# This is the heart of the Honeybee

#

# Honeybee: A Plugin for Environmental Analysis (GPL) started by Mostapha Sadeghipour Roudsari

#

# This file is part of Honeybee.

#

# Copyright (c) 2013-2020, Mostapha Sadeghipour Roudsari <mostapha@ladybug.tools>

# Honeybee is free software; you can redistribute it and/or modify

# it under the terms of the GNU General Public License as published

# by the Free Software Foundation; either version 3 of the License,

# or (at your option) any later version.

#

# Honeybee is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied warranty of

# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

# GNU General Public License for more details.

#

# You should have received a copy of the GNU General Public License

# along with Honeybee; If not, see <http://www.gnu.org/licenses/>.

#

# @license GPL-3.0+ <http://spdx.org/licenses/GPL-3.0+>

"""

This component carries all of Honeybee's main classes. Other components refer to these

classes to run the studies. Therefore, you need to let her fly before running the studies so the

classes will be copied to Rhinos shared space. So let her fly!

-

Honeybee: A Plugin for Environmental Analysis (GPL) started by Mostapha Sadeghipour Roudsari

You should have received a copy of the GNU General Public License

along with Honeybee; If not, see <http://www.gnu.org/licenses/>.

@license GPL-3.0+ <http://spdx.org/licenses/GPL-3.0+>

Source code is available at: https://github.com/mostaphaRoudsari/Honeybee

-

Provided by Honeybee 0.0.66

Args:

defaultFolder\_: Optional input for Honeybee default folder.

If empty default folder will be set to C:\ladybug or C:\Users\%USERNAME%\AppData\Roaming\Ladybug\

Returns:

report: Current Honeybee mood!!!

"""

ghenv.Component.Name = "Honeybee\_Honeybee"

ghenv.Component.NickName = 'Honeybee'

ghenv.Component.Message = 'VER 0.0.66\nJUL\_07\_2020'

ghenv.Component.IconDisplayMode = ghenv.Component.IconDisplayMode.icon

ghenv.Component.Category = "HB-Legacy"

ghenv.Component.SubCategory = "00 | Honeybee"

**try**: ghenv.Component.AdditionalHelpFromDocStrings = "1"

**except**: pass

**import** sys

sys.path = **sorted**(sys.path, key=**lambda** p: p.**find**("Python27"))

**import** rhinoscriptsyntax **as** rs

**import** Rhino **as** rc

**import** scriptcontext **as** sc

**import** Grasshopper

**import** Grasshopper.Kernel **as** gh

**from** Grasshopper **import** DataTree

**from** Grasshopper.Kernel.Data **import** GH\_Path

**import** math

**import** shutil

**import** os

**import** System.Threading.Tasks **as** tasks

**import** System

**import** time

**import** itertools

**import** datetime

**import** json

**import** copy

**import** urllib2 **as** urllib

**import** cPickle **as** pickle

**import** subprocess

**import** uuid

**import** re

**import** random

**import** zipfile

**try**:

System.Net.ServicePointManager.SecurityProtocol = System.Net.SecurityProtocolType.Tls12

**except** AttributeError:

# TLS 1.2 not provided by MacOS .NET Core; revert to using TLS 1.0

System.Net.ServicePointManager.SecurityProtocol = System.Net.SecurityProtocolType.Tls

PI = math.pi

rc.Runtime.HostUtils.**DisplayOleAlerts**(**False**)

tolerance = sc.doc.ModelAbsoluteTolerance

**class CheckIn**():

**def \_\_init\_\_**(self, defaultFolder, folderIsSetByUser = **False**):

self.folderIsSetByUser = folderIsSetByUser

self.letItFly = **True**

**if** defaultFolder:

# user is setting up the folder

defaultFolder = os.path.**normpath**(defaultFolder) + os.sep

# check if path has white space

**if** (" " **in** defaultFolder):

msg = "Default file path can't have white space. Please set the path to another folder." + \

"\nHoneybee failed to fly! :("

**print** msg

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

sc.sticky["Honeybee\_DefaultFolder"] = ""

self.letItFly = **False**

return

**else**:

# create the folder if it is not created

**if not** os.path.**isdir**(defaultFolder):

**try**: os.**mkdir**(defaultFolder)

**except**:

msg = "Cannot create default folder! Try a different filepath" + \

"\nHoneybee failed to fly! :("

**print** msg

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

sc.sticky["Honeybee\_DefaultFolder"] = ""

self.letItFly = **False**

return

# looks fine so let's set it up

sc.sticky["Honeybee\_DefaultFolder"] = defaultFolder

self.folderIsSetByUser = **True**

#set up default pass

**if not** self.folderIsSetByUser:

**if** os.path.**exists**("c:\\ladybug\\") **and** os.**access**(os.path.**dirname**("c:\\ladybug\\"), os.F\_OK):

# folder already exists so it is all fine

sc.sticky["Honeybee\_DefaultFolder"] = "c:\\ladybug\\"

**elif** os.**access**(os.path.**dirname**("c:\\"), os.F\_OK):

#the folder does not exists but write privileges are given so it is fine

sc.sticky["Honeybee\_DefaultFolder"] = "c:\\ladybug\\"

**else**:

# let's use the user folder

appdata = os.**getenv**("APPDATA")

# make sure username doesn't have space

**if** (" " **in** appdata):

msg = "User name on this system: " + appdata + " has white space." + \

" Default fodelr cannot be set.\nUse defaultFolder\_ to set the path to another folder and try again!" + \

"\nHoneybee failed to fly! :("

**print** msg

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

sc.sticky["Honeybee\_DefaultFolder"] = ""

self.letItFly = **False**

return

sc.sticky["Honeybee\_DefaultFolder"] = os.path.**join**(appdata, "Ladybug\\")

self.**updateCategoryIcon**()

@staticmethod

**def updateCategoryIcon**():

**try**:

url = "https://raw.githubusercontent.com/mostaphaRoudsari/Honeybee/master/resources/icon\_16\_16.png"

icon = os.path.**join**(sc.sticky["Honeybee\_DefaultFolder"], "HB\_icon\_16\_16.png")

**if not** os.path.**isfile**(icon):

client = System.Net.**WebClient**()

client.**DownloadFile**(url, icon)

iconBitmap = System.Drawing.**Bitmap**(icon)

Grasshopper.Instances.ComponentServer.**AddCategoryIcon**("Honeybee", iconBitmap)

**except**:

# download failed

pass

Grasshopper.Instances.ComponentServer.**AddCategoryShortName**("Honeybee", "HB")

Grasshopper.Instances.ComponentServer.**AddCategorySymbolName**("Honeybee", "H")

Grasshopper.Kernel.GH\_ComponentServer.**UpdateRibbonUI**() #Reload the Ribbon

**def getComponentVersion**(self):

monthDict = {'JAN':'01', 'FEB':'02', 'MAR':'03', 'APR':'04', 'MAY':'05', 'JUN':'06',

'JUL':'07', 'AUG':'08', 'SEP':'09', 'OCT':'10', 'NOV':'11', 'DEC':'12'}

# convert component version to standard versioning

ver, verDate = ghenv.Component.Message.**split**("\n")

ver = ver.**split**(" ")[1].**strip**()

month, day, year = verDate.**split**("\_")

month = monthDict[month.**upper**()]

version = ".".**join**([year, month, day, ver])

return version

**def isNewerVersionAvailable**(self, currentVersion, availableVersion):

# print int(availableVersion.replace(".", "")), int(currentVersion.replace(".", ""))

return **int**(availableVersion.**replace**(".", "")) > **int**(currentVersion.**replace**(".", ""))

**def checkForUpdates**(self, LB= **True**, HB= **True**, OpenStudio = **True**, template = **True**, therm = **True**):

url = "https://github.com/mostaphaRoudsari/ladybug/raw/master/resources/versions.txt"

versionFile = os.path.**join**(sc.sticky["Honeybee\_DefaultFolder"], "versions.txt")

client = System.Net.**WebClient**()

client.**DownloadFile**(url, versionFile)

with **open**("c:/ladybug/versions.txt", "r")**as** vf:

versions= **eval**("\n".**join**(vf.**readlines**()))

honeybeeDefaultFolder = sc.sticky["Honeybee\_DefaultFolder"]

**if** LB:

ladybugVersion = versions['Ladybug']

currentLadybugVersion = self.**getComponentVersion**() # I assume that this function will be called inside Ladybug\_ladybug Component

**if** self.**isNewerVersionAvailable**(currentLadybugVersion, ladybugVersion):

msg = "There is a newer version of Ladybug available to download! " + \

"We strongly recommend you to download the newer version from Food4Rhino: " + \

"http://www.food4rhino.com/project/ladybug-honeybee"

**print** msg

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

**if** HB:

honeybeeVersion = versions['Honeybee']

currentHoneybeeVersion = self.**getComponentVersion**() # I assume that this function will be called inside Honeybee\_Honeybee Component

**if** self.**isNewerVersionAvailable**(currentHoneybeeVersion, honeybeeVersion):

msg = "There is a newer version of Honeybee available to download! " + \

"We strongly recommend you to download the newer version from Food4Rhino: " + \

"http://www.food4rhino.com/project/ladybug-honeybee"

**print** msg

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

**if** OpenStudio:

# This should be called inside OpenStudio component which means Honeybee is already flying

# check if the version file exist

openStudioLibFolder = os.path.**join**(sc.sticky["Honeybee\_DefaultFolder"], "OpenStudio")

versionFile = os.path.**join**(openStudioLibFolder, "osversion.txt")

isNewerOSAvailable= **False**

**if not** os.path.**isfile**(versionFile):

isNewerOSAvailable= **True**

**else**:

# read the file

with **open**(versionFile) **as** verFile:

currentOSVersion= **eval**(verFile.**read**())['version']

OSVersion = versions['OpenStudio']

**if** isNewerOSAvailable **or** self.**isNewerVersionAvailable**(currentOSVersion, OSVersion):

sc.sticky["isNewerOSAvailable"] = **True**

**else**:

sc.sticky["isNewerOSAvailable"] = **False**

**if** therm:

thermFile = os.path.**join**(honeybeeDefaultFolder, 'thermMaterial.csv')

# check file doesn't exist then it should be downloaded

isNewerThermAvailable = **False**

**if not** os.path.**isfile**(thermFile):

isNewerThermAvailable = **True**

**else**:

# find the version

**try**:

with **open**(thermFile) **as** tempFile:

currentThermVersion = **eval**(tempFile.**readline**().**split**("!")[-1].**strip**())["version"]

**except**: isNewerThermAvailable = **True**

# finally if the file exist and already has a version, compare the versions

thermVersion = versions['THERM']

**if** isNewerThermAvailable **or** self.**isNewerVersionAvailable**(currentThermVersion, thermVersion):

sc.sticky["isNewerTHERMAvailable"] = **True**

**else**:

sc.sticky["isNewerTHERMAvailable"] = **False**

**if** template:

templateFile = os.path.**join**(honeybeeDefaultFolder, 'OpenStudioMasterTemplate.idf')

# check file doesn't exist then it should be downloaded

**if not** os.path.**isfile**(templateFile):

return **True**

# find the version

**try**:

with **open**(templateFile) **as** tempFile:

currentTemplateVersion = **eval**(tempFile.**readline**().**split**("!")[-1].**strip**())["version"]

**except** Exception, e:

return **True**

# finally if the file exist and already has a version, compare the versions

templateVersion = versions['Template']

return self.**isNewerVersionAvailable**(currentTemplateVersion, templateVersion)

**class versionCheck**(object):

**def \_\_init\_\_**(self):

self.version = self.**getVersion**(ghenv.Component.Message)

**def getVersion**(self, LBComponentMessage):

monthDict = {'JAN':'01', 'FEB':'02', 'MAR':'03', 'APR':'04', 'MAY':'05', 'JUN':'06',

'JUL':'07', 'AUG':'08', 'SEP':'09', 'OCT':'10', 'NOV':'11', 'DEC':'12'}

# convert component version to standard versioning

**try**: ver, verDate = LBComponentMessage.**split**("\n")

**except**: ver, verDate = LBComponentMessage.**split**("\\n")

ver = ver.**split**(" ")[1].**strip**()

month, day, year = verDate.**split**("\_")

month = monthDict[month.**upper**()]

version = ".".**join**([year, month, day, ver])

return version

**def isCurrentVersionNewer**(self, desiredVersion):

return **int**(self.version.**replace**(".", "")) >= **int**(desiredVersion.**replace**(".", ""))

**def isCompatible**(self, LBComponent):

code = LBComponent.Code

# find the version that is supposed to be flying

**try**:

version = code.**split**("compatibleHBVersion")[1].**split**("=")[1].**split**("\n")[0].**strip**()

**except** Exception, e:

**print** e

self.**giveWarning**(LBComponent)

return **False**

desiredVersion = self.**getVersion**(version)

**if not** self.**isCurrentVersionNewer**(desiredVersion):

self.**giveWarning**(LBComponent)

return **False**

return **True**

**def giveWarning**(self, GHComponent):

warningMsg = "You need a newer version of Honeybee to use this compoent." + \

"Use updateHoneybee component to update userObjects.\n" + \

"If you have already updated userObjects drag Honeybee\_Honeybee component " + \

"into canvas and try again."

w = gh.GH\_RuntimeMessageLevel.Warning

GHComponent.**AddRuntimeMessage**(w, warningMsg)

**def isInputMissing**(self, GHComponent):

isInputMissing = **False**

**for** param **in** GHComponent.Params.Input:

**if** param.NickName.**startswith**("\_") **and** \

**not** param.NickName.**endswith**("\_") **and** \

**not** param.VolatileDataCount:

warning = "Input parameter %s failed to collect data!"%param.NickName

GHComponent.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warning)

isInputMissing = **True**

return isInputMissing

**class hb\_findFolders**():

**def \_\_init\_\_**(self):

self.RADPath, self.RADFile = self.**which**('rad.exe')

self.EPPath, self.EPFile = self.**which**('EnergyPlus.exe')

self.DSPath, self.DSFile = self.**which**('gen\_dc.exe')

self.THERMPath, self.THERMFile = self.**which**('Therm7.exe')

**def which**(self, program):

"""

Check for path. Modified from this link:

http://stackoverflow.com/questions/377017/test-if-executable-exists-in-python

"""

**def is\_exe**(fpath):

# Avoid Radiance and Daysim that comes with DIVA as it has a different

# structure which doesn't match the standard Daysim

**if** fpath.**upper**().**find**("DIVA")<0:

# if the user has DIVA installed the component may find DIVA version

# of RADIANCE and DAYISM which can cause issues because of the different

# structure of folders in DIVA

return os.path.**isfile**(fpath) **and** os.**access**(fpath, os.F\_OK)

**else**:

return **False**

fpath, fname = os.path.**split**(program)

**if** fpath:

**if is\_exe**(program):

return program

**else**:

**for** path **in** os.environ["PATH"].**split**(os.pathsep):

path = path.**strip**('"')

exe\_file = os.path.**join**(path, program)

**if is\_exe**(exe\_file):

# This is a change to catch cases that user has radiance inastalled

# at C:\Program Files\Radiance

**if** path.**strip**().**find**(" ") == -1:

return path, exe\_file

return **None**, **None**

**class PrepareTemplateEPLibFiles**(object):

"""

Download Template files and check for available libraries for EnergyPlus

"""

**def \_\_init\_\_**(self, downloadTemplate = **False**, workingDir = **None**):

**if not** workingDir: workingDir = sc.sticky["Honeybee\_DefaultFolder"]

**if not** sc.sticky.**has\_key**("honeybee\_constructionLib"): sc.sticky ["honeybee\_constructionLib"] = {}

**if not** sc.sticky.**has\_key**("honeybee\_materialLib"): sc.sticky ["honeybee\_materialLib"] = {}

**if not** sc.sticky.**has\_key**("honeybee\_windowMaterialLib"): sc.sticky ["honeybee\_windowMaterialLib"] = {}

**if not** sc.sticky.**has\_key**("honeybee\_ScheduleLib"): sc.sticky["honeybee\_ScheduleLib"] = {}

**if not** sc.sticky.**has\_key**("honeybee\_ScheduleTypeLimitsLib"): sc.sticky["honeybee\_ScheduleTypeLimitsLib"] = {}

**if not** sc.sticky.**has\_key**("honeybee\_WindowPropLib"): sc.sticky["honeybee\_WindowPropLib"] = {}

**if not** sc.sticky.**has\_key**("honeybee\_SpectralDataLib"): sc.sticky["honeybee\_SpectralDataLib"] = {}

**if not** sc.sticky.**has\_key**("honeybee\_thermMaterialLib"): sc.sticky["honeybee\_thermMaterialLib"] = {}

self.downloadTemplate = downloadTemplate

self.workingDir = workingDir

self.failureMsg = ""

**def downloadFile**(self, url, workingDir):

localFilePath = workingDir + '/' + url.**split**('/')[-1]

client = System.Net.**WebClient**()

client.**DownloadFile**(url, localFilePath)

**def cleanHBLib**(self):

sc.sticky ["honeybee\_constructionLib"] = {}

sc.sticky ["honeybee\_materialLib"] = {}

sc.sticky ["honeybee\_windowMaterialLib"] = {}

sc.sticky["honeybee\_ScheduleLib"] = {}

sc.sticky["honeybee\_ScheduleTypeLimitsLib"] = {}

sc.sticky["honeybee\_WindowPropLib"] = {}

sc.sticky["honeybee\_SpectralDataLib"] = {}

**def cleanThermLib**(self):

sc.sticky["honeybee\_thermMaterialLib"] = {}

**def downloadTemplates**(self):

workingDir = self.workingDir

# create the folder if it is not there

**if not** os.path.**isdir**(workingDir): os.**mkdir**(workingDir)

# create a backup from the user's library

templateFile = os.path.**join**(workingDir, 'OpenStudioMasterTemplate.idf')

bckupfile = os.path.**join**(workingDir, 'OpenStudioMasterTemplate\_' + **str**(**int**(time.**time**())) +'.idf')

thermTemplateFile = os.path.**join**(workingDir, 'thermMaterial.csv')

thermBckupfile = os.path.**join**(workingDir, 'thermMaterial\_' + **str**(**int**(time.**time**())) +'.csv')

# download EP template file

**if** self.downloadTemplate **or not** os.path.**isfile**(templateFile):

# create a backup from users library

**try**: shutil.**copyfile**(templateFile, bckupfile)

**except**: pass

**try**:

## download File

**print** 'Downloading OpenStudioMasterTemplate.idf to ', workingDir

updatedLink = "https://github.com/mostaphaRoudsari/Honeybee/raw/master/resources/OpenStudioMasterTemplate.idf"

self.**downloadFile**(updatedLink, workingDir)

# clean current library

self.**cleanHBLib**()

**except**:

**print** 'Download failed!!! You need OpenStudioMasterTemplate.idf to use honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

**else**:

pass

**if not** os.path.**isfile**(workingDir + '\OpenStudioMasterTemplate.idf'):

iplibPath = ghenv.Script.**GetStandardLibPath**()

**print** 'Download failed!!! You need OpenStudioMasterTemplate.idf to use honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

**else**:

libFilePaths = [os.path.**join**(workingDir, 'OpenStudioMasterTemplate.idf')]

# download openstudio standards

**if not** os.path.**isfile**(workingDir + '\OpenStudio\_Standards.json'):

**try**:

## download File

**print** 'Downloading OpenStudio\_Standards.json to ', workingDir

self.**downloadFile**(r'https://github.com/mostaphaRoudsari/Honeybee/raw/master/resources/OpenStudio\_Standards.json', workingDir)

**except**:

**print** 'Download failed!!! You need OpenStudio\_Standards.json to use honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

**else**:

pass

**if not** os.path.**isfile**(workingDir + '\OpenStudio\_Standards.json'):

**print** 'Download failed!!! You need OpenStudio\_Standards.json to use honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

**else**:

# load the json file

filepath = os.path.**join**(workingDir, 'OpenStudio\_Standards.json')

**try**:

with **open**(filepath) **as** jsondata:

openStudioStandardLib = json.**load**(jsondata)

sc.sticky ["honeybee\_OpenStudioStandardsFile"] = openStudioStandardLib

**print** "Standard template file is loaded from %s"%filepath

**except**:

**print** 'Download failed!!! You need OpenStudio\_Standards.json to use honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

# add custom library

customEPLib = os.path.**join**(workingDir,"userCustomEPLibrary.idf")

**if not** os.path.**isfile**(customEPLib):

# create an empty file

with **open**(customEPLib, "w") **as** outf:

outf.**write**("!Honeybee custom EnergyPlus library\n")

**if** os.path.**isfile**(customEPLib):

libFilePaths.**append**(customEPLib)

#download THERM template file.

**if** sc.sticky.**has\_key**("isNewerTHERMAvailable") **and** sc.sticky["isNewerTHERMAvailable"] **or not** os.path.**isfile**(thermTemplateFile):

# create a backup from users library

**try**: shutil.**copyfile**(thermTemplateFile, thermBckupfile)

**except**: pass

**try**:

## download File

**print** 'Downloading thermMaterial.csv to ', workingDir

updatedLink = "https://raw.githubusercontent.com/mostaphaRoudsari/Honeybee/master/resources/thermMaterial.csv"

self.**downloadFile**(updatedLink, workingDir)

# clean current library

self.**cleanThermLib**()

**except**:

**print** 'Download failed!!! You need thermMaterial.csv to use the "export to THERM" capabilties of honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

**if not** os.path.**isfile**(thermTemplateFile):

**print** 'Download failed!!! You need thermMaterial.csv to use the "export to THERM" capabilties of honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

**else**:

# load the csv file

csvfilepath = os.path.**join**(workingDir, 'thermMaterial.csv')

**try**:

libFilePaths.**append**(csvfilepath)

**except**:

**print** 'Download failed!!! You need thermMaterial.csv to use the "export to THERM" capabilties of honeybee.' + \

'\nPlease check your internet connection, and try again!'

return -1

return libFilePaths

**class** HB\_GetEPLibraries:

**def \_\_init\_\_**(self):

self.libraries = {

"Material": {},

"WindowMaterial": {},

"Construction": {},

"Schedule" : {},

"ScheduleTypeLimits": {},

"ThermMaterial": {},

"WindowProperty": {},

"MaterialProperty": {}

}

**def getEPMaterials**(self):

return self.libraries["Material"]

**def getEPConstructions**(self):

return self.libraries["Construction"]

**def getEPWindowMaterial**(self):

return self.libraries["WindowMaterial"]

**def getEPWindowProp**(self):

return self.libraries["WindowProperty"]

**def getEPSpectralData**(self):

return self.libraries["MaterialProperty"]

**def getEPSchedule**(self):

return self.libraries["Schedule"]

**def getEPScheduleTypeLimits**(self):

return self.libraries["ScheduleTypeLimits"]

**def getTHERMMaterials**(self):

return self.libraries["ThermMaterial"]

**def importEPLibrariesFromFile**(self, EPfile, isMatFile, cleanCurrentLib = **True**, report = **True**):

**if not** os.path.**isfile**(EPfile):

**raise Exception**("Can't find EP library! at %s"%EPfile)

**if** isMatFile == **False**:

**print** "Loading EP materials, constructions, schedules and material properties from %s"%EPfile

EPObjects = self.**getEnergyPlusObjectsFromFile**(EPfile)

self.**loadEPConstructionsMaterialsAndSchedules**(EPObjects, cleanCurrentLib)

**else**:

**print** "Loading THERM materials from %s"%EPfile

self.**getThermObjectsFromFile**(EPfile)

**if** report:

self.**report**()

**def cleanHBLibs**(self):

self.libraries = {

"Material": {},

"WindowMaterial": {},

"Construction": {},

"Schedule" : {},

"ScheduleTypeLimits": {},

"ThermMaterial": {},

"WindowProperty": {},

"MaterialProperty": {}

}

# TODO: Support parsing for files with no next line

# TODO: Check if keys can be case insensitive

# TODO: Create EPObjects and not dictionaries

**def loadEPConstructionsMaterialsAndSchedules**(self, EPObjectsString, cleanCurrentLib = **True**):

**if** cleanCurrentLib: self.**cleanHBLibs**()

**for** EPObjectStr **in** EPObjectsString:

rawLines = EPObjectStr.**strip**().**split**("\n")

lines = []

**for** line **in** rawLines:

**if** line.**strip**() == '' **or** line.**startswith**('!'): continue

lines.**append**(line)

**if not** lines:

continue

**if** lines[0].**startswith**('MaterialProperty:GlazingSpectralData'):

key = 'MaterialProperty:GlazingSpectralData'

shortKey = 'MaterialProperty'

name = lines[1].**split**(",")[0].**strip**().**upper**()

self.libraries[shortKey][name] = **dict**() # create an empty dictonary

self.libraries[shortKey][name][0] = key

# store the data into the dictionary

**for** lineCount, line **in enumerate**(lines):

objValue = line.**split**("!")[0].**strip**()

**try**: objDescription = line.**split**("!")[1].**strip**()

**except**: objDescription = ""

**if** lineCount == 0:

self.libraries[shortKey][name][lineCount] = objValue[:-1]

**elif** lineCount == 1:

pass # name is already there as the key

**elif** objValue.**endswith**(","):

self.libraries[shortKey][name][lineCount-1] = objValue[:-1], objDescription

**elif** objValue.**endswith**(";"):

self.libraries[shortKey][name][lineCount-1] = objValue[:-1], objDescription

**else**:

**if len**(lines) < 2: continue

**if** lines[0].**split**(",")[0].**strip**().**isupper**():

key = lines[0].**split**(",")[0].**strip**().**title**()

**else**:

key = lines[0].**split**(",")[0].**strip**()

shortKey = key.**split**(":")[0]

name = lines[1].**split**(",")[0].**strip**().**upper**()

values = lines[2:]

# it's a two line object such as Any Number scheduleTypeLimit

**if** values == []:

name = lines[1].**split**(";")[0].**strip**().**upper**() # name is the last input

**if** shortKey **in** self.libraries:

self.libraries[shortKey][name] = **dict**() # create an empty dictonary

self.libraries[shortKey][name][0] = key

count = 1

delimiter = ","

**for** value **in** values:

**if not len**(value.**strip**()): continue #pass empty lines

**if** count==**len**(values): delimiter = ";"

v = value.**split**(delimiter)[0].**strip**() # find the value

**if** value.**find**("!")!= -1:

c = value.**split**("!")[-1].**rstrip**() # find the value

**else**:

c = ""

self.libraries[shortKey][name][count] = v, c

count += 1

**def report**(self):

# Report findings

**print** "%s EPConstructions are loaded available in Honeybee library"%**str**(**len**(self.libraries["Construction"]))

**print** "%s EPMaterials are now loaded in Honeybee library"%**str**(**len**(self.libraries["Material"]))

**print** "%s EPWindowMaterial are loaded in Honeybee library"%**str**(**len**(self.libraries["WindowMaterial"]))

**print** "%s EPShadingControl are loaded in Honeybee library"%**str**(**len**(self.libraries["WindowProperty"]))

**print** "%s EPMaterialProperty are loaded in Honeybee library"%**str**(**len**(self.libraries["MaterialProperty"]))

**print** "%s schedules are loaded available in Honeybee library"%**str**(**len**(self.libraries["Schedule"]))

**print** "%s schedule type limits are now loaded in Honeybee library"%**str**(**len**(self.libraries["ScheduleTypeLimits"]))

**print** "%s THERM materials are now loaded in Honeybee library"%**str**(**len**(self.libraries["ThermMaterial"]))

**print** "\n"

@staticmethod

**def getEnergyPlusObjectsFromString**(epFileString):

"""

Parse idf file string and return a list of EnergyPlus objects as separate strings

TODO: Create a class for each EnergyPlus object and return Python objects

instead of strings

Args:

epFileString: EnergyPlus data as a single string. The string can be multiline

Returns:

A list of strings. Each string represents a differnt Rdiance Object

"""

#rawEPObjects = re.findall(r'(.[^;]\*;.[^\n]\*)', epFileString + "\n",re.MULTILINE)

rawEPObjects = re.**findall**(r'(.[^;]\*;)', epFileString + "\n",re.MULTILINE)

return rawEPObjects

**def getEnergyPlusObjectsFromFile**(self, epFilePath):

"""

Parse EnergyPlus file and return a list of objects as separate strings

TODO: Create a class for each EnergyPlus object and return Python objects

instead of strings

Args:

epFilePath: Path to EnergyPlus file

Returns:

A list of strings. Each string represents a differnt Rdiance Object

Usage:

getEnergyPlusObjectsFromFile(r"C:\ladybug\21MAR900\imageBasedSimulation\21MAR900.rad")

"""

**if not** os.path.**isfile**(epFilePath):

**raise ValueError**("Can't find %s."%epFilePath)

with **open**(epFilePath, "r") **as** epFile:

return self.**getEnergyPlusObjectsFromString**("".**join**(epFile.**readlines**()))

**def getThermObjectsFromFile**(self, matFile):

**if not** os.path.**isfile**(matFile):

**raise ValueError**("Can't find %s."%matFile)

with **open**(matFile, "r") **as** mFile:

**for** rowCount, row **in enumerate**(mFile):

**if** rowCount > 1:

**try**:

matPropLine = row.**split**(',')

matNameLine = row.**split**('"')

matName = matNameLine[1].**upper**()

#Make a sub-dictionary for the material.

self.libraries["ThermMaterial"][matName] = {}

#Create the material with the values from the file.

self.libraries["ThermMaterial"][matName]["Name"] = matName

self.libraries["ThermMaterial"][matName]["Type"] = **int**(matPropLine[-1])

self.libraries["ThermMaterial"][matName]["Conductivity"] = **float**(matPropLine[-5])

self.libraries["ThermMaterial"][matName]["Absorptivity"] = **float**(matPropLine[-4])

**if** self.libraries["ThermMaterial"][matName]["Type"] == 0:

self.libraries["ThermMaterial"][matName]["Tir"] = "0.0"

**else**:

self.libraries["ThermMaterial"][matName]["Tir"] = "-1.0"

self.libraries["ThermMaterial"][matName]["Emissivity"] = **float**(matPropLine[-3])

self.libraries["ThermMaterial"][matName]["WindowDB"] = ""

self.libraries["ThermMaterial"][matName]["WindowID"] = "-1"

self.libraries["ThermMaterial"][matName]["RGBColor"] = System.Drawing.ColorTranslator.**FromHtml**("#" + matPropLine[-2])

**except**: pass

**def checkUnits**():

units = sc.doc.ModelUnitSystem

**if** `units` == 'Rhino.UnitSystem.Meters': conversionFactor = 1.00

**elif** `units` == 'Rhino.UnitSystem.Centimeters': conversionFactor = 0.01

**elif** `units` == 'Rhino.UnitSystem.Millimeters': conversionFactor = 0.001

**elif** `units` == 'Rhino.UnitSystem.Feet': conversionFactor = 0.305

**elif** `units` == 'Rhino.UnitSystem.Inches': conversionFactor = 0.0254

**else**:

**print** 'Kidding me! Which units are you using?'+ `units`+'?'

**print** 'Please use Meters, Centimeters, Millimeters, Inches or Feet'

return

**print** 'Current document units is in', sc.doc.ModelUnitSystem

**print** 'Conversion to Meters will be applied = ' + "%.3f"%conversionFactor

return conversionFactor

**class RADMaterialAux**(object):

**class** RadianceMaterial:

"""

Radiance Material

Attributes:

name: Material name as a string

type: Material type (e.g. glass, plastic, etc)

modifier: Material modifier. Default is void

values: A dictionary of material data. key is line number and item is the list of values

{0: [], 1: [], 2: ['0.500', '0.500', '0.500', '0.000', '0.050']}

"""

**def \_\_init\_\_**(self, name, type, values = **None**, modifier = "void"):

self.name = name.**rstrip**()

self.type = type.**rstrip**()

self.modifier = modifier.**rstrip**()

**if not** values: values = **dict**()

self.values = values

**def toRadString**(self):

firstLine = "%s %s %s"**%**(self.modifier, self.type, self.name)

material = [firstLine]

# order is important and that's why I'm using range

# and not the keys itself

**for** lineCount **in range**(**len**(self.values.**keys**())):

values = self.values[lineCount]

count = [**str**(**len**(values))]

line = " ".**join**(count + values).**rstrip**()

material.**append**(line)

material.**append**("\n")

return "\n".**join**(material)

**def addValues**(self, lineCount, values):

"""Add values to current material

Args:

lineCount: An integer that represnt the line number

values: Values as a list of string

"""

self.values[lineCount] = values

**def \_\_repr\_\_**(self):

return self.**toRadString**()

**def \_\_init\_\_**(self, reloadRADMaterial = **False**, materialLibrary = {}, HoneybeeFolder = "c:/ladybug"):

self.HoneybeeFolder = HoneybeeFolder

self.radMaterialLibrary = materialLibrary

self.radMatTypes = ["plastic", "glass", "trans", "metal",

"mirror", "texfunc", "mixedfunc", "dielectric", "transdata",

"light", "glow", "BRTDfunc"]

**if** reloadRADMaterial:

defaultMaterial = {

'Context\_Material' : {'type' : 'plastic', 'value': 0.35},

'Interior\_Ceiling' : {'type' : 'plastic', 'value': 0.80},

'Interior\_Floor' : {'type' : 'plastic', 'value': 0.20},

'Exterior\_Floor' : {'type' : 'plastic', 'value': 0.20},

'Exterior\_Roof' : {'type' : 'plastic', 'value': 0.80},

'Exterior\_Wall' : {'type' : 'plastic', 'value': 0.50},

'Interior\_Wall' : {'type' : 'plastic', 'value': 0.50},

'Interior\_Window' : {'type' : 'glass' , 'value': 0.60},

'Exterior\_Window' : {'type' : 'glass' , 'value': 0.60}

}

**for** materialName, materialData **in** defaultMaterial.**items**():

radMaterial = self.**RadianceMaterial**(materialName, materialData['type'])

value = materialData['value']

# add values to material

# first two lines are empty

radMaterial.**addValues**(0, [])

radMaterial.**addValues**(1, [])

**if** radMaterial.type == 'glass':

value = self.**getTransmissivity**(value)

radMaterial.**addValues**(2, 3 \* ['%.3f'%value]) # leave roughness specularity to 0

**else**:

radMaterial.**addValues**(2, 3 \* ['%.3f'%value] + ['0', '0']) # leave roughness specularity to 0

# add default materials to the library

self.**addMaterialToDocumentLibrary**(radMaterial)

# import user defined RAD library

RADLibraryFile = self.**getUserDefinedRadianceLibraryPath**()

**if** os.path.**isfile**(RADLibraryFile):

self.**importRADMaterialsFromFile**(RADLibraryFile)

**else**:

# This is only happening the first time

# that user lets the Honeybee fly on their system

# or changes the default folder

**if not** os.path.**isdir**(self.HoneybeeFolder):

os.**mkdir**(self.HoneybeeFolder)

with **open**(RADLibraryFile, "w") **as** outf:

outf.**write**("#Honeybee Radiance Material Library\n")

**print** "Loading RAD default materials..." + \

`**len**(self.radMaterialLibrary)` + " RAD materials are loaded\n"

**def duplicateMaterialWarning**(self, materialName, newMaterialString):

returnYN = {'YES': **True**, 'NO': **False**}

buttons = System.Windows.Forms.MessageBoxButtons.YesNo

icon = System.Windows.Forms.MessageBoxIcon.Warning

**try**:

currentMaterialString = self.**getRADMaterialString**(materialName)

**except**:

currentMaterialString = materialName

isAdded, materialName = self.**analyseRadMaterials**(materialName, **False**)

msg = materialName + " already exists in the library:\n\n" + \

currentMaterialString + "\n" + \

"Do you want to overwrite the current material with this new definition?\n\n" + \

newMaterialString + "\n\n" + \

"Tip: If you are not sure what to do select No and change the material name."

up = System.Windows.Forms.MessageBox.**Show**(msg, "Duplicate Material Name", buttons, icon)

return returnYN[up.**ToString**().**ToUpper**()]

# TODO: Rewite! This method is poorly written and is very hard to understand

# TODO: Should be probably moved to writeRAD. Here is not the right place

**def addRADMatToDocumentDict**(self, HBSrf, currentMatDict, currentMixedFunctionsDict):

"""Collect Radiance materials for a single run"""

# check if the material is already added

materialName = HBSrf.RadMaterial

**if not** materialName **in** currentMatDict.**keys**():

# find material type

materialType = self.**getRADMaterialType**(materialName)

materialModifier = self.**getRADMaterialModifier**(materialName)

# check if this is a mixed function

**if** materialType == "mixfunc":

# add mixedFunction

currentMixedFunctionsDict[materialName] = materialName

# find the base materials for the mixed function

mixfunMaterial = self.**getMaterialFromHBLibrary**(materialName)

material1 = mixfunMaterial.values[0][0]

material2 = mixfunMaterial.values[0][1]

**for** matName **in** [material1, material2]:

**if not** matName **in** currentMatDict.**keys**():

currentMatDict[matName] = matName

**elif** materialModifier != "void":

# add material itself

currentMixedFunctionsDict[materialName] = materialName

# check if modifier is in library and add it to dictionary

**if not** self.**isMatrialExistInLibrary**(materialModifier):

**raise Exception**("You're using %s as a modifier which is not added to the library!")%materialModifier

**if not** materialModifier **in** currentMatDict.**keys**():

currentMatDict[materialModifier] = materialModifier

**else**:

# add to dictionary

currentMatDict[materialName] = materialName

return currentMatDict, currentMixedFunctionsDict

**def getUserDefinedRadianceLibraryPath**(self):

return os.path.**join**(self.HoneybeeFolder, "HoneybeeRadMaterials.mat")

@staticmethod

**def getTransmissivity**(transmittance):

return (math.**sqrt**(0.8402528435 + 0.0072522239 **\*** (transmittance \*\* 2)) - 0.9166530661 ) / 0.0036261119 / transmittance

**def analyseRadMaterials**(self, radMaterialString, addToDocLib = **False**, overwrite = **True**):

"""Analyse Radiance Material string

Import a RAD material string, create a Honeybee RadianceMaterial

and add it to Honeybee library if needed.

Always return the a boolean and Radiance name

"""

**try**:

# get radince material as a single line

cleanedRadMaterialString = self.**cleanRadMaterial**(radMaterialString)

lineSegments = cleanedRadMaterialString.**split**(" ")

**if len**(lineSegments) == 1:

# this is just the name

# to be used for applying material to surfaces

return **False**, lineSegments[0].**rstrip**()

**else**:

materialModifier = lineSegments[0]

materialType = lineSegments[1]

materialName = lineSegments[2].**rstrip**()

# initiate Rad material

radMaterial = self.**RadianceMaterial**(materialName, materialType, modifier = materialModifier)

**if** addToDocLib:

**if** self.**isMatrialExistInLibrary**(materialName) **and not** overwrite:

# ask for user input before overwriting the material

upload = self.**duplicateMaterialWarning**(materialName, radMaterialString)

**if not** upload:

return **False**, materialName

counters = []

materialProp = lineSegments[3:]

#first counter is the first member of the list

counter = 0

counters.**append**(0)

**while** counter < **len**(materialProp):

counter += **int**(materialProp[counter]) + 1

**try**:

counters.**append**(counter)

**except**:

pass

**for** counter, count **in enumerate**(counters[1:]):

values = materialProp[counters[counter] + 1: count]

# add values to material

radMaterial.**addValues**(counter, values)

# add material to library

self.**addMaterialToDocumentLibrary**(radMaterial)

return **True**, materialName

**else**:

return **False**, materialName

**except**:

**raise Exception**("Faild to import %s"%radMaterialString)

**def addMaterialToDocumentLibrary**(self, radMaterial, overwrite = **True**):

"""Add Radiance material to current Grasshopper document library

Args:

radMaterial: A RadianceMaterial object

"""

# check if material already exists

**if** self.**isMatrialExistInLibrary**(radMaterial.name) **and not** overwrite:

# ask for user input before overwriting the material

upload = self.**duplicateMaterialWarning**(radMaterial.name, radMaterialString)

**if not** upload: return

# add to library

self.radMaterialLibrary[radMaterial.name] = radMaterial

**def isMatrialExistInLibrary**(self, materialName):

return materialName **in** self.radMaterialLibrary

**def cleanRadMaterial**(self, radMaterialString):

"""

inputs rad material string, remove comments, spaces, etc and returns

a single line string everything separated by a single space

"""

matStr = ""

lines = radMaterialString.**strip**().**split**("\n")

**for** line **in** lines:

**if not** line.**strip**().**startswith**("#"):

**if not len**(line.**rstrip**()): continue

line = line.**replace**("\t", " ")

lineSeg = line.**split**(" ")

**for** seg **in** lineSeg:

**if** seg.**strip**()!="":

matStr += seg + " "

return matStr[:-1] # remove the last space

**def createRadMaterialFromString**(self, radMaterialString):

"""Clean string and return a Radiance Material"""

cleanedRadMaterialString = self.**cleanRadMaterial**(radMaterialString)

lineSegments = cleanedRadMaterialString.**split**(" ")

materialModifier = lineSegments[0]

materialType = lineSegments[1]

materialName = lineSegments[2].**rstrip**()

# initiate Rad material

radMaterial = self.**RadianceMaterial**(materialName, materialType, modifier = materialModifier)

counters = []

materialProp = lineSegments[3:]

#first counter is the first member of the list

counter = 0

counters.**append**(0)

**while** counter < **len**(materialProp):

counter += **int**(materialProp[counter]) + 1

**try**:

counters.**append**(counter)

**except**:

pass

**for** counter, count **in enumerate**(counters[1:]):

values = materialProp[counters[counter] + 1: count]

# add values to material

radMaterial.**addValues**(counter, values)

return radMaterial

**def getRADMaterialString**(self, materialName):

"""Return radiance material string"""

material = self.**getMaterialFromHBLibrary**(materialName)

**if** material: return material.**toRadString**()

**def getMaterialFromHBLibrary**(self, materialName):

**try**:

return self.radMaterialLibrary[materialName]

**except**:

**if** materialName.**lower**() != 'void':

**raise ValueError**("%s can't be find in library"%**str**(materialName))

**else**:

return

**def getRADMaterialType**(self, materialName):

"""Return material type"""

material = self.**getMaterialFromHBLibrary**(materialName)

**if** material: return material.type

**def getRADMaterialModifier**(self, materialName):

"""Return material type"""

material = self.**getMaterialFromHBLibrary**(materialName)

**if** material: return material.modifier

**def getRADMaterialParameters**(self, materialName):

"""Return radiance material string"""

material = self.**getMaterialFromHBLibrary**(materialName)

**if** material:

lastLine = **sorted**(material.values)[-1]

return material.values[lastLine]

**def getSTForTransMaterials**(self, materialName):

"""Retuen st value for Trans materials"""

properties = self.**getRADMaterialParameters**(materialName)

properties = **map**(float, properties)

# check got translucant materials

PHAverage = 0.265 \* properties[0] + 0.670 \* properties[1] + 0.065 \* properties[2]

st = properties[5] \* properties[6] **\*** (1 - PHAverage \* properties[3])

return st

@staticmethod

**def getRadianceObjectsFromString**(radFileString):

"""

Parse rad file string and return a list of radiance objects as separate strings

Args:

radFileString: Radiance data as a single string. The string can be multiline

Returns:

A list of strings. Each string represents a differnt Rdiance Object

"""

raw\_rad\_objects = re.**findall**(

r'^\s\*([^0-9].\*(\s\*[\d|.]+.\*)\*)',

radFileString,

re.MULTILINE)

radObjects **=** (' '.**join**(radiance\_object[0].**split**())

**for** radiance\_object **in** raw\_rad\_objects)

radObjects = **tuple**(obj **for** obj **in** radObjects **if** obj **and** obj[0] != '#')

return radObjects

**def getRadianceObjectsFromFile**(self, radFilePath):

"""

Parse Radinace file and return a list of radiance objects as separate strings

TODO: Create a class for each Radiance object and return Python objects

instead of strings

Args:

radFilePath: Path to Radiance file

Returns:

A list of strings. Each string represents a differnt Rdiance Object

Usage:

getRadianceObjectsFromFile(r"C:\ladybug\21MAR900\imageBasedSimulation\21MAR900.rad")

"""

**if not** os.path.**isfile**(radFilePath):

**raise ValueError**("Can't find %s."%radFilePath)

with **open**(radFilePath, "r") **as** radFile:

return self.**getRadianceObjectsFromString**("".**join**(radFile.**readlines**()))

**def importRADMaterialsFromFile**(self, radFilePath, overwrite = **True**):

"""

Parse Radinace file and add them to Radiance Libraryreturn a list of radiance objects as separate strings

Args:

radFilePath: Path to a radiance file

"""

radianceObjects = self.**getRadianceObjectsFromFile**(radFilePath)

**for** materialString **in** radianceObjects:

**try**:

# try to import the string

self.**analyseRadMaterials**(materialString, **True**, overwrite)

**except**:

**raise Exception**("Faild to import %s"%materialString)

**def createDictionaryFromRADObjects**(self, radObjects):

"""Return Rad objects in a dictionary where each key is the name"""

result = **dict**()

**for** obj **in** radObjects:

name = self.**cleanRadMaterial**(obj).**split**(" ")[2]

result[name] = obj

return result

**def searchRadMaterials**(self, keywords, materialTypes):

keywords = [kw.**strip**().**upper**() **for** kw **in** keywords]

materialTypes = [mt.**strip**().**upper**() **for** mt **in** materialTypes]

materials = []

**for** radMaterial **in** self.radMaterialLibrary:

materialName = radMaterial.**upper**()

materialType = self.**getMaterialFromHBLibrary**(radMaterial).type.**upper**()

**if len**(materialTypes)==0 **or** materialType **in** materialTypes:

**if len**(keywords)!= 0 **and not** "\*" **in** keywords:

**for** keyword **in** keywords:

**if** materialName.**find**(keyword)!= -1 **or** keyword.**find**(materialName)!= -1:

materials.**append**(radMaterial)

**else**:

materials.**append**(radMaterial)

return materials

**def addToGlobalLibrary**(self, RADMaterialString):

"""Add a Radiance materil string to global library

Honeybee global library is a text file which is located under

Honeybee's default folder.

"""

RADLibraryFile = self.**getUserDefinedRadianceLibraryPath**()

# analyze string, add to local library and get the material name

added, materialName = self.**analyseRadMaterials**(RADMaterialString, **False**)

# read all the existing materials from the file

radObjects = self.**getRadianceObjectsFromFile**(RADLibraryFile)

objectsDict = self.**createDictionaryFromRADObjects**(radObjects)

# Check if material is not there append to the file

**if not** materialName **in** objectsDict:

# add to local library

added, materialName = self.**analyseRadMaterials**(RADMaterialString, **True**)

# get the material object

radMaterial = self.**getMaterialFromHBLibrary**(materialName)

with **open**(RADLibraryFile, 'a') **as** outf:

outf.**writelines**("\n" + radMaterial.**toRadString**() + "\n")

**print** "%s is added to global library."%materialName

return **True**

**else**:

# Material is already existed

# give a warning to user and ask for overwrite

# add to local library

added, materialName = self.**analyseRadMaterials**(RADMaterialString, **True**, **False**)

**if** added:

# replace the old material with the new one

objectsDict[materialName] = RADMaterialString

# write the file

with **open**(RADLibraryFile, 'w') **as** outf:

outf.**write**("#Honeybee Radiance Material Library\n")

**for** rawRADMaterialString **in** objectsDict.**values**():

radianceMaterial = self.**createRadMaterialFromString**(rawRADMaterialString)

outf.**writelines**(radianceMaterial.**toRadString**())

outf.**write**("\n")

**print** "%s is added to global library."%materialName

return **True**

return **False**

**def assignRADMaterial**(self, HBSurface, RADMaterial, component):

# 1.4 assign RAD Material

**if** RADMaterial!=**None**:

# if it is just the name of the material make sure it is already defined

**if len**(RADMaterial.**split**(" ")) == 1:

# if the material is not in the library add it to the library

**if** RADMaterial **not in** sc.sticky ["honeybee\_RADMaterialLib"].**keys**():

warningMsg = "Can't find " + RADMaterial + " in RAD Material Library.\n" + \

"Add the material to the library and try again."

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warningMsg)

return

**try**:

HBSurface.**setRADMaterial**(RADMaterial)

**print** "HBSurface Radiance Material has been set to " + RADMaterial

**except** Exception, e:

**print** e

warningMsg = "Failed to assign RADMaterial to " + HBSurface.name

**print** warningMsg

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warningMsg)

return

addedToLib = **True**

**else**:

# try to add the material to the library

addedToLib, HBSurface.RadMaterial = self.**analyseRadMaterials**(RADMaterial, **True**)

**if** addedToLib==**False**:

warningMsg = "Failed to add " + RADMaterial + " to the Library."

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warningMsg)

return

**class** DLAnalysisRecipe:

**def \_\_init\_\_**(self, type, \*arg):

"""

types:

0: image based analysis > Illuminance(lux) = 0, Radiation(kwh) = 1, Luminance (cd) = 2

1: node based analysis

2: annual simulation (Daysim for now)

3: daylight factor

4: vertical sky component

"""

self.type = type

self.component = arg[-1]

# based on the type it should return different outputs

**if** type == 0:

self.skyFile = arg[0]

self.viewNames = arg[1]

**try**: self.radParameters = arg[2].d

**except**: self.radParameters = arg[2]

self.cameraType = arg[3]

self.simulationType = arg[4]

self.imageSize = arg[5], arg[6]

self.sectionPlane = arg[7]

self.backupImages = arg[8]

self.studyFolder = "\\imageBasedSimulation\\"

**elif** type == 1:

self.skyFile = arg[0]

self.testPts = self.**convertTreeToLists**(arg[1])

self.vectors = self.**convertTreeToLists**(arg[2])

**try**: self.radParameters = arg[3].d

**except**: self.radParameters = arg[3]

self.simulationType = arg[4]

self.testMesh = self.**convertTreeToLists**(arg[5])

self.studyFolder = "\\gridBasedSimulation\\"

**elif** type == 2:

self.weatherFile = arg[0]

self.testPts = self.**convertTreeToLists**(arg[1])

self.vectors = self.**convertTreeToLists**(arg[2])

**try**: self.radParameters = arg[3].d

**except**: self.radParameters = arg[3]

self.DSParameters = arg[4]

self.testMesh = self.**convertTreeToLists**(arg[5])

self.northDegrees = arg[6]

self.studyFolder = "\\annualSimulation\\"

**elif** type == 3:

self.skyFile = arg[0]

self.testPts = self.**convertTreeToLists**(arg[1])

self.vectors = self.**convertTreeToLists**(arg[2])

**try**: self.radParameters = arg[3].d

**except**: self.radParameters = arg[3]

self.simulationType = 0 #illuminance

self.testMesh = self.**convertTreeToLists**(arg[4])

self.studyFolder = "\\DF\\"

**elif** type == 4:

self.skyFile = arg[0]

self.testPts = self.**convertTreeToLists**(arg[1])

self.vectors = self.**convertTreeToLists**(arg[2])

**try**: self.radParameters = arg[3].d

**except**: self.radParameters = arg[3]

self.testMesh = self.**convertTreeToLists**(arg[4])

self.simulationType = 0 #illuminance

self.studyFolder = "\\VSC\\"

# double check the sky in case of grid based and image based simulations

**if** type ==0 **or** type == 1:

self.**checkSky**()

**def convertTreeToLists**(self, l):

listOfLists = []

**for** path **in** l.Paths:

listOfLists.**append**(l.**Branch**(path))

return listOfLists

**def checkSky**(self):

**if** self.simulationType == 1:

# make sure the sky is either gencum or gendaylit

# edit in case of gendaylit

self.radSkyFile = '.'.**join**(self.skyFile.**split**("."))[:-1] + "\_radAnalysis.sky"

skyOut = **open**(self.radSkyFile, "w")

genDaylit = **False**

with **open**(self.skyFile, "r") **as** skyIn:

**for** line **in** skyIn:

**if** line.**startswith**("!gensky"):

self.skyFile = **None**

msg = "You need to use one of the climate-based skies for radiation analysis.\n" + \

"Change the skyFile and try again"

self.component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return

**elif** line.**startswith**("!gendaylit"):

line = line.**replace**("-O 0", "-O 1")

genDaylit = **True**

# write a new file

skyOut.**write**(line)

skyOut.**close**()

self.skyFile = self.radSkyFile

**if not** genDaylit:

self.simulationType = 1.1 # annual radiation analysis

**else**:

# make sure the sky is not from gencum

with **open**(self.skyFile, "r") **as** skyIn:

**for** line **in** skyIn:

**if** line.**strip**().**startswith**("2 skybright") **and** line.**strip**().**endswith**(".cal"):

self.skyFile = **None**

msg = "Cumulative sky can only be used for radiation analysis.\n" + \

"Change the skyFile and try again"

self.component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return

**def \_\_repr\_\_**(self):

return "Honybee.Recipe.%s"%self.studyFolder.**replace**("\\", "")

**class hb\_MSHToRAD**(object):

**def \_\_init\_\_**(self, mesh, fileName = **None**, workingDir = **None**, bitmap = **None**, radMaterial = **None**):

**if** fileName == **None**:

fileName = "unnamed"

self.name = fileName

**if** workingDir == **None**:

workingDir = sc.sticky["Honeybee\_DefaultFolder"]

workingDir = os.path.**join**(workingDir, fileName, "MSH2RADFiles")

**if not** os.path.**isdir**(workingDir): os.**mkdir**(workingDir)

self.workingDir = workingDir

self.mesh = mesh

self.RadianceFolder = sc.sticky["honeybee\_folders"]["RADPath"]

self.pattern = bitmap

**if** self.pattern != **None**:

# create material name based on bitmap

bitmapFileName = os.path.**basename**(self.pattern)

self.matName = ".".**join**(bitmapFileName.**split**(".")[:-1])

#copy the image into same folder

**try**:

shutil.**copyfile**(self.pattern, os.path.**join**(self.workingDir, bitmapFileName))

**except**:

pass

**else**:

self.matName = "radMaterial"

**if** radMaterial != **None**:

radMaterial = RADMaterialAux.**getRadianceObjectsFromString**(radMaterial)[0]

**try**:

self.matName = radMaterial.**strip**().**split**()[2]

**assert** self.matName != ""

**except**:

**raise Exception**("Failed to import %s. Double check the material definition."%radMaterial)

self.RADMaterial = " ".**join**(radMaterial.**split**())

**def meshToObj**(self):

objFilePath = os.path.**join**(self.workingDir, self.name + ".obj")

with **open**(objFilePath, "w") **as** outfile:

# objTxt = "# OBJ file written by TurtlePyMesh\n\n"

outfile.**write**("# OBJ file written by TurtlePyMesh\n\n")

# add material file name

mtlFile = self.name + ".mtl"

#objTxt += "mtllib " + mtlFile + "\n"

outfile.**write**("mtllib " + mtlFile + "\n")

**for** count, Tmesh **in enumerate**(self.mesh):

# add object name - for this version I keep it all as a single object

#objTxt += "o object\_" + str(count + 1) + "\n"

outfile.**write**("o object\_" + **str**(count + 1) + "\n")

# add material name - for now brick as test

#objTxt += "usemtl " + matName + "\n"

outfile.**write**("usemtl " + self.matName + "\n")

**if** Tmesh.Normals.Count == 0:

Tmesh.Normals.**ComputeNormals**()

# add vertices

**for** v **in** Tmesh.Vertices:

XYZ = v.X, v.Y, v.Z

XYZ = **map**(str, XYZ)

vString = " ".**join**(XYZ)

#objTxt += "v " + vString + "\n"

outfile.**write**("v " + vString + "\n")

# add texture vertices

**for** vt **in** Tmesh.TextureCoordinates:

XY = vt.X, vt.Y

XY = **map**(str, XY)

vtString = " ".**join**(XY)

#objTxt += "vt " + vtString + "\n"

outfile.**write**("vt " + vtString + "\n")

# add normals

**for** vn **in** Tmesh.Normals:

XYZ = vn.X, vn.Y, vn.Z

XYZ = **map**(str, XYZ)

vnString = " ".**join**(XYZ)

# objTxt += "vn " + vnString + "\n"

outfile.**write**("vn " + vnString + "\n")

# add faces

# vertices number is global so the number should be added together

fCounter = 0

**if** count > 0:

**for** meshCount **in range**(count):

fCounter += self.mesh[meshCount].Vertices.Count

# print fCounter

**if** self.pattern != **None**:

**for** face **in** Tmesh.Faces:

# objTxt += "f " + "/".join(3\*[`face.A + fCounter + 1`]) + " " + "/".join(3\*[`face.B + fCounter + 1`]) + " " + "/".join(3\*[`face.C + fCounter + 1`])

outfile.**write**("f " + "/".**join**(3\*[`face.A + fCounter + 1`]) + " " + "/".**join**(3\*[`face.B + fCounter + 1`]) + " " + "/".**join**(3\*[`face.C + fCounter + 1`]))

**if** face.IsQuad:

#objTxt += " " + "/".join(3\*[`face.D + fCounter + 1`])

outfile.**write**(" " + "/".**join**(3\*[`face.D + fCounter + 1`]))

#objTxt += "\n"

outfile.**write**("\n")

**else**:

**for** face **in** Tmesh.Faces:

outfile.**write**("f " + "//".**join**(2 \* [`face.A + fCounter + 1`]) + \

" " + "//".**join**(2 \* [`face.B + fCounter + 1`]) + \

" " + "//".**join**(2 \* [`face.C + fCounter + 1`]))

**if** face.IsQuad:

outfile.**write**(" " + "//".**join**( 2 \* [`face.D + fCounter + 1`]))

#objTxt += "\n"

outfile.**write**("\n")

# This method happened to be so slow!

# with open(objFile, "w") as outfile:

# outfile.writelines(objTxt)

return objFilePath

**def getPICImageSize**(self):

with **open**(self.pattern, "rb") **as** inf:

**for** count, line **in enumerate**(inf):

#print line

**if** line.**strip**().**startswith**("-Y") **and** line.**find**("-X"):

Y, YSize, X, XSize = line.**split**(" ")

return XSize, YSize

**def objToRAD**(self, objFile):

# prepare file names

radFile = objFile.**replace**(".obj", ".rad")

mshFile = objFile.**replace**(".obj", ".msh")

batFile = objFile.**replace**(".obj", ".bat")

path, fileName = os.path.**split**(radFile)

matFile = os.path.**join**(path, "material\_" + fileName)

**try**:

materialType = self.RADMaterial.**split**()[1]

materialTale = " ".**join**(self.RADMaterial.**split**()[3:])

**except** Exception, e:

# to be added here: if material is not full string then get it from the library

errmsg = "Failed to parse material:\n%s" % e

**print** errmsg

**raise ValueError**(errmsg)

# create material file

**if** self.pattern != **None**:

# find aspect ratio

**try**:

X, Y= self.**getPICImageSize**()

ar = **str**(**int**(X)/**int**(Y))

**except** Exception, e:

ar = **str**(1)

# mesh has a pattern

patternName = ".".**join**(os.path.**basename**(self.pattern).**split**(".")[:-1])

materialStr = "void colorpict " + patternName + "\_pattern\n" + \

"7 red green blue " + self.pattern + " . (" + ar + "\*(Lu-floor(Lu))) (Lv-floor(Lv)) \n" + \

"0\n" + \

"1 1\n" + \

patternName + "\_pattern " + materialType + " " + patternName + "\n" + \

materialTale

**else**:

materialStr = "void " + materialType + " " + self.matName + " " + \

materialTale

# write material to file

with **open**(matFile, "w") **as** outfile:

outfile.**write**(materialStr)

# create rad file

**if** self.pattern != **None**:

cmd = self.RadianceFolder + "\\obj2mesh -a " + matFile + " " + objFile + " > " + mshFile

with **open**(batFile, "w") **as** outfile:

outfile.**write**(cmd)

#outfile.write("\npause")

os.**system**(batFile)

radStr = "void mesh painting\n" + \

"1 " + mshFile + "\n" + \

"0\n" + \

"0\n"

with **open**(radFile, "w") **as** outfile:

outfile.**write**(radStr)

**else**:

# use object to rad

#create a fake mtl file - material will be overwritten by radiance material

mtlFile = objFile.**replace**(".obj", ".mtl")

mtlStr = "# Honeybee\n" + \

"newmtl " + self.matName + "\n" + \

"Ka 0.0000 0.0000 0.0000\n" + \

"Kd 1.0000 1.0000 1.0000\n" + \

"Ks 1.0000 1.0000 1.0000\n" + \

"Tf 0.0000 0.0000 0.0000\n" + \

"d 1.0000\n" + \

"Ns 0\n"

with **open**(mtlFile, "w") **as** mtlf:

mtlf.**write**(mtlStr)

# create a map file

#mapFile = objFile.replace(".obj", ".map")

#with open(mapFile, "w") as mapf:

# mapf.write(self.matName + " (Object \"" + self.matName + "\");")

#cmd = "c:\\radiance\\bin\\obj2rad -m " + mapFile + " " + objFile + " > " + radFile

cmd = self.RadianceFolder + "\\obj2rad -f " + objFile + " > " + radFile

with **open**(batFile, "w") **as** outfile:

outfile.**write**(cmd)

#outfile.write("\npause")

os.**system**(batFile)

time.**sleep**(.2)

return matFile, radFile

**class hb\_WriteRAD**(object):

**def \_\_init\_\_**(self, component = ghenv.Component):

self.component = component

self.hb\_writeRADAUX = sc.sticky["honeybee\_WriteRADAUX"**]**()

self.hb\_RADMaterialAUX = sc.sticky["honeybee\_RADMaterialAUX"]

self.lb\_preparation = sc.sticky["ladybug\_Preparation"**]**()

self.hb\_writeDS = sc.sticky["honeybee\_WriteDS"**]**()

self.hb\_radParDict = sc.sticky["honeybee\_RADParameters"**]**().radParDict

hb\_folders = sc.sticky["honeybee\_folders"]

self.hb\_RADPath = hb\_folders["RADPath"]

self.hb\_RADLibPath = hb\_folders["RADLibPath"]

self.hb\_DSPath = hb\_folders["DSPath"]

self.hb\_DSCore = hb\_folders["DSCorePath"]

self.hb\_DSLibPath = hb\_folders["DSLibPath"]

**def writeRADAndMaterialFiles**(self, originalHBObjects, subWorkingDir, radFileName, \

analysisRecipe, meshParameters, exportInteriorWalls):

# initiate RAD Parameters

**if** analysisRecipe.radParameters==**None**:

quality = 0

analysisRecipe.radParameters = {}

**print** "Default values are set for RAD parameters"

**for** key **in** self.hb\_radParDict.**keys**():

#print key + " is set to " + str(hb\_radParDict[key][quality])

analysisRecipe.radParameters[key] = self.hb\_radParDict[key][quality]

# collect information from analysis recipe

radParameters = analysisRecipe.radParameters

simulationType = analysisRecipe.type

radFileFullName = os.path.**join**(subWorkingDir, radFileName + '.rad')

IESObjects = {}

IESCount = 0

# call the objects from the lib

hb\_hive = sc.sticky["honeybee\_Hive"**]**()

HBObjects = hb\_hive.**callFromHoneybeeHive**(originalHBObjects)

geoRadFile = **open**(radFileFullName, 'w')

geoRadFile.**write**("#GENERATED BY HONEYBEE\n")

customRADMat = {} # dictionary to collect the custom material names

customMixFunRadMat = {} # dictionary to collect the custom mixfunc material names

surfaceList = []

rotateObjects = **False**

**if len**(HBObjects)!=0:

# if this is an annual analysis and north is not 0 rotate all Honeybee objects

**if** analysisRecipe.type == 2 **and** analysisRecipe.northDegrees!=0:

**print** "Rotating the scene for %d degrees"%analysisRecipe.northDegrees

transform = rc.Geometry.Transform.**Rotation**(-math.**radians**(analysisRecipe.northDegrees), \

rc.Geometry.Point3d.Origin)

rotateObjects = **True**

**for** objCount, HBObj **in enumerate**(HBObjects):

**if** rotateObjects:

HBObj.**transform**(transform, **None**, **False**)

# check if the object is zone or a surface (?)

**if** HBObj.objectType == "HBZone":

**if** HBObj.hasNonPlanarSrf **or** HBObj.hasInternalEdge:

HBObj.**prepareNonPlanarZone**(meshParameters)

**for** srf **in** HBObj.surfaces:

# check if an interior wall

**if not** exportInteriorWalls **and** self.hb\_writeRADAUX.**isSrfAirWall**(srf):

continue

# if it is an interior wall and the other wall is already written

# then don't write this wall

**if** self.hb\_writeRADAUX.**isSrfInterior**(srf) **and** srf.BCObject.name **in** surfaceList:

continue

surfaceList.**append**(srf.name)

# collect the custom material informations

**if** srf.RadMaterial!=**None**:

customRADMat, customMixFunRadMat = self.hb\_RADMaterialAUX.**addRADMatToDocumentDict**(srf, customRADMat, customMixFunRadMat)

# write the surfaces

**if** srf.isPlanar **and len**(srf.childSrfs)<2:

geoRadFile.**write**(self.**RADSurface**(srf))

**else**:

geoRadFile.**write**(self.**RADNonPlanarSurface**(srf))

**if** srf.hasChild:

# collect the custom material informations

**for** childSrf **in** srf.childSrfs:

**if** childSrf.RadMaterial!=**None**:

customRADMat, customMixFunRadMat = self.hb\_RADMaterialAUX.**addRADMatToDocumentDict**(childSrf, customRADMat, customMixFunRadMat)

**if not** srf.isPlanar **or len**(srf.childSrfs) > 1:

geoRadFile.**write**(self.**RADNonPlanarChildSurface**(srf))

**elif** HBObj.objectType == "HBSurface":

# I should wrap this in a function as I'm using it multiple times with minor changes

# collect the custom material informations

**if** HBObj.RadMaterial!=**None**:

**try**:

customRADMat, customMixFunRadMat = self.hb\_RADMaterialAUX.**addRADMatToDocumentDict**(HBObj, customRADMat, customMixFunRadMat)

**except**:

msg = HBObj.RadMaterial + " is not defined in the material library! Add the material to library and try again."

**print** msg

self.component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return -1

# check for material in child surfaces

**if not** HBObj.isChild **and** HBObj.hasChild:

# collect the custom material informations

**for** childSrf **in** HBObj.childSrfs:

**if** childSrf.RadMaterial!=**None**:

**try**:

customRADMat, customMixFunRadMat = self.hb\_RADMaterialAUX.**addRADMatToDocumentDict**(childSrf, customRADMat, customMixFunRadMat)

**except**:

msg = childSrf.RadMaterial + " is not defined in the material library! Add the material to library and try again."

**print** msg

self.component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return -1

**if** HBObj.isPlanar **and** (**not** HBObj.isChild **and len**(HBObj.childSrfs)<2):

# check for rad material

geoRadFile.**write**(self.**RADSurface**(HBObj))

**else**:

geoRadFile.**write**(self.**RADNonPlanarSurface**(HBObj))

**if not** HBObj.isChild **and** HBObj.hasChild:

geoRadFile.**write**(self.**RADNonPlanarChildSurface**(HBObj))

**elif** HBObj.objectType == "HBIES":

IESCount += 1

IESObjcIsFine = **True**

# check if the object has been move or scaled

**if** HBObj.**checkIfScaledOrRotated**(originalHBObjects[objCount]):

IESObjcIsFine = **False**

msg = "IES luminaire " + HBObj.name + " is scaled or rotated" + \

" and cannot be added to the scene."

**print** msg

self.component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

# check if the material name is already exist

**if** HBObj.name **in** customRADMat.**keys**():

IESObjcIsFine = **False**

msg = "IES luminaire " + HBObj.name + " cannot be added to the scene.\n" + \

"A material with the same name already exist."

**print** msg

self.component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

# if it is all fine then write the geometry

**if** IESObjcIsFine:

IESName = HBObj.name + "\_" + **str**(IESCount)

geoRadFile.**write**( HBObj.**getRADGeometryStr**(IESName, originalHBObjects[objCount]))

# downlight\_light polygon downlight.d

# add to IES Objects list so I can add the materials to the list later

**if** HBObj.name **not in** IESObjects.**keys**():

IESObjects[HBObj.name] = HBObj

geoRadFile.**close**()

########################################################################

######################## GENERATE THE BASE RAD FILE ####################

materialFileName = subWorkingDir + "\\material\_" + radFileName + '.rad'

# This part should be fully replaced with the new method where I generate the materials from the

# 0.1 material string

matStr = "# start of generic materials definition(s)\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Context\_Material') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Interior\_Ceiling') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Interior\_Floor') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Exterior\_Floor') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Exterior\_Window') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Interior\_Window') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Exterior\_Roof') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Exterior\_Wall') + "\n" + \

self.hb\_RADMaterialAUX.**getRADMaterialString**('Interior\_Wall') + "\n" + \

"# end of generic materials definition(s)\n"

with **open**(materialFileName, 'w') **as** matFile:

matFile.**write**(matStr)

matFile.**write**("\n# start of material(s) specific to this study (if any)\n")

**for** radMatName **in** customRADMat.**keys**():

**try**:

matFile.**write**(self.hb\_RADMaterialAUX.**getRADMaterialString**(radMatName) + "\n")

**except**:

# This is the case for void material

pass

# check if the material is is trans

**if** self.hb\_RADMaterialAUX.**getRADMaterialType**(radMatName) == "trans":

# get the st value

st = self.hb\_RADMaterialAUX.**getSTForTransMaterials**(radMatName)

**if** st < radParameters["\_st\_"]:

**print** "Found a trans material... " + \

"Resetting st parameter from " + **str**(radParameters["\_st\_"]) + " to " + **str**(st)

radParameters["\_st\_"] = st

# write mixedfun if any

**for** radMatName **in** customMixFunRadMat.**keys**():

matFile.**write**(self.hb\_RADMaterialAUX.**getRADMaterialString**(radMatName) + "\n")

# write IES material if any

**if len**(IESObjects.**keys**())!= 0:

**for** IESName **in** IESObjects.**keys**():

IESObj = IESObjects[IESName]

# write material file

matFile.**write**(IESObj.materialStr)

# add dat file to folder

datFileName = subWorkingDir + "\\" + IESName + '.dat'

with **open**(datFileName, "w") **as** outDat:

outDat.**write**(IESObj.datFile)

matFile.**write**("# end of material(s) specific to this study (if any)\n")

# export dayism shading geometries as radFiles

# this is only useful for dynamic shadings

dynamicCounter = 0

**if** simulationType == 2:

dynamicShadingRecipes = analysisRecipe.DSParameters.DShdR

**if len**(dynamicShadingRecipes) == 0:

return radFileFullName, materialFileName

customRADMat = {} # dictionary to collect the custom material names

customMixFunRadMat = {} # dictionary to collect the custom mixfunc material names

**for** shadingRecipe **in** dynamicShadingRecipes:

**if** analysisRecipe.type == 2 **and** analysisRecipe.northDegrees!=0:

**print** "Rotating %s for %d degrees" **%** (shadingRecipe.name, analysisRecipe.northDegrees)

**if** shadingRecipe.type == 2:

groupName = shadingRecipe.name

dynamicCounter+=1

**for** stateCount, shadingState **in enumerate**(shadingRecipe.shadingStates):

fileName = groupName + "\_state\_" + **str**(stateCount + 1) + ".rad"

**try**:

radStr = ""

shdHBObjects = hb\_hive.**callFromHoneybeeHive**(shadingState.shdHBObjects)

**for** HBObj **in** shdHBObjects:

**if** rotateObjects:

HBObj.**transform**(transform, **None**, **False**)

# collect the custom material informations

**if** HBObj.RadMaterial!=**None**:

customRADMat, customMixFunRadMat = self.hb\_RADMaterialAUX.**addRADMatToDocumentDict**(HBObj, customRADMat, customMixFunRadMat)

**if** HBObj.isPlanar **and** (**not** HBObj.isChild **and len**(HBObj.childSrfs)<2):

radStr += self.**RADSurface**(HBObj)

**else**:

radStr += self.**RADNonPlanarSurface**(HBObj)

**if not** HBObj.isChild **and** HBObj.hasChild:

# collect the custom material informations

**for** childSrf **in** HBObj.childSrfs:

**if** childSrf.RadMaterial!=**None**:

customRADMat, customMixFunRadMat = self.hb\_RADMaterialAUX.**addRADMatToDocumentDict**(childSrf, customRADMat, customMixFunRadMat)

radStr += self.**RADNonPlanarChildSurface**(HBObj)

# write the shading file

with **open**(subWorkingDir + "\\" + fileName, "w") **as** radInf:

radInf.**write**(matStr)

radInf.**write**("# material(s) specific to this study\n")

**for** radMatName **in** customRADMat.**keys**():

radInf.**write**(self.hb\_RADMaterialAUX.**getRADMaterialString**(radMatName) + "\n")

# write mixedfun if any

**for** radMatName **in** customMixFunRadMat.**keys**():

radInf.**write**(self.hb\_RADMaterialAUX.**getRADMaterialString**(radMatName) + "\n")

radInf.**write**(radStr)

**except** Exception, e:

# print `e`

# None object so just create an empty file

with **open**(subWorkingDir + "\\" + fileName , "w") **as** radInf:

radInf.**write**("#empty shading file")

pass

return radFileFullName, materialFileName

**def writeTestPtFile**(self, subWorkingDir, radFileName, numOfCPUs, analysisRecipe):

**if** analysisRecipe.type == 0: return [], [] #image-based simulation

testPoints = copy.**deepcopy**(analysisRecipe.testPts)

ptsNormals = copy.**deepcopy**(analysisRecipe.vectors)

# write a pattern file which I can use later to re-branch the points

ptnFileName = os.path.**join**(subWorkingDir, radFileName + '.ptn')

with **open**(ptnFileName, "w") **as** ptnFile:

**for** ptList **in** testPoints:

ptnFile.**write**(**str**(**len**(ptList)) + ", ")

# faltten the test points and make a copy

flattenTestPoints = [pt **for** pt **in** self.lb\_preparation.**flattenList**(testPoints)]

flattenPtsNormals = [v **for** v **in** self.lb\_preparation.**flattenList**(ptsNormals)]

# if this is an annual analysis and north is not 0 rotate all Honeybee objects

**if** analysisRecipe.type == 2 **and** analysisRecipe.northDegrees!=0:

**print** "Rotating test points for %d degrees"%analysisRecipe.northDegrees

transform = rc.Geometry.Transform.**Rotation**(-math.**radians**(analysisRecipe.northDegrees), \

rc.Geometry.Point3d.Origin)

**for** pt **in** flattenTestPoints: pt.**Transform**(transform)

**for** v **in** flattenPtsNormals: v.**Transform**(transform)

numOfPoints = **len**(flattenTestPoints)

**if** numOfCPUs > numOfPoints: numOfCPUs = numOfPoints

**if** numOfCPUs > 1:

ptsEachCpu = **int**(numOfPoints**/**(numOfCPUs))

remainder = numOfPoints%numOfCPUs

**else**:

ptsEachCpu = numOfPoints

remainder = 0

lenOfPts = []

**for** cpuCount **in range**(numOfCPUs):

**if** cpuCount < remainder:

lenOfPts.**append**(ptsEachCpu+1)

**else**:

lenOfPts.**append**(ptsEachCpu)

testPtsEachCPU = []

**for** cpuCount **in range**(numOfCPUs):

# write pts file

ptsForThisCPU = []

ptsFileName = os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '.pts')

ptsFile = **open**(ptsFileName, "w")

**for** ptCount **in range**(**sum**(lenOfPts[:cpuCount]), **sum**(lenOfPts[:cpuCount+1])):

ptsFile.**write**(self.hb\_writeRADAUX.**testPtsStr**(flattenTestPoints[ptCount], flattenPtsNormals[ptCount]))

ptsForThisCPU.**append**(flattenTestPoints[ptCount])

ptsFile.**close**()

testPtsEachCPU.**append**(ptsForThisCPU)

return testPtsEachCPU, lenOfPts

**def writeBatchFiles**(self, subWorkingDir, radFileName, radSkyFileName, \

radFileFullName, materialFileName, \

numOfCPUs, testPtsEachCPU, \

lenOfPts, analysisRecipe, additionalRadFiles, \

readyOCTFile = **None**, runOverture = **True**):

batchFiles = []

fileNames = [] # list of only names of the files

pcompFileName = ""

# initiate RAD Parameters

**if** analysisRecipe.radParameters==**None**:

quality = 0

analysisRecipe.radParameters = {}

**print** "Default values are set for RAD parameters"

**for** key **in** self.hb\_radParDict.**keys**():

#print key + " is set to " + str(hb\_radParDict[key][quality])

analysisRecipe.radParameters[key] = self.hb\_radParDict[key][quality]

**if** analysisRecipe.type == 2: # annual daylight analysis - Daysim

# read parameters

runAnnualGlare = analysisRecipe.DSParameters.runAnnualGlare

onlyAnnualGlare = analysisRecipe.DSParameters.onlyAnnualGlare

annualGlareViews = analysisRecipe.DSParameters.RhinoViewsName

outputUnits = analysisRecipe.DSParameters.outputUnits

adaptiveZone = analysisRecipe.DSParameters.adaptiveZone

dgp\_imageSize = analysisRecipe.DSParameters.dgp\_imageSize

dynamicShadingRecipes = analysisRecipe.DSParameters.DShdR

numOfIllFiles = analysisRecipe.DSParameters.numOfIll

northAngleRotation = analysisRecipe.northDegrees

# empty list for result file names

DSResultFilesAddress = []

# location string

epwFileAddress = analysisRecipe.weatherFile

locationStr, locName = self.hb\_writeDS.**DSLocationStr**(self.hb\_writeRADAUX, self.lb\_preparation, epwFileAddress)

newLocName = self.lb\_preparation.**removeBlankLight**(locName)

newLocName = newLocName.**replace**("/", "\_")

# copy .epw file to sub-directory

self.lb\_preparation.**copyFile**(epwFileAddress, subWorkingDir + "\\" + newLocName + '.epw')

pathStr = "SET RAYPATH=.;" + self.hb\_RADLibPath + ";" + self.hb\_DSPath + ";" + \

self.hb\_DSLibPath + ";\nPATH=" + self.hb\_RADPath + ";" + \

self.hb\_DSPath + ";" + self.hb\_DSLibPath + ";$PATH\n"

heaFileName = os.path.**join**(subWorkingDir, radFileName + '\_0.hea')

initBatchFileName = os.path.**join**(subWorkingDir, radFileName + '\_InitDS.bat')

initBatchFile = **open**(initBatchFileName, "w")

initBatchFile.**write**(pathStr)

xformCmds = []

**if** additionalRadFiles **and** northAngleRotation != 0:

# rotate additional radiance files:

cmdbase = 'xform -rz -%f {} > {}' % northAngleRotation

**for** count, adfile **in enumerate**(additionalRadFiles):

target = adfile[:-4] + '\_' + **str**(northAngleRotation) + adfile[-4:]

xformCmds.**append**(cmdbase.**format**(adfile, target))

additionalRadFiles[count] = target

initBatchStr = os.path.**splitdrive**(self.hb\_DSPath)[0] + '\n' + \

'CD ' + self.hb\_DSPath + '\n' + \

'epw2wea ' + subWorkingDir + "\\" + self.lb\_preparation.**removeBlankLight**(locName) + '.epw ' + subWorkingDir + "\\" + self.lb\_preparation.**removeBlankLight**(locName) + '.wea\n'

**if** xformCmds:

initBatchStr += ':: Rotate additional files if any\n' + '\n'.**join**(xformCmds) + '\n'

initBatchStr += ':: 1. Generate Daysim version of Radiance Files\n' + \

'radfiles2daysim ' + heaFileName + ' -m -g\n'

# rotate scene if angle is not 0!

#if northAngleRotation!=0:

# initBatchStr += \

# ':: 1.5. Roate geometry and test points\n' + \

# 'rotate\_scene ' + heaFileName + '\n'

**if** runAnnualGlare:

initBatchStr += \

':: 2. Generate Values for annual glare\n' + \

'gen\_dgp\_profile ' + heaFileName

initBatchFile.**write**(initBatchStr)

initBatchFile.**close**()

# annual glare only needs one headeing file and will run on a single cpu

**if** runAnnualGlare: # and onlyAnnualGlare:

numOfCPUs = 1

# write the rest of the files

**for** cpuCount **in range**(numOfCPUs):

heaFileName = os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '.hea')

heaFile = **open**(heaFileName, "w")

projectName = radFileName

tempDirName = subWorkingDir + '\\tmp\_' + `cpuCount`

heaFile.**write**(self.hb\_writeDS.**DSHeadingStr**(projectName, subWorkingDir, tempDirName, self.hb\_DSCore , cpuCount))

# delete the files in the old temp folder

tempWorkingDir = self.lb\_preparation.**makeWorkingDir**(tempDirName)

heaFile.**write**(locationStr)

heaFile.**write**(self.hb\_writeDS.**DSAnalysisUnits**(outputUnits, lenOfPts[cpuCount]))

# write view for annual glare if any

glareViewFileName = subWorkingDir + '\\' + projectName + '\_' + 'annualGlareView.vf'

vfFile = **open**(glareViewFileName, "w")

vfFile.**write**('')

**for** view **in** annualGlareViews:

viewLine = self.hb\_writeRADAUX.**exportView**(view, analysisRecipe.radParameters, 1, [dgp\_imageSize, dgp\_imageSize])

# I'm not sure why Daysim view file needs rview Perspective at the start line

vfFile.**write**("rview Perspective " + viewLine + "\n")

vfFile.**close**()

# building string

heaFile.**write**(self.hb\_writeDS.**DSBldgStr**(projectName, materialFileName, radFileFullName, \

adaptiveZone, dgp\_imageSize, dgp\_imageSize, cpuCount, \

northAngleRotation, additionalRadFiles))

# radiance parameters string

heaFile.**write**(self.hb\_writeDS.**DSRADStr**(analysisRecipe.radParameters))

# dynamic simulaion options

heaFile.**write**(self.hb\_writeDS.**DSDynamicSimStr**(dynamicShadingRecipes, projectName, subWorkingDir, testPtsEachCPU[cpuCount], cpuCount))

# heaFile.write(hb\_writeDS.resultStr(projectName, cpuCount))

heaFile.**close**()

**if not**(runAnnualGlare **and** onlyAnnualGlare):

# ill files

DSResultFilesAddress.**append**(os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '.ill'))

# 3. write the batch file

DSBatchFileName = os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '\_DS.bat')

DSBatchFile = **open**(DSBatchFileName, "w")

fileNames.**append**(DSBatchFileName.**split**("\\")[-1])

heaFileName = os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '.hea')

#SET PATH = " + subWorkingDir + "\n" + workingDrive +"\n"

DSBatchFile.**write**(pathStr)

DSBatchStr = ':: Calculate Daylight Coefficient File (\*.dc)\n' + \

'gen\_dc ' + heaFileName + ' -dif\n' + \

'gen\_dc ' + heaFileName + ' -dir\n' + \

'gen\_dc ' + heaFileName + ' -paste\n' + \

'\n' + \

':: Generate Illuminance Files (\*.ill)\n' + \

'ds\_illum ' + heaFileName + '\n'

DSBatchFile.**write**(DSBatchStr)

DSBatchFile.**close**()

batchFiles.**append**(DSBatchFileName)

return initBatchFileName, batchFiles, fileNames, pcompFileName, DSResultFilesAddress

######################## NOT ANNUAL SIMULATION #######################

# 3. write the batch file

HDRFileAddress = []

**if** analysisRecipe.type == 0:

self.rhinoViewNames = analysisRecipe.viewNames

# image based

initBatchFileName = os.path.**join**(subWorkingDir, radFileName + '\_IMGInit.bat')

**if** readyOCTFile ==**None**:

OCTFileName = radFileName + '\_IMG'

**else**:

OCTFileName **=** (".").**join**(os.path.**basename**(readyOCTFile).**split**(".")[:-1])

**else**:

# not annual and not image based

initBatchFileName = os.path.**join**(subWorkingDir, radFileName + '\_RADInit.bat')

**if** readyOCTFile ==**None**:

OCTFileName = radFileName + '\_RAD'

**else**:

OCTFileName **=** (".").**join**(os.path.**basename**(readyOCTFile).**split**(".")[:-1])

# create the batch file that initiate the simulation

with **open**(initBatchFileName, "w") **as** batchFile:

# write the path string (I should check radiance to be installed on the system

pathStr = "SET RAYPATH=.;" + self.hb\_RADLibPath + "\nPATH=" + self.hb\_RADPath + ";$PATH\n"

batchFile.**write**(pathStr)

batchFile.**write**(os.path.**splitdrive**(subWorkingDir)[0] + "\n")

batchFile.**write**("cd " + subWorkingDir + "\n")

# write OCT file

# 3.2. oconv line

sceneRadFiles = [materialFileName, radSkyFileName, radFileFullName]

**if** additionalRadFiles:

**for** additionalFile **in** additionalRadFiles:

**if** additionalFile!=**None**:

sceneRadFiles.**append**(additionalFile)

OCTLine = self.hb\_writeRADAUX.**oconvLine**(OCTFileName, sceneRadFiles)

**if** readyOCTFile ==**None**: batchFile.**write**(OCTLine)

**if** analysisRecipe.type == 0:

# add overture line in case it is an image-based analysis

view = sc.doc.Views.ActiveView.ActiveViewport.Name

viewLine = self.hb\_writeRADAUX.**exportView**(view, analysisRecipe.radParameters, analysisRecipe.cameraType, imageSize = [64, 64])

# write rpict lines

overtureLine = self.hb\_writeRADAUX.**overtureLine**(viewLine, OCTFileName, view, analysisRecipe.radParameters, **int**(analysisRecipe.type))

originalView = **str**(viewLine).**strip**()

**if** runOverture: batchFile.**write**(overtureLine)

**if** analysisRecipe.type == 0:

# write view files

**if len**(self.rhinoViewNames)==0:

self.rhinoViewNames = [sc.doc.Views.ActiveView.ActiveViewport.Name]

#recalculate vh and vv

nXDiv = **int**(math.**sqrt**(numOfCPUs))

**while** numOfCPUs%nXDiv !=0 **and** nXDiv < numOfCPUs:

nXDiv += 1

nYDiv = numOfCPUs/nXDiv

fileNames = []

HDRPieces = {}

**for** cpuCount **in range**(numOfCPUs):

# create a batch file

batchFileName = os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '\_IMG.bat')

batchFiles.**append**(batchFileName)

fileNames.**append**(batchFileName.**split**("\\")[-1])

batchFile = **open**(batchFileName, "w")

# write path files

batchFile.**write**(pathStr)

batchFile.**write**(os.path.**splitdrive**(subWorkingDir)[0] + "\n")

batchFile.**write**("cd " + subWorkingDir + "\n")

# calculate vs and vl for thi cpu

**try**: vs **= ((**(cpuCount%nXDiv)**/**(nXDiv-1)) - 0.5) **\*** (nXDiv - 1)

**except**: vs = 0

**try**: vl **= (**(**int**(cpuCount/nXDiv)**/**(nYDiv-1)) - 0.5) **\*** (nYDiv - 1)

**except**: vl = 0

# print vs, vl

**for** view **in** self.rhinoViewNames:

view = self.lb\_preparation.**removeBlank**(view)

**if** cpuCount == 0:

HDRFileAddress.**append**(subWorkingDir + "\\" + OCTFileName + "\_" + view + ".HDR")

HDRPieces[OCTFileName + "\_" + view + ".HDR"] = []

# collect name of the pieces of the picture

HDRPieces[OCTFileName + "\_" + view + ".HDR"].**append**(OCTFileName + "\_" + view + "\_" + `cpuCount` + ".HDR")

viewLine = self.hb\_writeRADAUX.**exportView**(view, analysisRecipe.radParameters, analysisRecipe.cameraType, \

analysisRecipe.imageSize, analysisRecipe.sectionPlane, \

nXDiv, nYDiv, vs, vl)

# write rpict lines

RPICTLines = self.hb\_writeRADAUX.**rpictLine**(viewLine, OCTFileName, view, analysisRecipe.radParameters, **int**(analysisRecipe.simulationType), cpuCount)

batchFile.**write**(RPICTLines)

# close the file

batchFile.**close**()

# PCOMP to merge images into a single HDR

pcompFileName = os.path.**join**(subWorkingDir, radFileName + '\_PCOMP.bat')

with **open**(pcompFileName, "w") **as** pcompFile:

# write path files

pcompFile.**write**(pathStr)

pcompFile.**write**(os.path.**splitdrive**(subWorkingDir)[0] + "\n")

pcompFile.**write**("cd " + subWorkingDir + "\n")

**for** mergedName, pieces **in** HDRPieces.**items**():

pcomposLine = "pcompos -a " + `nXDiv` + " "

# pieces.reverse()

**for** piece **in** pieces:

pcomposLine += piece.**replace**('.HDR', '.unf') + " "

pcomposLine += " > " + mergedName.**replace**('.HDR', '\_temp.HDR') + "\n"

pcompFile.**write**(pcomposLine)

pfiltLine = 'pfilt -r .6 -x /2 -y /2 {} | getinfo -a "VIEW= {}" > {}\n' \

.**format**(mergedName.**replace**('.HDR', '\_temp.HDR'), originalView, mergedName)

# add original view

pcompFile.**write**(pfiltLine)

return initBatchFileName, batchFiles, fileNames, pcompFileName, HDRFileAddress

**else**:

fileNames = []

RADResultFilesAddress = []

**for** cpuCount **in range**(numOfCPUs):

# create a batch file

batchFileName = os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '\_RAD.bat')

batchFiles.**append**(batchFileName)

RADResultFilesAddress.**append**(os.path.**join**(subWorkingDir, radFileName + '\_' + `cpuCount` + '.res'))

fileNames.**append**(batchFileName.**split**("\\")[-1])

batchFile = **open**(batchFileName, "w")

# write path files

batchFile.**write**(pathStr)

batchFile.**write**(os.path.**splitdrive**(subWorkingDir)[0] + "\n")

batchFile.**write**("cd " + subWorkingDir + "\n")

# 3.4. add rtrace lin

RTRACELine = self.hb\_writeRADAUX.**rtraceLine**(radFileName, OCTFileName, analysisRecipe.radParameters, **int**(analysisRecipe.simulationType), cpuCount)

batchFile.**write**(RTRACELine)

# close the file

batchFile.**close**()

return initBatchFileName, batchFiles, fileNames, pcompFileName, RADResultFilesAddress

**def executeBatchFiles**(self, batchFileNames, maxPRuns = **None**, shell = **False**, waitingTime = 0.5):

"""Run a number of batch files in parallel and

wait to end of the analysis.

Args:

batchFileNames: List of batch files

maxPRuns: max number of files to be ran in parallel (default = 0)

shell: set to True if you do NOT want to see the cmd window while the analysis is runnig

"""

**if not** maxPRuns : maxPRuns = 1

maxPRuns = **int**(maxPRuns)

total = **len**(batchFileNames)

**if** maxPRuns < 1: maxPRuns = 1

**if** maxPRuns > total: maxPRuns = total

running = 0

done = **False**

jobs = []

pid = 0

**try**:

**while not** done:

**if** running < maxPRuns **and** pid < total:

# execute the files

jobs.**append**(subprocess.**Popen**(batchFileNames[pid].**replace**("\\", "/") , shell = shell))

pid+=1

time.**sleep**(waitingTime)

# count how many jobs are running and how many are done

running = 0

finished = 0

**for** job **in** jobs:

**if** job.**poll**() **is None**:

#one job is still running

running += 1

**else**:

finished += 1

**if** running == maxPRuns:

# wait for half a second

#print "waiting..."

time.**sleep**(waitingTime)

**if** finished == total:

done = **True**

**except** Exception, e:

**print** "Something went wrong: %s"%**str**(e)

**def runBatchFiles**(self, initBatchFileName, batchFileNames, fileNames, \

pcompBatchFile, waitingTime, runInBackground = **False**):

self.**executeBatchFiles**([initBatchFileName], maxPRuns = 1, shell = runInBackground, waitingTime = waitingTime)

self.**executeBatchFiles**(batchFileNames, maxPRuns = **len**(batchFileNames), shell = runInBackground, waitingTime = waitingTime)

**if** pcompBatchFile!="":

os.**system**(pcompBatchFile) # put all the files together

**def collectResults**(self, subWorkingDir, radFileName, numOfCPUs, analysisRecipe, expectedResultFiles):

**if** analysisRecipe.type == 2:

#annual simulation

runAnnualGlare = analysisRecipe.DSParameters.runAnnualGlare

onlyAnnualGlare = analysisRecipe.DSParameters.onlyAnnualGlare

numOfIllFiles = analysisRecipe.DSParameters.numOfIll

annualGlareViews = analysisRecipe.DSParameters.RhinoViewsName

DSResultFilesAddress = []

**if not**(runAnnualGlare **and** onlyAnnualGlare):

# read the number of .ill files

# and the number of .dc files

**if** subWorkingDir[-1] == os.sep: subWorkingDir = subWorkingDir[:-1]

startTime = time.**time**()

# check if the results are available

files = os.**listdir**(subWorkingDir)

numIll = 0

numDc = 0

**for** file **in** files:

**if** file.**EndsWith**('ill'):

DSResultFilesAddress.**append**(os.path.**join**(subWorkingDir, file))

numIll+=1

**elif** file.**EndsWith**('dc'):

numDc+=1

# /2 in case of conceptual dynamic blinds in Daysim

**if** numIll!= numOfCPUs \* numOfIllFiles **or not** \

(numDc == numOfCPUs \* numOfIllFiles **or** \

numDc == numOfCPUs \* numOfIllFiles /2):

**print** "Can't find the results for the study"

DSResultFilesAddress = []

# check for results of annual glare analysis if any

annualGlareResults = {}

**for** view **in** annualGlareViews:

**if** view **not in** annualGlareResults.**keys**():

annualGlareResults[view] = []

dgpFile = os.path.**join**(subWorkingDir, radFileName + '\_0.dgp')

**if** runAnnualGlare **and** os.path.**isfile**(dgpFile):

with **open**(dgpFile, "r") **as** dgpRes:

**for** line **in** dgpRes:

**try**:

hourlyRes = line.**split**(" ")[4:]

# for each view there should be a number

**for** view, res **in zip**(annualGlareViews, hourlyRes):

annualGlareResults[view].**append**(res.**strip**())

**except**:

pass

return DSResultFilesAddress, annualGlareResults

**elif** analysisRecipe.type == 0:

# image-based analysis

return expectedResultFiles

**else**:

RADResultFilesAddress = expectedResultFiles

# grid-based analysis

numRes = 0

files = os.**listdir**(subWorkingDir)

**for** file **in** files:

**if** file.**EndsWith**('res'): numRes+=1

**if** numRes != numOfCPUs:

**print** "Cannot find the results of the study"

RADResultFilesAddress = []

time.**sleep**(1)

return RADResultFilesAddress

**def shiftList**(self, list, number = 1):

newList = []

newList.**extend**(list[-number:])

newList.**extend**(list[:-number])

return newList

**def getsurfaceStr**(self, surface, count, coordinates):

**if** surface.RadMaterial != **None**:

surface.construction = surface.RadMaterial

**elif not hasattr**(surface, 'construction'):

**if not hasattr**(surface, 'type'):

# find the type based on

surface.type = surface.**getTypeByNormalAngle**()

#assign the construction based on type

surface.construction = surface.cnstrSet[surface.type]

srfStr = surface.construction.**replace**(" ", "\_") + " polygon " + surface.name.**strip**() + '\_' + `count` + "\n" + \

"0\n" + \

"0\n" + \

**`**(**len**(coordinates)\*3)` + "\n"

ptStr = ''

**for** pt **in** coordinates:

ptStr = ptStr + '%.4f'%pt.X + ' ' + '%.4f'%pt.Y + ' ' + '%.4f'%pt.Z + '\n'

ptStr = ptStr + '\n'

# check for polygons with only two points.

# Yes! it is possible. Import a model from REVIT/SketchUp and create some breps out of it

# and you will get some!

**if len**(coordinates) < 3:

comment = " Polygon " + surface.name + " has less than 3 vertices and is removed by Honeybee.\n"

return "#" + comment

return srfStr + ptStr

**def RADSurface**(self, surface):

fullStr = []

# base surface coordinates

coordinatesList = surface.**extractPoints**(1, **True**)

**if** coordinatesList:

**if type**(coordinatesList[0])**is not** list **and type**(coordinatesList[0]) **is not** tuple:

coordinatesList = [coordinatesList]

**for** count, coordinates **in enumerate**(coordinatesList):

endCoordinate = rc.Geometry.Point3d.**Add**(coordinates[-1], rc.Geometry.**Vector3d**(0,0,0))

**if** surface.hasChild:

glzCoordinateLists = surface.**extractGlzPoints**(**True**)

**for** glzCount, glzCoorList **in enumerate**(glzCoordinateLists):

# glazingStr

**try**:

fullStr.**append**(self.**getsurfaceStr**(surface.childSrfs[glzCount], glzCount, glzCoorList))

**except**:

fullStr.**append**(self.**getsurfaceStr**(surface.childSrfs[0], glzCount, glzCoorList))

# shift glazing list

glzCoorList = self.**shiftList**(glzCoorList)

coordinates.**extend**(glzCoorList)

coordinates.**append**(glzCoorList[0])

coordinates.**extend**([endCoordinate, coordinates[0]])

fullStr.**append**(self.**getsurfaceStr**(surface, count, coordinates))

return ''.**join**(fullStr)

**else**:

**print** "one of the surfaces is not exported correctly"

return ""

**def RADNonPlanarSurface**(self, surface):

fullStr = []

# replace the geometry with the punched geometry

# for planar surfaces with multiple openings

**try**:

**if** surface.punchedGeometry!=**None**:

surface.geometry = surface.punchedGeometry

surface.hasInternalEdge = **True**

**except**:

#print e

# nonplanar surfaces with no openings

pass

# base surface coordinates

coordinatesList = surface.**extractPoints**(1, **True**)

**if type**(coordinatesList[0])**is not** list **and type**(coordinatesList[0]) **is not** tuple:

coordinatesList = [coordinatesList]

**for** count, coordinates **in enumerate**(coordinatesList):

#print count

fullStr.**append**(self.**getsurfaceStr**(surface, count, coordinates))

return ''.**join**(fullStr)

**def RADNonPlanarChildSurface**(self, surface):

fullStr = []

# I should test this function before the first release!

# Not sure if it will work for cases generated only by surface

# should probably check for meshed surface and mesh the geometry

# in case it is not meshed

# base surface coordinates

coordinatesList = surface.**extractGlzPoints**(**True**)

**if type**(coordinatesList[0])**is not** list **and type**(coordinatesList[0]) **is not** tuple:

coordinatesList = [coordinatesList]

**for** glzCount, glzCoorList **in enumerate**(coordinatesList):

# glazingStr`

**try**:

fullStr.**append**(self.**getsurfaceStr**(surface.childSrfs[glzCount], glzCount, glzCoorList))

**except**:

fullStr.**append**(self.**getsurfaceStr**(surface.childSrfs[0], glzCount, glzCoorList))

return ''.**join**(fullStr)

**class hb\_WriteRADAUX**(object):

**def \_\_init\_\_**(self):

self.hb\_radParDict = sc.sticky["honeybee\_RADParameters"**]**().radParDict

self.lb\_preparation = sc.sticky["ladybug\_Preparation"**]**()

self.hb\_serializeObjects = sc.sticky["honeybee\_SerializeObjects"]

self.hb\_dsParameters = sc.sticky["honeybee\_DSParameters"**]**()

self.radSkyCondition = {0: '-u',

1: '-c',

2: '-i',

3: '+i',

4: '-s',

5: '+s'}

self.DLAnalaysisTypes = {0: ["0: illuminance" , "lux"],

1: ["1: radiation" , "wh/m2"],

1.1: ["1.1: cumulative radiation" , "kWh/m2"],

2: ["2: luminance" , "cd/m2"],

3: ["3: DF", "%"],

4: ["4: VSC", "%"],

5: ["5: annual analysis", "var"]}

**def readAnalysisRecipe**(self, analysisRecipe):

self.analysisType = analysisRecipe.type

self.radParameters = analysisRecipe.radParameters

self.backupImages = 0 # will change to 1 or 2 in case the user set it to another number for image-based analysis

self.numOfIllFiles = 1

**if** self.radParameters==**None**:

quality = 0

self.radParameters = {}

**print** "Default values are set for RAD parameters"

**for** key **in** self.hb\_radParDict.**keys**():

#print key + " is set to " + str(hb\_radParDict[key][quality])

self.radParameters[key] = self.hb\_radParDict[key][quality]

**if** self.analysisType == 0:

**print** "Image-based simulation"

self.radSkyFileName = analysisRecipe.skyFile

self.rhinoViewNames = analysisRecipe.viewNames

self.cameraType = analysisRecipe.cameraType

self.imageSize = analysisRecipe.imageSize

self.simulationType = analysisRecipe.simulationType

self.studyFolder = analysisRecipe.studyFolder

self.sectionPlane = analysisRecipe.sectionPlane

self.backupImages = analysisRecipe.backupImages

**elif** self.analysisType == 1:

**print** "Grid-based Radiance simulation"

self.radSkyFileName = analysisRecipe.skyFile

self.testPoints = analysisRecipe.testPts

self.ptsNormals = analysisRecipe.vectors

self.simulationType = analysisRecipe.simulationType

self.studyFolder = analysisRecipe.studyFolder

self.testMesh = analysisRecipe.testMesh

**elif** self.analysisType == 2:

**print** "Annual climate-based analysis"

self.epwFileAddress = analysisRecipe.weatherFile

self.testPoints = analysisRecipe.testPts

self.ptsNormals = analysisRecipe.vectors

self.testMesh = analysisRecipe.testMesh

**if** analysisRecipe.DSParameters == **None**:

analysisRecipe.DSParameters = self.hb\_dsParameters

self.runAnnualGlare = analysisRecipe.DSParameters.runAnnualGlare

self.onlyAnnualGlare = analysisRecipe.DSParameters.onlyAnnualGlare

self.annualGlareViews = analysisRecipe.DSParameters.RhinoViewsName

self.outputUnits = analysisRecipe.DSParameters.outputUnits

self.adaptiveZone = analysisRecipe.DSParameters.adaptiveZone

self.dgp\_imageSize = analysisRecipe.DSParameters.dgp\_imageSize

self.dynamicShadingRecipes = analysisRecipe.DSParameters.DShdR

self.numOfIllFiles = analysisRecipe.DSParameters.numOfIll

self.studyFolder = analysisRecipe.studyFolder

**elif** self.analysisType == 3:

**print** "Daylight factor"

self.radSkyFileName = analysisRecipe.skyFile

self.testPoints = analysisRecipe.testPts

self.ptsNormals = analysisRecipe.vectors

self.simulationType = analysisRecipe.simulationType

self.studyFolder = analysisRecipe.studyFolder

self.testMesh = analysisRecipe.testMesh

**elif** self.analysisType == 4:

**print** "Vertical Sky Component"

self.radSkyFileName = analysisRecipe.skyFile

self.testPoints = analysisRecipe.testPts

self.ptsNormals = analysisRecipe.vectors

self.simulationType = analysisRecipe.simulationType

self.studyFolder = analysisRecipe.studyFolder

self.testMesh = analysisRecipe.testMesh

**def checkInputParametersForGridBasedAnalysis**(self):

**if** self.analysisType == 0:

# this is an image-based analysis

return

**print** "The component is checking ad, as, ar and aa values. " + \

"This is just to make sure that the results are accurate enough."

**if** self.radParameters["\_ad\_"] < 1000:

self.radParameters["\_ad\_"] = 1000

**print** "-ad is set to 1000."

**if** self.radParameters["\_as\_"] < 20:

self.radParameters["\_as\_"] = 20

**print** "-as is set to 20."

**if** self.radParameters["\_ar\_"] != 0 **and** self.radParameters["\_ar\_"] < 300:

# setting up the ar to 300 is tricky but I'm pretty sure in many

# cases there will shadings involved.

self.radParameters["\_ar\_"] = 300

**print** "-ar is set to 300."

**if** self.radParameters["\_aa\_"] > 0.1:

# the same here. I think it is good to let the user wait a little bit more

# but have a result that makes sense. If you are an exprienced user and don't

# like this feel free to remove the if condition. Keep in mind that I only

# apply this for grid based analysis, so the images can be rendered with any quality

self.radParameters["\_aa\_"] = 0.1

**print** "-aa is set to 0.1"

**print** "Good to go!"

**def prepareWorkingDir**(self, workingDir, radFileName = **None**, overwriteResults = **True**, analysisRecipe = **None**):

**if** analysisRecipe == **None**:

studyFolder = self.studyFolder

analysisType = self.analysisType

**if** analysisType == 0:

backupImages = self.backupImages

**else**:

studyFolder = analysisRecipe.studyFolder

analysisType = analysisRecipe.type

**if** analysisType == 0:

backupImages = analysisRecipe.backupImages

**if** workingDir:

workingDir = self.lb\_preparation.**removeBlankLight**(workingDir)

**else**:

workingDir = sc.sticky["Honeybee\_DefaultFolder"]

workingDir = self.lb\_preparation.**makeWorkingDir**(workingDir)

# make sure the directory has been created

**if** workingDir == -1: return -1

workingDrive = workingDir[0:1]

## check for the name of the file

**if** radFileName == **None**: radFileName = 'unnamed'

# make sure radfile name is a valid address

keepcharacters **=** ('.','\_')

radFileName = "".**join**([c **for** c **in** radFileName **if** c.**isalnum**() **or** c **in** keepcharacters]).**rstrip**()

# make new folder for each study

subWorkingDir = self.lb\_preparation.**makeWorkingDir**(workingDir + "\\" + radFileName + studyFolder).**replace**("\\\\", "\\")

**print** 'Current working directory is set to: ', subWorkingDir

**if** os.path.**exists**(subWorkingDir):

**if** analysisType == 0:

# for image-based analysis there is an option to backup the images

**if** backupImages != 0:

# create the backup folder and copy the images to the folder

imageFolder = workingDir + "\\" + radFileName + "\\imagesBackup"

**if not** os.path.**exists**(imageFolder): os.**mkdir**(imageFolder)

# copy the files into the folder

imageExtensions = ["JPEG", "JPG", "GIF", "TIFF", "TIF", "HDR", "PIC"]

timeID = self.**getTime**()

fileNames = os.**listdir**(subWorkingDir)

**if** backupImages == 1:

# keep all the files in the same folder

**for** fileName **in** fileNames:

**if** fileName.**split**(".")[-1].**upper**() **in** imageExtensions:

newFileName **=** (".").**join**(fileName.**split**(".")[:-1])

extension = fileName.**split**(".")[-1]

newFullName = newFileName + "\_" + timeID + "." + extension

self.**copyFile**(os.path.**join**(subWorkingDir, fileName), os.path.**join**(imageFolder, newFullName) , **True**)

**elif** backupImages == 2:

**for** fileName **in** fileNames:

**if** fileName.**split**(".")[-1].**upper**() **in** imageExtensions:

**if not** os.path.**exists**(imageFolder + "\\" + timeID):

os.**mkdir**(imageFolder + "\\" + timeID)

# copy the files to image backup folder with data and time added

self.**copyFile**(os.path.**join**(subWorkingDir, fileName), os.path.**join**(imageFolder + "\\" + timeID, fileName) , **True**)

**try**:

**if not** overwriteResults:

fileNames = os.**listdir**(subWorkingDir)

mainBackupFolder = subWorkingDir[:-1] + "\_backup"

counter = 0

backupFolder = os.path.**join**(mainBackupFolder, **str**(counter))

**while** os.path.**isdir**(backupFolder):

counter += 1

backupFolder = os.path.**join**(mainBackupFolder, **str**(counter))

os.**mkdir**(backupFolder)

**for** fileName **in** fileNames:

**try**:

# copy the files to image backup folder with data and time added

self.**copyFile**(os.path.**join**(subWorkingDir, fileName), os.path.**join**(backupFolder, fileName) , **True**)

**except**:

pass

**print** "Results of the previous study are copied to " + backupFolder

self.lb\_preparation.**nukedir**(subWorkingDir, rmdir = **False**)

**except** Exception, e:

**print** 'Failed to remove the old directory.'

**print** `e`

return subWorkingDir, radFileName

**def exportTestMesh**(self, subWorkingDir, radFileName, analysisRecipe = **None**):

**if** analysisRecipe != **None**:

analysisType = analysisRecipe.type

**if** analysisType ==0: return

testMesh = analysisRecipe.testMesh

**else**:

analysisType = self.analysisType

**if** analysisType ==0: return

testMesh = self.testMesh

# try to write mesh file if any

**if** analysisType != 0 **and** testMesh !=[]:

meshFilePath = os.path.**join**(subWorkingDir, radFileName + ".msh")

serializer = self.**hb\_serializeObjects**(meshFilePath, testMesh)

serializer.**saveToFile**()

**def exportTypeFile**(self, subWorkingDir, radFileName, analysisRecipe):

analysisType = analysisRecipe.type

**if** analysisType == 3 **or** analysisType == 4:

analysisTypeKey = analysisType

**elif** analysisType == 0 **or** analysisType == 1:

analysisTypeKey = analysisRecipe.simulationType

**elif** analysisType == 2:

# annual analysis

analysisTypeKey = 5

# try to write mesh file if any

typeFile = os.path.**join**(subWorkingDir, radFileName + ".typ")

with **open**(typeFile, "w") **as** typf:

typf.**write**(**str**(analysisTypeKey))

**def copySkyFile**(self, subWorkingDir, radFileName, analysisRecipe = **None**):

**if** analysisRecipe != **None**:

analysisType = analysisRecipe.type

**if** analysisType == 2: return

radSkyFileName = analysisRecipe.radSkyFileName

**else**:

analysisType = self.analysisType

**if** analysisType == 2: return

radSkyFileName = self.radSkyFileName

skyTempName = radSkyFileName.**split**("\\")[-1]

skyName = skyTempName.**split**("/")[-1]

self.**copyFile**(radSkyFileName, subWorkingDir + "\\" + skyName, **True**)

radSkyFileName = os.path.**join**(subWorkingDir, skyName)

return radSkyFileName

**def getTime**(self):

**def addZero**(number):

**if len**(**str**(number)) == 1:

return "0" + **str**(number)

**else**:

return **str**(number)

year, month, day, hour, minute, second = time.**localtime**()[0:6]

now = **addZero**(hour) + "\_" + **addZero**(minute) + "\_" + **addZero**(second)

date = **addZero**(year) + "\_" + **addZero**(month) + "\_" + **addZero**(day)

return date + "at" + now

**def copyFile**(self, inputFile, destinationFullpath, overwrite = **False**):

**if** overwrite: shutil.**copyfile**(inputFile, destinationFullpath)

**elif not** os.path.**isfile**(destinationFullpath): shutil.**copyfile**(inputFile, destinationFullpath)

**def RADLocation**(self, epw\_file):

epwfile = **open**(epw\_file,"r")

headline = epwfile.**readline**()

csheadline = headline.**split**(',')

**while** 1>0: #remove empty cells from the end of the list if any

**try**: **float**(csheadline[-1]); break

**except**: csheadline.**pop**()

locName = ''

**for** hLine **in range**(1,4):

**if** csheadline[hLine] != '-':

locName = locName + csheadline[hLine].**strip**() + '\_'

locName = locName[:-1].**strip**()

lat = csheadline[-4]

lngt = csheadline[-3]

timeZone = csheadline[-2]

elev = csheadline[-1].**strip**()

epwfile.**close**()

return locName, lat, lngt, timeZone, elev

**def RADRadiationSky**(self, projectName):

return "# start of sky definition for radiation studies\n" + \

"void brightfunc skyfunc\n" + \

"2 skybright " + projectName + ".cal\n" + \

"0\n" + \

"0\n" + \

"skyfunc glow sky\_glow\n" + \

"0\n" + \

"0\n" + \

"4 1 1 1 0\n" + \

"sky\_glow source sky\n" + \

"0\n" + \

"0\n" + \

"4 0 0 1 180\n" + \

"# end of sky definition for radiation studies\n\n"

**def RADDaylightingSky**(self, epwFileAddress, skyCondition, time, month, day):

locName, lat, long, timeZone, elev = self.**RADLocation**(epwFileAddress)

return "# start of sky definition for daylighting studies\n" + \

"# location name: " + locName + " LAT: " + lat + "\n" + \

"!gensky " + `month` + ' ' + `day` + ' ' + `time` + ' ' + self.radSkyCondition[skyCondition] + \

" -a " + lat + " -o " + `-**float**(long)` + " -m " + `-**float**(timeZone) \* 15` + "\n" + \

"skyfunc glow sky\_mat\n" + \

"0\n" + \

"0\n" + \

"4 1 1 1 0\n" + \

"sky\_mat source sky\n" + \

"0\n" + \

"0\n" + \

"4 0 0 1 180\n" + \

"skyfunc glow ground\_glow\n" + \

"0\n" + \

"0\n" + \

"4 1 .8 .5 0\n" + \

"ground\_glow source ground\n" + \

"0\n" + \

"0\n" + \

"4 0 0 -1 180\n" + \

"# end of sky definition for daylighting studies\n\n"

**def exportView**(self, viewName, radParameters, cameraType, imageSize, sectionPlane = **None**, nXDiv = 1, nYDiv = 1, vs = 0, vl = 0):

**if** viewName **in** rs.**ViewNames**():

viewName = rs.**CurrentView**(viewName, **True**)

**else**:

# change to RhinoDoc to get access to NamedViews

sc.doc = rc.RhinoDoc.ActiveDoc

namedViews = rs.**NamedViews**()

**if** viewName **in** namedViews:

viewName = rs.**RestoreNamedView**(viewName)

**else**:

viewName = **None**

# change back to Grasshopper

sc.doc = ghdoc

viewName = rs.**CurrentView**(viewName, **True**)

**if** viewName == **None**:

**print** "Illegal view name!"

viewName = "Perspective"

# Read camera type 0: Perspective, 1: fisheye, 2: parallel

**try**: cameraType = **int**(cameraType)

**except**:

**if** sc.doc.Views.ActiveView.ActiveViewport.IsPerspectiveProjection: cameraType = 0

**elif** sc.doc.Views.ActiveView.ActiveViewport.IsParallelProjection: cameraType = 2

# paralell view sizes

viewRect = sc.doc.Views.ActiveView.ActiveViewport.**GetNearRect**()

viewHSizeP = **int**(viewRect[0].**DistanceTo**(viewRect[1]))

viewVSizeP = **int**(viewRect[0].**DistanceTo**(viewRect[2]))

# read image size

viewHSize = **int**(sc.doc.Views.ActiveView.ActiveViewport.Size.Width)

viewVSize = **int**(sc.doc.Views.ActiveView.ActiveViewport.Size.Height)

# print viewHSize, viewVSize

userInputH = imageSize[0]

userInputV = imageSize[1]

**if** userInputH != **None and** userInputV != **None**:

**try**:

viewHSize = **float**(userInputH)

viewVSize = **float**(userInputV)

**except**:

**print** "Illegal input for view size."

pass

**elif** userInputH == **None and** userInputV != **None**:

**try**:

viewHSize = viewHSize **\*** (userInputV/viewVSize)

viewVSize = **float**(userInputV)

**except**:

**print** "Illegal input for view size."

pass

**elif** userInputH != **None and** userInputV == **None**:

**try**:

viewVSize = viewVSize **\*** (userInputH/viewHSize)

viewHSize = **float**(userInputH)

**except**:

**print** "Illegal input for view size."

pass

# print viewHSize, viewVSize

viewPoint = sc.doc.Views.ActiveView.ActiveViewport.CameraLocation

viewDirection = sc.doc.Views.ActiveView.ActiveViewport.CameraDirection

viewDirection.**Unitize**()

viewUp = sc.doc.Views.ActiveView.ActiveViewport.CameraUp

viewUp.**Unitize**()

**try**:

viewHA = 180 - rs.**VectorAngle**(sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumRightPlane**()[1][1], sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumLeftPlane**()[1][1])

**except**:

viewHA = 180 - rs.**VectorAngle**(sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumRightPlane**()[1].Normal, sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumLeftPlane**()[1].Normal)

**if** viewHA == 0: viewHA = 180

**try**:

viewVA = 180 - rs.**VectorAngle**(sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumBottomPlane**()[1][1], sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumTopPlane**()[1][1])

**except**:

viewVA = 180 - rs.**VectorAngle**(sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumBottomPlane**()[1].Normal, sc.doc.Views.ActiveView.ActiveViewport.**GetFrustumTopPlane**()[1].Normal)

**if** viewVA == 0: viewVA = 180

PI = math.pi

**if** cameraType == 2:

# Thank you to Brent Watanabe for the great discussion, and his help in figuring this out

# I should find the bounding box of the geometry and set X and Y based of that!

**if** nXDiv != 1:

viewHSizeP = viewHSizeP/nXDiv

viewHSize = viewHSize/nXDiv

**if** nYDiv != 1:

viewVSizeP = viewVSizeP/nYDiv

viewVSize = viewVSize/nYDiv

view = "-vtl -vp " + \

`viewPoint[0]` + " " + `viewPoint[1]` + " " + `viewPoint[2]` + " " + \

" -vd " + `viewDirection[0]` + " " + `viewDirection[1]` + " " + `viewDirection[2]` + " " + \

" -vu " + `viewUp[0]` + " " + `viewUp[1]` + " " + `viewUp[2]` + \

" -vh " + `**int**(viewHSizeP)` + " -vv " + `**int**(viewVSizeP)` + \

" -vs " + "%.3f"%vs + " -vl " + "%.3f"%vl + \

" -x " + `**int**(viewHSize)` + " -y " + `**int**(viewVSize)`

**elif** cameraType == 0:

# perspective

# recalculate vh and vv

**if** nXDiv != 1:

viewHA **=** (2.\*180./PI)\*math.**atan((**(PI/180./2.) \* viewHA)/nXDiv)

viewHSize = viewHSize/nXDiv

**if** nYDiv != 1:

viewVA **=** (2.\*180./PI)\*math.**atan**(math.**tan(**(PI/180./2.)\*viewVA)/nYDiv)

viewVSize = viewVSize/nYDiv

view = "-vtv -vp " + \

"%.3f"%viewPoint[0] + " " + "%.3f"%viewPoint[1] + " " + "%.3f"%viewPoint[2] + " " + \

" -vd " + "%.3f"%viewDirection[0] + " " + "%.3f"%viewDirection[1] + " " + "%.3f"%viewDirection[2] + " " + \

" -vu " + "%.3f"%viewUp[0] + " " + "%.3f"%viewUp[1] + " " + "%.3f"%viewUp[2] + " " + \

" -vh " + "%.3f"%viewHA + " -vv " + "%.3f"%viewVA + \

" -vs " + "%.3f"%vs + " -vl " + "%.3f"%vl + " -x " + `**int**(viewHSize)` + " -y " + `**int**(viewVSize)`

**elif** cameraType == 1:

# fish eye

# recalculate vh and vv

viewHA = 180

viewVA = 180

**if** nXDiv != 1:

viewHA **=** (2.\*180./PI)\*math.**asin**(math.**sin(**(PI/180./2.)\*viewHA)/nXDiv)

viewHSize = viewHSize/nXDiv

**if** nYDiv != 1:

viewVA **=** (2.\*180./PI)\*math.**asin**(math.**sin(**(PI/180./2.)\*viewVA)/nYDiv)

viewVSize = viewVSize/nYDiv

view = "-vth -vp " + \

`viewPoint[0]` + " " + `viewPoint[1]` + " " + `viewPoint[2]` + " " + \

" -vd " + `viewDirection[0]` + " " + `viewDirection[1]` + " " + `viewDirection[2]` + " " + \

" -vu " + `viewUp[0]` + " " + `viewUp[1]` + " " + `viewUp[2]` + " " + \

" -vh " + "%.3f"%viewHA + " -vv " + "%.3f"%viewVA + \

" -vs " + "%.3f"%vs + " -vl " + "%.3f"%vl + " -x " + `**int**(viewHSize)` + " -y " + `**int**(viewVSize)`

**if** sectionPlane!=**None**:

# map the point on the plane

pointOnPlane = sectionPlane.**ClosestPoint**(viewPoint)

distance = pointOnPlane.**DistanceTo**(viewPoint)

view += " -vo " + **str**(distance)

return view + " "

**def oconvLine**(self, octFileName, radFilesList):

# sence files

r = 1024 \* 2

senceFiles = ""

**for** address **in** radFilesList: senceFiles = senceFiles + address.**replace**("\\" , "/") + " "

line = "oconv -r " + **str**(r) + " -f " + senceFiles + " > " + octFileName + ".oct\n"

return line

**def overtureLine**(self, view, projectName, viewName, radParameters, analysisType = 0):

octFile = projectName + ".oct"

ambFile = projectName + ".amb" #amb file is view independent and can be used globally

unfFile = projectName + ".unf"

**if** analysisType==0:

# illuminance (lux)

line0 = "rpict -i "

**elif** analysisType==2:

# luminance (cd)

line0 = "rpict "

**else**:

# radiation analysis

line0 = "rpict -i "

line1\_1 = "-t 10 "+ \

view + " -af " + ambFile + " " + \

" -ps " + **str**(radParameters["\_ps\_"]) + " -pt " + **str**(radParameters["\_pt\_"]) + \

" -pj " + **str**(radParameters["\_pj\_"]) + " -dj " + **str**(radParameters["\_dj\_"]) + \

" -ds " + **str**(radParameters["\_ds\_"]) + " -dt " + **str**(radParameters["\_dt\_"]) + \

" -dc " + **str**(radParameters["\_dc\_"]) + " -dr " + **str**(radParameters["\_dr\_"]) + \

" -dp " + **str**(radParameters["\_dp\_"]) + " -st " + **str**(radParameters["\_st\_"]) + \

" -ab " + `radParameters["\_ab\_"]` + \

" -ad " + `radParameters["\_ad\_"]` + " -as " + `radParameters["\_as\_"]` + \

" -ar " + `radParameters["\_ar\_"]` + " -aa " + '%.3f'%radParameters["\_aa\_"] + \

" -lr " + `radParameters["\_lr\_"]` + " -lw " + '%.3f'%radParameters["\_lw\_"] + " -av 0 0 0 "

line1\_2 = " "

**if** radParameters.**has\_key**("additional"):

**for** par **in** radParameters["additional"]:

line1\_2 += "-%s "%par

line1\_3 = octFile + " > " + unfFile + "\n"

line2 = "del " + unfFile + "\n"

return line0 + line1\_1 + line1\_2 + line1\_3 + line2

**def rpictLine**(self, view, projectName, viewName, radParameters, analysisType = 0, cpuCount = 0):

octFile = projectName + ".oct"

ambFile = projectName + ".amb" #amb file is view independent and can be used globally

unfFile = projectName + "\_" + viewName + "\_" + `cpuCount` + ".unf"

outputFile = projectName + "\_" + viewName + "\_" + `cpuCount` + ".HDR"

**if** analysisType==0:

# illuminance (lux)

line0 = "rpict -i "

**elif** analysisType==2:

# luminance (cd)

line0 = "rpict "

**else**:

# radiation analysis

line0 = "rpict -i "

line1\_1 = "-t 10 "+ \

view + " -af " + ambFile + " " + \

" -ps " + **str**(radParameters["\_ps\_"]) + " -pt " + **str**(radParameters["\_pt\_"]) + \

" -pj " + **str**(radParameters["\_pj\_"]) + " -dj " + **str**(radParameters["\_dj\_"]) + \

" -ds " + **str**(radParameters["\_ds\_"]) + " -dt " + **str**(radParameters["\_dt\_"]) + \

" -dc " + **str**(radParameters["\_dc\_"]) + " -dr " + **str**(radParameters["\_dr\_"]) + \

" -dp " + **str**(radParameters["\_dp\_"]) + " -st " + **str**(radParameters["\_st\_"]) + \

" -ab " + `radParameters["\_ab\_"]` + \

" -ad " + `radParameters["\_ad\_"]` + " -as " + `radParameters["\_as\_"]` + \

" -ar " + `radParameters["\_ar\_"]` + " -aa " + '%.3f'%radParameters["\_aa\_"] + \

" -lr " + `radParameters["\_lr\_"]` + " -lw " + '%.3f'%radParameters["\_lw\_"] + " -av 0 0 0 "

line1\_2 = " "

**if** radParameters.**has\_key**("additional"):

**for** par **in** radParameters["additional"]:

line1\_2 += "-%s "%par

line1\_3 = " -e error.log " + octFile + " > " + unfFile + "\n"

return line0 + line1\_1 + line1\_2 + line1\_3

**def falsecolorLine**(self, projectName, viewName):

line = "c:\python27\python c:\honeybee\\falsecolor2.py -i " + projectName + "\_RAD\_" + viewName + "\_RadStudy.pic -s auto -n 10 -mask 0.1 -l kWhm-2 -z > " + projectName + "\_" + viewName + "\_FalseColored.pic\n" + \

"ra\_tiff " + projectName + "\_" + viewName + "\_FalseColored.pic " + projectName + "\_" + viewName + "\_FalseColored.tif\n" + \

"ra\_gif " + projectName + "\_" + viewName + "\_FalseColored.pic " + projectName + "\_" + viewName + "\_FalseColored.gif\n"

return line

**def rtraceLine**(self, projectName, octFileName, radParameters, simulationType = 0, cpuCount = 0):

ptsFile = projectName + "\_" + **str**(cpuCount) + ".pts"

outputFile = projectName + "\_" + **str**(cpuCount) + ".res"

**if** simulationType == 0:

line0 = "rtrace -I "

**elif** simulationType == 2:

line0 = "rtrace "

**else**:

# print "Fix this for radiation analysis"

line0 = "rtrace -I "

line1\_1 = " -h -dp " + **str**(radParameters["\_dp\_"]) + \

" -ds " + **str**(radParameters["\_ds\_"]) + " -dt " + **str**(radParameters["\_dt\_"]) + \

" -dc " + **str**(radParameters["\_dc\_"]) + " -dr " + **str**(radParameters["\_dr\_"]) + \

" -st " + **str**(radParameters["\_st\_"]) + " -lr " + **str**(radParameters["\_lr\_"]) + \

" -lw " + **str**(radParameters["\_lw\_"]) + " -ab " + **str**(radParameters["\_ab\_"]) + \

" -ad " + **str**(radParameters["\_ad\_"]) + " -as " + **str**(radParameters["\_as\_"]) + \

" -ar " + **str**(radParameters["\_ar\_"]) + " -aa " + **str**(radParameters["\_aa\_"])

line1\_2 = " "

**if** radParameters.**has\_key**("additional"):

**for** par **in** radParameters["additional"]:

line1\_2 += "-%s "%par

line1\_3 = " -af " + projectName + ".amb -e error.log " + octFileName + ".oct < " + ptsFile + \

" > " + outputFile + "\n"

return line0 + line1\_1 + line1\_2 + line1\_3

**def testPtsStr**(self, testPoint, ptsNormal):

return '%.4f'%testPoint.X + '\t' + \

'%.4f'%testPoint.Y + '\t' + \

'%.4f'%testPoint.Z + '\t' + \

'%.4f'%ptsNormal.X + '\t' + \

'%.4f'%ptsNormal.Y + '\t' + \

'%.4f'%ptsNormal.Z + '\n'

**def readRadiationResult**(self, resultFile):

result = []

resultFile = **open**(resultFile,"r")

**for** line **in** resultFile: result.**append**(**float**(line.**split**(' ')[0])\*179)

return result

**def readDLResult**(self, resultFile):

result = []

resultFile = **open**(resultFile,"r")

**for** line **in** resultFile:

R, G, B = line.**split**(' ')[0:3]

result.**append**( 179 **\*** (.265 \* **float**(R) + .67 \* **float**(G) + .065 \* **float**(B)))

return result

**def isSrfAirWall**(self, HBSrf):

# This can be tricky since some of interior walls may or may not be air walls

**if** HBSrf.type == 4:

return **True**

**else**:

return **False**

**def isSrfInterior**(self, HBSrf):

# This can be tricky since some of interior walls may or may not be air walls

**if** HBSrf.type == 0 **and** HBSrf.BC.**lower**() == "surface":

return **True**

**else**:

return **False**

**class hb\_WriteDS**(object):

**def isSensor**(self, testPt, sensors):

**for** pt **in** sensors:

**if** pt.**DistanceTo**(testPt) < sc.doc.ModelAbsoluteTolerance:

# this is a senor point

return **True**

# not a sensor

return **False**

**def DSHeadingStr**(self, projectName, subWorkingDir, tempFolder, hb\_DSPath, cpuCount = 0):

return '#######################################\n' + \

'#DAYSIM HEADING - GENERATED BY HONEYBEE\n' + \

'#######################################\n' + \

'project\_name ' + projectName + '\_' + `cpuCount` + '\n' + \

'project\_directory ' + subWorkingDir + '\\\n' + \

'bin\_directory ' + hb\_DSPath + '\\bin\\\n' + \

'tmp\_directory ' + tempFolder + '\\\n' + \

'Template\_File ' + hb\_DSPath + '\\template\\DefaultTemplate.htm\n'

**def DSLocationStr**(self, hb\_writeRADAUX, lb\_preparation, epwFileAddress):

# location information

locName, lat, long, timeZone, elev = hb\_writeRADAUX.**RADLocation**(epwFileAddress)

locName = locName.**replace**("/", "\_")

return'\n\n#################################\n' + \

'# LOCATION INFORMATION \n' + \

'#################################\n' + \

'place ' + lb\_preparation.**removeBlankLight**(locName) + '\n' + \

'latitude ' + lat + '\n' + \

'longitude ' + `-**float**(long)` + '\n' + \

'time\_zone ' + `-15 \* **float**(timeZone)` + '\n' + \

'site\_elevation ' + elev + '\n' + \

'time\_step ' + '60\n' + \

'wea\_data\_short\_file ' + lb\_preparation.**removeBlankLight**(locName) + '.wea\n' + \

'wea\_data\_short\_file\_units ' + '1\n' + \

'lower\_direct\_threshold ' + '2\n' + \

'lower\_diffuse\_threshold ' + '2\n', locName

**def DSAnalysisUnits**(self, outputUnits, pointsCount):

# I notice that setting output\_units to 1 return all 0 results and not the radiation values

# however assigning type 2 for each point using sensor\_file\_unit works! I think this is a bug

# in Daysim that I should report to the email list next week when I come back from Chicago.

outputUnits = outputUnits[0]

**if** outputUnits == 2:

return 'output\_units ' + `outputUnits` + '\n'

**elif** outputUnits == 1:

outputStr = "sensor\_file\_unit"

**for** pt **in range**(pointsCount): outputStr += " 2"

return outputStr +"\n"

# building information

**def DSBldgStr**(self, projectName, materialFileName, radFileFullName, adaptiveZone, \

dgp\_image\_x = 500, dgp\_image\_y = 500, cpuCount = 0, northAngle = 0, additionalFileNames = []):

# add additional rad files to scene

radFilesLength = **str**(2 + **len**(additionalFileNames))

radFileNames = ", ".**join**([radFilesLength, materialFileName, radFileFullName] + additionalFileNames)

return'\n\n#################################\n' + \

'# BUILDING INFORMATION \n' + \

'#################################\n' + \

'material\_file Daysim\_material\_' + projectName + '.rad\n' + \

'geometry\_file Daysim\_'+ projectName + '.rad\n' + \

'radiance\_source\_files ' + radFileNames + '\n' + \

'sensor\_file ' + projectName + '\_' + `cpuCount` + '.pts\n' + \

'viewpoint\_file ' + projectName + '\_' + 'annualGlareView.vf\n' + \

'AdaptiveZoneApplies ' + `adaptiveZone` + '\n' + \

'dgp\_image\_x\_size ' + `dgp\_image\_x` + '\n' + \

'dgp\_image\_y\_size ' + `dgp\_image\_y` + '\n'

# 'scene\_rotation\_angle ' + `northAngle` + '\n' # I just take care of this in Grasshopper

# radiance parameters

**def DSRADStr**(self, radParameters):

header = '\n\n#################################\n' + \

'# RADIANCE PARAMETERS \n' + \

'#################################\n'

**def checkkey**(k):

return k.**replace**('\_', '') **not in** ('xScale', 'yScale', 'additional')

params = '\n'.**join**('{} {}'.**format**(k.**replace**('\_', ''), v)

**for** k, v **in** radParameters.**iteritems**()

**if checkkey**(k))

return header + params + '\n'

**def DSDynamicSimStr**(self, shadingRecipes, projectName, subWorkingDir, testPts, cpuCount = 0):

dynOptStr = '\n==========================\n' + \

'= shading control system\n' + \

'==========================\n'

numOfDynamicShadings = 0

# find number of dynamic shadings

**for** shadingRecipe **in** shadingRecipes:

**if** shadingRecipe.type == 2:

numOfDynamicShadings += 1

dynamicShdHeading ='shading -' + **str**(numOfDynamicShadings) + '\n' + \

projectName + '\_' + `cpuCount` + '.dc ' + projectName + '\_' + `cpuCount` + '.ill\n'

dynamicCounter = 0

**for** recipeCount, shadingRecipe **in enumerate**(shadingRecipes):

name = shadingRecipe.name

type = shadingRecipe.type

**if** type == 1:

# no dynamic blind

sensorPts = []

dynamicShd ='shading ' + **str**(type) + ' ' + name + ' ' + projectName + '\_' + `cpuCount` + '.dc ' + projectName + '\_' + `cpuCount` + '.ill\n' + \

'\n'

**elif** type == 0:

# conceptual dynamic shading

sensors = shadingRecipe.sensorPts

dynamicShd ='shading ' + **str**(type) + '\n' + \

name + '\_' + **str**(recipeCount+1) + ' ' + projectName + '\_' + `cpuCount` + '.dc ' + projectName + '\_' + `cpuCount` + '\_up.ill\n' + \

projectName + '\_' + `cpuCount` + '\_down.ill\n\n'

**elif** type == 2:

dynamicCounter += 1

dynamicShd = ""

# advanced dynamic shading

glareControlRecipe = shadingRecipe.glareControlR

shadingStates = shadingRecipe.shadingStates

stateCount = **len**(**tuple**(s **for** s **in** shadingStates **if** s **is not None**))

controlSystem = shadingRecipe.controlSystem

# sensors = shadingRecipe.sensorPts #sensors are removed from this part and will be added later for the analysis

coolingPeriod = shadingRecipe.coolingPeriod

# add the heading for the first dynamic shading group

**if** dynamicCounter == 1: dynamicShd = dynamicShdHeading

groupName = name

**if** controlSystem == "ManualControl":

dynamicShd += groupName + '\n' + \

**str**(stateCount) + '\n' + \

"ManualControl " + subWorkingDir + "\\" + groupName + "\_state\_1.rad\n"

**for** stateCount **in range**(1, **len**(shadingStates)):

dynamicShd += subWorkingDir + "\\" + groupName + "\_state\_" + **str**(stateCount + 1) + ".rad " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".dc " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".ill\n"

**elif** controlSystem == "AutomatedThermalControl":

**if** glareControlRecipe!=**None**:

controlSystem = "AutomatedGlareControl"

threshold = glareControlRecipe.threshold

minAz = glareControlRecipe.minAz

maxAz = glareControlRecipe.maxAz

minAlt = glareControlRecipe.minAltitude

maxAlt = glareControlRecipe.maxAltitude

**if len**(coolingPeriod)!=0:

stMonth, stDay, hour = coolingPeriod[0]

endMonth, endDay, hour = coolingPeriod[1]

controlSystem += "WithOccupancy"

**if** controlSystem == "AutomatedThermalControl":

dynamicShd += groupName + '\n' + \

**str**(stateCount) + '\n' + \

"AutomatedThermalControl " + subWorkingDir + "\\" + groupName + "\_state\_1.rad\n"

**for** stateCount, shadingState **in enumerate**(shadingStates):

**try**:

dynamicShd += `**int**(shadingState.minIlluminance)` + " " + `**int**(shadingState.maxIlluminance)` + " " + \

subWorkingDir + "\\" + groupName + "\_state\_" + **str**(stateCount + 1) + ".rad " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".dc " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".ill\n"

**except**:

# empty shading states

pass

**elif** controlSystem == "AutomatedThermalControlWithOccupancy":

dynamicShd += groupName + '\n' + \

**str**(stateCount) + '\n' + \

"AutomatedThermalControlWithOccupancy " + \

`stMonth` + " " + `stDay` + " " + `endMonth` + " " + `endDay` + " " + \

subWorkingDir + "\\" + groupName + "\_state\_1.rad\n"

**for** stateCount, shadingState **in enumerate**(shadingStates):

**try**:

dynamicShd += `**int**(shadingState.minIlluminance)` + " " + `**int**(shadingState.maxIlluminance)` + " " + \

subWorkingDir + "\\" + groupName + "\_state\_" + **str**(stateCount + 1) + ".rad " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".dc " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".ill\n"

**except**:

pass

**elif** controlSystem == "AutomatedGlareControl":

dynamicShd += groupName + '\n' + \

**str**(stateCount) + '\n' + \

"AutomatedGlareControl \n" + \

`**int**(threshold)` + " " + `**int**(minAz)` + " " + `**int**(maxAz)` + " " + \

`**int**(minAlt)` + " " + `**int**(maxAlt)` + " " + subWorkingDir + "\\" + groupName + "\_state\_1.rad\n"

**for** stateCount, shadingState **in enumerate**(shadingStates):

**try**:

dynamicShd += `**int**(shadingState.minIlluminance)` + " " + `**int**(shadingState.maxIlluminance)` + " " + \

subWorkingDir + "\\" + groupName + "\_state\_" + **str**(stateCount + 1) + ".rad " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".dc " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".ill\n"

**except**:

pass

**elif** controlSystem == "AutomatedGlareControlWithOccupancy":

dynamicShd += groupName + '\n' + \

**str**(stateCount) + '\n' + \

"AutomatedGlareControlWithOccupancy \n" + \

`**int**(threshold)` + " " + `**int**(minAz)` + " " + `**int**(maxAz)` + " " + \

`**int**(minAlt)` + " " + `**int**(maxAlt)` + "\n" + \

`stMonth` + " " + `stDay` + " " + `endMonth` + " " + `endDay` + " " + \

subWorkingDir + "\\" + groupName + "\_state\_1.rad\n"

**for** stateCount, shadingState **in enumerate**(shadingStates):

**try**:

dynamicShd += `**int**(shadingState.minIlluminance)` + " " + `**int**(shadingState.maxIlluminance)` + " " + \

subWorkingDir + "\\" + groupName + "\_state\_" + **str**(stateCount + 1) + ".rad " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".dc " + \

groupName + "\_state\_" + **str**(stateCount + 1) + '\_' + `cpuCount` + ".ill\n"

**except**:

pass

dynOptStr += dynamicShd

# I removed the sensor point from here as it wasn't really nessecay to

# apply it here and it was also

#sensorInfoStr = 'sensor\_file\_info'

#if type == 0 or type == 2:

# for pt in testPts:

# if self.isSensor(pt, sensors):

# sensorInfoStr += ' BG' + str(recipeCount+1)

# # if BG1\_Ext

# # add external sensor\_ This should happen inside the loop for each group

# # as the point maybe part of multiple shading groups

# else:

# sensorInfoStr += ' 0'

#

#else:

# for pt in testPts: sensorInfoStr += ' 0'

#

#dynOptStr += sensorInfoStr

#'==========================\n' + \

#'= electric lighting system\n' + \

#'==========================\n' + \

#'electric\_lighting\_system 2\n' + \

#' 4 manual\_dimming 100 1 0.0 20 300\n' + \

#' 1 manual\_control 200 1\n' + \

#'\n' + \

#'sensor\_file\_info '

#for pt in range(lenOfPts[cpuCount]): dynOptStr = dynOptStr + '0 '

return dynOptStr + '\n'

**def resultStr**(self, projectName, cpuCount):

return '\n\n######################\n' + \

'# daylighting results \n' + \

'######################\n' + \

'daylight\_autonomy\_active\_RGB ' + projectName + '\_' + `cpuCount` + '\_autonomy.DA\n' + \

'electric\_lighting ' + projectName + '\_' + `cpuCount` + '\_electriclighting.htm\n' + \

'direct\_sunlight\_file ' + projectName + '\_' + `cpuCount` + '.dir\n' + \

'thermal\_simulation ' + projectName + '\_' + `cpuCount` + '\_intgain.csv\n'

**class hb\_ReadAnnualResultsAux**(object):

**def sortIllFiles**(self, illFilesTemp):

"""

This function sorts a list of \*.ill for an annual study

and put them in different branches based on shading groups and blind states

---------------------------------------------------------------------------

{0}

.ill files with no dynamic blinds. When there is no dynamic blinds or when

there are advanced dynamic blind these files should look like:

workingDir + ProjectName + "\_" + CPUCount + ".ill"

and should be sorted based on CPUCount.

In case of conceptualBlinds the files will look like:

workingDir + ProjectName + "\_" + CPUCount + "\_up.ill"

{1,0}

Branches with two numbers contain .ill files for shading groups with different

states. First number represents the shading group (which starts from 1) and

second number represents the state. For instance {1,0} includes .ill files

for first shading group and the first state of the blinds which is the most

open state. In case of simple blinds the file should look like:

workingDir + ProjectName + "\_" + CPUCount + "\_down.ill"

for advanced dynamic blinds the file should looks like:

workingDir + ProjectName + "\_" shadingGroupName + "\_state\_" + stateCount+ "\_" + CPUCount + ".ill"

"""

# check if there are multiple ill files in the folder for different shading groups

illFilesDict = {}

**for** fullPath **in** illFilesTemp:

fileName = os.path.**basename**(fullPath)

**if** fileName.**split**("\_")[:-1]!= []:

**if** fileName.**endswith**("\_down.ill") **or** fileName.**endswith**("\_up.ill"):

# conceptual blind

stateName = "\_".**join**(fileName.**split**("\_")[:-2]) + "\_" + fileName.**split**("\_")[-1]

**if** fileName.**endswith**("\_up.ill"):

groupName = -1

stateName = "up"

stateNumber = 0

**else**:

groupName = "conceptualBlinds"

stateName = "down"

stateNumber = 0

**elif** fileName.**Contains**("\_state\_"):

# dynamic blinds with several states

groupName = "\_".**join**(fileName.**split**("\_")[:-3])

stateName = "\_".**join**(fileName.**split**("\_")[-3:-1])

stateNumber = fileName.**split**("\_")[-2]

**else**:

groupName = -1

stateName = "\_".**join**(fileName.**split**("\_")[:-1])

stateNumber = -1 # no states

**else**:

groupName = -1

stateName = fileName

stateNumber = -1 # no states

# create an empty dictionary

**if** groupName **not in** illFilesDict.**keys**():

illFilesDict[groupName] = {}

# create an empty dictionary for each state

**if** stateName **not in** illFilesDict[groupName].**keys**():

illFilesDict[groupName][stateName] = []

# append the file to the list

illFilesDict[groupName][stateName].**append**(fullPath)

# sort the keys

illFiles = DataTree[System.Object**]**()

shadingGroupCount = 0

**for** key, fileListDict **in** illFilesDict.**items**():

stateCount = 0

shadingGroupCount+=1

**for** state, fileList **in** fileListDict.**items**():

**if** key== -1:

p = **GH\_Path**(0)

shadingGroupCount-=1

**else**:

p = **GH\_Path**(shadingGroupCount, stateCount)

stateCount+=1

**try**:

**if** fileName.**endswith**("\_down.ill") **or** fileName.**endswith**("\_up.ill"):

# conceptual blind

**if** fileList[0].**endswith**("\_down.ill"):

p = **GH\_Path**(1,0)

**else**:

p = **GH\_Path**(0)

illFiles.**AddRange**(**sorted**(fileList, key=**lambda** fileName: **int**(fileName.**split**(".")[-2].**split**("\_")[-2])), p)

**else**:

illFiles.**AddRange**(**sorted**(fileList, key=**lambda** fileName: **int**(fileName.**split**(".")[-2].**split**("\_")[-1])), p)

**except** Exception, e:

# failed to sort!

illFiles.**AddRange**(fileList, p)

return illFiles

**class hb\_EnergySimulatioParameters**(object):

**def readEPParams**(self, EPParameters):

**if** EPParameters == []:

timestep = 6

shadowPar = ["AverageOverDaysInFrequency", 30, 3000]

solarDistribution = "FullInteriorAndExteriorWithReflections"

simulationControl = [**True**, **True**, **True**, **False**, **True**, '', '']

ddyFile = **None**

terrain = 'City'

grndTemps = []

holidays = []

startDay = **None**

heatSizing = 1.25

coolSizing = 1.15

**else**:

timestep = **int**(EPParameters[0])

shadowPar = EPParameters[1:4]

solarDistribution = EPParameters[4]

simulationControl = EPParameters[5:12]

ddyFile = EPParameters[12]

terrain = EPParameters[13]

grndTemps = EPParameters[14]

holidays = EPParameters[15]

startDay = EPParameters[16]

heatSizing = EPParameters[17]

coolSizing = EPParameters[18]

return timestep, shadowPar, solarDistribution, simulationControl, ddyFile, terrain, grndTemps, holidays, startDay, heatSizing, coolSizing

**class EPMaterialAux**(object):

**def \_\_init\_\_**(self):

self.energyModelingStandards = {"0" : "ASHRAE 90.1-2004",

"1" : "ASHRAE 90.1-2007",

"2" : "ASHRAE 90.1-2010",

"3" : "ASHRAE 189.1",

"4" : "CBECS 1980-2004",

"5" : "CBECS Before-1980",

"ASHRAE9012004" : "ASHRAE 90.1-2004",

"ASHRAE9012007" : "ASHRAE 90.1-2007",

"ASHRAE9012010" : "ASHRAE 90.1-2010",

"ASHRAE1891" : "ASHRAE 189.1",

"CBECS19802004" : "CBECS 1980-2004",

"CBECSBEFORE1980" : "CBECS Before-1980"}

**def calcEPMaterialUValue**(self, materialObj, GHComponent = **None**):

# Dictionary of typical U-Values for different gases.

# All of these materials are taken from LBNL WINDOW 7.4 Gas Library assuming a 1 cm-thick gap.

gasUVal = {

"air": 2.407,

"argon": 1.6348,

"krypton": 0.8663,

"xenon": 0.516

}

materialType = materialObj[0]

**if** materialType.**lower**() == "windowmaterial:simpleglazingsystem":

UValueSI = **float**(materialObj[1][0])

**elif** materialType.**lower**() == "windowmaterial:glazing":

thickness = **float**(materialObj[3][0])

conductivity = **float**(materialObj[13][0])

UValueSI = conductivity/thickness

**elif** materialType.**lower**() == "windowmaterial:blind":

UValueSI = 2.407

**elif** materialType.**lower**() == "windowmaterial:shade":

UValueSI = 2.407

**elif** materialType.**lower**() == "material:nomass":

# Material:NoMass is defined by R-Value and not U-Value

UValueSI = 1 / **float**(materialObj[2][0])

**elif** materialType.**lower**() == "material":

thickness = **float**(materialObj[2][0])

conductivity = **float**(materialObj[3][0])

UValueSI = conductivity/thickness

**elif** materialType.**lower**() == "material:roofvegetation":

thickness = **float**(materialObj[8][0])

conductivity = **float**(materialObj[9][0])

UValueSI = conductivity/thickness

**elif** materialType.**lower**() == "material:airgap":

UValueSI = 1 / **float**(materialObj[1][0])

**elif** materialType.**lower**() == "windowmaterial:gas":

thickness = **float**(materialObj[2][0])

**if** thickness > 0.05:

warningMsg = "The thickness of your gas layer is beyond that typically seen in windows." + "\n" + \

"The U-Value calculated here might be fairly different from what E+ will use."

**if** GHComponent!=**None**:

w = gh.GH\_RuntimeMessageLevel.Warning

GHComponent.**AddRuntimeMessage**(w, warningMsg)

**try**:

UValueSI = gasUVal[materialObj[1][0].**lower**()]

**except**:

UValueSI = -1

warningMsg = "Honeybee can't calculate the UValue for " + materialObj[1][0] + ".\n" + \

"Let us know if you think it is really neccesary and we will add it to the list. :)"

**if** GHComponent!=**None**:

w = gh.GH\_RuntimeMessageLevel.Warning

GHComponent.**AddRuntimeMessage**(w, warningMsg)

**elif** materialType.**lower**() == "windowmaterial:gasmixture":

thickness = **float**(materialObj[1][0])

**if** thickness > 0.05:

warningMsg = "The thickness of your gas layer is beyond that typically seen in windows." + "\n" + \

"The U-Value calculated here might be fairly different from what E+ will use."

**if** GHComponent!=**None**:

w = gh.GH\_RuntimeMessageLevel.Warning

GHComponent.**AddRuntimeMessage**(w, warningMsg)

**try**:

UValueSI = 0

gas = 0

gasPercent = 0

**for** gasCount **in range**(3, **len**(materialObj)):

**if** (gasCount % 2 == 0):

gasPercent = **float**(materialObj[gasCount][0])

UValueSI = UValueSI **+** (gas\*gasPercent)

**else**:

gas = **float**(gasUVal[materialObj[gasCount][0].**lower**()])

**except**:

UValueSI = -1

warningMsg = "Honeybee can't calculate the UValue for " + materialObj[1][0] + ".\n" + \

"Let us know if you think it is really neccesary and we will add it to the list. :)"

**if** GHComponent!=**None**:

w = gh.GH\_RuntimeMessageLevel.Warning

GHComponent.**AddRuntimeMessage**(w, warningMsg)

**else**:

warningMsg = "Honeybee currently can't calculate U-Values for " + materialType + ".\n" +\

"Your Honeybee EnergyPlus simulations will still run fine with this material and this is only a Honeybee interface limitation." + ".\n" +\

"Let us know if you think HB should calcualte this material type and we will add it to the list. :)"

**if** GHComponent!=**None**:

w = gh.GH\_RuntimeMessageLevel.Warning

GHComponent.**AddRuntimeMessage**(w, warningMsg)

# http://bigladdersoftware.com/epx/docs/8-0/input-output-reference/page-010.html

UValueSI = -1

return UValueSI

**def calcEPConstructionUValue**(self, constructionObj, GHComponent=**None**):

# find material layers

uValues = []

**for** layer **in** constructionObj.**keys**()[1:]:

materialName, comment = constructionObj[layer]

**try**: values, comments, UValueSI, UValueIP = self.**decomposeMaterial**(materialName, GHComponent)

**except**: UValueSI = -1

uValues.**append**(UValueSI)

# calculate cumulative UValue

totalRValue = 0

**for** uValue **in** uValues:

totalRValue += 1/uValue

return 1/totalRValue

**def convertUValueToIP**(self, UValueSI):

return 0.176110 \* UValueSI

**def convertUValueToSI**(self, UValueIP):

return 5.678263 \* UValueIP

**def decomposeMaterial**(self, matName, GHComponent = **None**):

**try**:

thermTrigger = **False**

**try**:

materialObj = sc.sticky["honeybee\_materialLib"][matName.**upper**()]

**except**:

**try**:

materialObj = sc.sticky["honeybee\_windowMaterialLib"][matName.**upper**()]

**except**:

materialObj = sc.sticky["honeybee\_thermMaterialLib"][matName.**upper**()]

thermTrigger = **True**

comments = []

values = []

UValueSI, UValueIP = **None**, **None**

**if** thermTrigger == **False**:

**for** layer **in** materialObj.**keys**():

**try**:

value, comment = materialObj[layer]

# print value + ',\t!-' + comment + "\n"

values.**append**(value)

comments.**append**(comment)

**except**:

value = materialObj[layer]

values.**append**(value)

comments.**append**('Material Type')

UValueSI = self.**calcEPMaterialUValue**(materialObj, GHComponent)

UValueIP = self.**convertUValueToIP**(UValueSI)

**else**:

**for** layer **in** materialObj.**keys**():

**if** layer == 'Name': pass

**else**:

comments.**append**(layer)

values.**append**(materialObj[layer])

return values, comments, UValueSI, UValueIP

**except** Exception, e:

**print** `e`

return -1

**def decomposeEPCnstr**(self, cnstrName, GHComponent = **None**):

**try**:

constructionObj = sc.sticky ["honeybee\_constructionLib"][cnstrName.**upper**()]

comments = []

materials = []

# print cnstrName

**for** layer **in** constructionObj.**keys**():

**try**:

material, comment = constructionObj[layer]

materials.**append**(material)

comments.**append**(comment)

**except**:

material = constructionObj[layer]

materials.**append**(material)

comments.**append**("!- Material Type")

# place holder

UValue\_SI = self.**calcEPConstructionUValue**(constructionObj, GHComponent)

UValue\_IP = self.**convertUValueToIP**(UValue\_SI)

return materials[1:], comments[1:], UValue\_SI, UValue\_IP

**except** Exception, e:

**print** `e`

**print** "Failed to find " + cnstrName + " in the Honeybee construction library."

return -1

**def searchListByKeyword**(self, inputList, keywords):

""" search inside a list of strings for keywords """

**def checkMultipleKeywords**(name, keywordlist):

**for** kw **in** keywordlist:

**if** name.**find**(kw)== -1:

return **False**

return **True**

kWords = []

**for** kw **in** keywords:

kWords.**append**(kw.**strip**().**upper**().**split**(" "))

selectedItems = []

alternateOptions = []

**for** item **in** inputList:

**if len**(kWords)!= 0 **and not** "\*" **in** keywords:

**for** keyword **in** kWords:

**if len**(keyword) > 1 **and checkMultipleKeywords**(item.**ToUpper**(), keyword):

selectedItems.**append**(item)

**elif len**(keyword) == 1 **and** item.**ToUpper**().**find**(keyword[0])!= -1:

selectedItems.**append**(item)

**else**:

selectedItems.**append**(item)

return selectedItems

**def filterMaterials**(self, constrList, standard, climateZone, surfaceType, bldgProgram, keywords, sourceComponent):

hb\_EPTypes = **EPTypes**()

w = gh.GH\_RuntimeMessageLevel.Warning

**try**:

standard = **str**(standard).**upper**().**Replace**(" ", "").**Replace**("-", "").**Replace**(".", "")

standard = self.energyModelingStandards[standard]

**except**:

msg = "The input for standard is not valid. Standard is set to ASHRAE 90.1"

sourceComponent.**AddRuntimeMessage**(w, msg)

standard = "ASHRAE 90.1"

selConstr =[]

filtConstr =self.**searchListByKeyword**(constrList, keywords)

**for** cnstrName **in** filtConstr:

**if** cnstrName.**upper**().**find**(standard.**upper**())!=-1 **and** cnstrName.**upper**().**find**(surfaceType.**upper**())!=-1:

# check for climate zone

**if** climateZone!="":

clmZones = []

# split by space " "

possibleAlt, zoneCode = cnstrName.**split**(" ")[-2:]

clmZoneList = zoneCode.**split**("-")

**if len**(clmZoneList) != 1:

**try**:

clmZoneRange = **range**(**int**(clmZoneList[0]), **int**(clmZoneList[1]) + 1)

**for** clmZone **in** clmZoneRange: clmZones.**append**(**str**(clmZone))

**except**:

clmZones = [clmZoneList[0], clmZoneList[1]]

**else**:

clmZones = clmZoneList

**if** climateZone **in** clmZones:

selConstr.**append**(cnstrName)

**elif** climateZone[0] **in** clmZones:

# cases like 3a that is included in 3

selConstr.**append**(cnstrName)

**else**:

selConstr.**append**(cnstrName)

return selConstr

**def isEPMaterialObjectAlreadyExists**(self, name):

"""

Check if material or construction exist

"""

**if** name **in** sc.sticky ["honeybee\_constructionLib"].**keys**(): return **True**

**if** name **in** sc.sticky ["honeybee\_materialLib"].**keys**(): return **True**

**if** name **in** sc.sticky ["honeybee\_windowMaterialLib"].**keys**(): return **True**

return **False**

**def getEPObjectsStr**(self, objectName):

"""

This function should work for materials, and counstructions

"""

objectData = **None**

**if** objectName **in** sc.sticky ["honeybee\_windowMaterialLib"].**keys**():

objectData = sc.sticky ["honeybee\_windowMaterialLib"][objectName]

**elif** objectName **in** sc.sticky ["honeybee\_materialLib"].**keys**():

objectData = sc.sticky ["honeybee\_materialLib"][objectName]

**elif** objectName **in** sc.sticky ["honeybee\_constructionLib"].**keys**():

objectData = sc.sticky ["honeybee\_constructionLib"][objectName]

**elif** objectData **in** sc.sticky["honeybee\_WindowPropLib"].**keys**():

objectData = sc.sticky["honeybee\_WindowPropLib"][objectName]

**elif** objectName **in** sc.sticky["honeybee\_SpectralDataLib"].**keys**():

objectData = sc.sticky["honeybee\_SpectralDataLib"][objectName]

**if** objectData!=**None**:

numberOfLayers = **len**(objectData.**keys**())

# add material/construction type

# print objectData

objectStr = objectData[0] + ",\n"

# add the name

objectStr = objectStr + " " + objectName + ", !- name\n"

**for** layer **in range**(1, numberOfLayers):

**if** layer < numberOfLayers-1:

objectStr = objectStr + " " + **str**(objectData[layer][0]) + ", !- " + objectData[layer][1] + "\n"

**else**:

objectStr = objectStr + " " + **str**(objectData[layer][0]) + "; !- " + objectData[layer][1] + "\n\n"

return objectStr

**def getObjectKey**(self, EPObject):

EPKeys = ["Material", "WindowMaterial", "Construction"]

# check if it is a full string

**for** key **in** EPKeys:

**if** EPObject.**strip**().**startswith**(key):

return key

**def addEPConstructionToLib**(self, EPMaterial, overwrite = **False**):

key = self.**getObjectKey**(EPMaterial)

**if** key == **None**:

return **None**, **None**

HBLibrarieNames = {

"Construction" : "honeybee\_constructionLib",

"Material" : "honeybee\_materialLib",

"WindowMaterial" : "honeybee\_windowMaterialLib"

}

# find construction/material name

name = EPMaterial.**split**("\n")[1].**split**("!")[0].**strip**()[:-1].**upper**()

**if** name **in** sc.sticky[HBLibrarieNames[key]].**keys**():

#overwrite = True

**if not** overwrite:

# ask user if they want to overwrite it

add = self.**duplicateEPMaterialWarning**(name, EPMaterial)

**if not** add: return **False**, name

# add material/construction to the lib

# create an empty dictoinary for the material

sc.sticky[HBLibrarieNames[key]][name] = {}

lines = EPMaterial.**split**("\n")

# store the data into the dictionary

**for** lineCount, line **in enumerate**(lines):

objValue = line.**split**("!")[0].**strip**()

**try**: objDescription = line.**split**("!")[1].**strip**()

**except**: objDescription = ""

**if** lineCount == 0:

sc.sticky[HBLibrarieNames[key]][name][lineCount] = objValue[:-1]

**elif** lineCount == 1:

pass # name is already there as the key

**elif** objValue.**endswith**(","):

sc.sticky[HBLibrarieNames[key]][name][lineCount-1] = objValue[:-1], objDescription

**elif** objValue.**endswith**(";"):

sc.sticky[HBLibrarieNames[key]][name][lineCount-1] = objValue[:-1], objDescription

break

# add name to list

# sc.sticky [HBLibrarieNames[key]]["List"].append(name)

return **True**, name

**def duplicateEPMaterialWarning**(self, objectName, newMaterialString):

# this function is duplicate with duplicateEPObject warning and should be removed at some point

returnYN = {'YES': **True**, 'NO': **False**}

buttons = System.Windows.Forms.MessageBoxButtons.YesNo

icon = System.Windows.Forms.MessageBoxIcon.Warning

currentMaterialString = self.**getEPObjectsStr**(objectName)

msg = objectName + " already exists in the library:\n\n" + \

currentMaterialString + "\n" + \

"Do you want to overwrite the current with this new definition?\n\n" + \

newMaterialString + "\n\n" + \

"Tip: If you are not sure what to do select No and change the name."

up = rc.UI.Dialogs.**ShowMessageBox**(msg, "Duplicate Material Name", buttons, icon)

return returnYN[up.**ToString**().**ToUpper**()]

**class EPScheduleAux**(object):

**def getScheduleDataByName**(self, schName, component = **None**):

**if** schName.**lower**().**endswith**(".csv"):

# Check for the file

**if not** os.path.**isfile**(schName):

msg = "Failed to find the schedule file: " + schName

**print** msg

**if** component **is not None**:

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return **None**, **None**

return schName, "csv"

**try**:

scheduleObj = sc.sticky["honeybee\_ScheduleLib"][schName.**upper**()]

**except** Exception, e:

**if** schName != "NONE":

msg = "Failed to find " + schName + " in the Honeybee schedule library."

**print** msg

**if** component **is not None**:

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return **None**, **None**

comments = []

values = []

**for** layer **in** scheduleObj.**keys**():

**try**:

material, comment = scheduleObj[layer]

values.**append**(material)

comments.**append**(comment)

**except**:

scheduleType = scheduleObj[layer]

values.**append**(scheduleType)

comments.**append**("Schedule Type")

return values, comments

**def getScheduleTypeLimitsDataByName**(self, schName, component = **None**):

**try**:

scheduleObj = sc.sticky["honeybee\_ScheduleTypeLimitsLib"][schName.**upper**()]

**except** Exception, e:

**if** schName != "NONE":

msg = "Failed to find " + schName + " in the Honeybee schedule type limits library."

**print** msg

**if** component **is not None**:

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return **None**, **None**

comments = []

values = []

**for** layer **in** scheduleObj.**keys**():

**try**:

material, comment = scheduleObj[layer]

values.**append**(material)

comments.**append**(comment)

**except**:

scheduleType = scheduleObj[layer]

values.**append**(scheduleType)

comments.**append**("Schedule Type")

return values, comments

**class EPObjectsAux**(object):

**def isEPMaterial**(self, matName):

return matName.**upper**() **in** sc.sticky["honeybee\_materialLib"].**keys**() **or** \

matName.**upper**() **in** sc.sticky["honeybee\_windowMaterialLib"].**keys**()

**def isEPConstruction**(self, matName):

return matName.**upper**() **in** sc.sticky["honeybee\_constructionLib"].**keys**()

**def isSchedule**(self, scheduleName):

return scheduleName.**upper**() **in** sc.sticky["honeybee\_ScheduleLib"].**keys**()

**def isScheduleTypeLimits**(self, scheduleName):

return scheduleName.**upper**() **in** sc.sticky["honeybee\_ScheduleTypeLimitsLib"].**keys**()

**def isWindowProperty**(self, winPropName):

return winPropName.**upper**() **in** sc.sticky["honeybee\_WindowPropLib"].**keys**()

**def isSpectralData**(self, spectName):

return spectName.**upper**() **in** sc.sticky["honeybee\_SpectralDataLib"].**keys**()

**def customizeEPObject**(self, EPObjectName, indexes, inValues):

hb\_EPScheduleAUX = **EPScheduleAux**()

hb\_EPMaterialAUX = **EPMaterialAux**()

**if** self.**isSchedule**(EPObjectName):

values, comments = hb\_EPScheduleAUX.**getScheduleDataByName**(EPObjectName.**upper**())

**elif** self.**isScheduleTypeLimits**(EPObjectName):

values, comments = hb\_EPScheduleAUX.**getScheduleTypeLimitsDataByName**(EPObjectName.**upper**())

**elif** self.**isEPConstruction**(EPObjectName):

values, comments, uSI, uIP = hb\_EPMaterialAUX.**decomposeEPCnstr**(EPObjectName.**upper**())

**elif** self.**isEPMaterial**(EPObjectName):

values, comments, uSI, uIP = hb\_EPMaterialAUX.**decomposeMaterial**(EPObjectName.**upper**())

**else**:

return

# create a dictionary of index and values

**if len**(indexes)==0 **or** (**len**(indexes) != **len**(inValues)):

return

valuesDict = {}

**for** i, v **in zip**(indexes, inValues):

valuesDict[i] = v

count = 0

originalObj = ""

modifiedObj = ""

**for** value, comment **in zip**(values, comments):

**if** count == **len**(values):

separator = ";"

**else**:

separator = ","

**if** count == 1:

# add name

originalObj += "[" + `count` + "]\t" + EPObjectName.**upper**() + " ! Name\n"

**if** count **in** valuesDict.**keys**():

# update the value

modifiedObj += valuesDict[count].**upper**() + separator + " ! Name\n"

**else**:

# keep original

modifiedObj += EPObjectName.**upper**() + separator + " ! Name\n"

count = 2

originalObj += "[" + `count` + "]\t " + value + " !" + comment + "\n"

**if** count **in** valuesDict.**keys**():

modifiedObj += valuesDict[count] + separator + " !" + comment + "\n"

**else**:

modifiedObj += value + separator + " !" + comment + "\n"

count += 1

return originalObj, modifiedObj

**def getObjectKey**(self, EPObject):

EPKeys = ["Material", "WindowMaterial", "Construction", "ScheduleTypeLimits", "Schedule", "WindowProperty", "MaterialProperty:GlazingSpectralData"]

# check if it is a full string

**for** key **in** EPKeys:

**if** EPObject.**strip**().**startswith**(key):

return key

**def addEPObjectToLib**(self, EPObject, overwrite = **False**):

key = self.**getObjectKey**(EPObject)

**if** key == **None**:

return **None**, **None**

HBLibrarieNames = {

"Construction" : "honeybee\_constructionLib",

"Material" : "honeybee\_materialLib",

"WindowMaterial" : "honeybee\_windowMaterialLib",

"Schedule": "honeybee\_ScheduleLib",

"ScheduleTypeLimits" : "honeybee\_ScheduleTypeLimitsLib",

"WindowProperty" : "honeybee\_WindowPropLib",

"MaterialProperty:GlazingSpectralData" : "honeybee\_SpectralDataLib"

}

# find construction/material name

name = EPObject.**split**("\n")[1].**split**("!")[0].**strip**()[:-1].**upper**()

**if** name **in** sc.sticky[HBLibrarieNames[key]].**keys**():

#overwrite = True

**if not** overwrite:

# ask user if they want to overwrite it

add = self.**duplicateEPObjectWarning**(name, EPObject)

**if not** add: return **False**, name

# add material/construction to the lib

# create an empty dictoinary for the material

sc.sticky[HBLibrarieNames[key]][name] = {}

lines = EPObject.**split**("\n")

# store the data into the dictionary

**for** lineCount, line **in enumerate**(lines):

objValue = line.**split**("!")[0].**strip**()

**try**: objDescription = line.**split**("!")[1].**strip**()

**except**: objDescription = ""

**if** lineCount == 0:

sc.sticky[HBLibrarieNames[key]][name][lineCount] = objValue[:-1]

**elif** lineCount == 1:

pass # name is already there as the key

**elif** objValue.**endswith**(","):

sc.sticky[HBLibrarieNames[key]][name][lineCount-1] = objValue[:-1], objDescription

**elif** objValue.**endswith**(";"):

sc.sticky[HBLibrarieNames[key]][name][lineCount-1] = objValue[:-1], objDescription

break

# add name to list

#sc.sticky [HBLibrarieNames[key]]["List"].append(name)

return **True**, name

**def getEPObjectDataByName**(self, objectName):

objectData = **None**

objectName = objectName.**upper**()

**if** objectName **in** sc.sticky ["honeybee\_windowMaterialLib"].**keys**():

objectData = sc.sticky ["honeybee\_windowMaterialLib"][objectName]

**elif** objectName **in** sc.sticky ["honeybee\_materialLib"].**keys**():

objectData = sc.sticky ["honeybee\_materialLib"][objectName]

**elif** objectName **in** sc.sticky ["honeybee\_constructionLib"].**keys**():

objectData = sc.sticky ["honeybee\_constructionLib"][objectName]

**elif** objectName **in** sc.sticky["honeybee\_ScheduleLib"].**keys**():

objectData = sc.sticky ["honeybee\_ScheduleLib"][objectName]

**elif** objectName **in** sc.sticky["honeybee\_ScheduleTypeLimitsLib"].**keys**():

objectData = sc.sticky ["honeybee\_ScheduleTypeLimitsLib"][objectName]

**elif** objectName **in** sc.sticky["honeybee\_WindowPropLib"].**keys**():

objectData = sc.sticky["honeybee\_WindowPropLib"][objectName]

**elif** objectName **in** sc.sticky["honeybee\_SpectralDataLib"].**keys**():

objectData = sc.sticky["honeybee\_SpectralDataLib"][objectName]

return objectData

**def getEPObjectsStr**(self, objectName):

"""

This function should work for materials, and counstructions

"""

objectData = self.**getEPObjectDataByName**(objectName)

**if** objectData!=**None**:

numberOfLayers = **len**(objectData.**keys**())

# add material/construction type

# print objectData

objectStr = objectData[0] + ",\n"

# add the name

objectStr = objectStr + " " + objectName + ", !- name\n"

**for** layer **in range**(1, numberOfLayers):

**if** layer < numberOfLayers-1:

objectStr = objectStr + " " + **str**(objectData[layer][0]) + ", !- " + objectData[layer][1] + "\n"

**else**:

objectStr = objectStr + " " + **str**(objectData[layer][0]) + "; !- " + objectData[layer][1] + "\n\n"

return objectStr

**def duplicateEPObjectWarning**(self, objectName, newMaterialString):

returnYN = {'YES': **True**, 'NO': **False**}

buttons = System.Windows.Forms.MessageBoxButtons.YesNo

icon = System.Windows.Forms.MessageBoxIcon.Warning

currentMaterialString = self.**getEPObjectsStr**(objectName)

msg = objectName + " already exists in the library:\n\n" + \

currentMaterialString + "\n" + \

"Do you want to overwrite the current with this new definition?\n\n" + \

newMaterialString + "\n\n" + \

"Tip: If you are not sure what to do select No and change the name."

up = rc.UI.Dialogs.**ShowMessageBox**(msg, "Duplicate Material Name", buttons, icon)

return returnYN[up.**ToString**().**ToUpper**()]

**def assignEPConstruction**(self, HBSrf, EPConstruction, component):

**if not** EPConstruction: return

# if it is just the name of the material make sure it is already defined

**if len**(EPConstruction.**split**("\n")) == 1:

# if the material is not in the library add it to the library

**if not** self.**isEPConstruction**(EPConstruction):

warningMsg = "Can't find " + EPConstruction + " in EP Construction Library.\n" + \

"Add the construction to the library and try again."

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warningMsg)

return

**else**:

# it is a full string

added, EPConstruction = self.**addEPObjectToLib**(EPConstruction, overwrite = **True**)

**if not** added:

msg = name + " cannot be added to the project library! Make sure it is an standard EP construction."

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

**print** msg

return

**try**:

HBSrf.**setEPConstruction**(EPConstruction)

**except** Exception, e:

warningMsg = "Failed to assign new EPConstruction to " + HBSrf.name + "."

**print** warningMsg

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warningMsg)

return

**class ReadEPSchedules**(object):

**def \_\_init\_\_**(self, schName, startDayOfTheWeek):

self.hb\_EPScheduleAUX = sc.sticky["honeybee\_EPScheduleAUX"**]**()

self.hb\_EPObjectsAUX = sc.sticky["honeybee\_EPObjectsAUX"**]**()

self.lb\_preparation = sc.sticky["ladybug\_Preparation"**]**()

self.schName = schName

self.startDayOfTheWeek = startDayOfTheWeek

self.count = 0

self.startHOY = 1

self.endHOY = 24

self.unit = "unknown"

self.comapctKeywords = ['Weekdays', 'Weekends', 'Alldays', 'AllOtherDays', 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

**def getScheduleTypeLimitsData**(self, schName):

**if** schName == **None**: schName = self.schName

schedule, comments = self.hb\_EPScheduleAUX.**getScheduleTypeLimitsDataByName**(schName.**upper**(), ghenv.Component)

**try**:

lowerLimit, upperLimit, numericType, unitType = schedule[1:]

**except**:

lowerLimit, upperLimit, numericType = schedule[1:]

unitType = "unknown"

self.unit = unitType

**if** self.unit == "unknown":

self.unit = numericType

return lowerLimit, upperLimit, numericType, unitType

**def getDayEPScheduleValues**(self, schName = **None**):

**if** schName == **None**:

schName = self.schName

values, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(schName.**upper**(), ghenv.Component)

typeLimitName = values[1]

lowerLimit, upperLimit, numericType, unitType = \

self.**getScheduleTypeLimitsData**(typeLimitName)

numberOfDaySch = **int(**(**len**(values) - 3) /2)

hourlyValues = **range**(24)

startHour = 0

**for** i **in range**(numberOfDaySch):

value = **float**(values[2 \* i + 4])

untilTime = **map**(int, values[2 \* i + 3].**split**(":"))

endHour = **int**(untilTime[0] + untilTime[1]/60)

**for** hour **in range**(startHour, endHour):

hourlyValues[hour] = value

startHour = endHour

**if** numericType.**strip**().**lower**() == "district":

hourlyValues = **map**(int, hourlyValues)

return hourlyValues

**def getWeeklyEPScheduleValues**(self, schName = **None**):

"""

Schedule:Week:Daily

['Schedule Type', 'Sunday Schedule:Day Name', 'Monday Schedule:Day Name',

'Tuesday Schedule:Day Name', 'Wednesday Schedule:Day Name', 'Thursday Schedule:Day Name',

'Friday Schedule:Day Name', 'Saturday Schedule:Day Name', 'Holiday Schedule:Day Name',

'SummerDesignDay Schedule:Day Name', 'WinterDesignDay Schedule:Day Name',

'CustomDay1 Schedule:Day Name', 'CustomDay2 Schedule:Day Name']

"""

**if** schName == **None**:

schName = self.schName

values, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(schName.**upper**(), ghenv.Component)

**if** self.count == 1:

# set the last date of the schedule to one week

self.endHOY = 24 \* 7

sundaySchedule = self.**getScheduleValues**(values[1])

mondaySchedule = self.**getScheduleValues**(values[2])

tuesdaySchedule = self.**getScheduleValues**(values[3])

wednesdaySchedule = self.**getScheduleValues**(values[4])

thursdaySchedule = self.**getScheduleValues**(values[5])

fridaySchedule = self.**getScheduleValues**(values[6])

saturdaySchedule = self.**getScheduleValues**(values[7])

holidaySchedule = self.**getScheduleValues**(values[8])

summerDesignDaySchedule = self.**getScheduleValues**(values[9])

winterDesignDaySchedule = self.**getScheduleValues**(values[10])

customDay1Schedule = self.**getScheduleValues**(values[11])

customDay2Schedule = self.**getScheduleValues**(values[12])

hourlyValues = [sundaySchedule, mondaySchedule, tuesdaySchedule, \

wednesdaySchedule, thursdaySchedule, fridaySchedule, \

saturdaySchedule]

hourlyValues = hourlyValues[self.startDayOfTheWeek:] + \

hourlyValues[:self.startDayOfTheWeek]

return hourlyValues

**def getConstantEPScheduleValues**(self, schName):

"""

'Schedule:Constant'

['Schedule Type', 'Schedule Type Limits Name', 'Hourly Value']

"""

**if** schName == **None**:

schName = self.schName

values, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(schName.**upper**(), ghenv.Component)

typeLimitName = values[1]

lowerLimit, upperLimit, numericType, unitType = \

self.**getScheduleTypeLimitsData**(typeLimitName)

hourlyValues = [**float**(values[2])]

**if** numericType.**strip**().**lower**() == "district":

hourlyValues = **map**(int, hourlyValues)

return scheduleConstant

**def getCompactEPScheduleValues**(self, schName):

**if** schName == **None**: schName = self.schName

values, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(schName.**upper**(), ghenv.Component)

typeLimitName = values[1]

lowerLimit, upperLimit, numericType, unitType = \

self.**getScheduleTypeLimitsData**(typeLimitName)

#Separate out the different periods.

totalValues = []

periodValues = []

headerDone = **False**

**for** val **in** values:

newPeriod = **False**

**for** word **in** self.comapctKeywords:

**if** word **in** val: newPeriod = **True**

**if** newPeriod == **True**:

**if** headerDone == **True**:

totalValues.**append**(periodValues)

periodValues = [val]

headerDone = **True**

**elif** headerDone == **True**:

periodValues.**append**(val)

totalValues.**append**(periodValues)

#For each day period, construct a day schedule.

dayType = []

dayValues = []

**for** dayVals **in** totalValues:

dayType.**append**(dayVals[0].**title**().**split**('For: ')[-1])

numberOfDaySch = **int(**(**len**(dayVals) - 1) /2)

hourlyValues = **range**(24)

startHour = 0

**for** i **in range**(numberOfDaySch):

value = **float**(dayVals[2 \* i + 2])

untilTime = **map**(int, dayVals[2 \* i + 1].**split**(":")[1:])

endHour = **int**(untilTime[0] + untilTime[1]/60)

**for** hour **in range**(startHour, endHour):

hourlyValues[hour] = value

startHour = endHour

dayValues.**append**(hourlyValues)

#Construct a week schedule from the day schedules.

#Map the dayTypes to the days of the week.

['Weekdays', 'Weekends', 'Alldays', 'AllOtherDays', 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

weekVals = [-1, -1, -1, -1, -1, -1, -1]

**for** typeCount, type **in enumerate**(dayType):

**if** type == 'Alldays':

**for** count, val **in enumerate**(weekVals):

weekVals[count] = typeCount

**elif** type == 'Weekdays':

**for** count, val **in enumerate**(weekVals):

**if** count < 6 **and** count != 0: weekVals[count] = typeCount

**elif** type == 'Weekends':

weekVals[0] = typeCount

weekVals[-1] = typeCount

**elif** type == 'Sunday': weekVals[0] = typeCount

**elif** type == 'Monday': weekVals[1] = typeCount

**elif** type == 'Tuesday': weekVals[2] = typeCount

**elif** type == 'Wednesday': weekVals[3] = typeCount

**elif** type == 'Thursday': weekVals[4] = typeCount

**elif** type == 'Friday': weekVals[5] = typeCount

**elif** type == 'Saturday': weekVals[6] = typeCount

**elif** type == 'Allotherdays':

**for** count, val **in enumerate**(weekVals):

**if** val == -1: weekVals[count] = typeCount

#Turn the week schedule into a year schedule.

hourlyValues = []

dayVals = []

dayofWeek = -1

**for** day **in range**(365):

**if** dayofWeek == 6: dayofWeek = 0

**else**: dayofWeek += 1

dayVals.**append**(weekVals[dayofWeek])

**for** day **in** dayVals:

hourlyValues.**extend**(dayValues[day])

**print** hourlyValues

return hourlyValues

**def getYearlyEPScheduleValues**(self, schName = **None**):

# place holder for 365 days

hourlyValues = **range**(365)

# update last day of schedule

self.endHOY = 8760

**if** schName == **None**:

schName = self.schName

values, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(schName.**upper**(), ghenv.Component)

# generate weekly schedules

numOfWeeklySchedules = **int(**(**len**(values)-2)/5)

**for** i **in range**(numOfWeeklySchedules):

weekDayScheduleName = values[5 \* i + 2]

startDay = **int**(self.lb\_preparation.**getJD**(**int**(values[5 \* i + 3]), **int**(values[5 \* i + 4])))

endDay = **int**(self.lb\_preparation.**getJD**(**int**(values[5 \* i + 5]), **int**(values[5 \* i + 6])))

# 7 list for 7 days of the week

hourlyValuesForTheWeek = self.**getScheduleValues**(weekDayScheduleName)

**for** day **in range**(startDay-1, endDay):

hourlyValues[day] = hourlyValuesForTheWeek[day%7]

return hourlyValues

**def getScheduleValues**(self, schName = **None**):

**if** schName == **None**:

schName = self.schName

**if** self.hb\_EPObjectsAUX.**isSchedule**(schName):

scheduleValues, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(schName.**upper**(), ghenv.Component)

scheduleType = scheduleValues[0].**lower**()

**if** self.count == 0:

self.schType = scheduleType

self.count += 1

**if** scheduleType == "schedule:year":

hourlyValues = self.**getYearlyEPScheduleValues**(schName)

**elif** scheduleType == "schedule:day:interval":

hourlyValues = self.**getDayEPScheduleValues**(schName)

**elif** scheduleType == "schedule:week:daily":

hourlyValues = self.**getWeeklyEPScheduleValues**(schName)

**elif** scheduleType == "schedule:constant":

hourlyValues = self.**getConstantEPScheduleValues**(schName)

**elif** scheduleType == "schedule:compact":

hourlyValues = self.**getCompactEPScheduleValues**(schName)

**else**:

**print** "Honeybee doesn't support " + scheduleType + " currently." + \

"Email us the type and we will try to add it to Honeybee."

hourlyValues = []

return hourlyValues

**def getHolidaySchedValues**(self, schName = **None**):

hourlyValues = []

**if** schName == **None**:

schName = self.schName

**if** self.hb\_EPObjectsAUX.**isSchedule**(schName):

values, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(schName.**upper**(), ghenv.Component)

scheduleType = values[0].**lower**()

**if** scheduleType == "schedule:year":

# generate weekly schedules

numOfWeeklySchedules = **int(**(**len**(values)-2)/5)

**for** i **in range**(numOfWeeklySchedules):

weekDayScheduleName = values[5 \* i + 2]

startDay = **int**(self.lb\_preparation.**getJD**(**int**(values[5 \* i + 3]), **int**(values[5 \* i + 4])))

endDay = **int**(self.lb\_preparation.**getJD**(**int**(values[5 \* i + 5]), **int**(values[5 \* i + 6])))

weekValues, comments = self.hb\_EPScheduleAUX.**getScheduleDataByName**(weekDayScheduleName.**upper**(), ghenv.Component)

holidaySchedule = self.**getScheduleValues**(weekValues[8])

hourlyValues.**append**([startDay,endDay,holidaySchedule])

return hourlyValues

**class EPTypes**(object):

**def \_\_init\_\_**(self):

self.srfType = {0:'WALL',

0.5: 'UndergroundWall',

1:'ROOF',

1.5: 'UndergroundCeiling',

2:'FLOOR',

2.25: 'UndergroundSlab',

2.5: 'SlabOnGrade',

2.75: 'ExposedFloor',

3:'CEILING',

4:'AIRWALL',

5:'WINDOW',

6:'SHADING',

'WALL': 'WALL',

'ROOF':'ROOF',

'FLOOR': 'FLOOR',

'CEILING': 'CEILING',

'WINDOW':'WINDOW',

'SHADING': 'SHADING'}

self.bldgTypes = {0:'OFFICE',

'OFFICE':'OFFC',

1:'RETAIL',

'RETAIL':'RETAIL',

2:'APT',

'MIDRISEAPARTMENT':'APT',

3:'PRIMSCH',

'PRIMARYSCHOOL':'PRIMSCH',

4:'SECSCH',

'SECONDARYSCHOOL':'SECSCH',

5:'SMLHOTL',

'SMALLHOTEL':'SMLHOTL',

6:'LRGHTL',

'LARGEHOTEL':'LRGHTL',

7:'HOSP',

'HOSPITAL':'HOSP',

8:'OUTPT',

'OUTPATIENT':'OUTPT',

9:'WARE',

'WAREHOUSE':'WARE',

10:'MARKET',

'SUPERMARKET':'MARKET',

11:'FULLREST',

'FULLSERVICERESTAURANT':'FULLREST',

12:'QUICKREST',

'QUICKSERVICERESTAURANT':'QUICKREST'

}

#Restaurant(Full Service) = "FullServiceRestaurant"

#Restaurant(Quick Service) = "QuickServiceRestaurant"

#Mid-rise Apartment = "Mid-riseApartment"

#Hospital = "Hospital"

#Small Office = "Small Office"

#Medium Office = "Medium Office"

#Large Office = "Large Office"

#Small Hotel = "SmallHotel"

#Large Hotel = "LargeHotel"

#Primary School = "PrimarySchool"

#Secondary School = "SecondarySchool"

#Strip Mall = "StripMall"

#Retail = "Retail"

#Warehouse = "Warehouse"

**class materialLibrary**(object):

**def \_\_init\_\_**(self):

self.zoneProgram = {0: 'RETAIL',

1: 'OFFICE',

2: 'RESIDENTIAL',

3: 'HOTEL'}

self.zoneConstructionSet = {0: 'RETAIL\_CON',

1: 'OFFICE\_CON',

2: 'RESIDENTIAL\_CON',

3: 'HOTEL\_CON'}

self.zoneInternalLoad = {0: 'RETAIL\_INT\_LOAD',

1: 'OFFICE\_INT\_LOAD',

2: 'RESIDENTIAL\_INT\_LOAD',

3: 'HOTEL\_INT\_LOAD'}

self.zoneSchedule = {0: 'RETAIL\_SCH',

1: 'OFFICE\_SCH',

2: 'RESIDENTIAL\_SCH',

3: 'HOTEL\_SCH'}

self.zoneThermostat = {0: 'RETAIL\_SCH',

1: 'OFFICE\_SCH',

2: 'RESIDENTIAL\_SCH',

3: 'HOTEL\_SCH'}

**class BuildingProgramsLib**(object):

**def \_\_init\_\_**(self):

self.bldgPrograms = {

0 : 'Office',

1 : 'Retail',

2 : 'MidriseApartment',

3 : 'PrimarySchool',

4 : 'SecondarySchool',

5 : 'SmallHotel',

6 : 'LargeHotel',

7 : 'Hospital',

8 : 'Outpatient',

9 : 'Warehouse',

10 : 'SuperMarket',

11 : 'FullServiceRestaurant',

12 : 'QuickServiceRestaurant',

'Office' : 'Office',

'Retail' : 'Retail',

'MidriseApartment' : 'MidriseApartment',

'PrimarySchool' : 'PrimarySchool',

'SecondarySchool' : 'SecondarySchool',

'SmallHotel' : 'SmallHotel',

'LargeHotel' : 'LargeHotel',

'Hospital' : 'Hospital',

'Outpatient' : 'Outpatient',

'Warehouse' : 'Warehouse',

'SuperMarket' : 'SuperMarket',

'FullServiceRestaurant' : 'FullServiceRestaurant',

'QuickServiceRestaurant' : 'QuickServiceRestaurant'}

self.zonePrograms = { "MidriseApartment" : {

0: "Apartment",

1: "Office",

2: "Corridor",

},

'Outpatient' : {

0: "IT\_Room",

1: "ProcedureRoom",

2: "Conference",

3: "MedGas",

4: "Janitor",

5: "Cafe",

6: "OR",

7: "PhysicalTherapy",

8: "Lobby",

9: "Xray",

10: "MRI\_Control",

11: "Toilet",

12: "Elec/MechRoom",

13: "Stair",

14: "PACU",

15: "Anesthesia",

16: "MRI",

17: "CleanWork",

18: "NurseStation",

19: "PreOp",

20: "Lounge",

21: "BioHazard",

22: "Office",

23: "Hall",

24: "Soil Work",

25: "DressingRoom",

26: "Exam",

27: "LockerRoom",

},

'LargeHotel' : {

0: "Storage",

1: "Mechanical",

2: "Banquet",

3: "GuestRoom",

4: "Laundry",

5: "Retail",

6: "Kitchen",

7: "Cafe",

8: "Corridor",

9: "Lobby"

},

'FullServiceRestaurant' : {

0: "Kitchen",

1: "Dining"

},

'PrimarySchool' : {

0: "Mechanical",

1: "Library",

2: "Cafeteria",

3: "Gym",

4: "Restroom",

5: "Office",

6: "Classroom",

7: "Kitchen",

8: "Corridor",

9: "Lobby"

},

'SmallHotel' : {

0: "Storage",

1: "GuestLounge",

2: "Mechanical",

3: "StaffLounge",

4: "PublicRestroom",

5: "GuestRoom",

6: "Exercise",

7: "Laundry",

8: "Meeting",

9: "Office",

10: "Stair",

11: "Corridor"

},

'SuperMarket' : {

0: "Sales/Produce",

1: "DryStorage",

2: "Office",

3: "Deli/Bakery"

},

'SecondarySchool' : {

0: "Mechanical",

1: "Library",

2: "Auditorium",

3: "Cafeteria",

4: "Gym",

5: "Restroom",

6: "Office",

7: "Classroom",

8: "Kitchen",

9: "Corridor",

10: "Lobby"

},

'Retail' : {

0: "Back\_Space",

1: "Point\_of\_Sale",

2: "Entry",

3: "Retail"

},

'Hospital' : {

0: "ER\_Trauma",

1: "PatCorridor",

2: "ICU\_PatRm",

3: "ER\_NurseStn",

4: "ICU\_Open",

5: "NurseStn",

6: "PhysTherapy",

7: "ICU\_NurseStn",

8: "Radiology",

9: "Dining",

10: "PatRoom",

11: "OR",

12: "Office",

13: "Kitchen",

14: "Lab",

15: "ER\_Exam",

16: "ER\_Triage",

17: "Corridor",

18: "Lobby"

},

'Office' : {

0: "BreakRoom",

1: "Storage",

2: "Vending",

3: "OpenOffice",

4: "ClosedOffice",

5: "Conference",

6: "PrintRoom",

7: "Restroom",

8: "Elec/MechRoom",

9: "IT\_Room",

10: "Stair",

11: "Corridor",

12: "Lobby"

},

'Warehouse' : {

0: "Office",

1: "Fine",

2: "Bulk"

},

'QuickServiceRestaurant' : {

0: "Kitchen",

1: "Dining"

}

}

**class EPSurfaceLib**(object):

# I think I can remove this now

**def \_\_init\_\_**(self):

# 4 represents an Air Wall

self.srfType = {0:'WALL',

1:'ROOF',

2:'FLOOR',

3:'CEILING',

4:'AIRWALL',

5:'WINDOW'}

# surface construction should change later

# to be based on the zone program

self.srfCnstr = {0:'Exterior\_Wall',

1:'Exterior\_Roof',

2:'Exterior\_Floor',

3:'Interior\_Floor',

4:'Air\_Wall',

5:'Exterior\_Window'}

self.srfBC = {0:'Outdoors',

1:'Outdoors',

2: 'Outdoors',

3: 'Adiabatic',

4: 'surface',

5: 'Outdoors'}

self.srfSunExposure = {0:'SunExposed',

1:'SunExposed',

2:'SunExposed',

3:'NoSun',

4:'NoSun',

5:'SunExposed',}

self.srfWindExposure = {0:'WindExposed',

1:'WindExposed',

2:'WindExposed',

3:'NoWind',

4:'NoWind',

5:'WindExposed'}

**class EPHvac**(object):

**def \_\_init\_\_**(self, GroupID="GroupI", Index=0, airDetails=**None**, heatingDetails=**None**, coolingDetails=**None**):

self.objectType = "HBHvac"

self.geometry = **None**

self.ID = **str**(uuid.**uuid4**())

self.GroupID = GroupID

self.Index = Index

self.airDetails = airDetails

self.heatingDetails = heatingDetails

self.coolingDetails = coolingDetails

**class EPZone**(object):

"""This calss represents a honeybee zone that will be used for energy and daylighting

simulatios"""

**def \_\_init\_\_**(self, zoneBrep, zoneID, zoneName, program = [**None**, **None**], isConditioned = **True**):

self.north = 0

self.objectType = "HBZone"

self.origin = rc.Geometry.Point3d.Origin

self.geometry = zoneBrep

self.zoneType = 1

self.multiplier = 1

self.ceilingHeight = ''

self.volume = ''

self.floorArea = ''

self.insideConvectionAlgorithm = ''

self.outsideConvectionAlgorithm = ''

self.partOfArea = **True**

self.isPlenum = **False**

self.num = zoneID

self.ID = **str**(uuid.**uuid4**())

self.name = self.**cleanName**(zoneName)

self.hasNonPlanarSrf = **False**

self.hasInternalEdge = **False**

# Air Mixing with Adjacent Zones

self.mixAir = **False**

self.mixAirZoneList = []

self.mixAirFlowList = []

self.mixAirFlowRate = 0.0963

self.mixAirFlowSched = []

# Natural Ventilation Properties

self.natVent = **False**

self.natVentType = []

self.natVentMinIndoorTemp = []

self.natVentMaxIndoorTemp = []

self.natVentMinOutdoorTemp = []

self.natVentMaxOutdoorTemp = []

self.natVentDeltaTemp = []

self.windowOpeningArea = []

self.windowHeightDiff = []

self.natVentSchedule = []

self.natVentWindDischarge = []

self.natVentStackDischarge = []

self.windowAngle = []

self.fanFlow = []

self.FanEfficiency = []

self.FanPressure = []

# Zone Internal Masses (or Furniture)

self.internalMassNames = []

self.internalMassSrfAreas = []

self.internalMassConstructions = []

# Zone Surfaces

self.surfaces = []

# Zone Thresholds

self.coolingSetPt= ""

self.heatingSetPt= ""

self.coolingSetback= ""

self.heatingSetback= ""

self.humidityMax= ""

self.humidityMin= ""

self.outdoorAirReq = "Sum"

# Daylight Thresholds

self.daylightCntrlFract = 0

self.illumSetPt = 100000

self.illumCntrlSensorPt = **None**

self.glareView = 0

self.GlareDiscomIndex = 22

# Air System Properties.

self.recirculatedAirPerArea = 0

self.ventilationSched = ""

# Geomtry Properties.

**if** zoneBrep != **None**:

self.isClosed = self.geometry.IsSolid

**else**:

self.isClosed = **False**

**if** self.isClosed:

**try**:

planarTrigger = self.**checkZoneNormalsDir**()

**except** Exception, e:

**print** 'Checking normal directions failed:\n' + `e`

# Zone Program

self.bldgProgram = program[0]

self.zoneProgram = program[1]

# assign schedules

self.**assignScheduleBasedOnProgram**()

# assign loads

self.**assignLoadsBasedOnProgram**()

# Assign a default HVAC System.

**if** isConditioned: self.HVACSystem = **EPHvac**("GroupI", 0, **None**, **None**, **None**) # assign ideal loads as default

**else**: self.HVACSystem = **EPHvac**("NoHVAC", -1, **None**, **None**, **None**)# no system

self.isConditioned = isConditioned

self.isThisTheTopZone = **False**

self.isThisTheFirstZone = **False**

# Earthtube

self.earthtube = **False**

**def cleanName**(self, zname):

#illegal characters include : , ! ; ( ) { } [ ] .

return zname.**strip**().**replace**(" ","\_").**replace**(":","-").**replace**(",","-").**replace**("!","-").**replace**(";","-")\

.**replace**("(","|").**replace**(")","|").**replace**("{","|").**replace**("}","|").**replace**("[","|").**replace**("]","|").**replace**(".","-")

**def resetID**(self):

self.ID = **str**(uuid.**uuid4**())

**def atuoPositionDaylightSensor**(self):

zoneCentPt = rc.Geometry.VolumeMassProperties.**Compute**(self.geometry).Centroid

zoneBB = rc.Geometry.Brep.**GetBoundingBox**(self.geometry, rc.Geometry.Plane.WorldXY)

zOfPt = zoneBB.Min.Z + 0.8

self.illumCntrlSensorPt = rc.Geometry.**Point3d**(zoneCentPt.X, zoneCentPt.Y, zOfPt)

**def transform**(self, transform, newKey=**None**, clearSurfacesBC = **True**, flip = **False**):

# Gnerate a new name if none is provided.

**if** newKey == **None**:

self.name += **str**(uuid.**uuid4**())[:8]

**else**:

self.name += newKey

# Update air mixing accross air walls to refernce new zones

**if** clearSurfacesBC == **True**:

self.mixAir = **False**

self.mixAirZoneList = []

self.mixAirFlowList = []

self.mixAirFlowSched = []

**else**:

**for** count, mixZ **in enumerate**(self.mixAirZoneList):

self.mixAirZoneList[count] = mixZ + newKey

# Transform any daylight control sensor points.

**if** self.illumCntrlSensorPt != **None**:

self.illumCntrlSensorPt.**Transform**(transform)

#Transform the geometry.

self.geometry.**Transform**(transform)

self.cenPt.**Transform**(transform)

**if** flip == **True**:

self.geometry.**Flip**()

**for** surface **in** self.surfaces:

surface.**transform**(transform, newKey, clearSurfacesBC, flip)

**def assignScheduleBasedOnProgram**(self, component = **None**):

# create an open office is the program is not assigned

**if** self.bldgProgram == **None**: self.bldgProgram = "Office"

**if** self.zoneProgram == **None**: self.zoneProgram = "OpenOffice"

openStudioStandardLib = sc.sticky ["honeybee\_OpenStudioStandardsFile"]

**try**:

schedulesAndLoads = openStudioStandardLib['space\_types']['90.1-2007']['ClimateZone 1-8'][self.bldgProgram][self.zoneProgram]

**except**:

msg = "Either your input for bldgProgram > [" + self.bldgProgram + "] or " + \

"the input for zoneProgram > [" + self.zoneProgram + "] is not valid.\n" + \

"Use ListSpacePrograms component to find the available programs."

**print** msg

**if** component != **None**:

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return

self.occupancySchedule = schedulesAndLoads['occupancy\_sch']

self.occupancyActivitySch = schedulesAndLoads['occupancy\_activity\_sch']

self.heatingSetPtSchedule = schedulesAndLoads['heating\_setpoint\_sch']

self.coolingSetPtSchedule = schedulesAndLoads['cooling\_setpoint\_sch']

self.lightingSchedule = schedulesAndLoads['lighting\_sch']

self.equipmentSchedule = schedulesAndLoads['elec\_equip\_sch']

self.infiltrationSchedule = schedulesAndLoads['infiltration\_sch']

# find all the patameters and assign them to

self.isSchedulesAssigned = **True**

**def assignLoadsBasedOnProgram**(self, component=**None**):

# create an open office is the program is not assigned

**if** self.bldgProgram == **None**: self.bldgProgram = "Office"

**if** self.zoneProgram == **None**: self.zoneProgram = "OpenOffice"

openStudioStandardLib = sc.sticky ["honeybee\_OpenStudioStandardsFile"]

**try**:

schedulesAndLoads = openStudioStandardLib['space\_types']['90.1-2007']['ClimateZone 1-8'][self.bldgProgram][self.zoneProgram]

**except**:

msg = "Either your input for bldgProgram > [" + self.bldgProgram + "] or " + \

"the input for zoneProgram > [" + self.zoneProgram + "] is not valid.\n" + \

"Use ListSpacePrograms component to find the available programs."

**print** msg

**if** component != **None**:

component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

return

# numbers in OpenStudio standard library are in IP and I have to convert them to SI!

self.equipmentLoadPerArea = schedulesAndLoads['elec\_equip\_per\_area'] \* 10.763961 #Per ft^2 to Per m^2

self.infiltrationRatePerArea = schedulesAndLoads['infiltration\_per\_area\_ext'] \* 0.00508001 #1 ft3/min.m2 = 5.08001016E-03 m3/s.m2

self.lightingDensityPerArea = schedulesAndLoads['lighting\_w\_per\_area'] \* 10.763961 #Per ft^2 to Per m^2

self.numOfPeoplePerArea = schedulesAndLoads[ 'occupancy\_per\_area'] \* 10.763961 /1000 #Per 1000 ft^2 to Per m^2

self.ventilationPerArea = schedulesAndLoads['ventilation\_per\_area'] \* 0.00508001 #1 ft3/min.m2 = 5.08001016E-03 m3/s.m2

self.ventilationPerPerson = schedulesAndLoads['ventilation\_per\_person'] \* 0.0004719 #1 ft3/min.perosn = 4.71944743E-04 m3/s.person

self.isLoadsAssigned = **True**

**def getCurrentSchedules**(self, returnDictionary = **False**, component = **None**):

# assign the default is there is no schedule assigned

**if not** self.isSchedulesAssigned:

self.**assignScheduleBasedOnProgram**(component)

**if not** returnDictionary:

report = " Schedule list:\n" + \

"occupancySchedule: " + **str**(self.occupancySchedule) + "\n" + \

"occupancyActivitySch: " + **str**(self.occupancyActivitySch) + "\n" + \

"heatingSetPtSchedule: " + **str**(self.heatingSetPtSchedule) + "\n" + \

"coolingSetPtSchedule: " + **str**(self.coolingSetPtSchedule) + "\n" + \

"lightingSchedule: " + **str**(self.lightingSchedule) + "\n" + \

"equipmentSchedule: " + **str**(self.equipmentSchedule) + "\n" + \

"infiltrationSchedule: " + **str**(self.infiltrationSchedule)+ "\n" + \

"ventilationSchedule: " + **str**(self.ventilationSched)+ "."

return report

**else**:

scheduleDict = {"occupancySchedule" : **str**(self.occupancySchedule),

"occupancyActivitySch" : **str**(self.occupancyActivitySch),

"heatingSetPtSchedule" :**str**(self.heatingSetPtSchedule),

"coolingSetPtSchedule" : **str**(self.coolingSetPtSchedule),

"lightingSchedule" : **str**(self.lightingSchedule),

"equipmentSchedule" : **str**(self.equipmentSchedule),

"infiltrationSchedule" : **str**(self.infiltrationSchedule),

"ventilationSched: " : **str**(self.ventilationSched)}

return scheduleDict

**def getCurrentLoads**(self, returnDictionary = **False**, component = **None**):

# assign the default is there is no schedule assigned

**if not** self.isLoadsAssigned:

self.**assignLoadsBasedOnProgram**(component)

**if not** returnDictionary:

report = " Internal Loads [SI]:\n" + \

"EquipmentsLoadPerArea: " + "%.6f"%self.equipmentLoadPerArea + "\n" + \

"infiltrationRatePerArea: " + "%.6f"%self.infiltrationRatePerArea + "\n" + \

"lightingDensityPerArea: " + "%.6f"%self.lightingDensityPerArea + "\n" + \

"numOfPeoplePerArea: " + "%.6f"%self.numOfPeoplePerArea + "\n" + \

"ventilationPerPerson: " + "%.6f"%self.ventilationPerPerson + "\n" + \

"ventilationPerArea: " + "%.6f"%self.ventilationPerArea + "\n" + \

"recircAirPerArea: " + "%.6f"%self.recirculatedAirPerArea + "."

return report

**else**:

loadsDict = {"EquipmentsLoadPerArea" : "%.4f"%self.equipmentLoadPerArea,

"infiltrationRatePerArea" : "%.4f"%self.infiltrationRatePerArea,

"lightingDensityPerArea" : "%.4f"%self.lightingDensityPerArea,

"numOfPeoplePerArea" : "%.4f"%self.numOfPeoplePerArea,

"ventilationPerArea" : "%.4f"%self.ventilationPerArea,

"ventilationPerPerson" : "%.4f"%self.ventilationPerPerson}

return loadsDict

**def joinMesh**(self, meshList):

joinedMesh = rc.Geometry.**Mesh**()

**for** m **in** meshList: joinedMesh.**Append**(m)

return joinedMesh

**def checkZoneNormalsDir**(self):

**def checkSrfNormal**(HBSrf, anchorPts, nVecs, planarTrigger):

#Find the corresponding surface in the closed zone geometry.

tol = sc.doc.ModelAbsoluteTolerance

**for** count, cenpt **in enumerate**(anchorPts):

#If the center points are the same, then these two represent the same surface.

**if** cenpt.X <= HBSrf.cenPt.X +tol **and** cenpt.X >= HBSrf.cenPt.X - tol **and** cenpt.Y <= HBSrf.cenPt.Y +tol **and** cenpt.Y >= HBSrf.cenPt.Y - tol **and** cenpt.Z <= HBSrf.cenPt.Z +tol **and** cenpt.Z >= HBSrf.cenPt.Z - tol:

**if** nVecs[count] != HBSrf.normalVector:

**print** "Normal direction for " + HBSrf.name + " is fixed by Honeybee!"

HBSrf.geometry.**Flip**()

HBSrf.normalVector.**Reverse**()

HBSrf.basePlane.**Flip**()

# change the surface type if need be.

**if** HBSrf.srfTypeByUser == **False**:

**if int**(HBSrf.type) == 2:

HBSrf.**setType**(1)

**elif int**(HBSrf.type) == 1 **or int**(HBSrf.type) == 3:

HBSrf.**setType**(2)

**try**: HBSrf.punchedGeometry.**Flip**()

**except**: pass

**if** HBSrf.hasChild **and** HBSrf.isPlanar:

**for** childSrf **in** HBSrf.childSrfs:

**if** childSrf.normalVector != nVecs[count]:

**print** "Normal direction for " + childSrf.name + " is fixed by Honeybee!"

childSrf.geometry.**Flip**()

childSrf.normalVector.**Reverse**()

childSrf.basePlane.**Flip**()

**elif** HBSrf.hasChild:

**for** childSrf **in** HBSrf.childSrfs:

# print childSrf.normalVector

childSrf.cenPt = rc.Geometry.AreaMassProperties.**Compute**(childSrf.geometry).Centroid

uv = childSrf.geometry.Faces[0].**ClosestPoint**(childSrf.cenPt)

childSrf.normalVector = childSrf.geometry.Faces[0].**NormalAt**(uv[1], uv[2])

#If the childSrfs are differing by more than 45 degrees, there's something wrong and we should flip them.

vecAngleDiff = math.**degrees**(rc.Geometry.Vector3d.**VectorAngle**(nVecs[count], childSrf.normalVector))

**if** vecAngleDiff > 45:

**print** "Normal direction for " + childSrf.name + " is fixed by Honeybee!"

childSrf.geometry.**Flip**()

childSrf.normalVector.**Reverse**()

# find center point, it won't be used in this function!

MP3D = rc.Geometry.AreaMassProperties.**Compute**(self.geometry)

self.cenPt = MP3D.Centroid

MP3D.**Dispose**()

#Extract the center points and normal vectors from the closed brep geometry.

planarTrigger = **False**

anchorPts = []

nVecs = []

closedBrepGeo = self.geometry

**for** surface **in** closedBrepGeo.Faces:

**if** surface.IsPlanar **and** surface.IsSurface:

u\_domain = surface.**Domain**(0)

v\_domain = surface.**Domain**(1)

centerU **=** (u\_domain.Min + u\_domain.Max)/2

centerV **=** (v\_domain.Min + v\_domain.Max)/2

anchorPts.**append**(surface.**PointAt**(centerU, centerV))

nVecs.**append**(surface.**NormalAt**(centerU, centerV))

**else**:

planarTrigger = **True**

centroid = rc.Geometry.AreaMassProperties.**Compute**(surface).Centroid

uv = surface.**ClosestPoint**(centroid)

anchorPts.**append**(surface.**PointAt**(uv[1], uv[2]))

nVecs.**append**(surface.**NormalAt**(uv[1], uv[2]))

**for** HBSrf **in** self.surfaces:

**checkSrfNormal**(HBSrf, anchorPts, nVecs, planarTrigger)

return planarTrigger

**def decomposeZone**(self, maximumRoofAngle = 30):

# this method is useufl when the zone is going to be constructed from a closed brep

# materials will be applied based on the zones construction set

#This check fails for any L-shaped zone so it has been disabled. We check the normals well elsewhere.

**def getGHSrfNormal**(GHSrf):

cenPt, normalVector = self.**getSrfCenPtandNormal**(surface)

return normalVector, GHSrf

# explode zone

**for** i **in range**(self.geometry.Faces.Count):

surface = self.geometry.Faces[i].**DuplicateFace**(**False**)

# check surface Normal

normal, surface = **getGHSrfNormal**(surface)

angle2Z = math.**degrees**(rc.Geometry.Vector3d.**VectorAngle**(normal, rc.Geometry.Vector3d.ZAxis))

**if** angle2Z < maximumRoofAngle **or** angle2Z > 360- maximumRoofAngle:

# roof is the right assumption

# it will change to ceiling after solveAdj if it is a ceiling

surafceType = 1 #roof

#if self.isThisTheTopZone: surafceType = 1 #roof

#else: surafceType = 3 # ceiling

**elif** 160 < angle2Z <200:

surafceType = 2 # floor

**else**: surafceType = 0 #wall

HBSurface = **hb\_EPZoneSurface**(surface, i, self.name + '\_Srf\_' + `i`, self, surafceType)

self.**addSrf**(HBSurface)

**def createZoneFromSurfaces**(self, maximumRoofAngle = 30):

# this method recreate the geometry from the surfaces

srfs = []

# check if surface has a type

**for** srf **in** self.surfaces:

srf.parent = self

# check planarity and set it for parent zone

**if not** srf.isPlanar:

self.hasNonPlanarSrf = **True**

**if** srf.hasInternalEdge:

self.hasInternalEdge = **True**

# also chek for interal Edges

surface = srf.geometry.Faces[0].**DuplicateFace**(**False**)

#print surface

srfs.**append**(surface)

**try**:

surfaceType = srf.type

**except**:

srf.type = srf.**getTypeByNormalAngle**()

srf.**reEvaluateType**(**True**)

# check for child surfaces

**if** srf.hasChild: srf.**calculatePunchedSurface**()

# assign construction

srf.construction = srf.cnstrSet[srf.type]

**if** srf.EPConstruction == "":

# if it is not already assigned by user then use default based on type

srf.EPConstruction = srf.construction

**try**:

self.geometry = rc.Geometry.Brep.**JoinBreps**(srfs, sc.doc.ModelAbsoluteTolerance)[0]

self.isClosed = self.geometry.IsSolid

**if** self.isClosed:

planarTrigger = **False**

**try**:

planarTrigger = self.**checkZoneNormalsDir**()

**except** Exception, e:

**print** '0\_Check Zone Normals Direction Failed:\n' + `e`

**if** planarTrigger == **True**:

MP3D = rc.Geometry.AreaMassProperties.**Compute**(self.geometry)

self.cenPt = MP3D.Centroid

MP3D.**Dispose**()

**else**:

MP3D = rc.Geometry.AreaMassProperties.**Compute**(self.geometry)

self.cenPt = MP3D.Centroid

MP3D.**Dispose**()

**except** Exception, e:

**print** " Failed to create the geometry from the surface:\n" + `e`

**def getSrfCenPtandNormal**(self, surface):

brepFace = surface.Faces[0]

**if** brepFace.IsPlanar **and** brepFace.IsSurface:

u\_domain = brepFace.**Domain**(0)

v\_domain = brepFace.**Domain**(1)

centerU **=** (u\_domain.Min + u\_domain.Max)/2

centerV **=** (v\_domain.Min + v\_domain.Max)/2

centerPt = brepFace.**PointAt**(centerU, centerV)

normalVector = brepFace.**NormalAt**(centerU, centerV)

**else**:

centroid = rc.Geometry.AreaMassProperties.**Compute**(brepFace).Centroid

uv = brepFace.**ClosestPoint**(centroid)

centerPt = brepFace.**PointAt**(uv[1], uv[2])

normalVector = brepFace.**NormalAt**(uv[1], uv[2])

return centerPt, normalVector

**def addSrf**(self, srf):

self.surfaces.**append**(srf)

**def updateConstructionSet**(newProgramCode, level = 1):

"""level defines the level of the construction set

0: low performance; 1: normal; 2: high performance"""

self.constructionSet = constructionSet[newProgramCode]

**def cleanMeshedFaces**(self):

**for** srf **in** self.surfaces: srf.**disposeCurrentMeshes**()

**def prepareNonPlanarZone**(self, meshingParameters = **None**, isEnergyPlus = **False**):

# clean current meshedFaces

self.**cleanMeshedFaces**()

# collect walls and windows, and roofs

srfsToBeMeshed = []

**for** srf **in** self.surfaces:

#clean the meshedFaces if any

# if surface is planar just collect the surface

**if** srf.isPlanar **or not** srf.hasChild: srfsToBeMeshed.**append**(srf.geometry)

# else collect the punched wall and child surfaces

**else**:

**for** fenSrf **in** srf.childSrfs:

srfsToBeMeshed.**append**(fenSrf.geometry)

srfsToBeMeshed.**append**(fenSrf.parent.punchedGeometry)

# join surfaces

joinedBrep = rc.Geometry.Brep.**JoinBreps**(srfsToBeMeshed, sc.doc.ModelAbsoluteTolerance)[0]

# mesh the geometry

**if** meshingParameters == **None or type**(meshingParameters)!= rc.Geometry.MeshingParameters:

mp = rc.Geometry.MeshingParameters.Default; disFactor = 3

**else**:

disFactor = 1

mp = meshingParameters

meshedGeo = rc.Geometry.Mesh.**CreateFromBrep**(joinedBrep, mp)

**for** mesh **in** meshedGeo:

# generate quad surfaces for EnergyPlus model

# if isEnergyPlus:

# angleTol = sc.doc.ModelAngleToleranceRadians

# minDiagonalRatio = .875

# #print mesh.Faces.ConvertTrianglesToQuads(angleTol, minDiagonalRatio)

# mesh.Faces.ConvertTrianglesToQuads(angleTol, minDiagonalRatio)

mesh.FaceNormals.**ComputeFaceNormals**()

#mesh.FaceNormals.UnitizeFaceNormals()

**for** faceIndex **in range**(mesh.Faces.Count):

normal = mesh.FaceNormals[faceIndex]

cenPt = mesh.Faces.**GetFaceCenter**(faceIndex)

##check mesh normal direction

reverseList = **False**

## make a vector from the center point of the zone to center point of the surface

**try**:

testVector = rc.Geometry.**Vector3d**(cenPt - self.cenPt)

**except**:

MP3D = rc.Geometry.AreaMassProperties.**Compute**(self.geometry)

self.cenPt = MP3D.Centroid

testVector = rc.Geometry.**Vector3d**(cenPt - self.cenPt)

## check the direction of the vectors and flip zone surfaces if needed

**if** rc.Geometry.Vector3d.**VectorAngle**(testVector, normal)> 1:

normal.**Reverse**()

reverseList = **True**

## create a ray

#ray = rc.Geometry.Ray3d(cenPt, normal)

**for** srf **in** self.surfaces:

**if** srf.isPlanar **or not** srf.hasChild:

## shoot a ray from the center of the mesh to each surface

#intPt = rc.Geometry.Intersect.Intersection.RayShoot(ray, [srf.geometry], 1)

#if intPt:

**if** cenPt.**DistanceTo**(srf.geometry.**ClosestPoint**(cenPt))<0.05 \* disFactor:

srf.**collectMeshFaces**(mesh.Faces.**GetFaceVertices**(faceIndex), reverseList) ## if hit then add this face to that surface

break

**else**:

**for** fenSrf **in** srf.childSrfs:

#intPt = rc.Geometry.Intersect.Intersection.RayShoot(ray, [fenSrf.geometry], 1)

#if intPt:

**if** cenPt.**DistanceTo**(fenSrf.geometry.**ClosestPoint**(cenPt))<0.05 \* disFactor:

fenSrf.**collectMeshFaces**(mesh.Faces.**GetFaceVertices**(faceIndex), reverseList); break

#intPt = rc.Geometry.Intersect.Intersection.RayShoot(ray, [fenSrf.parent.punchedGeometry], 1)

#if intPt:

**if** cenPt.**DistanceTo**(fenSrf.parent.punchedGeometry.**ClosestPoint**(cenPt))<0.05 \* disFactor:

srf.**collectMeshFaces**(mesh.Faces.**GetFaceVertices**(faceIndex), reverseList); break

**def getFloorArea**(self, meterOverride=**False**):

totalFloorArea = 0

**for** HBSrf **in** self.surfaces:

**if int**(HBSrf.type) == 2:

totalFloorArea += HBSrf.**getTotalArea**(meterOverride)

return totalFloorArea

**def getZoneVolume**(self):

return self.geometry.**GetVolume**()\*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**def getExposedArea**(self):

totalExpArea = 0

**for** HBSrf **in** self.surfaces:

**if** HBSrf.BC.**lower**() == "outdoors":

totalExpArea += HBSrf.**getTotalArea**()

return totalExpArea

**def getFloorZLevel**(self):

# useful for gbXML export

minZ = **float**("inf")

**for** HBSrf **in** self.surfaces:

**if int**(HBSrf.type) == 2:

#get the center point

centerPt, normalVector = HBSrf.**getSrfCenPtandNormalAlternate**()

**if** centerPt.Z < minZ: minZ = centerPt.Z

return minZ

**def setName**(self, newName):

self.name = newName

**def \_\_str\_\_**(self):

**try**:

return 'Zone name: ' + self.name + \

'\nZone program: ' + self.bldgProgram + "::" + self.zoneProgram + \

'\n# of surfaces: ' + `**len**(self.surfaces)` + \

'\n-------------------------------------'

**except**:

return 'Zone name: ' + self.name + \

'\nZone program: Unknown' + \

'\n# of surfaces: ' + `**len**(self.surfaces)` + \

'\n-----------------------------------'

**class hb\_reEvaluateHBZones**(object):

"""

This class check Honeybee zones once more and zones with nonplanar surfaces

or non-rectangualr glazings recreates the surfaces so the output zones will

be all zones with planar surfaces, and they can be exported with two functions

for planar EPSurfaces and planar fenestration.

It also assigns the right boundary condition object to each sub surface

and checks duplicate names for zones and surfaces and give a warning

to user to get them fixed.

"""

**def \_\_init\_\_**(self, inHBZones, meshingParameters, pointOrient = "LowerLeftCorner"):

# import the classes

self.hb\_EPZone = sc.sticky["honeybee\_EPZone"]

self.hb\_EPSrf = sc.sticky["honeybee\_EPSurface"]

self.hb\_EPZoneSurface = sc.sticky["honeybee\_EPZoneSurface"]

self.hb\_EPFenSurface = sc.sticky["honeybee\_EPFenSurface"]

self.fakeSurface = rc.Geometry.Brep.**CreateFromCornerPoints**(

rc.Geometry.**Point3d**(0,0.5,0),

rc.Geometry.**Point3d**(-0.5,-0.5,0),

rc.Geometry.**Point3d**(0.5,-0.5,0),

sc.doc.ModelAbsoluteTolerance)

self.originalHBZones = inHBZones

self.meshingParameters = meshingParameters

#self.triangulate = triangulate

self.zoneNames = []

self.srfNames = []

self.modifiedSrfsNames= []

self.modifiedGlzSrfsNames = []

self.adjcGlzSrfCollection = []

self.adjcSrfCollection = {} #collect adjacent surfaces for nonplanar surfaces

self.pointOrient = pointOrient

**def checkSrfNameDuplication**(self, surface):

**if** surface.name **in** self.srfNames:

warning = "Duplicate surface name!"

name = surface.name

**while** name **in** self.srfNames:

name += "\_Dup"

surface.name = name

**print** warning + " Name is changed to: " + surface.name

self.srfNames.**append**(surface.name)

**if not** surface.isChild **and** surface.hasChild:

**for** child **in** surface.childSrfs:

self.**checkSrfNameDuplication**(child)

**def checkNameDuplication**(self, HBZone):

**if** HBZone.name **in** self.zoneNames:

warning = "Duplicate zone name!"

name = HBZone.name

**while** name **in** self.zoneNames:

name += "\_Dup"

HBZone.name = name

**print** warning + " Name is changed to: " + HBZone.name

self.zoneNames.**append**(HBZone.name)

**for** surface **in** HBZone.surfaces:

self.**checkSrfNameDuplication**(surface)

**def prepareNonPlanarZones**(self, HBZone):

# prepare nonplanar zones

**if** HBZone.hasNonPlanarSrf **or** HBZone.hasInternalEdge:

HBZone.**prepareNonPlanarZone**(self.meshingParameters)

**def createSurface**(self, pts):

"""

# it takes so long if I generate the geometry

if len(pts) == 3:

return rc.Geometry.Brep.CreateFromCornerPoints(pts[0], pts[1], pts[2], sc.doc.ModelAbsoluteTolerance)

elif len(pts) == 4:

return rc.Geometry.Brep.CreateFromCornerPoints(pts[0], pts[1], pts[2], pts[3], sc.doc.ModelAbsoluteTolerance)

else:

# create a planar surface

pts.append(pts[0])

pl = rc.Geometry.Polyline(pts).ToNurbsCurve()

return rc.Geometry.Brep.CreatePlanarBreps([pl])[0]

"""

return self.fakeSurface

**def evaluateZones**(self):

**if** sc.sticky["honeybee\_ConversionFactor"] != 1:

NUscale = rc.Geometry.Transform.**Scale**(rc.Geometry.**Plane**(rc.Geometry.Plane.WorldXY),sc.sticky["honeybee\_ConversionFactor"],sc.sticky["honeybee\_ConversionFactor"],sc.sticky["honeybee\_ConversionFactor"])

**for** HBZone **in** self.originalHBZones:

**if** sc.sticky["honeybee\_ConversionFactor"] != 1:

HBZone.**transform**(NUscale, "", **False**)

self.**checkNameDuplication**(HBZone)

self.**prepareNonPlanarZones**(HBZone)

modifiedSurfaces = []

**for** surface **in** HBZone.surfaces:

srfs = self.**checkZoneSurface**(surface)

**try**: modifiedSurfaces.**extend**(srfs)

**except**: modifiedSurfaces.**append**(srfs)

# replace surfaces with new ones

HBZone.surfaces = []

**for** HBSrf **in** modifiedSurfaces:

HBZone.surfaces.**append**(HBSrf)

**def createSubSurfaceFromBaseSrf**(self, surface, newSurfaceName, count, coordinates, glazingBase = **False**, nameAddition = **None**):

# pass the wrong geometry for now. I assume creating planar surface from

# coordinates will be computationally heavy and at this point geometry doesn't

# matter, since I have the coordinates.

newSurface = self.**hb\_EPZoneSurface**(self.**createSurface**(coordinates),

count, newSurfaceName, surface.parent, surface.type)

newSurface.coordinates = coordinates

newSurface.type = surface.type # protect the surface from reEvaluate

newSurface.construction = surface.construction

newSurface.EPConstruction = surface.EPConstruction

newSurface.BC = surface.BC

newSurface.sunExposure = surface.sunExposure

newSurface.windExposure = surface.windExposure

newSurface.groundViewFactor = surface.groundViewFactor

**if** surface.BC.**upper**() == 'SURFACE':

adjcSurface = surface.BCObject

**if not** glazingBase:

newAdjcSurfaceName = adjcSurface.name + "\_srfP\_" + `count`

**else**:

**try**: newAdjcSurfaceName = adjcSurface.name + **str**(nameAddition)

**except**: newAdjcSurfaceName = adjcSurface.name + "\_"

newAdjcSurface = self.**hb\_EPZoneSurface**(self.**createSurface**(coordinates),

count, newAdjcSurfaceName, adjcSurface.parent, adjcSurface.type)

# reverse the order of points

restOfcoordinates = **list**(coordinates[1:])

restOfcoordinates.**reverse**()

newAdjcSurface.coordinates = [coordinates[0]] + restOfcoordinates

newAdjcSurface.type = adjcSurface.type

newAdjcSurface.construction = adjcSurface.construction

newAdjcSurface.EPConstruction = adjcSurface.EPConstruction

newAdjcSurface.BC = adjcSurface.BC

newAdjcSurface.sunExposure = adjcSurface.sunExposure

newAdjcSurface.windExposure = adjcSurface.windExposure

newAdjcSurface.groundViewFactor = adjcSurface.groundViewFactor

# assign boundary objects

newSurface.BCObject = newAdjcSurface

newAdjcSurface.BCObject = newSurface

self.adjcSrfCollection[adjcSurface.name].**append**(newAdjcSurface)

return newSurface

**def createSubGlzSurfaceFromBaseSrf**(self, baseChildSurface, parentSurface, glzSurfaceName, count, coordinates):

newFenSrf = self.**hb\_EPFenSurface**(self.**createSurface**(coordinates),

count, glzSurfaceName, parentSurface, 5, punchedWall = **None**)

newFenSrf.coordinates = coordinates

newFenSrf.type = baseChildSurface.type

newFenSrf.construction = baseChildSurface.construction

newFenSrf.EPConstruction = baseChildSurface.EPConstruction

newFenSrf.parent = parentSurface

newFenSrf.groundViewFactor = baseChildSurface.groundViewFactor

newFenSrf.shadingControlName = baseChildSurface.shadingControlName

newFenSrf.frameName = baseChildSurface.frameName

newFenSrf.Multiplier = baseChildSurface.Multiplier

newFenSrf.shadeMaterialName = baseChildSurface.shadeMaterialName

newFenSrf.shadingSchName = baseChildSurface.shadingSchName

# Will be overwritten later if needed

newFenSrf.BCObject = baseChildSurface.BCObject

newFenSrf.BCObject = baseChildSurface.BCObject

return newFenSrf

**def getInsetGlazingCoordinates**(self, glzCoordinates):

# find the coordinates

**def averagePts**(ptList):

pt = rc.Geometry.**Point3d**(0,0,0)

**for** p **in** ptList: pt = pt + p

return rc.Geometry.**Point3d**(pt.X/**len**(ptList), pt.Y/**len**(ptList), pt.Z/**len**(ptList))

distance = 2 \* sc.doc.ModelAbsoluteTolerance

# offset was so slow so I changed the method to this

pts = []

**for** pt **in** glzCoordinates:

pts.**append**(rc.Geometry.**Point3d**(pt.X, pt.Y, pt.Z))

cenPt = **averagePts**(pts)

insetPts = []

**for** pt **in** pts:

movingVector = rc.Geometry.**Vector3d**(cenPt-pt)

movingVector.**Unitize**()

newPt = rc.Geometry.Point3d.**Add**(pt, movingVector \* 2 \* sc.doc.ModelAbsoluteTolerance)

insetPts.**append**(newPt)

return insetPts

**def isAntiClockWise**(self, pts, faceNormal):

**def crossProduct**(vector1, vector2):

return vector1.X \* vector2.X + vector1.Y \* vector2.Y + vector1.Z \* vector2.Z

# check if the order if clock-wise

vector0 = rc.Geometry.**Vector3d**(pts[1]- pts[0])

vector1 = rc.Geometry.**Vector3d**(pts[-1]- pts[0])

ptsNormal = rc.Geometry.Vector3d.**CrossProduct**(vector0, vector1)

# in case points are anti-clockwise then normals should be parallel

**if crossProduct**(ptsNormal, faceNormal) > 0:

return **True**

return **False**

**def checkChildSurfaces**(self, surface, pointOrient = 'LowerLeftCorner'):

**def isRectangle**(ptList):

vector1 = rc.Geometry.**Vector3d**(ptList[0] - ptList[1])

vector2 = rc.Geometry.**Vector3d**(ptList[1] - ptList[2])

vector3 = rc.Geometry.**Vector3d**(ptList[2] - ptList[3])

vector4 = rc.Geometry.**Vector3d**(ptList[3] - ptList[0])

**if** ptList[0].**DistanceTo**(ptList[2]) != ptList[1].**DistanceTo**(ptList[3]) **or** \

math.**degrees**(rc.Geometry.Vector3d.**VectorAngle**(vector1, vector2))!= 90 **or** \

math.**degrees**(rc.Geometry.Vector3d.**VectorAngle**(vector3, vector4))!= 90:

return **False**

**else**:

return **True**

# get glaing coordinates- coordinates will be returned as lists of lists

glzCoordinates = surface.**extractGlzPoints**(**False**, 2, pointOrient)

# check that the coordinates are going anticlockwise.

**for** i, coorList **in enumerate**(glzCoordinates):

**if not** self.**isAntiClockWise**(coorList, surface.normalVector):

# reverse the list of coordinates

coorList.**reverse**()

# Shift the list by 1 to make sure that the starting point is still in the correct corner (ie. LowerLeft).

glzCoordinates[i] = coorList[-1:] + coorList[:-1]

glzSrfs = []

**if** surface.isPlanar:

**for** count, coordinates **in enumerate**(glzCoordinates):

**try**: child = surface.childSrfs[count]

**except**: child = surface.childSrfs[0]

# if len(surface.childSrfs) == len(glzCoordinates): #not hasattr(glzCoordinates, '\_\_iter\_\_'):

**if len**(glzCoordinates)== 1: #not hasattr(glzCoordinates, '\_\_iter\_\_'):

# single rectangular glazing - All should be fine

# also the adjacent surface will be fine by itself

child.coordinates = coordinates

self.modifiedGlzSrfsNames.**append**(child.name)

**else**:

# surface is planar but glazing is not rectangular

# and so it is meshed now and is multiple glazing

**if len**(surface.childSrfs) == **len**(glzCoordinates):

# multiple rectangle windows

glzSurfaceName = child.name

**else**:

# multiple non-rectangle rectangle window

# this naming should be fixed and be based on original surface

glzSurfaceName = child.name + "\_glzP\_" + `count`

# create glazing surface

HBGlzSrf = self.**createSubGlzSurfaceFromBaseSrf**(child, surface, glzSurfaceName, count, coordinates)

HBGlzSrf.normalVector = surface.normalVector

# create adjacent glazingin case needed

**if** surface.BC.**upper**() == 'SURFACE':

# add glazing to adjacent surface

adjcSrf = surface.BCObject

#This well-intentioned check was stopping good geomtry from being run through EnergyPlus. It has thus been disabled. - Chris Mackey

#assert len(surface.childSrfs) != len(adjcSrf.childSrfs), \

# "Adjacent surfaces %s and %s do not have the same number of galzings.\n"%(surface.name, adjcSrf.name) + \

# "Check your energy model and try again."

# add glazing to adjacent surface

**if** count == 0:

adjcSrf = surface.BCObject

childSrfsNames = []

**for** childSurface **in** adjcSrf.childSrfs: childSrfsNames.**append**(childSurface.name)

adjcSrf.childSrfs = []

**if len**(surface.childSrfs) == **len**(glzCoordinates):

glzAdjcSrfName = childSrfsNames[count]

**else**:

**try**:

glzAdjcSrfName = childSrfsNames[count] + "\_glzP\_" + `count`

**except**:

glzAdjcSrfName = childSrfsNames[0] + "\_glzP\_" + `count`

adjcGlzPt = glzCoordinates[1:]

adjcGlzPt.**reverse**()

adjcGlzPt = [glzCoordinates[0]] + adjcGlzPt

adjHBGlzSrf = self.**createSubGlzSurfaceFromBaseSrf**(child, adjcSrf, glzAdjcSrfName, count, adjcGlzPt)

# overwrite BC Object

adjHBGlzSrf.BCObject = HBGlzSrf

HBGlzSrf.BCObject = adjHBGlzSrf

adjcSrf.**addChildSrf**(adjHBGlzSrf)

# collect surfaces

glzSrfs.**append**(HBGlzSrf)

# add to parent surface

**if len**(glzCoordinates) != 1:

surface.**removeAllChildSrfs**()

surface.**addChildSrf**(glzSrfs)

**else**:

# convert nonplanar surface to planar wall surfaces with offset glazing

# and treat them similar to other surfaces except the fact that if it has

# another surface next to it the surface should be generated regardless of

# being single geometry or not

newSurfaces =[]

count = 0

baseChildSrf = surface.childSrfs[0]

**for** count, glzCoordinate **in enumerate**(glzCoordinates):

**try**:

baseGlazingName = surface.childSrfs[count].name

**except**:

baseGlazingName = surface.childSrfs[0].name

# check if the points are recetangle

**if len**(glzCoordinate) == 3 **or isRectangle**(glzCoordinate):

insetGlzCoordinates = [glzCoordinate]

**else**:

# triangulate

insetGlzCoordinates = [glzCoordinate[:3], [glzCoordinate[0],glzCoordinate[2],glzCoordinate[3]]]

**for** glzCount, insetGlzCoordinate **in enumerate**(insetGlzCoordinates):

# self.modifiedGlzSrfsNames.append(child.name)

# create new Honeybee surfaces as parent surface for glass face

**if len**(insetGlzCoordinates) == 1:

newSurfaceName = surface.name + '\_glzP\_' + `count`

**else**:

newSurfaceName = surface.name + '\_glzP\_' + `count` + '\_' + `glzCount`

newSurface = self.**createSubSurfaceFromBaseSrf**(surface, newSurfaceName, count, insetGlzCoordinate, glazingBase = **True**, nameAddition = '\_glzP\_' + `count` + '\_' + `glzCount`)

# collect them here so it will have potential new BC

newSurfaces.**append**(newSurface)

# create glazing coordinate and add it to the parent surface

insetPts = self.**getInsetGlazingCoordinates**(insetGlzCoordinate)

# create new window and go for it

glzSurfaceName = baseGlazingName + "\_glzP\_" + `count`

HBGlzSrf = self.**createSubGlzSurfaceFromBaseSrf**(baseChildSrf, newSurface, glzSurfaceName, count, insetPts)

**if** surface.BC.**upper**() == 'SURFACE':

# add glazing to adjacent surface

**if** count == 0:

adjcSrf = newSurface.BCObject

**try**:

adjBaseGlazingName = adjcSrf.childSrfs[count]

**except**:

adjBaseGlazingName = adjcSrf.childSrfs[0]

adjcSrf.childSrfs = []

# add glazing to adjacent surface

adjcSrf = newSurface.BCObject

glzAdjcSrfName = adjBaseGlazingName + "\_glzP\_" + `count`

adjcGlzPt = insetPts[1:]

adjcGlzPt.**reverse**()

adjcGlzPt = [insetPts[0]] + adjcGlzPt

adjHBGlzSrf = self.**createSubGlzSurfaceFromBaseSrf**(baseChildSrf, adjcSrf, glzAdjcSrfName, count, adjcGlzPt)

# overwrite BC Object

adjHBGlzSrf.BCObject = HBGlzSrf

HBGlzSrf.BCObject = adjHBGlzSrf

adjcSrf.**addChildSrf**(adjHBGlzSrf)

# add to parent surface

newSurface.**addChildSrf**(HBGlzSrf)

return newSurfaces

**def checkZoneSurface**(self, surface):

**if not hasattr**(surface, 'coordinates'):

**if not** surface.isPlanar:

**if hasattr**(surface, 'punchedGeometry'):

surface.geometry = surface.punchedGeometry

coordinatesL = surface.**extractPoints**(1, **False**, **None**, self.pointOrient)

**else**:

coordinatesL = surface.coordinates

# case 0 : it is a planar surface so it is all fine

**if not hasattr**(coordinatesL[0], '\_\_iter\_\_'):

**if not** self.**isAntiClockWise**(coordinatesL, surface.normalVector):

# reverse the list of coordinates

coordinatesL.**reverse**()

# Shift the list by 1 to make sure that the starting point is still in the correct corner (ie. LowerLeft).

coordinatesL = coordinatesL[-1:] + coordinatesL[:-1]

# it is a single surface so just let it go to the modified list

surface.coordinates = coordinatesL

self.modifiedSrfsNames.**append**(surface.name)

**if not** surface.isChild **and** surface.hasChild:

self.**checkChildSurfaces**(surface, self.pointOrient)

return surface

# case 1 : it is not planar

**else**:

# case 1-1 : surface is a nonplanar surface and adjacent to another surface

# sub surfaces has been already generated based on the adjacent surface

**if** surface.BC.**upper**() == 'SURFACE' **and** surface.name **in** self.adjcSrfCollection.**keys**():

# print "collecting sub surfaces for surface " + surface.name

# surface has been already generated by the other adjacent surface

self.modifiedSrfsNames.**append**(surface.name)

return self.adjcSrfCollection[surface.name]

# case 1-2 : surface is a nonplanar surface and adjacent to another surface

# and hasn't been generated so let's generate this surface and the adjacent one

**elif** surface.BC.**upper**() == 'SURFACE':

adjcSurface= surface.BCObject

# find adjacent zone and create the surfaces

# create a place holder for the surface

# the surfaces will be collected inside the function

self.adjcSrfCollection[adjcSurface.name] = []

self.modifiedSrfsNames.**append**(surface.name)

newSurfaces = []

**for** count, coordinates **in enumerate**(coordinatesL):

# create new Honeybee surfaces

# makes sense to copy the original surface here but since

# copy.deepcopy fails on a number of systems I just create

# a new surface and assign necessary data to write the surface

newSurfaceName = surface.name + "\_srfP\_" + `count`

newSurface = self.**createSubSurfaceFromBaseSrf**(surface, newSurfaceName, count, coordinates)

newSurfaces.**append**(newSurface)

# nonplanar surface

**if not** surface.isChild **and** surface.hasChild:

glzPSurfaces = self.**checkChildSurfaces**(surface, self.pointOrient)

**if** glzPSurfaces != **None**:

newSurfaces += glzPSurfaces

return newSurfaces

**class hb\_EPSurface**(object):

**def \_\_init\_\_**(self, surface, srfNumber, srfID, \*arg):

"""EP surface Class

surface: surface geometry as a Brep

srfNumber: a unique number that is only for this surface

srfID: the unique name for this surface

\*arg is parentZone for EPZoneClasses

\*arg is parentSurface for child surfaces"""

self.objectType = "HBSurface"

self.geometry = surface

self.num = srfNumber

self.name = self.**cleanName**(srfID)

self.ID = **str**(uuid.**uuid4**())

self.isPlanar = self.**checkPlanarity**()

self.hasInternalEdge = self.**checkForInternalEdge**()

self.meshedFace = rc.Geometry.**Mesh**()

self.RadMaterial = **None**

self.EPConstruction = **None** # this gets overwritten below

self.cenPt, self.normalVector = self.**getSrfCenPtandNormalAlternate**()

self.basePlane = rc.Geometry.**Plane**(self.cenPt, self.normalVector)

# define if type and BC is defined by user and should be kept

self.srfTypeByUser = **False**

self.srfBCByUser = **False**

self.BCObject = self.**outdoorBCObject**()

# Special attribute for shading control on inidivdual windows that influences the zone properties

self.shdCntrlZoneInstructs = []

# PV - A Honeybee surface can hold one PV generator

self.PVgenlist = []

# Does this Honeybee surface contain a PV generator?

self.containsPVgen = **False**

# 4 represents an Air Wall

self.srfType = {0:'WALL',

0.5: 'UndergroundWall',

1:'ROOF',

1.5: 'UndergroundCeiling',

2:'FLOOR',

2.25: 'UndergroundSlab',

2.5: 'SlabOnGrade',

2.75: 'ExposedFloor',

3:'CEILING',

4:'AIRWALL',

5:'WINDOW',

6:'SHADING',

'WALL': 'WALL',

'ROOF':'ROOF',

'FLOOR': 'FLOOR',

'CEILING': 'CEILING',

'WINDOW':'WINDOW',

'SHADING': 'SHADING'}

self.cnstrSet = {0:'Exterior Wall',

0.5: 'Exterior Wall',

1: 'Exterior Roof',

1.5: 'Exterior Roof',

2:'Interior Floor',

2.25: 'Exterior Floor',

2.5: 'Exterior Floor',

2.75: 'Exterior Floor',

3:'Interior Ceiling',

4:'Air Wall',

5:'Exterior Window',

6:'Interior Wall'}

self.intCnstrSet = {

0:'Interior Wall',

0.5: 'Exterior Wall',

1:'Exterior Roof',

1.5:'Exterior Roof',

2:'Interior Floor',

2.25: 'Exterior Floor',

2.5: 'Exterior Floor',

2.75: 'Exterior Floor',

3:'Interior Ceiling',

4:'Air Wall',

5:'Interior Window',

6:'Interior Wall'}

self.srfBC = {0:'Outdoors',

0.5: 'ground',

1:'Outdoors',

1.5: 'ground',

2: 'outdoors', # this will be changed to surface once solveAdjacency is used

2.25: 'ground',

2.5: 'ground',

2.75: 'outdoors',

3: 'outdoors', # this will be changed to surface once solveAdjacency is used

4: 'surface',

5: 'Outdoors',

6: 'surface'}

self.srfSunExposure = {0:'SunExposed',

0.5:'NoSun',

1:'SunExposed',

1.5:'NoSun',

2:'NoSun',

2.25: 'NoSun',

2.5: 'NoSun',

2.75: 'SunExposed',

3:'NoSun',

4:'NoSun',

6: 'NoSun'}

self.srfWindExposure = {0:'WindExposed',

0.5:'NoWind',

1:'WindExposed',

1.5:'NoWind',

2:'NoWind',

2.25:'NoWind',

2.5:'NoWind',

2.75:'WindExposed',

3:'NoWind',

4:'NoWind',

6:'NoWind'}

self.numOfVertices = 'autocalculate'

**if len**(arg) == 0:

# minimum surface

# A minimum surface is a surface that will be added to a zone later

# or is a surface that will only be used for daylighting simulation

# so the concept of parent zone/surface is irrelevant

self.parent = **None**

self.**reEvaluateType**(**True**)

**elif len**(arg) == 1:

# represents an opening. The parent is the parent surafce

# honeybee only supports windows (and not doors) at this point so

# the type is always the same (window)

self.parent = arg[0]

**elif len**(arg) == 2:

# represents a normal EP surface

# parent is a parent zone and the type differs case by case

self.parent = arg[0] # parent zone

self.type = arg[1] # surface type (e.g. wall, roof,...)

self.BC = self.srfBC[self.type] # initial BC based on type

# check for special conditions(eg. slab underground, slab on ground

self.**reEvaluateType**(**True**) # I should give this another thought

# this should be fixed to be based on zone type

# I can remove default constructions at some point

self.construction = self.cnstrSet[**int**(self.type)]

self.EPConstruction = self.construction

**def cleanName**(self, sname):

#illegal characters include : , ! ; ( ) { } [ ] .

return sname.**strip**().**replace**(" ","\_").**replace**(":","-").**replace**(",","-").**replace**("!","-").**replace**(";","-")\

.**replace**("(","|").**replace**(")","|").**replace**("{","|").**replace**("}","|").**replace**("[","|").**replace**("]","|").**replace**(".","-")

**def resetID**(self):

self.ID = **str**(uuid.**uuid4**())

**def checkPlanarity**(self):

# planarity tolerance should change for different

return self.geometry.Faces[0].**IsPlanar**(1e-3)

**def checkForInternalEdge**(self):

edges = self.geometry.**DuplicateEdgeCurves**(**True**)

edgesJoined = rc.Geometry.Curve.**JoinCurves**(edges)

**if len**(edgesJoined)>1:

return **True**

**else**:

return **False**

**class outdoorBCObject**(object):

"""

BCObject for surfaces with outdoor BC

"""

**def \_\_init\_\_**(self, name = ""):

self.name = name

**def getAngle2North**(self):

types = [0, 4, 5] # vertical surfaces

northVector = rc.Geometry.Vector3d.YAxis

# rotate north based on the zone north vector

**try**: northVector.**Rotate**(math.**radians**(self.parent.north), rc.Geometry.Vector3d.ZAxis)

**except**: pass

normalVector = self.**getSrfCenPtandNormalAlternate**()[1]

**if** self.type **in** types:

angle = rc.Geometry.Vector3d.**VectorAngle**(northVector, normalVector, rc.Geometry.Plane.WorldXY)

#if normalVector.X < 0: angle = (2\* math.pi) - angle

**else**: angle = 0

self.angle2North = math.**degrees**(angle)

**def findDiscontinuity**(self, curve, style):

# copied and modified from rhinoScript (@Steve Baer @GitHub)

"""Search for a derivatitive, tangent, or curvature discontinuity in

a curve object.

Parameters:

curve\_id = identifier of curve object

style = The type of continuity to test for. The types of

continuity are as follows:

Value Description

1 C0 - Continuous function

2 C1 - Continuous first derivative

3 C2 - Continuous first and second derivative

4 G1 - Continuous unit tangent

5 G2 - Continuous unit tangent and curvature

Returns:

List 3D points where the curve is discontinuous

"""

dom = curve.Domain

t0 = dom.Min

t1 = dom.Max

points = []

get\_next = **True**

**while** get\_next:

get\_next, t = curve.**GetNextDiscontinuity**(System.Enum.**ToObject**(rc.Geometry.Continuity, style), t0, t1)

**if** get\_next:

points.**append**(curve.**PointAt**(t))

t0 = t # Advance to the next parameter

return points

**def extractMeshPts**(self, mesh, triangulate = **False**):

coordinatesList = []

**for** face **in range**(mesh.Faces.Count):

# get each mesh surface vertices

**if** mesh.Faces.**GetFaceVertices**(face)[3] != mesh.Faces.**GetFaceVertices**(face)[4]:

meshVertices = mesh.Faces.**GetFaceVertices**(face)[1:5]

# triangulation

**if** triangulate **or not** self.**isRectangle**(meshVertices):

coordinatesList.**append**(meshVertices[:3])

coordinatesList.**append**([meshVertices[0], meshVertices[2], meshVertices[3]])

**else**:

coordinatesList.**append**(**list**(meshVertices))

**else**:

meshVertices = mesh.Faces.**GetFaceVertices**(face)[1:4]

coordinatesList.**append**(**list**(meshVertices))

# check order of the points

**for** coorCount, coorList **in enumerate**(coordinatesList):

# check if clockWise and reverse the list in case it is not

**if not** self.**isAntiClockWise**(coorList):

**try**: coorList.**reverse**()

**except**:

**try**: coordinatesList[coorCount] = [coorList[3], coorList[2], coorList[1], coorList[0]]

**except**: coordinatesList[coorCount] = [coorList[2], coorList[1], coorList[0]]

#coordinatesList.reverse()

return coordinatesList

**def isAntiClockWise**(self, pts):

**def crossProduct**(vector1, vector2):

return vector1.X \* vector2.X + vector1.Y \* vector2.Y + vector1.Z \* vector2.Z

# check if the order if clock-wise

vector0 = rc.Geometry.**Vector3d**(pts[1]- pts[0])

vector1 = rc.Geometry.**Vector3d**(pts[-1]- pts[0])

ptsNormal = rc.Geometry.Vector3d.**CrossProduct**(vector0, vector1)

# in case points are anti-clockwise then normals should be parallel

**if crossProduct**(ptsNormal, self.basePlane.Normal) > 0:

return **True**

return **False**

**def extractPoints**(self, method = 1, triangulate = **False**, meshPar = **None**, firstVertex = 'LowerLeftCorner'):

# if not self.meshedFace.IsValid:

# meshed surface will be generated regardless

# to make sure it won't fail for surfaces with multiple openings

**if** meshPar == **None**:

**if** self.isPlanar:

meshPar = rc.Geometry.MeshingParameters.Coarse

meshPar.SimplePlanes = **True**

**else**:

meshPar = rc.Geometry.MeshingParameters.Smooth

self.meshedFace = rc.Geometry.Mesh.**CreateFromBrep**(self.geometry, meshPar)[0]

**if** self.meshedFace.IsValid **or** self.hasInternalEdge:

**if** self.isPlanar **and not** self.hasInternalEdge:

plSegments = self.meshedFace.**GetNakedEdges**()

segments = []

[segments.**append**(seg.**ToNurbsCurve**()) **for** seg **in** plSegments]

**else**:

return self.**extractMeshPts**(self.meshedFace,triangulate)

**else**:

segments = self.geometry.**DuplicateEdgeCurves**(**True**)

joinedBorder = rc.Geometry.Curve.**JoinCurves**(segments)

**if** method == 0:

pts = []

[pts.**append**(seg.PointAtStart) **for** seg **in** segments]

**else**:

pts = []

pts.**append**(joinedBorder[0].PointAtStart)

restOfpts = self.**findDiscontinuity**(joinedBorder[0], style = 4)

# for some reason restOfPts returns no pt!

**try**: pts.**extend**(restOfpts)

**except**: pass

**try**: centPt, normalVector = self.**getSrfCenPtandNormalAlternate**()

**except**: centPt, normalVector = self.parent.**getSrfCenPtandNormal**(self.geometry)

basePlane = rc.Geometry.**Plane**(centPt, normalVector)

# inclusion test

**if str**(joinedBorder[0].**Contains**(centPt, basePlane)).**lower**() != "inside":

# average points

cumPt = rc.Geometry.**Point3d**(0,0,0)

**for** pt **in** pts: cumPt += pt

centPt = cumPt/**len**(pts)

# move basePlane to the new place

basePlane = rc.Geometry.**Plane**(centPt, normalVector)

# sort based on parameter on curve

pointsSorted = **sorted**(pts, key =**lambda** pt: joinedBorder[0].**ClosestPoint**(pt)[1])

# check if clockWise and reverse the list in case it is

**if not** self.**isAntiClockWise**(pointsSorted):

pointsSorted.**reverse**()

# in case the surface still doesn't have a type

# it happens for radiance surfaces. For EP it won't happen

# as it has been already assigned based on the zone

**if not hasattr**(self, 'type'):

self.Type = self.**getTypeByNormalAngle**()

## find UpperRightCorner point

## I'm changing this to find the LowerLeftCorner point

## instead as it is how gbXML needs it

# check the plane

srfType = self.**getTypeByNormalAngle**()

rotationCount = 0

**if** srfType == 0:

# vertical surface

**while** basePlane.YAxis.Z <= sc.doc.ModelAbsoluteTolerance **and** rotationCount < 3:

# keep rotating for 90 degrees

basePlane.**Rotate**(math.**radians**(90), basePlane.ZAxis)

rotationCount += 1

**elif** srfType == 1 **or** srfType == 3:

# roof + ceiling

**while** basePlane.YAxis.Y <= sc.doc.ModelAbsoluteTolerance **and** rotationCount < 3:

# keep rotating for 90 degrees

basePlane.**Rotate**(math.**radians**(90), basePlane.ZAxis)

rotationCount += 1

**elif** srfType == 2:

# floor

**while** basePlane.YAxis.Y >= sc.doc.ModelAbsoluteTolerance **and** rotationCount < 3:

# keep rotating for 90 degrees

basePlane.**Rotate**(math.**radians**(90), basePlane.ZAxis)

rotationCount += 1

# remap point on the new plane

remPts = []

**for** pt **in** pointsSorted: remPts.**append**(basePlane.**RemapToPlaneSpace**(pt)[1])

# find UpperRightCorner point (x>0 and max y)

firstPtIndex = **None**

**if** firstVertex == 'LowerLeftCorner':

**for** ptIndex, pt **in enumerate**(remPts):

**if** pt.X < 0 **and** pt.Y < 0 **and** firstPtIndex == **None**:

firstPtIndex = ptIndex #this could be the point

**elif** pt.X < 0 **and** pt.Y < 0:

**if** pt.Y < remPts[firstPtIndex].Y: firstPtIndex = ptIndex

**elif** firstVertex == 'UpperLeftCorner':

**for** ptIndex, pt **in enumerate**(remPts):

**if** pt.X < 0 **and** pt.Y > 0 **and** firstPtIndex == **None**:

firstPtIndex = ptIndex #this could be the point

**elif** pt.X < 0 **and** pt.Y > 0:

**if** pt.Y > remPts[firstPtIndex].Y: firstPtIndex = ptIndex

**elif** firstVertex == 'UpperRightCorner':

**for** ptIndex, pt **in enumerate**(remPts):

**if** pt.X > 0 **and** pt.Y > 0 **and** firstPtIndex == **None**:

firstPtIndex = ptIndex #this could be the point

**elif** pt.X > 0 **and** pt.Y > 0:

**if** pt.Y > remPts[firstPtIndex].Y: firstPtIndex = ptIndex

**elif** firstVertex == 'LowerRightCorner':

**for** ptIndex, pt **in enumerate**(remPts):

**if** pt.X > 0 **and** pt.Y < 0 **and** firstPtIndex == **None**:

firstPtIndex = ptIndex #this could be the point

**elif** pt.X > 0 **and** pt.Y < 0:

**if** pt.Y < remPts[firstPtIndex].Y: firstPtIndex = ptIndex

**if** firstPtIndex!=**None and** firstPtIndex!=0:

pointsSorted = pointsSorted[firstPtIndex:] + pointsSorted[:firstPtIndex]

return **list**(pointsSorted)

**def isRectangle**(self, ptList):

vector1 = rc.Geometry.**Vector3d**(ptList[0] - ptList[1])

vector2 = rc.Geometry.**Vector3d**(ptList[1] - ptList[2])

vector3 = rc.Geometry.**Vector3d**(ptList[2] - ptList[3])

vector4 = rc.Geometry.**Vector3d**(ptList[3] - ptList[0])

**if** ptList[0].**DistanceTo**(ptList[2]) != ptList[1].**DistanceTo**(ptList[3]) **or** \

math.**degrees**(rc.Geometry.Vector3d.**VectorAngle**(vector1, vector2))!= 90 **or** \

math.**degrees**(rc.Geometry.Vector3d.**VectorAngle**(vector3, vector4))!= 90:

return **False**

**else**:

return **True**

**def extractGlzPoints**(self, RAD = **False**, method = 2, firstVertex = 'LowerLeftCorner'):

glzCoordinatesList = []

**for** glzSrf **in** self.childSrfs:

sortedPoints = glzSrf.**extractPoints**(1, **False**, **None**, firstVertex)

# check numOfPoints

**if len**(sortedPoints) < 4 **or** (self.isPlanar **and** RAD==**True**):

glzCoordinatesList.**append**(sortedPoints) #triangle

**elif len**(sortedPoints) == 4 **and** self.isPlanar **and** self.**isRectangle**(sortedPoints):

glzCoordinatesList.**append**(sortedPoints) #rectangle

**else**:

**if** method == 1:

sortedPoints.**append**(sortedPoints[0])

border = rc.Geometry.**Polyline**(sortedPoints)

mesh = rc.Geometry.Mesh.**CreateFromClosedPolyline**(border)

**elif** method == 2:

mp = rc.Geometry.MeshingParameters.Smooth

mesh = rc.Geometry.Mesh.**CreateFromBrep**(glzSrf.geometry, mp)[0]

**if** mesh:

# Make sure non-rectangular shapes with 4 edges will be triangulated

**if len**(sortedPoints) == 4 **and** self.isPlanar: triangulate= **True**

**else**: triangulate= **False**

**try**: glzCoordinatesList.**extend**(self.**extractMeshPts**(mesh, triangulate))

**except**: glzCoordinatesList.**append**(self.**extractMeshPts**(mesh, triangulate))

return glzCoordinatesList

**def collectMeshFaces**(self, meshVertices, reverseList = **False**):

mesh = rc.Geometry.**Mesh**()

**if** meshVertices[3]!= meshVertices[4:]:

mesh.Vertices.**Add**(meshVertices[1]) #0

mesh.Vertices.**Add**(meshVertices[2]) #1

mesh.Vertices.**Add**(meshVertices[3]) #2

mesh.Vertices.**Add**(meshVertices[4]) #3

**if not** reverseList: mesh.Faces.**AddFace**(0, 1, 2, 3)

**else**: mesh.Faces.**AddFace**(0, 1, 2, 3)

**else**:

mesh.Vertices.**Add**(meshVertices[1]) #0

mesh.Vertices.**Add**(meshVertices[2]) #1

mesh.Vertices.**Add**(meshVertices[3]) #2

**if not** reverseList: mesh.Faces.**AddFace**(0, 1, 2)

**else**: mesh.Faces.**AddFace**(0, 1, 2)

self.meshedFace.**Append**(mesh)

#print self.meshedFace.Faces.Count

**def disposeCurrentMeshes**(self):

**if** self.meshedFace.Faces.Count>0:

self.meshedFace.**Dispose**()

self.meshedFace = rc.Geometry.**Mesh**()

**if** self.hasChild:

**for** fenSrf **in** self.childSrfs:

**if** fenSrf.meshedFace.Faces.Count>0:

fenSrf.meshedFace.**Dispose**()

fenSrf.meshedFace = rc.Geometry.**Mesh**()

**def getSrfCenPtandNormalAlternate**(self):

brepFace = self.geometry.Faces[0]

**if** brepFace.IsPlanar **and** brepFace.IsSurface:

u\_domain = brepFace.**Domain**(0)

v\_domain = brepFace.**Domain**(1)

centerU **=** (u\_domain.Min + u\_domain.Max)/2

centerV **=** (v\_domain.Min + v\_domain.Max)/2

centerPt = brepFace.**PointAt**(centerU, centerV)

normalVector = brepFace.**NormalAt**(centerU, centerV)

**else**:

centroid = rc.Geometry.AreaMassProperties.**Compute**(brepFace).Centroid

uv = brepFace.**ClosestPoint**(centroid)

centerPt = brepFace.**PointAt**(uv[1], uv[2])

normalVector = brepFace.**NormalAt**(uv[1], uv[2])

return centerPt, normalVector

**def isUnderground**(self, wall = **False**):

"""

check if this surface is underground

"""

# extract points

coordinatesList = self.**extractPoints**()

# create a list of list

**if type**(coordinatesList[0])**is not** list **and type**(coordinatesList[0]) **is not** tuple:

coordinatesList = [coordinatesList]

**for** ptList **in** coordinatesList:

**for** pt **in** ptList:

**if not** wall **and** pt.Z - rc.Geometry.Point3d.Origin.Z >= sc.doc.ModelAbsoluteTolerance: return **False**

**elif** pt.Z >= sc.doc.ModelAbsoluteTolerance: return **False**

return **True**

**def isOnGrade**(self):

"""

check if this surface is underground

"""

# extract points

coordinatesList = self.**extractPoints**()

# create a list of list

**if type**(coordinatesList[0])**is not** list **and type**(coordinatesList[0]) **is not** tuple:

coordinatesList = [coordinatesList]

**for** ptList **in** coordinatesList:

**for** pt **in** ptList:

**if abs**(pt.Z - rc.Geometry.Point3d.Origin.Z) >= sc.doc.ModelAbsoluteTolerance: return **False**

return **True**

**def reEvaluateType**(self, overwrite= **True**):

"""

Find special surface types

"""

**if not** overwrite **and hasattr**(self, "type"): return self.type

**if** self.srfTypeByUser: return self.type

**if** self.srfBCByUser: return self.type

# find initial type it has no type yet

**if not hasattr**(self, "type"):

self.type = self.**getTypeByNormalAngle**()

self.BC = "OUTDOORS"

**if** self.type == 0:

**if** self.**isUnderground**(**True**):

self.type += 0.5 #UndergroundWall

self.BC = "GROUND"

**elif** self.type == 1:

# A roof underground will be assigned as UndergroundCeiling!

**if** self.**isUnderground**():

self.type += 0.5 #UndergroundCeiling

self.BC = "GROUND"

**elif** self.BC.**upper**() == "SURFACE":

self.type == 3 # ceiling

**elif** self.type == 2:

# floor

**if** self.**isOnGrade**():

self.type += 0.5 #SlabOnGrade

self.BC = "GROUND"

**elif** self.**isUnderground**():

self.type += 0.25 #UndergroundSlab

self.BC = "GROUND"

**elif** self.BC.**upper**() != "SURFACE":

self.type += 0.75 #Exposed floor

# update boundary condition based on new type

self.BC = self.srfBC[self.type]

**def getTypeByNormalAngle**(self, maximumRoofAngle = 30):

# find the normal

**try**: findNormal = self.**getSrfCenPtandNormalAlternate**()

**except**: findNormal = self.parent.**getSrfCenPtandNormal**(self.geometry) #I should fix this at some point - Here just for shading surfaces for now

**if** findNormal:

**try**:

normal = findNormal[1]

angle2Z = math.**degrees**(rc.Geometry.Vector3d.**VectorAngle**(normal, rc.Geometry.Vector3d.ZAxis))

**except**:

**print** self

**print** rc.Geometry.AreaMassProperties.**Compute**(self.geometry).Centroid

angle2Z = 0

**else**:

#print findNormal

angle2Z = 0

**if** angle2Z < maximumRoofAngle **or** angle2Z > 360- maximumRoofAngle:

**try**:

**if** self.isThisTheTopZone:

return 1 #roof

**else**:

return 3 # ceiling

**except**:

return 1 #roof

**elif** 160 < angle2Z <200:

return 2 # floor

**else**:

return 0 #wall

**def transform**(self, transform, newKey=**None**, clearBC = **True**, flip = **False**):

"""Transform EPSurface using a transform object

Transform can be any valid transform object (e.g Translate, Rotate, Mirror)

"""

**try**:

**if** newKey == **None**:

self.name += **str**(uuid.**uuid4**())[:8]

**elif** newKey != **None**:

self.name += newKey

**except**:

pass

self.geometry.**Transform**(transform)

self.meshedFace.**Transform**(transform)

# move center point and normal

self.cenPt.**Transform**(transform)

self.normalVector.**Transform**(transform)

# move plane

self.basePlane.**Transform**(transform)

# Flip the normal if necessary

**if** flip:

self.normalVector.**Reverse**()

# Reset the angle to North

**try**:

self.**getAngle2North**()

**except**:

pass

**try**:

**if** clearBC:

self.**setBC**("Outdoors", **False**)

self.**setBCObjectToOutdoors**()

**elif** self.BCObject.name != '':

self.BCObject = copy.**deepcopy**(self.BCObject)

self.BCObject.name = self.BCObject.name + newKey

**except**:

pass

**if not** self.isChild **and** self.hasChild:

self.punchedGeometry.**Transform**(transform)

**if** flip: self.punchedGeometry.**Flip**()

**for** childSrf **in** self.childSrfs:

childSrf.**transform**(transform, newKey, clearBC, flip)

**def getTotalArea**(self, meterOverride=**False**):

**if** meterOverride == **True**:

return (self.geometry.**GetArea**())

**else**:

return (self.geometry.**GetArea**())\*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**def setType**(self, type, isUserInput = **False**):

self.type = type

self.srfTypeByUser = isUserInput

**def setBC**(self, BC, isUserInput = **False**):

self.BC = BC

self.srfBCByUser = isUserInput

**def setBCObject**(self, BCObject):

self.BCObject = BCObject

**def setBCObjectToOutdoors**(self):

self.BCObject = self.**outdoorBCObject**()

**def setEPConstruction**(self, EPConstruction):

self.EPConstruction = EPConstruction

**def setRADMaterial**(self, RADMaterial):

self.RadMaterial = RADMaterial

**def setName**(self, newName, isUserInput = **False**):

self.name = newName

self.srfNameByUser = isUserInput

**def setSunExposure**(self, exposure = 'NoSun'):

self.sunExposure = exposure

**def setWindExposure**(self, exposure = 'NoWind'):

self.windExposure = exposure

**def getArea**(self):

return rc.Geometry.AreaMassProperties.**Compute**(self.geometry).Area \*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**def \_\_str\_\_**(self):

**try**:

return 'Surface name: ' + self.name + '\nSurface number: ' + **str**(self.num) + \

'\nThis surface is a ' + **str**(self.srfType[self.type]) + "."

**except**:

return 'Surface name: ' + self.name + '\n' + 'Surface number: ' + **str**(self.num) + \

'\nSurface type is not assigned. Honeybee thinks this is a ' + **str**(self.srfType[self.**getTypeByNormalAngle**()]) + "."

**class hb\_EPZoneSurface**(hb\_EPSurface):

"""..."""

**def \_\_init\_\_**(self, surface, srfNumber, srfName, \*args):

"""This function initiates the class for an EP surface.

surface: surface geometry as a Brep

srfNumber: a unique number that is only for this surface

srfName: the unique name for this surface

parentZone: class of the zone that this surface belongs to"""

**if len**(args)==2:

parentZone, surafceType = args

hb\_EPSurface.**\_\_init\_\_**(self, surface, srfNumber, srfName, parentZone, surafceType)

self.**getAngle2North**()

self.BCObject = self.**outdoorBCObject**()

**else**:

hb\_EPSurface.**\_\_init\_\_**(self, surface, srfNumber, srfName)

# Check for possible surface type and assign the BC based on that

# This will be re-evaluated in write idf file

srfType = self.**getTypeByNormalAngle**()

self.BC = self.srfBC[srfType]

self.BCObject = self.**outdoorBCObject**()

self.sunExposure = self.srfSunExposure[srfType]

self.windExposure = self.srfWindExposure[srfType]

self.**getAngle2North**()

**if hasattr**(self, 'parent') **and** self.parent!=**None**:

# in both of this cases the zone should be meshed

**if not** self.isPlanar:

self.parent.hasNonPlanarSrf = **True**

**if** self.hasInternalEdge:

self.parent.hasInternalEdge = **True**

**if hasattr**(self, 'type'):

self.sunExposure = self.srfSunExposure[self.type]

self.windExposure = self.srfWindExposure[self.type]

self.groundViewFactor = 'autocalculate'

self.hasChild = **False**

self.isChild = **False**

self.childSrfs = []

**def isPossibleChild**(self, chidSrfCandidate, tolerance = sc.doc.ModelAbsoluteTolerance):

# check if all the vertices has 0 distance with the base surface

segments = chidSrfCandidate.**DuplicateEdgeCurves**(**True**)

pts = []

[pts.**append**(seg.PointAtStart) **for** seg **in** segments]

**for** pt **in** pts:

ptOnSrf = self.geometry.**ClosestPoint**(pt)

**if** pt.**DistanceTo**(ptOnSrf) > tolerance: return **False**

# all points are located on the surface and the area is less so it is all good!

return **True**

**def addChildSrf**(self, childSurface, percentage = 40):

# I should copy/paste the function here so I can run it as

# a method! For now I just collect them here together....

# use the window function

**try**: self.childSrfs.**extend**(childSurface)

**except**: self.childSrfs.**append**(childSurface)

self.hasChild = **True**

pass

**def calculatePunchedSurface**(self):

**def checkCrvArea**(crv):

**try**:

area = rc.Geometry.AreaMassProperties.**Compute**(crv).Area \*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**except**:

area = 0

return area > sc.doc.ModelAbsoluteTolerance

**def checkCrvsPts**(crv):

# in some cases crv generates a line with similar points

pts = []

pts.**append**(crv.PointAtStart)

restOfpts = self.**findDiscontinuity**(crv, style = 4)

# for some reason restOfPts returns no pt!

**try**: pts.**extend**(restOfpts)

**except**: pass

**def isDuplicate**(pt, newPts):

**for** p **in** newPts:

# print pt.DistanceTo(p)

**if** pt.**DistanceTo**(p) < 2 \* sc.doc.ModelAbsoluteTolerance:

return **True**

return **False**

newPts = [pts[0]]

**for** pt **in** pts[1:]:

**if not isDuplicate**(pt, newPts):

newPts.**append**(pt)

**if len**(newPts) > 2:

return **True**

return **False**

glzCrvs = []

childSrfs = []

**for** glzSrf **in** self.childSrfs:

glzEdges = glzSrf.geometry.**DuplicateEdgeCurves**(**True**)

jGlzCrv = rc.Geometry.Curve.**JoinCurves**(glzEdges)[0]

# in some cases glazing based on percentage generates very small glazings

# here I check and remove them

# check area of curve

**try**:

**if** self.isPlanar:

area = rc.Geometry.AreaMassProperties.**Compute**(jGlzCrv).Area \*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**else**:

area = rc.Geometry.AreaMassProperties.**Compute**(glzSrf.geometry).Area \*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**except**:

# in case area calulation fails

# let it go anyways!

area = 10 \* sc.doc.ModelAbsoluteTolerance

**if abs**(area) > sc.doc.ModelAbsoluteTolerance **and checkCrvsPts**(jGlzCrv):

# check normal direction of child surface and base surface

# print math.degrees(rc.Geometry.Vector3d.VectorAngle(glzSrf.normalVector, self.normalVector))

childSrfs.**append**(glzSrf)

glzCrvs.**append**(jGlzCrv)

**else**:

**print** "A very tiny glazing is removed from " + self.name+ "."

self.childSrfs = childSrfs

baseEdges = self.geometry.**DuplicateEdgeCurves**(**True**)

jBaseCrv = rc.Geometry.Curve.**JoinCurves**(baseEdges)

# convert array to list

jBaseCrvList = **list**(jBaseCrv)

**try**:

**if** self.isPlanar:

# works for planar surfaces

punchedGeometries = rc.Geometry.Brep.**CreatePlanarBreps**(glzCrvs + jBaseCrvList)

**if len**(punchedGeometries) == 1:

self.punchedGeometry = punchedGeometries[0]

**else**:

# curves are not in the same plane so let's

# project the curves on surface plane

srfPlane = rc.Geometry.**Plane**(self.cenPt, self.normalVector)

PGlzCrvs = []

**for** curve **in** glzCrvs + jBaseCrvList:

pCrv = rc.Geometry.Curve.**ProjectToPlane**(curve, srfPlane)

**if** checkCrvArea:

PGlzCrvs.**append**(pCrv)

punchedGeometries = rc.Geometry.Brep.**CreatePlanarBreps**(PGlzCrvs)

# in some cases glazing with very minor areas are generated

# which causes multiple surfaces

self.punchedGeometry = punchedGeometries[-1]

**else**:

# split the base geometry - Good luck!

splitBrep = self.geometry.Faces[0].**Split**(glzCrvs, sc.doc.ModelAbsoluteTolerance)

#splitBrep.Faces.ShrinkFaces()

**for** srfCount **in range**(splitBrep.Faces.Count):

surface = splitBrep.Faces.**ExtractFace**(srfCount)

edges = surface.**DuplicateEdgeCurves**(**True**)

joinedEdges = rc.Geometry.Curve.**JoinCurves**(edges)

**if len**(joinedEdges)>1:

self.punchedGeometry = surface

**except** Exception, e:

self.punchedGeometry = **None**

self.hasChild = **False**

self.childSrfs = []

**print** "Failed to calculate opaque part of the surface. " + \

"Glazing is removed from " + self.name

**def getOpaqueArea**(self):

**if** self.hasChild:

**try**:

return self.punchedGeometry.**GetArea**()\*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**except**:

self.**calculatePunchedSurface**()

return self.punchedGeometry.**GetArea**()\*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

**else**:

return self.**getTotalArea**()

**def getGlazingArea**(self):

**if** self.hasChild:

glzArea = 0

**for** childSrf **in** self.childSrfs:

glzArea += childSrf.**getTotalArea**()

return glzArea

**else**:

return 0

**def getWWR**(self):

return self.**getGlazingArea**()/self.**getTotalArea**()

**def removeAllChildSrfs**(self):

self.childSrfs = []

self.hasChild = **False**

self.**calculatePunchedSurface**()

**class hb\_EPShdSurface**(hb\_EPSurface):

**def \_\_init\_\_**(self, surface, srfNumber, srfName):

hb\_EPSurface.**\_\_init\_\_**(self, surface, srfNumber, srfName, self)

self.PVgenlist = **None**

self.containsPVgen = **None**

self.TransmittanceSCH = ''

self.isChild = **False**

self.hasChild = **False**

self.construction = 'Exterior Wall' # just added here to get the minimum surface to work

self.EPConstruction = 'Exterior Wall' # just added here to get the minimum surface to work

self.childSrfs = [self] # so I can use the same function as glazing to extract the points

self.type = 6

pass

**def getSrfCenPtandNormal**(self, surface):

brepFace = surface.Faces[0]

**if** brepFace.IsPlanar **and** brepFace.IsSurface:

u\_domain = brepFace.**Domain**(0)

v\_domain = brepFace.**Domain**(1)

centerU **=** (u\_domain.Min + u\_domain.Max)/2

centerV **=** (v\_domain.Min + v\_domain.Max)/2

centerPt = brepFace.**PointAt**(centerU, centerV)

normalVector = brepFace.**NormalAt**(centerU, centerV)

**else**:

centroid = rc.Geometry.AreaMassProperties.**Compute**(brepFace).Centroid

uv = brepFace.**ClosestPoint**(centroid)

centerPt = brepFace.**PointAt**(uv[1], uv[2])

normalVector = brepFace.**NormalAt**(uv[1], uv[2])

return centerPt, normalVector

**class hb\_EPFenSurface**(hb\_EPSurface):

"""..."""

**def \_\_init\_\_**(self, surface, srfNumber, srfName, parentSurface, surafceType, punchedWall = **None**):

"""This function initiates the class for an EP surface.

surface: surface geometry as a Brep

srfNumber: a unique number that is only for this surface

srfName: the unique name for this surface

parentZone: class of the zone that this surface belongs to"""

hb\_EPSurface.**\_\_init\_\_**(self, surface, srfNumber, srfName, parentSurface, surafceType)

# Special inputs for shading control.

self.shadingSchName = []

self.shadingControlName = []

self.shadeMaterialName = []

**if not** self.isPlanar:

**try**:

self.parent.parent.hasNonplanarSrf = **True**

**except**:

# surface is not part of a zone yet.

pass

# calculate punchedWall

self.parent.punchedGeometry = punchedWall

self.frameName = ''

self.Multiplier = 1

self.BCObject = self.**outdoorBCObject**()

self.groundViewFactor = 'autocalculate'

self.isChild = **True** # is it really useful?

**class hb\_GlzGeoGeneration**(object):

**def \_\_init\_\_**(self):

self.tol = sc.doc.ModelAbsoluteTolerance

**def getRestOfSurfacePlanar**(self, baseSrf, glazing):

selfDir = baseSrf.Faces[0].**NormalAt**(0,0)

glzCrvs = []

**for** glzSrf **in** glazing:

glzEdges = glzSrf.**DuplicateEdgeCurves**(**True**)

jGlzCrv = rc.Geometry.Curve.**JoinCurves**(glzEdges)[0]

glzCrvs.**append**(jGlzCrv)

baseEdges = baseSrf.**DuplicateEdgeCurves**(**True**)

jBaseCrv = rc.Geometry.Curve.**JoinCurves**(baseEdges)

# convert array to list

jBaseCrvList = []

**for** crv **in** jBaseCrv: jBaseCrvList.**append**(crv)

**try**:

punchedGeometries = rc.Geometry.Brep.**CreatePlanarBreps**(glzCrvs + jBaseCrvList)

**if len**(punchedGeometries)>1:

crvDif = rc.Geometry.Curve.**CreateBooleanDifference**(jBaseCrvList[0], glzCrvs)

punchedGeometries = rc.Geometry.Brep.**CreatePlanarBreps**(crvDif)

punchedGeometryDir = punchedGeometries[0].Faces[0].**NormalAt**(0,0)

**if** punchedGeometryDir.X < selfDir.X + self.tol **and** punchedGeometryDir.X > selfDir.X - self.tol **and** punchedGeometryDir.Y < selfDir.Y + self.tol **and** punchedGeometryDir.Y > selfDir.Y - self.tol **and** punchedGeometryDir.Z < selfDir.Z + self.tol **and** punchedGeometryDir.Z > selfDir.Z - self.tol:

pass

**else**: punchedGeometries[0].**Flip**()

return punchedGeometries[0]

**except** Exception, e:

return []

**print** "failed to calculate opaque part of the surface:\n" + `e`

**def getTopBottomCurves**(self, brep):

#Write a function to find if a given line is horizontal or vertical.

**def isEdgeHorizontal**(edge):

**if** edge.PointAtStart.Z **<** (edge.PointAtEnd.Z + sc.doc.ModelAbsoluteTolerance) **and** edge.PointAtStart.Z **>** (edge.PointAtEnd.Z - sc.doc.ModelAbsoluteTolerance):

return **True**

**else**:

return **False**

**def isEdgeVertical**(edge):

**if** edge.PointAtStart.X **<** (edge.PointAtEnd.X + sc.doc.ModelAbsoluteTolerance) **and** edge.PointAtStart.X **>** (edge.PointAtEnd.X - sc.doc.ModelAbsoluteTolerance) **and** edge.PointAtStart.Y **<** (edge.PointAtEnd.Y + sc.doc.ModelAbsoluteTolerance) **and** edge.PointAtStart.Y **>** (edge.PointAtEnd.Y - sc.doc.ModelAbsoluteTolerance):

return **True**

**else**:

return **False**

# duplicate the edges of the wall

edges = brep.**DuplicateEdgeCurves**(**True**)

# sort the edges based on the z of mid point of the edge and get the top and bottom edges.

sortedEdges = **sorted**(edges, key=**lambda** edge: edge.**PointAtNormalizedLength**(0.5).Z)

btmEdge = sortedEdges[0]

isBtmHorizontal = **isEdgeHorizontal**(btmEdge)

topEdge = sortedEdges[-1]

isTopHorizontal = **isEdgeHorizontal**(topEdge)

#Test to see if any of the side edges are vertical and, if there are two, there may be a rectangle that we can pull out.

vertEdges = []

nonVertEdge = []

**for** edge **in** sortedEdges:

**if isEdgeVertical**(edge) == **True**:

vertEdges.**append**(edge)

**else**: nonVertEdge.**append**(edge)

**if len**(vertEdges) == 2:

are2LinesVert = **True**

**else**: are2LinesVert = **False**

return btmEdge, isBtmHorizontal, topEdge, isTopHorizontal, vertEdges, are2LinesVert

**def getCurvePoints**(self, curve):

exploCurve = rc.Geometry.PolyCurve.**DuplicateSegments**(curve)

individPts = []

**for** line **in** exploCurve:

individPts.**append**(line.PointAtStart)

return individPts

**def cleanCurve**(self, curve):

#Define a function that cleans up curves by deleting out points that lie in a line and leaves the curved segments intact. Also, have it delete out any segments that are shorter than the tolerance.

#First check if there are any curved segements and make a list to keep track of this

curveBool = **False**

exploCurve = rc.Geometry.PolyCurve.**DuplicateSegments**(curve)

**for** segment **in** exploCurve:

**if** segment.**IsLinear**() == **False**: curveBool = **True**

**else**: pass

# Get the curve points.

curvePts = self.**getCurvePoints**(curve)

**if** curveBool == **False**:

#Test if any of the points lie in a line and use this to generate a new list of curve segments and points.

newPts = []

newSegments = []

**for** pointCount, point **in enumerate**(curvePts):

testLine = rc.Geometry.**Line**(point, curvePts[pointCount-2])

**if** testLine.**DistanceTo**(curvePts[pointCount-1], **True**) > self.tol **and len**(newPts) == 0:

newPts.**append**(curvePts[pointCount-1])

**elif** testLine.**DistanceTo**(curvePts[pointCount-1], **True**) > self.tol **and len**(newPts) != 0:

newSegments.**append**(rc.Geometry.**LineCurve**(newPts[-1], curvePts[pointCount-1]))

newPts.**append**(curvePts[pointCount-1])

**else**: pass

#Add a segment to close the curves and join them together into one polycurve.

newSegments.**append**(rc.Geometry.**LineCurve**(newPts[-1], newPts[0]))

#Shift the lists over by 1 to ensure that the order of the points and curves match the input

newCurvePts = newPts[1:]

newCurvePts.**append**(newPts[0])

newCurveSegments = newSegments[1:]

newCurveSegments.**append**(newSegments[0])

#Join the segments together into one curve.

newCrv = rc.Geometry.**PolyCurve**()

**for** seg **in** newCurveSegments:

newCrv.**Append**(seg)

newCrv.**MakeClosed**(self.tol)

**else**:

newCrv = curve

#return the new curve and the list of points associated with it.

return newCrv

**def createGlazingTri**(self, triSrf, glazingRatio, scalePt):

#Calculate the center point if one is not provided.

**if** scalePt:

cenPt = scalePt

**else**:

cenPt = rc.Geometry.AreaMassProperties.**Compute**(triSrf).Centroid

#Scale the wall geometry to get to the appropriate glazingRatio.

scaleFactor = glazingRatio \*\* .5

scaleT = rc.Geometry.Transform.**Scale**(cenPt, scaleFactor)

glzSrfBrep = triSrf.**DuplicateBrep**()

glzSrfBrep.**Transform**(scaleT)

glzSrf = [glzSrfBrep]

return glzSrf

**def createGlazingQuad**(self, quadSrf, glazingRatio, scalePt):

#Calculate the center point if one is not provided.

**if** scalePt:

cenPt = scalePt

**else**:

cenPt = rc.Geometry.AreaMassProperties.**Compute**(quadSrf).Centroid

#Check to see if the center point of the quadrilaterial is inside the quadrilateral (which means that we can just scale the quadrilateral and the result will be inside it).

cenPt = rc.Geometry.AreaMassProperties.**Compute**(quadSrf).Centroid

closestPt = quadSrf.**ClosestPoint**(cenPt)

**if** cenPt.X **<** (closestPt.X + sc.doc.ModelAbsoluteTolerance) **and** cenPt.X **>** (closestPt.X - sc.doc.ModelAbsoluteTolerance) **and** cenPt.Y **<** (closestPt.Y + sc.doc.ModelAbsoluteTolerance) **and** cenPt.Y **>** (closestPt.Y - sc.doc.ModelAbsoluteTolerance) **and** cenPt.Z **<** (closestPt.Z + sc.doc.ModelAbsoluteTolerance) **and** cenPt.Z **>** (closestPt.Z - sc.doc.ModelAbsoluteTolerance):

checkCent = **True**

**else**:

checkCent = **False**

#If the polygon's center point lies within the polygon, use the typical scaling method to get the window.

**if** checkCent == **True**:

scaleFactor = glazingRatio \*\* .5

scaleT = rc.Geometry.Transform.**Scale**(cenPt, scaleFactor)

glzSrfBrep = quadSrf.**DuplicateBrep**()

glzSrfBrep.**Transform**(scaleT)

glzSrf = [glzSrfBrep]

#If the polygon's center point lies outside of the polygon, split the polygon into two triangles and scale each to its own center.

**else**:

pts = quadSrf.**DuplicateVertices**()

diagonal1 = rc.Geometry.Brep.**CreateFromCornerPoints**(pts[0], pts[1], pts[2], sc.doc.ModelAbsoluteTolerance)

diagonal2 = rc.Geometry.Brep.**CreateFromCornerPoints**(pts[1], pts[2], pts[3], sc.doc.ModelAbsoluteTolerance)

quadSrfSplit1 = rc.Geometry.Brep.**Split**(quadSrf, diagonal1, sc.doc.ModelAbsoluteTolerance)

quadSrfSplit2 = rc.Geometry.Brep.**Split**(quadSrf, diagonal2, sc.doc.ModelAbsoluteTolerance)

quadSrfSplit = quadSrfSplit1 + quadSrfSplit2

glzSrf = []

**for** brep **in** quadSrfSplit:

glzSrf.**append**(self.**createGlazingTri**(brep, glazingRatio, **None**)[0])

return glzSrf

**def createGlazingOddPlanarGeo**(self, baseSrf, glazingRatio):

#Define the meshing paramters to break down the surface in a manner that produces only trinagles and quads.

meshPar = rc.Geometry.MeshingParameters.Default

#Create a mesh of the base surface.

windowMesh = rc.Geometry.Mesh.**CreateFromBrep**(baseSrf, meshPar)[0]

#Create breps of all of the mesh faces and use them to make each window.

glzSrf = []

srfFaceList = windowMesh.Faces

srfVertList = windowMesh.Vertices

srfFaceCen = []

**for** faceNum, face **in enumerate**(srfFaceList):

**if** face.IsQuad == **True**:

glzSrf.**append**(self.**createGlazingQuad**(rc.Geometry.Brep.**CreateFromCornerPoints**(srfVertList[face[0]], srfVertList[face[1]], srfVertList[face[2]], srfVertList[face[3]], sc.doc.ModelAbsoluteTolerance), glazingRatio, windowMesh.Faces.**GetFaceCenter**(faceNum))[0])

**else**:

glzSrf.**append**(self.**createGlazingTri**(rc.Geometry.Brep.**CreateFromCornerPoints**(srfVertList[face[0]], srfVertList[face[1]], srfVertList[face[2]], sc.doc.ModelAbsoluteTolerance), glazingRatio, windowMesh.Faces.**GetFaceCenter**(faceNum))[0])

return glzSrf

**def createGlazingForRect**(self, rectBrep, glazingRatio, windowHeight, sillHeight, breakUpDist, splitGlzVertDist, conversionFactor):

#Define a default window height, sill height, breakup distance and vertical glazing dist of windows.

**if** windowHeight != **None**: winHeight = windowHeight

**else**: winHeight = 2/conversionFactor

**if** sillHeight != **None**: silHeight = sillHeight

**else**: silHeight = 0.8/conversionFactor

**if** breakUpDist != **None**: distBreakup = breakUpDist

**else**: distBreakup = 2/conversionFactor

**if** splitGlzVertDist != **None**: splitVertDist = splitGlzVertDist

**else**: splitVertDist = 0/conversionFactor

**if** rectBrep:

#Calculate the target area to make the glazing.

targetArea **=** (rc.Geometry.AreaMassProperties.**Compute**(rectBrep).Area) \* glazingRatio

#Find the maximum acceptable area for breaking up the window into smaller, taller windows.

rectBtmCurve = self.**getTopBottomCurves**(rectBrep)[0]

rectTopCurve = self.**getTopBottomCurves**(rectBrep)[2]

maxAreaBreakUp **=** (rectBtmCurve.**GetLength**() \* 0.98) \* winHeight

#Find the maximum acceptable area for setting the glazing at the sill height.

heightClosestPt = rc.Geometry.Curve.**PointAt**(rectTopCurve, rc.Geometry.LineCurve.**ClosestPoint**(rectTopCurve, rectBtmCurve.PointAtEnd)[1])

rectHeight = rc.Geometry.Point3d.**DistanceTo**(heightClosestPt, rectBtmCurve.PointAtEnd)

rectHeightVec = rc.Geometry.**Vector3d**(heightClosestPt.X - rectBtmCurve.PointAtEnd.X, heightClosestPt.Y - rectBtmCurve.PointAtEnd.Y, heightClosestPt.Z - rectBtmCurve.PointAtEnd.Z)

maxWinHeightSill = rectHeight - silHeight

#If the window height given from the formulas above is greater than the height of the wall, set the window height to be just under that of the wall.

**if** winHeight **>** (0.98 \* rectHeight): winHeightFinal **=** (0.98 \* rectHeight)

**else**: winHeightFinal = winHeight

#If the sill height given from the formulas above is less than 1% of the wall height, set the sill height to be 1% of the wall height.

**if** silHeight **<** (0.01 \* rectHeight): silHeightFinal **=** (0.01 \* rectHeight)

**else**: silHeightFinal = silHeight

#Find the window geometry in the case that the target area is below that of the maximum acceptable area for breaking up the window into smaller, taller windows.

**if** targetArea < maxAreaBreakUp:

#Divide up the rectangle into points on the bottom.

rectBtmCurveLength = rectBtmCurve.**GetLength**()

**if** rectBtmCurveLength **>** (distBreakup/2):

numDivisions = **round**(rectBtmCurveLength/distBreakup, 0)

**else**:

numDivisions = 1

btmDivPts = []

rectBtmCurve.**Reverse**()

#print numDivisions

**for** parameter **in** rectBtmCurve.**DivideByCount**(numDivisions, **True**):

btmDivPts.**append**(rc.Geometry.Curve.**PointAt**(rectBtmCurve, parameter))

#Connect the points to form lines to be used to generate the windows

winLinesStart = []

ptIndex = 0

**for** point **in** btmDivPts:

**if** ptIndex < numDivisions:

winLinesStart.**append**(rc.Geometry.**Line**(point, btmDivPts[ptIndex+1]))

ptIndex += 1

#Move the lines to the appropriate sill height.

sillUnitVec = rectHeightVec

sillUnitVec.**Unitize**()

maxSillHeight **=** (rectHeight\*0.99) - winHeightFinal

**if** silHeightFinal < maxSillHeight: sillVec = rc.Geometry.Vector3d.**Multiply**(silHeightFinal, sillUnitVec)

**else**: sillVec = rc.Geometry.Vector3d.**Multiply**(maxSillHeight, sillUnitVec)

transformMatrix = rc.Geometry.Transform.**Translation**(sillVec)

**for** line **in** winLinesStart:

rc.Geometry.Line.**Transform**(line, transformMatrix)

#Scale the lines to their center points based on the width that they need to be to satisfy the glazing ratio.

lineCentPt = []

**for** line **in** winLinesStart:

lineCentPt.**append**(line.**PointAt**(0.5))

winLineBaseLength = winLinesStart[0].Length

winLineReqLength **=** (targetArea / winHeightFinal) / numDivisions

winLineScale = winLineReqLength / winLineBaseLength

centPtIndex = 0

**for** line **in** winLinesStart:

transformMatrixScale = rc.Geometry.Transform.**Scale**(lineCentPt[centPtIndex], winLineScale)

line.**Transform**(transformMatrixScale)

centPtIndex += 1

#Find the maximum acceptable area for splitting the glazing vertically.

maxSplitVert = rectHeight - silHeightFinal - winHeightFinal **-** (0.02\*rectHeight)

#If the splitVertDist is beyond the maximum acceptable, set it to this maximum.

**if** splitVertDist < 0 **or** maxSplitVert < 0: splitVertDist = 0

**elif** splitVertDist != 0 **and** splitVertDist > maxSplitVert: splitVertDist = maxSplitVert

#If there is a non-zero vertical breakup dist and the value is less than the maximum accpetable, break up the window surface verticaly.

**if** splitVertDist != 0:

#Extrude the lines to create the windows

extruUnitVec = rectHeightVec

extruUnitVec.**Unitize**()

extruVec = rc.Geometry.Vector3d.**Multiply**(extruUnitVec**,** (winHeightFinal/2))

vertMovingVec = rc.Geometry.Vector3d.**Multiply**(extruUnitVec**,** (winHeightFinal/2)+splitVertDist)

vertMovingTransform = rc.Geometry.Transform.**Translation**(vertMovingVec)

finalWinSrfs = []

**for** line **in** winLinesStart:

finalWinSrfs.**append**(rc.Geometry.Surface.**CreateExtrusion**(line.**ToNurbsCurve**(), extruVec))

line.**Transform**(vertMovingTransform)

finalWinSrfs.**append**(rc.Geometry.Surface.**CreateExtrusion**(line.**ToNurbsCurve**(), extruVec))

**else**:

#Extrude the lines to create the windows

extruUnitVec = rectHeightVec

extruUnitVec.**Unitize**()

extruVec = rc.Geometry.Vector3d.**Multiply**(extruUnitVec, winHeightFinal)

finalWinSrfs = []

**for** line **in** winLinesStart:

finalWinSrfs.**append**(rc.Geometry.Surface.**CreateExtrusion**(line.**ToNurbsCurve**(), extruVec))

rectWinBreps=[]

**for** srf **in** finalWinSrfs:

rectWinBreps.**append**(rc.Geometry.Surface.**ToBrep**(srf))

#Find the window geometry in the case that the target area is above that of the maximum acceptable area for breaking up the window in which case we have to make one big window.

**if** targetArea > maxAreaBreakUp:

#Move the bottom curve of the window to the appropriate sill height.

sillUnitVec = rectHeightVec

sillUnitVec.**Unitize**()

rectBtmCurveLength = rectBtmCurve.**GetLength**()

maxSillHeight **=** (rectHeight\*0.99) **-** (targetArea **/** (rectBtmCurveLength \* 0.98))

**if** silHeightFinal < maxSillHeight:

sillVec = rc.Geometry.Vector3d.**Multiply**(silHeightFinal, sillUnitVec)

**else**:

sillVec = rc.Geometry.Vector3d.**Multiply**(maxSillHeight, sillUnitVec)

#Move the window to the sill height.

transformMatrix = rc.Geometry.Transform.**Translation**(sillVec)

winStartLine = rectBtmCurve

rc.Geometry.NurbsCurve.**Transform**(winStartLine, transformMatrix)

#Scale the curve so that it is not touching the edges of the surface.

lineCentPt = rc.Geometry.**Point3d((**(winStartLine.PointAtStart.X + winStartLine.PointAtEnd.X)/2)**, (**(winStartLine.PointAtStart.Y + winStartLine.PointAtEnd.Y)/2)**, (**(winStartLine.PointAtStart.Z + winStartLine.PointAtEnd.Z)/2))

transformMatrixScale = rc.Geometry.Transform.**Scale**(lineCentPt, 0.98)

winStartLine.**Transform**(transformMatrixScale)

#Find the maximum acceptable area for splitting the glazing vertically.

maxSplitVert = rectHeight - silHeightFinal **-** (targetArea **/** (rectBtmCurveLength \* 0.98)) **-** (0.02\*rectHeight)

#If the splitVertDist is beyond the maximum acceptable, set it to this maximum.

**if** splitVertDist < 0 **or** maxSplitVert < 0: splitVertDist = 0

**elif** splitVertDist != 0 **and** splitVertDist > maxSplitVert: splitVertDist = maxSplitVert

**if** splitVertDist != 0:

#Extrude the line to create the window

extruUnitVec = rectHeightVec

extruUnitVec.**Unitize**()

extruVec = rc.Geometry.Vector3d.**Multiply**(extruUnitVec**,** (targetArea **/** (rectBtmCurveLength \* 0.98))/2)

vertMovingVec = rc.Geometry.Vector3d.**Multiply**(extruUnitVec**, (**(targetArea **/** (rectBtmCurveLength \* 0.98))/2)+splitVertDist)

vertMovingTransform = rc.Geometry.Transform.**Translation**(vertMovingVec)

finalWinSrf1 = rc.Geometry.Surface.**CreateExtrusion**(winStartLine.**ToNurbsCurve**(), extruVec)

winStartLine.**Transform**(vertMovingTransform)

finalWinSrf2 = rc.Geometry.Surface.**CreateExtrusion**(winStartLine.**ToNurbsCurve**(), extruVec)

rectWinBreps = [rc.Geometry.Surface.**ToBrep**(finalWinSrf1), rc.Geometry.Surface.**ToBrep**(finalWinSrf2)]

**else**:

**if** (sc.doc.ModelAbsoluteTolerance > 0.01\* rectBtmCurveLength):

warning = "Your model tolerance is too high and for this reason the base surface is being split into two \n" + \

"instead of making a window in the base surface! Lower your model tolerance or decrease your glazing ratio to fix this issue"

**print** warning

#Extrude the line to create the window

extruUnitVec = rectHeightVec

extruUnitVec.**Unitize**()

extruVec = rc.Geometry.Vector3d.**Multiply**(extruUnitVec**,** (targetArea **/** (rectBtmCurveLength \* 0.98)))

finalWinSrf = rc.Geometry.Surface.**CreateExtrusion**(winStartLine, extruVec)

rectWinBreps = [rc.Geometry.Surface.**ToBrep**(finalWinSrf)]

**else**:

rectWinBreps = []

return rectWinBreps

**def createGlazingThatContainsRectangle**(self, topEdge, btmEdge, baseSrf, glazingRatio, windowHeight, sillHeight, breakUpWindow, breakUpDist, splitVertDist, conversionFactor):

#Get the rectangle vertices points from the arrangement of closest points of the top and bottom curves.

rectPt1 = rc.Geometry.Curve.**PointAt**(topEdge, rc.Geometry.LineCurve.**ClosestPoint**(topEdge, btmEdge.PointAtEnd)[1])

rectPt2 = rc.Geometry.Curve.**PointAt**(btmEdge, rc.Geometry.LineCurve.**ClosestPoint**(btmEdge, topEdge.PointAtEnd)[1])

rectPt3 = rc.Geometry.Curve.**PointAt**(topEdge, rc.Geometry.LineCurve.**ClosestPoint**(topEdge, btmEdge.PointAtStart)[1])

rectPt4 = rc.Geometry.Curve.**PointAt**(btmEdge, rc.Geometry.LineCurve.**ClosestPoint**(btmEdge, topEdge.PointAtStart)[1])

#Create the rectangle

rectPlane = rc.Geometry.**Plane**(rectPt4, rectPt2, rectPt3)

rect = rc.Geometry.**Rectangle3d**(rectPlane, rectPt2, rectPt1)

rectBrep = rc.Geometry.Brep.**CreateFromCornerPoints**(rectPt1, rectPt3, rectPt2, rectPt4, sc.doc.ModelAbsoluteTolerance)

**def areEdgesLinear**(brepList):

**for** srf **in** brepList:

**for** edge **in** srf.Edges:

**if not** edge.**IsLinear**():

return **False**

return **True**

#Split the base surface with the rectangle

**if** rectBrep:

srfSplit = rc.Geometry.Brep.**Split**(baseSrf, rectBrep, sc.doc.ModelAbsoluteTolerance)

# make sure split doesn't generate curves shapes!

# happens for some strange surfaces:

# https://github.com/mostaphaRoudsari/Honeybee/issues/115

**if** srfSplit!=[] **and not areEdgesLinear**(srfSplit): srfSplit =[]

**else**:

srfSplit = []

**if len**(srfSplit) == 0:

**if** rectBrep:

srfSplit = [baseSrf]

**else**:

srfSplit = []

middle = []

sides = []

#Determine which Breps are rectangular and which are not by testing their angles and number of sides.

middle = []

sides = []

**for** srf **in** srfSplit:

edges = srf.Edges

joinedEdges = rc.Geometry.Curve.**JoinCurves**(edges)[0]

joinedEdges = self.**cleanCurve**(joinedEdges)

simplificationOpt = rc.Geometry.CurveSimplifyOptions.All

joinedEdgesSimplified = joinedEdges.**Simplify**(simplificationOpt, sc.doc.ModelAbsoluteTolerance, sc.doc.ModelAngleToleranceRadians)

**try**:

reconstructSrf = rc.Geometry.Brep.**CreatePlanarBreps**(joinedEdgesSimplified)[0]

**except**:

reconstructSrf = srf

# On some systems there was an error with using Brep.Vertices

# I assume this should be an issue with one of Rhinocommon versions

# Hopefully this will fix it -

vertices = reconstructSrf.**DuplicateVertices**()

angle1 = rc.Geometry.Vector3d.**VectorAngle**(rc.Geometry.Vector3d.**Subtract**(rc.Geometry.**Vector3d**(vertices[0]), rc.Geometry.**Vector3d**(vertices[1])), rc.Geometry.Vector3d.**Subtract**(rc.Geometry.**Vector3d**(vertices[0]), rc.Geometry.**Vector3d**(vertices[**len**(vertices) - 1])))

angle2 = rc.Geometry.Vector3d.**VectorAngle**(rc.Geometry.Vector3d.**Subtract**(rc.Geometry.**Vector3d**(vertices[1]), rc.Geometry.**Vector3d**(vertices[2])), rc.Geometry.Vector3d.**Subtract**(rc.Geometry.**Vector3d**(vertices[1]), rc.Geometry.**Vector3d**(vertices[0])))

numSides = reconstructSrf.Edges.Count

rectBool = reconstructSrf.IsValid

**if** rectBool **and** numSides == 4 **and** angle1 < 1.570796 + sc.doc.ModelAngleToleranceRadians **and** angle1 > 1.570796 - sc.doc.ModelAngleToleranceRadians **and** angle2 < 1.570796 + sc.doc.ModelAngleToleranceRadians **and** angle2 > 1.570796 - sc.doc.ModelAngleToleranceRadians:

middle.**append**(reconstructSrf)

**else**:

sides.**append**(reconstructSrf)

#Generate glazing for the non-rectangular surfaces.

sideGlaz = []

**for** srf **in** sides:

**if** srf.Edges.Count == 3:

sideGlaz.**append**(self.**createGlazingTri**(srf, glazingRatio, **None**))

**elif** srf.Edges.Count == 4:

sideGlaz.**append**(self.**createGlazingQuad**(srf, glazingRatio, **None**))

**else**:

sideGlaz.**append**(self.**createGlazingOddPlanarGeo**(srf, glazingRatio))

#Find the glazing for the rectangle part of the wall

rectWinBreps = []

**if** breakUpWindow == **True**:

**for** rect **in** middle:

rectWinBreps.**append**(self.**createGlazingForRect**(rect, glazingRatio, windowHeight, sillHeight, breakUpDist, splitVertDist, conversionFactor))

**else**:

**for** rect **in** middle:

rectWinBreps.**append**(self.**createGlazingQuad**(rect, glazingRatio, **None**))

#Add all of the glazings together into one list.

glzSrf = []

**for** item **in** rectWinBreps:

**for** window **in** item:

glzSrf.**append**(window)

**for** item **in** sideGlaz:

**for** window **in** item:

glzSrf.**append**(window)

#If the surface failed to split and there was no rectangle, chances are that the surface is really oblique so I should get the glazing using the quad function or odd geo function.

**if len**(srfSplit) == 0 **and** rectBrep == **None**:

**try**:

glzSrf = self.**createGlazingQuad**(baseSrf, glazingRatio, **None**)

**except**:

glzSrf = self.**createGlazingOddPlanarGeo**(baseSrf, glazingRatio)

return glzSrf

**def bisect**(self, a, b, fn, epsilon, target):

# This function is taken from the util.js script of the CBE comfort tool page.

**while** (**abs**(b - a) > 2 \* epsilon):

midpoint **=** (b + a) / 2

a\_T = **fn**(a)

b\_T = **fn**(b)

midpoint\_T = **fn**(midpoint)

**if** (a\_T - target) **\*** (midpoint\_T - target) < 0: b = midpoint

**elif** (b\_T - target) **\*** (midpoint\_T - target) < 0: a = midpoint

**else**: return -999

return midpoint

**def secant**(self, a, b, fn, epsilon):

# This function is taken from the util.js script of the CBE comfort tool page.

# root-finding only

f1 = **fn**(a)

**if abs**(f1) <= epsilon: return a

f2 = **fn**(b)

**if abs**(f2) <= epsilon: return b

**for** i **in range**(100):

slope **=** (f2 - f1) **/** (b - a)

c = b - f2 / slope

f3 = **fn**(c)

**if abs**(f3) < epsilon: return c

a = b

b = c

f1 = f2

f2 = f3

return 'NaN'

**def createGlazingCurved**(self, baseSrf, glzRatio, planar):

**def getOffsetDist**(cenPt, edges):

distList = []

[distList.**append**(cenPt.**DistanceTo**(edge.**PointAtNormalizedLength**(0.5))) **for** edge **in** edges]

return **min**(distList)/2

**def getAreaAndCenPt**(surface):

MP = rc.Geometry.AreaMassProperties.**Compute**(surface)

**if** MP:

area = MP.Area \*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

centerPt = MP.Centroid

MP.**Dispose**()

bool, centerPtU, centerPtV = surface.Faces[0].**ClosestPoint**(centerPt)

normalVector = surface.Faces[0].**NormalAt**(centerPtU, centerPtV)

return area, centerPt, normalVector

**else**: return **None**, **None**, **None**

**def OffsetCurveOnSurface**(border, glazingBrep, offsetDist, normalvector, planar):

success = **False**

glzArea = 0

direction = 1

splittedSrfs = []

# Offset the curves on the surface with RhinoCommon

surface = glazingBrep.Faces[0]

glzCurve = border.**OffsetOnSurface**(surface, offsetDist, sc.doc.ModelAbsoluteTolerance)

**if** glzCurve==**None**:

glzCurve = border.**OffsetOnSurface**(surface, -offsetDist, sc.doc.ModelAbsoluteTolerance)

direction = -1

**if** glzCurve!=**None**:

splitBrep = surface.**Split**(glzCurve, sc.doc.ModelAbsoluteTolerance)

**for** srfCount **in range**(splitBrep.Faces.Count):

splSurface = splitBrep.Faces.**ExtractFace**(srfCount)

splittedSrfs.**append**(splSurface)

edges = splSurface.**DuplicateEdgeCurves**(**True**)

joinedEdges = rc.Geometry.Curve.**JoinCurves**(edges)

**if len**(joinedEdges) == 1:

glzSrf = splSurface

glzArea = glzSrf.**GetArea**()\*sc.sticky["honeybee\_ConversionFactor"]\*sc.sticky["honeybee\_ConversionFactor"]

success = **True**

**else**:

**print** "Offseting boundary and spliting the surface failed!"

splittedSrfs = **None**

success = **False**

return success, glzArea, glzCurve, splittedSrfs

face = baseSrf

edges = face.**DuplicateEdgeCurves**(**True**)

border = rc.Geometry.Curve.**JoinCurves**(edges)[0]

area, cenPt, normalvector = **getAreaAndCenPt**(face)

targetArea = area \* glzRatio

offsetDist = **getOffsetDist**(cenPt, edges)

i = 0

glzArea = 2 \* targetArea

inwardDirection = 1 #define the inward direction for the surface

success = **False**

lastSuccessfulGlzSrf = **None**

lastSuccessfulRestOfSrf = **None**

lastSuccessfulSrf = **None**

lastSuccessfulArea = area

srfs = []

**try**: coordinatesList = baseSrf.Vertices

**except**: coordinatesList = baseSrf.**DuplicateVertices**()

succ, glzArea, glzCurve, splittedSrfs = **OffsetCurveOnSurface**(border, face, offsetDist, normalvector, planar)

**if** baseSrf!= **None**:

srfCent = rc.Geometry.AreaMassProperties.**Compute**(baseSrf).Centroid

srfClstParam = border.**ClosestPoint**(srfCent)[1]

srfClstPt = border.**PointAt**(srfClstParam)

glzO\_l = 0.01

glzO\_r = srfCent.**DistanceTo**(srfClstPt) - 0.01

eps = 0.01 # precision of glazing ratio.

**def fn**(offDist):

return (targetArea - **OffsetCurveOnSurface**(border, face, offDist, normalvector, planar)[1])

**try**:

offsetDist = self.**secant**(glzO\_l, glzO\_r, fn, eps)

**except** System.DivideByZeroException:

offsetDist = self.**bisect**(glzO\_l, glzO\_r, fn, eps, 0)

**else**:

**if** offsetDist == 'NaN':

offsetDist = self.**bisect**(glzO\_l, glzO\_r, fn, eps, 0)

succ, glzArea, glzCurve, splittedSrfs = **OffsetCurveOnSurface**(border, face, offsetDist, normalvector, planar)

**if** succ:

srfs.**append**(splittedSrfs)

**try**:

lastSuccessfulGlzSrf = splittedSrfs[1]

lastSuccessfulRestOfSrf = splittedSrfs[0]

lastSuccessfulArea = glzArea

**except** Exception, e:

lastSuccessfulGlzSrf = **None**

lastSuccessfulRestOfSrf = **None**

lastSuccessfulArea = 0

return lastSuccessfulGlzSrf, lastSuccessfulRestOfSrf

**def createSkylightGlazing**(self, baseSrf, glazingRatio, planarBool, edgeLinear, breakUpWindow, breakUpDist, conversionFactor):

**if** breakUpWindow == **True or** breakUpWindow == **None**:

#Define the meshing paramters to break down the surface in a manner that produces only trinagles and quads.

meshPar = rc.Geometry.MeshingParameters.Default

#Define the grid size break down based on the model units.

**if** breakUpDist != **None**: distBreakup = breakUpDist

**else**: distBreakup = 3

distBreakup = distBreakup/conversionFactor

meshPar.MinimumEdgeLength **=** (distBreakup)

meshPar.MaximumEdgeLength **=** (distBreakup\*2)

#Create a mesh of the base surface.

windowMesh = rc.Geometry.Mesh.**CreateFromBrep**(baseSrf, meshPar)[0]

#Define all of the vairables that will be used in the following steps

glzSrf = []

srfFaceList = windowMesh.Faces

srfVertList = windowMesh.Vertices

curvedOk = **True**

lastSuccessfulRestOfSrf = []

#If the surface is curved, check to see if all of the faces are quads, in which case, the generated glazing should look pretty nice. Otherwise, abandon this method and use the offset algorithm.

**if** planarBool == **False**:

**for** face **in** srfFaceList:

**if** face.IsQuad == **True**: pass

**else**: curvedOk = **False**

**if** curvedOk == **False**:

glzSrf, lastSuccessfulRestOfSrf = self.**createGlazingCurved**(baseSrf, glazingRatio, planarBool)

**else**: pass

**else**:pass

#Create breps of all of the mesh faces and use them to make each window.

**if** curvedOk == **True**:

**for** faceNum, face **in enumerate**(srfFaceList):

**if** face.IsQuad == **True**:

glzSrf.**append**(self.**createGlazingQuad**(rc.Geometry.Brep.**CreateFromCornerPoints**(srfVertList[face[0]], srfVertList[face[1]], srfVertList[face[2]], srfVertList[face[3]], sc.doc.ModelAbsoluteTolerance), glazingRatio, windowMesh.Faces.**GetFaceCenter**(faceNum))[0])

**else**:

glzSrf.**append**(self.**createGlazingTri**(rc.Geometry.Brep.**CreateFromCornerPoints**(srfVertList[face[0]], srfVertList[face[1]], srfVertList[face[2]], sc.doc.ModelAbsoluteTolerance), glazingRatio, windowMesh.Faces.**GetFaceCenter**(faceNum))[0])

#If the surface is curved and has not been generated using the offset method, project the quad breps onto the curved surface and split it.

**if** planarBool == **False and** curvedOk == **True**:

faceNormals = []

curvedGlz = []

surface = baseSrf.Faces[0]

**for** faceNum, face **in enumerate**(srfFaceList):

facePlane = rc.Geometry.**Plane**(srfVertList[face[0]], srfVertList[face[1]], srfVertList[face[2]])

faceNormals.**append**(facePlane.Normal)

**for** srfNum, srf **in enumerate**(glzSrf):

edges = srf.Edges

edge = rc.Geometry.Curve.**JoinCurves**(edges)

projectEdge = rc.Geometry.Curve.**ProjectToBrep**(edge, [baseSrf], faceNormals[srfNum], sc.doc.ModelAbsoluteTolerance)[0]

projectBrep = surface.**Split**([projectEdge], sc.doc.ModelAbsoluteTolerance)

splSurface = projectBrep.Faces.**ExtractFace**(1)

curvedGlz.**append**(splSurface)

glzSrf = curvedGlz

**else**:

#Check to see if it is a triangle for which we can use a simple mathematical relation.

**if** planarBool == **True and** baseSrf.Edges.Count == 3:

glzSrf = self.**createGlazingTri**(baseSrf, glazingRatio, **None**)

lastSuccessfulRestOfSrf = []

#Since the surface does not seem to have a rectangle and is not a triangle, check to see if it is a just an odd-shaped quarilateral for which we can use a mathematical relation.

**elif** planarBool == **True and** edgeLinear == **True and** baseSrf.Edges.Count == 4:

glzSrf = self.**createGlazingQuad**(baseSrf, glazingRatio, **None**)

lastSuccessfulRestOfSrf = []

#Since the surface does not have a rectangle, is not a triangle, and is not a quadrilateral but still may be planar, we will break it into triangles and quads by meshing it such that we can use the previous formulas.

**elif** planarBool == **True and** edgeLinear == **True and** breakUpWindow == **True**:

glzSrf = self.**createGlazingOddPlanarGeo**(baseSrf, glazingRatio)

lastSuccessfulRestOfSrf = []

#If everything has failed up until this point, this means that the wall geometry is likely curved. The best way forward is just to try to offset the curve on the surface to get the window.

**else**:

glzSrf, lastSuccessfulRestOfSrf = self.**createGlazingCurved**(baseSrf, glazingRatio, planarBool)

return glzSrf, lastSuccessfulRestOfSrf

**class HB\_generatorsystem**(object):

**def \_\_init\_\_**(self,generatorsystem\_name,simulationinverter,battery,windgenerators,PVgenerators,fuelgenerators,contextsurfaces,HBzonesurfaces,maintenance\_cost):

self.name = generatorsystem\_name

**if** simulationinverter == []:

self.simulationinverter = **None**

**else**:

self.simulationinverter = simulationinverter

self.maintenance\_cost = maintenance\_cost

self.contextsurfaces = contextsurfaces

self.HBzonesurfaces = HBzonesurfaces

self.battery = battery

self.windgenerators = windgenerators # Category includes Generator:WindTurbine

self.PVgenerators = PVgenerators # Category includes Generator:Photovoltaic

self.fuelgenerators = fuelgenerators # Category includes Generators:Mircoturbine,Generator:Combustion Turbine,Generator:InternalCombustionEngine

**class Wind\_gen**(object):

**def \_\_init\_\_**(self,name\_,rotortype,powercontrol,rotor\_speed,rotor\_diameter,overall\_height,number\_of\_blades,power\_output,rated\_wind\_speed,cut\_in\_windspeed,cut\_out\_windspeed,overall\_turbine\_n,max\_tip\_speed\_ratio,max\_power\_coefficient,local\_av\_windspeed,height\_local\_metrological\_station,turbine\_cost,powercoefficients):

self.name = name\_

self.type = 'Generator:WindTurbine'

self.rotortype = rotortype

self.powercontrol = powercontrol

self.numblades = number\_of\_blades

self.rotorspeed = rotor\_speed

self.rotor\_diameter = rotor\_diameter

self.overall\_height = overall\_height

self.powerout = power\_output

self.rated\_wind\_speed = rated\_wind\_speed

self.cut\_in\_windspeed = cut\_in\_windspeed

self.cut\_out\_windspeed = cut\_out\_windspeed

self.overall\_turbine\_n = overall\_turbine\_n

self.max\_tip\_speed\_ratio = max\_tip\_speed\_ratio

self.local\_av\_windspeed = local\_av\_windspeed

self.height\_local\_metrological\_station = height\_local\_metrological\_station

self.cost\_ = turbine\_cost

**if** (powercoefficients != **None**) **or** (powercoefficients != []) :

# Wind turbine is analaytical wind turbine

self.powercoefficients = powercoefficients

**else**:

self.powercoefficients = **None**

**if** max\_power\_coefficient == **None**:

# Only simple wind turbine

self.max\_power\_coefficient = ''

**else**:

self.max\_power\_coefficient = max\_power\_coefficient

**class PV\_gen**(object):

# XXX possible generator types

"""

Generator:InternalCombustionEngine

Generator:CombustionTurbine

Generator:Photovoltaic

Generator:FuelCell

Generator:MicroCHP

Generator:MicroTurbine

Generator:WindTurbine

"""

**def \_\_init\_\_**(self,\_name,mountedsurface\_,No\_parallel,No\_series,powerout,SA\_solarcells,cell\_n,cost):

self.name = \_name

self.mountedSurface = mountedsurface\_

self.type = 'Generator:Photovoltaic'

self.NOparallel = No\_parallel

self.NOseries = No\_series

self.surfaceareacells = SA\_solarcells

self.efficiency = cell\_n

self.cost\_ = cost **or** 0

# Cost and power out of the Generator is the cost and power of each module by the number of modules in each generator

# number in series by number in parallel.

self.powerout = powerout\*No\_series\*No\_parallel

self.inverter = **None** # Define the inverter for this PV generator all PVgenerations being used in the same - run energy simulation must have the same inverter

**class PVinverter**(object):

**def \_\_init\_\_**(self,inverter\_name,inverter\_cost,inverter\_zone,inverter\_n,replacement\_time):

**if** inverter\_zone == **None**:

inverter\_zone = ""

**if** inverter\_n == **None**:

inverter\_n = 0.9

self.name = inverter\_name

self.cost\_ = inverter\_cost

self.efficiency = inverter\_n

self.zone = inverter\_zone

self.replacementtime = replacement\_time

self.ID = **str**(uuid.**uuid4**())

# Need to be able to compare inverters to make sure that only one inverter is servicing all the PV in the system

# For some reason the class ID of the inverters was changing when putting in the hive this is a more fool proof way of comparing them.

# Note the zone that the inverter is attached to is not considered.

**def \_\_hash\_\_**(self):

return **hash**(self.ID)

**def \_\_eq\_\_**( self, other ):

return self.ID == self.ID

**def \_\_ne\_\_**(self,other):

return self.ID != self.ID

**class simple\_battery**(object):

**def \_\_init\_\_**(self,\_name,zone\_name,n\_charging,n\_discharging,battery\_capacity,max\_discharging,max\_charging,initial\_charge,bat\_cost,replacement\_time):

**if** zone\_name == **None**:

zone\_name = ""

self.name = \_name

self.type = 'Battery:simple'

self.zonename = zone\_name

self.chargingefficiency = n\_charging

self.dischargingeffciency = n\_discharging

self.batterycap = battery\_capacity

self.maxcharge = max\_charging

self.maxdischarge = max\_discharging

self.initalcharge = initial\_charge

self.cost\_ = bat\_cost

self.replacementtime = replacement\_time

self.ID = **str**(uuid.**uuid4**())

**class generationhb\_hive**(object):

# A hive that only accepts Honeybee generation objects

**def addToHoneybeeHive**(self, genObjects, GHComponentID):

**if not** sc.sticky.**has\_key**('HBHivegeneration'): sc.sticky['HBHivegeneration'] = {}

generationobjectkeys = []

**if isinstance**(genObjects, tuple):

key = GHComponentID

sc.sticky['HBHivegeneration'][key] = genObjects

generationobjectkeys.**append**(key)

return generationobjectkeys

**else**:

**for** genObject **in** genObjects:

key = GHComponentID

sc.sticky['HBHivegeneration'][key] = genObject

generationobjectkeys.**append**(key)

return generationobjectkeys

**def callFromHoneybeeHive**(self, HBObjectslist):

generationobjects = []

**for** HBObjectkey **in** HBObjectslist:

genobject = sc.sticky['HBHivegeneration'][HBObjectkey]

generationobjects.**append**(genobject)

return generationobjects

**class thermDefaults**(object):

**def \_\_init\_\_**(self):

#Set a default adiabatic boundary condition.

self.adiabaticBCProperties = {}

self.adiabaticBCProperties['Name'] = 'Adiabatic'

self.adiabaticBCProperties['Type'] = "0"

self.adiabaticBCProperties['H'] = '0.000000'

self.adiabaticBCProperties['HeatFlux'] = "0.000000"

self.adiabaticBCProperties['Temperature'] = '0.000000'

self.adiabaticBCProperties['RGBColor'] = '0x000000'

self.adiabaticBCProperties['Tr'] = '0.000000'

self.adiabaticBCProperties['Hr'] = '0.000000'

self.adiabaticBCProperties['Ei'] = "1.000000"

self.adiabaticBCProperties['Viewfactor'] = "1.000000"

self.adiabaticBCProperties['RadiationModel'] = "0"

self.adiabaticBCProperties['ConvectionFlag'] = "0"

self.adiabaticBCProperties['FluxFlag'] = "1"

self.adiabaticBCProperties['RadiationFlag'] = "0"

self.adiabaticBCProperties['ConstantTemperatureFlag'] = "0"

self.adiabaticBCProperties['EmisModifier'] = "1.000000"

#Set some default Boundary Conditions for air cavity surfaces

self.frameCavityBCProperties = {}

self.frameCavityBCProperties['Name'] = 'Frame Cavity Surface'

self.frameCavityBCProperties['Type'] = "7"

self.frameCavityBCProperties['H'] = '0.000000'

self.frameCavityBCProperties['HeatFlux'] = "0.000000"

self.frameCavityBCProperties['Temperature'] = '0.000000'

self.frameCavityBCProperties['RGBColor'] = '0xFF0000'

self.frameCavityBCProperties['Tr'] = '0.000000'

self.frameCavityBCProperties['Hr'] = '0.000000'

self.frameCavityBCProperties['Ei'] = "1.000000"

self.frameCavityBCProperties['Viewfactor'] = "1.000000"

self.frameCavityBCProperties['RadiationModel'] = "0"

self.frameCavityBCProperties['ConvectionFlag'] = "0"

self.frameCavityBCProperties['FluxFlag'] = "1"

self.frameCavityBCProperties['RadiationFlag'] = "0"

self.frameCavityBCProperties['ConstantTemperatureFlag'] = "0"

self.frameCavityBCProperties['EmisModifier'] = "1.000000"

**def addThermMatToLib**(self, materialString):

# Get the name

materialName = materialString.**split**('Material Name=')[-1].**split**(' Type=')[0].**upper**()

#Make a sub-dictionary for the material.

sc.sticky["honeybee\_thermMaterialLib"][materialName] = {}

#Parse the string.

type = **int**(materialString.**split**('Type=')[-1].**split**(' ')[0])

conductivity = **float**(materialString.**split**('Conductivity=')[-1].**split**(' ')[0])

**try**:

absorptivity = **float**(materialString.**split**('Absorptivity=')[-1].**split**(' ')[0])

**except**:

absorptivity = 0.5

**try**:

emissivity = **float**(materialString.**split**('Emissivity=')[-1].**split**(' ')[0])

**except**:

emissivity = **float**(materialString.**split**('EmissivityFront=')[-1].**split**(' ')[0])

**try**:

RGBColor = System.Drawing.ColorTranslator.**FromHtml**(materialString.**split**('RGBColor=')[-1].**split**('/>')[0])

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Tir"] = "0.0"

**except**:

**try**:

RGBColor = System.Drawing.ColorTranslator.**FromHtml**(materialString.**split**('RGBColor=')[-1].**split**(' ')[0])

CavityModel = **int**(materialString.**split**('CavityModel=')[-1].**split**('/>')[0])

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Tir"] = "-1.0"

**except**:

RGBColor = System.Drawing.ColorTranslator.**FromHtml**(materialString.**split**('RGBColor=')[-1].**split**('>')[0])

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Tir"] = "0.0"

#Create the material with values from the original material.

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Name"] = materialName

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Type"] = type

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Conductivity"] = conductivity

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Absorptivity"] = absorptivity

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Emissivity"] = emissivity

sc.sticky["honeybee\_thermMaterialLib"][materialName]["RGBColor"] = RGBColor

sc.sticky["honeybee\_thermMaterialLib"][materialName]["WindowDB"] = ""

sc.sticky["honeybee\_thermMaterialLib"][materialName]["WindowID"] = "-1"

**try**:

sc.sticky["honeybee\_thermMaterialLib"][materialName]["CavityModel"] = CavityModel

**except**: pass

return materialName

**class thermPolygon**(object):

**def \_\_init\_\_**(self, surfaceGeo, material, srfName, plane, RGBColor, ghComp=**None**):

#Set the name and material.

self.objectType = "ThermPolygon"

self.hasChild = **False**

self.name = srfName

self.splitNeeded = **False**

self.warning = **None**

#Check if the material exists in the THERM Library and, if not, add it.

**if** material.**upper**() **in** sc.sticky["honeybee\_materialLib"].**keys**() **or** material.**upper**() **in** sc.sticky["honeybee\_windowMaterialLib"].**keys**(): material = self.**makeThermMatFromEPMat**(material, RGBColor)

**elif** material.**upper**() **in** sc.sticky["honeybee\_thermMaterialLib"].**keys**():

**if** RGBColor == **None**: RGBColor = sc.sticky["honeybee\_thermMaterialLib"][material.**upper**()]["RGBColor"]

**elif** sc.sticky["honeybee\_thermMaterialLib"][material.**upper**()]["RGBColor"] == RGBColor: pass

**else**:

material = self.**makeThermMatCopy**(material, material+**str**(RGBColor), RGBColor)

**else**:

self.warning = 'Failed to find material ' + material + ' in either the therm maerial, EP Material, or EP Window Material libraries.'

material = **None**

self.material = material

#Extract the segments of the polyline and make sure none of them are curved.

segm = surfaceGeo.**DuplicateEdgeCurves**()

self.segments = []

**for** seg **in** segm:

**if** seg.**IsLinear**():

self.segments.**append**(seg)

**elif** seg.Degree == 1:

self.segments.**append**(seg)

**else**:

**print** seg.**CurvatureAt**(0.5)

seg = seg.**ToPolyline**(3,0,0,0,0,0,0,0,**True**)

self.segments.**append**(seg)

msg = "A segment of your polygon is curved and THERM cannot simulate curved geometry.\n" + \

"It has been automatically converted into a polyline with three line segments."

**try**:

ghComp.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

**except**:

pass

#Build a new Polygon from the segments.

self.polylineGeo = rc.Geometry.Curve.**JoinCurves**(self.segments, sc.doc.ModelAbsoluteTolerance)

**if len**(self.polylineGeo) > 1:

self.splitNeeded = **True**

**elif len**(self.polylineGeo) == 1:

self.polylineGeo = self.polylineGeo[0]

#Build surface geometry and extract the vertices.

self.geometry = rc.Geometry.Brep.**CreatePlanarBreps**(self.polylineGeo)[0]

self.vertices = []

**for** vertex **in** self.geometry.**DuplicateVertices**():

self.vertices.**append**(vertex)

#Make note of the plane in which the surface lies and the normal vector.

self.plane = plane

self.normalVector = plane.Normal

self.normalVector.**Unitize**()

self.**resetID**()

return self.geometry

**def resetID**(self):

self.ID = **str**(uuid.**uuid4**())

**def makeThermMatCopy**(self, orgigMat, materialName, RGBColor):

#Make a sub-dictionary for the material.

sc.sticky["honeybee\_thermMaterialLib"][materialName] = {}

#Create the material with values from the original material.

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Name"] = materialName

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Type"] = sc.sticky["honeybee\_thermMaterialLib"][orgigMat]["Type"]

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Conductivity"] = sc.sticky["honeybee\_thermMaterialLib"][orgigMat]["Conductivity"]

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Absorptivity"] = sc.sticky["honeybee\_thermMaterialLib"][orgigMat]["Absorptivity"]

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Tir"] = sc.sticky["honeybee\_thermMaterialLib"][orgigMat]["Tir"]

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Emissivity"] = sc.sticky["honeybee\_thermMaterialLib"][orgigMat]["Emissivity"]

sc.sticky["honeybee\_thermMaterialLib"][materialName]["WindowDB"] = sc.sticky["honeybee\_thermMaterialLib"][materialName]["WindowDB"]

sc.sticky["honeybee\_thermMaterialLib"][materialName]["WindowID"] = sc.sticky["honeybee\_thermMaterialLib"][materialName]["WindowID"]

sc.sticky["honeybee\_thermMaterialLib"][materialName]["RGBColor"] = RGBColor

return materialName

**def makeThermMatFromEPMat**(self, material, RGBColor):

warning = **None**

#Make a sub-dictionary for the material.

sc.sticky["honeybee\_thermMaterialLib"][material] = {}

#Create the material with just default values.

sc.sticky["honeybee\_thermMaterialLib"][material]["Name"] = material

sc.sticky["honeybee\_thermMaterialLib"][material]["Type"] = 0

sc.sticky["honeybee\_thermMaterialLib"][material]["Conductivity"] = **None**

sc.sticky["honeybee\_thermMaterialLib"][material]["Absorptivity"] = 0.5

sc.sticky["honeybee\_thermMaterialLib"][material]["Tir"] = "0.0"

sc.sticky["honeybee\_thermMaterialLib"][material]["Emissivity"] = 0.9

sc.sticky["honeybee\_thermMaterialLib"][material]["WindowDB"] = ""

sc.sticky["honeybee\_thermMaterialLib"][material]["WindowID"] = "-1"

**if** RGBColor != **None**:

**if not** RGBColor.**startswith**('#'):

color = System.Drawing.Color.**FromName**(RGBColor)

RGBColor = System.String.**Format**("#{0:X2}{1:X2}{2:X2}", color.R, color.G, color.B)

sc.sticky["honeybee\_thermMaterialLib"][material]["RGBColor"] = RGBColor.**replace**('#','0x')

**else**:

r = **lambda**: random.**randint**(0,255)

randColor **=** ('#%02X%02X%02X' **%** (**r**(),**r**(),**r**()))

sc.sticky["honeybee\_thermMaterialLib"][material]["RGBColor"] = System.Drawing.ColorTranslator.**FromHtml**(randColor)

#Unpack values from the decomposed material to replace the defaults.

hb\_EPMaterialAUX = **EPMaterialAux**()

values, comments, UValueSI, UValueIP = hb\_EPMaterialAUX.**decomposeMaterial**(material)

**for** count, comment **in enumerate**(comments):

**if** "CONDUCTIVITY" **in** comment.**upper**(): sc.sticky["honeybee\_thermMaterialLib"][material]["Conductivity"] = **float**(values[count])

**if** "EMISSIVITY" **in** comment.**upper**(): sc.sticky["honeybee\_thermMaterialLib"][material]["Emissivity"] = **float**(values[count])

#If there is no conductivity found, it might be an air material, in which case we set the value.

**if** values[0] == "WindowMaterial:Gas":

sc.sticky["honeybee\_thermMaterialLib"][material]["Type"] = 1

sc.sticky["honeybee\_thermMaterialLib"][material]["Conductivity"] = 0.435449

sc.sticky["honeybee\_thermMaterialLib"][material]["CavityModel"] = 4

sc.sticky["honeybee\_thermMaterialLib"][materialName]["Tir"] = "-1.0"

**elif** sc.sticky["honeybee\_thermMaterialLib"][material]["Conductivity"] == **None**:

#This is a no-mass material and we are not going to be able to figure out a conductivity. The best we can do is give a warning.

**if** values[0] == "WindowMaterial:SimpleGlazingSystem": sc.sticky["honeybee\_thermMaterialLib"][material]["Conductivity"] = **float**(values[2])\*0.01

**elif** values[0] == "WindowMaterial:NoMass": sc.sticky["honeybee\_thermMaterialLib"][material]["Conductivity"] = **float**(values[3])/0.01

self.warning = "You have connected a No-Mass material and, as a result, Honeybee can not figure out what conductivity the material has. \n " +\

"Honeybee is going to assume that the No-mass material is very thin with a thickness of 1 cm but we might be completely off. \n " +\

"Try connecting a material with mass or make you own EnergyPlus material and connect it to this component."

**print** self.warning

return material

**def \_\_str\_\_**(self):

return 'THERM Polygon Object:' + **str**(self.name) + \

'\nMaterial: ' + **str**(self.material) + \

'\n# of vertices: ' + `**len**(self.vertices)` + \

'\n-------------------------------------'

**class thermBC**(object):

**def \_\_init\_\_**(self, lineGeo, BCName, temperature, filmCoeff, plane, radTemp, radTransCoeff, RGBColor, uFactorTag, emissOverride, viewFactor=**None**, envEmiss=**None**, heatFlux=**None**, ghComp=**None**):

#Set the name and object type.

self.objectType = "ThermBC"

self.hasChild = **False**

self.name = BCName

self.**resetID**()

#Create a dictionary with all of the inputs for the BC properties.

self.BCProperties = {}

self.BCProperties['Name'] = BCName

self.BCProperties['Type'] = "1"

self.BCProperties['H'] = **str**(filmCoeff)

self.BCProperties['Temperature'] = **str**(temperature)

**if** RGBColor != **None**:

bColor = **str**(System.Drawing.ColorTranslator.**ToHtml**(RGBColor))

**if not** bColor.**startswith**('#'):

color = System.Drawing.Color.**FromName**(bColor)

bColor = System.String.**Format**("#{0:X2}{1:X2}{2:X2}", color.R, color.G, color.B)

self.BCProperties['RGBColor'] = bColor.**replace**('#','0x')

**else**:

self.BCProperties['RGBColor'] = '0x80FFFF'

**if** radTemp == **None**:

self.BCProperties['Tr'] = **str**(temperature)

**else**:

self.BCProperties['Tr'] = **str**(radTemp)

**if** radTransCoeff == **None**:

self.BCProperties['Hr'] = "-431602080.000000"

**else**:

self.BCProperties['Hr'] = **str**(radTransCoeff)

**if** envEmiss == **None**:

self.BCProperties['Ei'] = "1.000000"

**else**:

self.BCProperties['Ei'] = **str**(envEmiss)

**if** viewFactor == **None**:

self.BCProperties['Viewfactor'] = "1.000000"

self.BCProperties['RadiationModel'] = "3"

**else**:

self.BCProperties['Viewfactor'] = **str**(viewFactor)

self.BCProperties['RadiationModel'] = "1"

self.BCProperties['ConvectionFlag'] = "1"

self.BCProperties['RadiationFlag'] = "1"

self.BCProperties['ConstantTemperatureFlag'] = "0"

self.BCProperties['EmisModifier'] = "1.000000"

**if** heatFlux == **None**:

self.BCProperties['HeatFlux'] = "0.000000"

self.BCProperties['FluxFlag'] = "0"

**else**:

self.BCProperties['HeatFlux'] = **str**(heatFlux)

self.BCProperties['FluxFlag'] = "1"

#Create a dictionary for the geometry.

self.BCGeo = {}

self.BCGeo['ID'] = **str**(sc.sticky["thermBCCount"])

self.BCGeo['BC'] = BCName

self.BCGeo['EnclosureID'] = "0"

self.BCGeo['UFactorTag'] = ""

self.BCGeo['Emissivity'] = "0.900000"

#Increase the Therm ID count.

sc.sticky["thermBCCount"] = sc.sticky["thermBCCount"] + 1

#Extract the segments of the polyline and make sure none of them are curved.

segm = lineGeo.**DuplicateSegments**()

**if** segm.Count == 0:

segm = [lineGeo]

self.segments = []

**for** seg **in** segm:

**if** seg.**IsLinear**():

self.segments.**append**(seg)

**elif str**(seg.**CurvatureAt**(0.5)) == '0,0,0':

self.segments.**append**(seg)

**else**:

seg = seg.**ToPolyline**(3,0,0,0,0,0,0,0,**True**)

self.segments.**append**(seg)

msg = "A segment of your boundary condition is curved and THERM cannot simulate curved geometry.\n" + \

"It has been automatically converted into a polyline with three line segments."

**try**:

ghComp.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

**except**:

pass

#Build a new Polygon from the segments.

self.vertices = [self.segments[0].PointAtStart]

**for** seg **in** self.segments:

self.vertices.**append**(seg.PointAtEnd)

self.geometry = rc.Geometry.**PolylineCurve**(self.vertices)

#Make note of the plane in which the surface lies.

self.plane = plane

self.normalVector = plane.Normal

self.normalVector.**Unitize**()

#Set the U-Factor tag information.

self.uFactorTag = **None**

**if** uFactorTag != **None**:

self.uFactorTag = uFactorTag

#Set any emissivity over-rides.

self.emissivityOverride = emissOverride

return self.geometry

**def resetID**(self):

self.ID = **str**(uuid.**uuid4**())

**def \_\_str\_\_**(self):

**if str**(self.BCProperties['H']) == 'INDOOR' **or str**(self.BCProperties['H']) == 'OUTDOOR':

return 'THERM Boundary Object:' + **str**(self.name) + \

'\nTemperature: ' + **str**(self.BCProperties['Temperature']) + ' C' + \

'\nFilm Coefficient: ' + **str**(self.BCProperties['H']) + \

'\n-------------------------------------'

**else**:

return 'THERM Boundary Object:' + **str**(self.name) + \

'\nTemperature: ' + **str**(self.BCProperties['Temperature']) + ' C' + \

'\nFilm Coefficient: ' + **str**(self.BCProperties['H']) + ' W/m2-K' + \

'\n-------------------------------------'

**class viewFactorInfo**(object):

**def \_\_init\_\_**(self, testPtViewFactor=**None**, zoneSrfNames=**None**, testPtSkyView=**None**, testPtBlockedVec=**None**, testPtZoneWeights=**None**, \

testPtZoneNames=**None**, ptHeightWeights=**None**, zoneInletInfo=**None**, zoneHasWindows=**None**, outdoorIsThere=**None**, outdoorNonSrfViewFac=**None**, \

outdoorPtHeightWeights=**None**, testPtBlockName=**None**, zoneWindowTransmiss=**None**, zoneWindowNames=**None**, finalFloorRefList=**None**, \

constantTransmis=**None**, finalAddShdTransmiss=**None**):

#Set the name and object type.

self.objectType = "ViewFactorInfo"

self.hasChild = **False**

self.parent = **None**

self.isChild = **False**

self.hasChild = **False**

self.type = -1

self.BCObject = 'none'

self.BC = 'none'

self.name = **str**(uuid.**uuid4**())[:8]

self.ID = **str**(uuid.**uuid4**())

#Set all of the properties.

self.testPtViewFactor = testPtViewFactor

self.zoneSrfNames = zoneSrfNames

self.testPtSkyView = testPtSkyView

self.testPtBlockedVec = testPtBlockedVec

self.testPtZoneWeights = testPtZoneWeights

self.testPtZoneNames = testPtZoneNames

self.ptHeightWeights = ptHeightWeights

self.zoneInletInfo = zoneInletInfo

self.zoneHasWindows = zoneHasWindows

self.outdoorIsThere = outdoorIsThere

self.outdoorNonSrfViewFac = outdoorNonSrfViewFac

self.outdoorPtHeightWeights = outdoorPtHeightWeights

self.testPtBlockName = testPtBlockName

self.zoneWindowTransmiss = zoneWindowTransmiss

self.zoneWindowNames = zoneWindowNames

self.finalFloorRefList = finalFloorRefList

self.constantTransmis = constantTransmis

self.finalAddShdTransmiss = finalAddShdTransmiss

# Calculate the number of points.

self.NumPts = 0

**if** testPtViewFactor != **None**:

**for** zList **in** testPtViewFactor:

self.NumPts = self.NumPts + **len**(zList)

**def calcNumPts**(self):

self.NumPts = 0

**if** self.testPtViewFactor != **None**:

**for** zList **in** self.testPtViewFactor:

self.NumPts = self.NumPts + **len**(zList)

**def recallAllProps**(self):

return [self.testPtViewFactor, self.zoneSrfNames, self.testPtSkyView, self.testPtBlockedVec, self.testPtZoneWeights, \

self.testPtZoneNames, self.ptHeightWeights, self.zoneInletInfo, self.zoneHasWindows, self.outdoorIsThere, self.outdoorNonSrfViewFac, \

self.outdoorPtHeightWeights, self.testPtBlockName, self.zoneWindowTransmiss, self.zoneWindowNames, self.finalFloorRefList, \

self.constantTransmis, self.finalAddShdTransmiss]

**def \_\_str\_\_**(self):

return 'View Factor Info' + '\nNumber of Points: ' + **str**(self.NumPts)

**class hb\_Hive**(object):

**class CopyClass**(object):

pass

**def checkifTransformed**(self, brep, HBO):

"""

This method ensures that Honeybee objects are not rotated or moved

by Grasshopper components

This test is not restrict enough for mirroring and rotation

but I don't want to make whole Honeybee process slow because of

this extra test

"""

msg = " %s has been moved, scaled or rotated.\nIf you need to move or rotate "%HBO.name + \

"a Honeybee object you should use Honeybee move, rotate or mirror components." + \

" You can find them under 12|WIP tab."

**if** HBO.objectType == "HBIES": return

bb1 = brep.**GetBoundingBox**(**True**)

bb2 = HBO.geometry.**GetBoundingBox**(**True**)

**if** bb1.Min.**DistanceTo**(bb2.Min) > 5 \* sc.doc.ModelAbsoluteTolerance:

**raise Exception**(msg)

**elif** bb1.Max.**DistanceTo**(bb2.Max) > 5 \* sc.doc.ModelAbsoluteTolerance:

**raise Exception**(msg)

@staticmethod

**def addToHoneybeeHive**(HBObjects, Component, removeCurrent=**True**):

"""Add honeybee objects to memory so they can be passed between the components.

removeCurrent: Set false if the same component generates honeybee objects

multiple times in the same component, except for the first time.

"""

**if not** sc.sticky.**has\_key**('HBHive'):

sc.sticky['HBHive'] = {}

**try**:

# get document ID

docId = Component.**OnPingDocument**().DocumentID

**except** AttributeError **as** e:

**if str**(e) == "'str' object has no attribute 'OnPingDocument'":

**raise Exception**('Honeybee version mismatch! Update the component.')

**else**:

**raise Exception**('Failed to add object to HoneybeeHive:\n\t{}'.**format**(e))

baseKey = '{}\_{}'.**format**(docId, Component.InstanceGuid)

# clean the dictionary if it's the first run

**if** removeCurrent **and** Component.RunCount == 1:

**if** baseKey **in** sc.sticky['HBHive']:

**del**(sc.sticky['HBHive'][baseKey])

sc.sticky['HBHive'][baseKey] = {}

# create an empty dictionary for this component

outGeometry = []

**for** HBObject **in** HBObjects:

HBObject.**resetID**()

key = '{}'.**format**(HBObject.ID)

sc.sticky['HBHive'][baseKey][key] = HBObject

# calculate punched geometry if HBobject has a child surface

**try**:

**if** HBObject.objectType != "HBZone" **and** HBObject.hasChild:

# Honeybee surface with openings

**if** HBObject.punchedGeometry == **None**:

HBObject.**calculatePunchedSurface**()

geometries = [childObject.geometry **for** childObject **in** HBObject.childSrfs]

geometries.**append**(HBObject.punchedGeometry)

# join geometries into a single surface

geometry = rc.Geometry.Brep.**JoinBreps**(geometries, sc.doc.ModelAbsoluteTolerance)[0]

**elif** HBObject.objectType == "HBZone":

srfs = []

zoneHasChildSrf = **False**

**for** HBSrf **in** HBObject.surfaces:

**if** HBSrf.hasChild:

zoneHasChildSrf = **True**

srfs.**append**(HBSrf.punchedGeometry)

**for** childObject **in** HBSrf.childSrfs:

srfs.**append**(childObject.geometry)

**else**:

srfs.**append**(HBSrf.geometry)

**if** zoneHasChildSrf:

geometry = rc.Geometry.Brep.**JoinBreps**(srfs, sc.doc.ModelAbsoluteTolerance)[0]

**else**:

geometry = HBObject.geometry

**else**:

# if there is not child object use the geometry as it is

geometry = HBObject.geometry

# assign the key to surface

geometry.UserDictionary.**Set**('HBID', '{}#{}'.**format**(baseKey, key))

outGeometry.**append**(geometry)

**except** Exception **as** e:

**print** `e`

# return geometry with the ID

return outGeometry

**def addNonGeoObjToHive**(self, HBObject, Component):

docId = Component.**OnPingDocument**().DocumentID

baseKey = '{}\_{}'.**format**(docId, Component.InstanceGuid)

sc.sticky['HBHive'][baseKey] = {}

key = '{}'.**format**(HBObject.ID)

sc.sticky['HBHive'][baseKey][key] = HBObject

HBID = '{}#{}'.**format**(baseKey, key)

return 'Honeybee View Factor Info - ' + HBID

**def callFromHoneybeeHive**(self, geometryList):

HBObjects = []

**for** geometry **in** geometryList:

**try**:

hbkey = geometry.UserDictionary['HBID']

**except**:

hbkey = geometry.**split**(' ')[-1]

**if** '#' **not in** hbkey:

**raise Exception**('Honeybee version mismatch! Update the input component.')

baseKey, key = hbkey.**split**('#')[0], '#'.**join**(hbkey.**split**('#')[1:])

**if** sc.sticky['HBHive'].**has\_key**(baseKey):

HBObject = sc.sticky['HBHive'][baseKey][key]

# make sure Honeybee object is not moved or rotated

**try**:

self.**checkifTransformed**(geometry, HBObject)

**except**:

pass

**try**:

# after the first round meshedFace makes copy.deepcopy crash

# so I need to regenerate meshFaces

bc = []

**if** HBObject.objectType == "HBZone":

**for** surface **in** HBObject.surfaces:

newMesh = rc.Geometry.**Mesh**()

newMesh.**Append**(surface.meshedFace)

surface.meshedFace = newMesh

# keep track of boundary conditions

# and then set them to None not to create

# memory issues for large models.

bc.**append**(copy.**copy**(surface.BCObject))

surface.BCObject = **None**

**for** csrf **in** surface.childSrfs:

bc.**append**(copy.**copy**(csrf.BCObject))

csrf.BCObject = **None**

**elif** HBObject.objectType == "HBSurface":

newMesh = rc.Geometry.**Mesh**()

newMesh.**Append**(HBObject.meshedFace)

HBObject.meshedFace = newMesh

# keep track of boundary conditions

# and then set them to None not to create

# memory issues for large models.

bc.**append**(copy.**copy**(HBObject.BCObject))

HBObject.BCObject = **None**

**for** csrf **in** HBObject.childSrfs:

bc.**append**(copy.**copy**(csrf.BCObject))

csrf.BCObject = **None**

newObject = copy.**deepcopy**(HBObject)

# put the boundary condition objects back

count = 0

**if** HBObject.objectType == "HBZone":

**for** c, surface **in enumerate**(newObject.surfaces):

surface.BCObject = bc[count]

HBObject.surfaces[c].BCObject = bc[count]

count += 1

**for** cc, csrf **in enumerate**(surface.childSrfs):

csrf.BCObject = bc[count]

HBObject.surfaces[c].childSrfs[cc].BCObject = bc[count]

count += 1

**elif** HBObject.objectType == "HBSurface":

newObject.BCObject = bc[count]

HBObject.BCObject = bc[count]

count += 1

**for** cc, csrf **in enumerate**(newObject.childSrfs):

csrf.BCObject = bc[count]

HBObject.childSrfs[cc].BCObject = bc[count]

count += 1

HBObjects.**append**(newObject)

**del**(bc)

**except** Exception, e:

**print** `e`

**print** "Failed to copy the object. Returning the original objects...\n" +\

"This can cause strange behaviour!"

HBObjects.**append**(sc.sticky['HBHive'][baseKey][key])

**else**:

**raise Exception**('HoneybeeKeyMismatch: Failed to call the object from Honeybee hive.')

return HBObjects

**def visualizeFromHoneybeeHive**(self, geometryList):

HBObjects = []

**for** geometry **in** geometryList:

**try**:

hbkey = geometry.UserDictionary['HBID']

**except**:

hbkey = geometry.**split**(' ')[-1]

**if** '#' **not in** hbkey:

**raise Exception**('Honeybee version mismatch! Update the input component.')

baseKey, key = hbkey.**split**('#')[0], '#'.**join**(hbkey.**split**('#')[1:])

**if** sc.sticky['HBHive'].**has\_key**(baseKey):

HBObjects.**append**(sc.sticky['HBHive'][baseKey][key])

**else**:

**raise Exception**('HoneybeeKeyMismatch: Failed to call the object from Honeybee hive.')

return HBObjects

**class hb\_RADParameters**(object):

**def \_\_init\_\_**(self):

self.radParDict = {

"\_ab\_": [2, 3, 6],

"\_ad\_": [512, 2048, 4096],

"\_as\_": [128, 2048, 4096],

"\_ar\_": [16, 64, 128],

"\_aa\_": [.25, .2, .1],

"\_ps\_": [8, 4, 2],

"\_pt\_": [.15, .10, .05],

"\_pj\_": [.6, .9, .9],

"\_dj\_": [0, .5, .7],

"\_ds\_": [.5, .25, .05],

"\_dt\_": [.5, .25, .15],

"\_dc\_": [.25, .5, .75],

"\_dr\_": [0, 1, 3],

"\_dp\_": [64, 256, 512],

"\_sj\_": [.3, .7, 1], # only for daysim which uses older version of rtrace

"\_st\_": [.85, .5, .15],

"\_lr\_": [4, 6, 8],

"\_lw\_": [.05, .01, .005],

"\_av\_": [0, 0, 0],

"xScale": [1, 2, 6],

"yScale": [1, 2, 6]

}

self.additionalRadPars = ["\_u\_", "\_bv\_", "\_dv\_", "\_w\_"]

**class hb\_DSParameters**(object):

**def \_\_init\_\_**(self, outputUnits = [2], dynamicSHDGroup\_1 = **None**, dynamicSHDGroup\_2 = **None**, RhinoViewsName = [] , adaptiveZone = **False**, dgp\_imageSize = 250, onlyRunGlareAnalysis = **True**):

**if len**(outputUnits)!=0 **and** outputUnits[0]!=**None**: self.outputUnits = outputUnits

**else**: self.outputUnits = [2]

self.onlyAnnualGlare = onlyRunGlareAnalysis

self.runAnnualGlare = **False**

self.RhinoViewsName = RhinoViewsName

**if** RhinoViewsName != []:

self.runAnnualGlare = **True**

**if** adaptiveZone == **None**: adaptiveZone = **False**

self.adaptiveZone = adaptiveZone

**if not** dgp\_imageSize: dgp\_imageSize = 250

self.dgp\_imageSize = dgp\_imageSize

**if** dynamicSHDGroup\_1 == **None and** dynamicSHDGroup\_2==**None**:

**class dynamicSHDRecipe**(object):

**def \_\_init\_\_**(self, type = 1, name = "no\_blind"):

self.type = type

self.name = name

self.DShdR = [**dynamicSHDRecipe**(type = 1, name = "no\_blind")]

**else**:

self.DShdR = []

**if** dynamicSHDGroup\_1 != **None**: self.DShdR.**append**(dynamicSHDGroup\_1)

**if** dynamicSHDGroup\_2 != **None**: self.DShdR.**append**(dynamicSHDGroup\_2)

# Number of ill files

self.numOfIll = 1

**for** shadingRecipe **in** self.DShdR:

**if** shadingRecipe.name == "no\_blind":

pass

**elif** shadingRecipe.name == "conceptual\_dynamic\_shading":

self.numOfIll += 1

**else**:

# advanced dynamic shading

self.numOfIll += **len**(shadingRecipe.shadingStates) - 1

# print "number of ill files = " + str(self.numOfIll)

**class CalculateGridBasedDLAnalysisResults**(object):

"""

calculate results of any grid based analysis

analysisType: [0] illuminance, [1] radiation, [2] luminance, [3] daylight factor, [4] vertical sky component

"""

**def \_\_init\_\_**(self, resultFiles, analysisType):

self.analysisType = analysisType

self.resultFiles = resultFiles

**def getResults**(self):

resultValues = []

studyType= self.analysisType

**for** fileCount, resultFile **in enumerate**(self.resultFiles):

**if** studyType == 0 **or** studyType == 2:

#illuminance / luminance

resultValues.**extend**(self.**readDLResult**(resultFile))

**elif** studyType == 1:

# radiation

resultValues.**extend**(self.**readRadiationResult**(resultFile))

**elif** studyType == 3 **or** studyType == 4:

resultValues.**extend**(self.**readDFResult**(resultFile))

return resultValues

**def readRadiationResult**(self, resultFile):

result = []

resultFile = **open**(resultFile,"r")

**for** line **in** resultFile:

result.**append**(**float**(line.**split**(' ')[0]))

return result

**def readDLResult**(self, resultFile):

result = []

resultFile = **open**(resultFile,"r")

**for** line **in** resultFile:

R, G, B = line.**split**(' ')[0:3]

result.**append**(179**\***(.265 \* **float**(R) + .67 \* **float**(G) + .065 \* **float**(B)))

return result

**def readDFResult**(self, resultFile):

result = []

resultFile = **open**(resultFile,"r")

**for** line **in** resultFile:

R, G, B = line.**split**(' ')[0:3]

# divide by the sky horizontal illuminance = 100000

res = 17900**\***(.265 \* **float**(R) + .67 \* **float**(G) + .065 \* **float**(B))/100000

**if** res > 100: res = 100

result.**append**(res)

return result

**class SerializeObjects**(object):

**def \_\_init\_\_**(self, filePath, data = **None**):

self.filePath = filePath

self.data = data

**def saveToFile**(self):

with **open**(self.filePath, 'wb') **as** outf:

pickle.**dump**(self.data, outf)

**def readFromFile**(self):

with **open**(self.filePath, 'rb') **as** inf:

self.data = pickle.**load**(inf)

**class hb\_hvacProperties**(object):

**def \_\_init\_\_**(self):

# A dictionary that contains all of the names of the HVAC systems that correspond to the integer IDs.

self.sysDict = {

-1:'THERMOSTAT/HUMIDISTAT ONLY',

0:'IDEAL AIR LOADS',

1:'PACKAGED TERMINAL AIR CONDITIONING',

2:'PACKAGED TERMINAL HEAT PUMP',

3:'PACKAGED SINGLE ZONE - AC',

4:'PACKAGED SINGLE ZONE - HP',

5:'PACKAGED VAV WITH REHEAT',

6:'PACKAGED VAV WITH PARALLEL FAN POWERED BOXES',

7:'VARIABLE AIR VOLUME WITH REHEAT',

8:'VARIABLE AIR VOLUME WITH PARALLEL FAN POWERED BOXES',

9:'WARM AIR FURNACE - GAS FIRED',

10:'WARM AIR FURNACE - ELECTRIC',

11:'FAN COIL UNITS + DOAS',

12:'ACTIVE CHILLED BEAMS + DOAS',

13:'RADIANT FLOORS + DOAS',

14:'CUSTOM RAD SURFACES + DOAS',

15:'HEATED SURFACES + VAV COOLING',

16:'VRF + DOAS',

17:'WSHP + DOAS'

}

# Dictionaries that state which features can be changed for each of the different systems.

# It is used to give warnings to the user if they set a parameter that does not exist on the assigned HVAC system.

self.thresholdCapabilities = {

0: {'recirc' : **False**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

1: {'recirc' : **True**, 'humidCntrl' : **False**, 'dehumidCntrl' : **False**, 'ventSched' : **False**},

2: {'recirc' : **True**, 'humidCntrl' : **False**, 'dehumidCntrl' : **False**, 'ventSched' : **False**},

3: {'recirc' : **False**, 'humidCntrl' : **True**, 'dehumidCntrl' : **False**, 'ventSched' : **False**},

4: {'recirc' : **False**, 'humidCntrl' : **True**, 'dehumidCntrl' : **False**, 'ventSched' : **False**},

5: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **False**, 'ventSched' : **True**},

6: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **False**, 'ventSched' : **True**},

7: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

8: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

9: {'recirc' : **False**, 'humidCntrl' : **True**, 'dehumidCntrl' : **False**, 'ventSched' : **False**},

10: {'recirc' : **False**, 'humidCntrl' : **True**, 'dehumidCntrl' : **False**, 'ventSched' : **False**},

11: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

12: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **False**},

13: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

14: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

15: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

16: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**},

17: {'recirc' : **True**, 'humidCntrl' : **True**, 'dehumidCntrl' : **True**, 'ventSched' : **True**}

}

self.airCapabilities = {

0: {'FanTotEff': **False**, 'FanMotEff': **False**, 'FanPres': **False**, 'FanPlace': **False**, 'airSysHardSize': **False**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **False**, 'Econ' : **True**, 'HeatRecov' : **True**},

1: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **False**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **False**, 'Recirculation': **False**, 'Econ' : **False**, 'HeatRecov' : **False**},

2: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **False**, 'HeatSupTemp' : **False**, 'CoolSupTemp' : **False**, 'Recirculation': **False**, 'Econ' : **False**, 'HeatRecov' : **False**},

3: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

4: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

5: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **False**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

6: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **False**, 'HeatSupTemp' : **False**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

7: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **False**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

8: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **False**, 'HeatSupTemp' : **False**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

9: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

10: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **True**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

11: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **False**, 'Econ' : **True**, 'HeatRecov' : **True**},

12: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **False**, 'Econ' : **True**, 'HeatRecov' : **True**},

13: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **False**, 'Econ' : **True**, 'HeatRecov' : **True**},

14: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **False**, 'Econ' : **True**, 'HeatRecov' : **True**},

15: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **False**, 'FanCntrl': **False**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **True**, 'Econ' : **True**, 'HeatRecov' : **True**},

16: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **True**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **False**, 'Econ' : **True**, 'HeatRecov' : **True**},

17: {'FanTotEff': **True**, 'FanMotEff': **True**, 'FanPres': **True**, 'FanPlace': **False**, 'airSysHardSize': **True**, 'centralAirLoop' : **True**, 'FanCntrl': **True**, 'HeatSupTemp' : **True**, 'CoolSupTemp' : **True**, 'Recirculation': **False**, 'Econ' : **True**, 'HeatRecov' : **True**}

}

self.heatCapabilities = {

0: {'COP' : **False**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

1: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'HeatHardSize': **True**, 'CentralPlant' : **False**},

2: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

3: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

4: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

5: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'HeatHardSize': **False**, 'CentralPlant' : **True**},

6: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

7: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'HeatHardSize': **True**, 'CentralPlant' : **True**},

8: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

9: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

10: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **False**},

11: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'HeatHardSize': **True**, 'CentralPlant' : **True**},

12: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'HeatHardSize': **True**, 'CentralPlant' : **True**},

13: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'HeatHardSize': **True**, 'CentralPlant' : **True**},

14: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'HeatHardSize': **True**, 'CentralPlant' : **True**},

15: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **False**, 'HeatHardSize': **True**, 'CentralPlant' : **True**},

16: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **True**},

17: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **False**, 'HeatHardSize': **False**, 'CentralPlant' : **True**}

}

self.coolCapabilities = {

0: {'COP' : **False**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

1: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

2: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

3: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

4: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

5: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

6: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

7: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'CoolHardSize': **True**, 'CentralPlant' : **True**, 'ChillType' : **True**},

8: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'CoolHardSize': **True**, 'CentralPlant' : **True**, 'ChillType' : **True**},

9: {'COP' : **False**, 'Avail' : **False**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

10: {'COP' : **False**, 'Avail' : **False**, 'SupTemp' : **False**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **False**, 'ChillType' : **False**},

11: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'CoolHardSize': **True**, 'CentralPlant' : **True**, 'ChillType' : **True**},

12: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'CoolHardSize': **True**, 'CentralPlant' : **True**, 'ChillType' : **True**},

13: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'CoolHardSize': **True**, 'CentralPlant' : **True**, 'ChillType' : **True**},

14: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'CoolHardSize': **True**, 'CentralPlant' : **True**, 'ChillType' : **True**},

15: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **True**, 'CoolHardSize': **True**, 'CentralPlant' : **True**, 'ChillType' : **True**},

16: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **True**, 'ChillType' : **True**},

17: {'COP' : **True**, 'Avail' : **True**, 'SupTemp' : **True**, 'PumpEff' : **False**, 'CoolHardSize': **False**, 'CentralPlant' : **True**, 'ChillType' : **True**}

}

@staticmethod

**def generateWarning**(sysType, varType, detailType):

msg = 'The HVAC system type, ' + sysType + ' does not support the assigning of \n' + varType + \

' but one has been assigned in the ' + detailType +'.'

return msg

@staticmethod

**def checkSchedule**(schedule):

error = **None**

schedule= schedule.**upper**()

**if** schedule!=**None and not** schedule.**lower**().**endswith**(".csv") **and** schedule **not in** sc.sticky["honeybee\_ScheduleLib"].**keys**():

error = "Cannot find " + schedule + " in Honeybee schedule library."

return **False**, error

**elif** schedule!=**None and** schedule.**lower**().**endswith**(".csv"):

# check if csv file is existed

**if not** os.path.**isfile**(schedule):

error = "Cannot find the shchedule file: " + schedule

return **False**, error

return **True**, error

**class hb\_airDetail**(object):

**def \_\_init\_\_**(self, HVACAvailabiltySched=**None**, fanTotalEfficiency=**None**, fanMotorEfficiency=**None**, fanPressureRise=**None**, \

fanPlacement=**None**, airSysHardSize = **None**, centralAirLoop=**None**, fanControl = **None**, heatingSupplyAirTemp=**None**, \

coolingSupplyAirTemp=**None**, recirculation=**None**, airsideEconomizer=**None**, sensibleHeatRecovery=**None**, latentHeatRecovery=**None**):

self.areInputsChecked = **False**

self.sysProps = **hb\_hvacProperties**()

self.ID = **str**(uuid.**uuid4**())

self.objectType = "HBair"

self.economizerCntrlDict = {

0:'NoEconomizer',

1:'DifferentialDryBulb',

2:'DifferentialEnthalpy',

3:'FixedDryBulb',

4:'FixedEnthalpy',

5:'ElectronicEnthalpy',

6:'FixedDewpointandDryBulb',

7:'DifferentialDryBulbandEnthalpy',

'NoEconomizer': 'NoEconomizer',

'DifferentialDryBulb':'DifferentialDryBulb',

'DifferentialEnthalpy':'DifferentialEnthalpy',

'FixedDryBulb':'FixedDryBulb',

'FixedEnthalpy':'FixedEnthalpy',

'ElectronicEnthalpy':'ElectronicEnthalpy',

'FixedDewpointandDryBulb':'FixedDewpointandDryBulb',

'DifferentialDryBulbandEnthalpy':'DifferentialDryBulbandEnthalpy'

}

self.fanPlaceDict = {

**True**: 'Draw Through',

**False**: 'Blow Through',

'Draw Through': 'Draw Through',

'Blow Through': 'Blow Through'

}

self.fanControlDict = {

**True**: 'Variable Volume',

**False**: 'Constant Volume',

'Variable Volume': 'Variable Volume',

'Constant Volume': 'Constant Volume'

}

**if** HVACAvailabiltySched:

self.HVACAvailabiltySched = HVACAvailabiltySched

**else**:

self.HVACAvailabiltySched = "ALWAYS ON"

**if** fanTotalEfficiency:

self.fanTotalEfficiency = **float**(fanTotalEfficiency)

**else**:

self.fanTotalEfficiency = "Default"

**if** fanMotorEfficiency:

self.fanMotorEfficiency = **float**(fanMotorEfficiency)

**else**:

self.fanMotorEfficiency = "Default"

**if** fanPressureRise:

self.fanPressureRise = **float**(fanPressureRise)

**else**:

self.fanPressureRise = "Default"

**if** fanPlacement != **None**:

self.fanPlacement = self.fanPlaceDict[fanPlacement]

**else**:

self.fanPlacement = "Default"

**if** airSysHardSize != **None**:

self.airSysHardSize = airSysHardSize

**else**:

self.airSysHardSize = "Default"

**if** centralAirLoop != **None**:

self.centralAirLoop = centralAirLoop

**else**:

self.centralAirLoop = "Default"

**if** fanControl != **None**:

self.fanControl = self.fanControlDict[fanControl]

**else**:

self.fanControl = "Default"

**if** heatingSupplyAirTemp:

self.heatingSupplyAirTemp = **float**(heatingSupplyAirTemp)

**else**:

self.heatingSupplyAirTemp = "Default"

**if** coolingSupplyAirTemp:

self.coolingSupplyAirTemp = **float**(coolingSupplyAirTemp)

**else**:

self.coolingSupplyAirTemp = "Default"

**if** recirculation != **None**:

self.recirculation = recirculation

**else**:

self.recirculation = "Default"

**if** airsideEconomizer != **None**:

**try**:

self.airsideEconomizer = **int**(airsideEconomizer)

**except**:

self.airsideEconomizer = self.economizerCntrlDict[airsideEconomizer]

**else**:

self.airsideEconomizer = "Default"

**if** sensibleHeatRecovery != **None**:

self.sensibleHeatRecovery = **float**(sensibleHeatRecovery)

**else**:

self.sensibleHeatRecovery = "Default"

**if** latentHeatRecovery != **None**:

self.latentHeatRecovery = **float**(latentHeatRecovery)

**else**:

self.latentHeatRecovery = "Default"

@classmethod

**def fromTextStr**(cls, textStr):

paramList = []

success = **True**

**for** count, line **in enumerate**(textStr.**split**('\n')):

**if** count == 0:

**if** 'AIR DETAILS' **not in** line.**upper**():

success = **False**

**else**:

param = line.**split**(': ')[-1]

**if** param.**upper**() != 'DEFAULT':

paramList.**append**(param)

**else**:

paramList.**append**(**None**)

**if** success == **True**:

airDetailObj = **cls**(paramList[0], paramList[1], paramList[2], paramList[3], paramList[4], paramList[5], paramList[6], paramList[7], paramList[8], paramList[9], paramList[10], paramList[11], paramList[12], paramList[13])

airDetailObj.areInputsChecked = **True**

return airDetailObj

**else**:

return **None**

**def checkInputVariables**(self):

errors = []

success = **True**

**if** self.HVACAvailabiltySched != "ALWAYS ON":

success, error = self.sysProps.**checkSchedule**(self.HVACAvailabiltySched)

**if** success **is False**:

errors.**append**(error)

**if** self.fanTotalEfficiency != "Default":

**if** self.fanTotalEfficiency > 1 **or** self.fanTotalEfficiency < 0:

success = **False**

errors.**append**("Fan Total Efficiency must be betweeon 0 and 1.")

**if** self.fanMotorEfficiency != "Default":

**if** self.fanMotorEfficiency > 1 **or** self.fanMotorEfficiency < 0:

success = **False**

errors.**append**("Fan Motor Efficiency must be betweeon 0 and 1.")

**if** self.airSysHardSize != "Default":

**if** self.airSysHardSize < 0:

success = **False**

errors.**append**("airSystemHardSize\_ cannot be less than 0.")

**if** self.airsideEconomizer != "Default":

**if** self.airsideEconomizer > 7 **or** self.airsideEconomizer < 0:

success = **False**

errors.**append**("Air Side Economizer not a valid control type.")

**else**:

self.airsideEconomizer = self.economizerCntrlDict[self.airsideEconomizer]

**if** self.sensibleHeatRecovery != 'Default':

**if** self.sensibleHeatRecovery > 1 **or** self.sensibleHeatRecovery < 0:

success = **False**

errors.**append**("Sensible Heat Recovery Effeictiveness must be between 0 and 1.")

**if** self.latentHeatRecovery != 'Default':

**if** self.latentHeatRecovery > 1 **or** self.latentHeatRecovery < 0:

success = **False**

errors.**append**("Latent Heat Recovery Effeictiveness must be between 0 and 1.")

return success, errors

**def checkSysCompatability**(self, sysInt):

errors = []

sysType = self.sysProps.sysDict[sysInt]

hvacCapabilities = self.sysProps.airCapabilities[sysInt]

**if** self.fanTotalEfficiency != 'Default' **and** hvacCapabilities['FanTotEff'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'FAN TOTAL EFFICIENCY', 'airDetails'))

**if** self.fanMotorEfficiency != 'Default' **and** hvacCapabilities['FanMotEff'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'FAN MOTOR EFFICIENCY', 'airDetails'))

**if** self.fanPressureRise != 'Default' **and** hvacCapabilities['FanPres'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'FAN PRESSURE RISE', 'airDetails'))

**if** self.fanPlacement != 'Default' **and** hvacCapabilities['FanPlace'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'FAN PLACEMENT', 'airDetails'))

**if** self.airSysHardSize != 'Default' **and** hvacCapabilities['airSysHardSize'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'AIR SYSTEM HARD SIZE', 'airDetails'))

**if** self.centralAirLoop != 'Default' **and** hvacCapabilities['centralAirLoop'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'CENTRAL AIR LOOP', 'airDetails'))

**if** self.fanControl != 'Default' **and** hvacCapabilities['FanCntrl'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'DEMAND CONTROLLED VENTILATION', 'airDetails'))

**if** self.heatingSupplyAirTemp != 'Default' **and** hvacCapabilities['HeatSupTemp'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'HEATING SUPPLY AIR TEMPERATURE', 'airDetails'))

**if** self.coolingSupplyAirTemp != 'Default' **and** hvacCapabilities['CoolSupTemp'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING SUPPLY AIR TEMPERATURE', 'airDetails'))

**if** self.recirculation != 'Default' **and** hvacCapabilities['Recirculation'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'RECIRCULATION', 'airDetails'))

**if** self.airsideEconomizer != 'Default' **and** hvacCapabilities['Econ'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'AN AIRSIDE ECONOMIZER', 'airDetails'))

**if** sysInt == 0:

**if** self.airsideEconomizer == 'Default' **or** self.airsideEconomizer == 'NoEconomizer' **or** self.airsideEconomizer == 'DifferentialDryBulb' **or** self.airsideEconomizer == 'DifferentialEnthalpy':

pass

**else**:

errors.**append**('Airside economizer type ' + self.airsideEconomizer + ' is not supported for IDEAL AIR LOADS SYSTEMS.')

**if** self.sensibleHeatRecovery != 'Default' **and** hvacCapabilities['HeatRecov'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'A HEAT RECOVERY SYSTEM', 'airDetails'))

**if** self.latentHeatRecovery != 'Default' **and** hvacCapabilities['HeatRecov'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'A HEAT RECOVERY SYSTEM', 'airDetails'))

return errors

**def class2Str**(self):

allGood = **True**

**if not** self.areInputsChecked:

allGood, errors = self.**checkInputVariables**()

**if** allGood:

textStr = 'Air Details\n' + \

' Availability Schedule: ' + **str**(self.HVACAvailabiltySched) + '\n' + \

' Fan Total Efficiency: ' + **str**(self.fanTotalEfficiency) + '\n' + \

' Fan Motor Efficiency: ' + **str**(self.fanMotorEfficiency) + '\n' + \

' Fan Pressure Rise: ' + **str**(self.fanPressureRise) + '\n' + \

' Fan Placement: ' + **str**(self.fanPlacement) + '\n' + \

' Air System Hard Size: ' + **str**(self.airSysHardSize) + '\n' + \

' Central Air Loop: ' + **str**(self.centralAirLoop) + '\n' + \

' Demand Controlled Ventilation: ' + **str**(self.fanControl) + '\n' + \

' Heating Supply Air Temperature: ' + **str**(self.heatingSupplyAirTemp) + '\n' + \

' Cooling Supply Air Temperature: ' + **str**(self.coolingSupplyAirTemp) + '\n' + \

' Air Loop Recirculation: ' + **str**(self.recirculation) + '\n' + \

' Airside Economizer Method: ' + **str**(self.airsideEconomizer) + '\n' + \

' Sensible Heat Recovery Effectiveness: ' + **str**(self.sensibleHeatRecovery) + '\n' + \

' Latent Heat Recovery Effectiveness: ' + **str**(self.latentHeatRecovery)

return **True**, textStr

**else**:

return **False**, errors

**class hb\_heatingDetail**(object):

**def \_\_init\_\_**(self, heatingAvailSched=**None**, heatingEffOrCOP=**None**, supplyTemperature=**None**, pumpMotorEfficiency=**None**, heatHardSize = **None**, centralPlant=**None**):

self.areInputsChecked = **False**

self.sysProps = **hb\_hvacProperties**()

self.ID = **str**(uuid.**uuid4**())

self.objectType = "HBheat"

**if** heatingAvailSched:

self.heatingAvailSched = heatingAvailSched

**else**:

self.heatingAvailSched = "ALWAYS ON"

**if** heatingEffOrCOP != **None**:

self.heatingEffOrCOP = **float**(heatingEffOrCOP)

**else**:

self.heatingEffOrCOP = "Default"

**if** supplyTemperature != **None**:

self.supplyTemperature = **float**(supplyTemperature)

**else**:

self.supplyTemperature = "Default"

**if** pumpMotorEfficiency != **None**:

self.pumpMotorEfficiency = **float**(pumpMotorEfficiency)

**else**:

self.pumpMotorEfficiency = "Default"

**if** heatHardSize != **None**:

self.heatHardSize = heatHardSize

**else**:

self.heatHardSize = "Autosize"

**if** centralPlant != **None**:

self.centralPlant = centralPlant

**else**:

self.centralPlant = "Default"

@classmethod

**def fromTextStr**(cls, textStr):

paramList = []

success = **True**

**for** count, line **in enumerate**(textStr.**split**('\n')):

**if** count == 0:

**if** 'HEATING DETAILS' **not in** line.**upper**():

success = **False**

**else**:

param = line.**split**(': ')[-1]

**if** param.**upper**() != 'DEFAULT':

paramList.**append**(param)

**else**:

paramList.**append**(**None**)

**if** success == **True**:

heatDetailObj = **cls**(paramList[0], paramList[1], paramList[2], paramList[3], paramList[4], paramList[5])

heatDetailObj.areInputsChecked = **True**

return heatDetailObj

**else**:

return **None**

**def checkInputVariables**(self):

errors = []

success = **True**

**if** self.heatingAvailSched != "ALWAYS ON":

success, error = self.sysProps.**checkSchedule**(self.heatingAvailSched)

**if** success **is False**:

errors.**append**(error)

**if** self.pumpMotorEfficiency != "Default":

**if** self.pumpMotorEfficiency > 1 **or** self.pumpMotorEfficiency < 0:

success = **False**

errors.**append**("Pump Motor Efficiency must be betweeon 0 and 1.")

return success, errors

**def checkSysCompatability**(self, sysInt):

errors = []

sysType = self.sysProps.sysDict[sysInt]

heatCapabilities = self.sysProps.heatCapabilities[sysInt]

**if** self.heatingAvailSched != 'ALWAYS ON' **and** heatCapabilities['Avail'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'HEATING AVAILABILITY SCHEDULE', 'heatingDetails'))

**if** self.heatingEffOrCOP != 'Default' **and** heatCapabilities['COP'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'HEATING SYSTEM EFFICIENCY OR COP', 'heatingDetails'))

**if** self.supplyTemperature != 'Default' **and** heatCapabilities['SupTemp'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'HEATING SYSTEM SUPPLY TEMPERATURE', 'heatingDetails'))

**if** self.pumpMotorEfficiency != 'Default' **and** heatCapabilities['PumpEff'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'HEATING SYSTEM PUMP MOTOR EFFICIENCY', 'heatingDetails'))

**if** self.heatHardSize != 'Autosize' **and** heatCapabilities['HeatHardSize'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'HEATING SYSTEM HARD SIZE', 'heatingDetails'))

**if** self.centralPlant != 'Default' **and** heatCapabilities['CentralPlant'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'HEATING SYSTEM CENTRALIZED PLANT', 'heatingDetails'))

return errors

**def class2Str**(self):

allGood = **True**

**if not** self.areInputsChecked:

allGood, errors = self.**checkInputVariables**()

**if** allGood:

textStr = 'Heating Details\n' + \

' Heating Availability Schedule: ' + **str**(self.heatingAvailSched) + '\n' + \

' Heating System Efficiency or COP: ' + **str**(self.heatingEffOrCOP) + '\n' + \

' Heating System Supply Temperature: ' + **str**(self.supplyTemperature) + '\n' + \

' Heating System Pump Motor Efficiency: ' + **str**(self.pumpMotorEfficiency) + '\n' + \

' Heating System Hard Size: ' + **str**(self.heatHardSize) + '\n' + \

' Heating System Centralized Plant: ' + **str**(self.centralPlant)

return **True**, textStr

**else**:

return **False**, errors

**class hb\_coolingDetail**(object):

**def \_\_init\_\_**(self, coolingAvailSched=**None**, coolingCOP=**None**, supplyTemperature=**None**, pumpMotorEfficiency=**None**, coolHardSize = **None**, centralPlant=**None**, chillerType=**None**):

self.areInputsChecked = **False**

self.sysProps = **hb\_hvacProperties**()

self.ID = **str**(uuid.**uuid4**())

self.objectType = "HBcool"

self.chillerTypeDict = {

-1: 'GroundSourced',

0: 'WaterCooled',

1: 'AirCooled',

'WaterCooled': 'WaterCooled',

'AirCooled': 'AirCooled',

'GroundSourced': 'GroundSourced'

}

**if** coolingAvailSched:

self.coolingAvailSched = coolingAvailSched

**else**:

self.coolingAvailSched = "ALWAYS ON"

**if** coolingCOP != **None**:

self.coolingCOP = **float**(coolingCOP)

**else**:

self.coolingCOP = "Default"

**if** supplyTemperature != **None**:

self.supplyTemperature = **float**(supplyTemperature)

**else**:

self.supplyTemperature = "Default"

**if** pumpMotorEfficiency != **None**:

self.pumpMotorEfficiency = **float**(pumpMotorEfficiency)

**else**:

self.pumpMotorEfficiency = "Default"

**if** coolHardSize != **None**:

self.coolHardSize = coolHardSize

**else**:

self.coolHardSize = "Autosize"

**if** centralPlant != **None**:

self.centralPlant = centralPlant

**else**:

self.centralPlant = "Default"

**if** chillerType != **None**:

self.chillerType = self.chillerTypeDict[chillerType]

**else**:

self.chillerType = "Default"

@classmethod

**def fromTextStr**(cls, textStr):

paramList = []

success = **True**

**for** count, line **in enumerate**(textStr.**split**('\n')):

**if** count == 0:

**if** 'COOLING DETAILS' **not in** line.**upper**():

success = **False**

**else**:

param = line.**split**(': ')[-1]

**if** param.**upper**() != 'DEFAULT':

paramList.**append**(param)

**else**:

paramList.**append**(**None**)

**if** success == **True**:

coolDetailObj = **cls**(paramList[0], paramList[1], paramList[2], paramList[3], paramList[4], paramList[5], paramList[6])

coolDetailObj.areInputsChecked = **True**

return coolDetailObj

**else**:

return **None**

**def checkInputVariables**(self):

errors = []

success = **True**

**if** self.coolingAvailSched != "ALWAYS ON":

success, error = self.sysProps.**checkSchedule**(self.coolingAvailSched)

**if** success **is False**:

errors.**append**(error)

**if** self.pumpMotorEfficiency != "Default":

**if** self.pumpMotorEfficiency > 1 **or** self.pumpMotorEfficiency < 0:

success = **False**

errors.**append**("Pump Motor Efficiency must be betweeon 0 and 1.")

return success, errors

**def checkSysCompatability**(self, sysInt):

errors = []

sysType = self.sysProps.sysDict[sysInt]

coolCapabilities = self.sysProps.coolCapabilities[sysInt]

**if** self.coolingAvailSched != 'ALWAYS ON' **and** coolCapabilities['Avail'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING AVAILABILITY SCHEDULE', 'coolingDetails'))

**if** self.coolingCOP != 'Default' **and** coolCapabilities['COP'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING SYSTEM COP', 'coolingDetails'))

**if** self.supplyTemperature != 'Default' **and** coolCapabilities['SupTemp'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING SYSTEM SUPPLY TEMPERATURE', 'coolingDetails'))

**if** self.pumpMotorEfficiency != 'Default' **and** coolCapabilities['PumpEff'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING SYSTEM PUMP MOTOR EFFICIENCY', 'coolingDetails'))

**if** self.coolHardSize != 'Autosize' **and** coolCapabilities['CoolHardSize'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING SYSTEM HARD SIZE', 'coolingDetails'))

**if** self.centralPlant != 'Default' **and** coolCapabilities['CentralPlant'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING SYSTEM CENTRALIZED PLANT', 'coolingDetails'))

**if** self.chillerType != 'Default' **and** coolCapabilities['ChillType'] == **False**:

errors.**append**(self.sysProps.**generateWarning**(sysType, 'COOLING SYSTEM HEAT REJECTION TPYE', 'coolingDetails'))

**if** (sysInt == 7 **or** 11 <= sysInt <= 15) **and** self.chillerType == 'GroundSourced' **and** self.coolHardSize == 'Autosize':

errors.**append**('Cooling system must be hard sized when using GROUND SOURCED systems with ' + sysType)

**elif** sysInt == 17 **and** self.chillerType == 'AirCooled':

errors.**append**('The system ' + sysType + ' cannot be air cooled.')

return errors

**def class2Str**(self):

allGood = **True**

**if not** self.areInputsChecked:

allGood, errors = self.**checkInputVariables**()

**if** allGood:

textStr = 'Cooling Details\n' + \

' Cooling Availability Schedule: ' + **str**(self.coolingAvailSched) + '\n' + \

' Cooling System COP: ' + **str**(self.coolingCOP) + '\n' + \

' Cooling System Supply Temperature: ' + **str**(self.supplyTemperature) + '\n' + \

' Cooling System Pump Motor Efficiency: ' + **str**(self.pumpMotorEfficiency) + '\n' + \

' Cooling System Hard Size: ' + **str**(self.coolHardSize) + '\n' + \

' Cooling System Centralized Plant: ' + **str**(self.centralPlant) + '\n' + \

' Cooling System Heat Rejection Type: ' + **str**(self.chillerType)

return **True**, textStr

**else**:

return **False**, errors

**class OPSChoice**(object):

**def \_\_init\_\_**(self, originalString):

self.originalString = originalString

self.value = self.**get\_value**()

self.display\_name = self.**get\_display\_name**()

**def get\_display\_name**(self):

return self.originalString.**split**("<display\_name>")[-1].**split**("</display\_name>")[0]

**def get\_value**(self):

return self.originalString.**split**("<value>")[-1].**split**("</value>")[0]

**def \_\_repr\_\_**(self):

return self.display\_name

**class OPSMeasureArg**(object):

**def \_\_init\_\_**(self, originalString):

self.originalString = originalString

self.name = self.**get\_name**()

self.display\_name = self.**get\_display\_name**()

self.description = self.**get\_description**()

self.type = self.**get\_type**()

self.required = self.**get\_required**()

**if** self.required == **True**:

self.display\_name = "\_" + self.display\_name

**else**:

self.display\_name = self.display\_name + "\_"

self.model\_dependent = self.**get\_model\_dependent**()

self.default\_value = self.**get\_default\_value**()

self.choices = self.**get\_choices**()

self.validChoices = [choice.value.**lower**() **for** choice **in** self.choices]

self.userInput = **None**

**def get\_name**(self):

return self.originalString.**split**("<name>")[-1].**split**("</name>")[0]

**def get\_display\_name**(self):

return self.originalString.**split**("</display\_name>")[0].**split**("<display\_name>")[-1]

**def get\_description**(self):

return self.originalString.**split**("<description>")[-1].**split**("</description>")[0]

**def get\_type**(self):

return self.originalString.**split**("<type>")[-1].**split**("</type>")[0]

**def get\_required**(self):

req = self.originalString.**split**("<required>")[-1].**split**("</required>")[0]

return **True if** req.**strip**() == "true" **else False**

**def get\_model\_dependent**(self):

depends = self.originalString.**split**("<model\_dependent>")[-1].**split**("</model\_dependent>")[0]

return **True if** depends.**strip**() == "true" **else False**

**def get\_default\_value**(self):

**if not** "<default\_value>" **in** self.originalString:

return **None**

**else**:

value = self.originalString.**split**("<default\_value>")[-1].**split**("</default\_value>")[0]

**if** self.type.**lower**() != "boolean": return value

return **True if** value.**strip**() == "true" **else False**

**def get\_choices**(self):

choicesContainer = self.originalString.**split**("<choices>")[-1].**split**("</choices>")[0]

choices = [arg.**split**("<choice>")[-1] **for** arg **in** choicesContainer.**split**("</choice>")][:-1]

return [**OPSChoice**(choice) **for** choice **in** choices]

**def update\_value**(self, userInput):

#currently everything is string

**if len**(self.validChoices) == 0:

self.userInput = userInput

**elif str**(userInput).**lower**() **not in** self.validChoices:

#give warning

msg = **str**(userInput) + " is not a valid input for " + self.display\_name + ".\nValid inputs are: " + **str**(self.choices)

**give\_warning**(msg)

**else**:

self.userInput = userInput

**def \_\_repr\_\_**(self):

return (self.display\_name + "<" + self.type + "> " + **str**(self.choices) + \

" Current Value: %s")**%**(self.default\_value **if not** self.userInput **else** self.userInput)

**class OpenStudioMeasure**(object):

**def \_\_init\_\_**(self, xmlFile):

self.nickName = os.path.**normpath**(xmlFile).**split**("\\")[-2]

with **open**(xmlFile, "r") **as** measure:

lines = "".**join**(measure.**readlines**())

self.name = lines.**split**("</display\_name>")[0].**split**("<display\_name>")[-1]

self.description = lines.**split**("</description>")[0].**split**("<description>")[-1]

**if** 'EnergyPlusMeasure' **in** lines:

self.type = 'EnergyPlus'

**elif** 'ModelMeasure' **in** lines:

self.type = 'OpenStudio'

**else**:

self.type = 'Reporting'

self.path = os.path.**normpath**(os.path.**split**(xmlFile)[0])

self.args = self.**get\_measureArgs**(xmlFile)

**def get\_measureArgs**(self, xmlFile):

# there is no good XML parser for IronPython

# here is parsing the file

with **open**(xmlFile, "r") **as** measure:

lines = measure.**readlines**()

argumentsContainer = "".**join**(lines).**split**("<arguments>")[-1].**split**("</arguments>")[0]

arguments = [arg.**split**("<argument>")[-1] **for** arg **in** argumentsContainer.**split**("</argument>")][:-1]

#collect arguments in a dictionary so I can map the values on update

args = **dict**()

**for** count, arg **in enumerate**(arguments):

args[count+1] = **OPSMeasureArg**(arg)

return args

**def \_\_repr\_\_**(self):

return "OpenStudio " + self.name

**class hb\_NonConvexChecking**(object):

"""

This class currently holds isConvex function only. Eventually, this class shall be merged with the other zone spliting class.

"""

**def \_\_init\_\_**(self, surface):

self.surface = surface

**def isConvex**(self):

"""

This function takes a brep surface and checks whether that is convex or non-convex

Args

surface: A brep surface

return

check : True if the surface is convex and False if it is not non-convex.

faultyGeometry : A list of faultyGeometry.

"""

#Getting the center of the base brep surface to find the vector at this point

center = rc.Geometry.AreaMassProperties.**Compute**(self.surface)

center = center.Centroid

#Getting the vector at the center of the base brep surface

face = self.surface.Faces[0]

centerVector = face.**NormalAt**(center[0], center[1])

#Now getting vertices of the base brep surface and sorting those vertice in order

joinedBorder = rc.Geometry.Curve.**JoinCurves**(self.surface.**DuplicateEdgeCurves**())

pts = self.surface.**DuplicateVertices**()

pointsSorted = **sorted**(pts, key =**lambda** pt: joinedBorder[0].**ClosestPoint**(pt)[1])

#Creating two item pairs for all the vertices

#Connecting points of each pair will give us a line per pair

#This line can be used for split the brep

#However, since a brep can't be split by a line in rhinocommon, we'll have to create cuttingBrepss

permutations = itertools.**combinations**(pointsSorted, 2)

pointPairs = [item **for** item **in** permutations]

#Each pair of points are projected on the both the sides of the surface by a certain distance (factor)

#This gives four points for every two points.

#These four points are used to create cuttingBreps.

cutBreps = []

factor = 2

**for** pair **in** pointPairs:

point01 = pair[0]

point02 = pair[1]

direction = centerVector

vertice01 = point01 + direction \* factor

vertice02 = point02 + direction \* factor

vertice03 = point02 + direction \* factor \* -1

vertice04 = point01 + direction \* factor \* -1

cutSurface = rc.Geometry.Brep.**CreateFromCornerPoints**(vertice01, vertice02, vertice03, vertice04, tolerance)

cutBreps.**append**(cutSurface)

#Filtering breps by intersection. This intersection returns a list of curves.

#If the length of the list if 0, there's no intersecction.

#If a baseBrep is not a valid baseBrep, then such baseBreps are to be caught as faultyGeometry

faultyGeometry = []

**try**:

intersections = [rc.Geometry.Intersect.Intersection.**BrepBrep**(cutter, self.surface, tolerance)[1] **for** cutter **in** cutBreps]

**for** curveList **in** intersections:

curveLengthList = [**len**(item) **for** item **in** intersections]

**if** 0 **in** curveLengthList:

check = **False**

**else**:

check = **True**

**except** Exception:

faultyGeometry.**append**(self.surface)

check = **None**

return (check, faultyGeometry)

checkIn = **CheckIn**(defaultFolder\_)

letItFly = **True**

**def checkGHPythonVersion**(target = "0.6.0.3"):

currentVersion = **int**(ghenv.Version.**ToString**().**replace**(".", ""))

targetVersion = **int**(target.**replace**(".", ""))

**if** targetVersion > currentVersion: return **False**

**else**: return **True**

**try**:

downloadTemplate = checkIn.**checkForUpdates**(LB= **False**, HB= **True**, OpenStudio = **True**, template = **True**, therm = **True**)

**except**:

# no internet connection

downloadTemplate = **False**

GHPythonTargetVersion = "0.6.0.3"

**try**:

**if not checkGHPythonVersion**(GHPythonTargetVersion):

**assert False**

**except**:

msg = "Honeybee failed to fly! :(\n" + \

"You are using an old version of GHPython. " +\

"Please update to version: " + GHPythonTargetVersion

**print** msg

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, msg)

checkIn.letItFly = **False**

sc.sticky["honeybee\_release"] = **False**

**if** checkIn.letItFly:

**if not** sc.sticky.**has\_key**("honeybee\_release") **or True**:

w = gh.GH\_RuntimeMessageLevel.Warning

sc.sticky["honeybee\_release"] = **versionCheck**()

folders = **hb\_findFolders**()

# Function to sort vrsions of software

**def getversion**(filePath):

ver = ''.**join**(s **for** s **in** filePath **if** (s.**isdigit**() **or** s == '.'))

return **sum**(**int**(d) **\*** (10 \*\* i) **for** i, d **in enumerate**(**reversed**(ver.**split**('.'))))

sc.sticky["honeybee\_folders"] = {}

# supported versions for EnergyPlus

EPVersions = ["V9-3-0", "V9-2-0", "V9-1-0", "V9-0-1", "V9-0-0", "V8-9-0", "V8-8-0", \

"V8-7-0", "V8-6-0", "V8-5-0", "V8-4-0", "V8-3-0", "V8-2-10", \

"V8-2-9", "V8-2-8", "V8-2-7", "V8-2-6", \

"V8-2-5", "V8-2-4", "V8-2-3", "V8-2-2", "V8-2-1", "V8-2-0", \

"V8-1-5", "V8-1-4", "V8-1-3", "V8-1-2", "V8-1-1", "V8-1-0"]

EPVersion = ''

**if** folders.EPPath != **None**:

# Honeybee has already found EnergyPlus make sure it's an acceptable version

EPVersion = os.path.**split**(folders.EPPath)[-1].**split**("EnergyPlus")[-1]

**if** EPVersion **not in** EPVersions:

#Not an acceptable version so remove it from the path

folders.EPPath = **None**

**if** folders.EPPath == **None**:

**for** EPVers **in** EPVersions:

**if** os.path.**isdir**("C:\EnergyPlus" + EPVers + "\\"):

folders.EPPath = "C:\EnergyPlus" + EPVers + "\\"

EPVersion = EPVers

# check for OpenStudio Folder.

openStudioLibFolder = **None**

QtFolder = **None**

installedOPS1 = [f **for** f **in** os.**listdir**("C:\\Program Files") **if** f.**lower**().**startswith**("openstudio")]

installedOPS2 = [f **for** f **in** os.**listdir**("C:\\") **if** f.**lower**().**startswith**("openstudio")]

**try**:

installedOPS1 = **sorted**(installedOPS1, key=getversion, reverse=**True**)

installedOPS2 = **sorted**(installedOPS2, key=getversion, reverse=**True**)

**except** Exception **as** e:

**print**('Failed to sort OpenStudio installation folders.')

**if len**(installedOPS2) != 0:

installedOPS = installedOPS2[0]

openStudioLibFolder = "C:/%s/CSharp/openstudio"%installedOPS

QtFolder = "C:/%s/Ruby/"%installedOPS

# Grab the version of EP that installs with OpenStudio

**if** os.path.**isdir**("C:/%s/EnergyPlus/"%installedOPS):

folders.EPPath = "C:/%s/EnergyPlus/"%installedOPS

**try**:

opsNum = **int**(''.**join**(installedOPS.**split**('-')[-1].**split**('.')))

**if** opsNum >= 300:

EPVersion = ">=9-3-0"

**elif** opsNum >= 270:

EPVersion = ">9-0-0"

**else**:

EPVersion = "<8-9-0"

**except**:

EPVersion = ""

**if** os.path.**isdir**(openStudioLibFolder) **and** os.path.**isfile**(os.path.**join**(openStudioLibFolder, "OpenStudio.dll")):

# Add Openstudio to the path if it is not there already.

**if not** openStudioLibFolder **in** os.environ['PATH'] **or** QtFolder **not in** os.environ['PATH']:

os.environ['PATH'] = ";".**join**([openStudioLibFolder, QtFolder, os.environ['PATH']])

# Try to download the EP Run Files that are no longer distributed with OpenStudio but are necessary for Honeybee.

**if not** os.path.**isfile**('C:/%s/EnergyPlus/Epl-run.bat'%installedOPS):

**try**:

client = System.Net.**WebClient**()

client.**DownloadFile**('https://github.com/mostaphaRoudsari/honeybee/raw/master/resources/EPRunFiles/Epl-run.bat', 'C:/%s/EnergyPlus/Epl-run.bat'%installedOPS)

**except** Exception, e:

warning = 'Failed to download the files needed to run EnergyPlus with OpenStudio 2.x.'

warning += '\n' + `e`

**print** warning

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warning)

**if not** os.path.**isdir**('C:/%s/EnergyPlus/PostProcess/'%installedOPS):

**try**:

client = System.Net.**WebClient**()

client.**DownloadFile**('https://github.com/mostaphaRoudsari/honeybee/raw/master/resources/EPRunFiles/PostProcess.zip', 'C:/%s/EnergyPlus/PostProcess.zip'%installedOPS)

sourceFile = 'C:/%s/EnergyPlus/PostProcess.zip'%installedOPS

with zipfile.**ZipFile**(sourceFile) **as** zf:

**for** member **in** zf.**infolist**():

words = member.filename.**split**('\\')

path = 'C:/%s/EnergyPlus/'%installedOPS

**for** word **in** words[:-1]:

drive, word = os.path.**splitdrive**(word)

head, word = os.path.**split**(word)

**if** word **in** (os.curdir, os.pardir, ''): continue

path = os.path.**join**(path, word)

zf.**extract**(member, path)

**except** Exception, e:

warning = 'Failed to download the files needed to PostProcess EnergyPlus results with OpenStudio 2.x.'

warning += '\n' + `e`

**print** warning

ghenv.Component.**AddRuntimeMessage**(gh.GH\_RuntimeMessageLevel.Warning, warning)

**if not** os.path.**isdir**('C:/%s/EnergyPlus/PreProcess/'%installedOPS):

os.**mkdir**('C:/%s/EnergyPlus/PreProcess/'%installedOPS)

**else**:

openStudioLibFolder = **None**

QtFolder = **None**

**elif len**(installedOPS1) != 0:

installedOPS = installedOPS1[0]

openStudioLibFolder = "C:/Program Files/%s/CSharp/openstudio/"%installedOPS

QtFolder = "C:/Program Files/%s/Ruby/openstudio/"%installedOPS

**for** EPVers **in** EPVersions:

versStr = EPVers.**replace**('V', '-')

**if** os.path.**isdir**("C:/Program Files/%s/share/openstudio/"%installedOPS + "EnergyPlus" + versStr + "/"):

folders.EPPath = "C:/Program Files/%s/share/openstudio/"%installedOPS + "EnergyPlus" + versStr + "//"

EPVersion = EPVers

**elif** os.path.**isdir**("C:/Program Files/%s/share/openstudio/"%installedOPS + "EnergyPlus" + EPVers + "/"):

folders.EPPath = "C:/Program Files/%s/share/openstudio/"%installedOPS + "EnergyPlus" + EPVers + "//"

EPVersion = EPVers

**if** os.path.**isdir**(openStudioLibFolder) **and** os.path.**isfile**(os.path.**join**(openStudioLibFolder, "openStudio.dll")):

# openstudio is there and we are good to go.

# add folders to path.

**if not** openStudioLibFolder **in** os.environ['PATH'] **or** QtFolder **not in** os.environ['PATH']:

os.environ['PATH'] = ";".**join**([openStudioLibFolder, QtFolder, os.environ['PATH']])

**else**:

openStudioLibFolder = **None**

QtFolder = **None**

**if** openStudioLibFolder == **None or** QtFolder == **None**:

msg1 = "Honeybee cannot find OpenStudio on your system.\n" + \

"You wont be able to use the Export to OpenStudio component.\n" + \

"Download the latest OpenStudio for Windows from:\n"

msg2 = "https://www.openstudio.net/downloads"

**print** msg1

**print** msg2

ghenv.Component.**AddRuntimeMessage**(w, msg1)

ghenv.Component.**AddRuntimeMessage**(w, msg2)

**else**:

**print** "Found installation of " + installedOPS + "."

**if** folders.EPPath == **None**:

# give a warning to the user

msg= "Honeybee cannot find an EnergyPlus folder on your system.\n" + \

"You wont be able to use the Run Energy Simulation component.\n" + \

"Honeybee supports following versions of EnergyPlus:\n"

versions = ", ".**join**(EPVersions)

msg += versions

**print** msg

**else**:

**print** "Found installation of EnergyPlus" + EPVersion + "."

sc.sticky["honeybee\_folders"]["OSLibPath"] = openStudioLibFolder

sc.sticky["honeybee\_folders"]["OSQtPath"] = QtFolder

sc.sticky["honeybee\_folders"]["EPPath"] = folders.EPPath

sc.sticky["honeybee\_folders"]["EPVersion"] = EPVersion.**replace**("-", ".")[1:]

# Check for an installation of Radiance.

**if** folders.RADPath == **None**:

**if** os.path.**isdir**("c:\\radiance\\bin\\"):

folders.RADPath = "c:\\radiance\\bin\\"

**else**:

msg= "Honeybee cannot find RADIANCE folder on your system.\n" + \

"Make sure you have RADIANCE installed on your system.\n" + \

"You won't be able to run daylighting studies without RADIANCE.\n" + \

"A good place to install RADIANCE is c:\\radiance"

ghenv.Component.**AddRuntimeMessage**(w, msg)

folders.RADPath = ""

**if** folders.RADPath != **None**:

versiFile = "\\".**join**(folders.RADPath.**split**('\\')[:-1]) + "\\NREL\_ver.txt"

**if** os.path.**isfile**(versiFile):

with **open**(versiFile) **as** verFile:

currentRADVersion = verFile.**readline**().**strip**()

**print** "Found installation of " + currentRADVersion + "."

**else**:

**print** "Found installation of Radiance."

**if** folders.RADPath.**find**(" ") > -1:

msg = "There is a white space in RADIANCE filepath: " + folders.RADPath + "\n" + \

"Please install RADIANCE in a valid address (e.g. c:\\radiance)"

ghenv.Component.**AddRuntimeMessage**(w, msg)

folders.RADPath = ""

**if** folders.RADPath.**endswith**("\\"): segmentNumber = -2

**else**: segmentNumber = -1

hb\_RADLibPath = "\\".**join**(folders.RADPath.**split**("\\")[:segmentNumber]) + "\\lib"

sc.sticky["honeybee\_folders"]["RADPath"] = folders.RADPath

sc.sticky["honeybee\_folders"]["RADLibPath"] = hb\_RADLibPath

# Check for installation of DAYSIM

**if** folders.DSPath == **None**:

**if** os.path.**isdir**("c:\\daysim\\bin\\"):

folders.DSPath = "c:\\daysim\\bin\\"

**print** "Found installation of DAYSIM."

**else**:

msg= "Honeybee cannot find DAYSIM folder on your system.\n" + \

"Make sure you have DAYISM installed on your system.\n" + \

"You won't be able to run annual climate-based daylighting studies without DAYSIM.\n" + \

"A good place to install DAYSIM is c:\\DAYSIM"

ghenv.Component.**AddRuntimeMessage**(w, msg)

folders.DSPath = ""

**else**:

**print** "Found installation of DAYSIM."

**if** folders.DSPath.**find**(" ") > -1:

msg = "There is a white space in DAYSIM filepath: " + folders.DSPath + "\n" + \

"Please install Daysism in a valid address (e.g. c:\\daysim)"

ghenv.Component.**AddRuntimeMessage**(w, msg)

folders.DSPath = ""

**if** folders.DSPath.**endswith**("\\"): segmentNumber = -2

**else**: segmentNumber = -1

hb\_DSCore = "\\".**join**(folders.DSPath.**split**("\\")[:segmentNumber])

hb\_DSLibPath = "\\".**join**(folders.DSPath.**split**("\\")[:segmentNumber]) + "\\lib"

sc.sticky["honeybee\_folders"]["DSPath"] = folders.DSPath

sc.sticky["honeybee\_folders"]["DSCorePath"] = hb\_DSCore

sc.sticky["honeybee\_folders"]["DSLibPath"] = hb\_DSLibPath

# Check for an installation of THERM.

THERMVersions = ["7.5", "7.6"]

THERMVersion = ''

THERMSettingsFile = ''

**if** folders.THERMPath != **None**:

# Honeybee has already found a version of THERM. Make sure it's an acceptable version

THERMVersion = os.path.**split**(folders.THERMPath)[-1].**split**("THERM")[-1]

**if** THERMVersion **not in** THERMVersions:

#Not an acceptable version so remove it from the path

folders.THERMPath = **None**

**else**:

**print** "Found installation of THERM " + THERMVersion + "."

**if** folders.THERMPath == **None**:

**if** os.path.**isdir**("C:/Program Files (x86)/lbnl/"):

installedTHERM = [f **for** f **in** os.**listdir**("C:/Program Files (x86)/lbnl/") **if** f.**startswith**("THERM")]

**try**:

installedTHERM = **sorted**(installedTHERM, key=getversion, reverse=**True**)

**except** Exception **as** e:

**print**('Failed to sort THERM installation folders.')

**if len**(installedTHERM) != 0:

**for** thermInstall **in** installedTHERM:

THERMVersionInit = thermInstall.**split**("THERM")[-1]

**if** THERMVersionInit **in** THERMVersions:

THERMVersion = THERMVersionInit

folders.THERMPath = "C:/Program Files (x86)/lbnl/%s/"%thermInstall

**print** "Found installation of THERM " + THERMVersion + "."

**if** folders.THERMPath == **None**:

msg= "Honeybee cannot find a compatible LBNL THERM installation on your system.\n" + \

"You won't be able to run THERM simulations of heat flow through constructions.\n" + \

"Only the following versions of THERM are supported: {}".**format**(THERMVersions) + \

"\nDownload supported versions of THERM from:"

msg2 = "https://windows.lbl.gov/software/therm"

ghenv.Component.**AddRuntimeMessage**(w, msg)

ghenv.Component.**AddRuntimeMessage**(w, msg2)

folders.THERMPath = ""

**else**:

**if** os.path.**isfile**('C:/Users/Public/LBNL/Settings/therm%s.ini'%THERMVersion):

THERMSettingsFile = 'C:/Users/Public/LBNL/Settings/therm%s.ini'%THERMVersion

**else**:

msg= "Failed to load THERM settings file.\n" + \

"You won't be able to run THERM simulations."

ghenv.Component.**AddRuntimeMessage**(w, msg)

**if not** folders.THERMPath **in** os.environ['PATH']:

os.environ['PATH'] = ";".**join**([folders.THERMPath, os.environ['PATH']])

sc.sticky["honeybee\_folders"]["THERMPath"] = folders.THERMPath

sc.sticky["honeybee\_folders"]["ThermSettings"] = THERMSettingsFile

# initiate an empty library in case this is the first time honeybee is flying in this document

# otherwise it has been already created/

**print** ""

**if** "honeybee\_Hive" **not in** sc.sticky:

sc.sticky["honeybee\_RADMaterialLib"] = **dict**()

# set up radiance materials

RADMaterialAux = **RADMaterialAux**(**True**, sc.sticky["honeybee\_RADMaterialLib"], sc.sticky["Honeybee\_DefaultFolder"])

sc.sticky["honeybee\_RADMaterialAUX"] = RADMaterialAux

# Download EP libraries

templateFilesPrep = **PrepareTemplateEPLibFiles**(downloadTemplate)

libFilePaths = templateFilesPrep.**downloadTemplates**()

msg = "Failed to load EP constructions! You won't be able to run analysis with Honeybee!\n" + \

"Download the files from address below and copy them to: " + sc.sticky["Honeybee\_DefaultFolder"] + \

"\nhttps://github.com/mostaphaRoudsari/Honeybee/tree/master/resources\n"

**if** libFilePaths != -1:

EPLibs = **HB\_GetEPLibraries**()

**try**:

**for** pathCount, path **in enumerate**(libFilePaths):

**if** "honeybee\_Hive" **not in** sc.sticky:

# This is first time loading so clean the library

cleanLibs = **True if** pathCount == 0 **else False**

**else**:

cleanLibs = **False**

**if** path.**endswith**('.csv'): isMatFile = **True**

**else**: isMatFile = **False**

EPLibs.**importEPLibrariesFromFile**(path, isMatFile, cleanLibs, **False**)

EPLibs.**report**()

sc.sticky["honeybee\_materialLib"].**update**(EPLibs.**getEPMaterials**())

sc.sticky["honeybee\_windowMaterialLib"].**update**(EPLibs.**getEPWindowMaterial**())

sc.sticky ["honeybee\_constructionLib"].**update**(EPLibs.**getEPConstructions**())

sc.sticky["honeybee\_ScheduleLib"].**update**(EPLibs.**getEPSchedule**())

sc.sticky["honeybee\_ScheduleTypeLimitsLib"].**update**(EPLibs.**getEPScheduleTypeLimits**())

sc.sticky["honeybee\_thermMaterialLib"].**update**(EPLibs.**getTHERMMaterials**())

sc.sticky["honeybee\_WindowPropLib"].**update**(EPLibs.**getEPWindowProp**())

sc.sticky["honeybee\_SpectralDataLib"].**update**(EPLibs.**getEPSpectralData**())

**except**:

**print** msg

ghenv.Component.**AddRuntimeMessage**(w, msg)

**else**:

**print** msg

ghenv.Component.**AddRuntimeMessage**(w, msg)

sc.sticky["honeybee\_Hive"] = hb\_Hive

sc.sticky["honeybee\_generationHive"] = generationhb\_hive

sc.sticky["honeybee\_GetEPLibs"] = HB\_GetEPLibraries

sc.sticky["honeybee\_DefaultMaterialLib"] = materialLibrary

sc.sticky["honeybee\_DefaultSurfaceLib"] = EPSurfaceLib

sc.sticky["honeybee\_EPMaterialAUX"] = EPMaterialAux

sc.sticky["honeybee\_EPScheduleAUX"] = EPScheduleAux

sc.sticky["honeybee\_EPObjectsAUX"] = EPObjectsAux

sc.sticky["honeybee\_ReadSchedules"] = ReadEPSchedules

sc.sticky["honeybee\_BuildingProgramsLib"] = BuildingProgramsLib

sc.sticky["honeybee\_EPTypes"] = **EPTypes**()

sc.sticky["honeybee\_EPZone"] = EPZone

sc.sticky["honeybee\_EPHvac"] = EPHvac

sc.sticky["honeybee\_Measure"] = OpenStudioMeasure

sc.sticky["honeybee\_MeasureArg"] = OPSMeasureArg

sc.sticky["honeybee\_ThermPolygon"] = thermPolygon

sc.sticky["honeybee\_ThermBC"] = thermBC

sc.sticky["honeybee\_ThermDefault"] = thermDefaults

sc.sticky["honeybee\_ViewFactors"] = viewFactorInfo

sc.sticky["PVgen"] = PV\_gen

sc.sticky["PVinverter"] = PVinverter

sc.sticky["HB\_generatorsystem"] = HB\_generatorsystem

sc.sticky["wind\_generator"] = Wind\_gen

sc.sticky["simple\_battery"] = simple\_battery

sc.sticky["thermBCCount"] = 1

sc.sticky["hBZoneCount"] = 0

sc.sticky["honeybee\_reEvaluateHBZones"] = hb\_reEvaluateHBZones

sc.sticky["honeybee\_hvacProperties"] = hb\_hvacProperties

sc.sticky["honeybee\_hvacAirDetails"] = hb\_airDetail

sc.sticky["honeybee\_hvacHeatingDetails"] = hb\_heatingDetail

sc.sticky["honeybee\_hvacCoolingDetails"] = hb\_coolingDetail

sc.sticky["honeybee\_EPSurface"] = hb\_EPSurface

sc.sticky["honeybee\_EPShdSurface"] = hb\_EPShdSurface

sc.sticky["honeybee\_EPZoneSurface"] = hb\_EPZoneSurface

sc.sticky["honeybee\_EPFenSurface"] = hb\_EPFenSurface

sc.sticky["honeybee\_GlzGeoGeneration"] = hb\_GlzGeoGeneration

sc.sticky["honeybee\_DLAnalysisRecipe"] = DLAnalysisRecipe

sc.sticky["honeybee\_MeshToRAD"] = hb\_MSHToRAD

sc.sticky["honeybee\_WriteRAD"] = hb\_WriteRAD

sc.sticky["honeybee\_WriteRADAUX"] = hb\_WriteRADAUX

sc.sticky["honeybee\_WriteDS"] = hb\_WriteDS

sc.sticky["honeybee\_RADParameters"] = hb\_RADParameters

sc.sticky["honeybee\_DSParameters"] = hb\_DSParameters

sc.sticky["honeybee\_ReadAnnualResultsAux"] = hb\_ReadAnnualResultsAux

sc.sticky["honeybee\_EPParameters"] = hb\_EnergySimulatioParameters

sc.sticky["honeybee\_SerializeObjects"] = SerializeObjects

sc.sticky["honeybee\_GridBasedDLResults"] = CalculateGridBasedDLAnalysisResults

sc.sticky["honeybee\_DLAnalaysisTypes"] = {0: ["0: illuminance" , "lux"],

1: ["1: radiation" , "wh/m2"],

1.1: ["1.1: cumulative radiation" , "kWh/m2"],

2: ["2: luminance" , "cd/m2"],

3: ["3: DF", "%"],

4: ["4: VSC", "%"],

5: ["5: annual analysis", "var"]}

sc.sticky["honeybee\_NonConvexChecking"] = hb\_NonConvexChecking

sc.sticky["honeybee\_ConversionFactor"] = **checkUnits**()

# done! sharing the happiness.

**print** "Hooohooho...Flying!!\nVviiiiiiizzz..."

# push honeybee component to back

ghenv.Component.**OnPingDocument**().**SelectAll**()

ghenv.Component.Attributes.Selected = **False**

ghenv.Component.**OnPingDocument**().**BringSelectionToTop**()

ghenv.Component.**OnPingDocument**().**DeselectAll**()