```
import maya.cmds as mc
import maya.OpenMaya as om
from functools import partial
# Gina use regular expressions to match the shader name
import re

'''
22_07_15
--> load swatches from shaders

'''
def toggleDefaultMaterial(*args):
    # get teh current model panel
    currentPanel = mc.getPanel(withFocus=True)
    print currentPanel


    if mc.getPanel(typeOf=currentPanel) != 'modelPanel':
        print 'this is not a model panel'

    else:
        #query the current status
        status = mc.modelEditor(currentPanel, query=True,
useDefaultMaterial=True)

        # if off set the useDefaultMaterial attribute to on
        if status == False:
            mc.modelEditor('modelPanel4', edit=True,
useDefaultMaterial=True)

            # if on set the useDefaultMaterial attribute to off
        if status == True:
            mc.modelEditor('modelPanel4', edit=True,
useDefaultMaterial=False)

####API functions from Shaun
def getMObjectFromNode(mayaNode):
    ''' Get an MObject from a node name in Maya '''
    if not isinstance(mayaNode, basestring):
        print('Please pass in a string value')
        return None
    # Create a selection list object and add the node
    selectionList = om.MSelectionList()
    selectionList.add(mayaNode)
    # Grab the node object from the selection
    result = om.MObject()
    selectionList.getDependNode(0, result)
    # Return the MObject
    return result

def getNodeName(mObject):
    ''' Notice that this method takes an mObject as an input, not a maya
node name '''
    if mObject.hasFn(om.MFn.kDagNode):
        name = om.MFnDagNode(mObject).fullPathName()
    else:
        name = om.MFnDependencyNode(mObject).name()
    return name


def getSGfromMaterial(shaderName):
    material = shaderName

    nodeType = mc.nodeType(material)
    isShader = mc.getClassification(nodeType,satisfies="shader/surface")
    # print item, nodeType, isShader

    # Assert that there is a connection from the Material's .outColor
    if mc.connectionInfo("{material}.outColor".format(material=material),
isSource=True):
        # print 'material is source'

        # there may be more than one destination
        destinations =
mc.connectionInfo("{material}.outColor".format(material=material),
destinationFromSource=True)
        # print 'destinations are ', destinations

        #Iterate through connections and identify ShadingGroup sets
        for item in destinations:
            SG = item.split('.')[0]
            # print mc.nodeType(SG)
            if mc.nodeType(SG) == "shadingEngine":
                return SG

class ColourFacesTool(object):

    def __init__(self, parentLayout, numberOfColumns=8):
        self.shaderType = "surface"


        self.colourSwatchWidth = 45
        self.formOffset = 2
        self.colourPalette = None
        self.parentLayout = parentLayout
        self.defaultColour = [0.5, 0.5, 0.5]
        self.selectionSetCBList = []

        self.recordColourTweakStatus = False
        self.viewportBGTop = [0.5, 0.5, 0.5]
        self.viewportBGBottom = [0.5, 0.5, 0.5]

        self.rows = 2
        self.columns = numberOfColumns

        self.buildUI()


    def buildUI(self):
        separation = 15

        width = self.columns * self.colourSwatchWidth
        height = self.rows * self.colourSwatchWidth
        buttonWidth = self.colourSwatchWidth + self.colourSwatchWidth *
0.5
```

```
        #### RECORD COLOUR radio buttons
        mc.text('-----  Record colour tweaks and alter viewport display -
-----', parent=self.parentLayout, align='right' )
        CBlayout = mc.rowLayout(numberOfColumns=3,
parent=self.parentLayout)
        self.recordColourRB = mc.checkBox(label='Record colour tweaks',
parent=CBlayout, \
            changeCommand=(self.setRecordColourTweakStatus))
        self.viewportBGcB = mc.checkBox(label='Colour viewport BG',
parent=CBlayout, \
            changeCommand=(self.colourViewportBG))
        self.toggleDefaultMatButton = mc.button(label =
'defaultMaterial', parent = CBlayout, \
            command=toggleDefaultMaterial)

        #### adding ability for SS Blinn and Lambert
        mc.text('-----  Choose shader type ------',
parent=self.parentLayout, align='right' )
        materialTypeLayout = mc.rowLayout(numberOfColumns=3,
parent=self.parentLayout)
        self.surfaceShaderRB = mc.radioButton(label='Surface Shader',
parent=materialTypeLayout, \
            select=True, onCommand=(partial(self.surfaceShaderStatus,
True)))
        self.lambertRB = mc.radioButton(label='Lambert',
parent=materialTypeLayout, select=False,\
            onCommand=(partial(self.setLambertStatus, True)))
        self.blinnRB = mc.radioButton(label='Blinn',
parent=materialTypeLayout, select=False,\
            onCommand=(partial(self.setBlinnStatus, True)))

        mc.text('-----  Assign colours to objects and faces ------',
parent=self.parentLayout, align='right' )
        # separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        palletteRow = mc.rowLayout(numberOfColumns=10,
parent=self.parentLayout)
        #make the palettePort
        self.colourPalette = mc.palettePort(dimensions = (self.columns,
self.rows), width=width, height=height, \
            topDown=True, colorEdited=(self.makeOrEditShader),
parent=palletteRow)

        #set each swatch to grey
        for index in range(self.rows * self.columns):
            mc.palettePort(self.colourPalette, edit=True, \
                rgbValue =(index, self.defaultColour[0],
self.defaultColour[1], self.defaultColour[2]))

        # make buttons  -----------
        buttonRow = mc.rowLayout(numberOfColumns=6,
parent=self.parentLayout)
        self.assignColourButton = mc.button(label = '<<<<',
width=buttonWidth, parent = buttonRow, \
            command=self.assignColour, backgroundColor=(0.6, 0.45, 0.45))
        self.assignColourButton = mc.button(label = '\\ viewport', 
width=54, parent = buttonRow, \
            command=self.setViewportBGtop)
        self.assignColourButton = mc.button(label = '/ viewport', 
width=54, parent = buttonRow, \
            command=self.setViewportBGbottom)

        self.grabColourButton = mc.button(label = '>grab colour', 
width=buttonWidth, parent = buttonRow, \
            command=self.grabColour)
        self.resetSwtchButton = mc.button(label = 'reset swatch', 
width=buttonWidth, parent = buttonRow, \
            command=self.resetSwatch)

        self.resetSwtchButton = mc.button(label = 'LOAD', 
width=buttonWidth * 0.65, parent = buttonRow, \
            command=self.loadSwatches, backgroundColor=(0.6, 0.45, 0.45))

        separator = mc.separator(style ="none", w=300, h=separation, 
parent=self.parentLayout)
        mc.text('-----  Create selection sets ------', 
parent=self.parentLayout, align='right' )
        selectionButtonRow = mc.rowLayout(numberOfColumns=2, 
parent=self.parentLayout)
        self.createSelectionSetButton = mc.button(label = '>>>> create 
set', width=self.colourSwatchWidth+buttonWidth, \
            parent = selectionButtonRow, command=self.addSelectionSetUI)
        self.selectionNameTF = mc.textField('selectionNameTF', 
parent=selectionButtonRow, width=buttonWidth, text='setName')

        self.selectionGroupLayout = 
mc.columnLayout(parent=self.parentLayout)


############## set shader type
    def surfaceShaderStatus(self, surfaceShaderStatus, *args):
        self.shaderType = 'surface'
        print 'shader type is   ', self.shaderType

    def setLambertStatus(self, lambertStatus, *args):
        self.shaderType = "lambert"
        print 'shader type is   ', self.shaderType

    def setBlinnStatus(self, blinnStatus, *args):
        self.shaderType = "blinn"
        print 'shader type is   ', self.shaderType


############# viewportBG stuff
    def colourViewportBG(self, *args):
        ### get the status of the checkBox
        CBstatus = mc.checkBox(self.viewportBGcB, value=True, query=True)

        if CBstatus == True:
            mc.displayPref(displayGradient=True)
        else:
            mc.displayPref(displayGradient=False)
        #set the vieport ramp colour
        bgColourBottom = (mc.getAttr
("viewportBGcolour.colorEntryList[0].color"))[0]
        bgColourTop = (mc.getAttr
("viewportBGcolour.colorEntryList[1].color"))[0]
        mc.displayRGBColor
("backgroundBottom",bgColourBottom[0],bgColourBottom[1],bgColourBottom[2]
)
        mc.displayRGBColor
("backgroundTop",bgColourTop[0],bgColourTop[1],bgColourTop[2])
        # print 'self.viewportBGcolour is ', self.viewportBGcolour

    def setViewportBGtop(self, *args):
        ### get active swatch current colour
        currentColour = mc.palettePort( self.colourPalette, query=True,
rgb=True )
        #### set the self.viewportBGcolour to current colour
        self.viewportBGTop = currentColour
        self.updateBGramp()
        self.colourViewportBG()
        print 'ViewportBGtop currentColour is ', currentColour

    def setViewportBGbottom(self, *args):
        ### get active swatch current colour
        currentColour = mc.palettePort( self.colourPalette, query=True,
rgb=True )
        #### set the self.viewportBGcolour to current colour
        self.viewportBGBottom = currentColour
        self.updateBGramp()
        self.colourViewportBG()
        print 'ViewportBGbottom currentColour is ', currentColour

    def updateBGramp(self):
        ### create the ramp and / or update colours
        topColourR = self.viewportBGTop[0]
        topColourG = self.viewportBGTop[1]
        topColourB = self.viewportBGTop[2]
        bottColourR = self.viewportBGBottom[0]
        bottColourG = self.viewportBGBottom[1]
        bottColourB = self.viewportBGBottom[2]

        #### make a ramp if its not been made
        viewportBGramp = 'viewportBGcolour'
        if mc.objExists(viewportBGramp) == False:
            viewportBGramp = mc.shadingNode("ramp",asTexture=True, name =
'viewportBGcolour')
                    #### if theres more than two entries delete the third
            numEntries =
mc.getAttr('{baseRamp}.colorEntryList'.format(baseRamp=baseRamp),
size=True)
            if numEntries >= 3:

mc.removeMultiInstance('viewportBGcolour.colorEntryList[2]')

        ## update BG colours

mc.setAttr('{ramp}.colorEntryList[0].color'.format(ramp=viewportBGramp),
bottColourR,bottColourG,bottColourB)
```

```
mc.setAttr('{ramp}.colorEntryList[1].color'.format(ramp=viewportBGramp),
topColourR,topColourG,topColourB)

mc.setAttr('{ramp}.colorEntryList[1].position'.format(ramp=viewportBGramp
), 1)

#############################
    def setRecordColourTweakStatus(self, *args):
        ### check teh status of the checkbox
        status = mc.checkBox(self.recordColourRB, value=True, query=True)

        if status == True:
            self.recordColourTweakStatus = True
        else:
            self.recordColourTweakStatus = False
        print status, self.recordColourTweakStatus

    def addSelectionSetUI(self, *args):
        separation = 15
        # get the current cell colour
        currentcell = mc.palettePort(self.colourPalette, setCurCell=True,
query=True)
        currentColour = mc.palettePort( self.colourPalette, query=True,
rgb=True )

        selectionSetName = mc.textField(self.selectionNameTF, query=True,
text=True)
        print 'selectionSetName is ', selectionSetName
        ### check if name is unique
        if (mc.objExists(selectionSetName)) == True:
            print 'Please choose unique name for selection set'

        else:
            #create selection set
            selectionSet = self.createSelectionSet(selectionSetName)

            ### add text and check box with name of the selection grp
            separator = mc.separator(style ="none", w=300, h=separation,
parent=self.selectionGroupLayout)
            # selectionGrptext = mc.text(label= selectionSetName,
parent=self.selectionGroupLayout)
            ## create update button
            selectionCntrlRow = mc.rowLayout(numberOfColumns=4,
parent=self.selectionGroupLayout)
            print 'parent row == ', selectionCntrlRow

            addMemberButton =
mc.button('{selectionSetName}AddBtn'.format(selectionSetName=selectionSet
Name), label = '>> add', width=self.colourSwatchWidth,\
                parent = selectionCntrlRow,
command=(partial(self.addMember, selectionSetName)))


            updateButton =
mc.button('{selectionSetName}UpdateBtnn'.format(selectionSetName=selectio
nSetName), label = 'update', width=self.colourSwatchWidth,\
                parent = selectionCntrlRow,
command=(partial(self.updateMembers, selectionSetName)))

    #### set the background colour of the
button############################ press it to update
            colourSwatchButton =
mc.button('{selectionSetName}SwatchBtn'.format(selectionSetName=selection
SetName), label = 'swatch', width=self.colourSwatchWidth, \
                backgroundColor= (currentColour[0], currentColour[1],
currentColour[2]),
                parent = selectionCntrlRow,
command=partial(self.updateSelectionSwatch, selectionCntrlRow))

            #create the CB and add it to the list of CBs
            selectionSetCB = mc.checkBox(label= selectionSetName,
parent=selectionCntrlRow, value=True, onCommand=self.selectComponents, \
                offCommand=self.selectComponents)
            # print 'latest selectionSetCB is ', selectionSetCB
            self.selectionSetCBList.append(selectionSetCB)

    def updateSelectionSwatch(self, parentRow, *args):
        ### this function checks the currently selected pallette colour
and updates the swatch in selection set row
        # get the current cell colour
        currentcell = mc.palettePort(self.colourPalette, setCurCell=True,
query=True)
        currentColour = mc.palettePort( self.colourPalette, query=True,
rgb=True )
        #get the swatch button from its parent row
        swatchButton = (mc.rowLayout(parentRow, query=True,
childArray=True))[-2]
        ## edit theh colour
        mc.button(swatchButton, edit=True,
backgroundColor=(currentColour[0], currentColour[1], currentColour[2]))


    def updateMembers(self, selectionSetName, *args):
        ### I want to not clear this set and remake but rather add and
remove
        ### cant remember why but something to do with the autoProcedure
        currentSelection = mc.ls(selection=True, flatten=True)
                ###remove faces in this set from other sets
        self.removeFromPreviousObjectSet(currentSelection)

        ## clear the current selection set
        mc.sets(clear=selectionSetName)
        ### addCurrently selected elements
        mc.sets(currentSelection, addElement=selectionSetName)

        ### refresh component selection
        self.selectComponents()

    def removeFromPreviousObjectSet(self,faceList):
        ###NB have to check if it is in the current selection list
        #flatten the list
        flatFaceList = mc.ls(faceList, flatten=True)
        for face in flatFaceList:
            allSets = mc.listSets(object=face)
            #test what type of set to rule out deformer sets and
rendering sets
            if allSets != 'NoneType':
                for previousSet in allSets:
                    if mc.nodeType(previousSet) == 'objectSet':
                        print 'Previous objectSet was ', previousSet
                        #remove from that set
                        mc.sets( face, remove=previousSet)
            else:
                print '>>>> allSets == NoneType'


    def addMember(self, selectionSetName, *args):
        currentSelection = mc.ls(selection=True, flatten=True)

        facesInSet = mc.sets(selectionSetName, query=True)
        flatFacesInSet = mc.ls(facesInSet, flatten=True)
        ### test membership of each face
        for face in currentSelection:

            #test if Face is a member
            isMember = mc.sets(face,im=selectionSetName)
            # print face, isMember

            if isMember == False:
                # if it is not currently a member delete it from current
set if any
                ##get list of sets that face belongs to
                allSets = mc.listSets(object=face)
                print 'allSets == ', allSets

                #test what type of set to rule out deformer sets and
rendering sets
                for mySet in allSets:
                    if mc.nodeType(mySet) == 'objectSet':
                        print 'objectSet is ', mySet
                        #remove from that set
                        mc.sets( face, remove=mySet)

                #if it is not currently a member then add it
                mc.sets(face, addElement=selectionSetName)

        ### refresh component selection
        self.selectComponents()


    def createSelectionSet(self, selectionSetName):
        currentSelection = mc.ls(selection=True, flatten=True)
        surfaceShader = mc.ls(selection=True,
excludeType='surfaceShader')
        print '>>> surfaceShader = ', surfaceShader
        #get the selection without the surface shader
        if len(surfaceShader) >0:
            selection = currentSelection.remove(str(surfaceShader[0]))
            print '>>> selection = ', selection
        else:
            selection = currentSelection
        mc.sets(selection, name=selectionSetName, facets=True)
        print 'set ', selectionSetName, 'contains ', selection

    def selectComponents(self, *args):
        ### this function selects components based on value of check
boxes
        # ######### run through the list of check boxes
        for CB in self.selectionSetCBList:
            print CB
            currentSelection = mc.checkBox(CB, query=True, label=True)
            CBvalue = mc.checkBox(CB, query=True, value=True)
            if CBvalue == 1:

                # print 'for CB {CB} currentSelection is
{currentSelection}'.format(CB=CB, currentSelection=currentSelection)
                mc.select(currentSelection, add=True)
            #if the CB is off deselect????
            if CBvalue == 0:
                mc.select(currentSelection, deselect=True)


    def resetSwatch(self, *args):
        # ## reset the current palette swatch to grey
        currentcell = mc.palettePort(self.colourPalette, setCurCell=True,
query=True)
        print 'currentcell is ', currentcell
        ### resetCurrentColour
        mc.palettePort(self.colourPalette, edit=True, \
            rgbValue=[currentcell, self.defaultColour[0],
self.defaultColour[1], self.defaultColour[2]])
        mc.palettePort(self.colourPalette, edit=True, redraw=True )

        ###execute makeOrEditShader
        self.makeOrEditShader()

    def loadSwatches(self, *args):
        # this one added July 2015
        # make a dictionary set index number as key and colour as value
        shaderDict = {}

        surfaceShaderList = mc.ls(exactType = "surfaceShader")
        for shader in surfaceShaderList:
            if re.search("shaderCell_", shader):
                # get the colour and the index number
                cellIndex = re.findall(r'\d+', shader)[0]
                cellColour =
mc.getAttr("{shader}.outColor".format(shader=shader))[0]
                shaderDict[cellIndex] = cellColour


        lambertShaderList = mc.ls(exactType = "lambert")
        for shader in lambertShaderList:
            if re.search("shaderCell_", shader):
                cellIndex = re.findall(r'\d+', shader)[0]
                cellColour =
mc.getAttr("{shader}.color".format(shader=shader))[0]
                shaderDict[cellIndex] = cellColour
        blinnShaderList = mc.ls(exactType = "blinn")
        for shader in blinnShaderList:
            if re.search("shaderCell_", shader):
                cellIndex = re.findall(r'\d+', shader)[0]
                cellColour =
mc.getAttr("{shader}.color".format(shader=shader))[0]
                shaderDict[cellIndex] = cellColour

        for key in shaderDict:
            cell = int(key)
            colour = shaderDict[key]
            # set the colour pallete
            mc.palettePort(self.colourPalette, edit=True, \
                rgbValue=[cell, colour[0], colour[1], colour[2]])
            mc.palettePort(self.colourPalette, edit=True, redraw=True )


    def grabColour(self, *args):
        chosenColour = mc.grabColor()
        currentcell = mc.palettePort(self.colourPalette, setCurCell=True,
query=True)
        print 'currentcell is ', currentcell
        ### resetCurrentColour
        mc.palettePort(self.colourPalette, edit=True, \
            rgbValue=[currentcell, chosenColour[0], chosenColour[1],
chosenColour[2]])
        mc.palettePort(self.colourPalette, edit=True, redraw=True )

        ###execute makeShader here too
        self.makeOrEditShader()

    def assignColour(self, *args):
########### 02_09_14 update for lambert and blinn
        # current selection is geo or faces ###
        currentSelection = mc.ls(selection=True, flatten=True)
        print '>>> assignColour >>>> currentSelection = ',
currentSelection

        ##get the shader name if it is selected
        shaderList = []
        if self.shaderType == 'surfaceShader':
            shaderList = mc.ls(selection=True, type='surfaceShader')
        elif self.shaderType == 'lambert':
            shaderList = mc.ls(selection=True, type='lambert')
        elif self.shaderType == 'blinn':
            shaderList = mc.ls(selection=True, type='blinn')

        #get the selection without the surface shader
        if len(shaderList) >0:
            selection = currentSelection.remove(str(shaderList[0]))
            print '>>> surfaceShader = ', shaderList[0]
        else:
            selection = currentSelection
        print 'selection = ', selection

        ###get the current pallette index
        currentcell = mc.palettePort(self.colourPalette, setCurCell=True,
query=True)
        correspondingCell = currentcell
        ### get corresponding shader ################# just call shader
simple name
        # if self.shaderType == 'surfaceShader':
        #     shaderName =
'SScell_{correspondingCell}'.format(correspondingCell=correspondingCell)
        # elif self.shaderType == 'lambert':
        #     shaderName =
'Lcell_{correspondingCell}'.format(correspondingCell=correspondingCell)
        # elif self.shaderType == 'blinn':
        #     shaderName =
'Bcell_{correspondingCell}'.format(correspondingCell=correspondingCell)
        shaderName =
'shaderCell_{correspondingCell}'.format(correspondingCell=correspondingCe
ll)

        # get shading group from shader
        shadingGroup = getSGfromMaterial(shaderName)
        # print 'shadingGroup is ', shadingGroup

        ### assign corresponding material to selectionSet
        mc.sets(selection, edit=True, forceElement=shadingGroup)


    def makeOrEditShader(self, *args):
         # makes a surface shader, lambert or blinn
        #when a cell swatch is edited for the first time make a
corresponding  shader
        #### when the cell is edited for a second time just tweak the
shader
        currentcell = mc.palettePort(self.colourPalette, setCurCell=True,
query=True)
        correspondingCell = str(currentcell)

        ######## make shader one name regardless of type
        shaderName =
'shaderCell_{correspondingCell}'.format(correspondingCell=correspondingCe
ll)

        ### make shading group
        shadingGroupName = '{shaderName}SG'.format(shaderName=shaderName)
        rampName =
'rampCell_{correspondingCell}'.format(correspondingCell=correspondingCell
)

        #check if the corresponding shader exists
        shaderExistence = mc.objExists(shaderName)
        # print 'shaderExistence == ', shaderExistence
        if shaderExistence == 0:

            ## make shading group
            shadingGroup = mc.sets(renderable=True, noSurfaceShader=True,
empty=True, name=shadingGroupName)

            if self.shaderType == 'surface':
                #make the surface shader
                surfaceShader = mc.shadingNode('surfaceShader', 
asShader=True, name=shaderName)
            if self.shaderType == 'lambert':
                #make the lambert shader
                lambertShader = mc.shadingNode('lambert', asShader=True, 
name=shaderName)
            if self.shaderType == 'blinn':
                #make the lambert shader
                blinnShader = mc.shadingNode('blinn', asShader=True, 
name=shaderName)

            # ## Connect the material to the Shading Group.
            
mc.connectAttr("{shaderName}.outColor".format(shaderName=shaderName), \
            
"{shadingGroup}.surfaceShader".format(shadingGroup=shadingGroup), 
force=True)


            #get colour from current (corresponding) cell
            currentColour = mc.palettePort( self.colourPalette, 
query=True, rgb=True )
            # print 'currentColour is ', currentColour
            ########### make a ramp with one entry of current colour
            rampNode = mc.shadingNode("ramp",asTexture=True, name = 
rampName)
            
mc.setAttr('{rampNode}.colorEntryList[0].color'.format(rampNode=rampNode)
, \
                currentColour[0],currentColour[1],currentColour[2])

            ######### set the interp type to none
            
mc.setAttr('{rampNode}.interpolation'.format(rampNode=rampNode), 0)

            if self.shaderType == 'surface':
    ########### connect ramp colour to SS colour
                    
mc.connectAttr('{rampName}.outColor'.format(rampName=rampName),'{surfaceS
hader}.outColor'\
                    .format(surfaceShader=surfaceShader), force=True)
            elif self.shaderType == 'lambert':
    ########### connect ramp colour to lambert colour
                    
mc.connectAttr('{rampName}.outColor'.format(rampName=rampName),'{lambertS
hader}.color'\
                    .format(lambertShader=lambertShader), force=True)
            elif self.shaderType == 'blinn':
    ########### connect ramp colour to blinn colour
                    
mc.connectAttr('{rampName}.outColor'.format(rampName=rampName),'{blinnSha
der}.color'\
                    .format(blinnShader=blinnShader), force=True)

        if shaderExistence == 1:
            #get colour from current (corresponding) cell
            currentColour = mc.palettePort( self.colourPalette, 
query=True, rgb=True )
```

```
            print 'currentColour is ', currentColour

            if self.recordColourTweakStatus == True:
                #### make another colour swatch
                self.addColorEntry(rampName, currentColour)

            elif self.recordColourTweakStatus == False:
                ############# set ramp outColour to current colour

mc.setAttr('{rampName}.colorEntryList[0].color'.format(rampName=rampName)
, \
                    currentColour[0],currentColour[1],currentColour[2],
type='double3' )


    def addColorEntry(self, rampName, currentColour):
        # print '>>> adding a colorEntryList of ', currentColour, ' to
ramp ', rampName
        currentColourTuple = (currentColour[0], currentColour[1],
currentColour[2])
        ### get the number of colorEntryList entries in the ramp
        numEntries =
mc.getAttr('{rampName}.colorEntryList'.format(rampName=rampName),
size=True)
        endPos = numEntries * 0.001
        listIndex = numEntries -1


        ### test last entry against current colour
        latestEntryColourList =
mc.getAttr('{rampName}.colorEntryList[{i}].color'.format(rampName=rampNam
e, i=listIndex))
        latestEntryColour = latestEntryColourList[0]
        print 'latestEntryColour is ', latestEntryColour,
'currentColourTuple is ', currentColourTuple

        if latestEntryColour == currentColourTuple:
            print '>>> alreading added colorEntryList of ',
currentColourTuple, ' to ramp ', rampName
        else:
            print '>>> adding colorEntryList of ', currentColourTuple, '
to ramp ', rampName
            #### add colour entry list at endPos

mc.setAttr('{rampName}.colorEntryList[{i}].position'.format(rampName=ramp
Name, i=(listIndex+1)), endPos)
#### create ramp node
            rampSwatchName =
'tweak{i}_{rampName}'.format(i=(listIndex+1), rampName=rampName)
            rampSwatch = mc.shadingNode("ramp",asTexture=True, name =
rampSwatchName)

mc.setAttr('{rampSwatch}.colorEntryList[0].color'.format(rampSwatch=rampS
watch), \
                currentColour[0],currentColour[1],currentColour[2])


            ########### connect ramp colour to main ramp colour
```

```
mc.connectAttr('{rampSwatch}.outColor'.format(rampSwatch=rampSwatch),\

'{rampName}.colorEntryList[{i}].color'.format(rampName=rampName,
i=(listIndex+1)), force=True)
```