Tutorial 10: Python scripting

Table of Contents

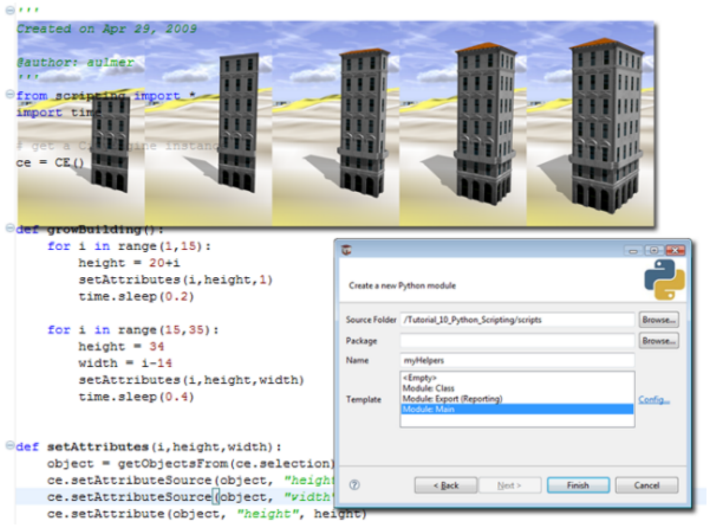
Tutorial 10: Python scripting

Download items

- Tutorial data
- Tutorial PDF

The Python scripting interface greatly enhances the possibilities of CityEngine. This tutorial explains the basic usage of the Python console and the editor and gives several examples on the automatization of CityEngine tasks.

More information on the CityEngine-specific Python command set can be found in the CityEngine help by clicking **Help** > **Help Contents** > **Python Scripting Reference**.



```
'''
Created on Apr 29, 2009

@author: aulmer
'''
from scripting import *
import tim

# get a C      gine instanc
ce = CE()

def growBuilding():
    for i in range(1,15):
        height = 20+i
        setAttributes(i,height,1)
        time.sleep(0.2)

    for i in range(15,35):
        height = 34
        width = i-14
        setAttributes(i,height,width)
        time.sleep(0.4)

def setAttributes(i,height,width):
    object = getObjectsFrom(ce.selection)
    ce.setAttributeSource(object, "height
    ce.setAttributeSource(object, "width'
    ce.setAttribute(object, "height", height)
```

Create a new Python module

Source Folder: /Tutorial_10_Python_Scripting/scripts
Package:
Name: myHelpers
Template:
<Empty>
Module: Class
Module: Export (Reporting)
Module: Main

The Python scripting interface is not available in all CityEngine versions.

Python console and editor

Tutorial setup

To get started, complete the following steps:

Steps:

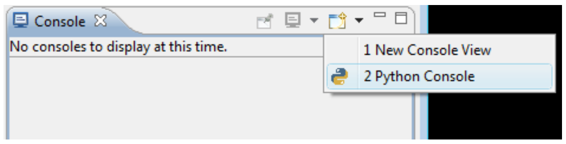
1. Import the `Tutorial_10_Python_Scripting` project into your CityEngine workspace.
2. Open the `Tutorial_10_Python_Scripting/scenes/01_PythonScripting.cej` scene.

Python console

Complete the following steps to open a new Python console:

Steps:

1. Open the console window by clicking **Window** > **Show Console**.
2. Open a Python console using the small triangle on the right side of the toolbar.

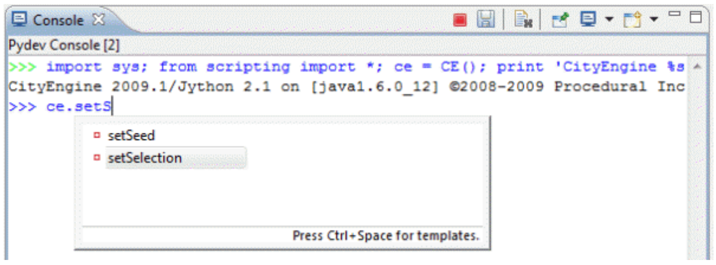


No consoles to display at this time.

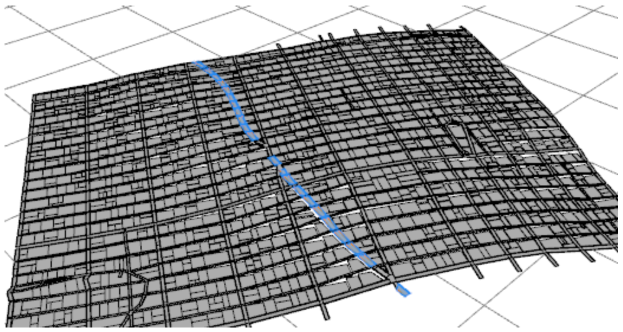
Your first CityEngine Python command is a fast way to select scene elements with a specific name.

3. Type `ce.setSelection`.
4. Press Ctrl+Space to show the command completion pop-up.
5. Type the `ce.setSelection(ce.getObjectsFrom(ce.scene, ce.withName("*Broadway*")))` command.
6. Press **Enter**.

   This selects all scene elements with names that contain the word "Broadway."



```
>>> import sys; from scripting import *; ce = CE(); print 'CityEngine %s
CityEngine 2009.1/Jython 2.1 on [java1.6.0_12] ©2008-2009 Procedural Inc
>>> ce.setS
```

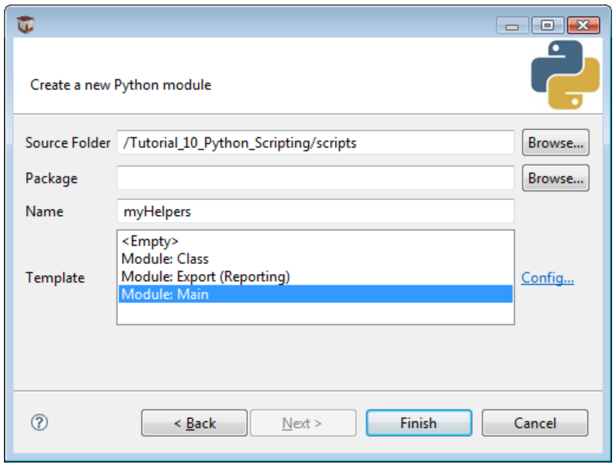


Python editor

As soon as you plan to use longer and more advanced Python commands, or a set of commands, it's helpful to use the Python editor in CityEngine.

Steps:

1. To create a new Python script, click **File** > **New** > **Python Module**.
   The Python module dialog box appears.
2. In the Python module dialog box, browse to the scripts folder of your project.
3. Type `myHelpers` as the name for your new Python module.
4. Select the **Module:Main** template.
5. Click **Finish**.



Source Folder: /Tutorial_10_Python_Scripting/scripts

Package:

Name: myHelpers

Template:
- <Empty>
- Module: Class
- Module: Export (Reporting)
- Module: Main

The new Python module myHelpers opens in the Python editor in CityEngine.

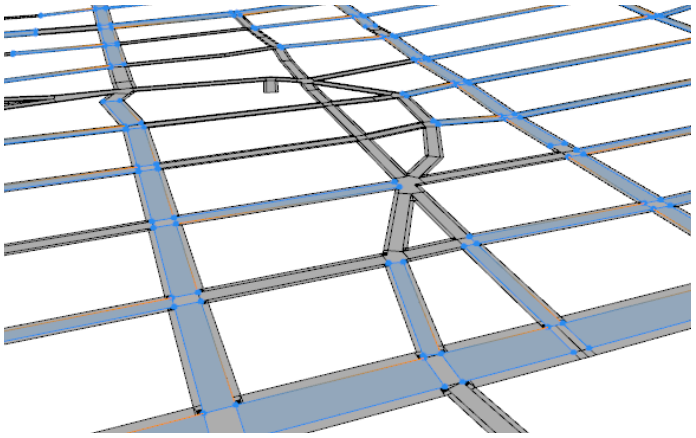
Add the new selectByAttribute(attr, value) function after the line ce = CE().

```
def selectByAttribute(attr, value):
    objects = ce.getObjectsFrom(ce.scene)
    selection = []
    for o in objects:
        attrvalue = ce.getAttribute(o, attr)
        if attrvalue  ==  value:
            selection.append(o)

    ce.setSelection(selection)
```

Call it with specific parameters in the main clause of the script. Make sure the main block is at the end of the file.

```
if __name__ == '__main__':
    selectByAttribute("connectionStart","JUNCTION")
```

To execute the script, click **Python** > **Run Script** in the menu, or press F9 while in the Python editor.



Run scripts from the console

Alternatively, you can call your helper scripts via the Python console by completing the following steps:

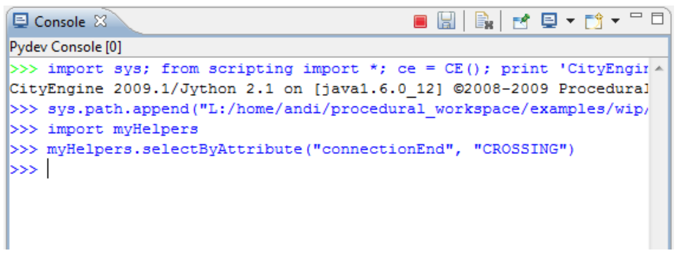
Steps:

1. In the Python console, add the path to your module to the system path.
2. Import your module.

```
>>> sys.path.append(ce.toFSPath("scripts"))
>>> import myHelpers
```

3. Call your helper function in the console in the following way, with arbitrary parameters:

```
myHelpers.selectByAttribute("connectionEnd", "JUNCTION")
```



```
>>> import sys; from scripting import *; ce = CE(); print 'CityEngin
CityEngine 2009.1/Jython 2.1 on [java1.6.0_12] ©2008-2009 Procedural
>>> sys.path.append("L:/home/andi/procedural_workspace/examples/wip/
>>> import myHelpers
>>> myHelpers.selectByAttribute("connectionEnd", "CROSSING")
>>> |
```

Extend scripting.py script

To extend scripting.py, complete the following steps:

Steps:

1. Create a new file `scripting.py` in your CityEngine workspace using the file browser of your operating system.
2. Add the following lines to automatically map your helper script at startup:

```
import sys

sys.path.append({PATH_TO_YOUR_SCRIPTS_DIRECTORY})
// e.g. sys.path.append("C:\user\CityEngine\MyProject\scripts")
import myHelpers
```

After a restart of CityEngine, your **myHelpers** module is loaded automatically. You can call the selection function in the console in the following way:

```
>>> myHelpers.selectByAttribute("connectionEnd", "JUNCTION")
```

Note: You can add arbitrary code to the `scripting.py` file. The startup module is executed automatically during CityEngine startup when a new Python console is opened and a script is run from the Python editor.
Make sure your `scripting.py` file is valid and executes correctly; otherwise, Python code in CityEngine cannot be executed. Open a Python console in CityEngine after you create or modify a `scripting.py` file; problems with executing the scripting file are displayed there.
`scripting.py` is read only once on CityEngine startup. If you modify the file, make sure to restart CityEngine.
If the script is not correctly updated on CityEngine startup, delete the Python cache directory `$USER_DIR/.cityengine/$CEVERSION_DIR/pythonCache/`.

Change street widths

Often, you may want to increment the street width attribute of many segments. If this cannot be accomplished easily in the GUI, a Python script can help.

Tutorial setup

Open the `Tutorial_10_Python_Scripting/scenes/02_PythonScripting.cej` scene.

Create new Python script

Steps:

1. Create a new rule file by clicking **File** > **New** > **Python** > **Python Module**.
2. Choose the project's script folder, name it `setStreetWidths`, and choose the **Module: Main** template.

incrementStreetWidths() function

This function increments the streetWidths attribute of all the selected street segments with a value specified by the user.

First, the function definition:

```
def incrementStreetWidths(increment):
```

You need to get all selected segments and loop over them.

```
selectedSegments = ce.getObjectsFrom(ce.selection, ce.isGraphSegment)
    for segment in selectedSegments:
```

To calculate the new street width, get the current value first using the ce.getAttribute() command. Note the syntax of the attribute name with the prefix /ce/street/; this accesses the user attributes of the object.

```
oldWidth = ce.getAttribute(segment, "/ce/street/streetWidth")
```

Finally, calculate the new street width by adding the user-provided parameter increment and assigning the new value to the segment.

```
newWidth = oldWidth+increment
ce.setAttribute(segment, "/ce/street/streetWidth", newWidth)
```

The entire function.

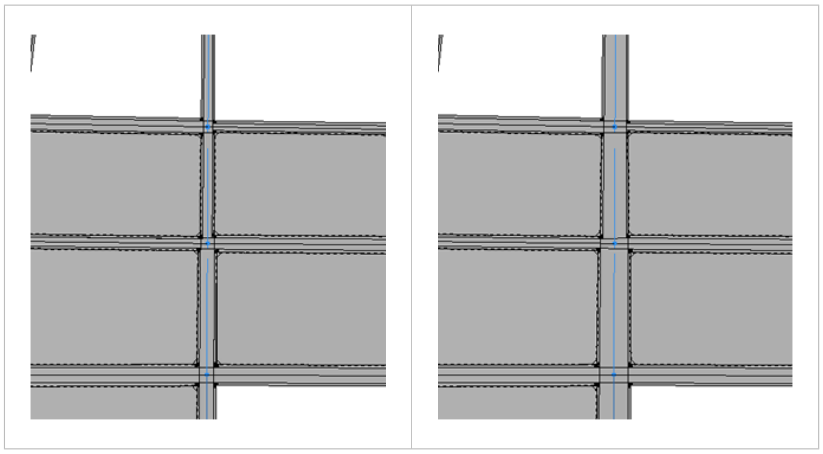
```
''' increment the street width parameter of all selected street segments'''
def incrementStreetWidths(increment):
    selectedSegments = ce.getObjectsFrom(ce.selection, ce.isGraphSegment)
    for segment in selectedSegments:
        oldWidth = ce.getAttribute(segment, "streetWidth")
        newWidth = oldWidth+increment
        ce.setAttribute(segment, "/ce/street/streetWidth", newWidth)
```

- In the main block of the script, add the function call, and choose an increment.

```
if __name__ == '__main__':
   incrementStreetWidths(10)
```

Select a set of street segments.

Run the Python script (Menu **Python** > **Run Script**), or pressF9 while in the Python editor.



Speed things up with @noUIupdate

Executing the previous script may take some time. This is because script execution in CityEngine runs in a separate thread and updates the GUI and the 3D viewport after every command. In this case, after every **setAttribute()** call, the street network is updated, and the 3D viewport is redrawn.

While this is convenient for this example, normal execution needs to be faster. This can be achieved by adding the **@noUIupdate** marker above the function definition.

```
@noUIupdate
def incrementStreetWidths(increment):
```

Functions marked this way will block GUI update during execution but, depending on what they do, will execute faster by factors.

⚠ **Caution:** Some combination of scripting commands with the **@noUIupdate** marker may freeze the user interface.
If you encounter a UI freeze or other unexpected behavior when using **@noUIupdate**, modify your scripts so that **@noUIupdate** only marks a small, specific function rather than marking your whole script.

multiplySegmentWidths() function

This function sets several attributes at the same time, namely, streetWidth, sidewalkWidthLeft, and sidewalkWidthRight. The user can specify a factor by which to multiply the widths.

```
@noUIupdate
def multiplySegmentWidths(factor):
    selectedSegments = ce.getObjectsFrom(ce.selection, ce.isGraphSegment)
    for segment in selectedSegments:
```

The helper function **multiplyAttribute** does the multiplication for the different attributes.

```
      multiplyAttribute(segment, "/ce/street/streetWidth", factor)
multiplyAttribute(segment, "/ce/street/sidewalkWidthLeft", factor)
multiplyAttribute(segment, "/ce/street/sidewalkWidthRight", factor)

def multiplyAttribute(object, attrname, factor):
    oldval = ce.getAttribute(object, attrname)
    newval = oldval*factor
    ce.setAttribute(object, attrname, newval)
```

multiplySegmentWidths and multiplyAttribute

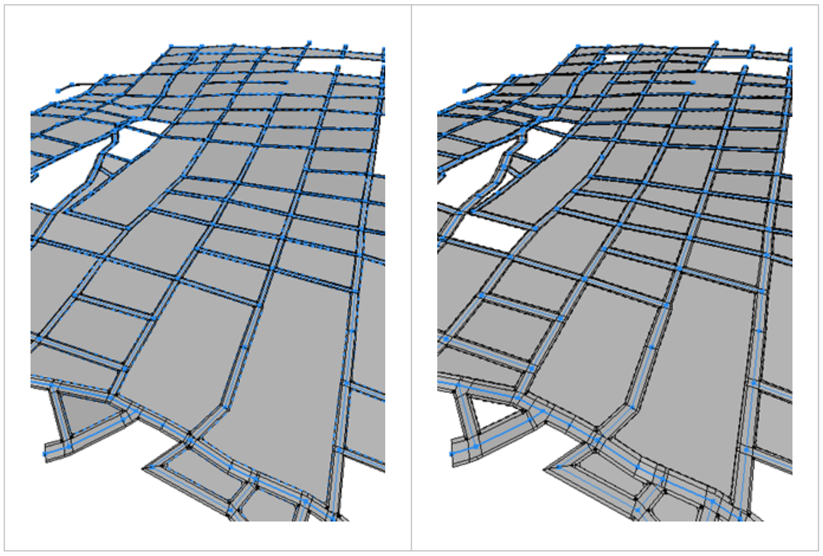
```
''' multiply street and sidewalk widths of all selected street segments by factor '''
@noUIupdate
def multiplySegmentWidths(factor):
    selectedSegments = ce.getObjectsFrom(ce.selection, ce.isGraphSegment)
    for segment in selectedSegments:
        multiplyAttribute(segment, "/ce/street/streetWidth", factor)
        multiplyAttribute(segment, "/ce/street/sidewalkWidthLeft", factor)
        multiplyAttribute(segment, "/ce/street/sidewalkWidthRight", factor)

''' multiply attribute of object by factor '''
def multiplyAttribute(object, attrname, factor):
    oldval = ce.getAttribute(object, attrname)
    newval = oldval*factor
    ce.setAttribute(object, attrname, newval)
```

In the main block of the script, add the function call, and choose a multiplication factor.

```
if __name__ == '__main__':
   multiplySegmentWidths(1.5)
```

Select a set of street segments.

Run the Python script by clicking **Python** > **Run Script**, or press F9 while in the Python editor.



Run from console

Rather than setting the function arguments in the Python editor, the functions described above can be called from the Python console after importing the script module.

```
>> scriptpath = ce.toFSPath("scripts")
>> sys.path.append(scriptpath)
>> import setStreetWidths
>> setStreetWidths.multiplySegmentWidths(0.5)
```

Set camera from the FBX file

This section shows how to import static camera data into CityEngine via FBX export from Maya.

Tutorial setup

Open the `Tutorial_10_Python_Scripting/scenes/02_PythonScripting.cej` scene.

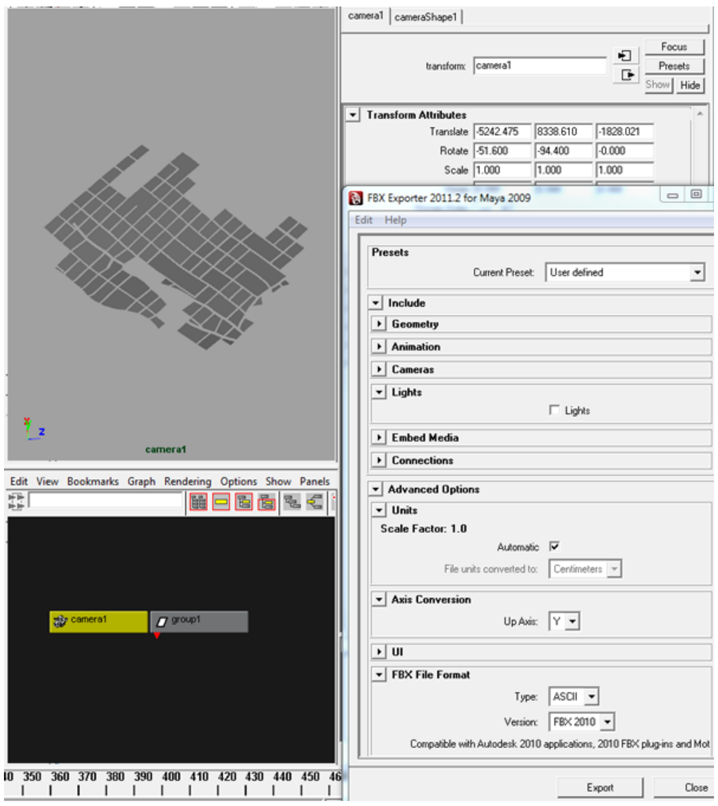
Export camera to FBX (Maya)

If you do not have Maya, you can skip the following steps, and use the existing `data/camera.fbx` file.

Steps:

1. In Maya, select the camera you want to export.
2. Click **File** > **Export Selection**.

In the export dialog box, make sure the settings are set as in the following screen capture:



transform: camera1

Transform Attributes

Translate	-5242.475	8338.610	-1828.021
Rotate	-51.600	-94.400	-0.000
Scale	1.000	1.000	1.000

Presets
Current Preset: User defined

Lights
☐ Lights

Units
Scale Factor: 1.0
Automatic ☑
File units converted to: Centimeters

Axis Conversion
Up Axis: Y

FBX File Format
Type: ASCII
Version: FBX 2010
Compatible with Autodesk 2010 applications, 2010 FBX plug-ins and Mot

Camera import script

Steps:

1. Create a new rule file by clicking **File** > **New** > **Python** > **Python Module**.

2. Choose the project's script folder, name it `importFBXCamera`, and choose the **Module: Main** template.

Parse the FBX file

Steps:

1. Parse lines and look for ID.
2. Prepare camera data in array.
   Nongeneric works for the specific .fbx file only.
3. Parse lines from the .fbx file that stores camera data.

```
def parseLine(lines, id):
    data = False
    for line in lines:
        if line.find(id) >=0 :
            data = line.partition(id)[2]
            break
    if data:
        data = data[:len(data)-1] # strip \n
        data = data.split(",")
    return data

def parseFbxCam(filename):
    f=open(filename)
    lines = f.readlines()
    cnt = 0
    loc =  parseLine(lines, 'Property: "Lcl Translation", "Lcl Translation", "A+",')
    rot =  parseLine(lines, 'Property: "Lcl Rotation", "Lcl Rotation", "A+",')
    return [loc,rot]
```

Set the CityEngine camera

Get the CityEngine viewport, and call the position and rotation set functions.

```
def setCamData(data):
    viewport = ce.getObjectsFrom(ce.get3DViews(), ce.isViewport)[0]
    setCamPosV(viewport, data[0])
    setCamRotV(viewport, data[1])
def setCamPosV(v, vec):
    v.setCameraPosition(vec[0], vec[1], vec[2])

def setCamRotV(v, vec):
    v.setCameraRotation(vec[0], vec[1], vec[2])
```

Master function

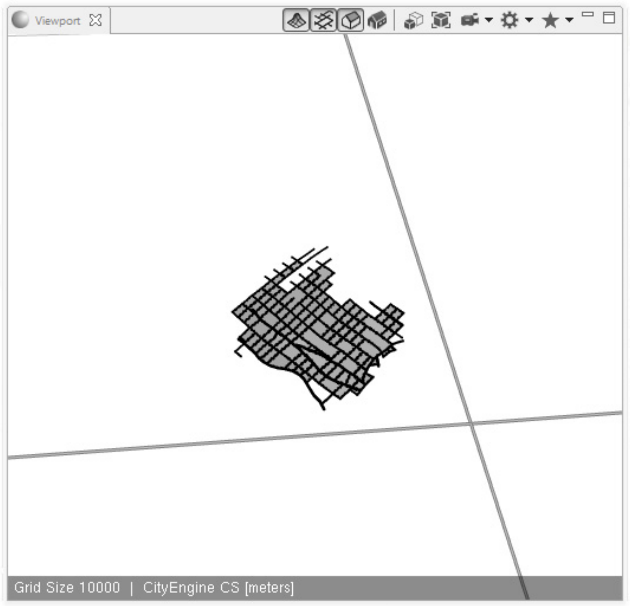
```
def importFbxCamera(fbxfile):

    data = parseFbxCam(fbxfile)
    if(data[0] and data[1]) :
        setCamData(data)
        print "Camera set to "+str(data)
    else:
        print "No camera data found in file "+file
```

Call in the main block

```
if __name__ == '__main__':
    camfile = ce.toFSPath("data/camera.fbx")
    importFbxCamera(camfile)
```

Run the Python script by clicking **Python** > **Run Script**, or press F9 while in the Python editor.

Your camera should be positioned as in the following screen capture:



Note: Animation curves are not read; only the transformation camera at the frame of exporting is read.
The camera needs to be exported as a single object.

Animation: Grow the building

Python scripts can be used to automate generation or export processes. The following example shows how to generate a building animation by setting the building attributes and exporting the set of resulting models.

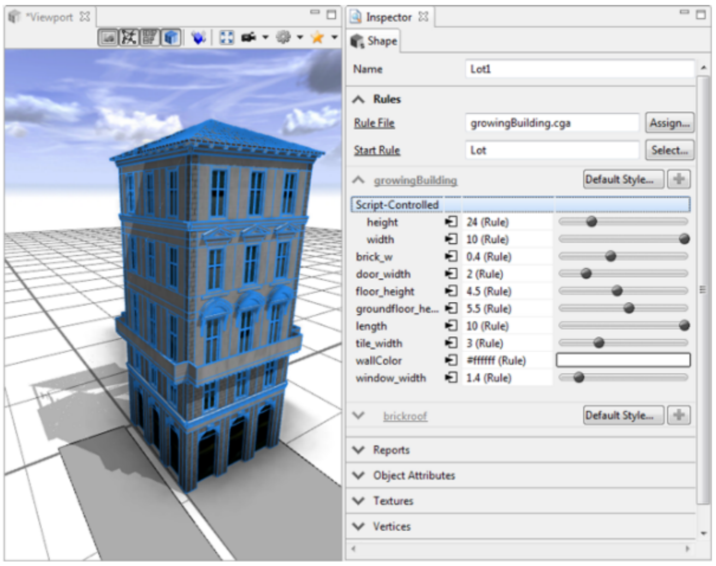
Tutorial setup

Open the `Tutorial_10_Python_Scripting/scenes/03_PythonScripting.cej` scene.

Generate the building

Steps:

1. Select a lot in the scene.
2. Assign the `growingBuilding.cga` rule file to the lot.
3. To generate the building, click **Shapes** > **Generate Models**.



Name	Lot1
Rule File	growingBuilding.cga
Start Rule	Lot

growingBuilding

Script-Controlled

height	24 (Rule)
width	10 (Rule)
brick_w	0.4 (Rule)
door_width	2 (Rule)
floor_height	4.5 (Rule)
groundfloor_he...	5.5 (Rule)
length	10 (Rule)
tile_width	3 (Rule)
wallColor	#ffffff (Rule)
window_width	1.4 (Rule)

The rule file includes attributes to change the dimensions of the building. Rather than manually setting these values, write a script that changes the values and batch generates the different versions of the model.

Animation script

Create a new Python main module `my_grow_building.py`.

def growBuilding

This function provides a time line that loops over two ranges and calls the setAttribute function.

```
def growBuilding():
    for i in range(1,14):
        height = 20+i
        doStep(i,height,1)

    for i in range(15,35):
        height = 34
        width = i-14
        doStep(i,height,width)
```

def doStep

On the lot object, the two attributes' height and width are modified.

```
def doStep(i,height,width):
    object = ce.getObjectsFrom(ce.scene, ce.withName("'Lot1'"))
    ce.setAttributeSource(object, "height", "OBJECT")
    ce.setAttributeSource(object, "width", "OBJECT")
    ce.setAttribute(object, "height", height)
    ce.setAttribute(object, "width", width)

    Generate(object)
```

def Generate

The following generates the building:

```
def Generate(object):
    ce.generateModels(object)
```

main

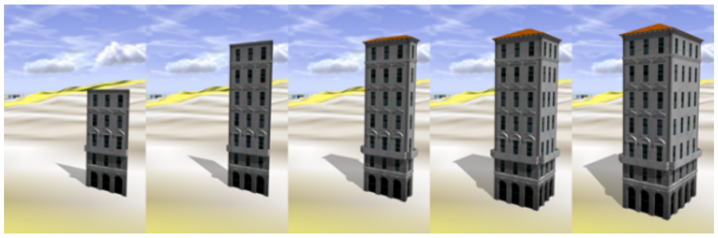
growBuilding is called in the main clause of the script.

```
if __name__ == '__main__':
   growBuilding()
```

Batch generate the building

Steps:

1. Select the lot in the scene.
2. Press F9 in the Python editor to run the script.



Batch export

Once you're confident with the generated models, add an additional function named `Export`.

```
def Export(i, object):
    dir = ce.toFSPath("models")
    file = "building_merge_" + str(i)
    // prepare export settings
    settings = OBJExportModelSettings()
    settings.setBaseName(file)
    settings.setOutputPath(dir)
    //do export
    ce.export(object, settings)
```

Replace the Generate call in `doStep()`.

```
//Generate(object)
        Export(i, object)
```

Find the exported models in the models folder. Make sure the Export function is before the main clause.

Write an asset library rule file

If you have a large number of assets, it might be helpful to look at all of them. This section shows how a CGA rule file can be generated automatically, which displays the project's assets.

Tutorial setup

Open the `Tutorial_10_Python_Scripting/scenes/03_PythonScripting.cej` scene.

The rule file you're going to write should have the following structure:

```
Lot -->  Geometries Textures

Geometries -->
         Geometry(assetpath)
         Geometry(assetpath)
         ...

Geometry(asset) --> i(asset)
```

This is for the geometry assets and the texture images.

- Create a new Python main module `my_asset_lib.py`.
- Add the new function `writeCGALib`.

```
def writeCGAlib():
```

Write header information, the starting rule Lot, and the `Geometries` rule.

```
...
    cga = "/*Asset Library Loader : Generated by asset_lib.py*/\n version \"2011.1\"\n\n"

    // write start rule
    cga += "Lot -->  Geometries Textures"

    // write rule showing geometries
    cga += "\n\nGeometries --> "
```

Iterate over all .obj files in the asset folder, and prepare the rule call `Geometry(assetpath)` for each asset.

```
...
    // get all .obj files from asset directory, and call their loader
    for obj in ce.getObjectsFrom("/", ce.isFile, ce.withName("/Tutorial_10*/assets/*.obj")):
        // and write
        cga += "\n\t t(2,0,0)  Geometry(\""+obj+"\")"
```

Similar rules are written for the texture assets.

```
...
    // write rule showing jpg textures
    cga+="\n\nTextures-->\n\ts(1,0,0) set(scope.ty,-2) set(scope.tz,0) i(\"facades/xy-plane.obj\")"

    // get all .jpg files from asset directory, and call their loader
    for jpg in ce.getObjectsFrom("/", ce.isFile, ce.withName("/Tutorial_10*/assets/*.jpg")):
        cga += "\n\tt(2,0,0)  Texture(\""+jpg+"\")"
```

Write the asset loader rules.

```
...
    //write geometry loader rule
    cga += "\n\n Geometry(asset) --> s(1,0,0) i(asset) set(scope.ty,0) set(scope.tz,0)"

    //write texture loader rule
    cga += "\n\n Texture(asset) --> set(material.colormap, asset)"
```

Open a file handle for the .cga file, and write the cga content.

```
...
    cgafile = ce.toFSPath("rules/asset_lib.cga")
    CGA = open(cgafile, "w")
    CGA.write(cga)
    CGA.close()
    print "written file "+cgafile
```

Add the new `assignAndGenerateLib()` function. It assigns the generated .cga file to a scene lot and generates the model.

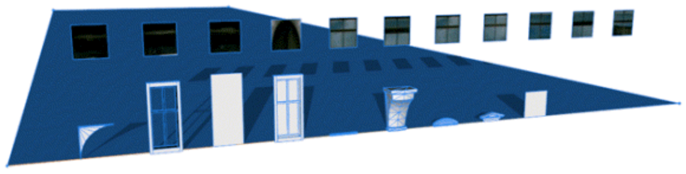
```
def assignAndGenerateLib():
    object = ce.getObjectsFrom(ce.scene, ce.withName("'Lot2'"))
    ce.refreshWorkspace()
    ce.setRuleFile(object, "asset_lib.cga")
    ce.setStartRule(object, "Lot")
    ce.generateModels(object)
```

Finally, call the two functions in the main clause:

```
if __name__ == '__main__':
    writeCGAlib()
    assignAndGenerateLib()
```

Generate the library model

In the Python editor, with the asset_lib.py file open, press F9.



Automate CityEngine tasks with startup.py

You can use python to automate larger or repetitive tasks. For example, you may want to automate model generation from parcel information across the county. To do this ,you would do the following:

Steps:

1. Create a project or use an existing one. We will use the existing `Tutorial_10_Python_Scripting__2018_0` tutorial project.
2. Create a Python main module in the project's scripts folder. The tutorial contains a minimal job called `automationJob.py`.
3. Insert a function with the tasks for the automated job. For testing, add a call to this function in the `'__main__'` section. The provided `fgdbToKml` example imports shapes from a fileGDB to generate models and writes them out to KML:

automationJob.py

```
from scripting import *
// get a CityEngine instance
ce = CE()

def fgdbToKml(pathFGDB,layerName,ruleName,startRule = "Generate"):
    // open scene in the automation project
    ce.newFile('/scenes/emptyScene.cej')

    // load a database
    importSettings = FGDBImportSettings()
    importSettings.setDatasetFilter(['/'+layerName])
    ce.importFile(ce.toFSPath(pathFGDB), importSettings)

    // assign rule file based on the layer name
    layer = ce.getObjectsFrom(ce.scene, ce.isShapeLayer, ce.withName(layerName))[0]
    shapes = ce.getObjectsFrom(layer, ce.isShape)
    ce.setRuleFile(shapes, ruleName)
    ce.setStartRule(shapes, startRule)

    // export models to KML
    exportSettings = KMLExportModelSettings()
    exportSettings.setOutputPath(ce.toFSPath("models"))
    exportSettings.setBaseName(layerName)
    exportSettings.setCompression(True)
    ce.export(shapes, exportSettings)

    // close CityEngine
    ce.waitForUIIdle()
    ce.closeFile()

if __name__ == '__main__':
    fgdbToKml("data/CityData.gdb", "NewShapes", "/ESRI.lib/rules/Buildings/Building_From_Footprint.cga", "Generate")
    pass
```

4. Create a configuration file to define the job parameters. The tutorial contains an example located in `\data\jobConfig.cfg`.
   jobConfig.cfg

```
[config]
pathFGDB=data/CityData.gdb
layerName=NewShapes
ruleName=/ESRI.lib/rules/Buildings/Building_From_Footprint.cga
startRule=Generate
```

5. Add the functions `run(cfg)` and `getCfgValue(cfg,name)` to `automationJob.py` in order to run the automation job with parameters stored int the configuration file.
   automationJob.py

```
...
def getCfgValue(cfg,name):
    for c in cfg:
        if  c[0] == name: return c[1]
    return None

def run(cfg):
    pathFGDB = getCfgValue(cfg,'pathfgdb')
    layerName = getCfgValue(cfg,'layername')
    ruleName = getCfgValue(cfg,'rulename')
    startRule = getCfgValue(cfg,'startrule')

    fgdbToKml(pathFGDB, layerName, ruleName, startRule)
```

6. It is recommended to use a separate CityEngine workspace for automation. Create a new `C:\Automation Workspace` folder on your system.
7. Copy the `/scripts/startup.py` file from the tutorial project to the new `C:\Automation Workspace` root directory. Commands in the `'__startup__'` section of this Python script get automatically executed at startup of CityEngine. The first startup argument defines the CityEngine project containing the automation job. It will be linked into the automation workspace. The second argument contains the `config` file. It gets parsed and handed over to the automation job as a list of (name,value) pairs.After the job has finished, CityEngine get's safely shut down.

startup.py

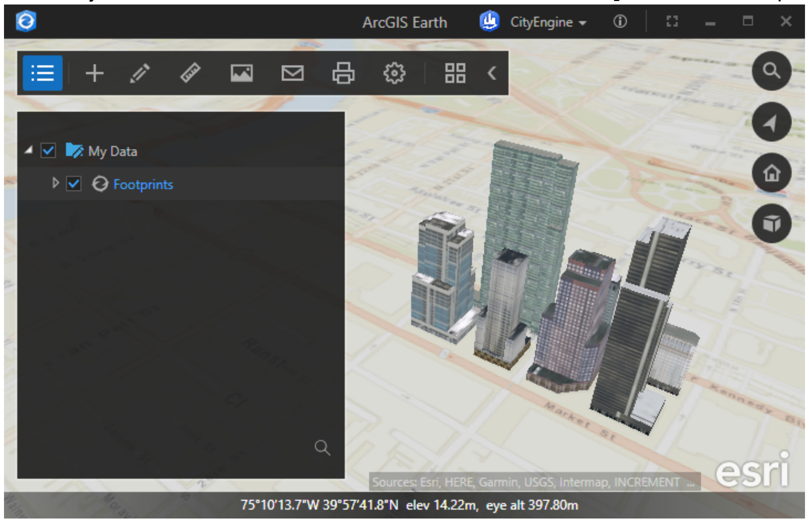
```
from scripting import *
from java import lang
import ConfigParser, sys

if __name__ == '__startup__':
   // get a CityEngine instance
    ce = CE()

    // get startup arguments
    projectFolder = lang.System.getProperty("projectFolder")
    configFilePath = lang.System.getProperty("configFilePath")

    // link the automation project into automation workspace
    if "automationProject" in ce.listProjects(): ce.removeProject("automationProject")
    ce.importProject(projectFolder, False, "automationProject")

    // read configuration file
    cp = ConfigParser.ConfigParser()
    cp.read(configFilePath)
    cfg = cp.items('config') # list of (name,value) pairs

    // run automation job
    sys.path.append(ce.toFSPath("/automationProject/scripts"))
    import automationJob
    automationJob.run(cfg)

    // safely shut down CityEngine
    ce.exit()
```

8. Open the command line and start CityEngine in the automation workspace and hand over the job definition and parameters:

   Command

```
<Path_to_CityEngine.exe> -data <Workspace_Folder> -vmargs -DprojectFolder=<Project_Folder> -DconfigFilePath=<Configuration_FilePath>

// Example:
> "C:\Program Files\Esri\CityEngine2018.0\CityEngine.exe" -data "C:\Automation Workspace" -vmargs -DprojectFolder="C:\CE_Workspace\Tutorial_10_Pyt
```

9. After the job has finished, the `\models` folder contains the `Footprints.kmz` output file shown below in ArcGIS Earth.


ArcGIS Earth
CityEngine
My Data
Footprints
Sources: Esri, HERE, Garmin, USGS, Intermap, INCREMENT
esri
75°10'13.7"W 39°57'41.8"N elev 14.22m, eye alt 397.80m

Note: The `automationJob.py` file contains a minimal job, use the CityEngine Python Help to tailor it to your exact needs.