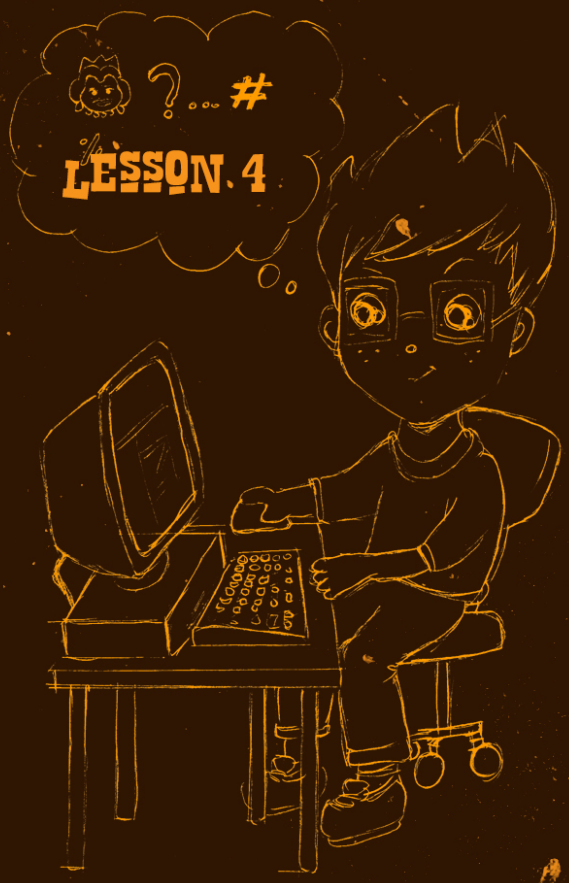
PYTHON SCRIPTING FOR SMART AND CURIOUS COMPOSITORS



Gianluca Dentici
www.gianlucadentici.com

LESSON 4

Hi guys welcome back to your “Python scripting for smart and curious compositors” ! lesson 4 ! wow ! How are you guys doing ? This lesson is going to be a little more tricky and you might find out there is so much stuff going on for just one lesson, so please feel free to ideally split it in two parts if you like and take your time to familiarise with all the stuff ! there’s no point on rushing off and don’t be scared of falling behind, your lessons are waiting for you on my website and you can catch up with it whenever you got ready!

Now first of all let me say thank you to all of you! it looks like you are enjoying the course and the feedback is really positive, the followers number is bursting ! Yet I’m very happy that many of you are getting more familiar with the language and trying to cook their own code!
In order to say a proper thank you I couldn’t help but baking Brownies! it is a good treat, isn’t it ?

so now you are thinking..well you bake your own brownies for yourself, how in the world we could taste them ? … yeah ok you are right...so I might suggest to bake your own but with my precious recipe ! so find it here!!!

http://www.gianlucadentici.com/brownies_recipe.pdf

you see ? thanks to this lesson you’ve got a +1: the authentic and outstanding american brownies recipe ! Yayyyy!
So now let’s jump onto the boring stuff...I’m joking ! this is going to be exciting !

Homework solution:
so the **assignment 1** was:
“Create a Python script that selects all grade nodes in a script, creates a list and returns with the total number”

```
s = nuke.allNodes('Grade')
txt="  Grade nodes found in your script"
nuke.message("<font size='12'><font color='cyan'><i>"+str(len(s))+txt)
```

Please note there is some space I put before the word " Grade" , It's because we want some space between the calculated number and the text. Then over the last line I'm just putting all the things together.
At the end of the day this one was quite straightforward, wasn't it ?

so let's move on to **assignment 2** :
"Create a Python script that counts all selected nodes, then warns the user the last node will be deleted and then does it"

..and here you are the solution:

```
lis=[]
txt = str(len(nuke.selectedNodes()))
nuke.message( txt + ' nodes in the script')
nuke.message("the last selected node will be deleted")
for s in nuke.selectedNodes():
    n = s['name'].value()
    lis.append(n)
del lis[1:]
for k in lis:
    f=nuke.toNode(k)
    nuke.delete(f)
```

well, not the end of the world right ? we are just using list commands we have been covering on previous lessons and a few other lines of codes we were already familiar with !

Now the **assignment 3** was:
"Create a Python script that delete all shuffle nodes in a script. This is something you should know since we did make 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"

so the first method is:

```
for k in nuke.allNodes('Shuffle'):
     nuke.delete(k)
```

….and the second method would be:

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

```
f=nuke.toNode(k)
nuke.delete(f)
```

Yeah!

...and here we go to the last but easier **assignment 4**:
"..just re-build the selection tool we had already created but this time instead of creating 3 buttons into a NoOp nodes just create top menus for your nuke interface".

Basically we have to create 3 py documents and put the right code within, then we have to save them all individually by naming them respectively **sel1.py**, **sel2.py** and **selcall.py f**or instance and ultimately we need to edit the **menu.py** file by adding following lines:

```
nuke.menu( 'Nuke' ).addCommand( 'selection_tools/store your selection', lambda:
nuke.createNode('sel1' ),'CTRL+g')
nuke.menu( 'Nuke' ).addCommand( 'selection_tools/add to your selection', lambda:
nuke.createNode('sel2' ),'CTRL+h')
nuke.menu( 'Nuke' ).addCommand( 'selection_tools/call back selection', lambda:
nuke.createNode('selcall' ),'CTRL+j')
```

You notice that I've also added 3 shortcuts **ctrl+ g,h,j** but you might use your own.
And now it should be working ! go check your brand new menu over the nuke's top bar and enjoy your fresh tools !

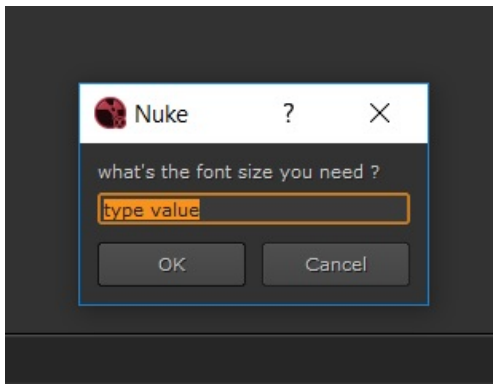
Now guys let's move on and try to be more proactive with out scripting. So far we have been writing scripts where we typed parameters' values by hand directly over the lines but what if we want the user to set his own ? so it comes the time to learn some more code folks!!! let's say we want a popping window where the user can actually input values or strings... Well all this verbose presentation would make you think that it is going to be tricky, whereas it is actually really easy !! I'm just introducing it like it was Spielberg ahaha !

What you need to do is using the command **getInput(),** So let's go through it and try a few examples.
Let's assume I want to perform a font size change over any selected node. I would write:

```
hh=nuke.getInput("what's the font size you need ?", 'type value' )
for a in nuke.selectedNodes():
    a['note_font_size'].setValue(int(hh))
```

As you can see the new guy has came through the first line ! let's analyse the code:
now as first argument between parenthesis you have to add a window's message - this appears on the top bar of your popping window - then, as a second argument, type a string, this will appear within the user input area. The deed is done !
Please note that I have also created a variable called **hh**.
Now if you look at the last line you will see that we are no longer using a numeric value to set the font size, in fact by placing the variable **hh** you will get the value typed by the user !! yay !
Obviously since the **getInput** command is looking for a string and you need to use a floating number it is imperative to convert your input string into a float value, this is why I put that **int(hh)**.

So if you now execute the code over your selected nodes you will see them changing their font size!!
So your brand new input window should be looking like this on execution:



wow how exciting ! just imagine the possibilities !!!

Consequently we could also add a nice popping windows at the end of the process just to confirm the font size you have been asking for has now been changed.

```
hh=nuke.getInput("what's the font size you need ?", 'type value' )
for a in nuke.selectedNodes():
    a['note_font_size'].setValue(float(hh))
nuke.message('now the font size is ' + hh)
```

If you take a closer look at the last line you will notice that I've just added the same variable **hh** to the **nuke.message** constructor**,** so the first part is a string and then I'm adding up the variable's output, hence the value the user typed in ! it is really simple like that !

Now why not testing some code we already covered and customize it with **getInput** ?
shall we ?
The following script is the one we have been using to change the label to selected nodes, we are now simply adding the **getInput** to allow the user typing his own label:

```
lb=nuke.getInput("what's your label ?", 'type value')
k=nuke.selectedNodes()
for ex in k:
    ex['label'].setValue(lb)
```

You see over the last line ? I've just typed in the variable **lb** within parenthesis, means it is going to put the user's input string outcome into it! Awesome !

Now let's use the **getInput** again with something even different. Let's say we want to extract just a kind of nodes out of our main tree. We are already familiar with the **nuke.extractSelected()** command, so just let's make it work with **getInput**!

```
rn=nuke.getInput("what kind of nodes you want to extract ?", 'type name with capitol letter')
sel=nuke.allNodes(rn)
for k in sel:
    k['selected'].setValue(True) # I've selected the nodes before executing next line
    nuke.extractSelected()
```

To a vigilant eye it couldn't slip away that for some mysterious reason I haven't put the variable straight within the parenthesis over the last line, the reason is **nuke.extractSelected()** doesn't allow any parameters.
The other point is: don't forget that as user you'll have to type the node class on capital letters into your input window!

So now here it comes the intriguing part! Since we have learned how to input values and strings, why not trying and test it on something very useful and create a kind of path replacing tool?
we all know how can be boring when we are given scripts coming from someone else's machine or pipeline and all read nodes go crazy because of the missing paths…. so what ? are you going to open each node and replacing the path manually ? **no way** ! well let's make python do it for us !!
This is something I've been using on some companies in London since, for production reasons, we had been given a bunch of scripts coming from other vendors and obviously the paths were a mess up! I've been asked to find out a solution to make it work and...
Ok find here the code:

```
import os
inpa=nuke.getInput("path replacer","type the path you want to replace")
inpb=nuke.getInput("path replacer","type your new path")
readnds = nuke.allNodes('Read')
for x in readnds:
    pth = x.knob('file').value()
    npth = pth.replace(inpa, inpb)
    x.knob('file').setValue(npth)
```

ok now let's go through it.
First of all there is an **import os** ...well I don't want to go too deep into it for now so I'm going to limit myself by telling you that it just imports a particular module, hence a library of further functions that deal with files and in general with the operative system; In fact **os** stands for operative system.
You might wondering why we are calling it right now, well the reason is that we are going to use a particular command that comes with that module. However we will go further into details with **os** on future lessons, so let's just take it for granted for now.

On the second and third lines I'm creating two variables **inpa** and **inpb** that are respectively the path to replace and the replacing path; here we are using our lovely **getInput** command. Then on the 4th line I'm telling him to act on all read nodes. So now the the loop starts here and over the very first line it is looking for the file path value, in fact the newly created variable **pth** prints out the path value thanks to the **value()** construct.

But the actual magic happens at line 7 where we are using the **replace** command; This is a specific command coming from the **os** module we have been importing into the code at first. What it is doing should be looking quite clear, it is taking the path (**pth)** and by taking the first input **inpa** it's replacing it with the second input **inpb**, so this is the reason for them sitting within parenthesis.

Ultimately the last line actually sets the new file path with the outcome value coming from the previous line. That's it !
So once executed you will be prompted with your beloved input window asking for the path to replace; now bare in mind that you only have to type the part you need to replace.

Let's assume we have a path like this:
/jobs/Starwars/sw01_tmp033_1140/ELEMENT/ smoke_puff_v006.####.exr

and we need to turn into this:
/jobs/SW/sw01_tmp033_1140/element_library/ smoke_puff_v006.####.exr

we shall input just the folder structure:
/jobs/Starwars/sw01_tmp033_1140/ELEMENT/

and as second input the desired replacing folder structure:
/jobs/SW/sw01_tmp033_1140/element_library/

...and the game is over !

However you might have experienced some difficulties on copying the paths you are targeting and paste them into the user input window. Yeah I agree, so let's fix it up by improving our code and telling him to automatically place the file path of one read node right into the first input window, in a manner so we can then edit it easily. In order to do that I'm assuming I have a **Read1** node into the script - I am sure you have it too :)

```
rn = nuke.toNode('Read1').knob('file').value()
inpa=nuke.getInput("path replacer",rn)
inpb=nuke.getInput("path replacer","type or copy/paste your new path")
readnds = nuke.allNodes('Read')
for x in readnds:
    pth = x.knob('file').value()
    npth = pth.replace(inpa, inpb)
    x.knob('file').setValue(npth)
```

The first line is actually calling and retrieving the file path
so now once the code is executed you'll find out that the Read1's file path will be placed there into your input window! wow ! but now the problem is that we have the entire folder structure as well as the file sequence name ! and we don't want that. We just need the folder structure.
So let's dip into the **os** module again and fish out another function. For the sake of clarity I'm just typing it here as a separate example :

```
import os
ipath = 'C:/GIANLUCA/pictures/jurassicworld/14feb2018/280213_135241.exr'
dn = os.path.dirname(ipath)
print dn
```

Execute that and you realise that **os.path.dirname** just returns with the entire folder structure and nothing more. Got it ? Now let's stick it into a new script.

```
import os
rrr = nuke.toNode('Read1').knob('file').value()
dn = os.path.dirname(rrr)
inpa=nuke.getInput("path replacer",dn)
inpb=nuke.getInput("path replacer","type or copy/paste your new path")
readnds = nuke.allNodes('Read')
for x in readnds:
    pth = x.knob('file').value()
    npth = pth.replace(inpa, inpb)
    x.knob('file').setValue(npth)
```

tip: if your replacing path is similar to the path to replace don't forget to ctrl+copy it so you can paste it into the following window and make your changes.

By the way...how funny is it ? think that by next lessons we could be able to enhance this script using decision making operators ! ...and once we cover "Panels" we will improve the

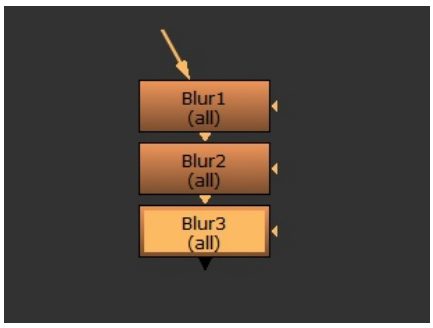
user input window on a much more elegant and convenient way.
In regards to **os** functions, we will be covering some more very soon !

now let's have a break ! wooooooah !
I'm going to put on Steve Vai's "Fire Garden" one of the best ever ! youtube it here:
https://www.youtube.com/watch?v=MKgFWNUIZXs

Woah! What a great sound!
Now let's start to peek at the '**range**' command; this is a very useful guy and you can use it for many purposes. You can repeat what's inside your loop as many times as you want by just specifying a range value or for generating increasing values etc..
Now let's assume we want to create 3 blur nodes, so instead of typing the standard creation string for 3 times that would look quite awkward,we could hit the goal this way:

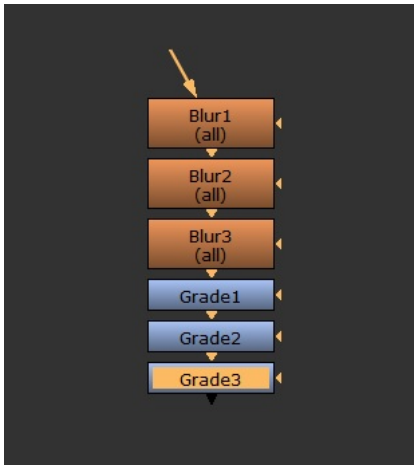
```
for ex in range(3):
    nuke.createNode('Blur', inpanel=False)
```

...and if you execute it you will see 3 connected blur nodes coming up! Easy peasy! The number of repetition is clearly the one between parenthesis. Furthermore the last node is kept selected unless specifically addressed.



now let’s investigate more with the range function ed execute the following:

```
for ex in range(3):
    nuke.createNode('Blur', inpanel=False)
for k in range(3):
    nuke.createNode('Grade', inpanel=False)
```



The outcome is quite obvious as we ran 2 loops with 2 different range functions, therefore we have been creating 3 blur nodes followed by 3 grade nodes.
...and what if we execute that:

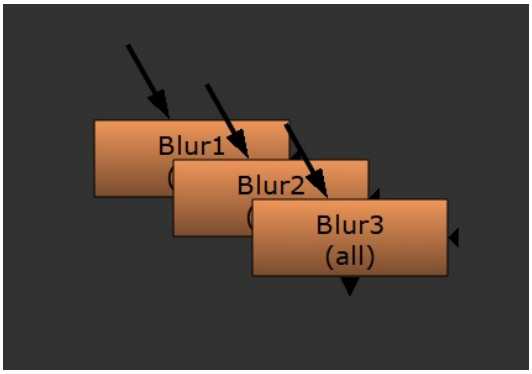
```
for ex in range(3):
    nuke.createNode('Blur', inpanel=False)
    nuke.createNode('Grade', inpanel=False)
```

well this creates an alternation of blurs and grade nodes, all connected.

Well the basic rule here is that everything following the loop beginning is executed multiple times in the exact order and don’t forget that the creation of nodes is always happening the

same way you would do in the graph editor, means that you must pay attention to what is currently selected.
In fact the nodes' selection is very critical when using loops with the **range** command, means that your code outcome will vary depending on whether the nodes are kept selected or not. As a matter of fact if we type a line to deselect the node within a loop this will be performed repetitively each time the loop executes its content. So basically we are creating 3 disconnected blur nodes and this is happening because it creates a node and then deselects it before creating the next and so on.

```
for ex in range(3):
    nuke.createNode('Blur', inpanel=False)
    nuke.selectedNode()['selected'].setValue(False)
```

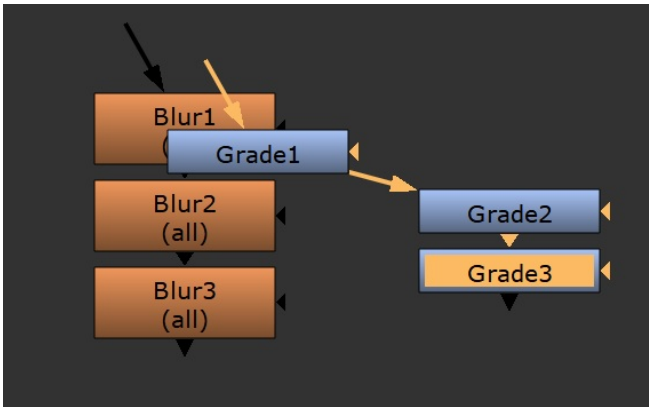


Other than that please watch out with what you get if you change the indentation !!!

now let's assume we want to keep the last created grade node selected, we could write something like this:

```
for ex in range(3):
    nuke.createNode('Blur', inpanel=False)
nuke.selectedNode()['selected'].setValue(False)
for gra in range(3):
    nuke.createNode('Grade', inpanel=False)
```

you see how I have written this code ? this is very important to understand, in fact If I had omitted the third line the result would have been completely different like the one you see in the picture:



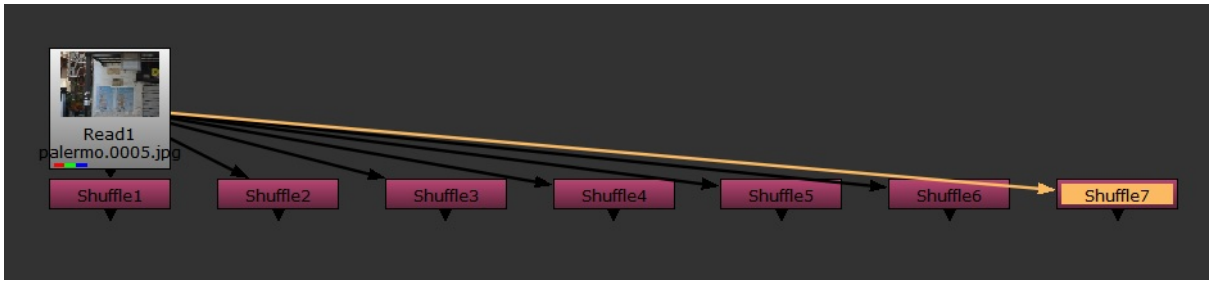
now why not using our beloved lists to improve somehow these scripts ? yeah let's go for it! The following code creates a list called **lis** then it creates 7 Blur nodes and store their names into the list.

```
lis=[]
for ex in range(7):
    bl=nuke.createNode('Blur', inpanel=False)
    n=bl['name'].value()
    lis.append(n)
print lis
```

...imagine the possibilities you could have now that you know exactly the name of each node!
Now let's carry on with something really cool ! let's assume we've got a Read node and we want to connect 7 different shuffles to it...well this is something you might be already able to do since we have covered how to connect inputs! so let's patch all the things together and put your knowledge in place and see what's going on!!
Now select a read node first and then execute the following:

```
sn = nuke.selectedNode()
for k in range (7):
    hh = nuke.createNode("Shuffle", inpanel = False)
    hh.connectInput(1,sn)
```

you should be getting something like this:

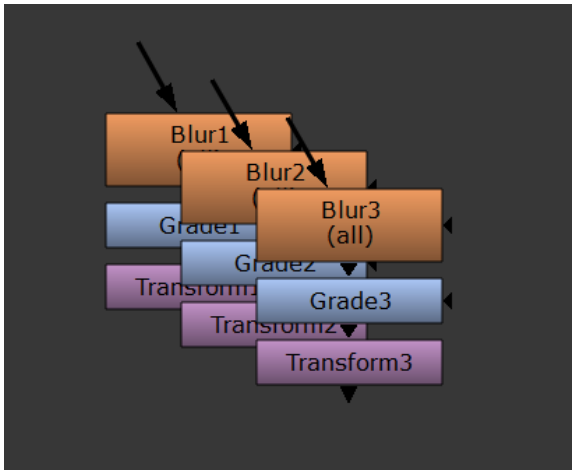


And this is happening because of the last line! basically the code that is connecting the

shuffle inputs are repeated as much as the nodes' creation, 7 times.

Impressive isn't it ? can you feel butterflies ?...does it sounds any bells at all ? ok let me answer that for you: yes! we could create a sort of AOVs comp template with all those shuffle nodes labeled on our outstanding CG passes for instance, what do you reckon ? and we could sex it up with even more functions! However it is not yet the right time to do so, but we will go through it very soon !
In the meantime let's carry on with other exciting stuff you can do with **Range** ! Now clean up your workspace and execute the following:

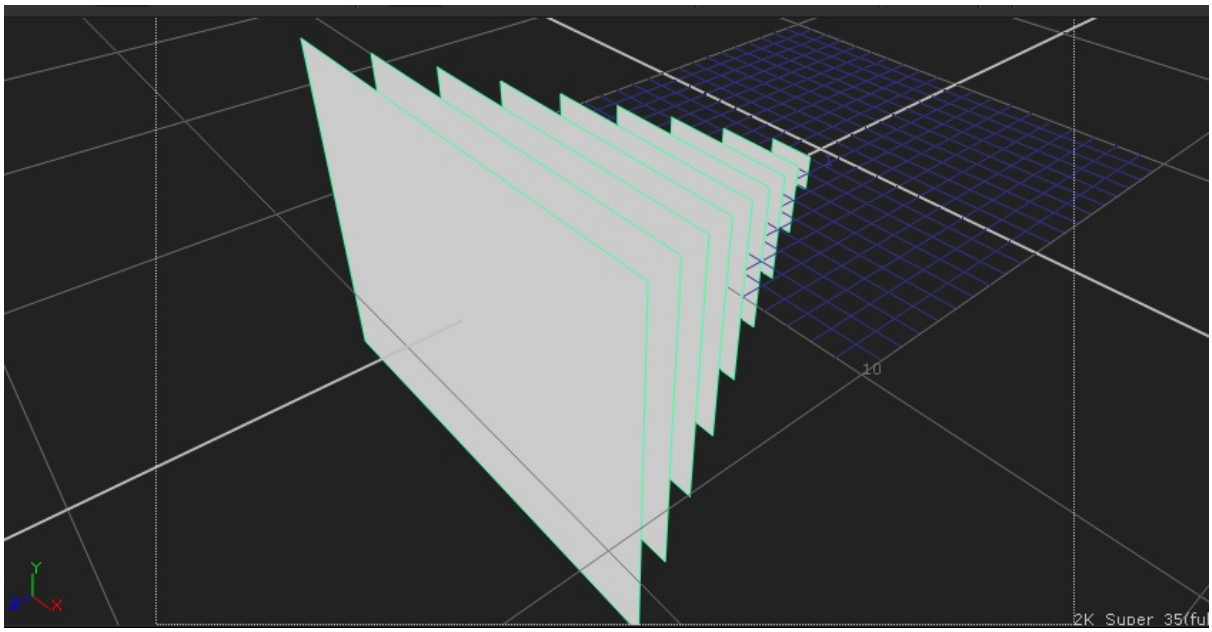
```
blurlist=[]
gradelist=[]
for ex in range(3):
    bl=nuke.createNode('Blur', inpanel=False)
    n=bl['name'].value()
    blurlist.append(n)
    gr=nuke.createNode('Grade', inpanel=False)
    gradelist.append(gr)
    tr=nuke.createNode('Transform', inpanel=False)['selected'].setValue(False)
print blurlist
print gradelist
```



Awesome! ...and very easy on creating 3 branches of nodes ! it also returns the name of the blur and grade nodes into your script window.
...but we want more!!! we are eager to learn even more than that! so let's move on into the 3D space and let's create 10 cards with increasing position over z axis and scaling:
Ok this is going to be a little more complicated but it is so just because we haven't covered yet ! :) - take a look at this:

```
amount=10
for s in range(amount):
    a= nuke.nodes.Card()
    pos= s * (amount/5)
    a['translate'].setValue([0,0,pos])
    a['uniform_scale'].setValue(pos)
```

Now in order to see the result, please manually create a Scene node and connect all your new cards and switch to the 3d view. and this is what you get:



ok I know this code might give you some headache...but let's not throw in the towel.
The first line define a value for a new variable I called **amount** and we are using this value to set the amount of the cards we want to create. The 4th line is the tough one ,or at least this is what it might looks like. Basically we are creating the new variable **pos** and we are adding increasing values to it...how ? since we are under a **range** loop it generates each time an increasing number starting from zero and the calculation would look like:

0 * (10/5) returns 0
1 * (10/5) returns 2
2 * (10/5) returns 4
3 * (10/5) returns 6 etc..

Where the first increasing column of values is coming from the new variable **s** that actually represents the range loop.

it means that each time the loop is repeated it assigns a different value to the **pos** variable and as a consequence the value in translation and scaling over the following lines will vary, hence the cards placement. Mystery solved !
Then if you want to read the exact value for each card you just **print pos**
and you'll have all the number that have been generated by the **range** command.

Wonderful stuff! But you might have realised that the very first card has a scaling value of zero, and it won't be visible in the 3D view at all, very useless actually, well this is happening because we have been using only one value into the range function, the number 10. And when just one value is provided the counting will always start from zero.So let's fix this up by adding something smart. In fact we could set proper minimum and maximum values into the Range function itself.

It is quite easy actually, let's just change the second line of the previous script (hence the loop starting) so to start like this:

```
for s in range(1,amount):
```

and so the entire script would be:

```
amount=10
for s in range(1,amount):
    a = nuke.nodes.Card()
    pos = s * (amount/5)
    a['translate'].setValue([0,0,pos])
    a['uniform_scale'].setValue(pos)
```

Ok now the first minimum value to use into our range calculation will be **1** and now it really makes sense.
Now one last thing with the **range** function: it is also possible to add a third value that stands for stepping. Basically let's say you write a range function like this and execute:

```
Test=range(1,50,5)
Print test
```

It will print values between 1 and 50 in step of 5, thus 5,10,15,20,25, etc.
Nice uh ? take some time to experiment!

now on executing the previous script you might have found it quite uncomfortable having to connect all your new cards to your Scene node ! fuckin' hell ! let's do it automatically with some more python lines !

```
amount=10
for s in range(amount):
    a = nuke.nodes.Card()
    pos = s * (amount/5)
    a['translate'].setValue([0,0,pos])
    a['uniform_scale'].setValue(pos)
    a['selected'].setValue(True)
    k=nuke.selectedNodes()
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
```

basically over the 7th line we are selecting the nodes we have been creating and saving this selection to a new variable **k** over the line 8.
On line 9 we are creating the Scene node and then over the followings lines we are connecting its input to all selected nodes. Nothing new actually, I'm just using a few simple lines we have already covered early on!

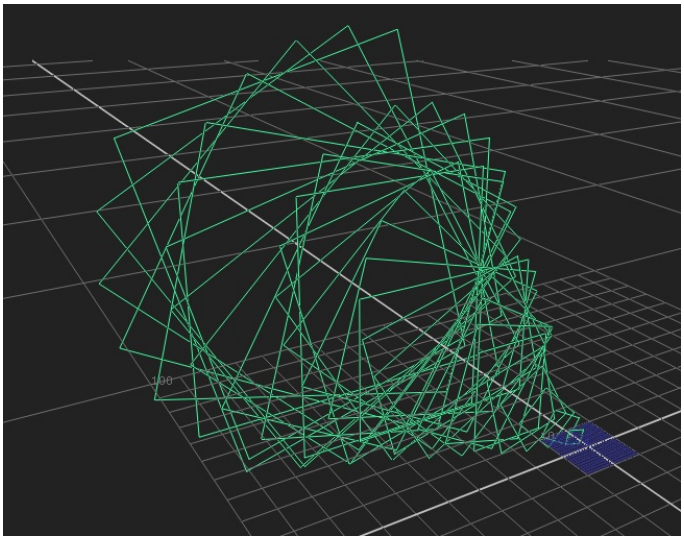
It is important to say that over the 4th line we should use floating values instead of integer values, especially when we are dealing with random values. That said our script should be looking like this:

```
amount=10
for s in range(amount):
    a= nuke.nodes.Card()
    pos= float( float(s) * float(amount/5) )
    a['translate'].setValue([0,0,pos])
    a['uniform_scale'].setValue(pos)
    a['selected'].setValue(True)
    k=nuke.selectedNodes()
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
```

Ok don't get scared I've just added the word float :-)
now let's continue with the card generator and shall add some more python line to perform a 15 degrees-increasing card rotation:

```
amount=30
for s in range(amount):
    a = nuke.nodes.Card()
    pos = s * (amount/5)
    rot= s*amount/2
    a['translate'].setValue([0,0,pos])
    a['uniform_scale'].setValue(pos)
    a['rotate'].setValue([0,0,rot])
    a['selected'].setValue(True)
    k=nuke.selectedNodes()
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
```

now if you look at the 3D view...surprise ! you'll find out 30 cards each one rotated by 15 degrees with the other. Please take some time to make out the calculation i've been using.



At this point you must be very excited with all the things you are learning ! ahahha
Now what follows is a nice procedure for the creation of 10 blur nodes connected to a Dot - each new blur will have an increasing value by 25.

```
dd=nuke.createNode('Dot')
ra=10
for ex in range(1,ra):
    bl=nuke.createNode('Blur', inpanel=False)
    bl['size'].setValue(ex * 25)
    bl['selected'].setValue(True)
    x=0
    bl.setInput(x,dd)
```

now what are you waiting for ? why not implementing a user's input window to specify the number of cards to create ? let's make it work:

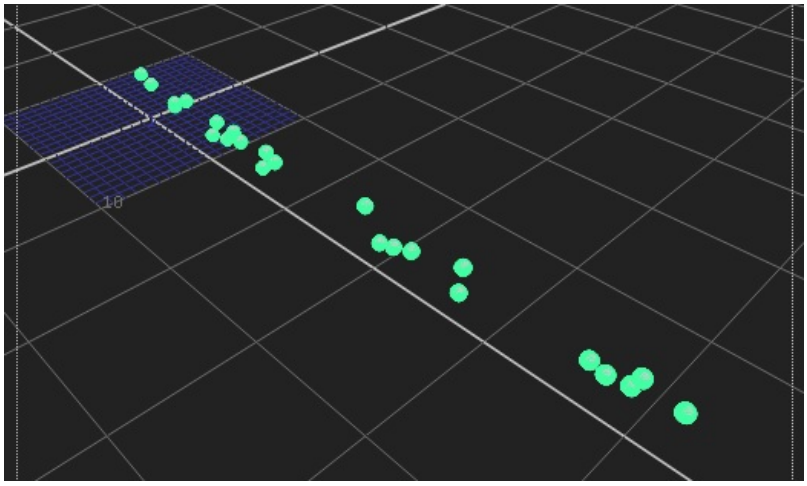
```
amount = int(nuke.getInput( 'Number of cards', 'type a number here' ))
for i in range(amount):
        a= nuke.nodes.Card()
        pos= float( float(i) * float(amount*2) )
        a['translate'].setValue([0,0,pos])
        a['uniform_scale'].setValue(pos)
        a['selected'].setValue(True)
        k=nuke.selectedNodes()
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
```

first thing to notice is at line 1, nothing crazy actually, I just put an **int** before opening my parenthesis because if you don't it will return with an error because he is expecting a float or

integer number and you are feeding him with a string ! that's it, no further hiddens secrets on this script !
ok I understand you must be quite shuttered at this point but before you pass out let's put some more funny stuff into the pan, don't worry it's going to be pleasing and don't forget! you always have time to revise all topics we covered.
Over the next snippet I'm going to import a new module called **random**. Basically it implements pseudo-random number generators that we might use for our convenience. There are many functions that are coming through with it and we will be using some of them throughout our future lessons but if you are curious you can find more by googling it.
Now let's make it work:

```
import random
amount = int(nuke.getInput( 'how many objects', 'type value' ))
for i in range(amount):
	a= nuke.nodes.Sphere()
	posx= float( float(i) * float(random.uniform(1,3)) )
	posy= float(random.uniform(2.5,5))
	a['translate'].setValue([posx,posy,0])
	a['uniform_scale'].setValue(0.5)
	a['selected'].setValue(True)
	k=nuke.selectedNodes()
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
```

execute it and when prompted with the user's input window type something like 25 for the objects you want to create, then please double click over your new Scene node and move to your 3D viewer, then frame it. What you are looking at will leave you flabbergasted ! your 3D space is now filled with 3D small spheres randomly scattered. how cool is it ?



Now take a look at the first orange-highlighted line; I've just declared variables and I'm running a **random.uniform** function within. The values I put within parenthesis are the maximum and minimum values, it means that it will generate floating random values within

that range. The result of this random generator is then multiplied by the **float(i)** value, that is the **range** outcome.
Now the second orange line is quite the same but here I'm just relying on a simple random generator. yayyyyyyyy !

Now the following code is basically the same but I'm just adding some more lines to store all my newly created nodes' names into a list and then print it into my script editor.

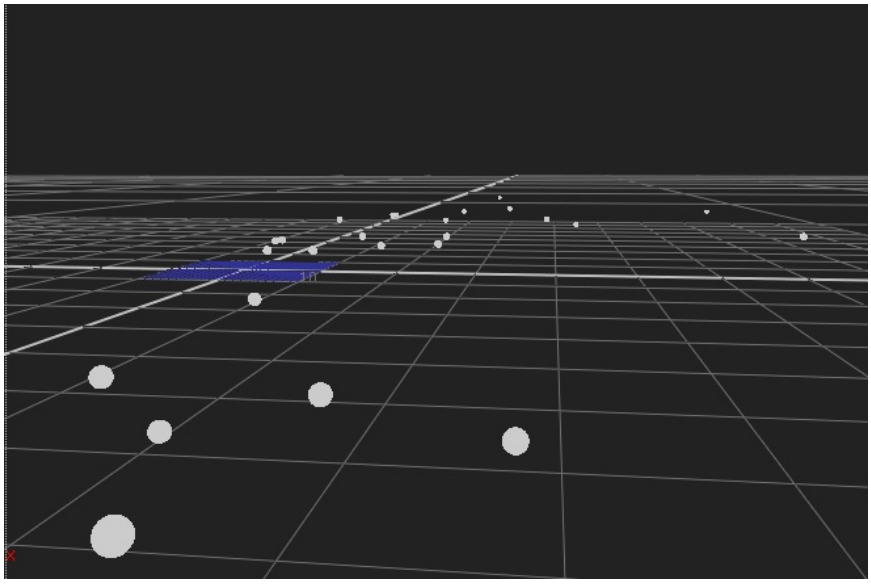
```
import random
nnn=[] # that's the new list name
amount = int(nuke.getInput( 'how many objects', 'type value' ))
for i in range(amount):
      a= nuke.nodes.Sphere()
      posx= float( float(i) * float(random.uniform(1,3)) )
      posy= float(random.uniform(2.5,5))
      a['translate'].setValue([posx,posy,0])
      a['uniform_scale'].setValue(0.5)
      a['selected'].setValue(True)
      k=nuke.selectedNodes()
for n in k:
    nnn.append(n.name())
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
print nnn
```

...and it works like a charm !
now that we have learned something more on **range** and **random** it comes the funniest part. Here I cooked an interesting snippet to make you smile ! In fact what follow is a script that you can use to create animations into your 3D space, like a traffic in the street...or at least this is what it is pretending to be. Try it and input a value around 35 within the user input window:.

```
import random
txt = nuke.getInput( 'how many objects', 'type value' )
lis=[]
for i in range(int(txt)):
      a= nuke.nodes.Sphere()
      posx= float( float(i) * float(random.uniform(1,3)) )
      posy= float(random.uniform(2.5,5))
      rndNum=float( float(i) * float(random.uniform(2,5)))
      a['translate'].setValue([posx,posy,0])
      a['translate'].setExpression('sin((frame+1000)/'+str(rndNum)+')'+'*'+str(rndNum*2),2)
      a['uniform_scale'].setValue(0.5)
      a['selected'].setValue(True) #or a.setSelected(True)
      k=nuke.selectedNodes()
b = nuke.createNode('Scene')
```

```
x=0
for i in k:
    b.setInput(x,i)
```

before the explanation just execute it, switch to the 3D view, hit play, sit back and enjoy the show.



Click here if you want to look at it playing !!
http://www.gianlucadentici.com/traffic_test.mp4

It is awesome isn't it ? well obviously this is something you could do even better by using particles, but I thought it was useful to show you how to deal with objects generation and random functions in Python.

Now the highlighted line is the beast and might cause disease and headache ahaha yeah it looks like a virus, but with some step-by-step explanation it turns out quite clear.
Basically what I have done is setting an expression that will sit into the Z translation knob.
In fact when you execute the code you'll see that each new sphere will have an expression stored in the Z translation knob. Now let's go deep into the syntax; for the sake of clarity I splitted it in 3 parts and the explanation:

'sin((frame+1000)/' this is creating a **sin** function and I'm using a value of 1000 and the reason is the final effects I want to get up to is related to the current frame number. Please note the **/** symbol right after the closing parenthesis, it is critical because we are going to divide this **sin** function by following values.

+str(rndNum)+')' here I placed a **+** because I need to extend my expression by adding other values, and in fact what follows is the outcome of the random variable **rndNum** - and as you can see it needs to be in string format.
Further to this I'm adding a closing parenthesis **')'**

+'*'+str(rndNum*2),2) here I'm adding a multiplication symbol **+'*'** and then I put the

result of **rndNum** multiplied by 2. In the end I added a comma followed by the number 2, that is the second value for the **sin** function.

Therefore one of your sphere's translate Z expression might look like this:
sin((frame+1000)/82.9601824899)*165.92036498

now what follows is the same code, I'm just using defined animation values. As you executed the code you will see blobby spheres and movements. Once again enjoy the show !

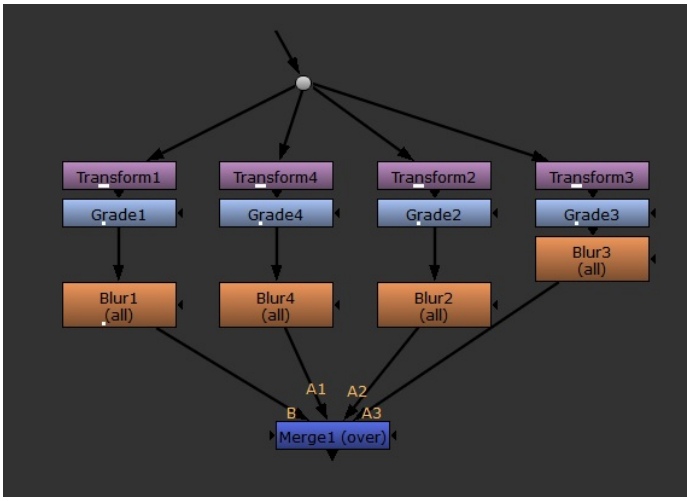
```
import random
txt = nuke.getInput( 'how many objects', 'type value' )
lis=[]
for i in range(int(txt)):
    a= nuke.nodes.Sphere()
    posx= float( float(i) * float(random.uniform(1,3)) )
    posy= float(random.uniform(2.5,5))
    rndNum=float( float(i) * float(random.uniform(2,5)))
    a['translate'].setValue([posx,posy,0])
    a['translate'].setExpression('sin((frame+1000)/'+str(rndNum)
+')'+'*'+str(rndNum*2),2)
    a['uniform_scale'].setAnimated(0) #triggering the animation here
    a['uniform_scale'].setValueAt(random.random()*i,1) # adding a value for the frame 1
    a['uniform_scale'].setValueAt(random.uniform(1,2)*i,50)
    a['uniform_scale'].setValueAt(random.random()*i,100)
    a['selected'].setValue(True) #or a.setSelected(True)
    k=nuke.selectedNodes()
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
```

Ok, setting keys is something we have already covered, so I'm just putting that at work.
Over the first highlighted line I'm telling Nuke to set up the animation for the Uniform Scale knob, then throughout the following lines I'm putting values respectively at frame 1, 50 and 100.
Obviously don't forget to properly set your own timeline to a framerange where these keys sits before playing.

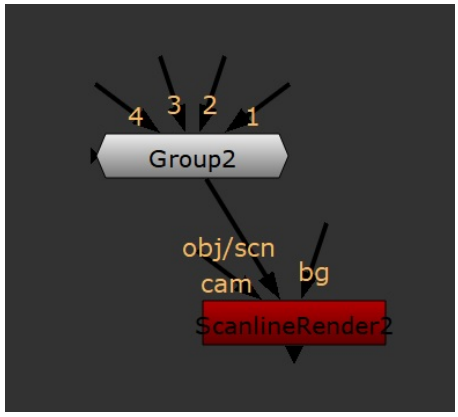
We are done for this lesson 4! ...I know I know we have been covering some demanding code with it, but I'm sure you will survive! and, once again, take your time !!
So now before saying hello…guess what ? ... here you are your wonderful assignments:

HOMEWORKS:

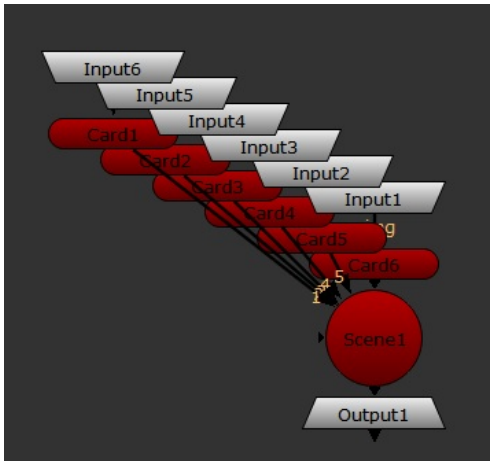
Assignment 1 : create four branches of nodes (1 Transform, 1 grade and 1 blur) and connect their inputs to a dot and their output to the A and B inputs of a merge node.
it should be looking like this:



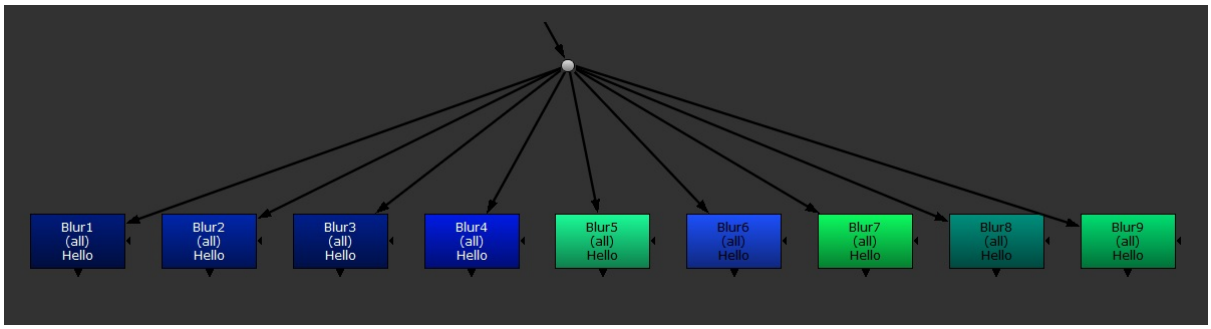
Assignment 2: create a script that ask the number of cards to create, then put them all into a new group, then connect its input to a scene node. In the end connect your group's output to a scanline render-
It should be looking like this:



and this is how it should be looking inside the group:



Assignment 3 : create an array of nodes using the range function and connect their inputs to a Dot. then change the label name for all of them and assign random colors and increasing blur values.
..and it should be lookin like this :



And that’s all lads ! Enjoy your fresh lesson, take your time to get familiar with all that stuff and let’s see ya soon on lesson 5 !!!! I love you all :) Cheers ! Bang !