Programming Nao using Python

In Choreographe there is a block named “Python Script” you can add to your project that you can program using python. It is also possible to view and edit the code of many of the other blocks. By looking at other blocks I was able to figure out a lot of useful features that I would like to outline so future Nao programmers can have a head start. I would like to discuss how to add custom parameters to a block, how to add input and output functions to blocks and how to properly call these functions.

Adding Custom Parameters to a Block and Referencing Them

When you create your own custom block you might want to set a few global parameters in the block that you can adjust without using code. I have only used this so far to create a Boolean parameter to turn a particular row of blocks on or off but most of the blocks in Choreographe have far more useful parameters such as the “Take Picture” block.

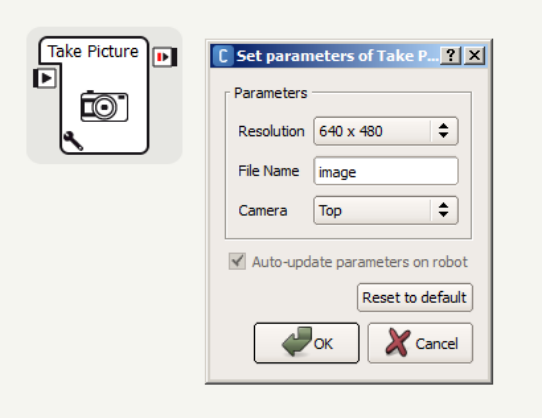

Figure 1: Take Picture Parameters

To add your own custom parameters first let’s make our own custom block. In the “Box libraries” section in Choreographe look for the “Templates” folder and drag a new “Python Script” into your project. **Right click the “Python Script” block and select “Edit Box”**. In the “Edit box” window **click the plus sign next to parameters** (figure 3).

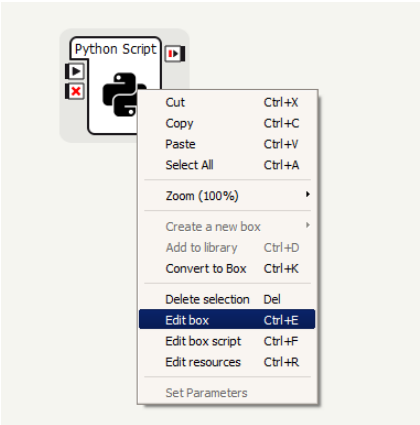

Figure 2: Editing a Block

In “Add a new parameter” you can name the parameter, give it a tooltip and choose its type. Options available for different types:

Boolean – default true or false
Integer – default, min value, max value
Float – default, min value, max value
String – multiple choices or custom strings, can be password (shows * instead of characters)
Attached file – allows files to be passed in directly as parameters

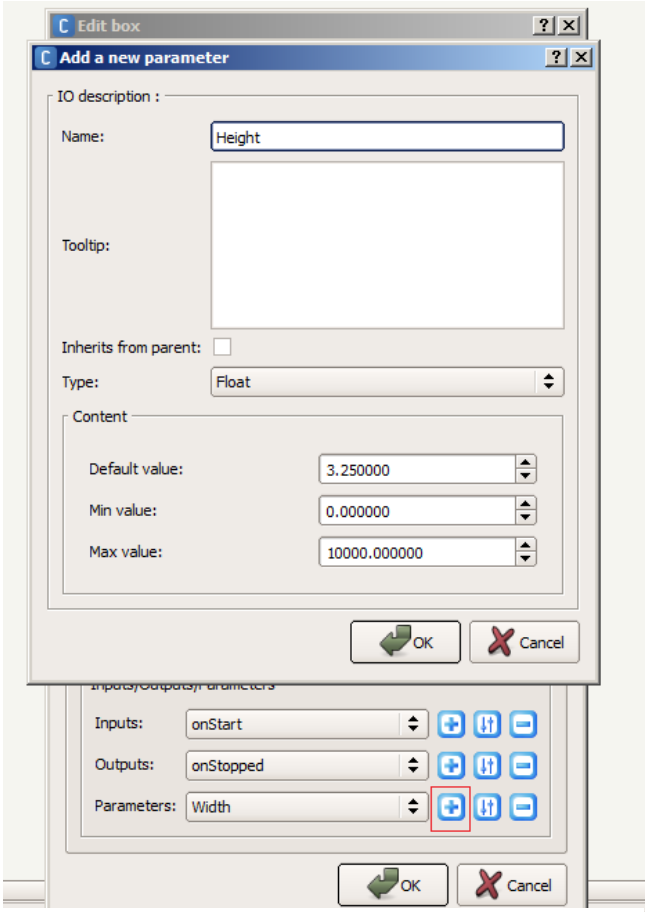

Figure 3: Adding A Parameter to a Block

In this tutorial I want you to **set up three new parameters Width, Length and Height** representing the width, length and height of a rectangular prism. I chose **min and max values of zero and 10,000** respectively for each parameter. In order to edit these parameters **click on the wrench on the new Python Script block**. I am going to use these three parameters to calculate the volume of a rectangular prism with those specifications. After I calculate the volume I will print it in the logs. **To reference a parameter in python code** you must use the `self.getParameter` function.

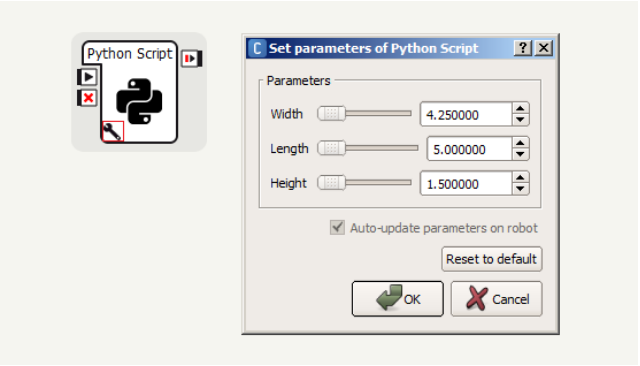

Figure 4: Setting Parameters

Adding the following code to the onInput_onStart function will retrieve the values of the three parameters, calculate the volume and print it to the log viewer.

```
w = self.getParameter("Width")
l = self.getParameter("Length")
h = self.getParameter("Height")

volume = w * l * h

print(volume)
```

Note: Print statements are only displayed in the log viewer if show all logs is checked on.

Adding Output Functions to a Block

Let's try to convert this module to output the Float volume instead of just printing it in the log so we can use it elsewhere in the project. First we must add an output to the python script. **Right click on the python scripts output and select "Add output"**. **Name your output appropriately and change the type at the bottom to "Number", ensure there is a 1 in the box to the right of the type menu**. After clicking ok our script should have a new output (figure 5).

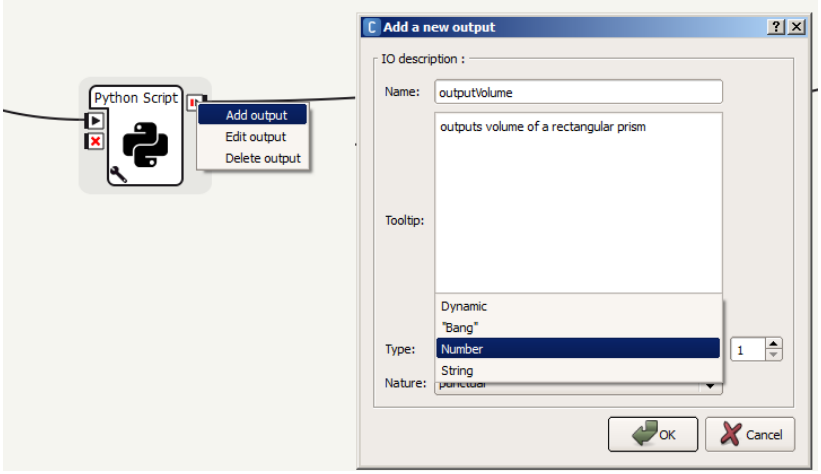

Figure 5: Adding New Output

I named mine **outputVolume** therefore when I want to call it in my code I execute the following command:

```
self.outputVolume(volume)
```

Adding Input Functions to a Block

Now that we have our first python script block calculating a volume and sending it out through **outputVolume** we need a second block to receive the data and use it. We are going to have a second block retrieve the Float volume and print it to the log viewer to keep things simple for now.

Drag a second Python Script into your project. I named mine Print Volume. **Right click the input side of the new block and select "Edit Input". Change the type from "Bang" to "Number" and ensure the value is 1 in the box next to the type drop down menu**. Click ok in "Edit existing input".

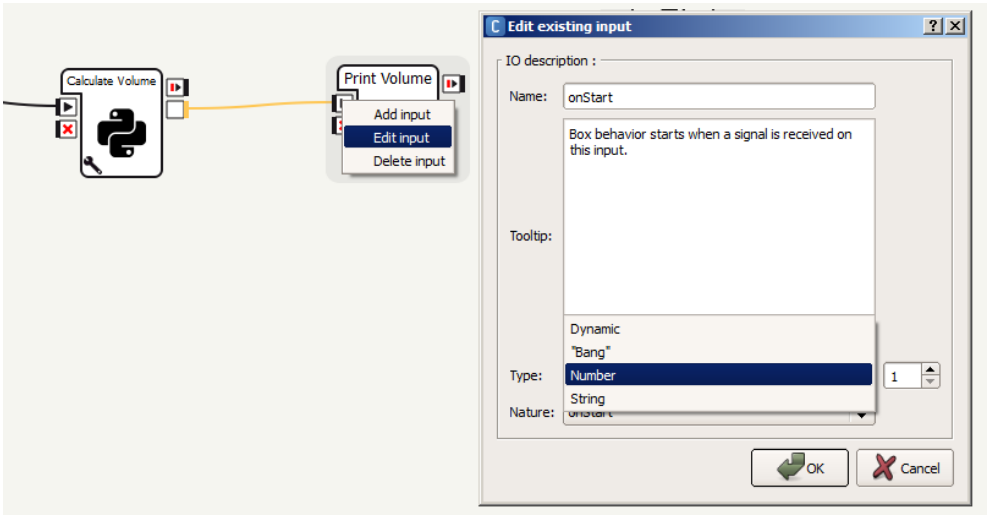

Figure 6: Changing existing input

Inside the new blocks script editor the onInput_onStart function should have added a second parameter `p` after `self`.

```
def onInput_onStart(self, p):
    #self.onStopped() #activate the output of the box
    pass
```

That is your parameter! To access it just use p.

```
13      def onInput_onStart(self, p):
14          print("The volume is " + str(p))
15
16          self.onStopped() #activate the output of the box
17          pass
18
19      def onInput_onStop(self):
20          self.onUnload() #it is recommended to reuse the clea
21          self.onStopped() #activate the output of the box
Ln 1
[INFO ] python :catchlog_logout:0 The volume is 31.875
Show all logs
```

Note: The input's name is **onStart** in the input edit windows (figure 6). In order to link an input to a function the function name must start with onInput_ to reference that it is an input. This is why the functions name is **onInput_onStart**.

The completed version of this tutorial is available on the disc, the project is called VolumeExample.

Breaking Blocks Into More Readable Code Using Inputs and Outputs

Our original project was to have Nao walk down a hall adjusting his movements based off his sonar readings to avoid walls and other obstacles. After a few weeks we had our first working project and it looked like this:

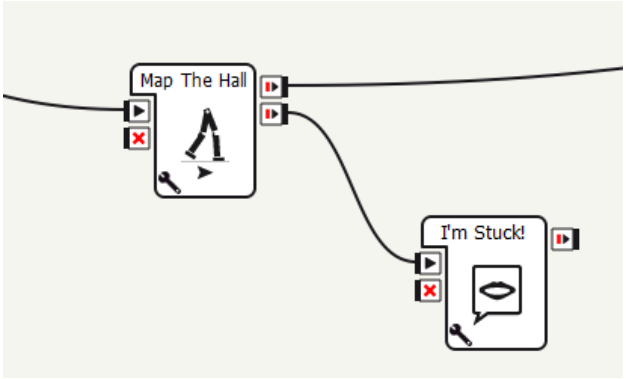

Figure 7: How NOT To Program

Although Nao avoided all the obstacles it was hard to show everyone what was going on since it was all essentially in one block. This was our algorithm:

1. Set up to walk straight ahead
2. Read in sonar values
3. If either left or right is too close, adjust next step towards opposite direction
4. Walk ahead with adjustments based on 3)
5. Call step 1 again

After seeing our algorithm it is easy to see what could have been separated into new blocks. Over the next month we worked on turning this project into something others could read and edit easily.

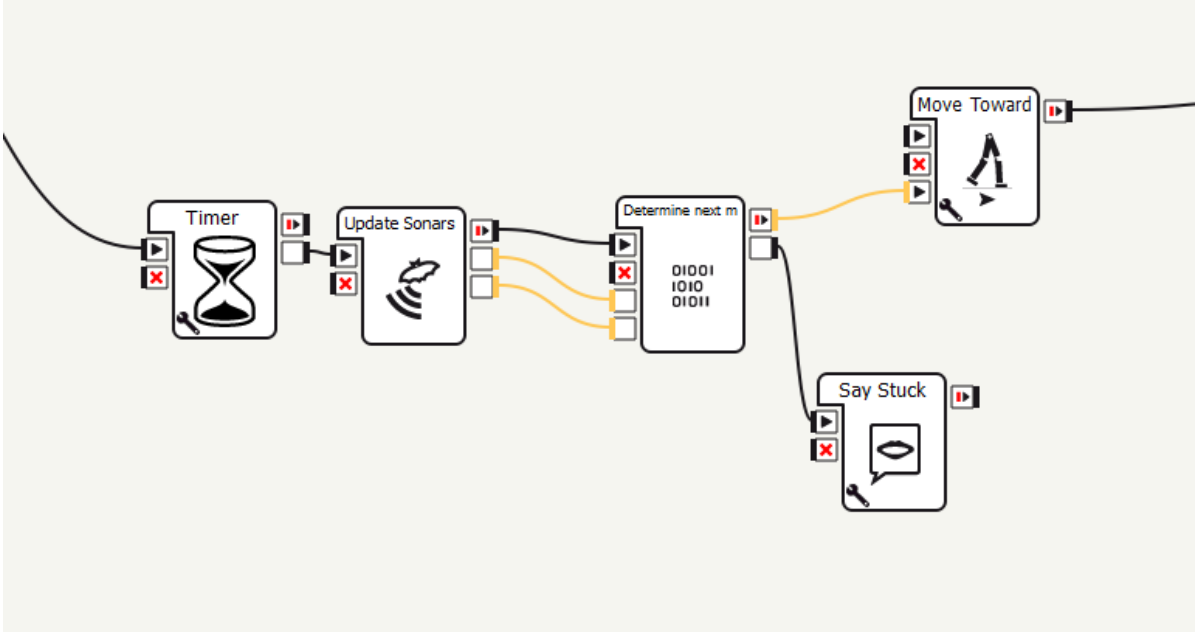

Figure 8: Splitting Up One Block into Many

In our new model a timer ticks every second. Every second the sonar is updated in "Update Sonars", Nao makes adjustments in "Determine next move" and the move is handled in the "Move Toward" block. In order to make our project function we had to send variables from block to block.

Our ability to split the block up came from being able to understand the input and output functions of the blocks. If you have done the tutorial above you already know how to add new inputs and outputs so I will not go over that again but I will include a quick reference guide to using input and output functions in Choreographe.

Sending single float, output name is updateLeft, python code:

```
leftSonar = 0.85
self.updateLeft(leftSonar)
```

Receiving single float, input name is updateLeftSonar, python code:

```
def onInput_updateLeftSonar(self, p):
	self.leftSonar = p
```

Sending three numbers, output name is onStopped, python code:

```
self.onStopped([1, 0, self.theta])
```

Note: The three values are in [,] which creates a new list of parameters. Even though the onStopped output takes three numbers they must be sent in a list as one object.

Receiving array of three numbers (a list object named p), input name is updateMovement, python code:

```
self.onInput_updateMovement(self, p):
	self.x = p[0]
	self.y = p[1]
	self.theta = p[2]
```

Examples of all these interactions can be seen in the project named MotionProject on the disc.

Below are findings from Tyler and Curtis on their respective areas of the group.

Tyler

I have worked extensively with Nao's networking capabilities. The Choreograph software comes with a built in drag and drop module that allows for network addresses, such as websites, to be fetched. It does this by using the python urllib and urllib2 module. These modules take care of a lot of the behind the scenes reading/writing networking protocols and are really only useful for opening up a webpage and reading its contents. For our project we wanted to give Nao the ability to tweet out pictures from his camera with custom pre-defined messages. To do so we would need a lot more than the built in Choreograph module.

What we decide on was to construct a 2-tiered networking architecture, where Nao would formulate the twitter data, upload this data to a server (on hopper.wlu.ca) which would have a PHP script with the appropriate twitter authentication API libraries to then send off to twitter's servers. We choose this design because Nao didn't have the capable cryptography python libraries to authenticate using twitters API directly. So we constructed a simple PHP script with a twitter API library that would take a multi-form POST request which included the picture data, and the message to tweet out.

Next we have Nao take a picture using its camera. This picture is stored on Nao's local storage drive as a jpg. We would open this file up using python and read in the raw binary data. After selecting a pre-defined message from Naos head tactile sensors we now had the string and picture data ready to tweet. To upload this to the server we used pythons socket module, which allowed us open up a direct connection to the web server and send a multi-part form encoded POST request that included the picture data as well as the string to tweet. The python socket module has the most low level networking functions available so we were able to perform such a complicated HTTP request. We learned that any serious networking tasks should be used by this module, as it allows you to craft very specific requests, especially when working with the HTTP protocol.

In the case of an error, or an unreachable network address, we included a simple try-catch conditional when trying to first connect using the the socket module. In the case of an error our tweet module would exit and Nao would announce out loud what the error message was.

Given Nao's networking abilities there are many exciting possibilities and cool features to incorporate. Nao's networking is done entirely through pythons networking libraries (and there are quite a few pick from). Given our experience working with Nao's networking abilities, we found it best to

use the socket python module for transmitting custom data between servers. We've learned a lot about Nao's networking capabilities and the python libraries that drive them. We've also learned how to structure and develop a multi-tied server architecture that transmits and processes data amongst several servers. We've also learned the workings of the Twitter API and all the cool and interesting things that can be done with it.

Curtis Smale

The standard behaviors of NAO are relatively unimpressive, but when put together and modified can achieve interesting forms of greatness. The *Walk Toward* command is somewhat boring with only the ability to affect the initial parameters, but by combining it with the use of the sonars NAO can achieve an interesting sort of maneuvers. The first we attempted was a simple *Obstacle Avoidance* and after that a modified version of the *Red Ball Tracker* using this concept.

The *Obstacle Avoidance* concept had origins in connecting with NAO's *ALMemory* function in order to retrieve data from NAO. Every single aspect of his body has some sort of memory to it from the angles of every motor to the distance between him and an object as sensed by his sonar. The *ALMemory* is the critical connection that was needed to retrieve this data. After that a simple mathematical operator was used to determine if either sensor was triggered as being 'too close' or if both were.

(leftSonar < 0.75 and leftSonar !=0) or
(rightSonar < 0.75 and rightSonar !=0)
(leftSonar < rightSonar)
(leftSonar < 0.25 and rightSonar < 0.25)

The first and second were used to see if either sensor was 'too close' and the third to see which was closer to a wall/object. The last was used to determine if NAO was too close to something with both sensors and therefore stuck. The numbers here are represented in meters. Depening on which trigger occured, the variable *theta* was adjusted by *1 - distance to wall* in order to have a scaled direction change that was not jagged. The *theta* variable was adjusted to the direction that would soonest veer away from obstruction.

An added feature to this that was not added but could be would be a simple 'too far' condition for one (or both) sides so that NAO would not travel too far away from obstruction. This would enable him to walk around the room or a table by holding along a path within a certain distance from the room's wall or table.

The *Red Ball Tracker* was significantly more complicated to operate since the movement of NAO was performed through location updates rather than movement commands. Thus, NAO could not be properly controlled from within the function. Instead, but using external modules it was possible to trigger the *Red Ball Tracker* and the *Obstacle Avoidance* functions separately. The sonar readings were taken and used to determine which of the two modules to make use of. The *Red Ball Tracker* could not be pinged repeatedly without errors so a j/k switch style variable was used to prevent the *Red Ball Tracker* from being updated more than once between times the *Obstacle Avoidance* was update.

Prior to these realization there was some attempt at performing a sort of manual *Red Ball Tracker* within the *Obstacle Avoidance* module. This required the use of *ALMemory* again and making use of the *HeadYaw* and *HeadPitch* data. The idea would be to use the *Red Ball Tracker* to lock onto the ball as an observation alone, and to use the current facial position of NAO to determine the new *theta* angle for him to travel in order to get to the ball (assuming there was no obstruction). This did not work out well and the idea was abandoned.

Use of the *ALMemory* was rather simple. A connection was established with the *ALMemory* and specific 'memories' were extracted. The two sonars were the ones most frequently used and the *HeadYaw and HeadPitch* were used for a while until the project was abandoned for a better function that did not require them. All the data gathered from *ALMemory* was used immediately upon calling the function and then the data was discarded for new data during the next module update. A timer module was used to set the frequency of update to have a nicely controlled function prior to the movement modules. Timers set too frequent could cause rushed math or for one update to overlap with another if it was not finished yet and leave NAO confused. Timers set not frequent enough could result in a failure for NAO to react quickly enough to obstruction.