# FreeCAD scripting guide

Author: Carlo Dormeletti

**FreeCAD** reference version: **0.19**
Document version: **0.55a**
Print Date: **January 11, 2022**

## License

## Disclaimer

## Thanks

- **FreeCAD** core developers.
- **FreeCAD** forum users:
  - chrisb
  - Chris_G
  - edwilliams16
  - Kunda1
  - Russ4262
  - sliptonic
  - TheMarkster

Last thank to all the developers of TEX, LATEX $2_\varepsilon$ e TikZ without their efforts this guide, will be simply an HTML file with some images attached.

## Contacts, bug reports, suggestions

To report errors, bugs, and suggestions please use the "issues" section on this GitHub page:

https://github.com/onekk/freecad-doc

Have fun with **FreeCAD**.

Carlo Dormeletti (onekk)

# Change log

## Version remarks

Version numbering is done according to the software release scheme, in other words, non definitive versions have a version number starting with 0.

First “finished” version will be 1.0 and so on.

To be in sync with the “original” Italian version this guide is starting from 0.50.

## History

**v0.50 - 07 June 2021** Starting efforts

**v0.51 - 03 January 2022** Some progress, mainly introductory translation from Italian text, avoided duplication with documentation on **FreeCAD** Wiki pages and some new images with English interface.

**v0.52a - 06 January 2022** Some modification around, advancement in translation, modified all the old URL for **FreeCAD** wiki and site, cleaned some code around.

# Contents

**Foreword** VII

**Text Appearance** VIII

**1. First Steps** 3
- 1.1. Settings . . . 4
- 1.2. The property editor . . . 5
- 1.3. The macro editor . . . 5

**2. Introduction to Scripting** 7
- 2.1. First approach . . . 7
- 2.2. Second approach . . . 7
- 2.3. Program blocks . . . 8
  - 2.3.1. Brief description of the code . . . 8
- 2.4. Listings - base template . . . 9

**3. 3D Solids** 12
- 3.1. 3D Spaces . . . 12
  - 3.1.1. Vectors . . . 12
  - 3.1.2. Data e View . . . 13
- 3.2. Geometry, Topology and other things . . . 14
  - 3.2.1. Geometry . . . 14
  - 3.2.2. Topology . . . 15
- 3.3. Geometry and topology in FreeCAD . . . 15
- 3.4. Modeling . . . 17

**4. CSG Modeling** 18
- 4.1. First objects . . . 18
  - 4.1.1. Listing - base-objects.py . . . 20

**5. Placement** 22
- 5.1. Reference Point . . . 22
- 5.2. Positioning . . . 24
  - 5.2.1. **Placement** Property . . . 24
  - 5.2.2. Listing - Reference point . . . 27

**6. Boolean Operations** 29
- 6.1. Union . . . 29
- 6.2. Subtraction . . . 31

6.3. Intersection . . . . . . . . . . 31
6.4. Listing - Boolean Operation . . . . . . . . . . 32

**7. BREP modeling 35**
7.1. Extrusion . . . . . . . . . . 36
7.2. Revolution . . . . . . . . . . 38
7.3. Loft . . . . . . . . . . 39
7.4. Sweep . . . . . . . . . . 42

**A. User Interface Elements 50**

**B. Glossary 51**

**C. Menu Item 52**

**D. FreeCAD Objects 53**

# List of Figures

1.1. User interface at first start . . . . 3
1.2. User interface . . . . 4
1.3. PE Data and View Tab . . . . 6

3.1. 3D space . . . . 12

5.1. Placement . . . . 24
5.2. Property Editor . . . . 25

6.1. Boolean Operations. . . . . 32

7.1. Extrusion . . . . 37
7.2. Revolution . . . . 39
7.3. Loft . . . . 42
7.4. Sweep example . . . . 49

# List of Tables

3.1. Geometrie . . . . . . . . . . 16

5.1. Reference point . . . . . . . . . . 22
5.2. Placement property . . . . . . . . . . 25
5.3. Tait-Bryan Angles . . . . . . . . . . 26
5.4. Rotation expressed using Tait-Bryan angles . . . . . . . . . . 26

# Foreword

This guide is intended as a beginner help to **FreeCAD** scripting, intended to be a way to model 3d part to be 3d printed with an hobby machine.

Carlo Dormeletti

onekk

# Text Appearance

Colors used in text have a precise meaning:

- **Report View** **FreeCAD** GUI element.
- **View** menu item or tree-view items.
  **View** ⇒ **Toolbars**, menu items sequence or tree in a tree-view.
- **Part::Box** methods (functions) and 3d primitives **FreeCAD**.
- **Placement** method properties.
- `variable` variable names or other code names in a text phrase.

Key sequence or mouse actions are written this way:

- **CTRL+SHIFT+F** push together Ctrl, Shift e F key
- **right/left/other click** click the corresponding mouse key.
- **right/left/other double click** double click mouse key.
- **right/left/other press** press and keep pressed mouse key during action.
- **drag** or when you will find **dragging** is intended to click and keep pressed **left pres** and move mouse in the desired point.

Colored Boxes have these meanings:

This box contains an exercise

this box is a generic note.

This box is used when describing some error prone operations or common misunderstanding about **FreeCAD** use.

This box is used to explain in more deep some **FreeCAD** internal working.

This box illustrate some particular behavior of **FreeCAD** or for different behavior on different OS.

### Portions of code

This is the rendering of a code part without line numbers:

```
for obj in DOC.Objects:
    DOC.removeObject(obj.Name)
```

This is the rendering of a part of code with line numbering:

```
1 for obj in DOC.Objects:
```

This is the rendering when a line is broken by the typesetting software:

```
obj_b.Placement = FreeCAD.Placement(Vector(0, 0, 0), FreeCAD
                ↪ .Rotation(0,0,0))
```

red arrow indicates when typesetting program has done a line feed, when writing the code you have not to put that line feed but write all the code as whole line.

### Images and screenshots

To take screenshots is used the Linux version of **FreeCAD 0.19**, so they may or may not resemble that you see on your screen.

Where a proper screenshot is not needed a ”fake” dialog window is shown as follows:

FreeCAD

Dialog window text

Yes No Cancel

### Text Messages

This is a conventional representation of textual **FreeCAD** output:

```
Placement [Pos=(0,0,0), Yaw-Pitch-Roll=(0,0,0)]
```

### Source code and part of code

Source code listed in this book are original creation of the author and made purposely for this guide, syntax highlighting colors are not those used in **FreeCAD** internal python editor, some efforts where done to resemble as close as possible colors used.

Many part of the code are not working as is, but have to be inserted between other lines of code, read carefully the text explaining the code.

Usually the whole code listings are put at the end of chapter or section, they could be found also on:

https://github.com/onekk/freecad-doc

Filename is usually put also in the “preamble” of the listing or reported in the explaining text..

# Overview

From **FreeCAD** wiki:

**FreeCAD** is a general purpose parametric 3D CAD modeler.

Key word are:

- **Modeler** It's scope is to create 3D CAD models.
- **Parametric** It will create models using parameters, in other word each object has some parameters that will modify the created models, and those parameters will modify the model without the need to recreate objects from scratch.

**FreeCAD** is built around some components (libraries):

- Open Cascade Technology (OCCT), a powerful CAD kernel;
- Coin3D, a toolkit for 3D graphics development compatible with Open Inventor;
- Qt, the world-famous user interface framework;
- Python, a modern scripting language. **FreeCAD** itself can also be used as a library by other Python programs.

This guide will be focus on "Scripting", they are a form of real "programs" that will use **FreeCAD** to create 3D models.

This thing is possible because **FreeCAD** has a real **Python** interpreter on board, so you will be no limited to use a "macro language" but you have the full power of a "programming language" integrated in **FreeCAD**.

This document don't want to be an introductory guide, but instead a "programming guide", because Scripting documentation present in **FreeCAD** Wiki is very fragmented, and sometimes out of date.

Many "theoretical introductions" and "technical documentation" presented in the official documentation are written very well and by competent people, so referring to:

https://www.freecad.org/

Is not a bad idea, if anything is not clear in my writing.

## FreeCAD installation

First thing to do is getting **FreeCAD**, actual stable version is **0.19**, while development version is **0.20**.

https://wiki.freecad.org/Download

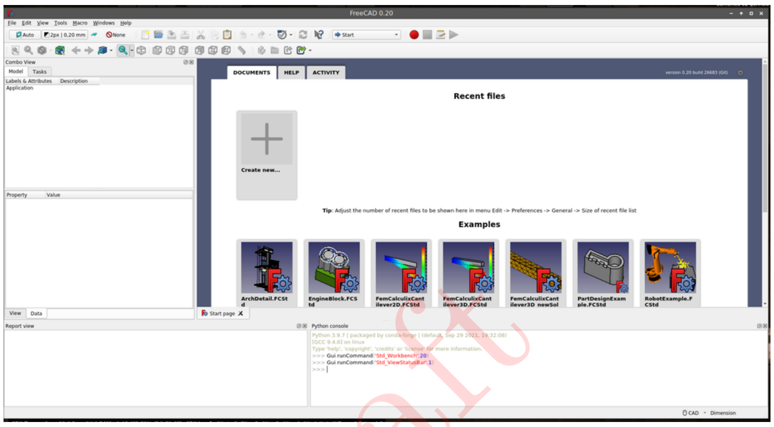
I want recommend to install “AppImages” on Linux and “Portable Builds” on Windows as they are very usable and have less problem with mismatching libraries on the OS used.

There are some drawbacks, mostly the fact that you could not use **FreeCAD** as a library with this installation method (at least on Linux with AppImages).

# Chapter 1

# First Steps

At **FreeCAD** first start we are presented with a screen similar to those in figure 1.1.



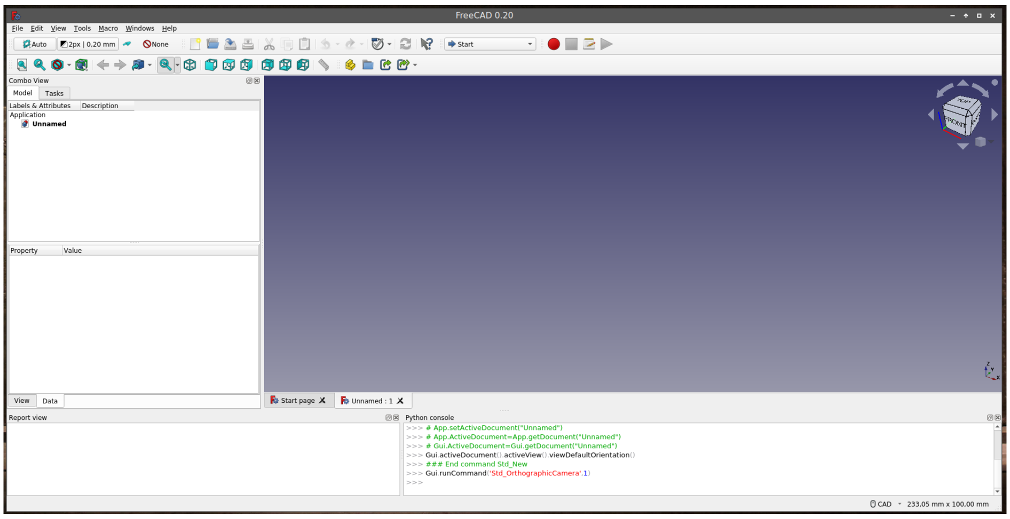
Tip: Adjust the number of recent files to be shown here in menu Edit -> Preferences -> General -> Size of recent file list

```
Python 3.9.7 | packaged by conda-forge | (default, Sep 29 2021, 19:32:08)
[GCC 9.4.0] on linux
Type 'help', 'copyright', 'credits' or 'license' for more information.
>>> Gui.runCommand('Std_Workbench',20)
>>> Gui.runCommand('Std_ViewStatusBar',1)
>>>
```

Figure 1.1.: User interface at first start

A complete tutorial on **FreeCAD** will be beyond the scope of this guide, so please refer to the official documentation present on official site.

Figure 1.2 show a figure of the user interface.



```
>>> # App.setActiveDocument("Unnamed")
>>> # App.ActiveDocument=App.getDocument("Unnamed")
>>> # Gui.ActiveDocument=Gui.getDocument("Unnamed")
>>> Gui.activeDocument().activeView().viewDefaultOrientation()
>>> ### End command Std_New
>>> Gui.runCommand('Std_OrthographicCamera',1)
>>>
```

Figure 1.2.: User interface

See:

https://wiki.freecad.org/Interface

for a detailed explanation.

We will use same terms explained in the above page, with few exceptions.

https://wiki.freecad.org/Introduction_to_Python

## 1.1. Settings

It is better to follow the directions in:

https://wiki.freecad.org/Python_scripting_tutorial

To have the interface behave better when using Scripting.

Most of these settings, seems to be now defaults, but just in case, the link above will help you to set some important things.

Maybe it will be better to remember some Python conventions:

- use spaces and not Tabs to make indentations
- use 4 spaces for each level of indentation.

Select **Edit** ⇒ **Preferences** at section **General** in **Editor** tab.

- In the box **Indentation**:

  - Put 4 in **Tab Size**.
  - Put 4 in **Indent Size**
  - Tick the option **Insert spaces** (**Keeps Tabs** will be disabled)
- In the box **Options** select:
  - **Enable line numbers**.

See also:

https://wiki.freecad.org/FreeCAD_Scripting_Basics

In **Report view** will be displayed errors and usually the standard Python trace that give an idea of what is gone wrong.

**Python Console** will show even and echo of the command issued using the GUI.

Sadly this help will be not of immediate use as it is simply a mimic of **FreeCAD** commands, that are not proper Python commands.

## 1.2. The property editor

**Property editor** has in his lower part two tabs:

- **Data tab** that contains "geometric" information about the created object.
- **View tab**, that contains "graphics" information about the created object, like "line colors" or "line Style".

We will use an abbreviated name **Data tab** when referring to the **property editor Data** Tab.

We will use also the abbreviated name of **View tab** when referring to the **View**.

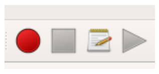
## 1.3. The macro editor

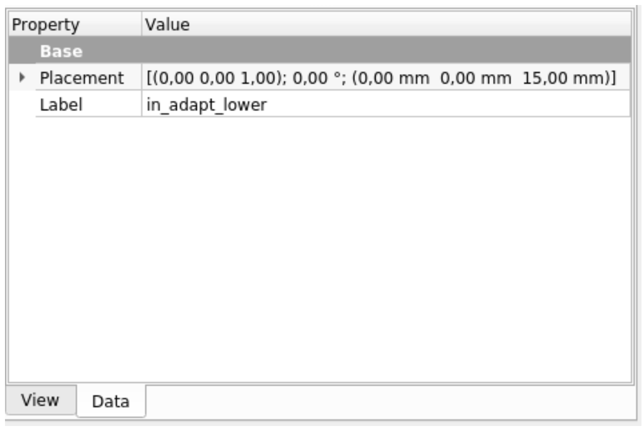
There is a decent editor built into **FreeCAD**, the **Macro editor** that is appearing when you open a file with **.py** extension or a **FreeCAD** Macro file.

It appear in place of **3D view**.

The **Macro editor** has a toolbar (maybe it has to be activated using some commands, described later) with a button (usually a Green Triangle), that launch the program shown in the editor, and show the result in the **3D view**.

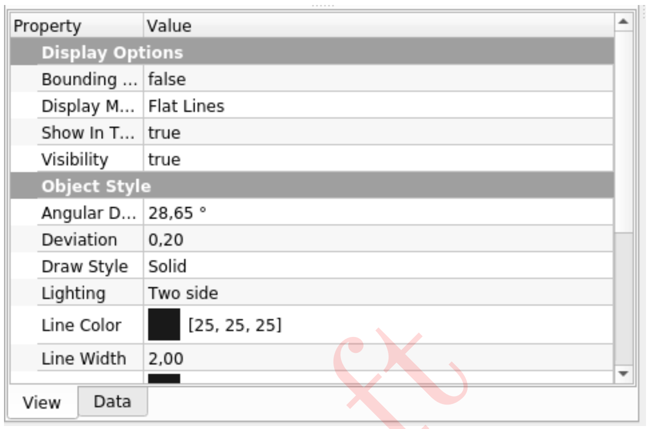
This toolbar is shown in **Toolbar area**, when you are into **Macro editor**, and resemble the image here on side, the "play" button will became "green" when is loaded a python file or a **FreeCAD** macro.





| Property | Value |
|---|---|
| **Base** | |
| Placement | [(0,00 0,00 1,00); 0,00 °; (0,00 mm 0,00 mm 15,00 mm)] |
| Label | in_adapt_lower |

(a) PE Data Tab.



| Property | Value |
|---|---|
| **Display Options** | |
| Bounding ... | false |
| Display M... | Flat Lines |
| Show In T... | true |
| Visibility | true |
| **Object Style** | |
| Angular D... | 28,65 ° |
| Deviation | 0,20 |
| Draw Style | Solid |
| Lighting | Two side |
| Line Color | ■ [25, 25, 25] |
| Line Width | 2,00 |

(b) PE View Tab.

Figure 1.3.: PE Data and View Tab

This internal editor is not very complete, as example it lacks of "search and replace" functions, so it will be better to choose a real programming editor to write scripts.

Luckily **FreeCAD** is smart enough to reload a file loaded in the editor if it detects a modification on the original file, it will ask with a popup window if you want to reload the file.

**Macro editor** at least on version **0.19** has no a "direct way" to be accessed.

To bring the editor up, as said it suffice to load a **.py** file into **FreeCAD** using **File** ⇒ **Open**.

Sometimes we will refer to the **Macro editor** as **Python editor**.

# Chapter 2

# Introduction to Scripting

There are many ways to make a Script.

We will use two approaches:

1. First approach is a sort of mimic of the GUI, we will create **Document object** for each of the entities, even for some type of "Operations".
2. Second approach is more pythonic, it create objects that are retained in memory until some of them must be shown or put in the final **FCStd file**.

## 2.1. First approach

This approach will use many call to `DOC.addObject()`, it is interesting for some reasons:

- It is easy to remember definitions as they are similar in writings.
- Each object is created as **document object** and shown in **tree view**.
- Intermediate operations are shown in the **tree view** and so intermediate "building blocks" could be shown as needed to see maybe what's is going wrong.

This approach has some drawbacks, the main, is that many objects are created in the **tree view** and each object will end in the final **FCStd file**, using memory and disk space.

But it is very educational and show some **FreeCAD** internals, so to start is a good method.

As the first steps involve creating models that are not very complex, the drawbacks are less visible.

## 2.2. Second approach

This approach avoid to create many intermediate **document objects**, and rather use directly objects methods to do operations.

It will be more lean and fast as for each object created there are lags due to **tree view** and **3D view** updates.

Only at the end and when it is really needed we will create **document objects** and visualize them.

## 2.3. Program blocks

When creating scripts, we will use “program blocks”, this is a coding approach that will break the code in many “blocks” usually created as “Python methods”.

Using this way the “program flow” will be more visible:

This approach has some advantage, one of them is that with simple “copy and paste” operations is possible to copy entire models, and shorten the length of the typed code.

In this guide we will be referring to “listings” that are presented at chapter end.

We will show when useful “code portions” and reference to the pages where code is shown.

There will be a GitHub page where you could download all the “listings”.

### 2.3.1. Brief description of the code

It may seems that there is too much “service” code, but it permits, to relaunch the execution of the code in Python Console without creating a new file and replacing existing objects with new objects.

There are some tricks to make writing code more compact, the use of:

```
from FreeCAD import Placement, Rotation, Vector
```

permit to shorten things, see these two writings:

```
# first writing without the \enquote{trick}

obj.Placement = FreeCAD.Placement(FreeCAD.Vector(0,0,0),
    ↪ FreeCAD.Rotation(0,0,0))

# using the \enquote{trick}

obj.Placement = Placement(Vector(0,0,0), Rotation(0,0,0))
```

Another trick that could speed up things, is the irregular use of some “Constants” like `ROTO` that permits to write the line in a more concise manner:

```
obj.Placement = Placement(Vector(0,0,0), ROTO)
```

These are mostly my “strange way” of coding, as I was told by many user in **FreeCAD** Forum, but take them “as is” and change if they don't fit your tastes.

## 2.4. Listings - base template

```python
"""base_tmpl.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2022
   Licence: CC BY-NC-ND 4.0 IT
"""

import os
from math import pi, sin, cos

import FreeCAD
from FreeCAD import Placement, Rotation, Vector
import Part


DOC_NAME = "test_file"

def activate_doc():
    """activate document"""
    FreeCAD.setActiveDocument(DOC_NAME)
    FreeCAD.ActiveDocument = FreeCAD.getDocument(DOC_NAME)
    FreeCADGui.ActiveDocument = FreeCADGui.getDocument(
         ↪ DOC_NAME)
    print("{0} activated".format(DOC_NAME))


def setview():
    """Rearrange View"""
    DOC.recompute()
    VIEW.viewAxometric()
    VIEW.setAxisCross(True)
    VIEW.fitAll()


def deleteObject(obj):
    if hasattr(obj, "InList") and len(obj.InList) > 0:
        for o in obj.InList:
            deleteObject(o)
            try:
                DOC.removeObject(o.Name)
            except RuntimeError as rte:
                errorMsg = str(rte)
                if errorMsg != "This object is currently not part of a document":
                    FreeCAD.Console.PrintError(errorMsg)
                    return False
    return True


def clear_DOC():
    """
    Clear the active DOCument deleting all the objects
    """
    while DOC.Objects:
        obj = DOC.Objects[0]
        name = obj.Name

        if not hasattr(DOC, name):
            continue

        if not deleteObject(obj):
            FreeCAD.Console.PrintError("Exiting on error")
            os.sys.exit()

        DOC.removeObject(obj.Name)

        DOC.recompute()


if FreeCAD.ActiveDocument is None:
    FreeCAD.newDocument(DOC_NAME)
    print("Document: {0} Created".format(DOC_NAME))

# test if there is an active document with a "proper" name
if FreeCAD.ActiveDocument.Name == DOC_NAME:
    print("DOC_NAME exist")
else:
    print("DOC_NAME is not active")
    # test if there is a document with a "proper" name
    try:
        FreeCAD.getDocument(DOC_NAME)
    except NameError:
        print("No Document: {0}".format(DOC_NAME))
        FreeCAD.newDocument(DOC_NAME)
        print("Document Created".format(DOC_NAME))

```

```
DOC = FreeCAD.getDocument(DOC_NAME)
GUI = FreeCADGui.getDocument(DOC_NAME)
VIEW = GUI.ActiveView

activate_doc()

clear_DOC()

ROTO = Rotation(0,0,0)

### CODE START HERE ###
```

# Chapter 3

# 3D Solids

The goal of this guide is to create **solids**.

These **solids** are created in a 3D space.

But to complicate things, there are many way to represent a 3D space, we will descrive in a very short manner the conventions adopted by **FreeCAD**.

## 3.1. 3D Spaces

In figure figure 3.1 we could see a schematic representation of a 3D space.

Each point in 3D space id defined using three numbers that are coordinates for each axis (X, Y, Z).

In this guide, if not indicated in other way, when we see such writing `(0, 0, 0)`, it means that we are referring to a position in the 3D space.

To not overcomplicate writing at this early stage, we will present coordinates made by integer, but obviously floating point numbers are admitted and usually used.

Axis convention is the same used in **FreeCAD** as seen in **3D view**.

Figure is showing a 3D space representatio, with a cube and the eight points that will define the cube.

Each point is defined using three numbers using convention explained above.

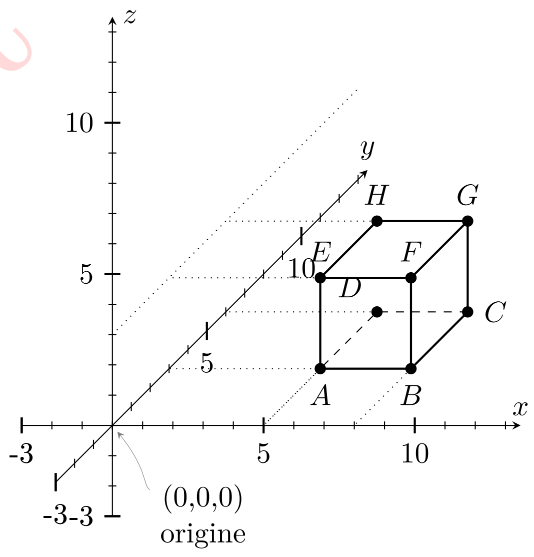


Figure 3.1.: 3D space

### 3.1.1. Vectors

**Vectors** are used in **FreeCAD** in many ways.

This writing `Vector(val_x, val_y, val_z)` is the most common one.

But in **FreeCAD** a vector is used also to contain values that are not 3D coordinates, but rather angles of rotations, or maybe "directions" for some 2D geometries.

Vectors are used because with vectors is is possible to make some complex operations.

A little analysis of the vector object could be done writing these line in **Python Console**.

```
a = FreeCAD.Vector(10,10,0)
print(dir(a))
```

And we will see in the **Report View**:

```
['Length', '__abs__', ... ... '__xor__', 'add', 'cross', '
    ↪ distanceToLine', 'distanceToLineSegment', '
    ↪ distanceToPlane', 'distanceToPoint', 'dot', '
    ↪ getAngle', 'isEqual', 'isOnLineSegment', 'multiply
    ↪ ', 'negative', 'normalize', 'projectToLine', '
    ↪ projectToPlane', 'scale', 'sub', 'x', 'y', 'z']
```

We could remark some things:

- At list end we read `'x', 'y', 'z'`, this will be useful to access single axis values contained in vector using as example to obtain **X** coordinate of **Vector a** this writing: `a.x`.
- There are some methods, like `'distanceToLine'` that will be useful to obatain some "lengths" between a vector and other entities.
- A vector has also some "operations" like `negative, scale, normalize` that reveal that a vector is not only a simple indication of a "coordinate in 3d space"

## 3.1.2. Data e View

This separation is very deep in **FreeCAD**.

There are at least two different components, in **FreeCAD**, each one is managed by a different "engine".

Every **FreeCAD** object (to be more precise every **document object**) is composed by:

- An **Object** component that contains objects data used by "modeling engine" that is shown in **property editor** in the **View tab**. This part is often referred as the **App** library and is related to the OCCT engine.
- A **ViewObject** component that contains view data, used by the "graphics engine", as example color and transparency; This component is shown in **property editor** in the **Data tab**. This part is managed by the **Gui** library and managed by Coin3D.

In listing **base_tmpl** we could see the use of the Gui part in **setview()** method, that is using some methods accesible through `FreeCAD.Gui`.

In **3D view** some curved objects could resemble as they are made using polygons.

This behaviour is related to the "View" component, to speed up graphic *rendering*, some approximation are setting some representation tolerances.

**FreeCAD** "modeling engine" (OCCT), will define curves using mathematical formulas, so a cruve is even a curve, each point has a precise formula.

We could tune, using appropriate parameters this "approximation" done by Coin3D:

- For the whole program, using values present in: **Edit** ⇒ **Preferences**, section **Part design** Tab **Shape view**, box **Tessellation**.
- For a single object, in **View tab**.

Parametrs that tune this approximation are:

- **Maximum deviation** expressed as a percentage value:
  - For the whole program in **Maximum deviaton depending on the model bounding box**.
  - For a single object, using the property **Deviation**.
- **Angular deflection** expressend in degree (default value is 28.5 ° 0.5 radians is tunable using:
  - For the whole program using the value **Maximum angular deflection**
  - For a single object, using the property **Angular Deflection**.

Some people suggest to modify these values as follows:

- **Deviation** 0.100 %
- **Angular Deflection** 5°.

## 3.2. Geometry, Topology and other things

One of the most difficult thing to learn about **FreeCAD** seems the distinction between Geometry and Topology.

Same concept could be declined in a different way, the "point of view is different" so the object is different.

### 3.2.1. Geometry

Geometry, could be thought as the most "low level" component of the modeling engine.

It permit to define simple enities.

- **Points** is a coordinate in the space 2D o 3D, it has no dimensions.

- **Curves** the most know example is a a **circle**, it has a precise mathematical representation, that permit to supply only two values: a **center** and a **radius**, the points that defines the curve could be calculated using these values and a math formula. Same thing if we want to define a **line** or better a **segment**, it suffice to define two points.
- **Surfaces** as example a **piano**, but also a more complex **surfaces** like a **Surfaces** BSpline.

### 3.2.2. Topology

Topology is the way **solids** are described in 3D space.

We could think 3 ”base components” of each solid:

- **Vertex**: A topological element corresponding to a **point**. It has zero dimension..
- **Edge**: A topological element corresponding to a restrained curve. An **edge** is generally limited by **vertexes**. It has one dimension.
- **Face**: In 2D it is part of a plane; in 3D it is part of a **surfaces**. Its geometry is constrained (trimmed) by **edges**. It is two dimensional.

These **TopoShapes** could be grouped to make more complex things:

- **Wire**: a series of **edges** connected by their **vertexes**. It can be an open or closed contour depending on whether the **edges** are linked or not.
- **Shell**: A set of **faces** connected by their **edges**. A shell can be open or closed.
- **Solid**: A part of space limited by **shell**. It is three dimensional.

## 3.3. Geometry and topology in FreeCAD

We could extablish some hyerarchies between **FreeCAD** objects:

- **Geometric primitives**
- **TopoShape**
- **Document object**

These things are “hidden” when using GUI, because every object created using GUI is created in 3D view, and became an **Document object**.

In OpenCascade terminology, that is **FreeCAD** “modeling engine”, there is a distinction between ”geometric primitives” and ”TopoShape”.

In table table 3.1 on the next page we will try to list most common ”geometric primitives” you will find in **FreeCAD**.

| Geometry | Subtype | |
|---|---|---|
| **point** | | |
| **curve** | | |
| | **line** | |
| | **circle** | **ellipse** |
| | **parabola** | **hyperbola** |
| | **Bezier curve** | **B-Spline curve** |
| **surface** | | |
| | **B-Spline surface** | **Bezier surface** |
| | **plane** | |

Table 3.1.: Geometrie

Starting from **FreeCAD** version 0.17, there is a distinction between a **Line** obtained using `Part.Line` and a **Segment** obtained using `Part.LineSegment`, this reflects more closely the geometry where a line is extending in infinite directions and is defined as passing to "two points", while a segment, is a "portion" of line **limited** by two points.
In the web and even in some documentation on FreeCAD there are many examples of code that don't work simply because they are using pre 0.17 definition of `Part.Line`.
To make them work, it usually suffice to substitute `Part.Line` with `Part.LineSegment`.

A TopoShape could be a **vertex**, an **edge**, a **wire**, a **face**, a **solid** or a compound of **TopoShapes**.

Geometric primitives are not made to be visualized, but rather to be used as building elements of **TopoShapes**. As example a **edge** could be a **line** but also a portion of circle (Arc).

This concept could lead to think that **FreeCAD** is complex and involuted.

But it is not a real hassle, it could be illustarted using some examples that could be typed even in the Python Console

We will simply use print function:

```
circle = Part.Circle()

print(circle.TypeId)

circle.Center = Vector(0, 0, 0)
circle.Radius = 10
```

```
cir1 = circle.toShape()

print(cir1.TypeId)

Part.show(cir1)
```

We will see in **Report View**:

```
Part::GeomCircle
Part::TopoShape
```

This style of coding, could even be more compact, but this is more meaningful, as every action is clearly shown:

- Create an empty “geometric primitive” `circle`
- Printed his **TypeId** that is **Part::GeomCircle**.
- Assigned proper values at **Center** and **Radius**.
- Transformed the “geometric primitive” in a **TopoShape** using the method **.toShape()**.
- Printed his **TypeId** to show thathas became **Part::TopoShape**.
- Visualized **TopoShape** using **.show()**, to create a **Document object**.

A **Document object** is an entity that is visualized in 3D view and is saved in **.FCStd** file.

In this guide we will use very often **Document objects** created as **Part::Feature**; we will assign a proper **TopoShape** to the object **Shape** property.

## 3.4. Modeling

**FreeCAD** could use two ways of costructing a solid:

- CSG it uses some base solids called **primitives** and operate on them using **Boolean operation**.
- BREP it create **faces** join them to form **shells** to limit a portion of space to create **solids**.

In the prosecution of this guide, we will present some code, if not indicated, this code listings has to be added to the code presented in section 2.4 on page 9 named **Listings - base template** after the line:

```
### CODE START HERE ###
```

# Chapter 4
# CSG Modeling

## 4.1. First objects

Let's introduce the first object, or better, first **primitive**, a parallelepiped, in **FreeCAD** the object is **Part::Box**;

To build our **solid**, let's write these lines:

```
def base_cube(name, lng, wid, hei):
    obj_b = DOC.addObject("Part::Box", name)
    obj_b.Length = lng
    obj_b.Width = wid
    obj_b.Height = hei

    DOC.recompute()

    return obj_b
```

This method will return an object, a parallelepiped, (cube is a special case of a parallelepiped).

Method use 4 parameters:

1. **name** object name, this name will appear in **Combo view** into **tree view**.
2. **lng** object length.
3. **lwid** object width.
4. **hei** object height.

```
obj = base_cube("test_cube", 5, 5, 5)

setview()
```

If we launch the program using "green arrow"a cube will be displayed in **3D view**.

Cube is done, our first solid, is shown on 3D view. To manage clearly things, we have to "orient" in 3d space:

Program 4.1: cylinder method

```
def base_cyl(name, ang, rad, hei):
    obj = DOC.addObject("Part::Cylinder", name)
    obj.Angle = ang
    obj.Radius = rad
    obj.Height = hei

    DOC.recompute()

    return obj
```

- We are passing some values to the build methods, to what axise they refer?
- What is object **Position** in 3d Space?

To answer to the second question, it suffice to look at the arrows that indicate axis directions. We have activate them in **setview()** methods, with:

```
VIEW.setAxisCross(True)
```

This command is controlling the View part of **FreeCAD** managed by Coin3D, and visualize the "origin" and the positive directions of each axis:

It is same action you do when select **View** ⇒ **Toggle Axis cross** menu item.

To answer to first question, and to make some exercise, try to change values in line:

```
obj = base_cube("test_cube", 5, 5, 5)
```

And try to see what `lng`, `wid` e `hei`, are related to axis `X`, `Y` e `Z`.

To make things much interesting another **primitive** is needed.

Let's create a different **primitives**, a cylinder; Using **FreeCAD** terminology an object of type **Part::Cylinder**.

We will insert immediately after `base_cube` method, lines in listing 4.1 named **cylinder method**, that contains a new method.

Method named `base_cyl` will create our new **solid**;

We will invoke the new method writing:

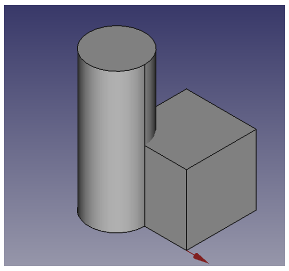
```
obj_1 = base_cyl("test_cylinder", 360, 2, 25)
```

This is very similar to the `base_cube` method:

1. `name` object name.

2. **ang** angole passed to the **Angle** property.
   We could assign values lower than 360 to make portion of cylinder.
3. **rad** cylinder radius passed to **Radius** property.
4. **hei** cylinder height passed to **Height** property.

Now we should have a script that resemble **Listing - base-objects.py** in Section 4.1.1, che una volta lanciato dovrebbe mostrare nella **3D view**, qualcosa che assomiglia a:



### 4.1.1. Listing - base-objects.py

You will find the complete listing on GitHub page in `base-objects-full.py`

```
#
"""base-objects.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2022
   Licence: CC BY-NC-ND 4.0 IT

     Attenzione: Questo listato va usato aggiungeno le linee
     da 18 in poi al codice presente in sc-base.py

     Warning: This code has to be adding the lines starting
     from 18 to the code in sc-base.py
"""

def base_cube(name, lng, wid, hei):
    obj_b = DOC.addObject("Part::Box", name)
    obj_b.Length = lng
    obj_b.Width = wid
    obj_b.Height = hei

    DOC.recompute()

    return obj_b

def base_cyl(name, ang, rad, hei):
    obj = DOC.addObject("Part::Cylinder", name)
    obj.Angle = ang
    obj.Radius = rad
    obj.Height = hei

    DOC.recompute()

    return obj


obj = base_cube("test_cube", 5, 5, 5)
obj_1 = base_cyl("test_cylinder", 360, 2, 10)

setview()
```

# Chapter 5
# Placement

See also:

https://wiki.freecad.org/Placement

At a first glance what we have obtained running the code could seem strange.

- Cube is created with lower left, angle on `0,0,0`
- Cylinder is created instead with the center of the bottom face in `0,0,0`

This is called in **FreeCAD** "Reference Point".

## 5.1. Reference Point

In table 5.1, we will list some **primitives** "Reference Point".

| Geometry | Reference Point |
|---|---|
| **Part::Box** | **vertex** left (min x), front (min y), lower (min z) |
| **Part::Sphere** | Center of the **Part::Sphere** (center of bounding box) |
| **Part::Cylinder** | Center of the bottom face |
| **Part::Cone** | Center of the bottom face (or apex if bottom radius is 0) |
| **Part::Torus** | Center of the torus |
| **Part::Wedge** | Xmin Zmin **vertex** |

Table 5.1.: Reference point

Some **primitives** have "Reference Point" in a position that is very simple to use, some other like parallelepiped have "Reference Point" in a "peculiar" place, we could to deal with it in two ways:

- Take it in account when calculating a new **Placement**.
- Create directly an appropriate **Placement** into **primitive** creation method.

A little example.

We will slightly modify listing section 4.1.1 on page 20 named **Listing - base-objects.py**, altering method `base_cube` as follows:

```
def base_cube(name, lng, wid, hei, cent = False, off_z = 0):
    obj_b = DOC.addObject("Part::Box", name)
    obj_b.Length = lng
    obj_b.Width = wid
    obj_b.Height = hei

    if cent == True:
        posiz = Vector(lung * -0.5, larg * -0.5, off_z)
    else:
        posiz = Vector(0, 0, off_z)

    rot_c = VZOR # Rotation center
    rot = ROTO # Rotation angles
    obj_b.Placement = FreeCAD.Placement(posiz, rot, rot_c)
    DOC.recompute()

    return obj_b
```

We have added two parameters:

- `cent` with a default value of `False`.
- `off_z` described later.

Use of "optional" parameter will permit to reuse all the preceding written code without altering invocations, and to add new behavior to the method.

When it is required to "center" the cube around origin it's a matter of simply add a `True` after the usual parameters.

Following lines:

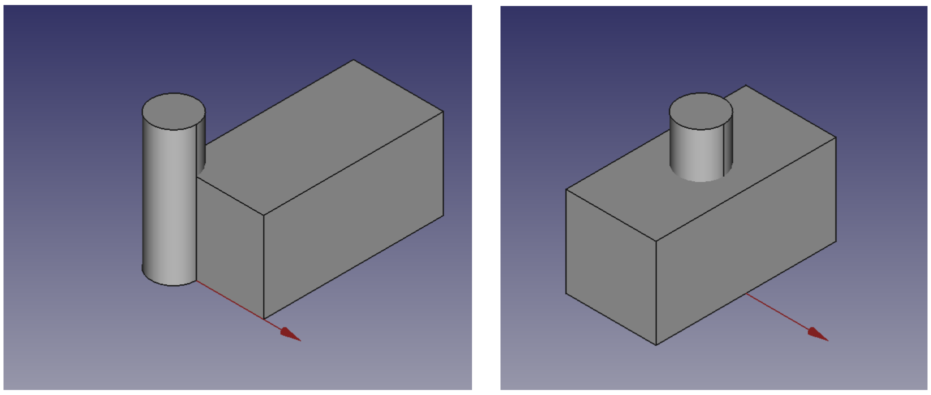
```
obj = base_cube("test_cube_cent", 10, 20, 10)
obj_1 = base_cyl("test_cylinder", 360, 2.5, 15)
```

Will produce result in figure 5.1a.

These lines:

```
obj = base_cube("test_cube_cent", 10, 20, 10, True)
obj_1 = base_cyl("test_cylinder", 360, 2.5, 15)
```

Will produce result shown in figure 5.1b in which cube is centered around the origin.



(a) Cube and cylinder with standard Reference Point.

(b) Cube and cylinder with modified Placement.

Figure 5.1.: Placement

## 5.2. Positioning

We have added to **test_cube** a second parameter **off_z**, it is used to modify Z position of **primitive**.

If we modify the code as follows:

```
obj = base_cube("test_cube_cent", 10, 20, 10, True, 10)
obj_1 = base_cyl("test_cylinder", 360, 2.5, 15)

print("Test Cube Placement = ", obj1.Placement)
print("Test Cylynder Placement = ", obj2.Placement)
```

And relaunch the script we will see that **test_cube**, is raised from origin. This is made by **off_z** that has now the value of 10.

If we look in **Report View** we will see that are printed **Placement** properties, of both objects.

```
Test Cube Placement =  Placement [Pos=(-5,-10,10), Yaw-Pitch
      ↪ -Roll=(0,0,0)]
Test Cylynder Placement =  Placement [Pos=(0,0,0), Yaw-Pitch
      ↪ -Roll=(0,0,0)]
```

### 5.2.1. Placement Property

**Placement** property is somewhat complicated as it has a variety of writings, we have specified this property in **test_cube** as:

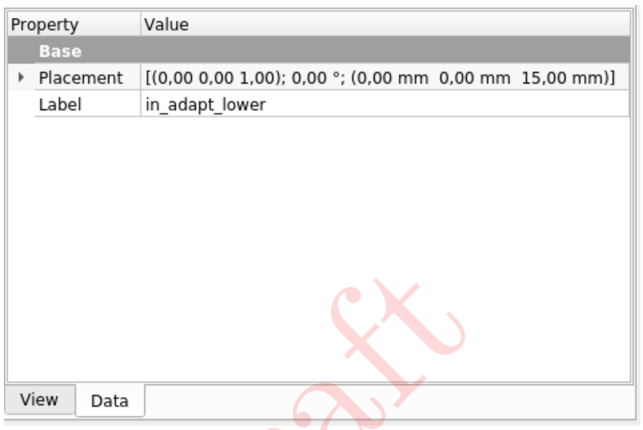
```
    rot_c = VZOR # Rotation center
    rot = ROTO # Rotation angles
    obj_b.Placement = FreeCAD.Placement(posiz, rot, rot_c)
    DOC.recompute()
```

This is one of the many ways to write this property, we have used the shortcuts illustrated before, to shorten definition line.

One of the most seen writing around is:

```
FreeCAD.Placement(
    Vector(pos_x, pos_y, pos_z),
    Rotation(Vector(axis_x, axis_y, axis_z), ang)
    )
```

This writing is very similar to the property present in **Combo view** into **property editor** as you could see in the **Data tab** of figure 5.2.



| Property | Value |
|---|---|
| **Base** | |
| Placement | [(0,00 0,00 1,00); 0,00 °; (0,00 mm 0,00 mm 15,00 mm)] |
| Label | in_adapt_lower |

Figure 5.2.: Property Editor

Item order is different from **View tab** to the way they are expressed in code, we could see that **Placement** has an arrow on left side that when clicked will show expanded parameters like in figure 5.2 and show three sub-properties, **Angle**, **Axis**, **Position**, that are explained in table 5.2.

| Name | Variable | Description |
|---|---|---|
| **Angle** | **ang** | rotation angle in degrees |
| **Axis** | **axis_x**, **axis_y**, **axis_z** | **Vector** containing values 0 o 1 about reference axis (X, Y and Z), bigger values are allowed. |
| **Position** | **pos_x**, **pos_y**, **pos_z** | **Vector** of Translation (X, Y, Z) |

Table 5.2.: Placement property

But surprisingly this writing is not what we will have when printing object Placement property:

```
Test Cube =  Placement [Pos=(-5,-10,10), Yaw-Pitch-Roll
       ↪ =(0,0,0)]
```

We could identify a component named `Pos`, that is the “Translation Vector” followed by a **tuple** of three values named `Yaw-Pitch-Roll`.

This Placement writing use the **Tait-Bryan angles**, names (Yaw, Pitch and Roll), are related to aerospace and nautical navigation:

| Nome | Description | Angle |
|---|---|---|
| Yaw | rotation around Z | **Psi** $\psi$ |
| Pitch | rotation around Y | **Phi** $\varphi$ |
| Roll | rotation around X | **Theta** $\theta$ |

Table 5.3.: Tait-Bryan Angles

If we use this writing for a **Placement**:

```
FreeCAD.Placement(
    Vector(pos_x, pos_y, pos_z),
    Rotation(Yaw, Pitch, Roll),
    Vector(c_rot_x, c_rot_y, c_rot_z)
    )
```

We are passing three different entities described in table 5.4

| Entity | Description |
|---|---|
| `pos_x, pos_y, pos_z` | Translation **Vector**. |
| `Yaw, Pitch, Roll` | Rotation angles. |
| `c_rot_x, c_rot_y, c_rot_z` | Rotation center. |

Table 5.4.: Rotation expressed using Tait-Bryan angles

There are other consideration about **Placement**, but for now let’s stop here.

In GUI we have a way to access to the different ways to specify a **Placement**, when you have an object selected in tree view chosen menu item **Edit**⇒ **Placement** will open a dialog.

During scripting, especially during some test it very useful to insert some `print()` instructions in code.

As example to visualize some object property, or to inspect some variable values:

```
print("My_value = ", my_value)
```

### 5.2.2. Listing - Reference point

```
#
"""ref-pnt.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2022
   Licence: CC BY-NC-ND 4.0 IT

    Attenzione: Questo listato va usato aggiungeno le linee
    da 18 in poi al codice presente in sc-base.py

    Warning: This code has to be adding the lines starting
    from 18 to the code in sc-base.py
"""


def base_cube(name, lng, wid, hei, cent = False, off_z = 0):
    obj_b = DOC.addObject("Part::Box", name)
    obj_b.Length = lng
    obj_b.Width = wid
    obj_b.Height = hei

    if cent == True:
        posiz = Vector(lung * -0.5, larg * -0.5, off_z)
    else:
        posiz = Vector(0, 0, off_z)

    rot_c = VZOR # Rotation center
    rot = ROTO # Rotation angles
    obj_b.Placement = FreeCAD.Placement(posiz, rot, rot_c)
    DOC.recompute()

    return obj_b


def base_cyl(name, ang, rad, hei):
    obj = DOC.addObject("Part::Cylinder", name)
    obj.Angle = ang
    obj.Radius = rad
    obj.Height = hei

    DOC.recompute()

    return obj


obj = base_cube("test_cube_cent", 10, 20, 10, True, 10)
obj_1 = base_cyl("test_cylinder", 360, 2.5, 15)

print("Test Cube Placement = ", obj1.Placement)
print("Test Cylynder Placement = ", obj2.Placement)

setview()
```

# Chapter 6
# Boolean Operations

A complex geometry could be created in many ways. One of the most "old" way is Costructive Solid Geometry, that is used also in **FreeCAD**.

This way of building is using **primitives**, and modify them using some "operations", most used are **Boolean operation**.

Let's introduce with **Boolean operation**:

| Name | Description |
|---|---|
| **Fusion** | Part::Fuse or Part::MultiFuse |
| **Subtraction (cut)** | Part::Cut |
| **Intersection** | Part::MultiCommon |

This explanation assumes a workflow that use the base code in **Listing - base-objects.py** in Section 4.1.1, completing it with code lines presented in **Listing - Boolean Operation** in Section 6.4. Line numbers shown here are referring to the code in `ob-ex.py`.
You will find the complete example in `ob-ex-full.py` on GitHub "code" page.

## 6.1. Union

Union boolean operation is sometimes called **fusion**, so when using the term fuse objects, we are speaking about Union boolean operation.

We will fuse the solids created in preceding chapter using Part::Fuse, as usual we create a method presented in **Fusion** in Listing 6.1.

This portion of code has to be inserted after method `base_cyl`.

Here a quick description of Part::Fuse properties:

- Base it's the object from which we add things.
- Tool it's the object to add.

Program 6.1: Fusion

```
13 def fuse_obj(name, obj_0, obj_1):
14     obj = DOC.addObject("Part::Fuse", name)
15     obj.Base = obj_0
16     obj.Tool = obj_1
17     obj.Refine = True
18     DOC.recompute()
19
20     return obj
```

Program 6.2: Multifuse

```
23 def mfuse_obj(name, obj_0, obj_1):
24     obj = DOC.addObject("Part::MultiFuse", name)
25     obj.Shapes = (obj_0, obj_1)
26     obj.Refine = True
27     DOC.recompute()
28
29     return obj
```

- using property **Refine** set to `True` will make a "refinement" of the resulting object eliminating "seams".

To use it is sufficient to write:

```
fuse_obj("cubo-cyl-fu", obj, obj1)
```

after:

```
obj_1 = base_cyl('test_cylinder', 360,2,10)
```

A side note is that objects passed to `fuse_obj` must be **Document objects**.

Resulting is shown in figure 6.1b on page 32, not much different from the image in figure 6.1a on page 32, but we could see that in Combo view original objects are disappared and his appear a different thing named `fusion_cube_cyl`.

Objects are not disappeared, but are become part `fusion_cube_cyl`.

If we click on small arrow ▶ it became ▼ and original objects names will appear, in light grey, this indicate that they are "components" of the object `fusion_cube_cyl`.

Union operation has also another form, Part::MultiFuse that could be used when is needed to fuse more then two objects, like in code presented in Listing 6.2.

We could note at line 25 property **Shapes** with a **tuple** that contains our objects.

Program 6.3: Sustraction

```
def cut_obj(name, obj_0, obj_1):
    obj = DOC.addObject("Part::Cut", name)
    obj.Base = obj_0
    obj.Tool = obj_1
    obj.Refine = True
    DOC.recompute()

    return obj
```

Program 6.4: Intersection

```
def int_obj(name, obj_0, obj_1):
    obj = DOC.addObject("Part::MultiCommon", name)
    obj.Shapes = (obj_0, obj_1)
    obj.Refine = True
    DOC.recompute()

    return obj
```

## 6.2. Subtraction

Subtraction is called also **cut** and is done in **FreeCAD** using, **Part::Cut**, as in Listing 6.3.

we will use it as follows:

```
c_obj = cut_obj("cut-cube-cyl", obj, obj_1)
```

We will obtain something like in figure 6.1c on the following page.

Object properties are same as in **Part::Fuse**.

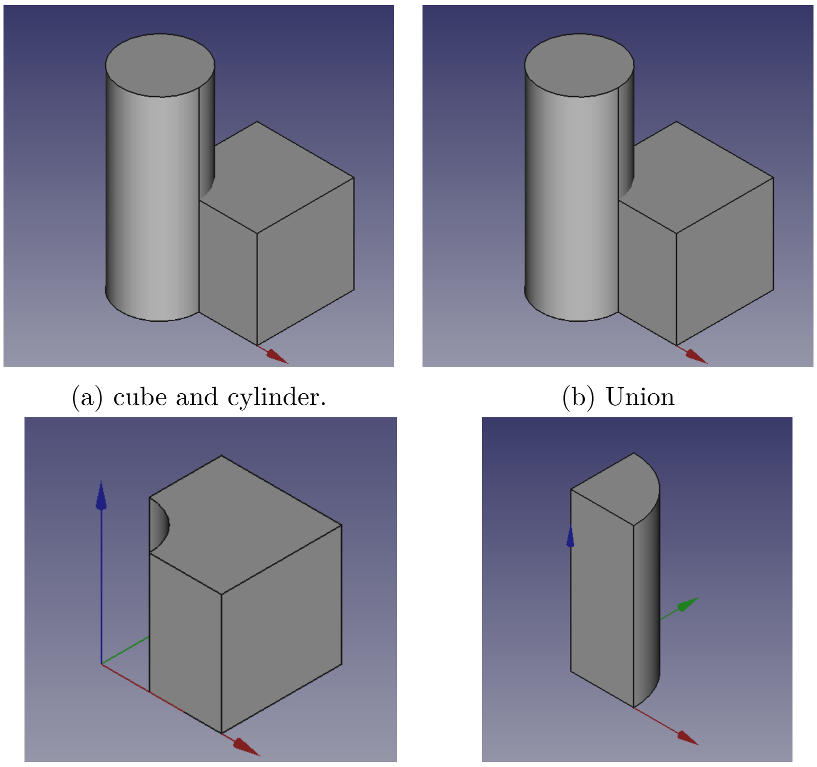
- **Base** Object from which we subtract other object.
- **Tool** object to subtract.
- **Refine** as above in **Part::Fuse**.

## 6.3. Intersection

Intersection, is done using an object of type **Part::MultiCommon**, it compute the "common" part of the supplied objects.

We will see the code in Listing 6.4.

We use it as follows:



(a) cube and cylinder. (b) Union

(c) Subtraction. (d) Intersection.

Figure 6.1.: Boolean Operations.

```
i_obj = int_obj("is-cube-cyl", obj, obj_1)
```

We obtain something as in figure 6.1d.

It's writing is very similar to those seen in **Part::MultiFuse**.

## 6.4. Listing - Boolean Operation

```
#
"""ob-ex.py

   This code was written as an sample code
   for "FreeCAD Scripting Guide"

   Author: Carlo Dormeletti
   Copyright: 2022
   Licence: CC BY-NC-ND 4.0 IT
"""


```

```
def fuse_obj(name, obj_0, obj_1):
    obj = DOC.addObject("Part::Fuse", name)
    obj.Base = obj_0
    obj.Tool = obj_1
    obj.Refine = True
    DOC.recompute()

    return obj


def mfuse_obj(name, obj_0, obj_1):
    obj = DOC.addObject("Part::MultiFuse", name)
    obj.Shapes = (obj_0, obj_1)
    obj.Refine = True
    DOC.recompute()

    return obj


def cut_obj(name, obj_0, obj_1):
    obj = DOC.addObject("Part::Cut", name)
    obj.Base = obj_0
    obj.Tool = obj_1
    obj.Refine = True
    DOC.recompute()

    return obj


def int_obj(name, obj_0, obj_1):
    obj = DOC.addObject("Part::MultiCommon", name)
    obj.Shapes = (obj_0, obj_1)
    obj.Refine = True
    DOC.recompute()

    return obj


obj = base_cube("test_cube", 5, 5, 5)

obj_1 = base_cyl('test_cylinder', 360, 2, 10)

f_obj = fuse_obj("fusion-cube-cyl", obj, obj_1)
f_obj.Placement = Placement(Vector(20, 0, 0), ROT0)

mf_obj = mfuse_obj("multifusion-cube-cyl", obj, obj_1)
```

```
mf_obj.Placement = Placement(Vector(20, 20, 0), ROT0)

c_obj = cut_obj("cut-cube-cyl", obj, obj_1)
c_obj.Placement = Placement(Vector(40, 0, 0), ROT0)

i_obj = int_obj("is-cube-cyl", obj, obj_1)
i_obj.Placement = Placement(Vector(40, 20, 0), ROT0)

setview()
```

# Chapter 7
# BREP modeling

This approach is more “modern” and is totally different from CSG.

To make things, we have to leave the approach used until now, and we must use a different paradigm.

Until now we have use the “first approach” explained in section 2.1 on page 7.

Here we will use “second approach” explained in section 2.2 on page 7.

Complete code could be found in `ext-full.py` on GitHub “code” page.

Let’s put following code after:

```
### CODE START HERE ###
```

Numbering will reflect line numbers starting from `base_tmp.py` code and adding subsequent lines from the above line.

Program 7.1: `base_figure` method

```
def base_figure():
    """Create a polygon."""
    points = (
        (0.0, 0.0, 0.0),
        (15.0, 0.0, 0.0),
        (15.0, 10.0, 0.0),
        (0.0, 10.0, 0.0),
        (0.0, 0.0, 0.0)
    )

    obj = Part.makePolygon(points)

    return obj
```

we create a polygon, or to be more precise a rectangle, but we create it using **Part.makePolygon()**.

It is a rather powerful method that permit to supply a sequence of coordinates and obtain a **Wire**

This sequence must respect some rules:

- Starting point and ending point must have same coordinate, because polygon must be **closed**.
- points must be ordered, I usually use counterclockwise order as some **Part::TopoShape** like **Part::GeomCircle** use counterclockwise convention to specify angles.

**Part.makePolygon()** method is very versatile and accept different sequence types it accepts.

- list of **vectors**.
- list of coordinates.
- tuple of coordinates.

In the above explanation coordinates are intended as tuples of three values representing XYZ coordinates in 3D Space.

We use this new method simply invoking it as usual with:

```
  obj_o = base_figure()

# Part.show(obj_o, "Wire")
```

We have now a **Wire**, but to create a **solid** we need a **Face**:

```
obj_f = Part.Face(obj_o)

# Part.show(obj_f, "Face")
```

This **Face** will be the base for some "operations".

In code we will leave some line commented like:

```
# Part.show(obj_o, "Wire")
```

They are intentionally left to permit a visualization of intermediate objects, simply deleting the **#** to see them.

One thing that we will note is that nothing is created yet in **3D view**, as we have not yet issue "visualization" command.

## 7.1. Extrusion

First "operation" we will see is **extrude**.

To make this "magic" we will write:

```
thi = 10

sol_ext = obj_f.extrude(Vector(0, 0, thi))

Part.show(sol_ext, "Extruded_Solid")
```

```

setview()
```

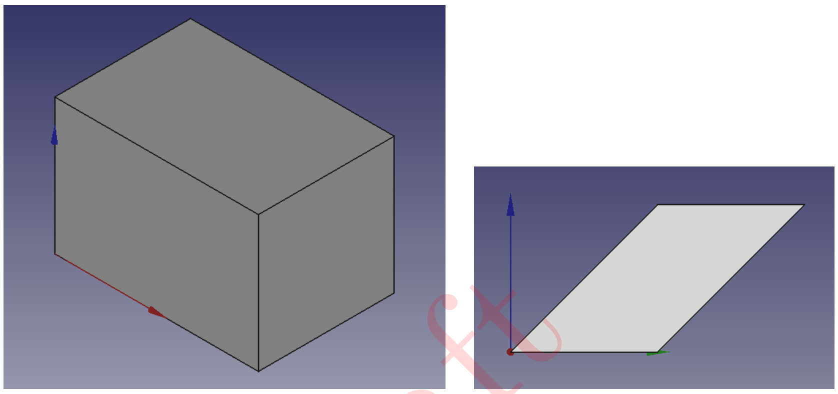
Code line 127 is performing  “operation”, note that we have to pass a **vector** as argument.

We have passed a simple **vector**, `Vector(0, 0, thi)`, where `thi` has been defined in line 125.

This will be very useful as it introduce some “parametrization”, in other words, we could make some **lengths** depend on others.

Some interesting things, could be achieved if you pass a different **vectors**, like `Vector(0, 10, 10)`.

In this case we obtain this figure 7.1b, view from **Right**, to properly show the “slanting”.



(a) Extruded Solid. (b) Slanted Solid.

Figure 7.1.: Extrusion

## 7.2. Revolution

No it is not the Beatles song.

It is another “operation” that could be done.

We reuse the preceding code, modifying it a little.

The most easy way is to copy and rename the existing file and delete all the content after line 123.

Complete code could be found in **`rev-full.py`** on GitHub “code” page.

```
125 # Base point of the rotation axis
126 pos = Vector(0, 20 ,0)
127 # Direction of the rotation axis
128 vec = Vector(1 ,0, 0)
129 # Rotation angle
130 angle = 45
131
132 sol_rev = obj_f.revolve(pos, vec, angle)
133
134 Part.show(sol_rev, "Revolved_Solid")
135
136 setview()
```

When we launch the code we will obtain something like in figure 7.2a on the following page.

we have introduced a new operation, **revolve** that has to be fed with three parameters:

- `pos` base point of the rotation axis.
- `vec` direction of the rotation axis
- `angle` Rotation angle

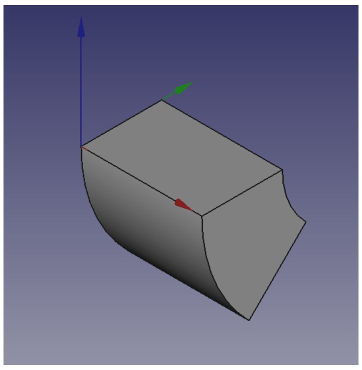
Not much to explain a part from `vec` that contains a “direction” **vector**.

This sort of writing is used in **FreeCAD** in many places, usually it contains 0 and 1 values, to indicate which axis has to be used. but it could contains also other numbers.

It could be even fed with something like:

```
# direction of the rotation axis
vec = Vector(0.5, 0.5, 0)
# Rotation angle
angle = 270
```

To obtain something like in figure 7.2b on the next page


(a) Revolved Solid rotated around only one axis.

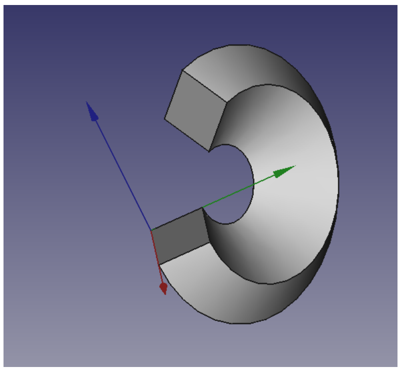

(b) Revolved Solid rotated around different axis.

Figure 7.2.: Revolution

## 7.3. Loft

To make use of the next operation we have to modify sligtly **base_figure**.

**Part.makeLoft()** is a complex operation, and sometime may fail, so we have to adopt some cautions, see below for some caveats.

We reuse the preceding code, modifying it a little.

The most easy way is to copy and rename the existing file and delete all the content after line 115.

Complete code could be found in **loft-full.py** on GitHub "code" page.

First of all we must redefine sligthly **base_figure** method, as follows

Program 7.2: **base_figure ver2**

```
102 def base_figure(dim_x, dim_y):
103     """Create a polygon."""
104     points = (
105         (0.0, 0.0, 0.0),
106         (dim_x, 0.0, 0.0),
107         (dim_x, dim_y, 0.0),
108         (0.0, dim_y, 0.0),
109         (0.0, 0.0, 0.0)
110     )
111
112     obj = Part.makePolygon(points)
113
114     return obj
```

These modifications permits to obtain returned object with different dimensions.

Next we have to prepare a container for the list of created objects

```
# elements hold created wires
elements = []
```

We create a tuple that hold some values needed to define objects, note the use of comments to describe things, it is a good practice because it improve readability and long term maintainability of produced code.

```
# dimensions hold values that are (dim_x, dim_y, z height)
                  ↪ of generating figures
dimensions = (
    (5.0, 5.0, 0.0),
    (5.0, 5.0, 5.0),
    (8.0, 8.0, 7.0),
    (10.0, 10.0, 10.0),
    (8.0, 8.0, 13.0),
    (5.0, 5.0, 20.0),
    (5.0, 5.0, 25.0),
    )
```

This part will create and store in **elements** container (it is alist note the [] used during creation).

```
for dims in dimensions:
    obj = base_figure(dims[0], dims[1])
    obj.Placement = Placement(Vector(dims[0] * -0.5, dims[1]
                  ↪ * -0.5, dims[2]), ROTO)
    elements.append(obj)
```

One caveat is to determine an axis to follow to make the stacked wires, one of the most useful tricks is to use 0,0,0 as the axis for lofting.

This line is doing the alignment.

```
    obj.Placement = Placement(Vector(dims[0] * -0.5, dims[1]
                  ↪ * -0.5, dims[2]), ROTO)
```

As the shapes are simple, we have adpted this tricks, note the **Z** component of **Placement Vector** that is the third element of **dimensions**, this place wires at approriate height.

This is one of the many ways to do things, it has one advantage, you have all the “settings” data in one unique place, the creation tuple.

Now we have put all the wires, ocrrectly positioned in the list **elements**, ready to be supplied to **Part.makeLoft()**.

This method is expecting some data and parameters:

if we put this simple code in **Python Console**:

```
print(Part.makeLoft.__doc__)
```

we will see:

```
# makeLoft(list of wires,[solid=False,ruled=False,closed=
                      ↪ False,maxDegree=5]) -- Create a loft
                      ↪ shape.
```

Sadly it is not the most clear explanation, but some things could be found in:

https://wiki.freecadweb.org/Part_Loft

That say something similar:

- If "Create solid" is "true" **FreeCAD** creates a solid if the profiles are of closed geometry, if "false" **FreeCAD** creates a face or (if more than one face) a shell for either open or closed profiles.
- If "Ruled surface" is "true" **FreeCAD** creates a face, faces or a solid from ruled surfaces, if "false" it will use transitions made of curves.
- If "Closed" is "true" **FreeCAD** attempts to loft the last profile to the first profile to create a closed figure.

It will suffice, but it lacks of explanation of the last parameter `maxDegree`.

`maxDegree` is used to tune the transitions between wires (sections) of the Loft, in case of curves as explained in https://wiki.freecadweb.org/Part_Loft_Technical_Details.

Some caveats:

- if you don't use Rule interpolation, if strange results are achieved, try to set `maxDegree` to 3 or 2
- if possible use wire with a matching number of segments.
- costantly space elements, and don't place them too near, as interpolations is done using **B-Spline curves**.

```
# makeLoft(list of wires,[solid=False,ruled=False,closed=
                      ↪ False,maxDegree=5])

sol_loft = Part.makeLoft(elements, True, True, False, 5)

Part.show(sol_loft, "Loft_Solid")
```

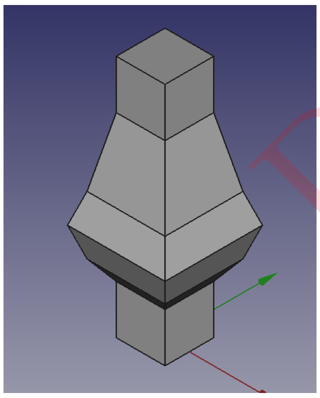
To obtain the solid in figure 7.3a on the following page

We have used the notion of axis, but only to visualize better things.

In fact we could have coded things differently and placed profiles using another paradigm.

```
elem2 = []

dims2 = (
    (8.0, 5.0),
    (8.0, 7.5),
    (8.0, 10.0),
    (8.0, 7.5),
    (8.0, 5.0),
    )

idx = 0

for dims in dims2:
    obj = base_figure(dims[0], dims[1])
    step = 25
    pl1 = Placement(Vector(20, 0, 0), Rotation(0, 0, 90))
    pl2 = Placement(Vector(0, 0, 0), Rotation(step * idx, 0,
                ↪ 0), Vector(0, 0, 0))
    obj.Placement = pl2.multiply(pl1)
    # Part.show(obj)
    elem2.append(obj)
    idx += 1

# makeLoft(list of wires,[solid=False,ruled=False,closed=
                ↪ False,maxDegree=5])

sol_loft2 = Part.makeLoft(elem2, True, False, False, 3)

Part.show(sol_loft2, "Loft_Solid_2")
```

To obtain the solid in figure 7.3b


(a) Loft_Solid.

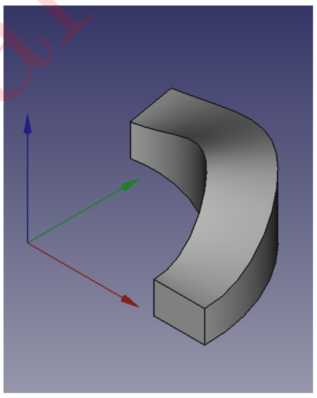

(b) Loft_Solid2.

Figure 7.3.: Loft

## 7.4. Sweep

Let's introduce **Sweep** tool, that is used to create a **Face**, a **Shell**, or a **Solid** from one or more profiles (cross-sections) projected along a path.

See:

https://wiki.freecadweb.org/Part_Sweep

For more infos.

Sadly we have no a method to apply a sweep.

To use it we have to create a **Document object** of type **Part::Sweep**.

The code will be rather complicated, and will use many techniques, so there will be a very articulated explanation.

Complete code could be found in **sweep-full.py** on GitHub "code" page, that will create a tube using sweep, starting from **Geometric primitives** or **TopoShape**.

Let's put following code after:

```
### CODE START HERE ###
```

Numbering will reflect line numbers starting from **base_tmp.py** code and adding subsequent lines from the above line.

```
102 VZOR = Vector(0, 0, 0)
103 EPS = 0.001
104 EPS_C = EPS * -0.5
105
106 c1rad = 120
107 spess = 20
108
109 c2rad = c1rad - spess
110
111 c3rad = 150
112
113 circ1 = Part.makeCircle(c1rad, Vector(0, 0, 0))
114
115 # Part.show(circ1, "circ1")
116
117 circ2 = Part.makeCircle(c2rad, Vector(0, 0, 0))
118
119 # Part.show(circ2, "circ2")
120
121 circ3 = Part.makeCircle(c3rad, Vector(0, 0, 0), Vector(0, 1, 0))
122
123 # Part.show(circ3, "circ3")
```

These lines, are preparatory lines, and show some **parametrization**, if you analyze the whole code you will se many values around the cod that reuse values present in:

```
VZOR = Vector(0, 0, 0)
EPS = 0.001
EPS_C = EPS * -0.5
```

```
c1rad = 120
spess = 20

c2rad = c1rad - spess

c3rad = 150
```

As usual values like `VZOR`, `EPS` and `EPS_C` are used to make some generalization of some lengths, to make some tuning more easy, and mostly to reduce typing.

We have use these value to feed **Part.makeCircle()** that wil create not surprisingly a **circle**.

This method will return a **TopoShape**, but it will be easy to obtain the underlying **Geometric primitive**.

Writing this line in the **Python Console**:

```
  print(Part.makeCircle.__doc__)
```

or using :

```
  print(dir(Part.makeCircle))
```

we will obtain:

```
makeCircle(radius,[pnt,dir,angle1,angle2]) -- Make a circle with a given radius By default pnt=Vector(0,0,0), dir=Vector(0,0,1), angle1=0 and angle2=360
```

So this method will return a circle, or even a portion of circle.

As usual `pnt`, `dir` are the circle center and the axis on which we would create the **Geometric primitive**, and `angle1`, and `angle2` are the starting and ending angle.

We will not use this writing, but another **Geometric primitive** to obtain an Arc.

```
sweep_crv_e = Part.ArcOfCircle(circ3.Curve, pi, pi / 4.0).toShape()

sweep_crv_i = Part.ArcOfCircle(circ3.Curve, pi - EPS, (pi / 4.0) + EPS).toShape()
```

**Part.ArcOfCircle** will return a **Geometric primitive** but has to be fed with a **Geometric primitive**, but we have all the circles creates as **TopoShape**, but if you see in the code we have used `circ3.Curve` that is a **Geometric primitive**.

To see this fact we could simply add after having created a **Part.makeCircle()** object this line:

```
print(circ3.Curve.TypeId)
```

That will print in **Report View**:

```
Part::GeomCircle
```

**Part.ArcOfCircle** need a starting point and a ending point, these are lengths expressed in radians so the use of `pi` in the code.

```
129 sec_e = DOC.addObject("Part::Feature", "ext_section")
130 sec_e.Shape = Part.Face(Part.Wire(circ1))
131
132 spine = DOC.addObject("Part::Feature", "spine")
133 spine.Shape = sweep_crv_e
134 spine.Placement = Placement(Vector(c3rad, 0, 0), ROTO)
135
136 DOC.recompute()
137
138 sweep_e = DOC.addObject("Part::Sweep", "sweep_ext")
139 sweep_e.Sections = sec_e
140 sweep_e.Spine = spine
141 sweep_e.Solid = True
142
143 sec_i = DOC.addObject("Part::Feature", "int_section")
144 sec_i.Shape = Part.Face(Part.Wire(circ2))
145
146 spine_i = DOC.addObject("Part::Feature", "spine")
147 spine_i.Shape = sweep_crv_i
148 spine_i.Placement = Placement(Vector(c3rad, 0, 0), ROTO)
149
150 DOC.recompute()
151
152 sweep_i = DOC.addObject("Part::Sweep", "sweep_int")
153 sweep_i.Sections = sec_i
154 sweep_i.Spine = spine_i
155 sweep_i.Solid = True
156
157 DOC.recompute()
```

This part of code is creating the sweep, let's concentrate on lines from 125 to 142, that create the external sweep of the tube.

We have to deal with **Document objects** because Part::Sweep will accept as inputs only **Document objects**.

So we have to create them. First two are not very complicated, a part from the population of **Shape** property, that needs a **TopoShape**, in line 130 we have already a **TopoShape** returned from **Part.Face** but we have said that **Part.makeCircle()** is a **Geometric primitive**, so we have put in lines 125 and 127 the method `toShape()` at the end of the closing round brackets of **Part.ArcOfCircle**, this methods will usually return a **TopoShape** from a **Geometric primitive**.

Once prepared the needed **Document objects** we fed them to Part::Sweep, in lines

138 to 141.

We sould use all the porperties that are listed in the wiki page, please note that there are some differences, as the page has not been updated for newer versions of **FreeCAD**.

Here you will find a legend to orient in names.

- **Sections** in wiki **Profiles**.
- **Spine** in wiki **Path**.
- **Solid** same as in wiki.
- **Frenet** same as in wiki.
- **Transition** same as in wiki.

In code we have not used all the properties, but it is not difficult to add them if needed, adding a line with a proper value.

Speaking of values, there is a way to find what you need to put in the field.

Using as example **Transition** we could modify it in the **Data tab**, and see that in **Python Console** something like this is printed:

```
FreeCAD.getDocument('sweep_ex').getObject('sweep_ext').
                    ↪ Transition = u"Right corner"
```

This writing is somewhat disorienting at first, but it reflects what is done by the GUI.

It has to retrieve the proper object, for us is simply **Document object** contained in `sweep_e` variable.

So:

- `FreeCAD.getDocument('sweep_ex')` is the reference to the document, for us is simply the content of `DOC`
- `.getObject('sweep_ext')` is the reference to the **Document object** contained in `sweep_e`.
- `.Transition` is the property
- `u"Right corner"` is the value that is assigned, when changing the value in with the GUI.

Not difficult once accustomed, as you could ever relaunch the script, it is no harmful to do some experiment and see what is printes in **Python Console**.

Lines from 143 to 157 will simply make the internal sweep of the tube, are very similar a part for the values.

Let's continue with code explication.

Following code will create a **boolean cut** of the external and internal shape, to make the tube.

Note the use in line 159 of **Shape** property to retrieve the **TopoShape** that is needed to directly use `.cut()`.

```
159 tube_shape = sweep_e.Shape.cut(sweep_i.Shape)
160
161 tube = DOC.addObject("Part::Feature", "tube")
162 tube.Shape = tube_shape.copy()
163 tube.Placement = Placement(Vector(0, c1rad * 2.5, 0), ROT0)
```

lines from 161 to 163 are simply create a new **Document object** with the tube and position it in an appropriate place, as "by design" we have not hidden the generating sweep solids.

**Placement** property is fed with a **Translation Vector** that use some mathematics, note two things:

- use of the radius of the external circle to make the positioning parametric.
- **Shape** property is fed with a copy of **tube_shape** **TopoShape** obtained using **.copy()**, we have to reuse the shape so it is better to use copies instead of original objects.

We reuse **tube_shape** solid to make a section of the tube, it is an exercise, but it may be useful to manage this technique.

There are for sure other ways to do the section, but let's explain the rationale behind the operations:

1. create a new object, in this case a **TopoShape** will suffice.
2. create a "cutting shape", in this case, a parallelepiped.
3. perform a **boolean cut** of the two shapes.
4. create a **Document object** to show the result.

Let's start with point 1.

```
165 sec_y_disp = c1rad * 5.0
166
167 sec_tube = tube_shape.copy()
168 sec_tube.Placement = Placement(Vector(0, sec_y_disp, 0),
                         ↪ ROT0)
```

Note the ue of a variable to specify Y part of the **Translation Vector**, we have to reuse the position to properly place our "cutting shape".

Point 2.

```
165 sec_dim = c3rad * 2.1 + c1rad * 2.1
166
167 cut_pos = sec_y_disp + (0.00 * c1rad)
168
169 section = Part.makeBox(sec_dim, c1rad * 2.1, sec_dim)
170 section.Placement = Placement(Vector(c1rad * -1.05, cut_pos,
                         ↪  sec_dim * -0.5), ROT0)
```

```

# Part.show(section, "cutting shape")
```

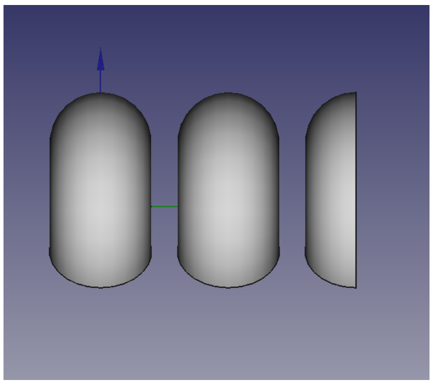
Things to be noted:

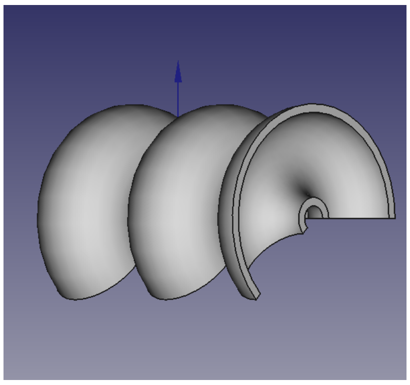
- variable `sec_dim` with some math to calulate "cutting shape" dimensions, as it is derived from a circular shape, it is easy to calculate the dimension using radius of generating objects, for X and Z dimension are the same Y is simply sligthly more than the external radius.
- variable `cut_pos` that is composed by two elements, `sec_y_disp` that is **Translation Vector** of the shape to be cut, and a factor that depends by `c1rad`, if you use a positive value cut is moved beyond the middle of the tube, a negative value will move the cut in opposite direction. This permit to tune the cut shown, another example of parametrization.
- It is possible to visualize "cutting shape" simply decommenting the `Part.show` statement, to check if something is wrong.

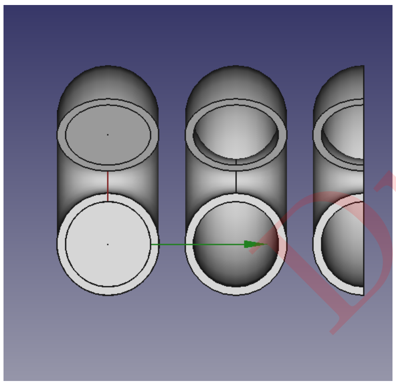
Last part of the code, is not too difficult as it don't contain no new things.

```
sect_tube = DOC.addObject("Part::Feature", "tube_section")
sect_tube.Shape = sec_tube.cut(section).removeSplitter()

DOC.recompute()

setview()
```

Here some images of the produced solids:


(a) Sweep solids side view.


(b) Sweep solids right-rear view.


(c) Sweep solids bottom view

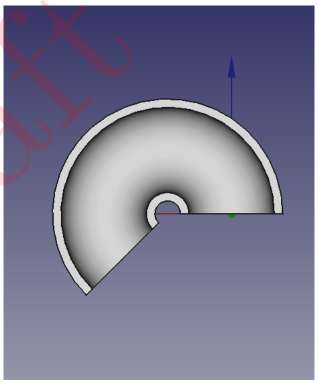

(d) Sweep solids rear view

Figure 7.4.: Sweep example

# Appendix A

## User Interface Elements

**3D view** La finestra 3D - p. 5, 8, 14, 18

**Combo view** - p. 18, 25

**Data tab** PE tab - p. 5, 13, 25, 46

**Macro editor** - p. 5, 6

**Property editor** - p. 5, 13, 25
**Python Console** - p. 40, 46
**Python editor** - p. 6

**Report View** - p. 17, 24, 44

**Toolbar area** - p. 5
**Tree view** - p. 7, 18, 26

**View tab** PE tab - p. 5, 13, 14, 25

# Appendix B

# Glossary

**Boolean operation** - p. 17, 29
  **Fusion** - p. 29
  **Intersection** - p. 29
  **Subtraction (cut)** - p. 29

**Tait-Bryan angles** - p. 26

# Appendix C

## Menu Item

**Edit** Menu item - p. 4, 14, 26
  **Placement** - p. 26
  **Preferences** - p. 4, 14
    **General** Preferences menu item - p. 4
      **Editor** Tab - p. 4
    **Part design** Preferences menu item - p. 14
    **Shape view** Tab - p. 14

**File** Menu item - p. 6
  **Open** - p. 6

**View** menu item - p. 19
  **Panels** - p.
    **Python Console** - p. 5
    **Report view** - p. 5
  **Toggle Axis cross** - p. 19

# Appendix D

## FreeCAD Objects

**Angle** Property - p. 20

**Base** Property - p. 29, 31
**Bezier curve** "Part::GeomBezierCurve" Geometry - p. 16
**Bezier surface** "Part::GeomBezierSurface" Geometry - p. 16
**B-Spline curve** "Part::GeomBSplineCurve" Geometry - p. 16, 41
**B-Spline surface** Geometria di tipo "Part::GeomBSplineSurface" - p. 16

**Center** Property - p. 17
**Circle** "Part::GeomCircle" Geometry - p. 16, 44
**Curve** Geometry Entity - p. 15, 16

**Document object** 3D view Element - p. 7, 8, 13, 15, 17, 30, 43, 45–47

**Edge** Topology Element - p. 15, 16
**Ellipse** "Part::GeomEllipse" Geometry - p. 16
**Extrude** Function - p. 36

**Face** Topology Element - p. 15–17, 42
**Frenet** Object Property - p. 46

**Height** Property - p. 20
**Hyperbola** "Part::GeomHyperbola" Geometry - p. 16

**Line** "Part::GeomLine" or "Part::GeomLineSegment" Geometry - p. 16

**Parabola** "Part::Parabola" Geometry - p. 16
**Part::Box** Geometry - p. 18, 22
**Part::Cut** Boolean operation - p. 29, 31
**Part::Cylinder** Geometry - p. 19, 22
**Part::Feature** Geometry - p. 17
**Part::Fuse** Boolean operation - p. 29, 31
**Part::MultiCommon** Boolean operation - p. 29, 31
**Part::MultiFuse** Boolean operation - p. 29, 30, 32
**Part::Sphere** Geometry - p. 22
**Part::Sweep** Tool - p. 43, 45

**Part::TopoShape** Geometry - p. 17
**Part.makeCircle()** Part Module Function - p. 44, 45
**Part.makeLoft()** Part Module Function - p. 39, 40
**Part.makePolygon()** Part Module Function - p. 35, 36
**Placement** Object property - p. 24–26, 47
  **Angle** - p. 25
  **Axis** - p. 25
  **Position** - p. 25
**Plane** "Part::GeomPlane" Geometry - p. 16
**Point** Geometry Entity - p. 14, 16
**Primitive** base solid - p. 17–19, 22, 24, 29

**Radius** Property - p. 17, 20
**Refine** Property - p. 30, 31
**Revolve** Function - p. 38

**Sections** Property - p. 46
**Shape** Property - p. 17, 45–47
**Shapes** Property - p. 30
**Shell** Topology Entity - p. 15, 17, 42
**Solid** Property - p. 46
**Solid** Generic name for a solid - p. 12, 15–19, 42
**Spine** Property - p. 46
**Surface** Geometry Entity - p. 15, 16

**Tool** Property - p. 29, 31
**TopoShape** - p. 15–17, 43–47
**Transition** Property - p. 46

**Vector** Geometry Entity - p. 12, 25, 26
**Vertex** Topology Entity - p. 15, 16, 22
**ViewObject** Property - p. 13

**Wire** Topology Entity - p. 15, 16, 35