PYTHON SCRIPTING FOR SMART AND CURIOUS COMPOSITORS

LESSON 2

Gianluca Dentici
www.gianlucadentici.com

LESSON 2

Heya welcome back to this crazy Python for smart and curious compositors ! yay !
In this lesson we will push ahead with the scripting !!! but first of all an important curiosity. In italian Python is “Pitone” and it is not only the name of the infamous snake ! In Sicily this is the name they use to call a special quiche filled with vegetables, mozzarella and tomato. Enjoy the pictures ! And if you want to download the recipe, just google “Pitoni Messinesi”.

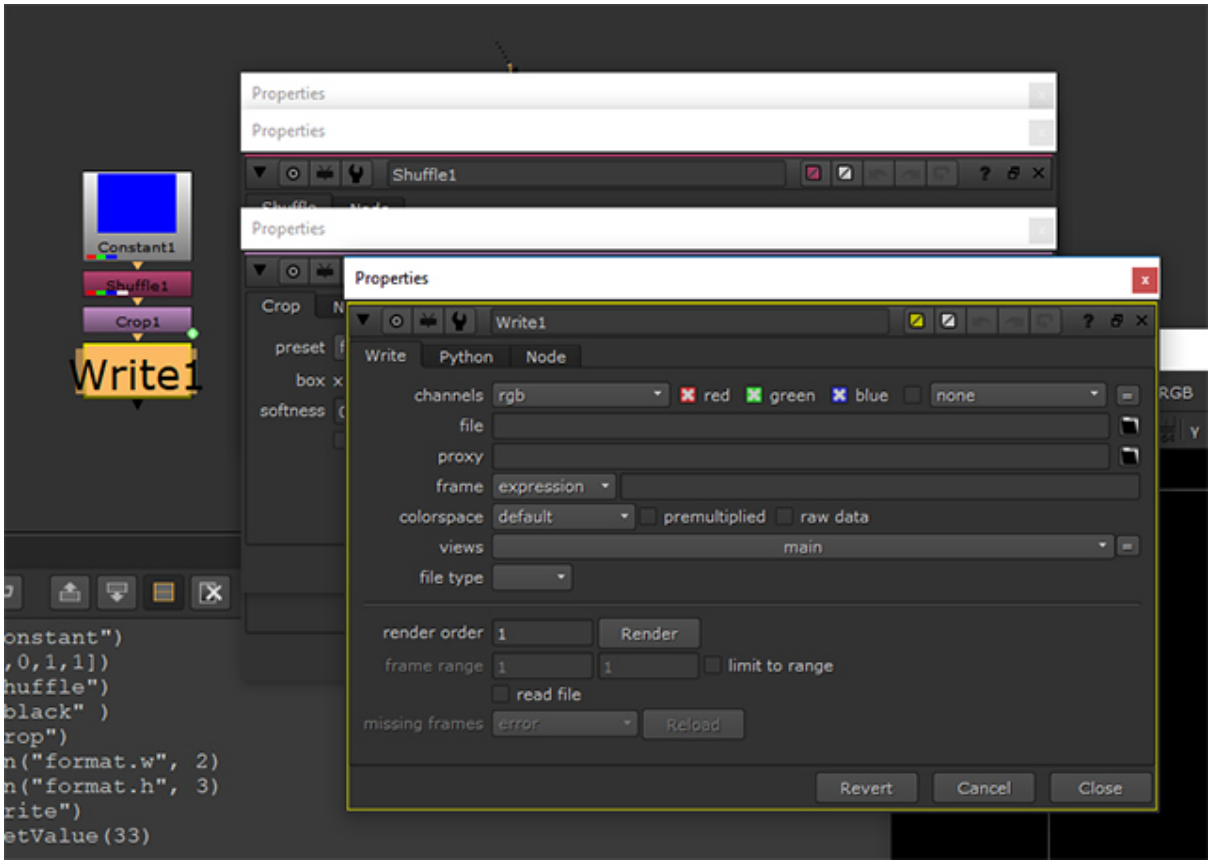
Ok now that you all know one more reason for an holiday to Italy, let’s take a look at the homework solutions from lesson 1.

The challenge was:
“Create a Constant blue color, connect it to a Shuffle with the alpha set to black, then attach its output to a Crop format set to your current script resolution and a Write node with its font value set to 33”.

And here you are the solution:

```
Cs = nuke.createNode("Constant")
Cs['color'].setValue([0,0,1,1])
sh = nuke.createNode("Shuffle")
sh['alpha'].setValue( "black" )
cro= nuke.createNode("Crop")
cro['box'].setExpression("format.w", 2)
cro['box'].setExpression("format.h", 3)
out= nuke.createNode("Write")
out['note_font_size'].setValue(33)
```

so your result should look like this:



You may have notice how boring is when attribute windows pop up as you create your nodes. But we are lucky enough to fix it by using the command inpanel=False. All you need to do is adding this command into the nodes‘ creation construct:

That is:

```
Cs = nuke.createNode("Constant", inpanel=False)
Cs['color'].setValue([0,0,1,1])
sh = nuke.createNode("Shuffle", inpanel=False)
sh['alpha'].setValue( "black" )
cro= nuke.createNode("Crop", inpanel=False)
cro['box'].setExpression("format.w", 2)
cro['box'].setExpression("format.h", 3)
out= nuke.createNode("Write", inpanel=False)
out['note_font_size'].setValue(33)
```

So useful isn’t it? Obviously if we want let’s say the Write node window kept open, just write True over its line.

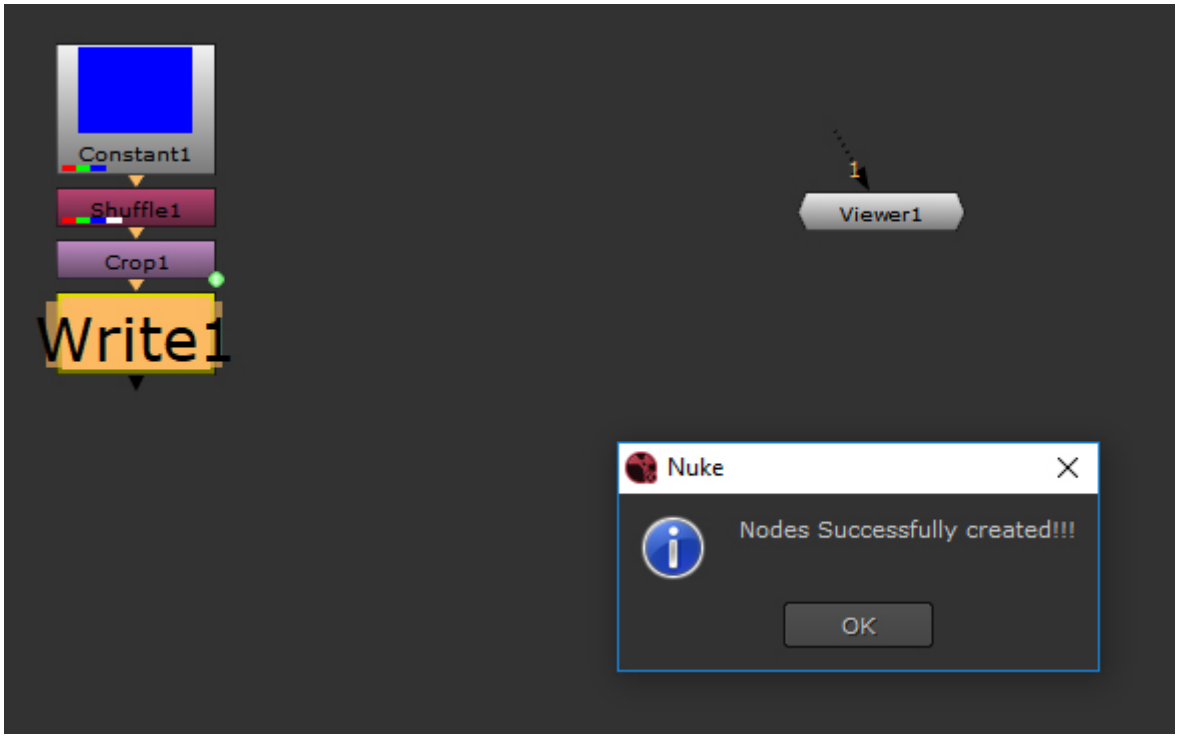
Hey at this point why not warning the user that he has just created all those nodes successfully by popping up a lovely message window ?

Can’t be much easier! Just add this line at the end of your snippet:

nuke.message(“Nodes successfully created!!!”)

So your final script would look like:

```
Cs = nuke.createNode("Constant", inpanel=False)
Cs['color'].setValue([0,0,1,1])
sh = nuke.createNode("Shuffle", inpanel=False)
sh['alpha'].setValue( "black" )
cro= nuke.createNode("Crop", inpanel=False)
cro['box'].setExpression("format.w", 2)
cro['box'].setExpression("format.h", 3)
out= nuke.createNode("Write", inpanel=False)
out['note_font_size'].setValue(33)
nuke.message('Nodes Successfully created!!!')
```



This is really cool isn’t it ? so basically by using **nuke.message()** you could create any popping message you like before or after your operations. I know, I know the temptation to pop up with silly sentences is irrepressible ;)

However on future lessons we will learn how to get values and other interesting things from your comp scripts and append it to the main message !!! ;)

Now last time we said not to worry with the file path into the Write node, but now it has come

the time to take care of it. Let's put it one manually. In order to do it we juśt need to add this line:

nuke.toNode('Out').knob('file').setValue('c:/hello.tif')

And thus the entire script could be:

```
Cs = nuke.createNode("Constant", inpanel=False)
Cs['color'].setValue([0,0,1,1])
sh = nuke.createNode("Shuffle", inpanel=False)
sh['alpha'].setValue( "black" )
cro= nuke.createNode("Crop", inpanel=False)
cro['box'].setExpression("format.w", 2)
cro['box'].setExpression("format.h", 3)
out= nuke.createNode("Write", "name Out", inpanel=False)  # I put a name to the write node here
out['note_font_size'].setValue(33)
nuke.toNode('Out').knob('file').setValue('c:/hello.tif')
nuke.message('Nodes Successfully created!!!')
```

As you can see I've just changed the code a little bit by naming the write node as "**Out**", this allows me to call it back with that specific name over the very last line of the snippetand set the path to the file.
On this occasion we just put the file path manually, but very soon we will learn how to automate this process, possibly linking it to a pipeline framework.

One last thing! If you like you could also type the second-to-last line this way:
nuke.toNode('Out')['file'].setValue('c:/hello.tif')
Which is the slightly different construct we already talked about some time ago.

Now can you remember when during our first lesson we learned how to set up an expression for specific parameters ? We animated a Sphere..how exciting! But we haven't studied how to set up simple keys! Like user's keys to parameters. Let's go through it now!

Let's assume to create a Blur node and setting up its value to 5 with a key. There is a special command that tells Nuke to set a key and then assigning it to a specific value; this command is '**setAnimated**' , so the script would look like this:

```
node=nuke.createNode('Blur')
k=node['size']
k.setAnimated()
k.setValue(5)
```

Easy peasy ! But now let's take a look at the following:

node=nuke.createNode('Blur')

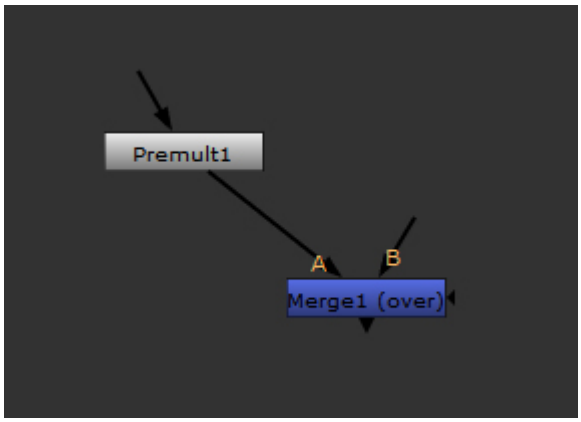
```
k=node['size']
k.setAnimated()
k.setValueAt(55,10,0)
k.setValueAt(66,20,1)
```

At the very first sight you see 2 similar lines where we are actually setting up values, it's because this time I decided to assign different values for the Blur on X and Y ! Yet you noticed those 3 digits between parenthesis... so what do those numbers stand for ?
well, the first number is the value you are going to assign your blur filter to (55 for the first line), the second is the frame in your timeline where you are actually setting up your key (10 for the first line), whilst the third is telling nuke which knob you are acting on with your Blur, hence **0** for the blur on **X** and **1** for **Y**. Once executed take a look at your timeline and you will see your brand new lovely keys !
Really cool isn't it ?

Now we move on, changing the topic and let's go through the build of a proper nodes' network on different methods.
Up to now we have just created nodes linked one after the other but how about connecting them differently ? Let's assume we create a Merge node and then other kind of nodes and then we want them linked to Merge's inputs. How to? and how about connecting a mask as well? Understanding these procedures is really important because they turn out very useful especially when we will go through the loops and functions! So by that time you could be skilled enough to cook up a compositing templates for AOVs for instance.
So let's kick it off! Just don't forget to clear up your nuke script first. Besides, for the sake of clarity, I would recommend to delete all your nodes any time before testing snippets we will be covering from now on.

I would start by creating a Premult node linked to the Merge's A slot.

```
p = nuke.createNode('Premult', inpanel=False)
mergeok=nuke.createNode('Merge2', inpanel=False)
mergeok.setInput(0,None)
mergeok.setInput(1,p)
```



You should be already familiar with the first two lines, those new are the subsequent.
The Merge node has been created and assigned to a new variable **mergeok** and then we are using the new construct **setInput** to set up our Merge's inputs. Now what the heck those parameters between the parenthesis on 3rd and 4th lines stand for?

So the **zero** refers to Merge's **B** slot, whilst **None** means that we want to leave this slot disconnected.
We decided to link the Multiply output with the Merge's slot **A** instead, therefore the parenthesis of the next line will have a number **1** inbetween and the **p**, who defines the node we are connecting to, which is the Premult.
For your convenience find here all inputs correspondings:

0 represent the B input
1 is A
2 is the mask input
From 3 onwards we are talking about merging other inputs like, you know, a2, a3, etc

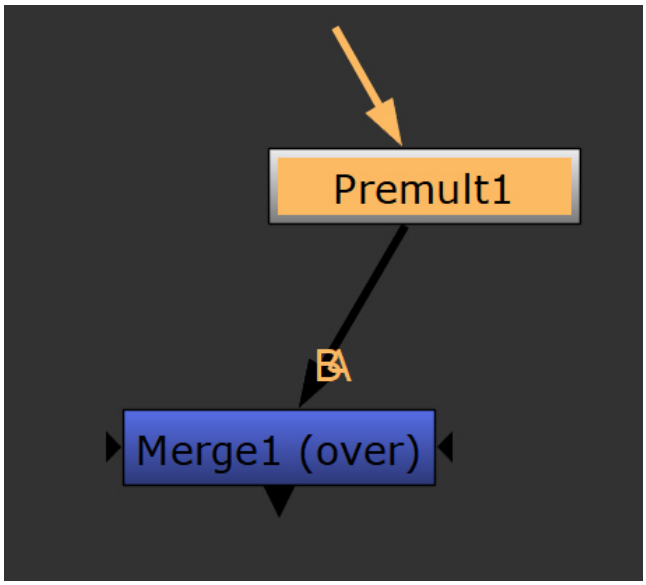
Be aware! This numeration works for those nodes accepting multiple inputs, whilst for all other nodes:

0 is the main input
1 is the mask input

For multiple inputs' nodes I always strongly recommend to declare the status for both of them, the reason is that even though declaring just one might look correct syntax-wise, it could return to unexpected results. As an example with that let's say we want to feed the slot B with our Multiply, so we might simply think of writing:

```
p = nuke.createNode('Premult', inpanel=False)
mergeok=nuke.createNode('Merge2', inpanel=False)
mergeok.setInput(0,p)
```

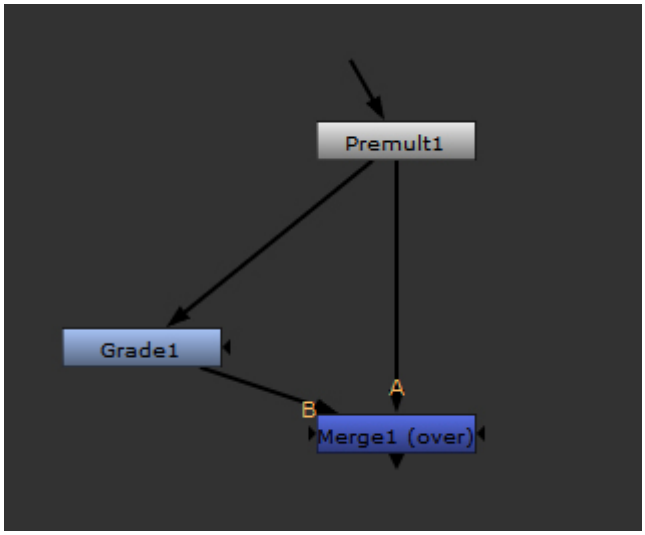
… but if you execute these lines you'll come across a disaster !!!… both the slots A and B will be connecting to the Premult node !! fuckin' hell !



So this is the reason for the nodes' input status being declared properly. At least this is my personal approach that I believe is quite safe.

Now let's create and link a Premult node to the Merge node's A slot, then right after the Premult let's stick a Grade, this one linked to the B slot.

```
p = nuke.createNode('Premult', inpanel=False)
r = nuke.createNode('Grade', inpanel=False)
mergeok=nuke.createNode('Merge2', inpanel=False)
mergeok.setInput(1,p)
mergeok.setInput(0,r)
```



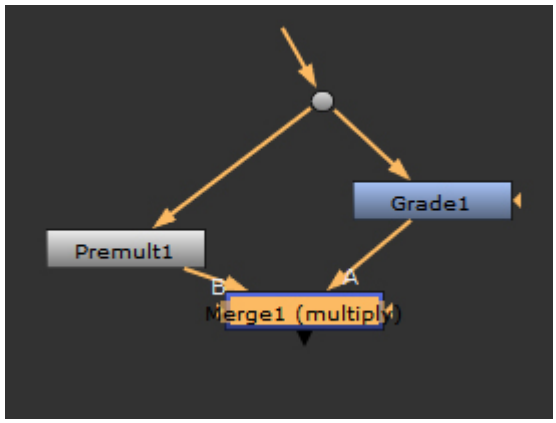
It looks nice too ! Now look how this snippet could also be written:

```
p = nuke.createNode('Premult', inpanel=False)
r = nuke.createNode('Grade', inpanel=False)
oo = nuke.nodes.Merge2(operation='multiply', inputs=[p,r] , output='rgb' )
```

Basically over the last line I've just created a variable who has created the Merge node and then I put further parameters within parenthesis. Among them we find both the inputs **p** and **r** that in fact are our Premult and Grade nodes; this time they sit between square brackets! why ? it means we have just created a "List". A list is just a list of parameters, variables, numbers or whatever you need! We will go through Lists later on the next lessons, but for now let's just accept it.
This kind of writing is really cool and we should get more familiar with it since we can save some space although it may look like a little confusing to someone.

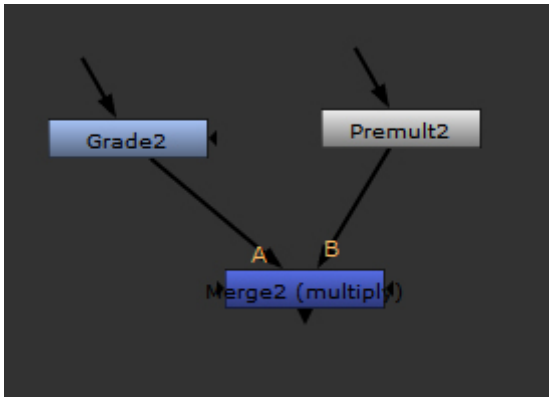
Now let's try to create a Merge node and connect its inputs to our usual Premult and Grade nodes, in their turn both linked to a dot:

```
se=nuke.createNode('Dot')
p = nuke.createNode('Premult', inpanel=False)
r = nuke.createNode('Grade', inpanel=False)
oo = nuke.nodes.Merge2( operation='multiply', inputs=[p,r] , output='rgb' )
r.setInput(0,se)
```



The following leave both the Premult and Grade nodes disconnected on their inputs but stick their outputs to the Merge node.

```
p = nuke.createNode('Premult', inpanel=False)
r = nuke.createNode('Grade', inpanel=False)
oo = nuke.nodes.Merge2( operation='multiply', inputs=[p,r] , output='rgb' )
r.setInput(0,None)
p.setInput(0,None)
```



...but we could also code it in a different way:

```
p=nuke.createNode('Premult', inpanel=False)
r=nuke.createNode('Grade', inpanel=False)
r.setInput(0,None)
m = nuke.nodes.Merge2(inputs=[p, r])
```

So what's the difference ? The point is that after creating the Premult node the following Grade will be connected to it by default, so down in the stream after the Grade creation we can order him to disconnect from the Premult, and in fact this is what we are doing on the

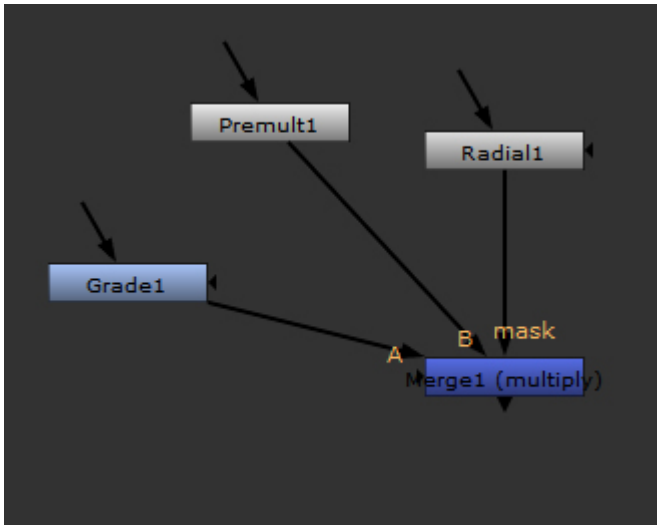
third line.
Then the script builds the Merge node and sets its inputs.
We could also shortening our snippet a little by fitting all Grade parameters on the same row, we just have to split them with a semicolon:

```
p=nuke.createNode('Premult', inpanel=False)
r=nuke.createNode('Grade', inpanel=False); r.setInput(0,None)
m = nuke.nodes.Merge2(inputs=[p,r])
```

Looks pretty doesn't it ?? So we learned how to shrink a bit our lines. Not vital, but it is good to know.

So let's carry on with our beloved Premult, Grade and Merge nodes by building a script that creates and connects a **Radial** straight to the Merge's mask slot.

```
p = nuke.createNode('Premult', inpanel=False); p.setInput(0,None)
r = nuke.createNode('Grade', inpanel=False); r.setInput(0,None)
ra = nuke.createNode('Radial', inpanel=False); ra.setInput(0,None)
oo = nuke.nodes.Merge2(operation='multiply', inputs=[p,r,ra] , output='rgb' )
```



As you see I worked it out by using the short scripting, so it is quite similar to the previous snippet actually, I've just added the variable **ra** as the third element of the list, that means into the Merge's slot numbered as 2…. You remember that python starts from zero… so for the list [p,r,ra] the inputs are 0,1,2

Let's move on even further by creating the same nodes and let's see how to add any other parameters on the same line, this time for the Merge node:

```
p = nuke.createNode('Premult', inpanel=False); p.setInput(0,None)
r = nuke.createNode('Grade', inpanel=False); r.setInput(0,None)
ra = nuke.createNode('Radial', inpanel=False); ra.setInput(0,None)
```

```
oo = nuke.nodes.Merge2(inputs=[p,r,ra] , output='rgb' );oo['invert_mask'].setValue(True)
```

Should be looking quite easy to understand now that we are more familiar with some syntax!
So now why don't we zoom over the last created node ?

```
p = nuke.createNode('Premult', inpanel=False); p.setInput(0,None)
r = nuke.createNode('Grade', inpanel=False); r.setInput(0,None)
ra = nuke.createNode('Radial', inpanel=False); ra.setInput(0,None)
oo = nuke.nodes.Merge2(inputs=[p,r,ra] , output='rgb' );oo['invert_mask'].setValue(True)
oo['selected'].setValue(True) # why not select the node here ?
nuke.zoom( 3, [ oo.xpos(), oo.ypos()])
```

It is easy like that! You just need to use the command **nuke.zoom**, whereas the other parameters are: **3** is a specific parameter that is telling nuke to zoom over the upper left corner of the selected node - by the way I'm sure you wouldn't miss your node :-) - yet, if for instance you write 1 it will broaden your view by framing the entire script.
The other **oo.xpos** and **oo.ypos** constructs are quite easy to understand, they just point to the node's position into the script.

One thing you might have noticed is the nodes we have been creating are quite close to the others, we will learn how to spread them around conveniently very soon!

So now before moving any further with programming techniques, let's go over an important side topic. We have been writing and executed our code straight into the script editor so far, but now let's assume we want to create a proper menu sitting somewhere on our nuke interface and fit one of our lovely executable python scripts into it, ready to act on our comp scripts.
In order to achieve that we must write a text file which include our lines of code first and then save it with the standard python file extension .py, like test.py.
Besides It is crucial to save it into the right folder that already contains other nuke's important files, this is usually called **.nuke**
Yet we have to edit the menu.py file that includes all nuke important start up tools, like plugins or gizmos and since we want our menu and tools to work from the moment nuke starts we have to add a line into this menu.py.
Now the call will look different and depends on the menu and tool appearance and the desired location over the nuke interface.
So now if you are already familiar to this process you could make a leap to the next topic to page 15.

Ok if you are reading this it means you decided to stay with us for this part ! :)
For instance let's add a menu on the menu bar, top of the nuke's interface - just add this line into your menu.py file:

```
nuke.menu( 'Nuke' ).addCommand( 'Mymenu/test', lambda: nuke.createNode('test' ),'o')
```

This construct is quite easy to understand, it uses **addCommand** to tell Nuke building a new command. Then you have to specify its name and you can see there are 2 names splitted by a slash.

The first is the name you see appearing onto your menu bar (Mymenu), whilst the second is the name of the python file that nuke will be executing on a call.
Then "**lambda**" is just a kind of null variable that is calling the python file. Then that '**o**' is the shortcut we wanted to add to this command. Nice uh ?

The shortcut can also be mapped to a CTRL+something or ALT etc.
nuke.menu('Nuke').addCommand('Mymenu/test', lambda: nuke.createNode('test'),'CTRL+o')

If you like to add your tool to a specific existent menu into the left side toolbar, just add this line:
nuke.menu('Nodes').addCommand('Image/test', lambda: nuke.createNode('test') ,'CTRL+o')

So by executing It will create a tool into the first menu of the toolbar, that is **Image**.

Now let's see how to add a custom button to your toolbar, with an image and a shortcut.
nuke.menu('Nodes').addCommand('test', lambda: nuke.createNode('test') ,'CTRL+o', icon='image.png')
And now we simply put the tool within a menu of the toolbar called mytool

nuke.menu('Nodes').addCommand('mytool/test', lambda: nuke.createNode('test') ,'CTRL+o', icon='image.png')
This time a simple icon with the Foundry logo will appear at the bottom of all standard buttons.

Looks quite easy isn't it?

Now the following lines create a menu called **Ciao** and then the second line links it to our tool, naming it "**test**":

Don't forget the icon you want to use shouldn't be bigger than 24x24 pixels to fit properly into the toolbar.

But there is more we can do by writing lines into the menu.py file!
What if we want to change Nuke standard nodes' default shortcuts or values?
For instance let's say we want to replace the shortcut for the Reformat node. The code to append into the menu.py file would look like this:

```
toolbar=nuke.menu('Nodes')
toolbar.addCommand('Transform/Reformat','nuke.createNode("Reformat")','ctrl+r')
```

Basically it is going to replace what was before, so from now on when we start nuke the reformat node could be created by pressing ctrl+r

Now somthing different! here the goal is changing the default value for a Blur node:

```
nuke.knobDefault("Blur.size", "30")
```

It is quite obvious that we have changed the default blur size parameter to 30. This kind of construct works the same way for all nuke's nodes, so basically the construct is the node's name followed by the parameter to edit and then the desired value.

We could go even further and let's set up the Framehold node to reflect the current frame on the

pipeline. Then we could also change its look. We could write something like this:

```
nuke.knobDefault( 'FrameHold.tile_color', '230') # this changes the tile color to a specific value
nuke.knobDefault( 'FrameHold.name', 'framehold at')
nuke.knobDefault('FrameHold.note_font_size' , '15')
nuke.menu('Nodes').addCommand( "Time/FrameHold", "nuke.createNode('FrameHold')
['first_frame'].setValue(nuke.frame())", icon='FrameHold.png')
```

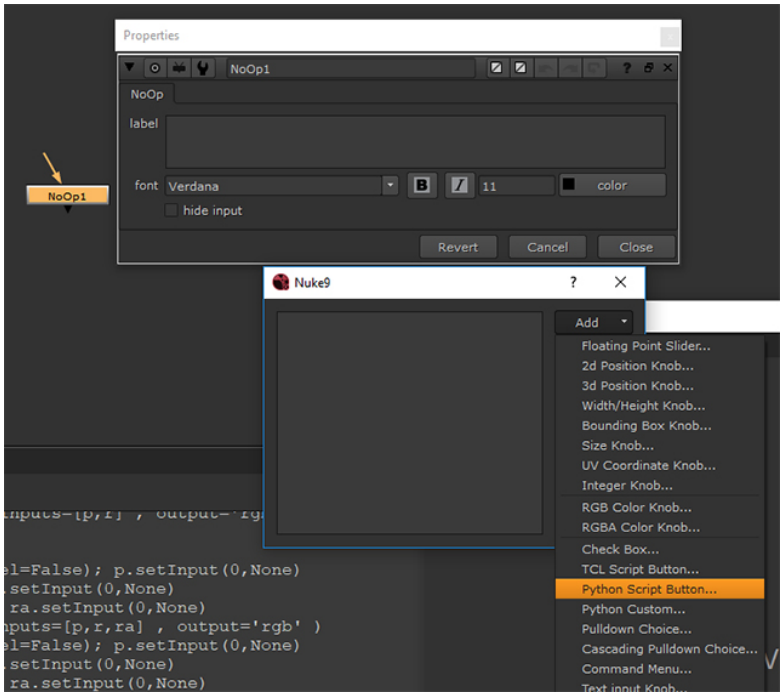
Warning! The last line of the code has to be typed on a single line.

As you may see over the last line we used a new construct **nuke.frame()**, well it means that the command **setValue** actually will set the Framehold's first frame value to the one it gets from the current frame of the timeline.
Awesome !

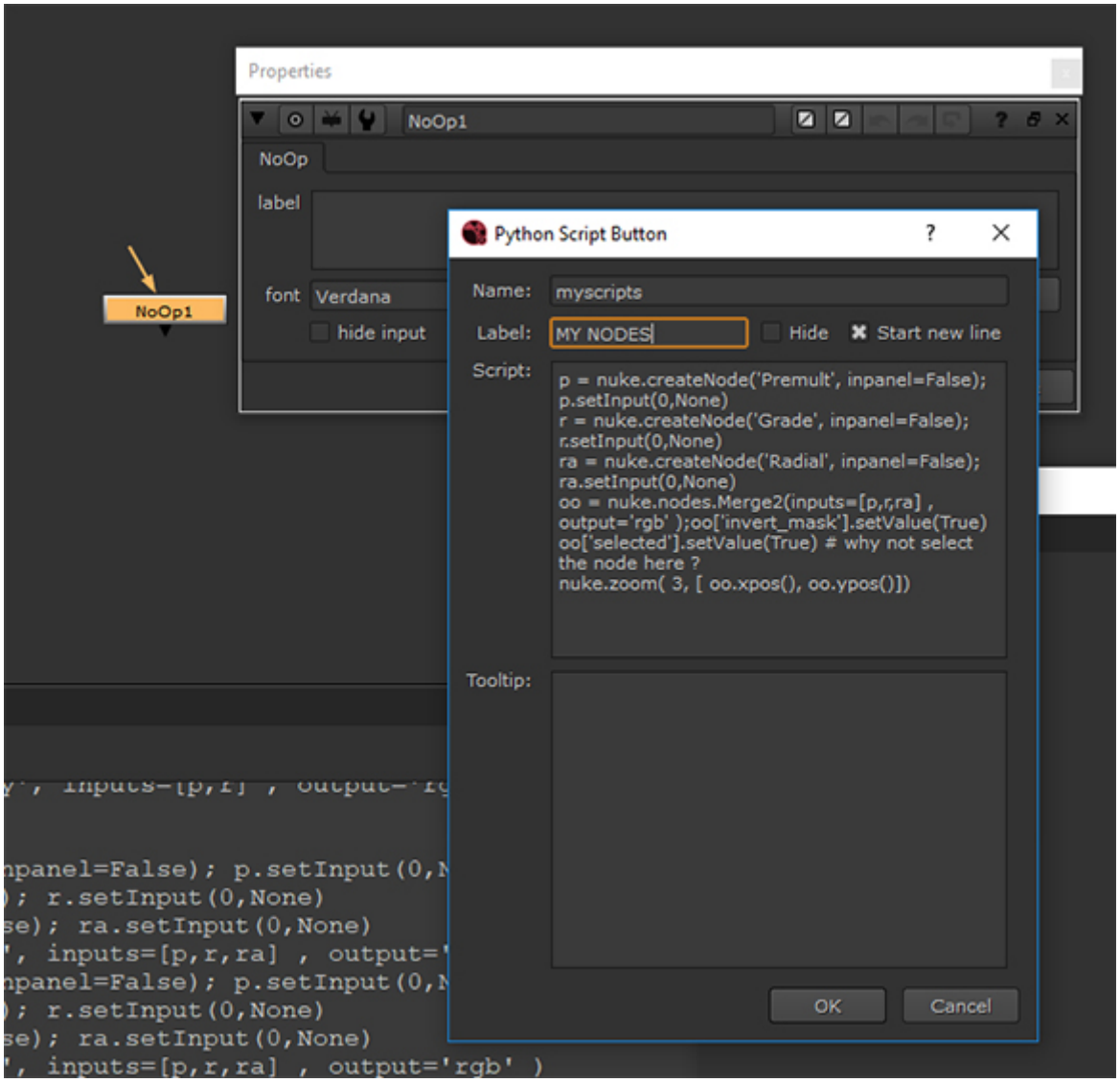
This was about putting our own tools, shortcuts and default values to the nuke UI.

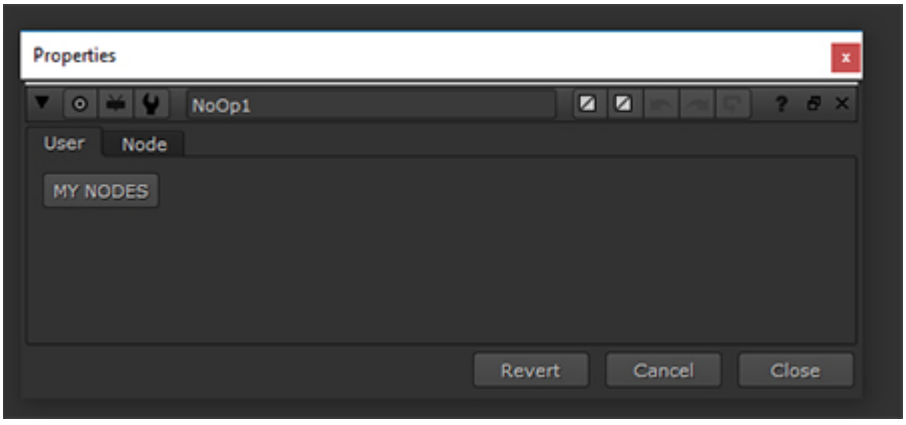
So now let's assume you are not suppose to change the menu.py file because you are working into a very locket off pipeline, hence you can't add your lovely menu and python tools over the Nuke interface (by the way feel free to take it with your supervisors) How can you get away with it?
There is a miserable but efficient way out of this. We could create a NoOp node which includes all the buttons to run our magic scripts!
Many of you are already aware how to do this, but I'll go over it either way.
So let's create a NoOp node and move our pointer down the left bottom of the window and right-click right there, just to the left of the Revert button.

You will be prompted with a list of choices, just click on "manage user knobs" and once the new window pops up hit the 'add' button and choose "Python script button". Basically we are about to create a button that once pressed will execute our python script.



So don’t forget to name the node and place a nice label for your lovely button and now paste your python snippet into the provided space below. Then just confirm and close all windows. Now you have a lovely button ready to execute your script.
Yes I know this is a very primitive and harsh solution but it can really help out to get around any possible locked pipelines.





Besides let's talk about formatting stuff ! Let's assume we want to build our own format and we want it to fit into the Nuke's available formats list. In order to do it we need to add this simple line into our menu.py file:

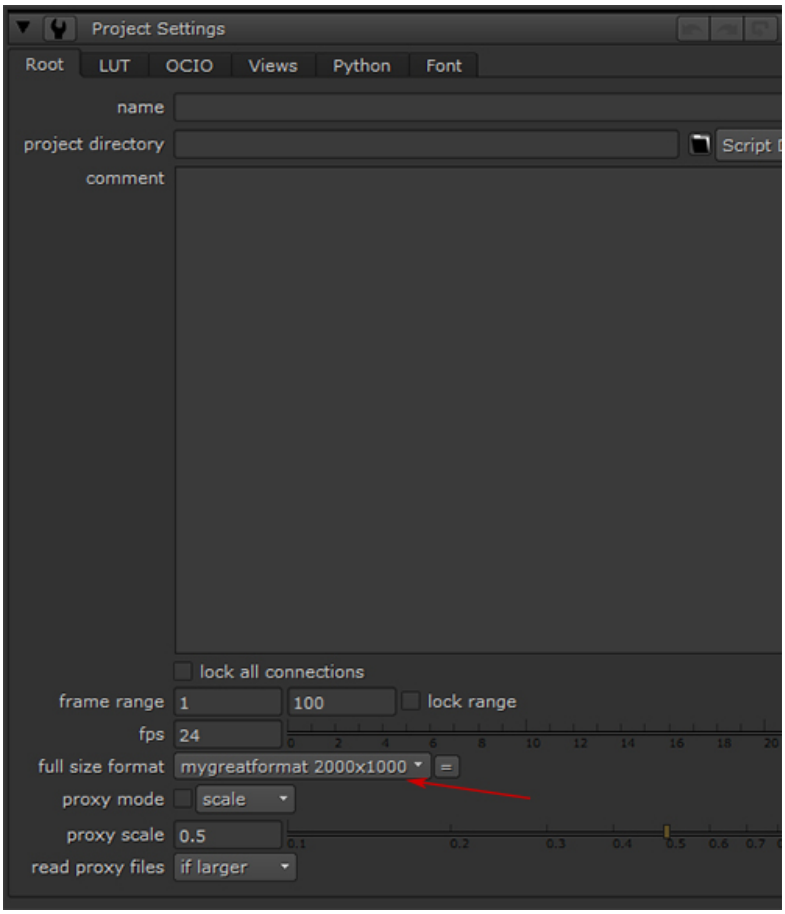
nuke.addFormat('2000 1000 mygreatformat')

It is quite easy to understand actually: **addFormat** is doing what he promises, then we put our desired resolution and then a format name.

At this point we are craving to learn how to start Nuke with our brand new format as default format. So let's do it !

nuke.knobDefault('Root.format', mygreatformat)

Basically **Root.format** is a special useful command that can set the root format (that sits into the project settings windows straight to your new format.



There we go !

And now Ladies and Gentlemen before diving into the foremost and exciting part of this lesson let's take a break with a cup of good porridge and a glass of cherry Dr.Pepper and enjoy these two very traditional music videos from Italy.

https://www.youtube.com/watch?v=2cUzo8BQ3B8

https://www.youtube.com/watch?v=li_FnW4BBIs

We have been writing really straightforward lines of code, maybe even quite boring for the most of you and possibly you won't use them at all for the rest of your life but I reckon that absorbing some good syntax won't hurt at all since it will rest in place throughout the whole nuke scripting experience.
So let's rock and roll by studying scripts to automate processes and we will begin with '**Loops**'.
Now what in the world is a loop?
Creating a script with a loop means that all the procedures and commands will be executed multiple times until all the conditions are respected. For instance I could tell Nuke to perform a specific action all over our script's nodes and only stop when it has been applied on all of them.

Let's load one of our favourite comp scripts, select a few nodes and execute the following:

```
for a in nuke.selectedNodes():
    a['note_font_size'].setValue(60)
```

This 2 lines-long and simple script will change your selected nodes font. Now let's go through it to understand how it works.

Possibly you noticed that the two lines are misaligned or, better saying, the second line has an indentation. This is the main and critical feature for loops. If you don't respect this rule your loop won't work at all or you could get to very unexpected outcomes. So respect the indentation!

In fact the indented lines are actually the body of your loop that starts with '**for**', hence all your lovely lines of code have to respect the indentation. In this first case just 1 space would be enough, but be more generous, I would recommend 2-3 spaces.

Well let's go through the first line. First of all you have noticed the "**for a in**" ; at the very beginning, this statement may sounds tricky, so in order to ease it out I use to read it a

different way into my mind and I start from the end of it.

nuke.selectedNodes(): is the command that is telling nuke that we are going to apply the subsequent action only to currently selected nodes.

So I would read it this way: "*to all my selected nodes, to whom I'm setting up a variable called "a", do....something*".
How it is sounds to you? Does it make any sense at all ?

So now over the second line I processed my new "**a**" variable by changing the font size using the same construct we already seen some time ago.
And it works like a charm !

Now let's make something different:

```
for k in nuke.selectedNodes():
  k['postage_stamp'].setValue(True)
```

This time by executing this code you will be changing the postage stamp status to all your selected nodes.
As a matter of fact you could also write the same code in a slightly different manner:

```
k=nuke.selectedNodes()
for ex in k:
  ex['postage_stamp'].setValue(True)
```

Nothing crazy though, I've just created a new variable called **k** that is now representing **nuke.selectedNodes()**.

Now what if we want to change something over all nodes of our lovely script ? Well this is quite straightforward ! let's grab the previous snippet we used to change the font size over selected nodes and just replace a part of the line to: **nuke.allNodes():**

Now let's move on think about our daily compositing practice, sometimes we need to turn all postage stamps off, so in this case you may want to slightly change the last script we saw to:

```
k=nuke.allNodes()
for ex in k:
  ex['postage_stamp'].setValue(False)
```

This is a very simple code, just 3 lines! but this is something useful you could possibly put into your own menu!

Then how about changing the font size all over script's nodes ? let's see how to:

```
for a in nuke.allNodes():
```

```
    a['note_font_size'].setValue(60)
```

Quite simple, isn't it ? So whether your nodes are selected or not, this code will change your font size all over them. However watch out on doing this on big comp scripts because some of your nodes could actually overlap others. But we will sort this out too very soon!

Well the code we have been looking to is of a great interest because, as we have seen, you could actually use it to create your own buttons to perform anything you like!
Let's extend the previous script a little bit more just to give you an idea of what you can actually do.

```
for a in nuke.selectedNodes():
    a['note_font_size'].setValue(60)
    a['mix'].setValue(0.5)
```

So this one changes the font size and set the mix value of your selected nodes to 0.5.

So now let's say you want to run this code again by trying it on all nodes instead of just those selected; Now the point is that if you have a Viewer in your script this will return to an error or you will get a weird outcome….
To understand the reason for that let's try this out:

```
for a in nuke.allNodes():
  a['note_font_size'].setValue(60)
  a['mix'].setValue(0.5)
```

Now execute it...and…. You see? this is happening simply because your viewer doesn't have a mix knob!!!

There are many different way-out for this, but we will sort this out later on when we will be more familiar with conditions constructs, etc.. In the meantime just watch out with yours "allNodes" snippets or just get rid of your viewer if you want to successfully execute this code. I know that it sounds weird, but we will learn how to get through it very soon.

Now let's move on something very useful: you know when you are working on your lovely script and you get to the point where you opened a lot of tools' attribute panels and they are still crowding your interface ? You could add a nice button on your toolbar with this code within:

```
s = nuke.allNodes()
for i in s :
  i.hideControlPanel()
```

This just closes them up all at once. Fantastic!

Another interesting one might be to select all your script's nodes all at once. Thus is:

```
s = nuke.allNodes()
for i in s :
  i['selected'].setValue(True)
```

that's nice, but what if you want to select just one type of nodes? Let's say Transform nodes ? well there are different ways to do it, but for now let's have a look at the easiest:

```
s = nuke.allNodes('Transform')
for i in s :
   i['selected'].setValue(True)
```

Basically the name between parenthesis is the class of nodes we are looking for.

Another useful script could be one disabling postage stamps all over read nodes. Here you are:

```
s = nuke.allNodes('Read')
for i in s :
  i['postage_stamp'].setValue(False)
```

And now let's extend it even more by turning their colors to green and scaling their fonts up, so they stand out more into the script:

```
s = nuke.allNodes('Read')
for i in s :
  i['postage_stamp'].setValue(False); i['note_font_size'].setValue(22)
  i['tile_color'].setValue(16711935)
```

So cool !

And now...before leave and giving you new homeworks assignments, I just want to pamper you with something else :

So why not writing a snippets that actually pops up with a color picker window and choose our favourite colour for all selected nodes ? this would be much easier than converting from RGB to HEX colors, wouldn't be ? yeah let's do it !

```
clr=nuke.getColor()
for a in nuke.selectedNodes():
    a['tile_color'].setValue(clr)
```

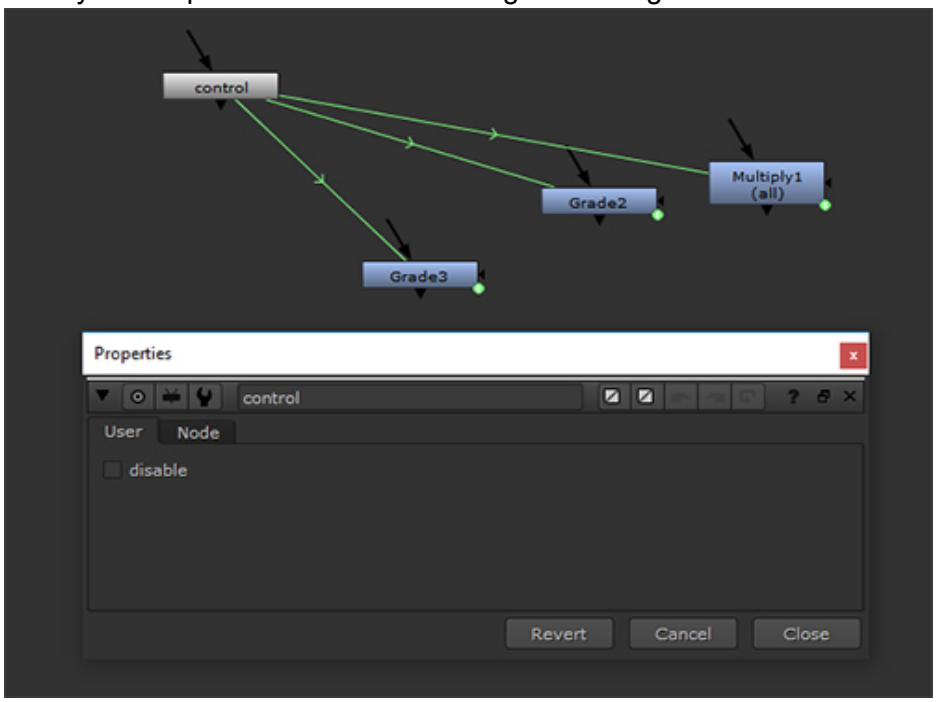
It is simple like that! The construct nuke.getColor() is just doing what we are looking for ! it pops up with a color picker window, so once you choose your favourite color just hit ok and you will see your beloved selected nodes turning into that color !
Gorgeous !

Now it comes the time for your "beloved/hated" assignments ! this time let's move to something a little bit more complicated, but I'm quite sure it won't be like that as you got very skilled with python!!!

HOMEWORKS:
this time 2 assignments ! just watch out with the first one….there is a trap somewhere...

1 - Open up one of your own comp scripts then now in Python: create a **NoOp** node, name it "control", then create a check box knob into it and name it "**Disable**"; then now we want it to drive the "**disable**" knob for our selected nodes!
Tip:You may want to use an expression like "parent.control.disable" to drive your 'enable' knobs.

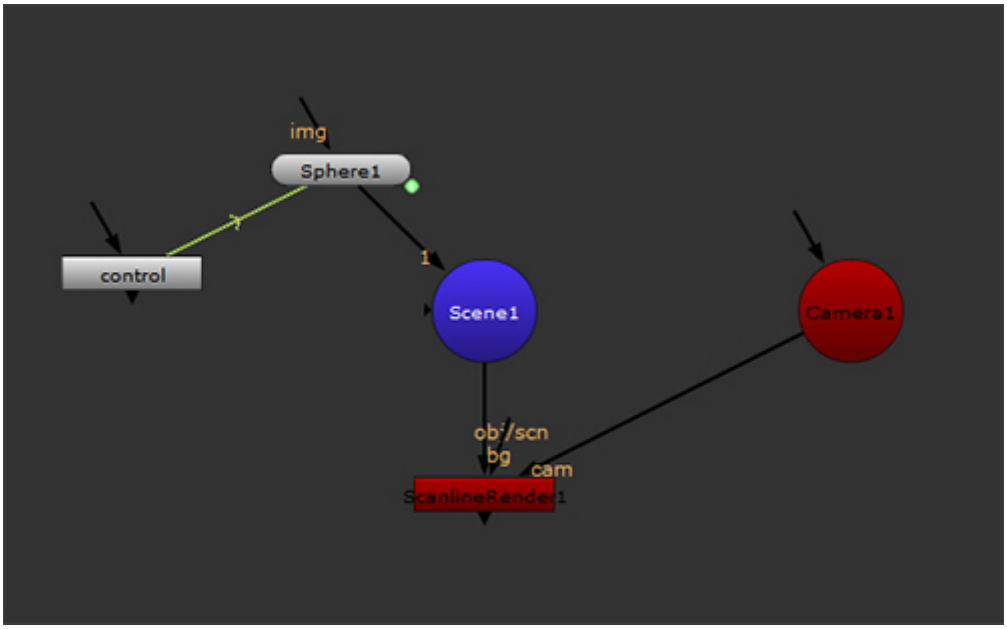
...it depends on your script but it should be looking something like this:



Looking at the picture you can see that the Control node we have to create needs to set up an expressions for the '**disable'** knob of your selected nodes, so once you switch on/off your check box it will act all at once on your selected nodes, by disabling them all together.

2 - Create a Sphere connected to a Scene node - close all the windows with them, then open up a window to set up a different color for the Scene node, let's say a blue.
Then create a NoOp node with an 'array node' (so we can type an integer value within the box) and link it to your Sphere 'uniform scaling' knob, so we can control the sphere scaling from that parameter. You want to have also a Scanline Render and a Camera and connect all these nodes on the proper way.

And it should look be looking like that:



Enjoy and see you soon !