueError – if no solids or compounds are found

Returns
a CQ object with the resulting face(s).

Return type
T

shell(*thickness: float*, *kind: Literal['arc', 'intersection'] = 'arc'*) → T

Remove the selected faces to create a shell of the specified thickness.

To shell, first create a solid, and *in the same chain* select the faces you wish to remove.

Parameters

- **self** (*T*) –
- **thickness** (*float*) – thickness of the desired shell. Negative values shell inwards, positive values shell outwards.
- **kind** (*Literal['arc', 'intersection']*) – kind of join, arc or intersection (default: arc).

Raises
ValueError – if the current stack contains objects that are not faces of a solid further up in the chain.

Returns
a CQ object with the resulting shelled solid selected.

Return type
T

This example will create a hollowed out unit cube, where the top most face is open, and all other walls are 0.2 units thick:

```
Workplane().box(1, 1, 1).faces("+Z").shell(0.2)
```

You can also select multiple faces at once. Here is an example that creates a three-walled corner, by removing three faces of a cube:

```
Workplane().box(10, 10, 10).faces(">Z or >X or <Y").shell(1)
```

Note: When sharp edges are shelled inwards, they remain sharp corners, but **outward** shells are automatically filleted (unless kind="intersection"), because an outward offset from a corner generates a radius.

shells(*selector: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select the shells of objects on the stack, optionally filtering the selection. If there are multiple objects on the stack, the shells of all objects are collected and a list of all the distinct shells is returned.

Parameters

- **self** (*T*) –
- **selector** (*Optional[Union[str, Selector]]*) – optional Selector object, or string selector expression (see *StringSyntaxSelector*)
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a CQ object whose stack contains all of the *distinct* shells of *all* objects on the current stack, filtered by the provided selector.

Return type
T

If there are no shells for any objects on the current stack, an empty CQ object is returned

Most solids will have a single shell, which represents the outer surface. A shell will typically be composed of multiple faces.

siblings(*kind: Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*, *level: int = 1*, *tag: Optional[str] = None*) → T

Select topological siblings.

Parameters

- **self** (*T*) –
- **kind** (*Literal['Vertex', 'Edge', 'Wire', 'Face', 'Shell', 'Solid', 'CompSolid', 'Compound']*) – kind of linking element, e.g. "Vertex" or "Edge"
- **level** (*int*) – level of relation - how many elements of kind are in the link
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a Workplane object whose stack contains selected siblings.

Return type
T

size() → int

Return the number of objects currently on the stack

Return type
int

sketch() → *Sketch*

Initialize and return a sketch

Returns
Sketch object with the current workplane as a parent.

Parameters
self (*T*) –

Return type
Sketch

slot2D(*length: float*, *diameter: float*, *angle: float = 0*) → T

Creates a rounded slot for each point on the stack.

Parameters

- **self** (*T*) –
- **diameter** (*float*) – desired diameter, or width, of slot
- **length** (*float*) – desired end to end length of slot
- **angle** (*float*) – angle of slot in degrees, with 0 being along x-axis

Returns
a new CQ object with the created wires on the stack

Return type
T

Can be used to create arrays of slots, such as in cooling applications:

```
Workplane().box(10, 25, 1).rarray(1, 2, 1, 10).slot2D(8, 1, 0).cutThruAll()
```

solids(*selector: Optional[Union[str,* Selector*]] = None*, *tag: Optional[str] = None*) → T

Select the solids of objects on the stack, optionally filtering the selection. If there are multiple objects on the stack, the solids of all objects are collected and a list of all the distinct solids is returned.

Parameters

- **self** (*T*) –
- **selector** (*Optional[Union[str,* Selector*]]*) – optional Selector object, or string selector expression (see *StringSyntaxSelector*)
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a CQ object whose stack contains all of the *distinct* solids of *all* objects on the current stack, filtered by the provided selector.

Return type
T

If there are no solids for any objects on the current stack, an empty CQ object is returned

The typical use is to select a single object on the stack. For example:

```
Workplane().box(1, 1, 1).solids().size()
```

returns 1, because a cube consists of one solid.

It is possible for a single CQ object (or even a single CAD primitive) to contain multiple solids.

sphere(*radius: float*, *direct: Union[Tuple[float, float], Tuple[float, float, float], Vector] = (0, 0, 1)*, *angle1: float = - 90*, *angle2: float = 90*, *angle3: float = 360*, *centered: Union[bool, Tuple[bool, bool, bool]] = True*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*) → T

Returns a 3D sphere with the specified radius for each point on the stack.

Parameters

- **self** (*T*) –
- **radius** (*float*) – The radius of the sphere
- **direct** (*A three-tuple*) – The direction axis for the creation of the sphere
- **angle1** (*float > 0*) – The first angle to sweep the sphere arc through
- **angle2** (*float > 0*) – The second angle to sweep the sphere arc through
- **angle3** (*float > 0*) – The third angle to sweep the sphere arc through
- **centered** (*Union[bool, Tuple[bool, bool, bool]]*) – If True, the sphere will be centered around the reference point. If False, the corner of a bounding box around the sphere will be on the reference point and it will extend in the positive x, y and z directions. Can also use a 3-tuple to specify centering along each axis.
- **combine** (*true to combine shapes, false otherwise*) – Whether the results should be combined with other solids on the stack (and each other)
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape

Returns
A sphere object for each point on the stack

Return type
T

One sphere is created for each item on the current stack. If no items are on the stack, one box using the current workplane center is created.

If combine is true, the result will be a single object on the stack. If a solid was found in the chain, the result is that solid with all spheres produced fused onto it otherwise, the result is the combination of all the produced spheres.

If combine is false, the result will be a list of the spheres produced.

spline(*listOfXYTuple: Iterable[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*, *tangents: Optional[Sequence[Union[Tuple[float, float], Tuple[float, float, float], Vector]]] = None*, *periodic: bool = False*, *parameters: Optional[Sequence[float]] = None*, *scale: bool = True*, *tol: Optional[float] = None*, *forConstruction: bool = False*, *includeCurrent: bool = False*, *makeWire: bool = False*) → T

Create a spline interpolated through the provided points (2D or 3D).

Parameters

- **self** (*T*) –

- **listOfXYTuple** (*Iterable[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – points to interpolate through
- **tangents** (*Optional[Sequence[Union[Tuple[float, float], Tuple[float, float, float], Vector]]]*) – vectors specifying the direction of the tangent to the curve at each of the specified interpolation points.

  If only 2 tangents are given, they will be used as the initial and final tangent.

  If some tangents are not specified (i.e., are None), no tangent constraint will be applied to the corresponding interpolation point.

  The spline will be C2 continuous at the interpolation points where no tangent constraint is specified, and C1 continuous at the points where a tangent constraint is specified.
- **periodic** (*bool*) – creation of periodic curves
- **parameters** (*Optional[Sequence[float]]*) – the value of the parameter at each interpolation point. (The interpolated curve is represented as a vector-valued function of a scalar parameter.)

  If periodic == True, then len(parameters) must be len(interpolation points) + 1, otherwise len(parameters) must be equal to len(interpolation points).
- **scale** (*bool*) – whether to scale the specified tangent vectors before interpolating.

  Each tangent is scaled, so it's length is equal to the derivative of the Lagrange interpolated curve.

  I.e., set this to True, if you want to use only the direction of the tangent vectors specified by `tangents`, but not their magnitude.
- **tol** (*Optional[float]*) – tolerance of the algorithm (consult OCC documentation)

  Used to check that the specified points are not too close to each other, and that tangent vectors are not too short. (In either case interpolation may fail.)

  Set to None to use the default tolerance.
- **includeCurrent** (*bool*) – use current point as a starting point of the curve
- **makeWire** (*bool*) – convert the resulting spline edge to a wire
- **forConstruction** (*bool*) –

Returns
a Workplane object with the current point at the end of the spline

Return type
T

The spline will begin at the current point, and end with the last point in the XY tuple list.

This example creates a block with a spline for one side:

```
s = Workplane(Plane.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),
```

(continues on next page)

(continued from previous page)

```
    (0, 1.0),
]
r = s.lineTo(3.0, 0).lineTo(3.0, 1.0).spline(sPnts).close()
r = r.extrude(0.5)
```

WARNING It is fairly easy to create a list of points that cannot be correctly interpreted as a spline.

splineApprox(*points: Iterable[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*, *tol: Optional[float] = 1e-06*, *minDeg: int = 1*, *maxDeg: int = 6*, *smoothing: Optional[Tuple[float, float, float]] = (1, 1, 1)*, *forConstruction: bool = False*, *includeCurrent: bool = False*, *makeWire: bool = False*) → T

Create a spline interpolated through the provided points (2D or 3D).

Parameters

- **self** (*T*) –
- **points** (*Iterable[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – points to interpolate through
- **tol** (*Optional[float]*) – tolerance of the algorithm (default: 1e-6)
- **minDeg** (*int*) – minimum spline degree (default: 1)
- **maxDeg** (*int*) – maximum spline degree (default: 6)
- **smoothing** (*Optional[Tuple[float, float, float]]*) – optional parameters for the variational smoothing algorithm (default: (1,1,1))
- **includeCurrent** (*bool*) – use current point as a starting point of the curve
- **makeWire** (*bool*) – convert the resulting spline edge to a wire
- **forConstruction** (*bool*) –

Returns

a Workplane object with the current point at the end of the spline

Return type

T

WARNING for advanced users.

split(*keepTop: bool = False*, *keepBottom: bool = False*) → T

split(*splitter: Union[T, Shape]*) → T

Splits a solid on the stack into two parts, optionally keeping the separate parts.

Parameters

- **self** –
- **keepTop** (*bool*) – True to keep the top, False or None to discard it
- **keepBottom** (*bool*) – True to keep the bottom, False or None to discard it

Raises

- **ValueError** – if keepTop and keepBottom are both false.
- **ValueError** – if there is no solid in the current stack or parent chain

Returns

CQ object with the desired objects on the stack.

The most common operation splits a solid and keeps one half. This sample creates a split bushing:

```
# drill a hole in the side
c = Workplane().box(1, 1, 1).faces(">Z").workplane().circle(0.25).cutThruAll()

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

sweep(*path: Union[*Workplane*, *Wire*, *Edge*]*, *multisection: bool = False*, *sweepAlongWires: Optional[bool] = None*, *makeSolid: bool = True*, *isFrenet: bool = False*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*, *transition: Literal['right', 'round', 'transformed'] = 'right'*, *normal: Optional[Union[Tuple[float, float], Tuple[float, float, float],* Vector*]] = None*, *auxSpine: Optional[*Workplane*] = None*) → T

Use all un-extruded wires in the parent chain to create a swept solid.

Parameters

- **self** (*T*) –
- **path** (*Union[Workplane, Wire, Edge]*) – A wire along which the pending wires will be swept
- **multiSection** – False to create multiple swept from wires on the chain along path. True to create only one solid swept along path with shape following the list of wires on the chain
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or "a" to combine the resulting solid with parent solids if found, "cut" or "s" to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **transition** (*Literal['right', 'round', 'transformed']*) – handling of profile orientation at C1 path discontinuities. Possible values are {'transformed','round', 'right'} (default: 'right').
- **normal** (*Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – optional fixed normal for extrusion
- **auxSpine** (*Optional[Workplane]*) – a wire defining the binormal along the extrusion path
- **multisection** (*bool*) –
- **sweepAlongWires** (*Optional[bool]*) –
- **makeSolid** (*bool*) –
- **isFrenet** (*bool*) –

Returns
a CQ object with the resulting solid selected.

Return type
T

tag(*name: str*) → T

Tags the current CQ object for later reference.

Parameters

- **self** (*T*) –
- **name** (*str*) – the name to tag this object with

Returns
self, a CQ object with tag applied

Return type
T

tangentArcPoint(*endpoint: Union[Tuple[float, float], Tuple[float, float, float],* Vector*], forConstruction: bool = False, relative: bool = True*) → T

Draw an arc as a tangent from the end of the current edge to endpoint.

Parameters

- **self** (*T*) –
- **endpoint** (*2-tuple, 3-tuple or* Vector) – point for the arc to end at
- **relative** (*bool*) – True if endpoint is specified relative to the current point, False if endpoint is in workplane coordinates
- **forConstruction** (*bool*) –

Returns
a Workplane object with an arc on the stack

Return type
T

Requires the the current first object on the stack is an Edge, as would be the case after a lineTo operation or similar.

text(*txt: str, fontsize: float, distance: float, cut: bool = True, combine: Union[bool, Literal['cut', 'a', 's']] = False, clean: bool = True, font: str = 'Arial', fontPath: Optional[str] = None, kind: Literal['regular', 'bold', 'italic'] = 'regular', halign: Literal['center', 'left', 'right'] = 'center', valign: Literal['center', 'top', 'bottom'] = 'center'*) → T

Returns a 3D text.

Parameters

- **self** (*T*) –
- **txt** (*str*) – text to be rendered
- **fontsize** (*float*) – size of the font in model units
- **distance** (*float, negative means opposite the normal direction*) – the distance to extrude or cut, normal to the workplane plane
- **cut** (*bool*) – True to cut the resulting solid from the parent solids if found
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or "a" to combine the resulting solid with parent solids if found, "cut" or "s" to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **font** (*str*) – font name
- **fontPath** (*Optional[str]*) – path to font file
- **kind** (*Literal['regular', 'bold', 'italic']*) – font type
- **halign** (*Literal['center', 'left', 'right']*) – horizontal alignment
- **valign** (*Literal['center', 'top', 'bottom']*) – vertical alignment

Returns
a CQ object with the resulting solid selected

Return type
T

The returned object is always a Workplane object, and depends on whether combine is True, and whether a context solid is already defined:

- if combine is False, the new value is pushed onto the stack.
- if combine is true, the value is combined with the context solid if it exists, and the resulting solid becomes the new context solid.

Examples:

```
cq.Workplane().text("CadQuery", 5, 1)
```

Specify the font (name), and kind to use an installed system font:

```
cq.Workplane().text("CadQuery", 5, 1, font="Liberation Sans Narrow", kind=
↪"italic")
```

Specify fontPath to use a font from a given file:

```
cq.Workplane().text("CadQuery", 5, 1, fontPath="/opt/fonts/texgyrecursor-bold.
↪otf")
```

Cutting text into a solid:

```
cq.Workplane().box(8, 8, 8).faces(">Z").workplane().text("Z", 5, -1.0)
```

threePointArc(*point1: Union[Tuple[float, float], Tuple[float, float, float], Vector]*, *point2: Union[Tuple[float, float], Tuple[float, float, float], Vector]*, *forConstruction: bool = False*) → T

Draw an arc from the current point, through point1, and ending at point2

Parameters

- **self** (*T*) –
- **point1** (*2-tuple, in workplane coordinates*) – point to draw through
- **point2** (*2-tuple, in workplane coordinates*) – end point for the arc
- **forConstruction** (*bool*) –

Returns
a workplane with the current point at the end of the arc

Return type
T

Future Enhancements:
provide a version that allows an arc using relative measures provide a centerpoint arc provide tangent arcs

toOCC() → Any

Directly returns the wrapped OCCT object.

Returns
The wrapped OCCT object

Return type
TopoDS_Shape or a subclass

toPending() → T

Adds wires/edges to pendingWires/pendingEdges.

Returns
same CQ object with updated context.

Parameters
self (*T*) –

Return type
T

toSvg(*opts: Any = None*) → str

Returns svg text that represents the first item on the stack.

for testing purposes.

Parameters
opts (*dictionary, width and height*) – svg formatting options

Returns
a string that contains SVG that represents this item.

Return type
str

transformed(*rotate: Union[Tuple[float, float], Tuple[float, float, float], Vector] = (0, 0, 0)*, *offset: Union[Tuple[float, float], Tuple[float, float, float], Vector] = (0, 0, 0)*) → T

Create a new workplane based on the current one. The origin of the new plane is located at the existing origin+offset vector, where offset is given in coordinates local to the current plane The new plane is rotated through the angles specified by the components of the rotation vector.

Parameters

- **self** (*T*) –
- **rotate** (*Union[Tuple[float, float], Tuple[float, float, float], Vector]*) – 3-tuple of angles to rotate, in degrees relative to work plane coordinates
- **offset** (*Union[Tuple[float, float], Tuple[float, float, float], Vector]*) – 3-tuple to offset the new plane, in local work plane coordinates

Returns
a new work plane, transformed as requested

Return type
T

translate(*vec: Union[Tuple[float, float], Tuple[float, float, float], Vector]*) → T

Returns a copy of all of the items on the stack moved by the specified translation vector.

Parameters

- **self** (*T*) –
- **tupleDistance** (*a 3-tuple of float*) – distance to move, in global coordinates

- **vec** (*Union[Tuple[float, float], Tuple[float, float, float], Vector]*) –

Returns
a CQ object

Return type
T

twistExtrude(*distance: float*, *angleDegrees: float*, *combine: Union[bool, Literal['cut', 'a', 's']] = True*, *clean: bool = True*) → T

Extrudes a wire in the direction normal to the plane, but also twists by the specified angle over the length of the extrusion.

The center point of the rotation will be the center of the workplane.

See extrude for more details, since this method is the same except for the the addition of the angle. In fact, if angle=0, the result is the same as a linear extrude.

NOTE This method can create complex calculations, so be careful using it with complex geometries

Parameters

- **self** (*T*) –
- **distance** (*float*) – the distance to extrude normal to the workplane
- **angle** – angle (in degrees) to rotate through the extrusion
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – True or "a" to combine the resulting solid with parent solids if found, "cut" or "s" to remove the resulting solid from the parent solids if found. False to keep the resulting solid separated from the parent solids.
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape
- **angleDegrees** (*float*) –

Returns
a CQ object with the resulting solid selected.

Return type
T

union(*toUnion: Optional[Union[*Workplane*, *Solid*, *Compound*]] = None*, *clean: bool = True*, *glue: bool = False*, *tol: Optional[float] = None*) → T

Unions all of the items on the stack of toUnion with the current solid. If there is no current solid, the items in toUnion are unioned together.

Parameters

- **self** (*T*) –
- **toUnion** (*Optional[Union[Workplane, Solid, Compound]]*) – a solid object, or a Workplane object having a solid
- **clean** (*bool*) – call *clean()* afterwards to have a clean shape (default True)
- **glue** (*bool*) – use a faster gluing mode for non-overlapping shapes (default False)
- **tol** (*Optional[float]*) – tolerance value for fuzzy bool operation mode (default None)

Raises
ValueError if there is no solid to add to in the chain

Returns
a Workplane object with the resulting object selected

Return type
T

vLine(*distance: float*, *forConstruction: bool = False*) → T

Make a vertical line from the current point the provided distance

Parameters

- **self** (*T*) –
- **distance** (*float*) –

(y) distance from current point

- **forConstruction** (*bool*) –

Returns
the Workplane object with the current point at the end of the new line

Return type
T

vLineTo(*yCoord: float*, *forConstruction: bool = False*) → T

Make a vertical line from the current point to the provided y coordinate.

Useful if it is more convenient to specify the end location rather than distance, as in *vLine()*

Parameters

- **self** (*T*) –
- **yCoord** (*float*) – y coordinate for the end of the line
- **forConstruction** (*bool*) –

Returns
the Workplane object with the current point at the end of the new line

Return type
T

val() → Union[*Vector*, *Location*, *Shape*, *Sketch*]

Return the first value on the stack. If no value is present, current plane origin is returned.

Returns
the first value on the stack.

Return type
A CAD primitive

vals() → List[Union[*Vector*, *Location*, *Shape*, *Sketch*]]

get the values in the current list

Return type
list of occ_impl objects

Returns
the values of the objects on the stack.

Contrast with *all()*, which returns CQ objects for all of the items on the stack

vertices(*selector: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select the vertices of objects on the stack, optionally filtering the selection. If there are multiple objects on the stack, the vertices of all objects are collected and a list of all the distinct vertices is returned.

Parameters

- **self** (*T*) –
- **selector** (*Optional[Union[str, Selector]]*) – optional Selector object, or string selector expression (see *StringSyntaxSelector*)
- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns

a CQ object whose stack contains the *distinct* vertices of *all* objects on the current stack, after being filtered by the selector, if provided

Return type

T

If there are no vertices for any objects on the current stack, an empty CQ object is returned

The typical use is to select the vertices of a single object on the stack. For example:

```
Workplane().box(1, 1, 1).faces("+Z").vertices().size()
```

returns 4, because the topmost face of a cube will contain four vertices. While this:

```
Workplane().box(1, 1, 1).faces().vertices().size()
```

returns 8, because a cube has a total of 8 vertices

Note Circles are peculiar, they have a single vertex at the center!

wedge(*dx: float, dy: float, dz: float, xmin: float, zmin: float, xmax: float, zmax: float, pnt: ~typing.Union[~typing.Tuple[float, float], ~typing.Tuple[float, float, float], ~cadquery.occ_impl.geom.Vector] = Vector: (0.0, 0.0, 0.0), dir: ~typing.Union[~typing.Tuple[float, float], ~typing.Tuple[float, float, float], ~cadquery.occ_impl.geom.Vector] = Vector: (0.0, 0.0, 1.0), centered: ~typing.Union[bool, ~typing.Tuple[bool, bool, bool]] = True, combine: ~typing.Union[bool, ~typing.Literal['cut', 'a', 's']] = True, clean: bool = True*) → T

Returns a 3D wedge with the specified dimensions for each point on the stack.

Parameters

- **self** (*T*) –
- **dx** (*float*) – Distance along the X axis
- **dy** (*float*) – Distance along the Y axis
- **dz** (*float*) – Distance along the Z axis
- **xmin** (*float*) – The minimum X location
- **zmin** (*float*) – The minimum Z location
- **xmax** (*float*) – The maximum X location
- **zmax** (*float*) – The maximum Z location
- **pnt** (*Union[Tuple[float, float], Tuple[float, float, float], Vector]*) – A vector (or tuple) for the origin of the direction for the wedge

- **dir** (*Union[Tuple[float, float], Tuple[float, float, float], Vector]*) – The direction vector (or tuple) for the major axis of the wedge
- **centered** (*Union[bool, Tuple[bool, bool, bool]]*) – If True, the wedge will be centered around the reference point. If False, the corner of the wedge will be on the reference point and it will extend in the positive x, y and z directions. Can also use a 3-tuple to specify centering along each axis.
- **combine** (*Union[bool, Literal['cut', 'a', 's']]*) – Whether the results should be combined with other solids on the stack (and each other)
- **clean** (*bool*) – True to attempt to have the kernel clean up the geometry, False otherwise

Returns
A wedge object for each point on the stack

Return type
T

One wedge is created for each item on the current stack. If no items are on the stack, one wedge using the current workplane center is created.

If combine is True, the result will be a single object on the stack. If a solid was found in the chain, the result is that solid with all wedges produced fused onto it otherwise, the result is the combination of all the produced wedges.

If combine is False, the result will be a list of the wedges produced.

wire(*forConstruction: bool = False*) → T

Returns a CQ object with all pending edges connected into a wire.

All edges on the stack that can be combined will be combined into a single wire object, and other objects will remain on the stack unmodified. If there are no pending edges, this method will just return self.

Parameters

- **self** (*T*) –
- **forConstruction** (*bool*) – whether the wire should be used to make a solid, or if it is just for reference

Return type
T

This method is primarily of use to plugin developers making utilities for 2D construction. This method should be called when a user operation implies that 2D construction is finished, and we are ready to begin working in 3d.

SEE '2D construction concepts' for a more detailed explanation of how CadQuery handles edges, wires, etc.

Any non edges will still remain.

wires(*selector: Optional[Union[str, Selector]] = None*, *tag: Optional[str] = None*) → T

Select the wires of objects on the stack, optionally filtering the selection. If there are multiple objects on the stack, the wires of all objects are collected and a list of all the distinct wires is returned.

Parameters

- **self** (*T*) –
- **selector** (*Optional[Union[str, Selector]]*) – optional Selector object, or string selector expression (see *StringSyntaxSelector*)

- **tag** (*Optional[str]*) – if set, search the tagged object instead of self

Returns
a CQ object whose stack contains all of the *distinct* wires of *all* objects on the current stack, filtered by the provided selector.

Return type
T

If there are no wires for any objects on the current stack, an empty CQ object is returned

The typical use is to select the wires of a single object on the stack. For example:

```
Workplane().box(1, 1, 1).faces("+Z").wires().size()
```

returns 1, because a face typically only has one outer wire

workplane(*offset: float = 0.0*, *invert: bool = False*, *centerOption: Literal['CenterOfMass', 'ProjectedOrigin', 'CenterOfBoundBox'] = 'ProjectedOrigin'*, *origin: Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]] = None*) → T

Creates a new 2D workplane, located relative to the first face on the stack.

Parameters

- **self** (*T*) –
- **offset** (*float*) – offset for the workplane in its normal direction . Default
- **invert** (*bool*) – invert the normal direction from that of the face.
- **centerOption** (*string or None='ProjectedOrigin'*) – how local origin of workplane is determined.
- **origin** (*Optional[Union[Tuple[float, float], Tuple[float, float, float], Vector]]*) – origin for plane center, requires 'ProjectedOrigin' centerOption.

Return type
Workplane object

The first element on the stack must be a face, a set of co-planar faces or a vertex. If a vertex, then the parent item on the chain immediately before the vertex must be a face.

The result will be a 2D working plane with a new coordinate system set up as follows:

- The centerOption parameter sets how the center is defined. Options are 'CenterOfMass', 'CenterOfBoundBox', or 'ProjectedOrigin'. 'CenterOfMass' and 'CenterOfBoundBox' are in relation to the selected face(s) or vertex (vertices). 'ProjectedOrigin' uses by default the current origin or the optional origin parameter (if specified) and projects it onto the plane defined by the selected face(s).
- The Z direction will be the normal of the face, computed at the center point.
- The X direction will be parallel to the x-y plane. If the workplane is parallel to the global x-y plane, the x direction of the workplane will co-incide with the global x direction.

Most commonly, the selected face will be planar, and the workplane lies in the same plane of the face (IE, offset=0). Occasionally, it is useful to define a face offset from an existing surface, and even more rarely to define a workplane based on a face that is not planar.

workplaneFromTagged(*name: str*) → *Workplane*

Copies the workplane from a tagged parent.

Parameters
name (*str*) – tag to search for

Returns
a CQ object with name's workplane

Return type
Workplane

cadquery.**sortWiresByBuildOrder**(*wireList: List[*Wire*]*) → List[List[*Wire*]]

Tries to determine how wires should be combined into faces.

Assume:
The wires make up one or more faces, which could have 'holes' Outer wires are listed ahead of inner wires there are no wires inside wires inside wires (IE, islands – we can deal with that later on) none of the wires are construction wires

Compute:
one or more sets of wires, with the outer wire listed first, and inner ones

Returns, list of lists.

Parameters
wireList (*List[*Wire*]*) –

Return type
List[*List*[Wire]]

class cadquery.occ_impl.shapes.**Mixin1D**

Bases: object

endPoint() → *Vector*

Returns
a vector representing the end point of this edge.

Parameters
self (*Mixin1DProtocol*) –

Return type
Vector

Note, circles may have the start and end points the same

locationAt(*d: float*, *mode: Literal['length', 'parameter'] = 'length'*, *frame: Literal['frenet', 'corrected'] = 'frenet'*, *planar: bool = False*) → *Location*

Generate a location along the underlying curve.

Parameters

- **self** (*Mixin1DProtocol*) –
- **d** (*float*) – distance or parameter value
- **mode** (*Literal['length', 'parameter']*) – position calculation mode (default: length)
- **frame** (*Literal['frenet', 'corrected']*) – moving frame calculation method (default: frenet)
- **planar** (*bool*) – planar mode

Returns
A Location object representing local coordinate system at the specified distance.

Return type
Location

locations(*ds: Iterable[float]*, *mode: Literal['length', 'parameter'] = 'length'*, *frame: Literal['frenet', 'corrected'] = 'frenet'*, *planar: bool = False*) → List[*Location*]

Generate location along the curve

Parameters

- **self** (*Mixin1DProtocol*) –
- **ds** (*Iterable[float]*) – distance or parameter values
- **mode** (*Literal['length', 'parameter']*) – position calculation mode (default: length)
- **frame** (*Literal['frenet', 'corrected']*) – moving frame calculation method (default: frenet)
- **planar** (*bool*) – planar mode

Returns
A list of Location objects representing local coordinate systems at the specified distances.

Return type
List[Location]

normal() → *Vector*

Calculate the normal Vector. Only possible for planar curves.

Returns
normal vector

Parameters
self (*Mixin1DProtocol*) –

Return type
Vector

paramAt(*d: float*) → float

Compute parameter value at the specified normalized distance.

Parameters

- **self** (*Mixin1DProtocol*) –
- **d** (*float*) – normalized distance [0, 1]

Returns
parameter value

Return type
float

positionAt(*d: float*, *mode: Literal['length', 'parameter'] = 'length'*) → *Vector*

Generate a position along the underlying curve.

Parameters

- **self** (*Mixin1DProtocol*) –
- **d** (*float*) – distance or parameter value
- **mode** (*Literal['length', 'parameter']*) – position calculation mode (default: length)

Returns
A Vector on the underlying curve located at the specified d value.

Return type
Vector

positions(*ds: Iterable[float]*, *mode: Literal['length', 'parameter'] = 'length'*) → List[*Vector*]

Generate positions along the underlying curve

Parameters

- **self** (*Mixin1DProtocol*) –
- **ds** (*Iterable[float]*) – distance or parameter values
- **mode** (*Literal['length', 'parameter']*) – position calculation mode (default: length)

Returns
A list of Vector objects.

Return type
List[Vector]

project(*face: Face*, *d: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *closest: bool = True*) → Union[T1D, List[T1D]]

Project onto a face along the specified direction

Parameters

- **self** (*T1D*) –
- **face** (Face) –
- **d** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) –
- **closest** (*bool*) –

Return type
Union[*T1D*, *List*[*T1D*]]

radius() → float

Calculate the radius.

Note that when applied to a Wire, the radius is simply the radius of the first edge.

Returns
radius

Raises
ValueError – if kernel can not reduce the shape to a circular edge

Parameters
self (*Mixin1DProtocol*) –

Return type
float

startPoint() → *Vector*

Returns
a vector representing the start point of this edge

Parameters
self (*Mixin1DProtocol*) –

Return type
Vector

Note, circles may have the start and end points the same

tangentAt(*locationParam: float = 0.5*, *mode: Literal['length', 'parameter'] = 'length'*) → *Vector*

Compute tangent vector at the specified location.

Parameters

- **self** (*Mixin1DProtocol*) –
- **locationParam** (*float*) – distance or parameter value (default: 0.5)
- **mode** (*Literal['length', 'parameter']*) – position calculation mode (default: parameter)

Returns
tangent vector

Return type
Vector

class cadquery.occ_impl.shapes.**Mixin3D**

Bases: object

chamfer(*length: float*, *length2: Optional[float]*, *edgeList: Iterable[Edge]*) → Any

Chamfers the specified edges of this solid.

Parameters

- **self** (*Any*) –
- **length** (*float*) – length > 0, the length (length) of the chamfer
- **length2** (*Optional[float]*) – length2 > 0, optional parameter for asymmetrical chamfer. Should be *None* if not required.
- **edgeList** (*Iterable[Edge]*) – a list of Edge objects, which must belong to this solid

Returns
Chamfered solid

Return type
Any

dprism(*basis: Optional[Face]*, *faces: List[Face]*, *depth: Optional[Union[float, int]] = None*, *taper: Union[float, int] = 0*, *upToFace: Optional[Face] = None*, *thruAll: bool = True*, *additive: bool = True*) → *Solid*

dprism(*basis: Optional[Face]*, *profiles: List[Wire]*, *depth: Optional[Union[float, int]] = None*, *taper: Union[float, int] = 0*, *upToFace: Optional[Face] = None*, *thruAll: bool = True*, *additive: bool = True*) → *Solid*

Make a prismatic feature (additive or subtractive)

Parameters

- **self** (*TS*) –
- **basis** (*Optional[Face]*) – face to perform the operation on
- **profiles** (*List[Wire]*) – list of profiles
- **depth** (*Optional[Union[float, int]]*) – depth of the cut or extrusion
- **upToFace** (*Optional[Face]*) – a face to extrude until

- **thruAll** (*bool*) – cut thruAll
- **taper** (*Union[float, int]*) –
- **additive** (*bool*) –

Returns
a Solid object

Return type
Solid

fillet(*radius: float*, *edgeList: Iterable[Edge]*) → Any

Fillets the specified edges of this solid.

Parameters

- **self** (*Any*) –
- **radius** (*float*) – float > 0, the radius of the fillet
- **edgeList** (*Iterable[Edge]*) – a list of Edge objects, which must belong to this solid

Returns
Filleted solid

Return type
Any

isInside(*point: Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*, *tolerance: float = 1e-06*) → bool

Returns whether or not the point is inside a solid or compound object within the specified tolerance.

Parameters

- **self** (*ShapeProtocol*) –
- **point** (*Union[Vector, Tuple[Union[int, float], Union[int, float]], Tuple[Union[int, float], Union[int, float], Union[int, float]]]*) – tuple or Vector representing 3D point to be tested
- **tolerance** (*float*) – tolerance for inside determination, default=1.0e-6

Returns
bool indicating whether or not point is within solid

Return type
bool

shell(*faceList: Optional[Iterable[Face]]*, *thickness: float*, *tolerance: float = 0.0001*, *kind: Literal['arc', 'intersection'] = 'arc'*) → Any

Make a shelled solid of self.

Parameters

- **self** (*Any*) –
- **faceList** (*Optional[Iterable[Face]]*) – List of faces to be removed, which must be part of the solid. Can be an empty list.
- **thickness** (*float*) – Floating point thickness. Positive shells outwards, negative shells inwards.
- **tolerance** (*float*) – Modelling tolerance of the method, default=0.0001.
- **kind** (*Literal['arc', 'intersection']*) –

Returns
A shelled solid.

Return type
Any

class cadquery.selectors.**AndSelector**(*left*, *right*)

Bases: *BinarySelector*

Intersection selector. Returns objects that is selected by both selectors.

class cadquery.selectors.**AreaNthSelector**(*n: int*, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Bases: _NthSelector

Selects the object(s) with Nth area

Applicability:

- Faces, Shells, Solids - Shape.Area() is used to compute area
- closed planar Wires - a temporary face is created to compute area

Will ignore non-planar or non-closed wires.

Among other things can be used to select one of the nested coplanar wires or faces.

For example to create a fillet on a shank:

```
result = (
    cq.Workplane("XY")
    .circle(5)
    .extrude(2)
    .circle(2)
    .extrude(10)
    .faces(">Z[-2]")
    .wires(AreaNthSelector(0))
    .fillet(2)
)
```

Or to create a lip on a case seam:

```
result = (
    cq.Workplane("XY")
    .rect(20, 20)
    .extrude(10)
    .edges("|Z or <Z")
```

(continues on next page)

(continued from previous page)

```
    .fillet(2)
    .faces(">Z")
    .shell(2)
    .faces(">Z")
    .wires(AreaNthSelector(-1))
    .toPending()
    .workplane()
    .offset2D(-1)
    .extrude(1)
    .faces(">Z[-2]")
    .wires(AreaNthSelector(0))
    .toPending()
    .workplane()
    .cutBlind(2)
)
```

Parameters

- **n** (*int*) –
- **directionMax** (*bool*) –
- **tolerance** (*float*) –

key(*obj: Shape*) → float

Return the key for ordering. Can raise a ValueError if obj can not be used to create a key, which will result in obj being dropped by the clustering method.

Parameters
obj (*Shape*) –

Return type
float

class cadquery.selectors.**BaseDirSelector**(*vector:* Vector, *tolerance: float = 0.0001*)

Bases: *Selector*

A selector that handles selection on the basis of a single direction vector.

Parameters

- **vector** (Vector) –
- **tolerance** (*float*) –

filter(*objectList: Sequence[Shape]*) → List[Shape]

There are lots of kinds of filters, but for planes they are always based on the normal of the plane, and for edges on the tangent vector along the edge

Parameters
objectList (*Sequence[Shape]*) –

Return type
List[*Shape*]

test(*vec:* Vector) → bool

Test a specified vector. Subclasses override to provide other implementations

Parameters
vec (Vector) –

Return type
bool

class cadquery.selectors.**BinarySelector**(*left*, *right*)

Bases: *Selector*

Base class for selectors that operates with two other selectors. Subclass must implement the :filterResults(): method.

filter(*objectList: Sequence[Shape]*)

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

class cadquery.selectors.**BoxSelector**(*point0*, *point1*, *boundingbox=False*)

Bases: *Selector*

Selects objects inside the 3D box defined by 2 points.

If *boundingbox* is True only the objects that have their bounding box inside the given box is selected. Otherwise only center point of the object is tested.

Applicability: all types of shapes

Example:

```
CQ(aCube).edges(BoxSelector((0, 1, 0), (1, 2, 1)))
```

filter(*objectList: Sequence[Shape]*)

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

class cadquery.selectors.**CenterNthSelector**(*vector: Vector*, *n: int*, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Bases: _NthSelector

Sorts objects into a list with order determined by the distance of their center projected onto the specified direction.

Applicability:
All Shapes.

Parameters

- **vector** (Vector) –
- **n** (*int*) –

- **directionMax** (*bool*) –
- **tolerance** (*float*) –

key(*obj: Shape*) → float

Return the key for ordering. Can raise a ValueError if obj can not be used to create a key, which will result in obj being dropped by the clustering method.

Parameters
obj (*Shape*) –

Return type
float

class cadquery.selectors.**DirectionMinMaxSelector**(*vector: Vector*, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Bases: *CenterNthSelector*

Selects objects closest or farthest in the specified direction.

Applicability:
All object types. for a vertex, its point is used. for all other kinds of objects, the center of mass of the object is used.

You can use the string shortcuts >(X|Y|Z) or <(X|Y|Z) if you want to select based on a cardinal direction.

For example this:

```
CQ(aCube).faces(DirectionMinMaxSelector((0, 0, 1), True))
```

Means to select the face having the center of mass farthest in the positive z direction, and is the same as:

```
CQ(aCube).faces(">Z")
```

Parameters

- **vector** (*Vector*) –
- **directionMax** (*bool*) –
- **tolerance** (*float*) –

class cadquery.selectors.**DirectionNthSelector**(*vector: Vector*, *n: int*, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Bases: *ParallelDirSelector*, *CenterNthSelector*

Filters for objects parallel (or normal) to the specified direction then returns the Nth one.

Applicability:
Linear Edges Planar Faces

Parameters

- **vector** (*Vector*) –
- **n** (*int*) –
- **directionMax** (*bool*) –
- **tolerance** (*float*) –

filter(*objectlist: Sequence[Shape]*) → List[Shape]

There are lots of kinds of filters, but for planes they are always based on the normal of the plane, and for edges on the tangent vector along the edge

Parameters
objectlist (*Sequence[Shape]*) –

Return type
List[*Shape*]

class cadquery.selectors.**DirectionSelector**(*vector:* Vector, *tolerance: float = 0.0001*)

Bases: *BaseDirSelector*

Selects objects aligned with the provided direction.

Applicability:
Linear Edges Planar Faces

Use the string syntax shortcut +/-(X|Y|Z) if you want to select based on a cardinal direction.

Example:

```
CQ(aCube).faces(DirectionSelector((0, 0, 1)))
```

selects faces with the normal in the z direction, and is equivalent to:

```
CQ(aCube).faces("+Z")
```

Parameters

- **vector** (Vector) –
- **tolerance** (*float*) –

test(*vec:* Vector) → bool

Test a specified vector. Subclasses override to provide other implementations

Parameters
vec (Vector) –

Return type
bool

class cadquery.selectors.**InverseSelector**(*selector*)

Bases: *Selector*

Inverts the selection of given selector. In other words, selects all objects that is not selected by given selector.

filter(*objectList: Sequence[Shape]*)

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

class cadquery.selectors.**LengthNthSelector**(*n: int*, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Bases: _NthSelector

Select the object(s) with the Nth length

Applicability:
All Edge and Wire objects

Parameters

- **n** (*int*) –
- **directionMax** (*bool*) –
- **tolerance** (*float*) –

key(*obj: Shape*) → float

Return the key for ordering. Can raise a ValueError if obj can not be used to create a key, which will result in obj being dropped by the clustering method.

Parameters
obj (*Shape*) –

Return type
float

class cadquery.selectors.**NearestToPointSelector**(*pnt*)

Bases: *Selector*

Selects object nearest the provided point.

If the object is a vertex or point, the distance is used. For other kinds of shapes, the center of mass is used to to compute which is closest.

Applicability: All Types of Shapes

Example:

```
CQ(aCube).vertices(NearestToPointSelector((0, 1, 0)))
```

returns the vertex of the unit cube closest to the point x=0,y=1,z=0

filter(*objectList: Sequence[Shape]*)

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

class cadquery.selectors.**ParallelDirSelector**(*vector:* Vector, *tolerance: float = 0.0001*)

Bases: *BaseDirSelector*

Selects objects parallel with the provided direction.

Applicability:
Linear Edges Planar Faces

Use the string syntax shortcut |(X|Y|Z) if you want to select based on a cardinal direction.

Example:

```
CQ(aCube).faces(ParallelDirSelector((0, 0, 1)))
```

selects faces with the normal parallel to the z direction, and is equivalent to:

```
CQ(aCube).faces("|Z")
```

Parameters

- **vector** (Vector) –
- **tolerance** (*float*) –

test(*vec:* Vector) → bool

Test a specified vector. Subclasses override to provide other implementations

Parameters
vec (Vector) –

Return type
bool

class cadquery.selectors.**PerpendicularDirSelector**(*vector:* Vector, *tolerance: float = 0.0001*)

Bases: *BaseDirSelector*

Selects objects perpendicular with the provided direction.

Applicability:
Linear Edges Planar Faces

Use the string syntax shortcut #(X|Y|Z) if you want to select based on a cardinal direction.

Example:

```
CQ(aCube).faces(PerpendicularDirSelector((0, 0, 1)))
```

selects faces with the normal perpendicular to the z direction, and is equivalent to:

```
CQ(aCube).faces("#Z")
```

Parameters

- **vector** (Vector) –
- **tolerance** (*float*) –

test(*vec:* Vector) → bool

Test a specified vector. Subclasses override to provide other implementations

Parameters
vec (Vector) –

Return type
bool

class cadquery.selectors.**RadiusNthSelector**(*n: int*, *directionMax: bool = True*, *tolerance: float = 0.0001*)

Bases: _NthSelector

Select the object with the Nth radius.

Applicability:
All Edge and Wires.

Will ignore any shape that can not be represented as a circle or an arc of a circle.

Parameters

- **n** (*int*) –
- **directionMax** (*bool*) –
- **tolerance** (*float*) –

key(*obj: Shape*) → float

Return the key for ordering. Can raise a ValueError if obj can not be used to create a key, which will result in obj being dropped by the clustering method.

Parameters
obj (*Shape*) –

Return type
float

class cadquery.selectors.**Selector**

Bases: object

Filters a list of objects.

Filters must provide a single method that filters objects.

filter(*objectList: Sequence[Shape]*) → List[Shape]

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

Return type
List[*Shape*]

class cadquery.selectors.**StringSyntaxSelector**(*selectorString*)

Bases: *Selector*

Filter lists objects using a simple string syntax. All of the filters available in the string syntax are also available (usually with more functionality) through the creation of full-fledged selector objects. see *Selector* and its subclasses

Filtering works differently depending on the type of object list being filtered.

Parameters
selectorString – A two-part selector string, [selector][axis]

Returns
objects that match the specified selector

Modifiers are `('|','+','-','<','>','%')`

`|`
: parallel to (same as *ParallelDirSelector*). Can return multiple objects.

`#`
: perpendicular to (same as *PerpendicularDirSelector*)

`+`
: positive direction (same as *DirectionSelector*)

`-`
: negative direction (same as *DirectionSelector*)

`>`
: maximize (same as *DirectionMinMaxSelector* with directionMax=True)

`<`
: minimize (same as *DirectionMinMaxSelector* with directionMax=False)

`%`
: curve/surface type (same as *TypeSelector*)

axisStrings are: `X,Y,Z,XY,YZ,XZ` or `(x,y,z)` which defines an arbitrary direction

It is possible to combine simple selectors together using logical operations. The following operations are supported

and
: Logical AND, e.g. >X and >Y

or
: Logical OR, e.g. |X or |Y

not
: Logical NOT, e.g. not #XY

exc(ept)
: Set difference (equivalent to AND NOT): |X exc >Z

Finally, it is also possible to use even more complex expressions with nesting and arbitrary number of terms, e.g.

(not >X[0] and #XY) or >XY[0]

Selectors are a complex topic: see *Selectors Reference* for more information

filter(*objectList: Sequence[Shape]*)

Filter give object list through th already constructed complex selector object

Parameters
objectList (*Sequence[Shape]*) –

class cadquery.selectors.**SubtractSelector**(*left*, *right*)

Bases: *BinarySelector*

Difference selector. Subtract results of a selector from another selectors results.

class cadquery.selectors.**SumSelector**(*left*, *right*)

Bases: *BinarySelector*

Union selector. Returns the sum of two selectors results.

class cadquery.selectors.**TypeSelector**(*typeString: str*)

Bases: *Selector*

Selects objects having the prescribed geometry type.

Applicability:
Faces: PLANE, CYLINDER, CONE, SPHERE, TORUS, BEZIER, BSPLINE, REVOLUTION, EXTRUSION, OFFSET, OTHER Edges: LINE, CIRCLE, ELLIPSE, HYPERBOLA, PARABOLA, BEZIER, BSPLINE, OFFSET, OTHER

You can use the string selector syntax. For example this:

```
CQ(aCube).faces(TypeSelector("PLANE"))
```

will select 6 faces, and is equivalent to:

```
CQ(aCube).faces("%PLANE")
```

Parameters
typeString (*str*) –

filter(*objectList: Sequence[Shape]*) → List[Shape]

Filter the provided list.

The default implementation returns the original list unfiltered.

Parameters
objectList (*list of OCCT primitives*) – list to filter

Returns
filtered list

Return type
List[*Shape*]

cadquery.occ_impl.exporters.assembly.**exportAssembly**(*assy: AssemblyProtocol*, *path: str*, *mode: Literal['default', 'fused'] = 'default'*, ***kwargs*) → bool

Export an assembly to a STEP file.

kwargs is used to provide optional keyword arguments to configure the exporter.

Parameters

- **assy** (*AssemblyProtocol*) – assembly
- **path** (*str*) – Path and filename for writing
- **mode** (*Literal['default', 'fused']*) – STEP export mode. The options are “default”, and “fused” (a single fused compound). It is possible that fused mode may exhibit low performance.
- **fuzzy_tol** (*float*) – OCCT fuse operation tolerance setting used only for fused assembly export.
- **glue** (*bool*) – Enable gluing mode for improved performance during fused assembly export. This option should only be used for non-intersecting shapes or those that are only touching or partially overlapping. Note that when glue is enabled, the resulting fused shape may be invalid if shapes are intersecting in an incompatible way. Defaults to False.

- **write_pcurves** (*bool*) – Enable or disable writing parametric curves to the STEP file. Default True. If False, writes STEP file without pcurves. This decreases the size of the resulting STEP file.
- **precision_mode** (*int*) – Controls the uncertainty value for STEP entities. Specify -1, 0, or 1. Default 0. See OCCT documentation.

Return type
bool

cadquery.occ_impl.exporters.assembly.**exportCAF**(*assy: AssemblyProtocol*, *path: str*) → bool

Export an assembly to a OCAF xml file (internal OCCT format).

Parameters

- **assy** (*AssemblyProtocol*) –
- **path** (*str*) –

Return type
bool

cadquery.occ_impl.exporters.assembly.**exportGLTF**(*assy: AssemblyProtocol*, *path: str*, *binary: Optional[bool] = None*, *tolerance: float = 0.001*, *angularTolerance: float = 0.1*)

Export an assembly to a gltf file.

Parameters

- **assy** (*AssemblyProtocol*) –
- **path** (*str*) –
- **binary** (*Optional[bool]*) –
- **tolerance** (*float*) –
- **angularTolerance** (*float*) –

cadquery.occ_impl.exporters.assembly.**exportVRML**(*assy: AssemblyProtocol*, *path: str*, *tolerance: float = 0.001*, *angularTolerance: float = 0.1*)

Export an assembly to a vrml file using vtk.

Parameters

- **assy** (*AssemblyProtocol*) –
- **path** (*str*) –
- **tolerance** (*float*) –
- **angularTolerance** (*float*) –

cadquery.occ_impl.exporters.assembly.**exportVTKJS**(*assy: AssemblyProtocol*, *path: str*)

Export an assembly to a zipped vtkjs. NB: .zip extensions is added to path.

Parameters

- **assy** (*AssemblyProtocol*) –
- **path** (*str*) –

cadquery.occ_impl.assembly.**toJSON**(*assy: AssemblyProtocol*, *color: Tuple[float, float, float, float] = (1.0, 1.0, 1.0, 1.0)*, *tolerance: float = 0.001*) → List[Dict[str, Any]]

Export an object to a structure suitable for converting to VTK.js JSON.

Parameters

- **assy** (*AssemblyProtocol*) –
- **color** (*Tuple[float, float, float, float]*) –
- **tolerance** (*float*) –

Return type
List[*Dict*[str, *Any*]]

class cadquery.occ_impl.exporters.dxf.**DxfDocument**(*dxfversion: str = 'AC1027', setup: Union[bool, List[str]] = False, doc_units: int = 4, *, metadata: Optional[Dict[str, str]] = None, approx: Optional[Literal['spline', 'arc']] = None, tolerance: float = 0.001*)

Create DXF document from CadQuery objects.

A wrapper for ezdxf providing methods for converting *cadquery.Workplane* objects to DXF entities.

The ezdxf document is available as the property `document`, allowing most features of ezdxf to be utilised directly.

Example usage

Listing 1: Single layer DXF document

```
rectangle = cq.Workplane().rect(10, 20)

dxf = DxfDocument()
dxf.add_shape(rectangle)
dxf.document.saveas("rectangle.dxf")
```

Listing 2: Multilayer DXF document

```
rectangle = cq.Workplane().rect(10, 20)
circle = cq.Workplane().circle(3)

dxf = DxfDocument()
dxf = (
    dxf.add_layer("layer_1", color=2)
    .add_layer("layer_2", color=3)
    .add_shape(rectangle, "layer_1")
    .add_shape(circle, "layer_2")
)
dxf.document.saveas("rectangle-with-hole.dxf")
```

Parameters

- **dxfversion** (*str*) –
- **setup** (*Union[bool, List[str]]*) –
- **doc_units** (*int*) –
- **metadata** (*Optional[Dict[str, str]]*) –
- **approx** (*Optional[Literal['spline', 'arc']]*) –
- **tolerance** (*float*) –

__init__(*dxfversion: str = 'AC1027'*, *setup: Union[bool, List[str]] = False*, *doc_units: int = 4*, ***, *metadata: Optional[Dict[str, str]] = None*, *approx: Optional[Literal['spline', 'arc']] = None*, *tolerance: float = 0.001*)

Initialize DXF document.

Parameters

- **dxfversion** (*str*) – DXF version specifier as string, default is "AC1027" respectively "R2013"
- **setup** (*Union[bool, List[str]]*) – setup default styles, `False` for no setup, `True` to set up everything or a list of topics as strings, e.g. `["linetypes", "styles"]` refer to `ezdxf.new()`.
- **doc_units** (*int*) – ezdxf document/modelspace units
- **metadata** (*Optional[Dict[str, str]]*) – document metadata a dictionary of name value pairs
- **approx** (*Optional[Literal['spline', 'arc']]*) – Approximation strategy for converting *cadquery.Workplane* objects to DXF entities:

  None
  no approximation applied

  "spline"
  all splines approximated as cubic splines

  "arc"
  all curves approximated as arcs and straight segments

- **tolerance** (*float*) – Approximation tolerance for converting *cadquery.Workplane* objects to DXF entities.

add_layer(*name: str*, ***, *color: int = 7*, *linetype: str = 'CONTINUOUS'*) → Self

Create a layer definition

Refer to ezdxf layers and ezdxf layer tutorial.

Parameters

- **name** (*str*) – layer definition name
- **color** (*int*) – color index. Standard colors include: 1 red, 2 yellow, 3 green, 4 cyan, 5 blue, 6 magenta, 7 white/black
- **linetype** (*str*) – ezdxf line type

Return type
Self

add_shape(*workplane:* Workplane, *layer: str = ''*) → Self

Add CadQuery shape to a DXF layer.

Parameters

- **workplane** (Workplane) – CadQuery Workplane
- **layer** (*str*) – layer definition name

Return type
Self

3.13 Importing and Exporting Files

3.13.1 Introduction

The purpose of this section is to explain how to import external file formats into CadQuery, and export files from it as well. While the external file formats can be used to interchange CAD model data with other software, CadQuery does not support any formats that carry parametric data with them at this time. The only format that is fully parametric is CadQuery's own Python format. Below are lists of the import and export file formats that CadQuery supports.

Import Formats

- DXF
- STEP

Export Formats

- DXF
- SVG
- STEP
- STL
- AMF
- TJS
- VRML
- VTP
- 3MF
- glTF

Notes on the Formats

- DXF is useful for importing complex 2D profiles that would be tedious to create using CadQuery's 2D operations. An example is that the 2D profiles of aluminum extrusion are often provided in DXF format. These can be imported and extruded to create the length of extrusion that is needed in a design.
- STEP files are useful for interchanging model data with other CAD and analysis systems, such as FreeCAD. Many parts such as screws have STEP files available, which can be imported and used in CadQuery assemblies.
- STL, AMF and 3MF files are mesh-based formats which are typically used in additive manufacturing (i.e. 3D printing). AMF and 3MF files support more features, but are not as universally supported as STL files.
- TJS is short for ThreeJS, and is a JSON mesh format that is useful for displaying 3D models in web browsers. The TJS format is used to display embedded 3D examples within the CadQuery documentation.
- VRML is a mesh-based format for representing interactive 3D objects in a web browser.
- VTP is a mesh-based format used by the VTK library.
- glTF is a mesh-based format useful for viewing models on the web. Whether the resulting glTF file is binary (.glb) or text (.gltf) is set by the file extension.

3.13.2 Importing DXF

DXF files can be imported using the *importers.importDXF()* method.

importers.**importDXF**(*tol: float = 1e-06*, *exclude: List[str] = []*, *include: List[str] = []*) → *Workplane*

Loads a DXF file into a Workplane.

All layers are imported by default. Provide a layer include or exclude list to select layers. Layer names are handled as case-insensitive.

Parameters

- **filename** (*str*) – The path and name of the DXF file to be imported
- **tol** (*float*) – The tolerance used for merging edges into wires
- **exclude** (*List[str]*) – a list of layer names not to import
- **include** (*List[str]*) – a list of layer names to import

Return type
Workplane

Importing a DXF profile with default settings and using it within a CadQuery script is shown in the following code.

```
import cadquery as cq

result = (
    cq.importers.importDXF("/path/to/dxf/circle.dxf").wires().toPending().extrude(10)
)
```

Note the use of the *Workplane.wires()* and *Workplane.toPending()* methods to make the DXF profile ready for use during subsequent operations. Calling `toPending()` tells CadQuery to make the edges/wires available to the next modelling operation that is called in the chain.

3.13.3 Importing STEP

STEP files can be imported using the `importers.importStep()` method (note the capitalization of "Step"). There are no parameters for this method other than the file path to import.

```
import cadquery as cq

result = cq.importers.importStep("/path/to/step/block.stp")
```

3.13.4 Exporting STEP

This section covers exporting CadQuery Workplane objects to STEP. For exporting assemblies to STEP, see the next section.

Default

The exporters module handles exporting Workplane objects to STEP. It is not necessary to set the export type explicitly since it will be determined from the file extension. Below is an example.

```
# Create a simple object
box = cq.Workplane().box(10, 10, 10)

# Export the box
cq.exporters.export(box, "/path/to/step/box.step")
```

Non-Default File Extensions

If there is a requirement to export the STEP file using an "stp" extension, CadQuery will throw an error saying that it does not recognize the file extension. In that case the export type has to be specified.

```
# Create a simple object
box = cq.Workplane().box(10, 10, 10)

# Export the box
cq.exporters.export(box, "/path/to/step/box.stp", cq.exporters.ExportTypes.STEP)

# The export type may also be specified as a literal
cq.exporters.export(box, "/path/to/step/box2.stp", "STEP")
```

Setting Extra Options

There are additional options that can be set when exporting an object to a STEP file. For an explanation of the options available, see the documentation of the *Shape.exportStep()* method or the `Assembly.exportAssembly`() method.

```
# Create a simple object
box = cq.Workplane().box(10, 10, 10)

# Export the box, provide additional options with the opt dict
cq.exporters.export(box, "/path/to/step/box.step", opt={"write_pcurves": False})

# or equivalently when exporting a lower level Shape object
box.val().exportStep("/path/to/step/box2.step", write_pcurves=False)
```

3.13.5 Exporting Assemblies to STEP

It is possible to export CadQuery assemblies directly to STEP. The STEP exporter has multiple options which change the way exported STEP files will appear and operate when opened in other CAD programs. All assembly export methods shown here will preserve the color information from the assembly.

Default

CadQuery assemblies have a *Assembly.save()* method which can write an assembly to a STEP file. An example assembly export with all defaults is shown below.

```
import cadquery as cq

# Create a sample assembly
assy = cq.Assembly()
body = cq.Workplane().box(10, 10, 10)
assy.add(body, color=cq.Color(1, 0, 0), name="body")
pin = cq.Workplane().center(2, 2).cylinder(radius=2, height=20)
assy.add(pin, color=cq.Color(0, 1, 0), name="pin")

# Save the assembly to STEP
assy.save("out.step")
```

This will produce a STEP file that is nested with auto-generated object names. The colors of each assembly object will be preserved, but the names that were set for each will not.

Fused

The following will attempt to create a single, fused shape while preserving the name and color information of each assembly object. The process of fusing the solid may cause performance issues in some cases, and is likely to alter the faces of the fused solids.

```
import cadquery as cq

# Create a sample assembly
assy = cq.Assembly()
body = cq.Workplane().box(10, 10, 10)
assy.add(body, color=cq.Color(1, 0, 0), name="body")
pin = cq.Workplane().center(2, 2).cylinder(radius=2, height=20)
assy.add(pin, color=cq.Color(0, 1, 0), name="pin")

# Save the assembly to STEP
assy.save("out.stp", "STEP", mode="fused")

# Specify additional options such as glue as keyword arguments
assy.save("out_glue.step", mode="fused", glue=True, write_pcurves=False)
```

Naming

It is also possible to set the name of the top level assembly object in the STEP file with either the DEFAULT or FUSED methods. This is done by setting the name property of the assembly before calling *Assembly.save()*.

```
assy = Assembly(name="my_assembly")
assy.save(
    "out.stp",
    cq.exporters.ExportTypes.STEP,
    mode=cq.exporters.assembly.ExportModes.FUSED,
)
```

If an assembly name is not specified, a UUID will be used to avoid name conflicts.

3.13.6 Exporting SVG

The SVG exporter has several options which can be useful for achieving the desired final output. Those options are as follows.

- *width* - Width of the resulting image (None to fit based on height).
- *height* - Height of the resulting image (None to fit based on width).
- *marginLeft* - Inset margin from the left side of the document.
- *marginTop* - Inset margin from the top side of the document.
- *projectionDir* - Direction the camera will view the shape from.
- *showAxes* - Whether or not to show the axes indicator, which will only be visible when the projectionDir is also at the default.
- *strokeWidth* - Width of the line that visible edges are drawn with.
- *strokeColor* - Color of the line that visible edges are drawn with.
- *hiddenColor* - Color of the line that hidden edges are drawn with.
- *showHidden* - Whether or not to show hidden lines.
- *focus* - If specified, creates a perspective SVG with the projector at the distance specified.

The options are passed to the exporter in a dictionary, and can be left out to force the SVG to be created with default options. Below are examples with and without options set.

Without options:

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)

exporters.export(result, "/path/to/file/box.svg")
```

Results in:

Note that the exporters API figured out the format type from the file extension. The format type can be set explicitly by using `exporters.ExportTypes`.

The following is an example of using options to alter the resulting SVG output by passing in the `opt` parameter.

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)

exporters.export(
    result,
    "/path/to/file/box_custom_options.svg",
    opt={
        "width": 300,
```

(continues on next page)

(continued from previous page)

```
        "height": 300,
        "marginLeft": 10,
        "marginTop": 10,
        "showAxes": False,
        "projectionDir": (0.5, 0.5, 0.5),
        "strokeWidth": 0.25,
        "strokeColor": (255, 0, 0),
        "hiddenColor": (0, 0, 255),
        "showHidden": True,
    },
)
```

Which results in the following image:

Exporting with the additional option `"focus": 25` results in the following output SVG with perspective:

3.13.7 Exporting STL

The STL exporter is capable of adjusting the quality of the resulting mesh, and accepts the following parameters.

Shape.**exportStl**(*fileName: str*, *tolerance: float = 0.001*, *angularTolerance: float = 0.1*, *ascii: bool = False*, *relative: bool = True*, *parallel: bool = True*) → bool

Exports a shape to a specified STL file.

Parameters

- **fileName** (*str*) – The path and file name to write the STL output to.
- **tolerance** (*float*) – A linear deflection setting which limits the distance between a curve and its tessellation. Setting this value too low will result in large meshes that can consume computing resources. Setting the value too high can result in meshes with a level of detail that is too low. Default is 1e-3, which is a good starting point for a range of cases.
- **angularTolerance** (*float*) – Angular deflection setting which limits the angle between subsequent segments in a polyline. Default is 0.1.
- **ascii** (*bool*) – Export the file as ASCII (True) or binary (False) STL format. Default is binary.
- **relative** (*bool*) – If True, tolerance will be scaled by the size of the edge being meshed. Default is True. Setting this value to True may cause large features to become faceted, or small features dense.
- **parallel** (*bool*) – If True, OCCT will use parallel processing to mesh the shape. Default is True.

Return type

bool

For more complex objects, some experimentation with `tolerance` and `angularTolerance` may be required to find the optimum values that will produce an acceptable mesh.

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)

exporters.export(result, "/path/to/file/mesh.stl")
```

3.13.8 Exporting AMF and 3MF

The AMF and 3MF exporters are capable of adjusting the quality of the resulting mesh, and accept the following parameters.

- `fileName` - The path and file name to write the AMF output to.
- `tolerance` - A linear deflection setting which limits the distance between a curve and its tessellation. Setting this value too low will result in large meshes that can consume computing resources. Setting the value too high can result in meshes with a level of detail that is too low. Default is 0.1, which is good starting point for a range of cases.
- `angularTolerance` - Angular deflection setting which limits the angle between subsequent segments in a polyline. Default is 0.1.

For more complex objects, some experimentation with `tolerance` and `angularTolerance` may be required to find the optimum values that will produce an acceptable mesh. Note that parameters for color and material are absent.

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)

exporters.export(result, "/path/to/file/mesh.amf", tolerance=0.01, angularTolerance=0.1)
```

3.13.9 Exporting TJS

The TJS (ThreeJS) exporter produces a file in JSON format that describes a scene for the ThreeJS WebGL renderer. The objects in the first argument are converted into a mesh and then form the ThreeJS geometry for the scene. The mesh can be adjusted with the following parameters.

- `fileName` - The path and file name to write the ThreeJS output to.
- `tolerance` - A linear deflection setting which limits the distance between a curve and its tessellation. Setting this value too low will result in large meshes that can consume computing resources. Setting the value too high can result in meshes with a level of detail that is too low. Default is 0.1, which is good starting point for a range of cases.
- `angularTolerance` - Angular deflection setting which limits the angle between subsequent segments in a polyline. Default is 0.1.

For more complex objects, some experimentation with `tolerance` and `angularTolerance` may be required to find the optimum values that will produce an acceptable mesh.

```
import cadquery as cq
from cadquery import exporters

```

(continues on next page)

(continued from previous page)

```
result = cq.Workplane().box(10, 10, 10)

exporters.export(
    result,
    "/path/to/file/mesh.json",
    tolerance=0.01,
    angularTolerance=0.1,
    exportType=exporters.ExportTypes.TJS,
)
```

Note that the export type was explicitly specified as `TJS` because the extension that was used for the file name was `.json`. If the extension `.tjs` had been used, CadQuery would have understood to use the `TJS` export format.

3.13.10 Exporting VRML

The VRML exporter is capable of adjusting the quality of the resulting mesh, and accepts the following parameters.

- `fileName` - The path and file name to write the VRML output to.
- `tolerance` - A linear deflection setting which limits the distance between a curve and its tessellation. Setting this value too low will result in large meshes that can consume computing resources. Setting the value too high can result in meshes with a level of detail that is too low. Default is 0.1, which is good starting point for a range of cases.
- `angularTolerance` - Angular deflection setting which limits the angle between subsequent segments in a polyline. Default is 0.1.

For more complex objects, some experimentation with `tolerance` and `angularTolerance` may be required to find the optimum values that will produce an acceptable mesh.

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)

exporters.export(
    result, "/path/to/file/mesh.vrml", tolerance=0.01, angularTolerance=0.1
)
```

3.13.11 Exporting DXF

See also:

cadquery.occ_impl.exporters.dxf.DxfDocument for exporting multiple Workplanes to one or many layers of a DXF document.

Options

approx
Approximation strategy for converting *cadquery.Workplane* objects to DXF entities:

None
no approximation applied

"spline"
all splines approximated as cubic splines

"arc"
all curves approximated as arcs and straight segments

tolerance
Approximation tolerance for converting *cadquery.Workplane* objects to DXF entities. See *Approximation strategy*.

doc_units
Ezdxf document/modelspace units. See *Units*.

Listing 3: DXF document without options.

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)

exporters.exportDXF(result, "/path/to/file/object.dxf")
# or
exporters.export(result, "/path/to/file/object.dxf")
```

Units

The default DXF document units are mm (`doc_units = 4`).

doc_units	Unit
0	Unitless
1	Inches
2	Feet
3	Miles
4	Millimeters
5	Centimeters
6	Meters

Document units can be set to any unit supported by ezdxf.

Listing 4: DXF document with units set to meters.

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)
```

(continues on next page)

(continued from previous page)

```
exporters.exportDXF(
    result,
    "/path/to/file/object.dxf",
    doc_units=6,  # set DXF document units to meters
)

# or

exporters.export(
    result,
    "/path/to/file/object.dxf",
    opt={"doc_units": 6},  # set DXF document units to meters
)
```

Approximation strategy

By default, the DXF exporter will output splines exactly as they are represented by the OpenCascade kernel. Unfortunately some software cannot handle higher-order splines resulting in missing curves after DXF import. To resolve this, specify an approximation strategy controlled by the following options:

- `approx` - `None`, `"spline"` or `"arc"`. `"spline"` results in all splines approximated with cubic splines. `"arc"` results in all curves approximated with arcs and line segments.
- `tolerance`: Acceptable error of the approximation, in document/modelspace units. Defaults to 0.001 (1 thou for inch-scale drawings, 1 µm for mm-scale drawings).

Listing 5: DXF document with curves approximated with cubic splines.

```
cq.exporters.exportDXF(result, "/path/to/file/object.dxf", approx="spline")
```

3.13.12 Exporting Other Formats

The remaining export formats do not accept any additional parameters other than file name, and can be exported using the following structure.

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)

exporters.export(result, "/path/to/file/object.[file_extension]")
```

Be sure to use the correct file extension so that CadQuery can determine the export format. If in doubt, fall back to setting the type explicitly by using `exporters.ExportTypes`.

For example:

```
import cadquery as cq
from cadquery import exporters

result = cq.Workplane().box(10, 10, 10)
```

(continues on next page)

(continued from previous page)

```
exporters.export(result, "/path/to/file/object.dxf", exporters.ExportTypes.DXF)
```

3.14 The CadQuery Gateway Interface

CadQuery is first and foremost designed as a library, which can be used as a part of any project. In this context, there is no need for a standard script format or gateway API.

Though the embedded use case is the most common, several tools have been created which run cadquery scripts on behalf of the user, and then render the result of the script visually.

These execution environments (EE) generally accept a script and user input values for script parameters, and then display the resulting objects visually to the user.

Today, three execution environments exist:

- CQ-editor, which runs scripts inside of a CadQuery IDE, and displays objects in the display window and includes features like debugging.
- The cq-directive, which is used to execute scripts inside of sphinx-doc, producing documented examples that include both a script and an SVG representation of the object that results.

The CQGI is distributed with CadQuery, and standardizes the interface between execution environments and CadQuery scripts.

3.14.1 The Script Side

CQGI compliant containers provide an execution environment for scripts. The environment includes:

- the cadquery library is automatically imported as 'cq'.
- the *cadquery.cqgi.ScriptCallback.show_object()* method is defined that should be used to export a shape to the execution environment
- the `cadquery.cqgi.ScriptCallBack.debug()` method is defined, which can be used by scripts to debug model output during execution.

Scripts must call show_object at least once. Invoking show_object more than once will send multiple objects to the container. An error will occur if the script does not return an object using the show_object() method.

This CQGI compliant script produces a cube with a circle on top, and displays a workplane as well as an intermediate circle as debug output:

```
base_cube = cq.Workplane("XY").rect(1.0, 1.0).extrude(1.0)
top_of_cube_plane = base_cube.faces(">Z").workplane()
debug(
    top_of_cube_plane,
    {
        "color": "yellow",
    },
)
debug(top_of_cube_plane.center, {"color": "blue"})

circle = top_of_cube_plane.circle(0.5)
debug(circle, {"color": "red"})
```

(continues on next page)

(continued from previous page)

```
show_object(circle.extrude(1.0))
```

Note that importing cadquery is not required. At the end of this script, one object will be displayed, in addition to a workplane, a point, and a circle

Future enhancements will include several other methods, used to provide more metadata for the execution environment:

- *cadquery.cqgi.ScriptCallback.add_error()*, indicates an error with an input parameter
- *cadquery.cqgi.ScriptCallback.describe_parameter()*, provides extra information about a parameter in the script,

3.14.2 The execution environment side

CQGI makes it easy to run cadquery scripts in a standard way. To run a script from an execution environment, run code like this:

```
from cadquery import cqgi

user_script = ...
build_result = cqgi.parse(user_script).build()
```

The *cadquery.cqgi.parse()* method returns a *cadquery.cqgi.CQModel* object.

The *metadata`p property of the object contains a `cadquery.cqgi.ScriptMetaData* object, which can be used to discover the user parameters available. This is useful if the execution environment would like to present a GUI to allow the user to change the model parameters. Typically, after collecting new values, the environment will supply them in the build() method.

This code will return a dictionary of parameter values in the model text SCRIPT::

parameters = cqgi.parse(SCRIPT).metadata.parameters

The dictionary you get back is a map where key is the parameter name, and value is an InputParameter object, which has a name, type, and default value.

The type is an object which extends ParameterType– you can use this to determine what kind of widget to render (checkbox for boolean, for example).

The parameter object also has a description, valid values, minimum, and maximum values, if the user has provided them using the describe_parameter() method.

Calling *cadquery.cqgi.CQModel.build()* returns a *cadquery.cqgi.BuildResult* object, ,which includes the script execution time, and a success flag.

If the script was successful, the results property will include a list of results returned by the script, as well as any debug the script produced

If the script failed, the exception property contains the exception object.

If you have a way to get inputs from a user, you can override any of the constants defined in the user script with new values:

```
from cadquery import cqgi

user_script = ...
build_result = cqgi.parse(user_script).build(
```

(continues on next page)

(continued from previous page)

```
    build_parameters={"param": 2}, build_options={}
)
```

If a parameter called 'param' is defined in the model, it will be assigned the value 2 before the script runs. An error will occur if a value is provided that is not defined in the model, or if the value provided cannot be assigned to a variable with the given name.

build_options is used to set server-side settings like timeouts, tessellation tolerances, and other details about how the model should be built.

3.14.3 More about script variables

CQGI uses the following rules to find input variables for a script:

- only top-level statements are considered
- only assignments of constant values to a local name are considered.

For example, in the following script:

```
h = 1.0
w = 2.0
foo = "bar"


def some_function():
    x = 1
```

h, w, and foo will be overridable script variables, but x is not.

You can list the variables defined in the model by using the return value of the parse method:

```
model = cqgi.parse(user_script)

# a dictionary of InputParameter objects
parameters = model.metadata.parameters
```

The key of the dictionary is a string , and the value is a *cadquery.cqgi.InputParameter* object See the CQGI API docs for more details.

Future enhancements will include a safer sandbox to prevent malicious scripts.

3.14.4 Automating export to STL

A common use-case for the CQGI is the automation of processing cadquery code into geometry, doing so via the CQGI rather than an export line in the script itself leads to a much tidier environment; you may need to do this as part of an automated-workflow, batch-conversion, exporting to another software for assembly, or running stress simulations on resulting bodies.

The below Python script demonstrates how to open, process, and export an STL file from any valid cadquery script:

```
# Load CQGI
import cadquery.cqgi as cqgi
import cadquery as cq
```

(continues on next page)

(continued from previous page)

```
# load the cadquery script
model = cqgi.parse(open("example.py").read())

# run the script and store the result (from the show_object call in the script)
build_result = model.build()

# test to ensure the process worked.
if build_result.success:
    # loop through all the shapes returned and export to STL
    for i, result in enumerate(build_result.results):
        cq.exporters.export(result.shape, f"example_output{i}.stl")
else:
    print(f"BUILD FAILED: {build_result.exception}")
```

3.14.5 Important CQGI Methods

These are the most important Methods and classes of the CQGI

parse(script_source)	Parses the script as a model, and returns a model.
CQModel.build([build_parameters, build_options])	Executes the script, using the optional parameters to override those in the model
BuildResult()	The result of executing a CadQuery script.
ScriptCallback.show_object(shape[, options])	Return an object to the executing environment, with options.

3.14.6 Complete CQGI API

The CadQuery Gateway Interface. Provides classes and tools for executing CadQuery scripts

class cadquery.cqgi.**BuildResult**

The result of executing a CadQuery script. The success property contains whether the execution was successful.

If successful, the results property contains a list of all results, and the first_result property contains the first result.

If unsuccessful, the exception property contains a reference to the stack trace that occurred.

class cadquery.cqgi.**CQModel**(*script_source*)

Represents a Cadquery Script.

After construction, the metadata property contains a ScriptMetaData object, which describes the model in more detail, and can be used to retrieve the parameters defined by the model.

the build method can be used to generate a 3d model

build(*build_parameters=None*, *build_options=None*)

Executes the script, using the optional parameters to override those in the model

Parameters

- **build_parameters** – a dictionary of variables. The variables must be assignable to the underlying variable type. These variables override default values in the script

- **build_options** – build options for how to build the model. Build options include things like timeouts, tessellation tolerances, etc

Raises
Nothing. If there is an exception, it will be on the exception property of the result. This is the interface so that we can return other information on the result, such as the build time

Returns
a BuildResult object, which includes the status of the result, and either a resulting shape or an exception

validate(*params*)

Determine if the supplied parameters are valid. NOT IMPLEMENTED YET– raises NotImplementedError

Parameters
params – a dictionary of parameters

class cadquery.cqgi.**ConstantAssignmentFinder**(*cq_model*)

Visits a parse tree, and adds script parameters to the cqModel

class cadquery.cqgi.**EnvironmentBuilder**

Builds an execution environment for a cadquery script. The environment includes the builtins, as well as the other methods the script will need.

class cadquery.cqgi.**InputParameter**

Defines a parameter that can be supplied when the model is executed.

Name, varType, and default_value are always available, because they are computed from a variable assignment line of code:

The others are only available if the script has used define_parameter() to provide additional metadata

default_value

the default value for the variable.

desc

help text describing the variable. Only available if the script used describe_parameter()

name

the name of the parameter.

valid_values

valid values for the variable. Only available if the script used describe_parameter()

varType

type of the variable: BooleanParameter, StringParameter, NumericParameter

exception cadquery.cqgi.**InvalidParameterError**

Raised when an attempt is made to provide a new parameter value that cannot be assigned to the model

exception cadquery.cqgi.**NoOutputError**

Raised when the script does not execute the show_object() method to return a solid

class cadquery.cqgi.**ParameterDescriptionFinder**(*cq_model*)

Visits a parse tree, looking for function calls to describe_parameter(var, description)

visit_Call(*node*)

Called when we see a function call. Is it describe_parameter?

class cadquery.cqgi.**ScriptCallback**

Allows a script to communicate with the container the show_object() method is exposed to CQ scripts, to allow them to return objects to the execution environment

add_error(*param*, *field_list*)

Not implemented yet: allows scripts to indicate that there are problems with inputs

debug(*obj*, *args={}*)

Debug print/output an object, with optional arguments.

describe_parameter(*var_data*)

Do Nothing– we parsed the ast ahead of execution to get what we need.

show_object(*shape*, *options={'name': 'door'}*, ***kwargs*)

Return an object to the executing environment, with options.

Parameters

- **shape** – a cadquery object
- **options** – a dictionary of options that will be made available to the executing environment

exception cadquery.cqgi.**ScriptExecutionError**(*line=None*, *message=None*)

Represents a script syntax error. Useful for helping clients pinpoint issues with the script interactively

class cadquery.cqgi.**ScriptMetadata**

Defines the metadata for a parsed CQ Script. the parameters property is a dict of InputParameter objects.

class cadquery.cqgi.**ShapeResult**

An object created by a build, including the user parameters provided

cadquery.cqgi.**parse**(*script_source*)

Parses the script as a model, and returns a model.

If you would prefer to access the underlying model without building it, for example, to inspect its available parameters, construct a CQModel object.

Parameters

script_source – the script to run. Must be a valid cadquery script

Returns

a CQModel object that defines the script and allows execution

3.15 Extending CadQuery

If you find that CadQuery does not suit your needs, you can easily extend it. CadQuery provides several extension methods:

- You can load plugins others have developed. This is by far the easiest way to access other code
- You can define your own plugins.
- You can use OCP scripting directly

3.15.1 Using OpenCascade methods

The easiest way to extend CadQuery is to simply use OpenCascade/OCP scripting inside of your build method. Just about any valid OCP script will execute just fine. For example, this simple CadQuery script:

```
return cq.Workplane("XY").box(1.0, 2.0, 3.0).val()
```

is actually equivalent to:

```
return cq.Shape.cast(
    BRepPrimAPI_MakeBox(
        gp_Ax2(Vector(-0.1, -1.0, -1.5), Vector(0, 0, 1)), 1.0, 2.0, 3.0
    ).Shape()
)
```

As long as you return a valid OCP Shape, you can use any OCP methods you like. You can even mix and match the two. For example, consider this script, which creates a OCP box, but then uses CadQuery to select its faces:

```
box = cq.Shape.cast(
    BRepPrimAPI_MakeBox(
        gp_Ax2(Vector(-0.1, -1.0, -1.5), Vector(0, 0, 1)), 1.0, 2.0, 3.0
    ).Shape()
)
cq = Workplane(box).faces(">Z").size()  # returns 6
```

3.15.2 Extending CadQuery: Plugins

Though you can get a lot done with OpenCascade, the code gets pretty nasty in a hurry. CadQuery shields you from a lot of the complexity of the OpenCascade API.

You can get the best of both worlds by wrapping your OCP script into a CadQuery plugin.

A CadQuery plugin is simply a function that is attached to the CadQuery *cadquery.CQ()* or *cadquery.Workplane()* class. When connected, your plugin can be used in the chain just like the built-in functions.

There are a few key concepts important to understand when building a plugin

3.15.3 The Stack

Every CadQuery object has a local stack, which contains a list of items. The items on the stack will be one of these types:

- **A CadQuery SolidReference object**, which holds a reference to a OCP solid
- **A OCP object**, a Vertex, Edge, Wire, Face, Shell, Solid, or Compound

The stack is available by using self.objects, and will always contain at least one object.

Note: Objects and points on the stack are **always** in global coordinates. Similarly, any objects you create must be created in terms of global coordinates as well!

3.15.4 Preserving the Chain

CadQuery's fluent API relies on the ability to chain calls together one after another. For this to work, you must return a valid CadQuery object as a return value. If you choose not to return a CadQuery object, then your plugin will end the chain. Sometimes this is desired for example *cadquery.Workplane.size()*

There are two ways you can safely continue the chain:

1. **return self** If you simply wish to modify the stack contents, you can simply return a reference to self. This approach is destructive, because the contents of the stack are modified, but it is also the simplest.
2. *cadquery.Workplane.newObject()* Most of the time, you will want to return a new object. Using newObject will return a new CQ or Workplane object having the stack you specify, and will link this object to the previous one. This preserves the original object and its stack.

3.15.5 Helper Methods

When you implement a CadQuery plugin, you are extending CadQuery's base objects. As a result, you can call any CadQuery or Workplane methods from inside of your extension. You can also call a number of internal methods that are designed to aid in plugin creation:

- `cadquery.Workplane._makeWireAtPoints()` will invoke a factory function you supply for all points on the stack, and return a properly constructed cadquery object. This function takes care of registering wires for you and everything like that
- *cadquery.Workplane.newObject()* returns a new Workplane object with the provided stack, and with its parent set to the current object. The preferred way to continue the chain
- *cadquery.Workplane.findSolid()* returns the first Solid found in the chain, working from the current object upwards in the chain. commonly used when your plugin will modify an existing solid, or needs to create objects and then combine them onto the 'main' part that is in progress
- `cadquery.Workplane._addPendingWire()` must be called if you add a wire. This allows the base class to track all the wires that are created, so that they can be managed when extrusion occurs.
- *cadquery.Workplane.wire()* gathers up all of the edges that have been drawn (eg, by line, vline, etc), and attempts to combine them into a single wire, which is returned. This should be used when your plugin creates 2D edges, and you know it is time to collect them into a single wire.
- `cadquery.Workplane.plane()` provides a reference to the workplane, which allows you to convert between workplane coordinates and global coordinates: * `cadquery.occ_impl.geom.Plane.toWorldCoords()` will convert local coordinates to global ones * `cadquery.occ_impl.geom.Plane.toLocalCoords()` will convert from global coordinates to local coordinates

3.15.6 Coordinate Systems

Keep in mind that the user may be using a work plane that has created a local coordinate system. Consequently, the orientation of shapes that you create are often implicitly defined by the user's workplane.

Any objects that you create must be fully defined in *global coordinates*, even though some or all of the users' inputs may be defined in terms of local coordinates.

3.15.7 Linking in your plugin

Your plugin is a single method, which is attached to the main Workplane or CadQuery object.

Your plugin method's first parameter should be 'self', which will provide a reference to base class functionality. You can also accept other arguments.

To install it, simply attach it to the CadQuery or Workplane object, like this:

```
def _yourFunction(self, arg1, arg):
    # do stuff
    return whatever_you_want


cq.Workplane.yourPlugin = _yourFunction
```

That's it!

3.15.8 CadQueryExample Plugins

Some core cadquery code is intentionally written exactly like a plugin. If you are writing your own plugins, have a look at these methods for inspiration:

- *cadquery.Workplane.polygon()*
- *cadquery.Workplane.cboreHole()*

3.15.9 Plugin Example

This ultra simple plugin makes cubes of the specified size for each stack point.

(The cubes are off-center because the boxes have their lower left corner at the reference points.)

```
def makeCubes(self, length):
    # self refers to the CQ or Workplane object

    # inner method that creates a cube
    def _singleCube(loc):
        # loc is a location in local coordinates
        # since we're using eachpoint with useLocalCoordinates=True
        return cq.Solid.makeBox(length, length, length, pnt).locate(loc)

    # use CQ utility method to iterate over the stack, call our
    # method, and convert to/from local coordinates.
    return self.eachpoint(_singleCube, True)


# link the plugin into CadQuery
cq.Workplane.makeCubes = makeCubes

# use the plugin
result = (
    cq.Workplane("XY")
    .box(6.0, 8.0, 0.5)
```

(continues on next page)

(continued from previous page)

```
    .faces(">Z")
    .rect(4.0, 4.0, forConstruction=True)
    .vertices()
    .makeCubes(1.0)
    .combineSolids()
)
```

3.16 RoadMap: Planned Features

CadQuery is not even close to finished!!!

Many features are planned for later versions. This page tracks them. If you find that you need features not listed here, let us know!

3.16.1 Workplanes

rotated workplanes
support creation of workplanes at an angle to another plane or face

workplane local rotations
rotate the coordinate system of a workplane by an angle.

make a workplane from a wire
useful to select outer wire and then operate from there, to allow offsets

3.16.2 Assemblies

implement more constraints
in plane, on axis, parallel to vector

3.16.3 2D operations

arc construction using relative measures
instead of forcing use of absolute workplane coordinates

tangent arcs
after a line

centerpoint arcs
including portions of arcs as well as with end points specified

trimming
ability to use construction geometry to trim other entities

construction lines
especially centerlines

2D fillets
for a rectangle, or for consecutive selected lines

2D chamfers
based on rectangles, polygons, polylines, or adjacent selected lines

mirror around centerline
using centerline construction geometry

midpoint selection
select midpoints of lines, arcs

face center
explicit selection of face center

manipulate spline control points
so that the shape of a spline can be more accurately controlled

feature snap
project geometry in the rest of the part into the work plane, so that they can be selected and used as references for other features.

polyline edges
allow polyline to be combined with other edges/curves

3.16.4 3D operations

rotation/transform that return a copy
The current rotateAboutCenter and translate method modify the object, rather than returning a copy

primitive creation

Need primitive creation for:

- cone
- torus
- wedge

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W