First interesting commands:

🟧 = modifying spaces to put your system characteristics.

to load an existing project:
loadProject('name_of_file.FLU')
to open a new project: newProject()
to save a project: saveProject()
to save a project as: saveProjectAs('name_of_file')
to close a project: closeProject()
to activate batch mode: putBatchMode()
to remove batch mode: resetBatchMode()
to get the PyFlux command: getPyFluxCommand()

Geometry:

Comments: These functions give an example for each object but there sometimes exist different ways for creating an entity.
When we define an object name, Flux automatically put it in capital letters, so it is important to handle them in capital after.

To define a coordinates system:
```
CoodSysCartesian(name='name_of_coordinate_system',
              parentCoodSys=Local(coordSys=CoorSys['XYZ1']),
              origin=['Coord1', 'Coord2', 'Coord3'],
              rotationAngles=RotationAngles(angleZ='angle'))
```
# this is a cartesian coordinate system defined by a local coordinate system XYZ1.

To define a parameter:
```
ParameterGeom(name='name_of_parameter',
              expression='algebraic_value')
```
To create a point:
```
PointCoordinates(color=Color['White'],
               Visibility=Visibility['VISIBLE'],
               coordSys=CoordSys['XYZ1'],
               uvw=['Coord1', 'Coord2', 'Coord3'],
                nature=Nature['STANDARD'])
```
To create a line:
```
LineSegment(color=Color['White'],
          visibility=Visibility['VISIBLE'],
          defPoint=[Point[num_P1], Point[num_P2]],
          nature=Nature['STANDARD'])
```
# Creation of a segment defined by starting and ending points.

To build surfaces and volumes:
buildFaces()
buildVolumes()

To delete an element:
Point[1].delete() / Point[1].deleteForce()
Line[4].delete() / Line[4].deleteForce()

To create an infinite box:
```
InfiniteBoxCylinderZ(size=['inner_radius', 'outer_radius',
       'inner_size_half_height',  'outer_size_half_height'])
```
# build an infinite box which is a Z cylinder box.

To complete an infinite box:
```
InfiniteBoxCylinderZ['InfiniteBoxCylinderZ'].complete3D(
                  buildingOption=
    'ADD VOLUMES, FACES, LINES AND POINTS',
                  coordSys=CoorSys['XYZ1'],
                  linkedMesh='ADD LINK MESH ASSOCIATED')
```
To define a symmetry:
```
SymmetryXYplane(physicalType=SymmetryTangentMagFields(),
              Position='0')
```
To define a periodicity:
```
PeriodicityAngularZaxis(physicalType=PeriodicityCyclicType(),
                 angles=['included_angle',
                        'offset_angle_with_respect_to_plan'])
```
# Define a periodicity in rotation about Z axis with angle of domain.

To create a transformation:
```
TransfTranslationVector(name='name_of_transformation',
                 coordSys=CoordSys['XYZ1'],
                 Vector=['coord1', 'coord2', 'coord3'])
```
# transformation defined by a translation vector.

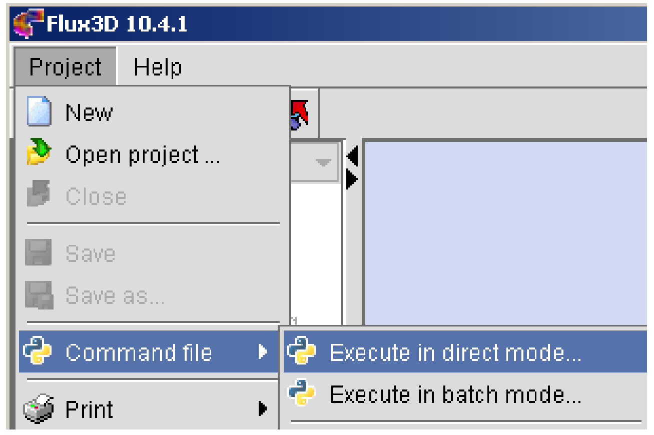
To apply a transformation by propagation:
```
PointCoordinates[3].propagate(transformation=Transf['trans1'],
                     repetitionNumber=1)
```
# propagation of all the points with the transformation transf1.

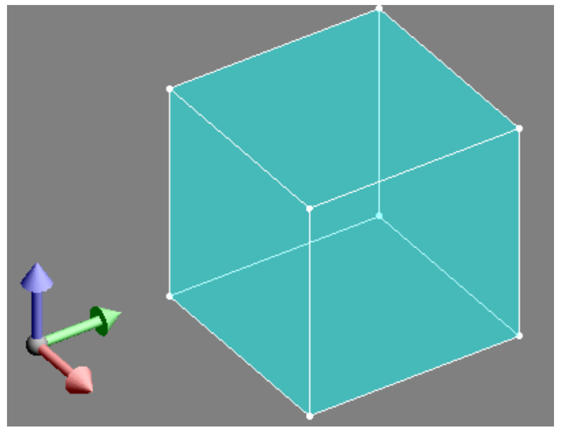
To apply a transformation by extrusion:
```
PointCoordinates[5].extrude(transformation=Transf['transf1'],
                     repetitionNumber=1,
                     extrusionType='STANDARD')
```
To modify a parameter of an already created funtion:
Point[3].visibility=Visibility['INVISIBLE']
# the point 3 which was already existing became invisible.

Example of geometry building:

```
#! Flux3D 10.4

newProject()

List1=[['0', '0', '0'],
       ['0', '1', '0'],
       ['1', '1', '0'],
       ['1', '0', '0'],
       ['0', '0', '1'],
       ['0', '1', '1'],
       ['1', '1', '1'],
       ['1', '0', '1']]

for i in List1:      # cf. loops “for” in the Python memento
   PointCoordinates(color=Color['White'],
                    visibility=Visibility['VISIBLE'],
                    coordSys=CoordSys['XYZ1'],
                    uvw=i,
                    nature=Nature['STANDARD'])

List2=[(1,2),
       (2,3),
       (3,4),
       (4,1),
       (1,5),
       (2,6),
       (3,7),
       (4,8),
       (5,6),
       (6,7),
       (7,8),
       (8,5)]

for i in List2:
   LineSegment(color=Color['White'],
               visibility=Visibility['VISIBLE'],
               defPoint=[Point[i[0]], Point[i[1]]],
               nature=Nature['STANDARD'])

buildFaces()
buildVolumes()
```

Result in Flux:





Mesh:

To create a mesh point:
```
MeshPoint(name='name_mesh_point'
        lenghtUnit=LenghtUnit['MILLIMETER'],
        value='algebraic_value')
```
To create a mesh line:
```
MeshLineArithmetic(name='name_mesh_line',
               color=Color['White'],
               number=N)
```
# Creation of an arithmetical mesh line which has a number N(integer) of discretization by line.

To generate a mesh:
```
MeshGeneratorLinked(name='name_mesh',
               color=Color['White'],
               linked=Transf['transf1'])
```
# Linked mesh which use the transformation transf1..

To attribute a mesh:
Point[6].mesh=MeshPoint['name_mesh_point']
Line[3].mesh=MeshLine['name_mesh_line']
# Or, you can directly precise the type of meshing with the mesh parameter to add in the creating functions to lines and points.

To check a mesh:
checkMesh()

To mesh faces and volumes:
meshDomain()

To delete a mesh:
deleteMesh()

Physics:

To define the application:
```
ApplicationMagneticDC3D(formulationModel=
                      MagneticDC3DAutomatic(),
    scalarVariableOrder=ScalarVariableAutomaticOrder(),
    vectorNodalVariableOrder=VectorNodalVariableAutomaticOrder(),
    coilCoefficient=CoilCoefficientAutomatic())
```
# magneto static application

To delete the current application:
DeleteCurrentApplication()

To create a material:
```
Material(name='name_material',
       propertyBH=PropertyBhLinear(mur='valeur'),
       propertyJE=PropertyJeLinear(rhp='valeur'))
```
# Creation of a material with electric and magnetic properties.

To modify a material:
Material['name_material'].thermalConductivity=KtIsotropic(k='2')
# I add a thermal property.

To create a face region:
```
RegionFace(name='name_of_face_region',
         magneticDC3D=MagneticDC3DFaceAirGap(thickness='5'),
         color=Color['Cyan'],
         visibility=Visibility['VISIBLE'])
```
# /!\ the chosen application appeared at the second line, hence it is compulsory to define the application before and put the same.

To create a volume region:
```
RegionoVolume(name='name_of_volume_region',
           magneticDC3D=MagneticDC3DVolumeVacuum(),
           color=Color['Turquoise'],
           visibility=Visibility['VISIBLE'])
```
To apply a region to a face, a volume:
Face[10].region=RegionFace['nom_region']
Volume[1].region=RegionVolume['nom_region']

To create a mechanical set:
```
MechanichalSetRotationAxis(name='name',
           kinematic=RotatingMultiStatic(),
           rotationAxis=RotationXAxis(coordSys=CoorsSys['XYZ1'],
                                  pivot=['0', '0', '0']))
```
To check physics:
checkPhysic()

To create a non meshed coil:
```
CoilCircular(strandedCoil=CoilConductor['name'],
          turnNumber='value' ;
          seriesOrParallel=AllInSeries(),
          coilDuplicationBySymmetriesPeriodicities=CoilDuplication(),
          coordSys=CoordSys['XYZ1'],
          center=['coord1',
                 'coord2',
                 'coord3'],
          radius='radius',
   section=ComposedCoilPointSection(sectionRadius='value'),
          color=Color['Turquoise'],
          visibility=Visibility['VISIBLE'])
```

Parameter/Quantity:

To create an I/O parameter:
```
VariationParameterPilot(name='name_of_parameter',
                 referenceValue=value)
```
To create a sensor:
```
SensorPredefinedMagForce(name='name_of_sensor',
        support=
   ComputationSupportVolumeRegion(
  region=[RegionVolume['name_of_face_region_1'],
         RegionVolume['name_of_face_region_2']]))
```

Solving:

To create a solving scenario:
Scenario(name='name_of_scenario')
```
Scenario['name_of_scenario'].addPilot(pilot
        =MultiValues(parameter=VariationParameter['TIME'],
  Intervals=[IntervalStepValue(minValue=0.0,
                         maxValue=30,
                         stepValue=0.1)]))
```
To check a project:
checkProject()

To solve a project:
```
Scenario['name_of_scenario'].solve(projectName=
                     'name_of_project.FLU')
```
# You can solve a scenario in saving the project in the same file or in a different file.

To select the step:
# first step:
Scenario['scenario_name'].selectFirstStep()
# go to the next step:
Scenario[' scenario_name '].selectNextStep()
# with a loop:
```
i=1
while (Scenario[' scenario_name '].existNextStep()==1):
   Scenario[' scenario_name '].selectNextStep()
   i = i +1
```
# Selection by index:
Scenario[' scenario_name '].selectIndexStep(index=2)
# Selection by values:
```
selectCurrentStep(activeScenario=Scenario['scenario_name'],
             parameterValue=['parameter_name=value'])
```

Post processing:

To quit the post processing mode (delete all the results):
DeleteAllResults(deletePostprocessingResults='YES')

To create a spacial group:
```
GroupeRegionVolume(name='name_of_group',
           regionVolume=[RegionVolume['region1'],
                       RegionVolume['regionN']])
```
To create a path:
```
PathLocalizationConstraine2PTS(name='name_of_path',
             startPoint=PathPoint(uvw=['coord1',
                                   'coord2',
                                   'coord3'],
                           constraint='Face[2]'),
             endPoint=PathPoint(uvw=['coord1',
                                  'coord2',
                                  'coord3'],
                           constraint='Face[2]'),
        localizationConstrainded='ShortOnFace[2]',
             discretization=50,
             regionVolume=['region1'],
             color=Color['White'],
             visibility=Visibility['VISIBLE'])
```
To display isovalues:
```
IsovaluesSpatialGroup(name='isovalues_1',
             formula='H',
             forceVisibility='YES',
             smoothValue='YES',
             regionInternalExternal='AUTOMATIC',
             internalComputation='NO',
             regionType='VOLUME REGION',
             group=[Groupspatial['spacial_group_1'],
                   Groupspatial['spacial_group_2']])
```
# The isovalues will be displayed on the spatial groups. The characteristics which are sentences are options that you have to choose among severals. They are in English in the English version.

To erase isovalues:
displayIsovalues()
# if this function is used when there is not any isovalue, then it will display the Flux density isovalues by default.

To display isolines:
```
IsolineSpatialGroup(name='ISOLI NE_1',
             formula='dEmagV',
             forceVisibility='YES',
             smoothValue='YES',
             regionInternalExternal='AUTOMATIC',
             internalComputation='NO',
             regionType='VOLUME REGION',
             group=[Groupspatial['SPATIALGROUP_1'],
                  Groupspatial['SPATIALGROUP_2']])
```
To erase isolines:
displayIsolines()

# Same comments as previously, if we use this function at the begining, it will display the Flux isolines by default.

To display arrows:

```
ArrowSpacialGroup(name='ARROWS_1',
                  formula='Mur',
                  forceVisibility='YES',
                  regionType='VOLUME REGION',
                  group=[Groupspatial['region1'],
                         Groupspatial['regionN']])
```

To erase arrows:

```
displayArrows()
```

To calculate a curve on a path:

```
SpatialCurve(name='nom',
             path=Path['path_1'],
             formula=['Comp(1,H)'])
```

To calculate a 2D curve:

```
EvolutiveCurve2D(name='Curve2D',
    evolutivePath=EvolutivePath(parameterSet=
        [SetParameterXVariable(paramEvol=
            VariationParameter['ANGPOS_ROTOR'],
                          limitMin=0.0,
                          limitMax=7.5)]),
    formula=['AngPos(ROTOR)',
       'TIME'])
```

Predefined computation:

- o **Magnetic force:**

```
result=computeForceElecMagVolumeRegion(volumeRegion=
                          RegionVolume['REGION_1'],
                          …
                          RegionVolume['REGION_N']],
                resultName='ElecMagForceRegVol_1')
```

- o **Magnetic torque:**

```
result = ComputeTorqueElecMagVolumeRegion(axis=
    xAxis(pivotPoint=AnalysingPoint(coordSys=CoordSys['XYZ1'],
                                uvw=['0',
                                     '0',
                                     '0'])),
                volumeRegion=[RegionVolume['REGION_1'],
                          RegionVolume['REGION_N']],
                resultName='ElecMagTorqueRegVol_1')
```

- o **Magnetic energy:**

```
result = computeEnergyElecMagDomaine(resultName=
                'ElecMagEnergyDomain_1')
```

Integral on face:

```
result = IntegralFace(support=SupportIntegralFace(faces=[Face[98]],
                          regionFace='STATOR_AIR'),
                spatialFormula='J',
                integrationType=DomainDepthIntegration(),
                resultName='IntergralSurf_1')
```

The Overlays:

Comments: The overlays allow simplify the building of a complex system like a motor for instance. Thus, the user gives his characteristics and the overlay create a dynamic motor.

To load a certified overlay:

```
loadOverlay(name='name_overlay.PFO')
```

To use a certified overlay:

Example with the overlay:
'Brushless_Permanent_Magnet_Motors_V2.PFO'

```
MotorBPM(name='name_motor',
        lengthUnit='MILLIMETER',
        mesh_factor=0.5,
        GAP=1.0,
        excentricity=ExcentricityNo(periodicity='YES',
                     bdr='TWO_LAYERS_AIRGAP'),
        rotor=RotorBreadLoaf(POLES=4,
           rotorAng=0.0,
           RADSH=10.0,
           LM=10.0,
           BETAM=120.0,
           NMBP=1,
           magnetType=RotorBreadLoafNot(INSET=0.0)),
        stator=StatorAirGapWdg(NSLOTS=12,
           lamShape=Circle(RAD3=50.0),
           statorConfiguration='NORMAL',
           statorAng=0.0,
           SD=14.0,
           TWS=5.0,
           TPR_T=0.0))
```

To quit a certified overlay:

```
leaveOverlay()
```

To delete a certified overlay:

Comments: When the system is created, the overlay is not usefull anymore but each time you launch the project, the overlay is load again. So, it is judicious to unload the overlay after use.

```
Overlay['NAME_OVERLAY'].unloadOverlay()
```

The macros:

To load a macro:

```
loadMacro(fileName='C:/Cedrat/Extensions/
          Macros/name_macro.PFM')
```

To use a macro:

```
name_macro(function_parameter)
```

# The Python syntax depends on the macro, each macro is different from the others and have different parameters.

To create a macro:

- o In the file C:/Cedrat/Extension/Macros you have to create a new folder with the extension « .PFM ».

Rules for the macro's name: no space, the capital letters are significant to distinguish the words.

- o In this folder, you have to create two files: MyMacro.py and MyMacro.gif.

  Comments: each file and folder of the macro must have the same name: MyMacro.PFM, MyMacro.py, MyMacro.gif. The text file allows writing the script and the picture file allows defining the icon.
- o Layout of a macro:

```
#! Preflu 3D 9.31

#----------------------
#  date, author
#  modifying date
#  usefulness of the macro
#----------------------

import repertory    # if there is

" " "
@param coordSys  CoordSys  1 1 XYZ1  Coordinate system
 …
" " "

def MaMacro (coordSys, …):
          # instructions
```

`#! Preflu 3D 9.31`	gives the Flux version. If you use a 2D version and you indicate a 3D version, the macro will not work.
`@param`	introduce a parameter
`coordSys`	gives the name of the parameter
`CoordSys`	gives the PyFlux type
`1 1`	minimal and maximal cardinalities
`XYZ1`	default value
`Coordinate system`	labels of parameter

This part between «"""» concerned the parameters of the function which will be informed by the user.

Indications for the writing of a PyFlux script:

When we proceed to a study in Flux, we follow five step:

1 – The geometry
2 – The mesh
3 – The physics
4 – The solving
5 – The post processing

Furthermore, in PyFlux like in Python, the capital letters matter in the entity's name. Nevertheless, Flux put in capital letter all the names. So when you create an entity:
TransfTranslationVector(name='Transformation_1', …)
You would have to write 'TRANSFORMATION_1' when you will call it in a function.

Example:
color=Color['White']

Comments: Colors are the only parameters in which only the first letter is in capital.

Furthermore, we can say that PyFlux is based on a previous version of Python. So you may find that some syntaxes are different and some methods are not defined.

Some interesting methods in Flux:

- object.existItem('name_of_objet')

Check if an object is existing in Flux. It will return a Boolean value.

- object[ALL]

Select all entities from the object (for instance all points).

- dir(entity)

Display all methods available for an entity

- Object.help()

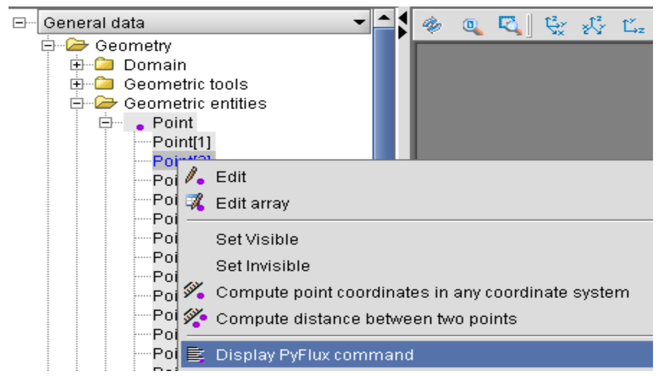
Write on screen a documentation explaining fields and methods available for the objet

Form of the most used types:

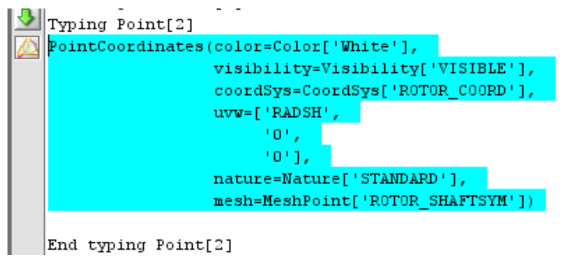
Type	Meaning
C80	Character string
R08	Value
I04	integer
File	File
Boolean	Boolean (can only be two values: true or false)
Color	Color
CoordSys	Coordinate System
Volume	Volume
RegionLine	Region line
RegionFace	Region face
RegionVolume	Region volume
Transf	Transformation
Material	Material
CoilConductor	Coil conductor
SensorPonctual	Sensor
VariationParameter	I/O parameter
Scenario	Scenario
Curve2d	2D curve

Displaying the PyFlux expression of an entity:

In Flux, you can display the PyFlux expression in doing a right click on it. For instance, if you want to know the PyFlux expression of a point:



Then, you can see the PyFlux expression in the horizontal tab:



```
Typing Point[2]
PointCoordinates(color=Color['White'],
                 visibility=Visibility['VISIBLE'],
                 coordSys=CoordSys['ROTOR_COORD'],
                 uvw=['RADSH',
                      '0',
                      '0'],
                 nature=Nature['STANDARD'],
                 mesh=MeshPoint['ROTOR_SHAFTSYM'])

End typing Point[2]
```

The first coordinate 'RADSH' is a parameter defined during the geometry step. Using a parameter as a coordinate can simplify its manipulation, all the more if this parameter is used for different points.

MEMENTO

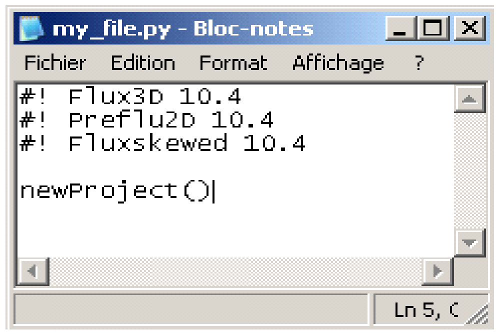
The PyFlux language for

Flux®

Tools for programming

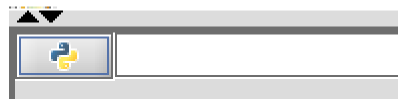
To begin:

- **To write and execute a script in Flux:**
  - o Open a basic text program like Pad
  - o Save it with the extension « name_of_file.py »
  - o Beginning of the script: with the version used
  - o In Flux: Project> Command File> execute in direct mode
  - o Choose the file in the directory



```
#! Flux3D 10.4
#! Preflu2D 10.4
#! Fluxskewed 10.4

newProject()
```

- **To write a PyFlux command directly in Flux:**
  - o In the horizontal tab at the low side of the windows
  - o Yoy only have to write the command at the right of this symbol and press « enter ».
  - o To open a script space bigger, you have to click on the same symbol and press to execute the script.



- **To see the PyFlux script of a hand-made action:**
  - o In the lower horizontal tab
  - o Or go in the folder and open the file « Flux3D_log.py » which is a temp file that record the last PyFlux commands that you have made in Flux.
    /!\ If you close your project and open it again, the script will be erase. To save it, you have to save it with an other file's name.

Altair
INNOVATION INTELLIGENCE®