Annex B: Creating Custom Component Scripts in Plant 3D

What is a Component Script?

Plant 3D delivers a large catalog of predefined components. To get 3D representations of these components into 3D drawings "**Scripts**" are used. A "**Script**" is a small (**Python**) subroutine that takes the dimensions of a specific component as input and created a 3D representation (typically a solid within a block) as output.

Approximately 20,000 different scripts are part of **Plant 3D**, and additional scripts can easily be added. The available **scripts** cover nearly all types of components commonly used in plant design: pipes, elbows, flanges, tees, crosses, nozzles, olets, different types of valves and many more.

What is Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types and classes.

Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems and is extensible in C or C++.

IT is also usable as an extension language for applications that need a programmable interface.

Python is portable: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

Python website: http://www.python.org

Python FAQ: http://docs.python.org/faq/general.html#what-is-python

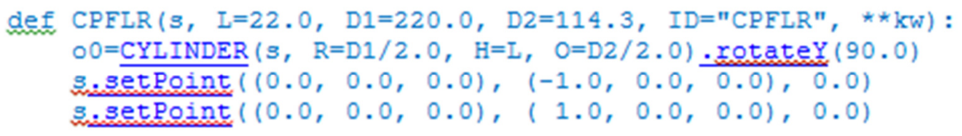
How Does Python Work with Plant 3D

In Plant 3D, AutoCAD and Python work together by embedding the Python interpreter in our application and providing a few C++/Python bindings as a Python extension module.

We define a Python module that exposes some modeler functionality (i.e. the primitives like BOX, CYLINDER, etc.) and some common operations along with a few calls to define the port data.

This Python module is then used in various content scripts. These scripts are evaluated with a set of parameters by a component that is called a content adapter to produce the geometry.

A Typical Component Script



```
def CPFLR(s, L=22.0, D1=220.0, D2=114.3, ID="CPFLR", **kw):
    o0=CYLINDER(s, R=D1/2.0, H=L, O=D2/2.0).rotateY(90.0)
    s.setPoint((0.0, 0.0, 0.0), (-1.0, 0.0, 0.0), 0.0)
    s.setPoint((0.0, 0.0, 0.0), ( 1.0, 0.0, 0.0), 0.0)
```

This one creates a simple lapped flange with two connection points (ports). Connection points have a position and a direction vector that allows for automatic alignment of connected components.

To get data from the database we need the database field information. That is also done inside the **script** file at the construction routine (normally directly after the **script** definition end) and looks like:

```
activate(CPFLR,
"""
[Type AQA-VCPFLR]
VID=STRING,32
DN=STRING
L=LENGTH
D1=LENGTH
D2=LENGTH
Units=STRING,8
;
uniqId=CALC =$self.VID$ $self.DN$
;
boltCalcVal=CALC CPF_GetNeededBoltParam(B=self.L.value(),
L=self.L.value())
;
@key=VID,DN
""",
"@VarDataDefault0",
)
```

This defines, for the lapped flange, the parameter of the parametric construction data that has to be stored inside database or needed for editors, the needed fields and some calculated fields.

Primitives

- ARC3D
  defines a 'normal' elbow
- ARC3D2
  defines a reduced elbow
- ARC3DS
  defines a segmented elbow
- BOX
  defines a box
- CONE
  defines a cone and also a frustum
- CYLINDER
  defines a 'normal' or an elliptical cylinder
- ELLIPSOIDHEAD
  defines a normalized ellipsoid head
- ELLIPSOIDHEAD2
  defines an ellipsoid head (0.5 * R)
- ELLIPSOIDSEGMENT
  defines an ellipsoid (football/rugby ball like)
- HALFSPHERE
  defines a half sphere
- PYRAMID
  defines a pyramid or a frustum of pyramid
- ROUNDRECT
  defines a transition from a rectangle to a circle
- SPHERESEGMENT
  defines a sphere segment

- **TORISPHERICHEAD**
  defines a normalized torispheric head

- **TORISPHERICHEAD2**
  defines a torispheric head (small radius = 25.00)

- **TORISPHERICHEADH**
  defines a normalized torispheric head with height

- **TORUS**
  defines a torus

Each **script** can also be used as a **primitive** to create other geometric bodies.

ARC3D

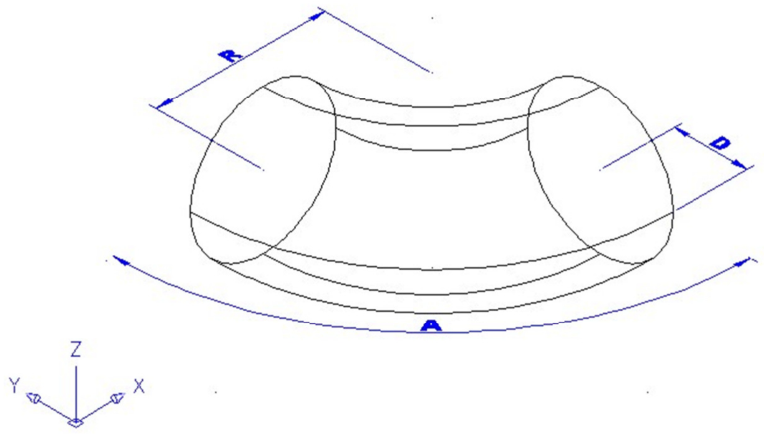
ARC3D defines a 'normal' elbow

The call from Python should be like:

```
s=ARC3D(s, D, R, A)
```

The parameters means:

s	the main object
D	the 1/2 diameter
R	the bend radius
A	the bend angle



The base point is the intersection between the thought centerline trough booth ends

ARC3D2

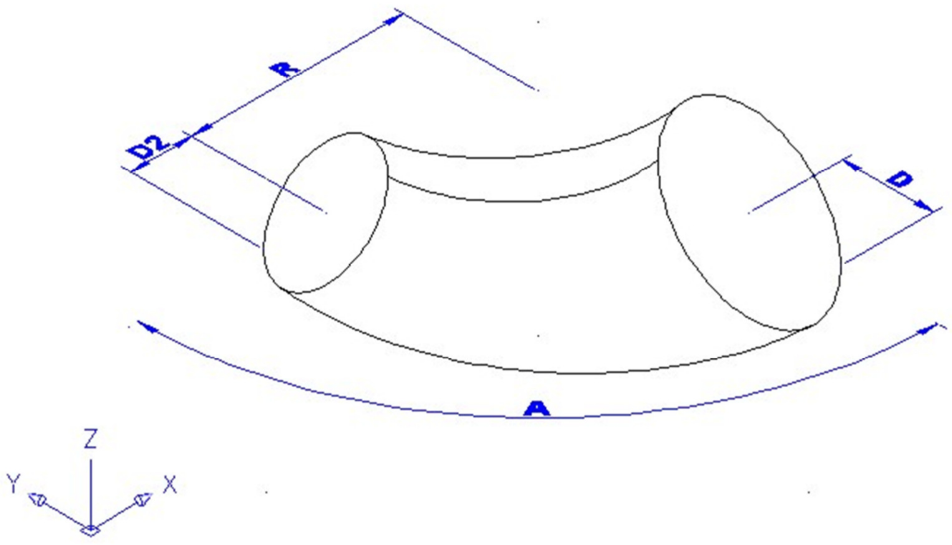
ARC3D2 defines a reduced elbow (also a 'normal' elbow can be done with them)

a call from **Python** should be like

```
s=ARC3D2(s, D, D2, R, A)
```

The parameters means:

s	the main object
D	the 1/2 diameter
D2	the second 1/2 diameter - if not set D ist used as D2
R	the bend radius
A	the bend angle



The base point is the intersection between the thought centerline trough booth ends

ARC3DS

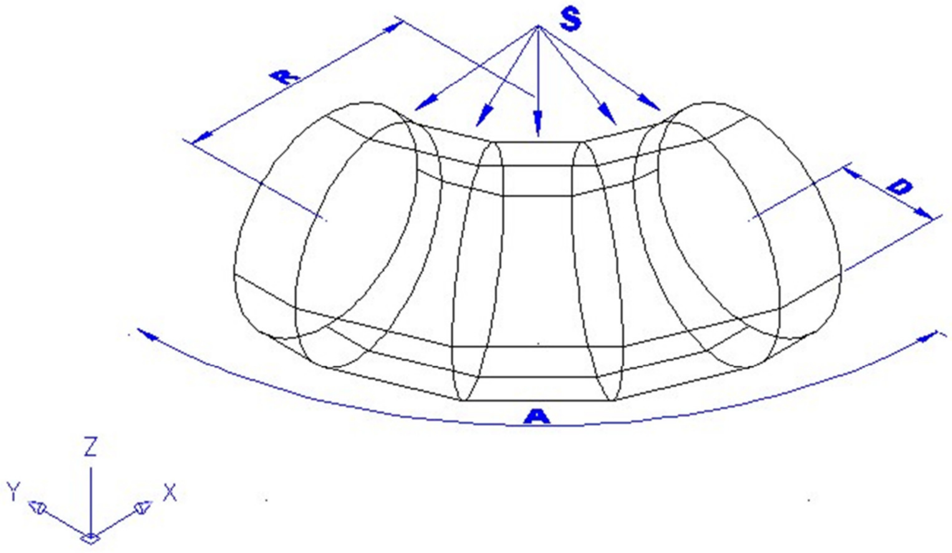
ARC3DS defines a segmented elbow

a call from **Python** should be like

```
s=ARC3DS(s, D, D2, R, A, S)
```

The parameters means:

s	the main object
D	the 1/2 diameter
R	the bend radius
A	the bend angle
S	number of segments



The base point is the intersection between the thought centerline trough booth ends

BOX

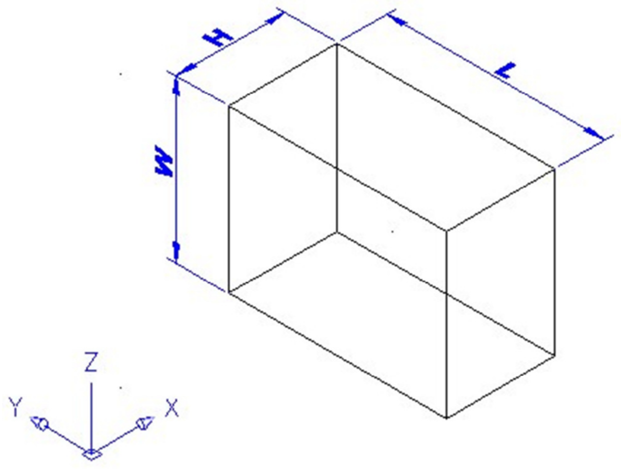
BOX defines a box

a call from **Python** should be like

```
s=BOX(s, L, W, H)
```

The parameters means:

s	the main object
L	the box length
W	the box width
H	the box height



The base point is the center of gravity.

CONE

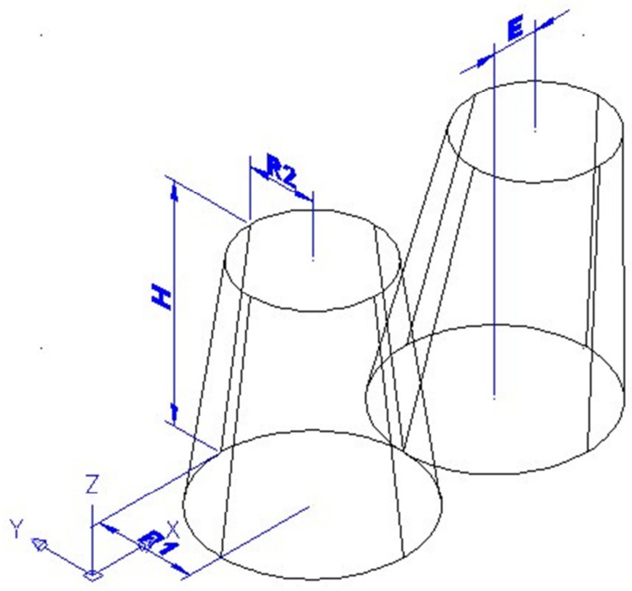
CONE defines a cone and also a frustum

a call from **Python** should be like

```
s=CONE(s, R1, R2, H, E)
```

The parameters means:

s	the main object
R1	the bottom radius
R2	the upper radius - if 0.0 a full cone, if > 0.0 a frustum is created
H	the height
E	the eccentricity between upper and bottom center



The base point is the center of bottom.

CYLINDER

CYLINDER defines a 'normal' or an elliptical cylinder

a call from Python should be like

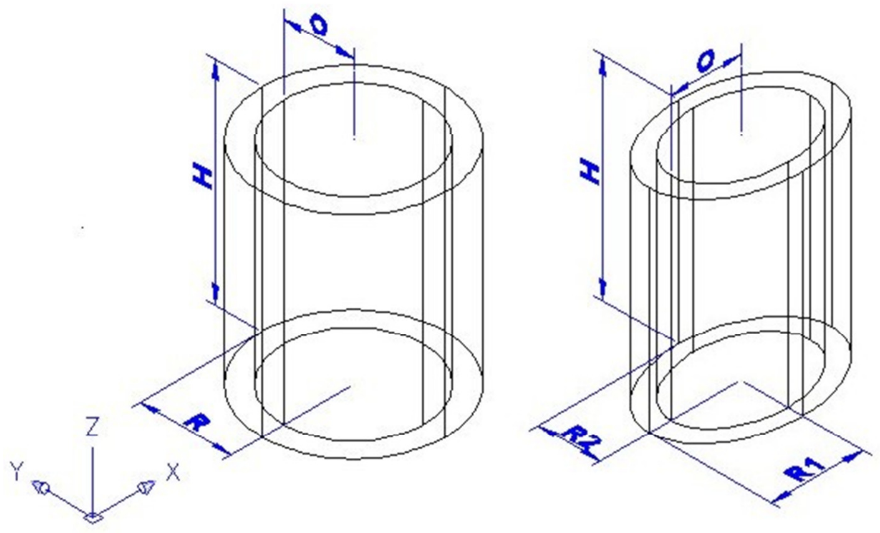
```
s=CYLINDER(s, R, H, O)
```

or

```
s=CYLINDER(s, R1, R2, H, O)
```

The parameters means:

s	the main object
R	the radius
R1	the 1. radius
R2	the 2. radius
H	the height
O	the hole radius



The base point is the center of bottom.

ELLIPSOIDHEAD

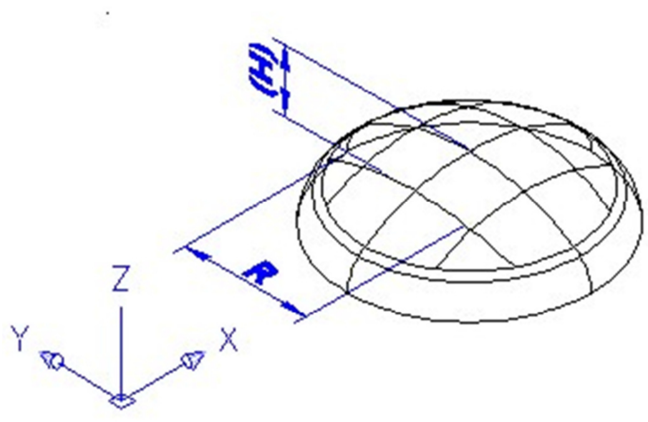
ELLIPSOIDHEAD defines a normalized ellipsoid head

a call from **Python** should be like

```
s=ELLIPSOIDHEAD(s, R)
```

The parameters means:

s	the main object
R	the radius



The base point is the center of bottom

with obj.**parameters**() we can get the height (H) of the primitive

Calculates the intersection iteratively. **ELLIPSOIDHEAD** is formed as an ellipsis. The body is interpolated with cones which are used to find the intersection points.

ELLIPSOIDHEAD2

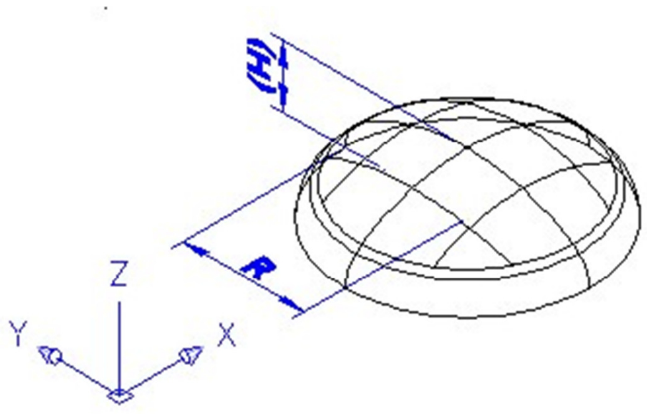
ELLIPSOIDHEAD2 defines a ellipsoid head (0.5 * R)

a call from **Python** should be like

```
s=ELLIPSOIDHEAD2(s, R)
```

The parameters means:

s	the main object
R	the radius



The base point is the center of bottom

with obj**.parameters**() we can get the height (H) of the primitive

Calculates the intersection iteratively. **ELLIPSOIDHEAD2** is formed as an ellipsis with the main axes a = radius and b = 0.5*radius .The body is interpolated with cones which are used to find the intersection points.

ELLIPSOIDSEGMENT

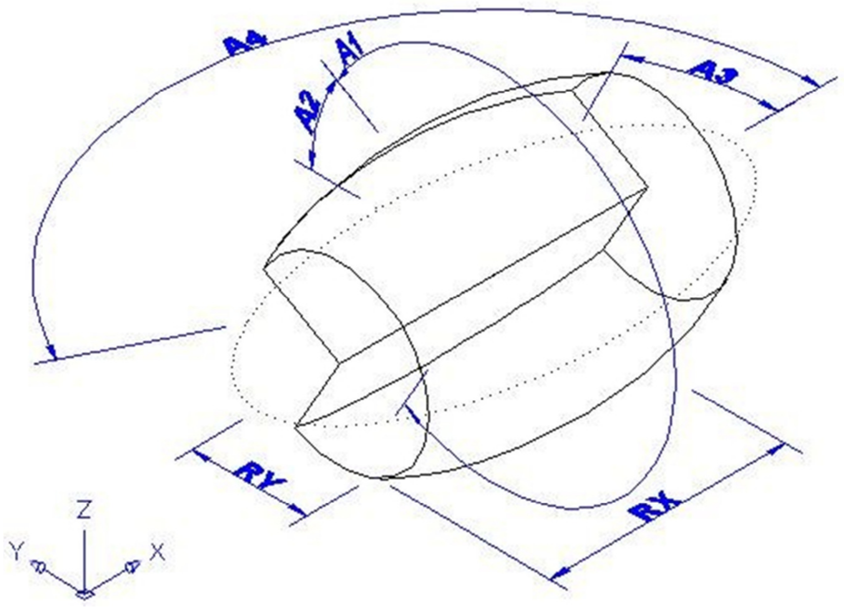
ELLIPSOIDSEGMENT defines an ellipsoid body (like a *football*/*rugby ball*)

a call from **Python** should be like

```
s=ELLIPSOIDSEGMENT(s, RX, RY, A1, A2, A3, A4)
```

The parameters means:

s	the main object
RX	big ellipsoid axis
RY	small ellipsoid axis
A1	complete rotation angle
A2	start angle of rotation
A3	start angle of ellipse
A4	end angle of ellipse



The base point is the center of gravity.

HALFSPHERE

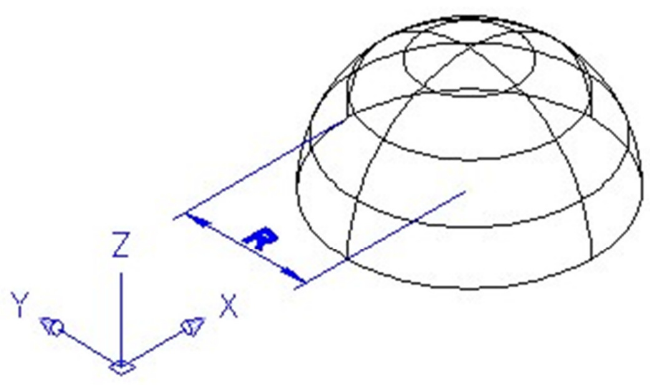
HALFSPHERE defines a half sphere

a call from **Python** should be like

```
s=HALFSPHERE(s, R)
```

The parameters means:

s	the main object
R	the radius



The base point is the center of bottom.

PYRAMID

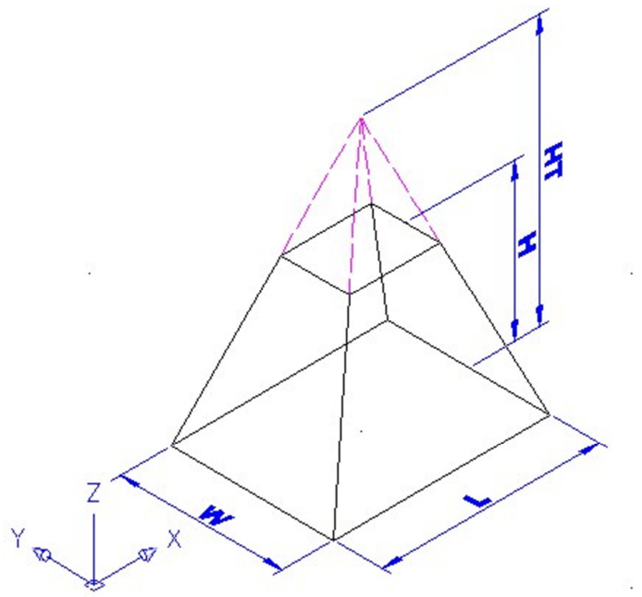
PYRAMID defines a pyramid or a frustum of pyramid

a call from **Python** should be like

```
s=PYRAMID(s, L, W, H, HT)
```

The parameters means:

s	the main object
L	the length
W	the width
H	the frustum height. if **HT** isn't set it's the total pyramid height
HT	the total pyramid height



The base point is the center of the bottom rectangle.

ROUNDRECT

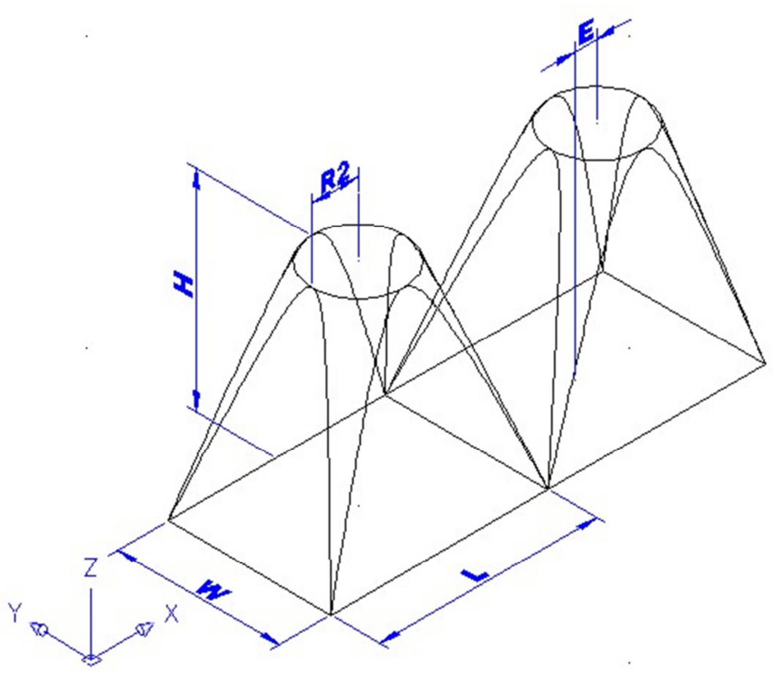
ROUNDRECT defines a transition from a rectangle to a circle

a call from **Python** should be like

```
s=ROUNDRECT(s, L, W, H, R2, E)
```

The parameters means:

s	the main object
L	the length
W	the width
H	the height
R2	the circle radius
E	the eccentricity between upper and bottom center



The base point is the center of the rectangle.

SPHERESEGMENT

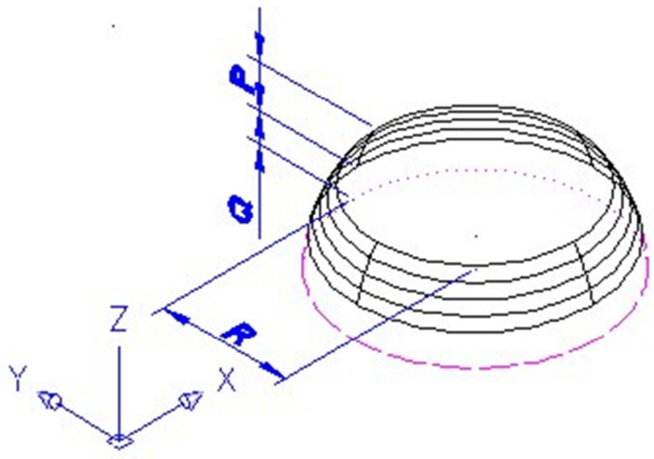
SPHERESEGMENT defines a sphere segment

a call from Python should be like

```
s=SPHERESEGMENT(s, R, P, Q)
```

The parameters means:

s	the main object
R	the sphere radius
P	the segment height
Q	the height where the segment starts from center



The base point is the center of the lower part of the segment.

TORISPHERICHEAD

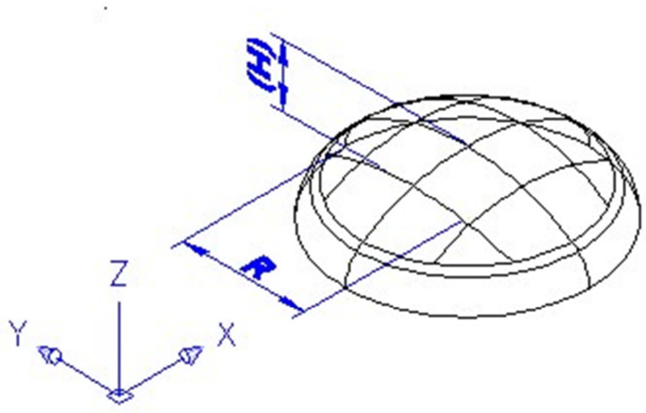
TORISPHERICHEAD defines a normalized torispheric head

a call from **Python** should be like

```
s=TORISPERICHEAD(s, R)
```

The parameters means:

s	the main object
R	the radius



The base point is the center of bottom.

with obj**.parameters**() we can get the height (H) of the primitive.

TORISPHERICHEAD2

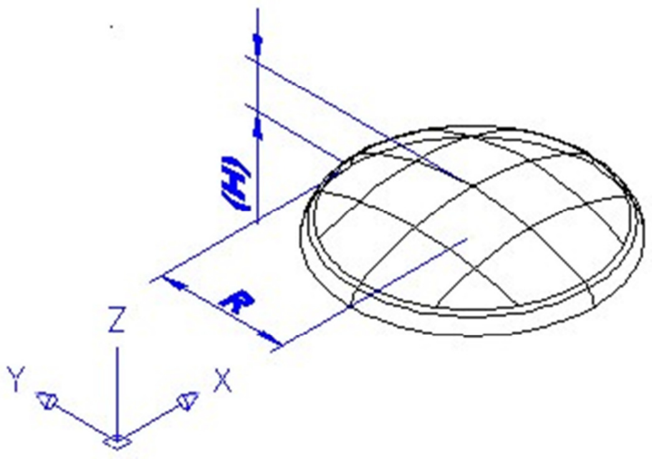
TORISPHERICHEAD2 defines a torispheric head (small radius = 25.00)

a call from **Python** should be like

```
s=TORISPERICHEAD2(s, R)
```

The parameters means:

s	the main object
R	the radius



The base point is the center of bottom

with obj.**parameters**() we can get the height (H) of the primitive.

TORISPHERICHEADH

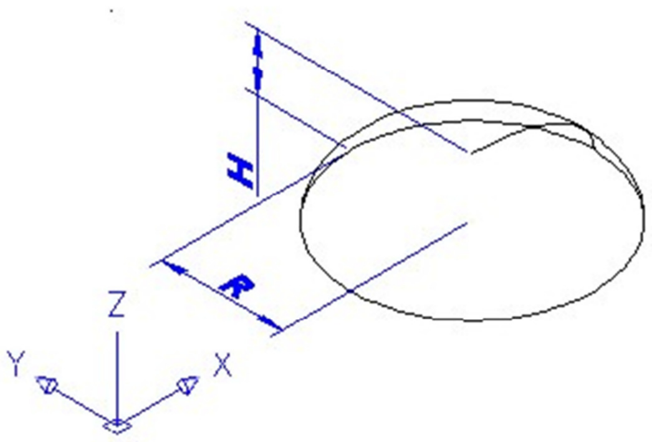
TORISPHERICHEADH defines a normalized torispheric head with height

a call from **Python** should be like

```
s=TORISPERICHEADH(s, R, H)
```

The parameters means:

s	the main object
R	the radius
H	the height



The base point is the center of bottom.

TORUS

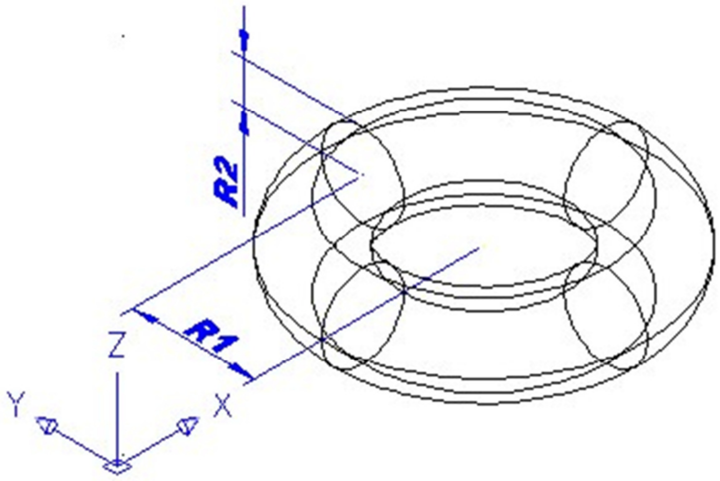
TORUS defines a torus

a call from **Python** should be like

```
s=TORUS(s, R1, R2)
```

The parameters means:

s	the main object
R1	the outer radius
R2	the inner radius



The base point is the intersection between the thought centerline trough booth ends.

Modifier Functions

Plant 3D offers some member functions to modify objects (primitives) like rotate, move,...

There are also some member request functions to get additional information from the object.

obj**.uniteWith**
unites 2 objects

obj**.subtractFrom**
subtracts 1 object from another 1

obj**.intersectWith**
creates an intersection of 2 objects

obj**.erase**
removes an object from memory

obj**.rotateX**
rotate the object round the X-axis

obj**.rotateY**
rotate the object round the Y-axis

obj**.rotateZ**
rotate the object round the Z-axis

obj**.translate**
moves the object

obj**.setTransformationMatrix**
set the object's transformation matrix

obj**.setPoint**
append connection point to the body

Request functions

obj**.parameters**
return the object's construction parameters

obj**.transformationMatrix**
return the object's current transformation matrix

obj**.numberOfPoints**
return number of (connection) points

obj**.pointAt**
return the position of a connection point

obj**.directionAt**
return the direction at a connection point

.uniteWith

The modifier function **.uniteWith** unites the calling object ***obj*** with the second object ***oobj***.

The second object ***oobj*** is given to the function as argument.

The result is set to ***obj***.

After the unite the second object ***oobj*** has to be removed from memory with **.erase**

a call from **Python** should be like

```
obj.uniteWith(oobj)
```

It looks like:

```
def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", **kw):
    O=CON_OF_DIV(D2)/2.0
    o1=CYLINDER(s, R=D2/2.0, H=L-B, O=O).rotateY(90).translate((B,
0, 0))
    o0=CYLINDER(s, R=D1/2.0, H=B, O=O).rotateY(90)
    o0.uniteWith(o1)
    o1.erase()
    o2=CYLINDER(s, R=O, H=L-B, O=0.0) ).rotateY(90).translate((B, 0,
0))
    o0.subtractFrom(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((L, 0, 0), ( 1, 0, 0))
```

.subtractFrom

The modifier function **.subtractFrom** subtracts the second object ***oobj*** from the calling object ***obj***.

The second object ***oobj*** is given to the function as argument.

The result is set to ***obj***.

After the subtract the second object ***oobj*** has to be removed from memory with **.erase**.

a call from **Python** should be like

```
obj.subtractFrom(oobj)
```

It looks like:

```
def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", **kw):
    O=CON_OF_DIV(D2)/2.0
    o1=CYLINDER(s, R=D2/2.0, H=L-B, O=O).rotateY(90).translate((B,
0, 0))
    o0=CYLINDER(s, R=D1/2.0, H=B, O=O).rotateY(90)
    o0.uniteWith(o1)
    o1.erase()
    o2=CYLINDER(s, R=O, H=L-B, O=0.0).rotateY(90).translate((B, 0,
0))
    o0.subtractFrom(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((L, 0, 0), ( 1, 0, 0))
```

.intersectWith

The modifier function **.intersectWith** creates an intersect object with the calling object ***obj*** and the second object ***oobj***.

The second object ***oobj*** is given to the function as argument.

The result is set to ***obj***.

After the intersect the second object ***oobj*** has to be removed from memory with **.erase**.

a call from **Python** should be like

```
obj.intersectWith(oobj)
```

It looks like:

```
def CADAPT_SQ2RO_Sub(s, LL=715.0, LW=700.0, R=225.0, OL=919.19,
OW=698.28, H=2515.45, ID="CADAPT_SQ2RO_Sub", **kw):
    TL=R*2.0
    TW=R*2.0
    RB=sqrt(pow(LL, 2.0)+pow(LW, 2.0))/2.0
    RA=90.0-asDegrees(atan2(OL, OW))
    RE=sqrt(pow(OL, 2.0) + pow(OW, 2.0))
    o1=CONE(s, R1=RB, R2=R, H=H, E=RE).rotateZ(RA)
    o2=CADAPT_SQ2SQ_Sub(s, LL=LL, LW=LW, TL=TL, TW=TW, OL=OL, OW=OW,
H=H)
    o1.intersectWith(o2)
    o2.erase()
    s.setPoint((0.0, 0.0, 0.0), (0.0, 0.0, -1.0))
    s.setPoint((OL,  OW,  H  ), (0.0, 0.0,  1.0))
```

.erase

The modifier function **.erase** has always to be called after **.uniteWith**, **.subtractFrom** and **.intersectWith**.

.erase removes ***obj*** from the memory.

a call from **Python** should be like

```
obj.erase()
```

It looks like:

```
def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", **kw):
    O=CON_OF_DIV(D2)/2.0
    o1=CYLINDER(s, R=D2/2.0, H=L-B, O=O).rotateY(90).translate((B, 
0, 0))
    o0=CYLINDER(s, R=D1/2.0, H=B, O=O).rotateY(90)
    o0.uniteWith(o1)
    o1.erase()
    o2=CYLINDER(s, R=O, H=L-B, O=0.0) ).rotateY(90).translate((B, 0, 
0))
    o0.subtractFrom(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((L, 0, 0), ( 1, 0, 0))
```

.rotateX

The modifier function **.rotateX** rotates the calling object ***obj*** around the X-axis.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

```
obj.rotateX(a)
```

It looks like:

```
def CSGC004(s, D=116.0, L=30.0, W=4.0, K=0.0, H=95.0, M1=10.0,
M2=0.0, ID="CSGC004", **kw):
    R1=D/2.0
    R2=R1+W
    H1=H-M2
    if H1<(R2+W) or M2==0.0:
        H1=R2+W
    KWdt=M1*tan(aqa.math.asRadiants(30.0))
    H2=H-H1
    o1a=BOX(s, L=M1, W=KWdt, H=H2).translate((0.0-(H2/2.0)-H1, 0.0,
0.0))
    o1b=BOX(s, L=M1, W=KWdt, H=H2).translate((0.0-(H2/2.0)-H1, 0.0,
0.0)).rotateX( 60.0)
    o1c=BOX(s, L=M1, W=KWdt, H=H2).translate((0.0-(H2/2.0)-H1, 0.0,
0.0)).rotateX(-60.0)
    o2=CSGC003(s, D=D, L=L, W=W, K=K, H=H1-R1)
    o1a.uniteWith(o1b)
    o1b.erase()
    o1a.uniteWith(o1c)
    o1c.erase()
    o1a.rotateZ(180.0)
    o1a.uniteWith(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((0, 0, 0), (1, 0, 0))
    s.setPoint((H, 0, 0), (1, 0, 0))
```

.rotateY

The modifier function **.rotateY** rotates the calling object obj around the Y-axis.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

```
obj.rotateY(a)
```

It looks like:

```
def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", **kw):
    O=CON_OF_DIV(D2)/2.0
    o1=CYLINDER(s, R=D2/2.0, H=L-B, O=O).rotateY(90).translate((B,
0, 0))
    o0=CYLINDER(s, R=D1/2.0, H=B, O=O).rotateY(90)
    o0.uniteWith(o1)
    o1.erase()
    o2=CYLINDER(s, R=O, H=L-B, O=0.0).rotateY(90).translate((B, 0,
0))
    o0.subtractFrom(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((L, 0, 0), ( 1, 0, 0))
```

.rotateZ

The modifier function **.rotateZ** rotates the calling object obj around the Z-axis.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

```
obj.rotateZ(a)
```

It looks like:

```
def CSGC004(s, D=116.0, L=30.0, W=4.0, K=0.0, H=95.0, M1=10.0, 
M2=0.0, ID="CSGC004", **kw):
    R1=D/2.0
    R2=R1+W
    H1=H-M2
    if H1<(R2+W) or M2==0.0:
        H1=R2+W
    KWdt=M1*tan(aqa.math.asRadiants(30.0))
    H2=H-H1
    o1a=BOX(s, L=M1, W=KWdt, H=H2).translate((0.0-(H2/2.0)-H1, 0.0, 
0.0))
    o1b=BOX(s, L=M1, W=KWdt, H=H2).translate((0.0-(H2/2.0)-H1, 0.0, 
0.0)).rotateX( 60.0)
    o1c=BOX(s, L=M1, W=KWdt, H=H2).translate((0.0-(H2/2.0)-H1, 0.0, 
0.0)).rotateX(-60.0)
    o2=CSGC003(s, D=D, L=L, W=W, K=K, H=H1-R1)
    o1a.uniteWith(o1b)
    o1b.erase()
    o1a.uniteWith(o1c)
    o1c.erase()
    o1a.rotateZ(180.0)
    o1a.uniteWith(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((0, 0, 0), (1, 0, 0))
    s.setPoint((H, 0, 0), (1, 0, 0))
```

.translate

The modifier function **.translate** moves the calling object ***obj*** along the given vector ***v***.

The argument ***v*** can be a **mPoint**, **mVector** or a **3-Tupel *(x, y, z)***.

a call from **Python** should be like

```
obj.translate(v)
```

It looks like:

```
def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", **kw):
    O=CON_OF_DIV(D2)/2.0
    o1=CYLINDER(s, R=D2/2.0, H=L-B, O=O).rotateY(90).translate((B,
0, 0))
    o0=CYLINDER(s, R=D1/2.0, H=B, O=O).rotateY(90)
    o0.uniteWith(o1)
    o1.erase()
    o2=CYLINDER(s, R=O, H=L-B, O=0.0) ).rotateY(90).translate((B, 0,
0))
    o0.subtractFrom(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((L, 0, 0), ( 1, 0, 0))
```

.setTransformationMatrix

The modifier function **.setTransformationMatrix** sets the object's transformation matrix ***t***.

a call from **Python** should be like

```
    obj.setTransformationMatrix(t)
```

It looks like:

```
def CPNO(s, D=114.3, R=300.0, L=100.0, D2=0.0, A=90.0, OF=0.0,
ID="CPNO", **kw):
    vC=CPNO_util(D=D, D2=D2, OF=OF, L=L, R=R, A=A)
    R=vC[0]/2.0
    R2=vC[1]/2.0
    o1=ARC3D(s, D=R, R=vC[6], A=vC[2])
    p2 = o1.pointAt(1)
    if vC[2] == 90.0:
        xMove = 0.0 - p2.x()
        yMove = vC[7] - p2.y()
    else:
        xMove = 0.0 - p2.x() - (L * cos(asRadiants(vC[2])))
        yMove = vC[7] - p2.y() - (L - (L * sin(asRadiants(vC[2]))))
    t1 = mTransform().translate((xMove, yMove, 0.0))
    o1.setTransformationMatrix(t1)
    s.setPoint((0, 0, 0), (0, -1, 0))
    s.setPoint(t1 * o1.pointAt(1), o1.directionAt(1))
    if vC[5]:
        d1 = ARC3D(s, D=R-vC[4], R=vC[6], A=vC[2])
        d1.setTransformationMatrix(t1)
        o1.subtractFrom(d1)
        d1.erase()
    o3=CYLINDER(s, R=R2, H=vC[6]*2.0,
O=0.0).rotateY(270).translate((vC[6], 0, 0))
    if vC[7] < vC[6]:
        o2 = BOX(s, H=vC[6]*2.0, W=vC[0]*2.0,
L=vC[6]).translate(((vC[6]/2.0), -(vC[6]/2.0), 0.0))
        o3.uniteWith(o2)
        o2.erase()
    o1.subtractFrom(o3)
    o3.erase()
```

.setPoint

The modifier function **.setPoint** insert a point at position ***p*** and set the direction ***v*** to the next part inside this point.

Alternative we can also set a rotation angle ***a*** (most needed for elliptical flanges to get the right alignment)

The argument ***p*** and ***v*** can be a **mPoint**, **mVector** or a **3-Tupel *(x, y, z)***.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

```
obj.setPoint(p,v)
```

or

```
obj.setPoint(p,v,a)
```

It looks like:

```
def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", **kw):
    O=CON_OF_DIV(D2)/2.0
    o1=CYLINDER(s, R=D2/2.0, H=L-B, O=O).rotateY(90).translate((B,
0, 0))
    o0=CYLINDER(s, R=D1/2.0, H=B, O=O).rotateY(90)
    o0.uniteWith(o1)
    o1.erase()
    o2=CYLINDER(s, R=O, H=L-B, O=0.0) ).rotateY(90).translate((B, 0,
0))
    o0.subtractFrom(o2)
    o2.erase()
    s.setPoint((0, 0, 0), (-1, 0, 0))
    s.setPoint((L, 0, 0), ( 1, 0, 0), 0)
```

.parameters

The request function **.parameters** returns the object's construction parameters

a call from **Python** should be like

```
obj.parameters()
```

It looks like:

```
def CPC(s, D=114.30, L=64.00, DISP=0, OF=-1.0, ID="CPC", **kw):
    if DISP == 0:
        o2 = TORISPHERICHEAD(s, R=D/2.0).rotateY(90.0)
        vT = o2.parameters()
        vTH = float(vT["H"])
        if L - vTH < 0.0:
            o2.erase()
            o2 = TORISPHERICHEAD(s, R=D/2.0, H=L).rotateY(90.0)
            vTH = L
        o2.translate((L-vTH, 0.0, 0.0))
    else:
        vTH = CON_OF_MULT(D)-D
        o2 = CYLINDER(s, R=(D+vTH)/2.0, H=vTH,
O=0.0).rotateY(90).translate((L-vTH, 0.0, 0.0))
    if L - vTH > 0.0:
        o1 = CYLINDER(s, R=D/2.0, H=L-vTH, O=0.0).rotateY(90.0)
        o1.uniteWith(o2)
        o2.erase()
    if not OF == 0.0:
        if OF == -1:
            O = CON_OF_DIV(D)
        else:
            O = D-(OF * 2.0)
        L2 = L-(D-O)+OF
        if DISP == 0:
            o4 = TORISPHERICHEAD(s, R=O/2.0).rotateY(90.0)
            o4.translate((L-vTH, 0.0, 0.0))
        else:
            o4 = None
        if L - vTH > 0.0:
            o3 = CYLINDER(s, R=O/2.0, H=L-vTH, O=0.0).rotateY(90.0)
            if o4:
                o3.uniteWith(o4)
                o4.erase()
            else:
```

```
            o3 = o4
    o1.subtractFrom(o3)
    o3.erase()
s.setPoint((0.0, 0.0, 0.0), (-1.0, 0.0, 0.0), 0.0)
```

.transformationMatrix

The request function .transformationMatrix returns the object's current transformation matrix

a call from **Python** should be like

```
obj.transformationMatrix()
```

.numberOfPoints

The request function **.numberOfPoints** returns the object's number of (connection) points

a call from **Python** should be like

```
obj.numberOfPoints()
```

.pointAt

The request function **.pointAt** returns the position of connection point ***n*** as mPoint.

If **inECS** is set to ***1*** the mPoint is not transformed to WCS - default for **inECS** is ***0***.

a call from **Python** should be like

```
obj.pointAt(n, inECS)
```

It looks like:

```
def CPNO(s, D=114.3, R=300.0, L=100.0, D2=0.0, A=90.0, OF=0.0,
ID="CPNO", **kw):
    vC=CPNO_util(D=D, D2=D2, OF=OF, L=L, R=R, A=A)
    R=vC[0]/2.0
    R2=vC[1]/2.0
    o1=ARC3D(s, D=R, R=vC[6], A=vC[2])
    p2 = o1.pointAt(1)
    if vC[2] == 90.0:
        xMove = 0.0 - p2.x()
        yMove = vC[7] - p2.y()
    else:
        xMove = 0.0 - p2.x() - (L * cos(asRadiants(vC[2])))
        yMove = vC[7] - p2.y() - (L - (L * sin(asRadiants(vC[2]))))
    t1 = mTransform().translate((xMove, yMove, 0.0))
    o1.setTransformationMatrix(t1)
    s.setPoint((0, 0, 0), (0, -1, 0))
    s.setPoint(t1 * o1.pointAt(1), o1.directionAt(1))
    if vC[5]:
        d1 = ARC3D(s, D=R-vC[4], R=vC[6], A=vC[2])
        d1.setTransformationMatrix(t1)
        o1.subtractFrom(d1)
        d1.erase()
    o3=CYLINDER(s, R=R2, H=vC[6]*2.0,
O=0.0).rotateY(270).translate((vC[6], 0, 0))
    if vC[7] < vC[6]:
        o2 = BOX(s, H=vC[6]*2.0, W=vC[0]*2.0,
L=vC[6]).translate(((vC[6]/2.0), -(vC[6]/2.0), 0.0))
        o3.uniteWith(o2)
        o2.erase()
    o1.subtractFrom(o3)
    o3.erase()
```

.directionAt

The request function **.directionAt** returns the direction of connection point ***n*** as mVector.

If **inECS** is set to ***1*** the mVector is not transformed to WCS - default for **inECS** is ***0***.

a call from **Python** should be like

```
obj.directionAt(n, inECS)
```

It looks like:

```
def CPNO(s, D=114.3, R=300.0, L=100.0, D2=0.0, A=90.0, OF=0.0,
ID="CPNO", **kw):
    vC=CPNO_util(D=D, D2=D2, OF=OF, L=L, R=R, A=A)
    R=vC[0]/2.0
    R2=vC[1]/2.0
    o1=ARC3D(s, D=R, R=vC[6], A=vC[2])
    p2 = o1.pointAt(1)
    if vC[2] == 90.0:
        xMove = 0.0 - p2.x()
        yMove = vC[7] - p2.y()
    else:
        xMove = 0.0 - p2.x() - (L * cos(asRadiants(vC[2])))
        yMove = vC[7] - p2.y() - (L - (L * sin(asRadiants(vC[2]))))
    t1 = mTransform().translate((xMove, yMove, 0.0))
    o1.setTransformationMatrix(t1)
    s.setPoint((0, 0, 0), (0, -1, 0))
    s.setPoint(t1 * o1.pointAt(1), o1.directionAt(1))
    if vC[5]:
        d1 = ARC3D(s, D=R-vC[4], R=vC[6], A=vC[2])
        d1.setTransformationMatrix(t1)
        o1.subtractFrom(d1)
        d1.erase()
    o3=CYLINDER(s, R=R2, H=vC[6]*2.0,
O=0.0).rotateY(270).translate((vC[6], 0, 0))
    if vC[7] < vC[6]:
        o2 = BOX(s, H=vC[6]*2.0, W=vC[0]*2.0,
L=vC[6]).translate(((vC[6]/2.0), -(vC[6]/2.0), 0.0))
        o3.uniteWith(o2)
        o2.erase()
    o1.subtractFrom(o3)
    o3.erase()
```

Additional Functions

The **variants** need and offer some functions to link to **Plant 3D** or for testing:

There are:

activate()
activate the variant and defines the needed database fields

TESTACPSCRIPT
an AutoLISP® function to see and test the variant parameters

demand loader
Loader for the variants on demand

activate

To use a variant definition as Plant 3D part we have to **activate** the function.

To get and convert the data from database we need also the field information. That is also done inside activate.

a call from **Python** should be like

```
activate(func, dbtype, defkey)
```

The parameters means:

func	the **Python** function to activate (in this case a variant)
dbtype	this is the database definition. this definition has to be done between **"""** and **"""** (each has 3 quotation marks)
defkey	database definition key (it's always **"@VarDataDefault0"** inside **Plant 3D**)

For example:

```
activate(CPFLR,
"""
[Type AQA-VCPFLR]
VID=STRING,32
DN=STRING
L=LENGTH
D1=LENGTH
D2=LENGTH
Units=STRING,8
;
uniqId=CALC =$self.VID$ $self.DN$
;
boltCalcVal=CALC CPF_GetNeededBoltParam(B=self.L.value(),
L=self.L.value())
;
@key=VID,DN
""",
"@VarDataDefault0",
)
```

This defines for the lapped flange now the needed fields and some calculated fields.

dbtype

This is the database / parameter definition.

This definition has to be done between **"""** and **"""** (each has 3 quotation marks)

And then it looks like:

```
"""
[Type AQA-VCPFLR]
VID=STRING,32
DN=STRING
L=LENGTH
D1=LENGTH
D2=LENGTH
Units=STRING,8
;
uniqId=CALC =$self.VID$ $self.DN$
;
boltCalcVal=CALC CPF_GetNeededBoltParam(B=self.L.value(),
L=self.L.value())
;
@key=VID,DN
""",
```

Its defined as:

1. Each dbtype definition starts with 3 quotation marks
   “””
2. The next line describes the **variant** name and should start with **[AQA_V** and end with **]**
   [Type AQA-VCPFLR**]**
   (the definition with **AQA-V** comes from the adapter which read ***ACPlant Designer*** parametric scripts and can’t changed now)
3. The next 2 lines are also always the same (come also from adapter and should be added for compatibility)

   VID=STRING,32
   DN=STRING
4. Now we define for each needed script argument an extra definition line
   each line starts with the argument name then we use the = sign as delimiter and now we add the type.

After the definition we can also add char, numbers length , delimited with ‘**,**’, but not needed inside **Plant 3D** (**ACPlant Designer** used this definition to create database tables)

DN=STRING
L=LENGTH
D1=LENGTH
D2=LENGTH
Units=STRING,8

allowed values for the type are

a. **LENGTH** for length values that also can be calculated to another unit (like mm -> in)
b. **ANGLE** is used for set and real angle valve
c. **INT** a real integer value
d. **DOUBLE** a float value
e. **STRING** a string

5. Now we can define some help fields like
   uniqId (for calculating unique blocknames)
   boltCalcVal,…
   we can set here each parameter that we want and with **=CALC** we can call each function that we want and set the return value inside the wished parameter
   uniqId=CALC =$self.VID$ $self.DN$
   boltCalcVal=CALC CPF_GetNeededBoltParam(B=self.L.value(), L=self.L.value())

6. The line **@key=** was also use inside **ACPlant Designer** for database creation and defined the table index
   @key=VID,DN

7. Comments can be set with ‘**;**’

8. And now comes the ending 3 quotation marks
   “””

TESTACPSCRIPT / TESTACPSCRIPT1

To test and check the geometry body we have the **AutoLISP®** function **TESTACPSCRIPT** and **TESTACPSCRIPT1** created.

The first parameter at booth script types is the variants name (like "CPFLR")

The next parameters for **TESTACPSCRIPT** are couples of values: the fieldname and the needed value (like "D1" "300.5").

For **TESTACPSCRIPT1** we have to set all the parameters inside 1 string (like it’s done inside catalog/spec dbs: “D1=300.5”)

If a field isn't set the function takes the default values from the **variants** definition.

That looks like:

```
(TESTACPSCRIPT "CPFLR")

(TESTACPSCRIPT "CPFLR" "D1" "300.5")

(TESTACPSCRIPT "CPFLR" "L" "40" "D1" "300.5" "D2" "88.9")
```

Or

```
(TESTACPSCRIPT1 "CPFLR")

(TESTACPSCRIPT1 "CPFLR" "D1=300.5")

(TESTACPSCRIPT1 "CPFLR" "L=40,D1=300.5,D2=88.9")
```

Before we can use **TESTACPSCRIPT** or **TESTACPSCRIPT1** inside **Plant 3D** we have to load **PnP3dACPAdapter.arx**.

That can be done with

```
(arxload "PnP3dACPAdapter.arx")
```

or with the **appload** function

demand loader

Approximately 20.000 different **variant** subroutines needs there time to load. To speed up the start procedure we created the **Variant Demand Loader**.

First we packed all the variants into an zip archive so we solved the operating system problems with the much directories **- Python** use this zip archive as package and get the files really fast.

At second we load the **variants** only if we need them. For that we created a *dummy* loader file that includes only links to the variants.

This file is compiled at build time to **variants.map**.

The contents of the file **dummy_var.load** looks like (variant;python_package_like_location):

```
CPFBO;varmain.flangesub.cpfbo
CPFBR;varmain.flangesub.cpfbr
CPFLO;varmain.flangesub.cpflo
CPFLR;varmain.flangesub.cpflr
CPFWO;varmain.flangesub.cpfwo
CPFWO_F_SF;varmain.flangesub.cpfwo_f_sf
CPFWR;varmain.flangesub.cpfwr
CPFWR_F_SF;varmain.flangesub.cpfwr_f_sf
CPFWRI;varmain.flangesub.cpfwri
CPFWRI_F_SF;varmain.flangesub.cpfwri_f_sf
FLANGE;varmain.flangesub.cpf_old
CPFW;varmain.flangesub.cpf_old
```

How to add a Custom Script and its Metadata

The following steps create a custom script named “TESTSCRIPT” and its metadata including script image previews:

1. Create a folder named “CustomScripts” under content folder “C:\AutoCAD Plant 3D 2012 Content/CPak Common/”

2. Create a custom python script, such as “TestScript.py” under the created folder.
   How to create custom python script is the key point, generally, there are three steps to create as follows:

   a. Import needed Python packages:

      ```
      from varmain.primitiv import *
      from varmain.custom import *
      ```

   b. Write metadata section:

      ```
      @activate(Group="Support", TooltipShort="Test script", TooltipLong="This is a
      custom Testscript", LengthUnit="in")
      @group("MainDimensions")
      @param(D=LENGTH, TooltipShort="Cylinder Diameter", Ask4Dist=True)
      @param(L=LENGTH, TooltipLong="Length of the Cylinder")
      @param(OF=LENGTH0)
      @group(Name="meaningless enum")
      @param(K=ENUM)
      @enum(1, "align X")
      @enum(2, "align Y")
      @enum(3, "align Z")
      ```

      The purpose for above code is to create metadata for custom script. Meta data includes tooltips for script, tooltips for script parameters, group information for script and its parameters.

Following is a more detailed description for the above metadata section:

- GROUP= script is in the "Support" group
- TooltipShort= display name is "test script"
- TooltipLong= description is "This is a custom testscript"
- LengthUnit= length unit is Inches
- Parameters D, L, and OF are in "MainDimensions" group. May not be zero (0).
- D carries a flag (Ask4Dist) that tells us its length may also be given by specifying a distance in the model.
- Parameter K is in "meaningless enum" group. K is an enumerated Value, where 1 means "align X", 2 means "align Y", etc.

Following are key words used in this section:

@activate:
==========
must be the very first decorator. Declares the scripts own metadata.
Allows these properties to be specified:

    Group, FirstPortEndtypes, Ports, TooltipShort, TooltipLong,
    ThumbnailImage, ParamDimImage, SCEditDimImage, Language, LengthUnit

@group:
=======
Defines a parameter group.
It usually precedes @param declarations and allows these properties to be specified:

    Name, TooltipLong

@param:
=======
Defines a parameter's metadata.
Has these properties:

    Name, Type, Value, TooltipShort, TooltipLong, Ask4Dist

Name and Type can be either specified like @param(Name="A", Type="ANGLE", ...) or by @param(A=ANGLE, ...)

@enum:
======
Defines the enumeration for its preceding ENUM type parameter.
Arguments are:

    Value, TooltipLong

The declaration arguments follow standard Python function call syntax; they may be given by position or by keyword.

c. Write main function for the custom script, for example:

```
def TESTSCRIPT(s, D=80.0, L=150.0, OF=-1, K=1, **kw):
    CYLINDER(s, R=D/2, H=L, O=0.0).rotateY(90)
```

The above code defines a script named “TESTSCRIPT” which creates a cylinder. Here, users can use their imagination to create what they want.

3. Now, we have a custom script named “TestScript.py” under folder “C:\AutoCAD Plant 3D 2012 Content /CPak Common/CustomScripts”. We need to register it. In the AutoCAD command window, use command “PLANTREGISTERCUSTOMSCRIPTS” to register the newly added script. If successful, some metadata files will be created in the custom script folder as follows:
   a. TestScript.pyc
      This is the complied binary file for python source file “TestScript.py”.
   b. ScriptGroup.xml
      This file includes the group information of all added custom scripts. For example, the group name of added script “TESTSCRIPT” is “Support”. The group names can be the Plant 3D class names, such as “Elbow”, “Tee” and so on.
   c. TESTSCRIPT.xml
      This file describes the metadata of script “TestScript”.
   d. variants.xml
      This file holds the tooltips for the script and its parameters, parameter group names, and enumerator values.
   e. variants.map
      This is a map file which contains mapping information from script name to script path.

   If syntax problems present in the “TestScript.py” file, there will be some output information in the AutoCAD command window. In this case, we need to fix the errors and register again.

   If successful, the added script could be used by “pipe supports” feature in plant. The scripts within “Support” group will be recognized by this feature.

4. In command window, Use lisp command (TESTACPSCRIPT "TESTSCRIPT") to test the added script. *NOTE: Before we can use TESTACPSCRIPT inside Plant 3D we have to make sure PnP3dACPAdapter.arx is loaded. If not, it can be loaded using the AutoCAD APPLOAD command.*
   If the script works, a block will be inserted into current model at coordinates 0,0,0.

5. Assuming we have inserted the block created by the script, we can create its preview images as followings:

   a. In AutoCAD command window, run the “PlantSnapShot” command.
   b. Select “Part” option to create image for a single part.

c. Select image size in options [200/64/32].
d. Input the image name in Save File dialog. Following is the naming rule for custom script previews:

Scriptname_200.png	200x200
Scriptname_64.png	64x64
Scriptname_32.png	32x32

For example, we need to input “TestScript_200.png” for its 200x200 preview.

Save the image to custom script folder. When user request the custom script images from the image API, the image API will scan the custom script folder to fetch the images which meet requirement.