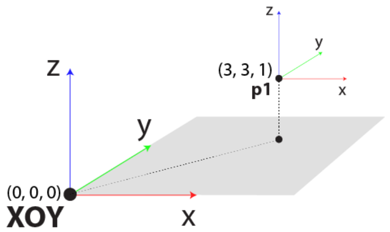
1 Positioning Para*Py* objects

The main classes for positioning are `Point(x, y, z)`, `Vector(x, y, z)`, `Orientation(vx, vy, vz)` and `Position(point, orientation)`. These classes provide their own set of attributes and methods. The most important functions for translation and rotation purposes, are `translate(point|position, dir, dist)` and `rotate(point|position, axis, angle)`. Refer to the separately provided "Positioning Cheatsheet" to get an immediate overview of the respective APIs. All class and function names are importable from the `parapy.geom` package.

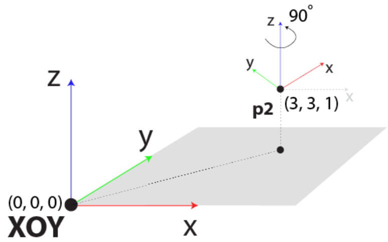
In ParaPy, primitive geometry can be located and oriented using an axis system. We refer to it as the object's "position" in space. To specify the position of an object, you need to create a `Position` instance. A `Position` instance is a local axis system, defined by a location and orientation in Cartesian space. The global axis system is available in ParaPy as the constant `XOY`. Its location is fixed at `Point(0, 0, 0)`, available as constant `ORIGIN`. Its orientation equals the identity matrix `Orientation(x=Vector(1, 0, 0), y=Vector(0, 1, 0), z=Vector(0, 0, 1))`, available as constant `XY`.

It is common practice in ParaPy, to create new `Position` instances relative to this global axis system by one or multiple translations and/or rotations. While rotation is a relatively straightforward principle, the key to translation is to appreciate that translation it is always <u>relative</u> to the orientation of the reference `Position` instance. The schematic below provides a high-level overview of this idea. This tutorial will guide you through it in a step-by-step fashion.

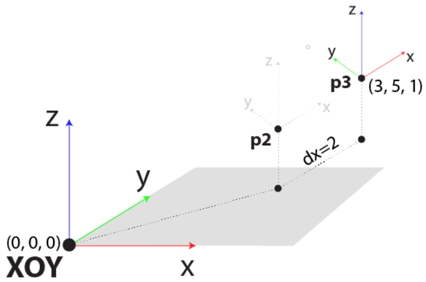
```
p1 = translate(XOY, x=3, y=3, z=1)
```




```
p2 = rotate90(p1, 'z')
```




```
p3 = translate(p2, x=2)
```



Classes that derive from `GeomBase`:

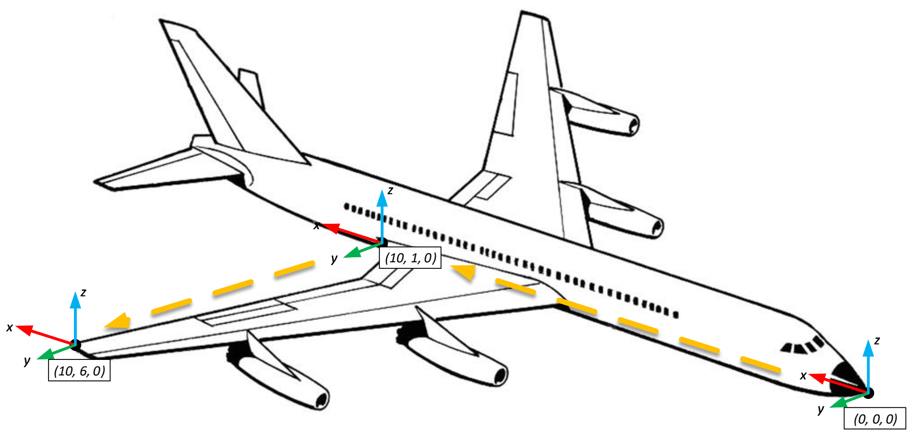
```
class Aircraft(GeomBase):
```

inherit a `position`, `location`, `orientation`.[1] `GeomBase` was conceived to ease relative positioning of (sub-) assemblies. When creating child objects (`@Part`) that also derive from `GeomBase`, their `position` will be coupled to their parent's `position`. This parent object can in turn be composed inside its own parent that also inherits from `GeomBase` and the same type of value binding occurs. As such, a tree of objects is created with coupled positions. If the root of the tree is re-positioned, the entire underlying object tree will translate and/or rotate accordingly.

[1] and a `bbox` (bounding box), but this isn't part of this tutorial.

Take a wing for example, it has a local axis system located at the wing root trailing edge point. Its x-axis corresponds to its chordwise direction and the y-axis to its spanwise direction. The root airfoil object could be positioned at this axis system, while a tip airfoil could be positioned relative to this axis system in y-direction over the wing span. If this wing would then later be composed as a right wing inside an aircraft, the wing could be placed at an offset from the aircraft's coordinate system (the nose by convention) in both x- and y-directions. A high-level implementation of this in Para*Py* could be like this:

```
from parapy.core import *
from parapy.geom import *


class Aircraft(GeomBase):
    x_wing = Input(10)
    y_wing = Input(1)

    @Part
    def right_wing(self):
        return Wing(position=translate(self.position, 'x', self.x_wing, 'y', self.y_wing))


class Wing(GeomBase):
    span = Input(5)

    @Part
    def root_airfoil(self):
        return Airfoil()

    @Part
    def tip_airfoil(self):
        return Airfoil(position=translate(self.position, 'y', self.span))


class Airfoil(GeomBase):
    pass

>>> obj = Aircraft()
>>> obj.position.location
Point(0, 0, 0)
>>> obj.right_wing.position.location
Point(10, 1, 0)
>>> obj.right_wing.root_airfoil.position.location
Point(10, 1, 0)
>>> obj.right_wing.tip_airfoil.position.location
Point(10, 6, 0)
```



As witnessed, lower-level objects are relatively translated with respect to their parent object. The `root_airfoil` is somewhat special in that it wasn't explicitly positioned at all, but still its position corresponds to that of the wing. This is because of *defaulting* behavior in Para*Py*. The standard behavior of the inherited `GeomBase.position` Input was defined to be defaulting:

```
    position = Input(XOY, defaulting=True)
```

Unless given a different value, a *defaulting* slot will trace up the object tree for a default value (first its direct parent, then the parent of that parent, etc.). If it finds any Slot with the <u>same name</u>, it will bind to that value. If it couldn't find any similarly named Slot, it will take its own default value or raise a `MissingRequiredInput` exception if there was no default value. In case of `position`, you see that `root_airfoil` will find a "position" Slot in its parent wing object, while the aircraft has no parent and will default to the global axis system `XOY`.

As a general guideline, try to use relative positioning by inheriting `GeomBase`. Avoid hard-coding of object positions because it will seriously limit the re-usability of objects from an assembly perspective.

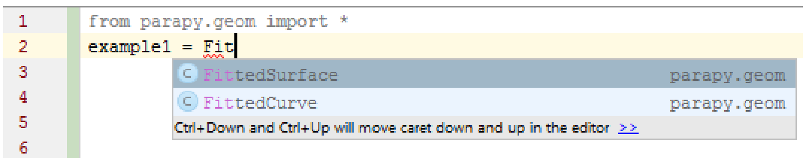
Exercise 4: Boxes – positioning single objects

This tutorial will guide you through the positioning of single geometry objects. You will also learn how to read through some of the source codes of Para*Py*.

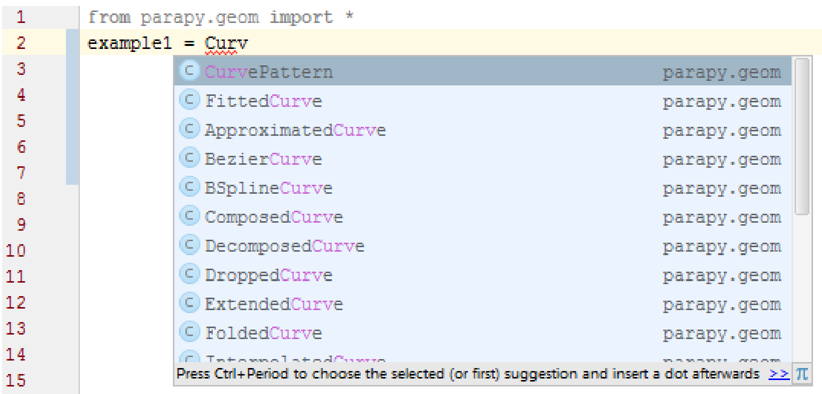
1. Create a new module and import the Para*Py* geometry library with the following statement (at the top of your module):

```
from parapy.geom import *
```

The advantage of a *wild* import as opposed to other forms of import is that it imports *everything* from the designated package or module at once. This allows using the various geometry variables, classes, and functions without prefixing them with the module's name.[2] During development in an IDE like PyCharm (and some other IDEs as well), the IDE will search through this library and provide you auto-completion suggestions based on what you are typing. For instance, if you are looking for a fit through a list of Points, type `Fit`. PyCharm will show all the Para*Py* classes that have `Fit` in their name. Simply hitting enter will complete the class name in your editor. In general, try to exploit such features, they make you a productive programmer and prevent errors.



Similarly, if you are interested in all the classes with “Curve” in their name, type `Curve`. PyCharm will show all Curve classes.



2. Create a new class, say `MyClass`, and inherit from `GeomBase`:

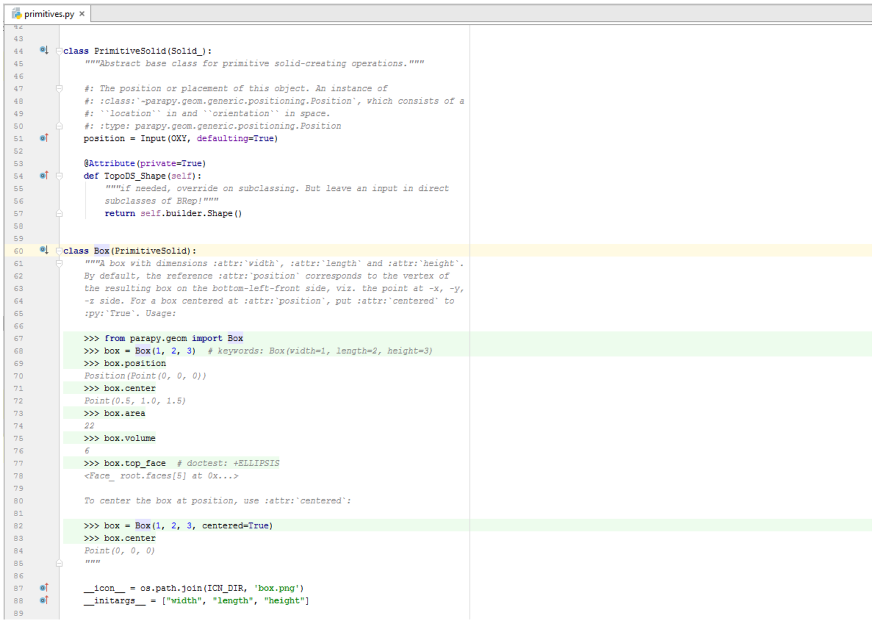
```
class MyClass(GeomBase):
```

3. Make a `Box` Part, called `box1`.

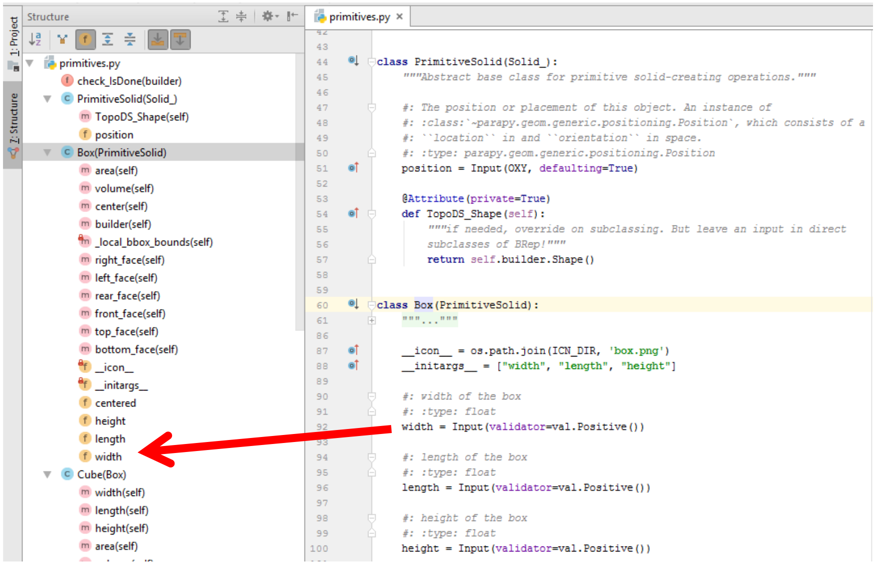
```
    @Part
    def box1(self):
        return Box()
```

[2] Importing everything from the `parapy.geom` package is considered safe practice. We took care of limiting what gets importing to a set of roughly 200 frequently used names. In general, however, importing everything from a module or package is discouraged. It may lead to namespace collisions (importing the same name from different modules), can be inefficient and is rather implicit programming.

4. The `Box` object in Para*Py* has three required inputs. You can either consult the API in the separately provided HTML documentation or you can quickly check what these inputs are by navigating to the declaration of Box inside the ParaPy source code. There are two ways to navigate to class declarations. If you prefer the mouse, hover over the word `Box` and, while holding *Ctrl*, left-click the word. Alternatively, you can use the keyboard and type *Ctrl + B* for this purpose. Verify that your editor jumps to the class definition inside `primitives.py`, as shown below:



The triple-quoted string immediately following the class statement is better known as a *docstring* and will typically contain a short description of the class and provide a simple usage example. It may also prove convenient to quickly scan the entire API of a class using the *Structure* window in PyCharm (`Alt+7`). Inputs are shown at the bottom as *fields*, while Attributes and Parts are (wrongfully) shows as *methods*. You can toggle the "Show inherited" button to visualize slots as inherited from ancestor classes.



On various occasions, you may notice an `__initargs__` assignment. This statement defines an additional, non-keyword-based constructor signature using positional arguments. For a `Box` it is perfectly valid to instantiate an object either following the de-facto keyword-based notation

```
>>> Box(width=1, length=2, height=3)
```

or to use a shorter notation with positional arguments

```
>>> Box(1, 2, 3)
```

When using keywords, order doesn't matter. The following line will give you the same result.

```
>>> Box(length=2, height=3, width=1)
```

Finally, remaining Input should always follow positional arguments.

```
>>> Box(1, 2, 3, color="red")
```

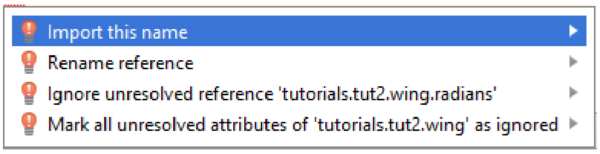
5. Pass the following arguments to `box1`: width=1, length=1 and height=1. Moreover, specify that is has the color red.
6. Now let's create another `Box` Part `box2`, similar in dimensions, but with a custom `position` input. Translate the position in x- and y-direction by 3:

```
@Part
def box2(self):
    return Box(width=1,
               length=1,
               height=1,
               position=translate(self.position, 'x', 3, 'y', 3))
```

7. Rotate the position by 60° around the x-axis:

```
@Part
def box2(self):
    return Box(width=1,
               length=1,
               height=1,
               position=rotate(translate(self.position, 'x', 3, 'y', 3),
                               Vector(1, 0, 0), radians(60)))
```

Note that the word `radians` in your editor is marked with a red line. You will need to import `radians` from the built-in `math` module. You can either type this import statement yourself at the top of your module, or, in case you are using PyCharm, locate your cursor on the word `radians` and press ALT+ENTER. The pop-up menu makes suggestions, one of which is to import this name:



Press enter again and select `math.radians(x)`. Pycharm will now add the following statement at the top of your module.

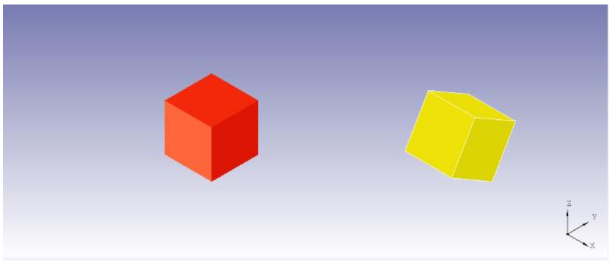
```
from math import radians
```

Now, carefully look at the positioning syntax of the example above:

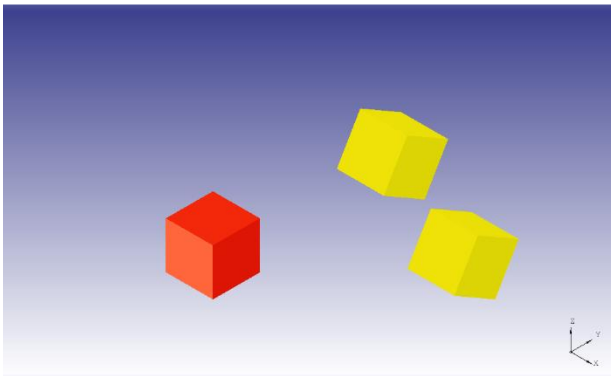
```
               position=rotate(translate(self.position, 'x', 3, 'y', 3),
                               Vector(1, 0, 0), radians(60)))
```

You can see that the result of `translate(self.position, 'x', 3, 'y', 3)` is used as the first argument to `rotate`. Why does this work? Functions like translate take a `Position` instance as first argument and return a new `Position` instances as result. As such, you are passing the outcome of `translate`, a `Position`, as the first argument to `rotate`. In turn, `rotate` will return another `Position` instance that will be passed to the `Box`.

8. Instantiate `MyClass`, and display it in the ParaPy GUI. Visualize `box1` and `box2`:

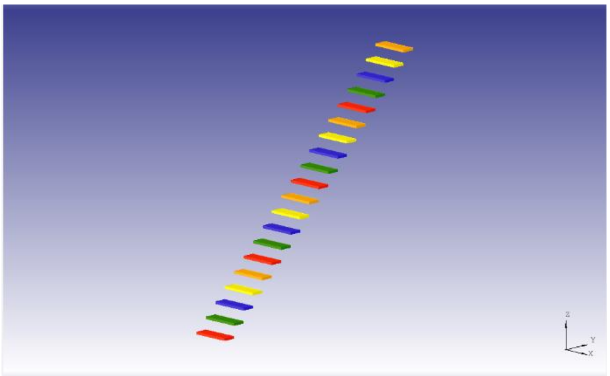


9. Make a third Box `box3`, like `box2`, but reverse the order of operations:
   a. first, rotate by 60° around the x-axis;
   b. then, translate in x- and y-direction by 3.
10. Note the difference between `box2` and `box3`, due to the different order of translation and rotation steps:



Exercise 5: Staircase 1 – positioning quantified objects

This tutorial will guide you through the positioning and orienting of quantified geometry objects, applied to a simple staircase:



Use these inputs for the number of steps, width, length and thickness of individual steps, and the height between these:

Staircase	Type	Value
n_step	int	20
w_step	float	3
l_step	float	1
h_step	float	1
t_step	float	0.2

1. Create a class and inherit `GeomBase` again.
2. Define inputs per the table above. You may add a Python comment about the assumed units.

3. As the number of steps is variable, you can't predefine 20 individual box objects. Instead, we will use a quantified sequence of objects. This is done by specifying the `quantify` keyword as an input to your `Box` class. Refer to exercise 2 of the previous tutorial on how to use `quantify`. Create a `Part` that returns a sequence of `n_step` `Box` objects with dimensions equal to `w_step`, `l_step` and `t_step`.
4. Position each `Box` object in your sequence. Translate each step position in y- and z-direction by the step width and height, respectively. You need the `child.index` syntax as explained in exercise 2.
5. Color your steps. To do so, first add an Input slot to your class with a list of colors:

```
colors = Input(["red", "green", "blue", "yellow", "orange"])
```

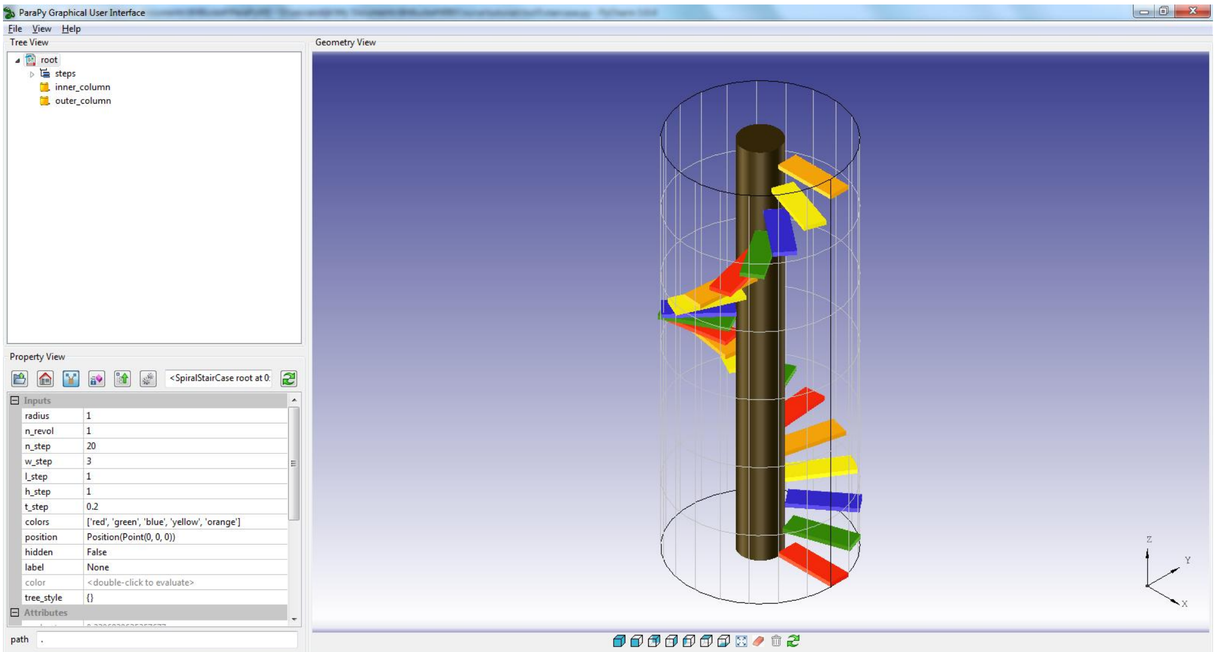
Then, pass the following argument to the return class of your steps Part:

```
color=self.colors[child.index % len(self.colors)]
```

The `%` operator in Python, called modulo, will compute the remainder of dividing two numbers, x and y. It returns `x - int(x/y) * y`. As an example, consider x = 5, y = 2. Then `x%y = 5 - int(5/2) * 2 = 5 - 2 * 2 = 5 - 4 = 1`.

Exercise 6: Staircase 2 – positioning quantified objects

Make a spiral version of your stair case that has exactly one revolution. Take a radius of 1.



2 Curve geometry

Exercise 7: Para*Py* curve classes, methods and attributes

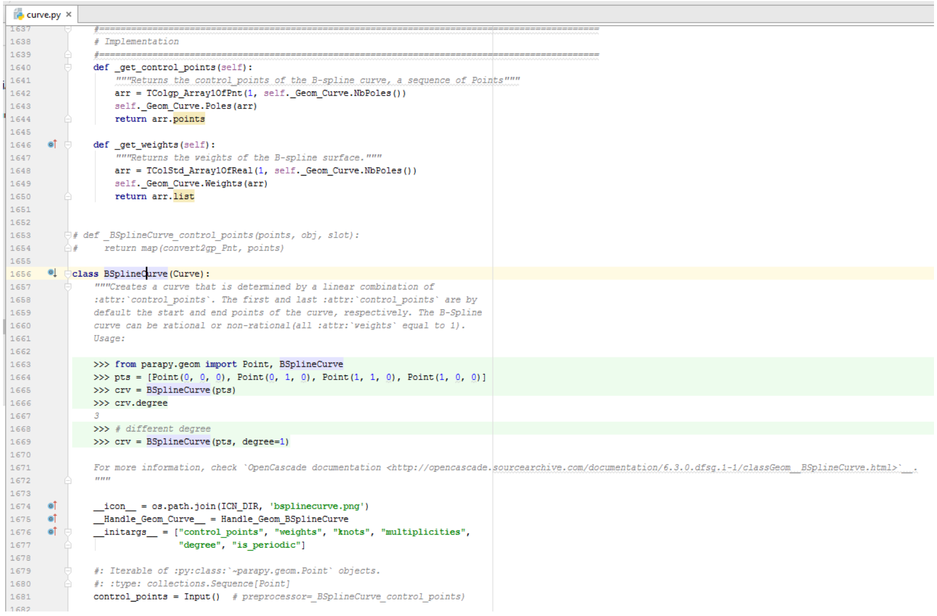
All geometry is described by a mathematical model. Curves are parametrized in the form $f(u) \;{-}{>}\; Point(x, y, z)$, while surface are parametrized as $f(u, v) \;{-}{>}\; Point(x, y, z)$. B-Spline curves and surfaces in Para*Py* are parameterized using the so-called Non-Uniform Rational Basis Spline, or NURBS representation. This model is often used in computer aided modelling due to its precise and well-known definition, flexibility in geometrical modelling and is the industry-standard method for exchange between different programs. Search the internet for a more in-depth overview of the underlying geometry. In short, Non-uniform rational basis spline means:

- Basis Spline: curves and surfaces are controlled by a list or grid of 3D control points.
- Rational: weights are used to affect the geometry. In case of a curve, each point on the curve is determined by taking a weighted sum of the control points.

- Non-Uniform: curves and surfaces have knot vectors that determine where and how the control points influence the NURBS geometry.

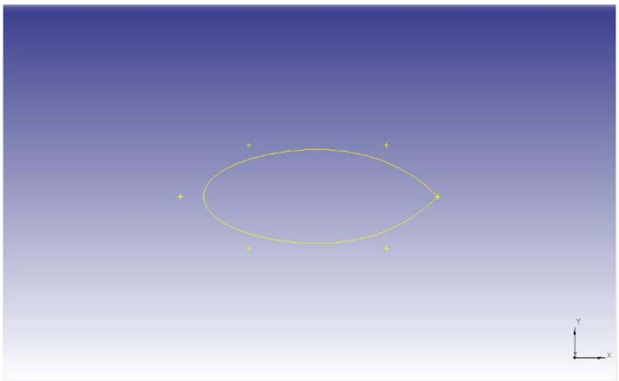
In this tutorial we will use the `BSplineCurve` class from the Para*Py* geom library to construct several NURBS curves

1. Create a class definition called `BSplineSamples`.
2. In this class, create a list of points. The Points should have the following x-, y-, and z-coordinates: (0,0,0), (-3,3,0), (-11,3,0), (-15,0,0), (-11,-3,0), (-3,-3,0), (0,0,0).
3. Check the required inputs for the `BSplineCurve` in the HTML docs or jump to its declaration.

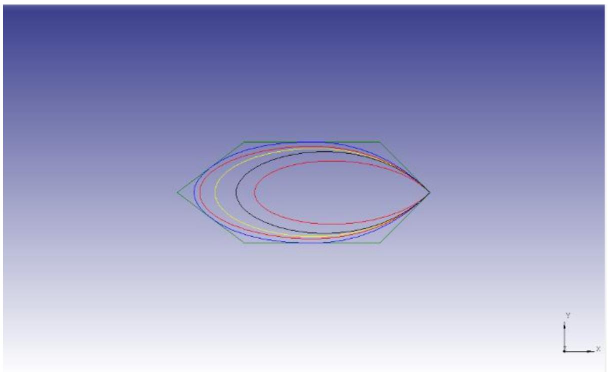


```
curve.py

    #==========================================================================================
    # Implementation
    #==========================================================================================
    def _get_control_points(self):
        """Returns the control_points of the B-spline curve, a sequence of Points"""
        arr = TColgp_Array1OfPnt(1, self._Geom_Curve.NbPoles())
        self._Geom_Curve.Poles(arr)
        return arr.points

    def _get_weights(self):
        """Returns the weights of the B-spline surface."""
        arr = TColStd_Array1OfReal(1, self._Geom_Curve.NbPoles())
        self._Geom_Curve.Weights(arr)
        return arr.list


# def _BSplineCurve_control_points(points, obj, slot):
#     return map(convert2gp_Pnt, points)

class BSplineCurve(Curve):
    """Creates a curve that is determined by a linear combination of
    :attr:`control_points`. The first and last :attr:`control_points` are by
    default the start and end points of the curve, respectively. The B-Spline
    curve can be rational or non-rational(all :attr:`weights` equal to 1).
    Usage:

    >>> from parapy.geom import Point, BSplineCurve
    >>> pts = [Point(0, 0, 0), Point(0, 1, 0), Point(1, 1, 0), Point(1, 0, 0)]
    >>> crv = BSplineCurve(pts)
    >>> crv.degree
    3
    >>> # different degree
    >>> crv = BSplineCurve(pts, degree=1)

    For more information, check `OpenCascade documentation <http://opencascade.sourcearchive.com/documentation/6.3.0.dfsg.1-1/classGeom__BSplineCurve.html>`__.
    """

    __icon__ = os.path.join(ICN_DIR, 'bsplinecurve.png')
    __Handle_Geom_Curve__ = Handle_Geom_BSplineCurve
    __initargs__ = ["control_points", "weights", "knots", "multiplicities",
                    "degree", "is_periodic"]

    #: Iterable of :py:class:`~parapy.geom.Point` objects.
    #: :type: collections.Sequence[Point]
    control_points = Input()  # preprocessor=_BSplineCurve_control_points)
```

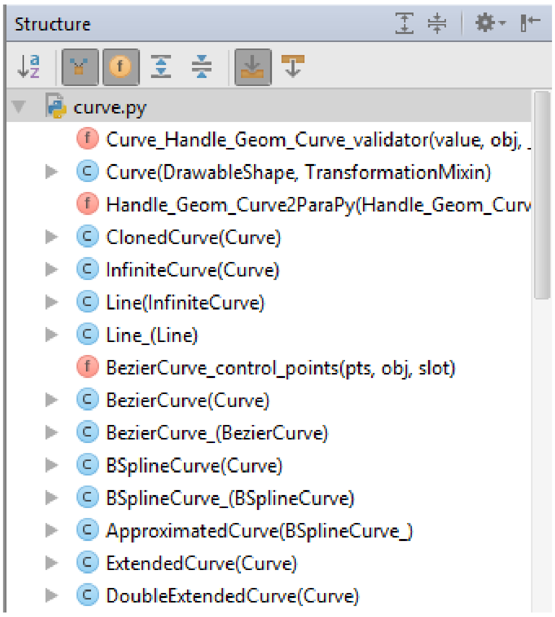
4. Create a Part `curve` that returns an instance of type `BSplineCurve` and pass the list of points as its `control_points`. Beware of the common misconception that a B-Spline curve fits through the `control_points`. Generally, this is true for only the first and last control point, where the others act like "magnets" to the curve. If you actually need a curve that fits through all points, use the `FittedCurve` instead.
5. Visualize both the `control_points` and the `BSplineCurve` in the GUI. You can visualize the `control_points` by right-clicking the `control_points` slot in the GUI property view and selecting "Display".



6. A B-Spline curve has adjustable `weights` for closer approximations to arbitrary shapes. The default weight of a control point in Para*Py* is 1.0 (this type of B-Spline curve is called non-rational). Modify the second weight value to 2.0 in the GUI. Observe the difference. Play with the other `weights` and observe the differences again
7. Make another Part in the `BSplineSamples` class called `curves` that returns a sequence of 6 curves where the degree is varying from 1 to 6. Visualize the curves in the GUI.



8. Curve classes in Para*Py*, such as `BSplineCurve` and `FittedCurve`, have several Attributes and methods that will make your life a lot easier when working with curve objects. These Attributes and methods can be found in the base class of all the Curve objects, called `Curve`. Navigate to its declaration. Once in curve.py, use the *Structure* window in PyCharm (`Alt+7`). Clicking on Curve will bring you directly to the Curve class' source code.



Typing *Ctrl + f* and searching for "class Curve" will also bring you to the class.

```
from parapy.geom.occ.patched.tcol import *  # @UnusedWildImport
from parapy.geom.occ.patched.gp import *  # @UnusedWildImport
from parapy.geom.occ.utilities import resolve_gce_status


def Curve_Handle_Geom_Curve_validator(value, obj, _):
    """Curve Handle_Geom_Curve should be of type __Handle_Geom_Curve__"""
    return isinstance(value, obj.__Handle_Geom_Curve__)


class Curve(DrawableShape, TransformationMixin):
    """Curve is the abstract base class of all Curve object and wraps around a OpenCascade
    Geom_Curve
    """

    __icon__ = os.path.join(ICN_DIR, 'curve.png')
    __Handle_Geom_Curve__ = Handle_Geom_Curve

    # 7:Vertex,6:Edge,5:Wire,4:Face,3:Shell,2:Solid,1:CompSolid,0:Compound
    TOPOLEVEL = 6
    TOPODIM = 1
    EdgeClass = None

```

Examples of attributes that you may find in this class are: `point1`, `midpoint`, `tangent2` and `normal1`. Examples of methods that you may find are: `projected_point`, `tangent` and `normal_at_point`. Try and find these Attributes and Methods.

```python
    def tangent(self, u):
        """The unit tangent vector at parameter u.

        :param float u: parameter on curve.
        :rtype: Vector
        """
        return self.derivate(u)

    def tangent_at_point(self, point):
        """Unit tangent vector at point.

        :param Point point: point on curve.
        :rtype: Vector
        """
        return self.tangent(self.parameter_at_point(point))


    def normal(self, u, binormal=None, normalized=True):
        """The (unit) normal vector of this curve at parameter ``u``. It is
```

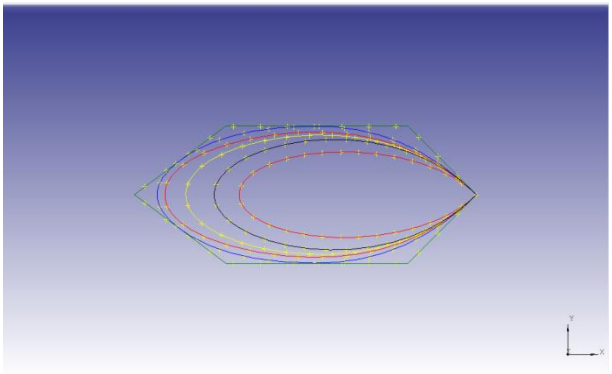
9. Make an Attribute that returns a list of 30 equispaced points for each B-Spline curve in the sequence of varying-degree curves you made above. Since your Part returns a sequence of `BSplineCurve` objects, a for-loop or list comprehension should be used to access each individual curve. Use the method `equispaced_points`:

```python
crv.equispaced_points(30)
```

10. You can visualize your Attribute with the equispaced points in the tree by adding `(in_tree=True)` to the Attribute decorator.

```python
@Attribute(in_tree=True)
```

11. Check the equispaced point distributions in the GUI and verify that it display:



12. Determine the length of all your `BSplineCurve` objects in the GUI. Search for the `length` Attribute in the Attributes table of the GUI and evaluate the slot. Does the length increase or decrease with an increasing degree?
13. Make an Attribute that returns a `Point` at length 2 of each `BSplineCurve` object. Use the Curve method `point_at_length`.

```python
crv.point_at_length(2)
```

14. Translate all `BSplineCurve` objects by 5 in z-direction. Your first thought might be to pass a translated position as argument to the `BSplineCurve` class. However, this will not work. ParaPy does not allow this, since there is no clear position or "axis system" for a B-Spline curve. Control points are taken relative to the global axis system. One way is to use the built-in method translated. Make a sequence of `TranslatedCurve` or `TransformedCurve` classes. These classes will also be used in later tutorials.

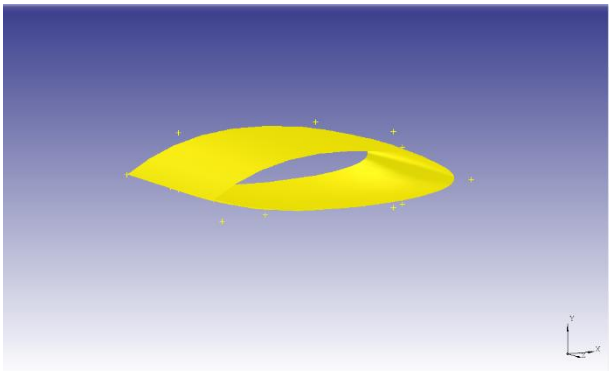
Exercise 8: Para*Py* surface classes, methods and attributes

B-Spline curves are 1-dimensional parametric curves (its parameter is termed u) and require a flat list of control points. NURBS surfaces are 2-dimensional (its parameters are termed u and v) and require a list of lists, a 2-dimensional grid of control points. In this section, you will make a `BSplinceSurface` object. Just as for a `BSplinceCurve` in the previous example, it is in general not true that the `BSplineSurface` is fitted through all the control-points (see `FittedSurface` instead).

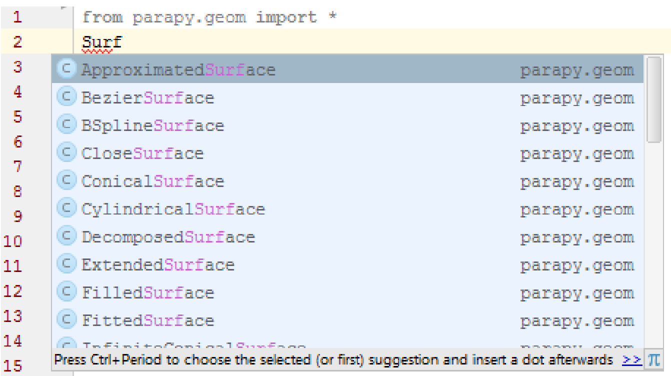
1. Copy the following data:

   data = [
   [(0, 0, 0), (3, 2, 0), (11, 2, 0), (15, 0, 0), (11, -1, 0), (3, -1, 0), (0, 0, 0)],
   [(0, 0, 5), (3, 2, 5), (13, 2, 5), (15, 0, 5), (13, -2, 5), (3, -2, 5), (0, 0, 5)],
   [(0, 0, 10), (3, 2, 10), (11, 2, 10), (15, 0, 10), (11, -1, 10), (3, -1, 10), (0, 0, 10)]]

   Note that data consists of a list of three lists of tuples
2. Transform this data into a list of three lists of `Point` objects. Note that you will need to use two for-loops since `data` is a list of lists.
   a. first, access each sub-list by looping through the outer list;
   b. then, access the tuples in each list and make `Point` objects from each tuple.
   c. `Append` the `Point` objects in a new list.
   d. `Append` this list to a new outer list.
   e. If you feel like a pro, use double list comprehension instead, this is shorter and faster.
3. Make a surface with the `BSplineSurface` class. Specify its `control_points`.
4. Visualize the `control_points` and the `BSplineSurface` in the GUI.



5. Just like curve classes, surface classes in Para*Py* have several Attributes and methods that will make your life a lot easier when working with surface objects. First, find the `Surface` base class. Then, find `area`, `point`, `u_tangent` and `cog`.
6. Determine in the GUI the `area` of the surface, look under category Attributes in the GUI property view.
7. Visualize the center of gravity (`cog`) of the `BSplineSurface` in the GUI.
8. Make a new `Attribute` that returns the area of the surface.
9. The Para*Py* geometry library has many surface classes. Look at this collection by typing "Surface" in the editor. PyCharm will show all the current Surface classes currently available in Para*Py*.



4 Boolean operations

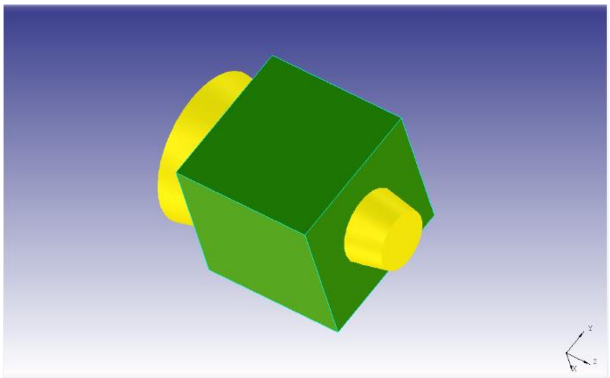
Exercise 9: Boolean operations

Working in 3D usually involves the use of solid objects. At times, you may need to combine multiple parts into one, or remove sections from a solid. Para*Py* has several Boolean operation classes that make this easy for you. In this example you will fuse, subtract, intersect and partition a box and cylinder with the following dimensions.

Box	Type	Value
height	float	1.0
width	float	1.0
length	float	1.0

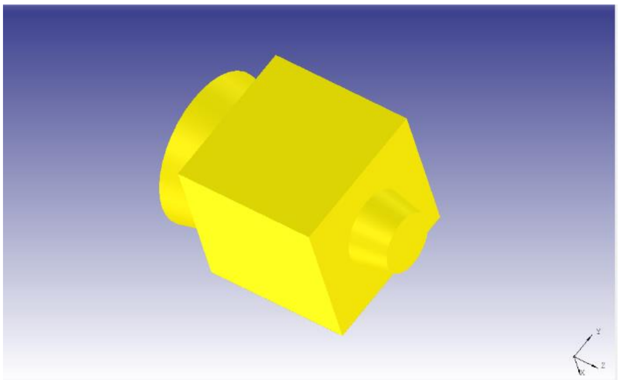
Cone	Type	Value
radius1	float	0.5
radius2	float	0.2
height	float	1.5

1. Define a class and make a `Box` and `Cone` Part.
2. Position the Parts such that the `Cone` crosses the entire `Box`.

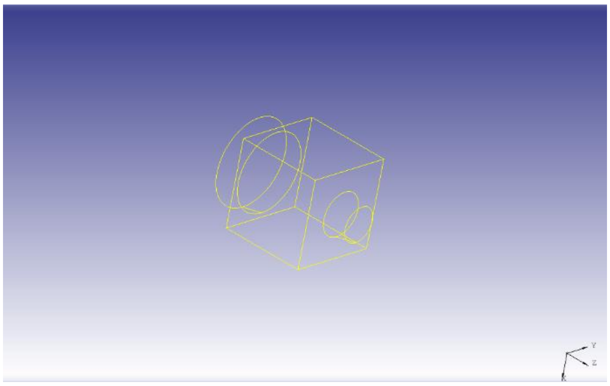


3. Make a single solid from the `Box` and `Cone` objects by using the `FusedSolid` class with the `Cone` as tool and visualize it in the GUI.

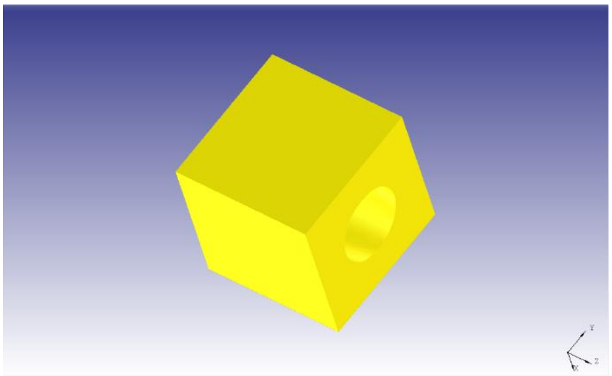
```
@Part
def fused(self):
    return FusedSolid(shape_in=self.box,
                      tool=self.cone)
```



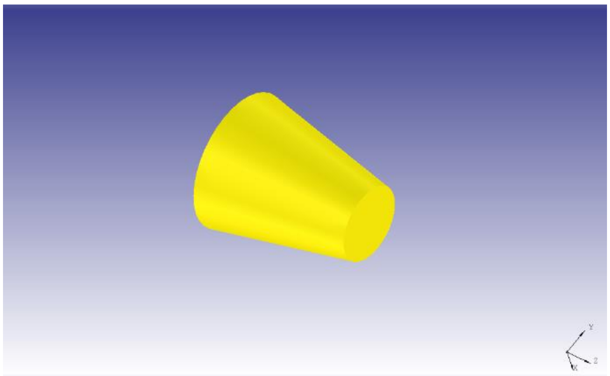
4. Switch to the wireframe viewing mode by pressing “w” in the GUI viewport. Note the difference with the wireframe from step 2: edges are present at place where the `Cone` intersects the `Box`.



5. Subtract the `Cone` from the `Box` with the `SubtractedSolid` class.



6. Determine the intersection between the `Box` and the `Cone`. Use the `CommonSolid` class.



7. The partition operation allows you to create different volumes in a shape. This may be convenient for assigning different materials to your shape or for multi-domain simulations with different, touching meshes. Create a partition by using the `PartitionedSolid` class. Note in the GUI both the PartionedSolid can be visualized (Display Node), but the separate volumes are also accessible as `solids`. Modify the `keep_tool` Input from False to True.