PYTHON SCRIPTING FOR SMART AND CURIOUS COMPOSITORS

Gianluca Dentici
www.gianlucadentici.com

LESSON 3

Hey guys! welcome back to Python for smart and curious compositors ! We got to Lesson 3!

Ok first of all I thought it was important to share with you guys at least one picture of the heaven where I spent part of my lovely 2018 summer holidays !
Many of you were bloody curious and I've been told that someone couldn't literally sleep well without knowing where I was heading to. So now I can finally reveal it ! It's Favignana ! One of the Egadi Highlands ! I strongly recommend it to you all !
If you like the sea this is a must ! indeed ! I'm very opened to give you guys any advices just in case you want to get there ;)

… but don't even think for a moment that I've forgotten you ! In fact I was delighted with the idea that you were doing your Python homeworks under your beach umbrella! Good guys ! aahahahaa :)

So now that I've got my skin tanned I might be ready to go through the
HOMEWORKS CORRECTION !!! yay !!

so the first assignment was:
"Open up one of your scripts then 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!"

And here you are the solution:

```
sel=nuke.selectedNodes()
nn=nuke.createNode("NoOp", "name control")
k=nuke.Boolean_Knob ("disable", "disable")
nn.addKnob(k)
for a in sel:
        a['disable'].setExpression('parent.control.disable') #setting the expression here
nn.setInput(0,None)
```

It is important to create a variable that saves the selected nodes first (first line), in fact if you wrote something like this:

```
nn=nuke.createNode("NoOp", "name control")
k=nuke.Boolean_Knob ("disable", "disable")
nn.addKnob(k)
for a in nuke.selectedNodes():
          a['disable'].setExpression('parent.control.disable')
nn.setInput(0,None)
```

you will find out that it doesn't work !! You see? It is doing something wront and turns with an error in your script editor saying there is no postage_stamp knob!!!
Why that ? Well, you should always imagine Python scripting like the actions you normally do on Nuke, so if you have any nodes selected and then you create another node for instance, the previous selection is gone ! this is exactly what is happening with your python scripting!
Therefore in order to get around this problem just make sure to save your selection first so you can call it back later when you need it, downstream.

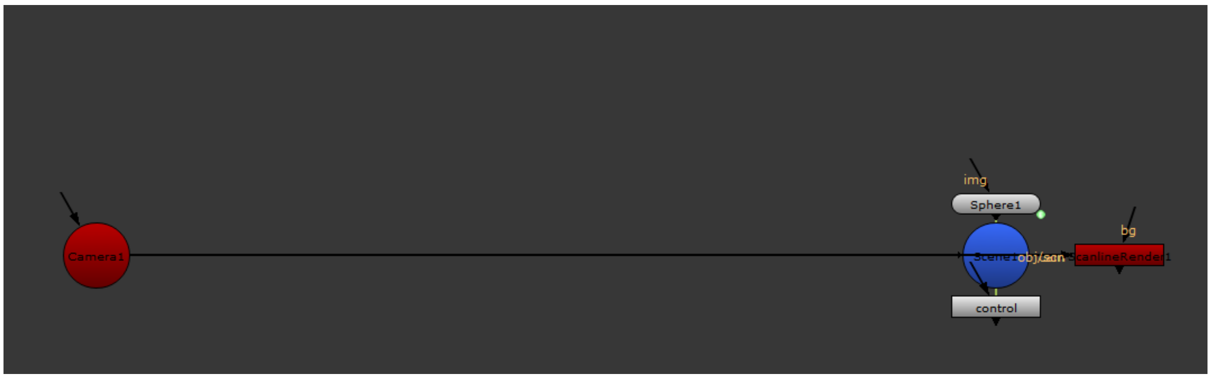
Now the second assignment was:
In Python create a Sphere connected to a Scene node - close all the windows with them, then open up a window to pick a different color for the scene node, let's say a blue.
Then create a NoOp node with an 'array knob' (so we can type an integer value within the box) and link it to your Sphere 'uniform scaling' knob, so we can control the sphere scaling from that parameter.

And the solution is:

```
s1=nuke.createNode('Sphere',inpanel=False);s3=nuke.createNode('Scene', inpanel=False)
s3.setInput(0,s1)
clr=nuke.getColor()
s3['tile_color'].setValue(clr)
bb=nuke.createNode("NoOp", "name control")
k=nuke.Array_Knob ("scaling_factor", "scaling factor")
bb.addKnob(k)
bb['scaling_factor'].setValue(12)
bb.setInput(0,None)
```

```
s1['uniform_scale'].setExpression('parent.control.scaling_factor')# this is the expression that parents the scaling factor
ca=nuke.createNode('Camera', inpanel=False); ca.setInput(0,None)
scan=nuke.createNode('ScanlineRender', inpanel=False); scan.setInput(1,s3)
```

Now you might have noticed that when you execute your lovely code it scatters the nodes far away one with the others, like the following image:



so now let's figure out how to arrange them in a more neat formation; There are more than a few methods to fix this up but let's start with the two easiest:

First of all you could try the command **node.autoplace():** this makes sure that your nodes won't overlap. So just add the following lines at the very end of the script:

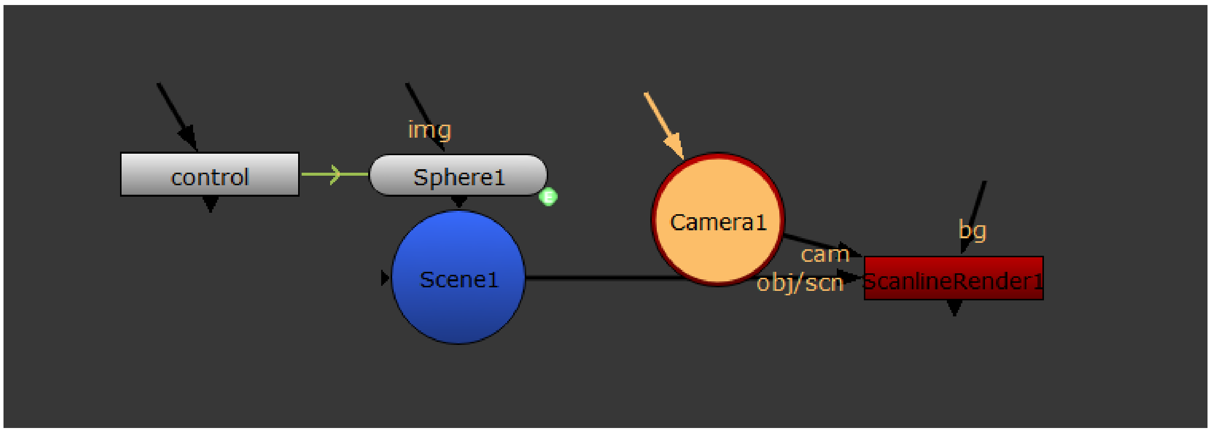
```
for n in nuke.allNodes():
    n.autoplace()
```

Obviously in this case we are invoking the command over all nodes.

And so the script would look like this:

```
s1=nuke.createNode('Sphere',inpanel=False);s3=nuke.createNode('Scene', inpanel=False)
s3.setInput(0,s1)
clr=nuke.getColor()
s3['tile_color'].setValue(clr)
bb=nuke.createNode("NoOp", "name control")
k=nuke.Array_Knob ("scaling_factor", "scaling factor")
bb.addKnob(k)
bb['scaling_factor'].setValue(12)
bb.setInput(0,None)
s1['uniform_scale'].setExpression('parent.control.scaling_factor')# this is the expression that parents the scaling factor
ca=nuke.createNode('Camera', inpanel=False); ca.setInput(0,None)
scan=nuke.createNode('ScanlineRender', inpanel=False); scan.setInput(1,s3)
for n in nuke.allNodes():
    n.autoplace()
```

Right ? So now by executing the code you will notice that not only your nodes won't overlap anymore, but the formation is quite neat actually. Good guy!



Another good method would make use of the **autoplaceSnap** command that actually snaps the nodes to the nearest grid position over the nuke space:

```
for n in nuke.allNodes():
   nuke.autoplaceSnap(n)
```

We also might decide to set precise values for nodes' positions, let's have a look at the following lines:

```
ca=nuke.createNode('Camera', inpanel=False); ca.setInput(0,None); ca.setXYpos( 113, 224 )
ca2=nuke.createNode('Camera', inpanel=False); ca2.setInput(0,None); ca2.setXYpos( 213, 224 )
ca3=nuke.createNode('Camera', inpanel=False); ca3.setInput(0,None); ca3.setXYpos( 313, 224 )
```

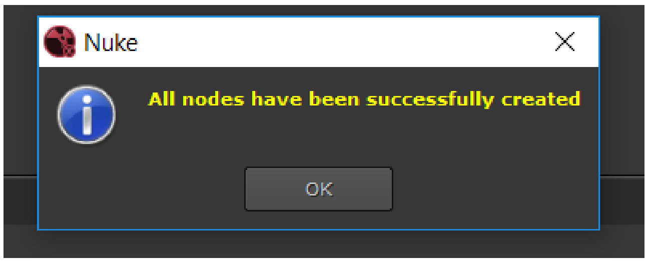
Here we are creating 3 Camera nodes. It is quite obvious that the command **setXYpos** sets the values for both X and Y over specific nodes. As you can see I just set the X position to an incremental 100 pixel for each one, so the nodes will arrange in a neat formation, 100 pixels away one with the other.
Actually there are even more ways to place nodes but we will go through them very soon once we will get more familiar with further syntax.

Now let’s jump onto something else and shall extend our script by adding a pop up window at the very end of it warning the user that all nodes have been created - this is something we have already learned at the beginning of the previous lesson, but this time let’s make it more sexy by changing the message color and something else!
So let’s add this line at the very end of the previous script:

```
nuke.message("<font color='red'>All nodes have been successfully created")
```

Once executed you’ll see a marvelous red text window popping up ! ...yes I know what you are thinking...yes!! you just need to type you favourite colour into the line and Nuke will get it from the standard palette.
That’s thrilling ! let’s do more! more! Now let’s create a yellow one with the message in bold!

```
nuke.message("<font color='yellow'><b>All nodes have been successfully created</b>")
```



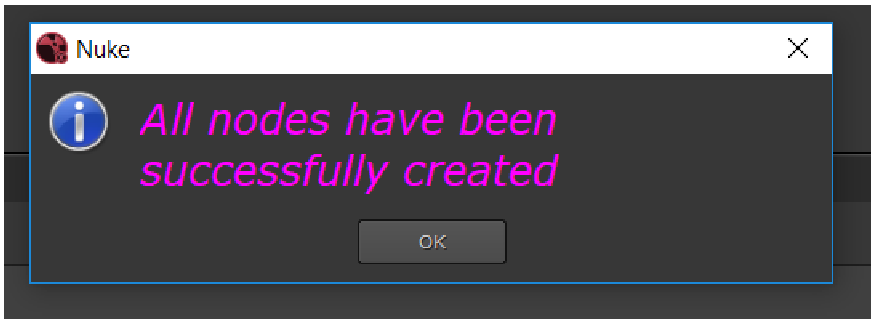
All nodes have been successfully created

You see that we are using a sort of html scripting with it, so in order to make it bold we just added <b> and </b> before and after our lovely text. Quite similar story if you want to turn your text into Italic, just add <i>   </i> before and after the text.

```
nuke.message("<font color='yellow'><i>All nodes have been successfully created</i>")
```

Now let's make it crazy big by changing the font size!

```
nuke.message("<font size='12'><font color='magenta'><i>All nodes have been successfully created</i>")
```



All nodes have been successfully created

That was very cool actually ! so we learned how to customize our own message windows ! So you can play around with it filling up your nuke interface with gigantic texts and bad words ahahah :)

However, we will see how to build much more complex popping windows very soon.

Now let's carry on with our Python for nuke and shall take a look at some more useful commands acting over nodes.
In fact there are many interesting functions and commands you can apply to your comp nodes. Actually some of those ideas may comes out from your daily practice, so I really encourage you to think about the things you would like to improve by using your scripting skills in nuke and just writing those ideas down on a scrap of paper and soon enough I'm quite confident you will make it happen !

Let's make an example, a very basic one: we all know that in many occasion "DeepToPoints" nodes may strongly influence you comp script performance since they weight over your comp even if they are not live into your main tree ! then most of the time they aren't really critical for the comp itself, maybe you have just used them to check your point clouds or for placing other geometries etc and you have left them hanging around your script. That can be

bad.
Well, at this point, why don't cook a code that actually get rid of those heavy nodes automatically ? let's take a look at the following easy lines:

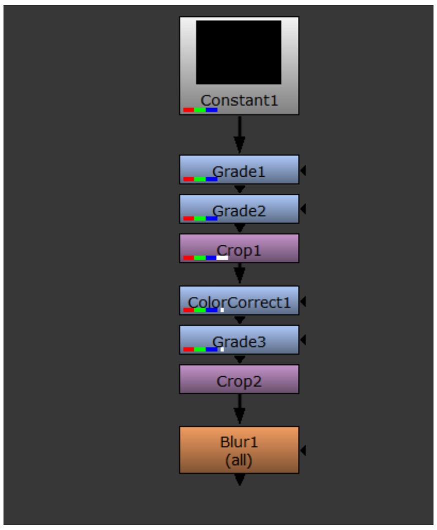
```
s=nuke.allNodes('DeepToPoints');
for n in s:
  nuke.delete(n)
```

Well not the end of the world actually ! This is just using the construct **nuke.delete** to perform nodes deletion.
You see that on the first line I just specified the kind of node I want to act to, which is DeepToPoints.
Tip: Watch out with your capitol letters with this node and in general with all nodes' names !!

Another interesting thing might be the way to extract nodes from your tree. This is something you usually do by using the shortcut ctrl+alt+x but with the power of the python scripting you can actually perform the action over all nodes or, better saying, at least over a number of specific nodes.

So this is the command: **nuke.extractSelected()**

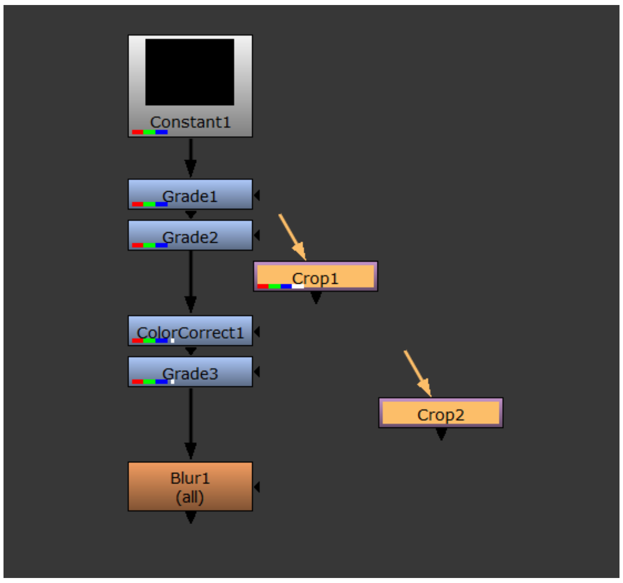
Let's see how it works with one of your scripts. Load one of your favourite comp containing any crop nodes or just create a new one with a few nodes connected together and drop a few crop nodes in the tree, something like this:



Now execute the following code:

```
sel=nuke.allNodes('Crop')
for k in sel:
    k['selected'].setValue(True) # I've selected the nodes before executing next line
    nuke.extractSelected()
```

As soon as you execute the script it extracts your crop nodes from the tree and places them just on the right side.



Watch out when using this command into your script !
Never forget that if you are acting on a Merge node and execute the code, just the data stream previously connected to the B slot will keep going through the tree.
We will soon learn how to improve these lines by telling Nuke to extract a type of nodes only if a few conditions are true, let's say if a specific parameter of a node is over a certain value, etc.

Now let's talk about Groups. At time we have been talking about simple nodes' operations performed over single nodes, but what about groups?
Well a Group is a Class of nodes, so the good news is that we can actually perform all the standard operations the same way we used to do over simple nodes. Obviously there is more you can do within a group but now we are talking normal actions.
Let's say for instance we want to pull a group out of our main tree.. we might use the same scripting we have been using on previous examples, simply replacing the node's type.

```
sel=nuke.allNodes('Group')
for k in sel:
   k['selected'].setValue(True)
   nuke.extractSelected()
```

well done !

Now I'll carry on with something even more interesting about Groups ! In fact by using Python we could actually create or displace gropus and it is very straightforward!
So first of all let's manually selected some nodes in one of our scripts and execute the following:

```
nuke.makeGroup(show=True)
```

this will make a group out of your selected nodes by creating a copy, so it won't delete any node. You will notice that it also turns your Graph window into the group visualization, so you can actually see what's inside. This can be boring, So in order to avoid that you just need to replace the 'Show' command between your parenthesis, to "**False**"

```
nuke.makeGroup(show=False)
```

ahh much better! But the nodes you have made the group from are still there! so why not cooking up a script that actually perform the same but even deleting those nodes at the end of it ? Here you are :

```
sel=nuke.selectedNodes()
nuke.makeGroup(show=False)
for n in sel:
  nuke.delete(n)
```

Looks like working like a charm ! This is simply using the **nuke.delete** command we have already covered.

So now the point is that we have created the group, we got rid of the old nodes but now the new group is disconnected at all, so we would love it being connected within the tree in the right point. How to ? Well this is a little tricky and in order to do it we need to learn some more. So I will put it off for now! But don't worry I won't forget to go through it!

Now let's carry on with smart operations over nodes. There is some time where you just need to clear the animation all over selected nodes. Actually across all knobs or just a few of them. There is a simple command to to that: **clearAnimated()**

Let's build something that actually act on all parameters of a selected node:

```
for a in nuke.selectedNode().knobs():
        nuke.selectedNode()[a].clearAnimated()
```

you could also write it down as following:

```
sel=nuke.selectedNode()
for a in sel.knobs():
  sel[a].clearAnimated()
```

Ok both these constructs need some more explanation: You see there is an "**a**" within square brackets right ? This means that "**a**" is actually a list ! Yes a list, hence when you call "**a**" with square brackets it references all knobs in your selected node. As a consequence the following operations will be performed on all knobs.

Now if we want to remove animations from all selected nodes we will need to write something a little different; one way might be:

```
sel=nuke.selectedNodes()
for a in sel:
  for k in a.knobs():
    a[k].clearAnimated()
```

As you see we are now using 2 “**for**” loops; the first loop sets the variable “**a**” for the selected nodes, the following “**k**” variable references all knobs within the selected nodes, whereas the last row will do the magic for us. Really cool !
Now another interesting thing ! Some time we might need to select nodes before or after a currently selected node. There are a couple of functions that come to help on doing that:

```
for a in nuke.selectedNode().dependencies():
    print a.name()
```

this returns the name of the node before the currently selected one.

```
for a in nuke.selectedNode().dependent():
    print a.name()
```

This code select the node after the currently selected node.

So now let's make it work by creating something thrilling. Let's write a Python script that append a shuffle and a crop node right before our selected write node.
Obviously we need to select it first and then execute this code:

```
for a in nuke.selectedNode().dependencies():
     a['selected'].setValue(True)
     nuke.createNode("Crop", inpanel=False)
     nuke.createNode("Shuffle", inpanel=False)
```

sounds interesting! now let's make it work over all write nodes in your script:

```
ww=nuke.allNodes('Write') #we are acting over write nodes only
for a in ww:
   for y in a.dependencies():
      y['selected'].setValue(True)
      cr=nuke.createNode("Crop", inpanel=False)
      sh=nuke.createNode("Shuffle", inpanel=False)
      sh['selected'].setValue(False)
```

warning: please make sure your write nodes' input is connected to your script tree otherwise this Python script won't work! It is a dependencies script !!!

Now if our nodes looks too close one with the other why not using the **autoplace** command to clean the things up a bit? Let's do it:

```
ww=nuke.allNodes('Write')
```

```
for a in ww:
  for y in a.dependencies():
      y['selected'].setValue(True)
      cr=nuke.createNode("Crop", inpanel=False)
      sh=nuke.createNode("Shuffle", inpanel=False)
      sh['selected'].setValue(False)
for k in ww:  # here I'm selecting all writes nodes in the script
   nuke.autoplace(k)
```

It is important to notice what the line 7 is doing: in fact since we are creating the shuffle node as a last node, it remains selected and if you don't deselect it other write nodes in your script could automatically connect to it destroying your lovely script logic. There are other ways round, but let's do that for now.

Now let's spice our script up by turning it into something useful.
Now I want to add a crop node with precise values to reformat to the root format and then instead of using a Shuffle node I'm going to call a Remove node to get rid of the alpha. So take a look at the highlighted lines.

```
ww=nuke.allNodes('Write')
for a in ww:
    for y in a.dependencies():
        y['selected'].setValue(True)
        cr=nuke.createNode("Crop", inpanel=False)
        cr['box'].setExpression("format.w", 2); cr['box'].setExpression("format.h", 3)
        cr['label'].setValue("reformat to root ")
        sh=nuke.createNode("Remove", inpanel=False); sh['channels'].setValue("alpha")
        sh['selected'].setValue(False)
for k in ww: # here I'm selecting all writes nodes in the script
        nuke.autoplace(k)
```

please be careful with your indentations with all these “for” loops !!

And now Ladies and Gentlemen has come the time to take a look at the “**len**” command! Yay !

len is a particularly useful command to effectively counting nodes or something.
Let's say for instance that we want to know how many Blur nodes sit into our script.
So let's give it try!
Bring up one of your lovely script and do this:

```
s = nuke.allNodes('Blur')
for i in s:
  print len(s)
```

if you take a look at your script window it returns with the number of blur nodes into your

script ! Really nice ! It is worth to mention that if you have a group with a few blur within, those nodes won't appear in the calculation !!! we will sort it out soon !
Now let's make it more interesting:

```
s = nuke.allNodes('Blur')
L=str(len(s))
nuke.message(L)
```

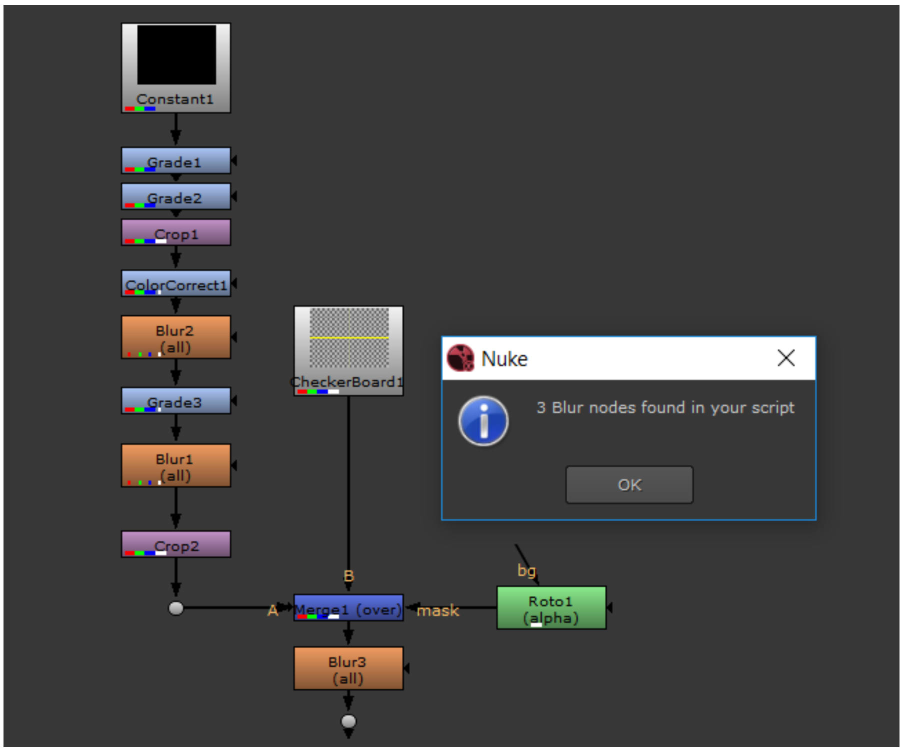
Yeah, now it pops up with a small message window displaying just the number of blur nodes on your script. Besides you have noticed that I put an "**str**" just before the **len** operation, well this is crucial to return a string to feed up our **nuke.message** construct.
We could also reduce our 3 lines of code by writing it like that:

```
s = nuke.allNodes('Blur')
nuke.message(str(len(s)))
```

now let's sexy it up by adding a nicer text:

```
s = nuke.allNodes('Blur')
txt=" Blur nodes found in your script "
nuke.message(str(len(s))+ txt)
```

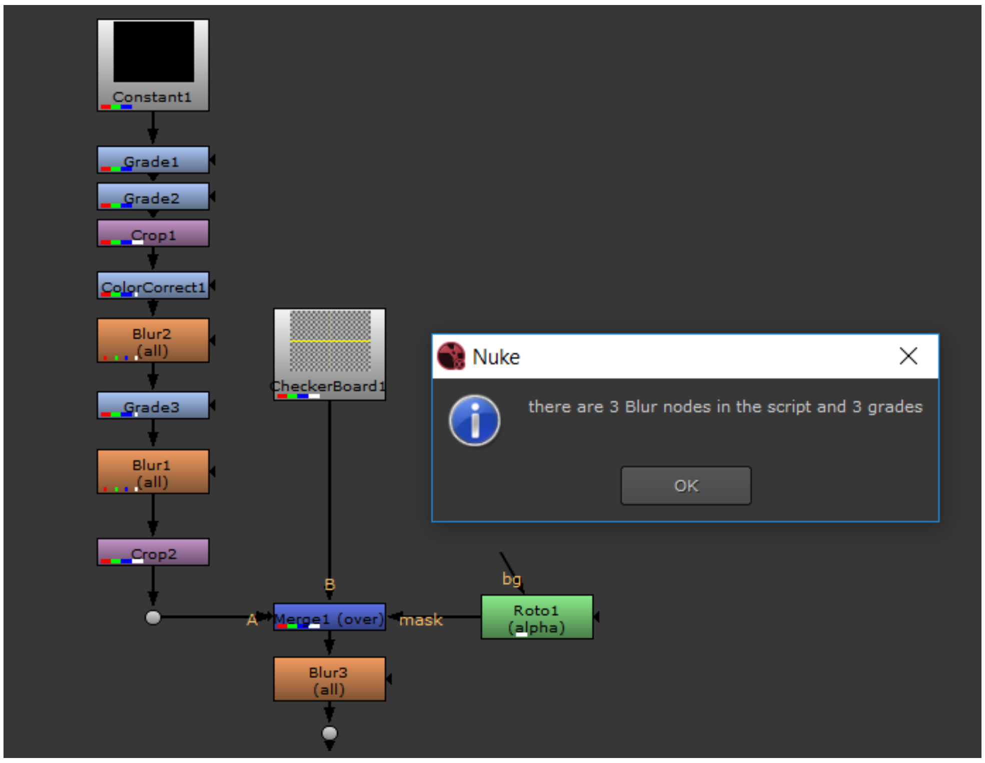
Ok we are learning something new here too! Basically In order to add some text before or after the result of the calculation within the nuke.message, what you need to do is just typing it with a "+".
Please also notice that I added a space between the quotes and the word "Blur" over the second line, this is just to create a space between the calculated values and the text.
Eventually the outcome should be looking like this:



3 Blur nodes found in your script

now let's run a two-nodes counting:

```
blurs_n=len(nuke.allNodes('Blur'))
grades_n=len(nuke.allNodes('Grade'))
nuke.message("there are " + str(blurs_n) + " Blur nodes in the script and " +
str(grades_n) + " grades")
```



there are 3 Blur nodes in the script and 3 grades

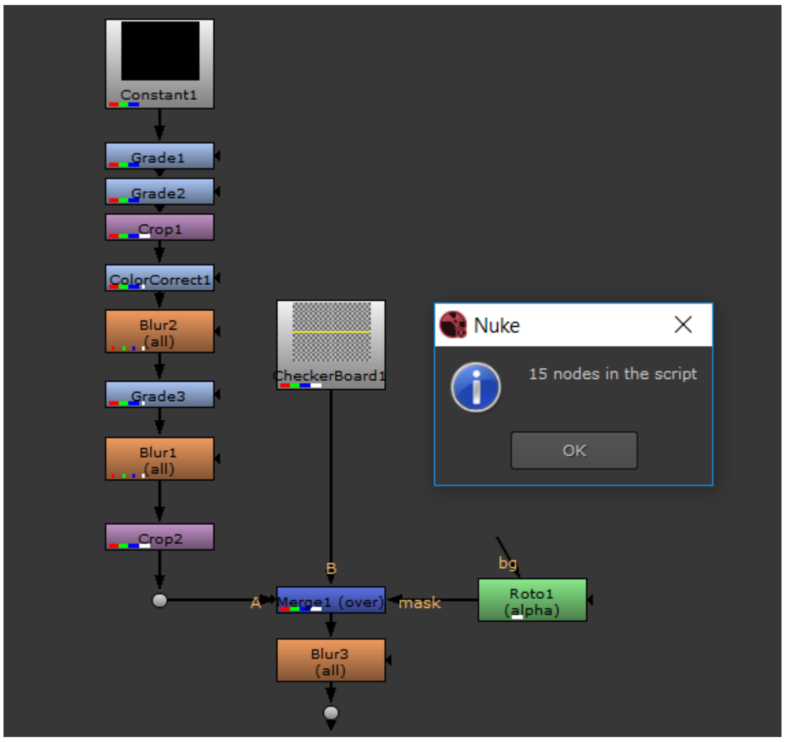
You see ? Basically over the third line I'm just adding texts and strings with corresponding values!
Just for your information, you could also write the code this way too:

```
blurs_n=len(nuke.allNodes('Blur'))
grades_n=len(nuke.allNodes('Grade'))
nuke.message("there are %s Blur nodes in the script and %d Grades" % (blurs_n,
grades_n))
```

...where the **%s** means **str** and **%d** means a decimal number and the last **%** is taking values from those within subsequent parenthesis. Well actually this syntax is now considered water under the bridge, so you may use the syntax we covered before.

Now what if we want to count the total number of nodes within our comp scrip ?
Well this is an easy one!

```
txt = str( len( nuke.allNodes() ) )
nuke.message( txt + ' nodes in the script')
```



15 nodes in the script

nice one ! Just notice that it is also counting the dots nodes ! So what if you want it to take them out of your counting ? Well it is very easy! Just try that :

```
txt = str(len(nuke.allNodes()) - len(nuke.allNodes('Dot')))
nuke.message( txt + ' nodes in the script')
```

What I have done is just the difference between the **allNodes** counting and the **dots**

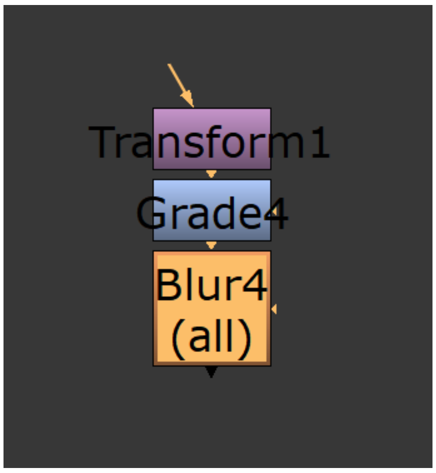
counting ! It's a piece of cake ! So now you should have a result of 13 instead of 15 !!
Be very careful with all the parenthesis you are putting in ! Always count what you are opening and closing !

You see ? We are getting even more skilled!! and this is happening page by page! So now let's start covering “lists” ! Ahhh Finally we are getting there!

So what the hell is a list ? Well the good news is that a list is basically a list ahahha :)
ok I'll be serious :) basically by using a list you can create a proper list of objects and they could be either nuke nodes, values or even other lists themselves and then you can use them throughout your Python script to perform many operations like automating processes etc..
First unforgettable rule: lists must be created within square brackets !

You can fill a list by adding objects or nodes based on particular conditions, yet you can add, remove or replace elements from your list, so basically what is really nice about it is that once a list is created you can perform operations with all objects within or, on the contrary, leave them out of your processes, etc.
However going through some practical examples will clarify more than too much words, so let's try this:

```
t = nuke.createNode('Transform', inpanel=False)
g = nuke.createNode('Grade', inpanel=False)
b = nuke.createNode('Blur', inpanel=False)
nn = [t, g, b] # this is the writing for creating a list
for k in nn:
   k['note_font_size'].setValue(32)
```



So basically this script starts by creating 3 simple nodes then at the line 4 we are creating a list by adding the node's variables into it. Over the following line we are setting up a new variable **k** to the entire list, whereas at the very last line we are increasing the font values...and this will be performed over the 3 nodes in the list, all at once ! Awesome !

Well this was a very first primitive example showing what you can do with a simple scripting and a list, but there's much more you can do actually! In fact before carrying on with further code I have to mention how a list can be manipulated.
In order to explain it as simple as possible I'll go through a few basic examples.
So let's assume we are creating a simple list like this:

```
starwars= ['R2D2','Solo','Yoda']
```

now let's print it:

```
print starwars
```

it will return with:
```
['R2D2','Solo','Yoda']
```

So far so good. But now let's set up a very basic nuke message popping window:

```
starwars= ['R2D2','Solo','Yoda']
nuke.message(str(starwars))
```

….and, again, if you execute that it will return a nice pop-up message with the content of the entire list. So far so good !!!

Now let's assume we want to print out just the first object of the list. Ok Now we are going to learn some further interesting syntax:

```
starwars= ['R2D2','Solo','Yoda']
nuke.message(str(starwars[0]))
```

I just added a zero between square brackets after the list name. So this is telling Python to get just the first object in the list, as you remember it always counts from 0 so the first element is **'R2D2'**. So then if you try with 1 or 2 it will return with subsequent words.

Now let's try that:

```
starwars= ['R2D2','Solo','Yoda']
nuke.message(str(starwars[1:]))
```

if you execute that it gives you “**Solo**” and “**Yoda**”, so that [1:] means that it is only considering objects from an index of 1 onward (counting from zero obviously).

Now what if we write:

```
starwars= ['R2D2','Solo','Yoda']
nuke.message(str(starwars[:2]))
```

well now I have inverted the colon and the number and it will return with “**R2D2**” and “**Solo**”, hence the first and second object in the list. Basically it is taking out the index n.2 object that is “Yoda” …..Mmmmm Interesting!

Now let's stretch our list by adding further objects into it:

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
nuke.message(str(starwars[0:3]))
```

Executing it will return the first 3 objects **'R2D2', 'Solo', 'Yoda'**.
Basically this time the number 3 is telling python to get all objects up to the third in the list, that is “Yoda”, counting from zero obviously.

You can play with index numbers further more to get any other outcomes :

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
nuke.message(str(starwars[2:4]))
```

So this time we have: **'Yoda','Luke'**
Another interesting construct might be:

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
nuke.message(str(starwars[-2]))
```

That will return with the name '**Luke**' – so basically by adding a negative sign before the index number we are forcing Python to count from the bottom of the list.
Intriguing uh ?

Don't forget these nice rules and the syntax we have been learning because they might turn to be very useful during our programming practice with Nuke !
So let's study something more about lists.

A list is like a container, so basically you could decide to put other objects within, replace some or even delete. Each of these operations have its own construct. Let's go through and check out some of the most used.

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
starwars.append('my_favourite_characters')
nuke.message(str(starwars))
```

As you can see on the second line I'm using the **append** function to add another object into my list.

now let's remove a specific object in the list:

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
del starwars[2]
nuke.message(str(starwars))
```

so here we are using **del** to get rid of **'Yoda'** (actually It sounds very sad aahhaa :)
Please bare in mind that the command **del** can be used only if you know exactly where your object sits, its index, otherwise if you want to remove just the object by its name, you have to use:

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
starwars.remove('Solo')
nuke.message(str(starwars))
```

To remove an object from a list you can also use the **pop** command that is quite similar to the previous but this guy removes the objects at specific index and returns it ! However the syntax is a little bit different:

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
starwars.pop(2)
nuke.message(str(starwars))
```

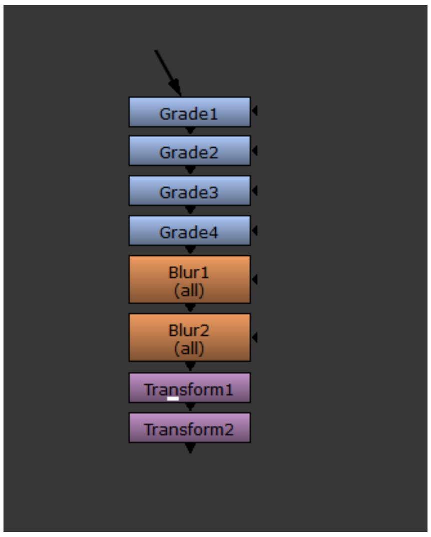
Please note that after the **pop** command there are two normal parenthesis not squares !

So now let's take a look at the replacing syntax, that is a little bit tricky:

```
starwars= ['R2D2','Solo','Yoda','Luke', 'Chewbaka']
starwars[starwars.index('Chewbaka')] = 'Mr.Bean'
print starwars
```

We are getting rid of Chewbaka (so sad with it), to see Mr.Bean driving the Falcon ! Well the idea is fascinating actually. Basically executing that code we are removing '**Chewbaka**' giving way to '**Mr.Bean**'.
Quite bizarre but interesting Python-wise.

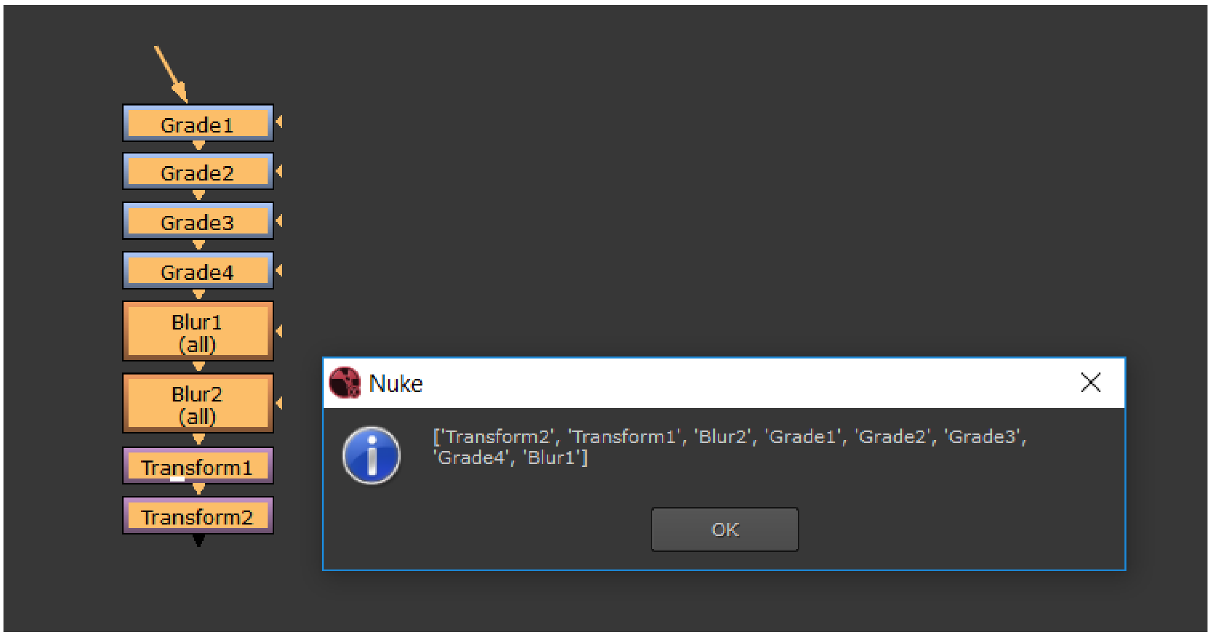
Ok at this point you might be wondering why I'm putting all this stuff over the table, well the point is that we have been playing with character's names and stuff, but what if we use Nuke nodes instead ? In fact by using lists you could create many interesting scripts for manipulating your beloved nodes! Possibilities are endless, so let's take a look to some of them.
Now in order to do tests clear out our nuke script and let's create a few nodes: 4 grades, 2 blurs and 2 transforms. Like this:



so now let's highlight all of them and execute this:

```
lis = []
for s in nuke.selectedNodes():
   n = s['name'].value()
   lis.append(n)
nuke.message(str(lis))
```

Ok first of all don't get scared with the first line, that is not a celtic or ancient symbols, they are just a couple of square brackets!! Why that ? This is simple, we are creating an empty list that we are going to fill up later on in the script.
On the second line we are creating a new variable representing all selected nodes and then over the third one we are telling Nuke to create another variable (**n**) that is only taking care of the node's names.
So over the 4th line we are appending all names coming from all selected nodes into our list; as a consequence the last line pops up a message printing the list's content.
This is what you get once you execute it:



```
['Transform2', 'Transform1', 'Blur2', 'Grade1', 'Grade2', 'Grade3',
'Grade4', 'Blur1']
```

This is a basic script just useful to print on screen a pop up message with the names of all selected nodes. Not the end of the world actually but it is something and we will learn a lot from it. Obviously Python lists are going to be much more useful when we will go through conditions.

Now deselect all your nodes and select them again making sure to do it one by one from top to the bottom. I am asking it because your list will be created accordingly to the selection order, so please never forget that !

now execute this:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[0]
print lis
```

...you will see your list appearing and you notice that the last node (Transform2) has been took out of your list. This gives us the opportunity to talk about the indexing order! You might have realized that Nuke is actually counting our nodes from the bottom, so the index 0 will be our last selected node. Good to know !
Similarly if we write **[2]** instead of **[0]** and run your script again you will see that the **Blur2** has now been taken out your list.

Now let's put in something more. With the following lines we are actually deleting all nodes but the last selected:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[0:1]
print lis
for k in lis:
    f=nuke.toNode(k)
    nuke.delete(f)
```

...you see ? This time we are using 2 digits **[0:1]** ; The script is selecting all nodes indexes from 1 onward except for the index 0 (Transform2) and deletes them all thus leaving the **Transform2** untouched!
Over the last 3 lines we are basically creating a variable representing the empty list, then we are selecting and deleting it.

Obviously if we write **[0:2]** the last two **Transform1** and **Transform2** nodes will be kept into the script.
It might sounds tricky at the very beginning but some practice will make you more familiar with this stuff.

Now try the following lines: it just deselects your first selected node:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[0:-1]
print lis
for k in lis:
    f=nuke.toNode(k)
    f['selected'].setValue(False)
```

Well all of this might sounds weird at first sight but it is actually taking all nodes out of the list except for the Grade1, so then we select the list back again and deselect the node within, we are actually acting only over the **Transform2**.

…. or you can deselect all nodes but the last selected:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[0:1]
print lis
for k in lis:
    f=nuke.toNode(k)
    f['selected'].setValue(False)
```

Yet, this is getting rid of the first selected node:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[0:-1]
print lis
for k in lis:
     f=nuke.toNode(k); nuke.delete(f)
```

Another interesting code is the following:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[1:-1]
print lis
for k in lis:
    f=nuke.toNode(k)
    f['selected'].setValue(False)
```

So basically here we deselected the first and last node of your selection !!!
It is worth to notice there is no reason for creating a new variable **f** just to perform the node deletion actually, so we could simplify it a little bit over the last lines:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[1:-1]
print lis
for k in lis:
     nuke.toNode(k).setSelected(False)
```

Similarly we could perform a deletion of the first and last selected nodes:

```
lis = []
for s in nuke.selectedNodes():
   n = s['name'].value()
   lis.append(n)
```

```
del lis[1:-1]
print lis
for k in lis:
   f=nuke.toNode(k); nuke.delete(f)
```

… and now, similarly to the syntax we have already seen with the starwars examples, we could write:

```
lis = []
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[1:]
print lis
for k in lis:
    f=nuke.toNode(k)
    f['selected'].setValue(False)
```

this is actually selecting all nodes, then it takes them all out of the list a part for the index 0 that represent the **Transform2** that is actually the only node remaining in the list, then over the last line we deselect it.
The result is the last selected node is now deselected.

Now if you invert the value within the square brackets to **[:1]** it is going to deselect everything except for the only node remaining in the list that is, again, the **Transform2**.

Now take some time to play around with this stuff! In the mean time I'll take a Girella

Yay !

Don't forget to pay attention on loop writing and indentation!! never overlook that ! As anticipated if you type a wrong indentation your script might perform in a very different way !

The following is using a different way to perform a deletion. It will acts on just one precise indexed node and it is using the **pop** command :

```
lis = []
for s in nuke.selectedNodes():
   n = s['name'].value()
   lis.append(n)
re=lis.pop(2)
for k in re:
   f=nuke.toNode(re); nuke.delete(f)
```

this syntax is a little bit different with others, here you see I'm using normal parenthesis instead of square brackets to assign the index number.
However it will remove the **Blur2** that is in the index position 2.

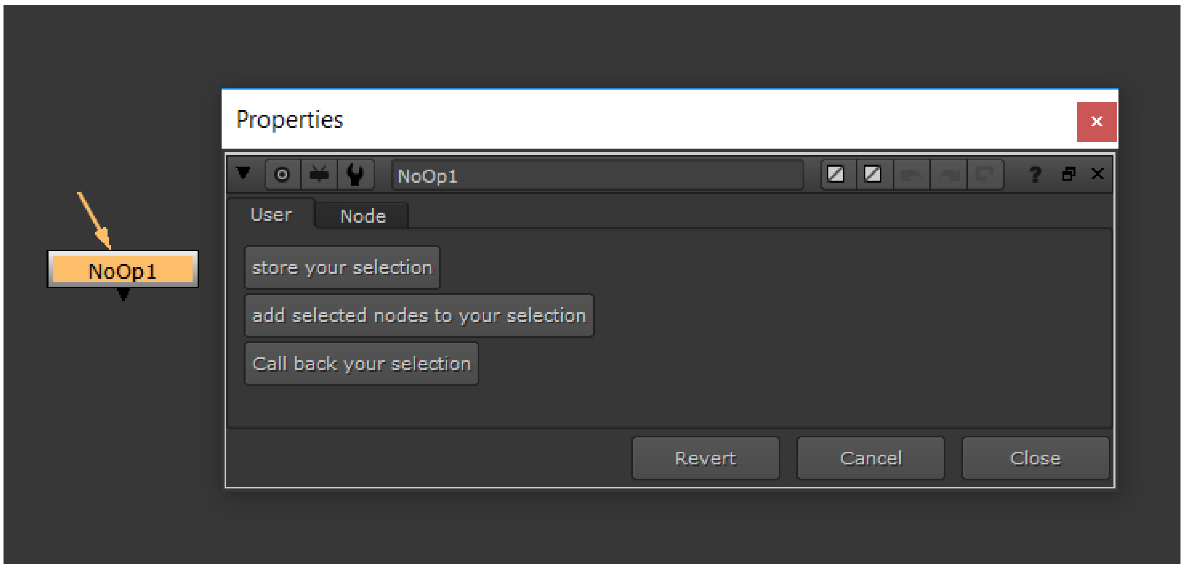
So now that we covered how to manipulate lists and before wrapping up lesson 3 I just wanted to show you a nice idea that can be created with the Python stuff we have been through.

The goal is creating a nice tool (that you could store into your toolsets or append it to your menus) to save a selection of nodes and then recall it any time you need it during your comp work. I've been using it when I worked on some stereo show and I needed to temporary disable a few stereo nodes. Obviously your selections is kept stored until your current nuke session is up.

So let's make it very simple for now by using Python script buttons:
Just pull in a NoOp node and add 3 Python Script Buttons to it.

You could name them like: “store your selection” , “ add selected nodes to your selection” , “Call back your selection”

and it should be looking like this:



So now we have to type some code within each node python script's area. You know we have already covered how to do it on the previous lesson. Let's start with the first button. First, we need to store selected nodes into a list, so let's write this:

```
lis=[]
sel=nuke.selectedNodes()
for k in sel:
    lis.append(k)
```

so now with the second button we might want to append further selected nodes to the same list, so let's type this into the second button's python script area.

```
for b in nuke.selectedNodes():
     lis.append(b)
```

In the end, by using the third button we want to call back the selection, so let's type that:

```
for nuke.allNodes in lis:
    nuke.allNodes['selected'].setValue(True)
```

Beware: if you are copying and paste from this page straight to your python script button's area please make sure you are respecting the right indentation. So once pasted, please close and re-open the python script button's window to check all is correctly indented.

Now save this lovely NoOp node into your toolsets. Then open it up and select a few nodes into your favourite script and experiment with your new tool !
I think you will enjoy it! You see ? You are actually creating something very useful in a very

easy way !

In Conclusion:
There are many others commands for manipulating lists, we have just covered what might be more useful for our comp practice...who knows one day if these lessons will becoming a proper book I could go deeper with it :)

And now....hey hold on! where are you going ? Don't run away ! haven't finished with you yet !

Homeworks!!!!!!!!!!!!!
ok since you have been very good with previous assignments now you scored 3 amazing new ! Ahaha don't worry if you haven't drift off during this lesson it is going to be looking quite easy.
So here you are:

1. create a Python script that select all grade nodes in a script, creates a list and returns with the total number.

2.Create a Python script that counts all selected nodes, then warns the user the last node will be deleted and then does it.

3. Create a Python script that delete all shuffle nodes in a script. This is something you should be already familiar with since we did something similar with the basic functions we already covered back to the first lesson, so please try with it and then provide a different solution by using lists.

4. why not a further exercise ? Don't worry this is going to be very easy:
just re-build the selection tool we have created but this time instead of creating 3 buttons into a NoOp nodes just create menus on top of your nuke interface.
That's it !

I hope you had some fun with this very long lesson !!!

Let's see you soon to the next one !!

Cheeeeeeers!