Addis Ababa Institute of Technology

TARMAC

Distributed Nodal Processing System Framework for Python

Learning the Basics

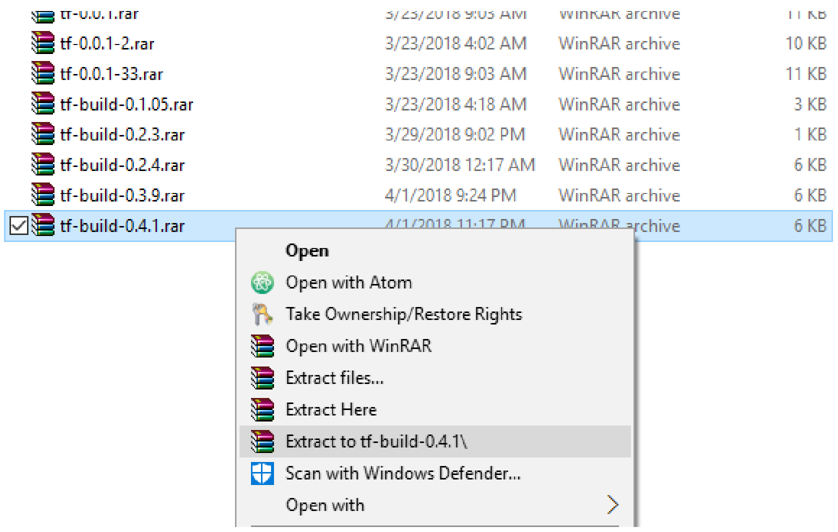
Samuel Sisay

Introduction

The Tarmac framework is a lightweight DINOPS framework built in python. It allows the creation of DINOPS applications with minimal concern for architecture implementation, letting the developer focus more on the functionality of the application by handling most of the architectural requirements. Despite being in its early stages, at the time of writing the framework already supports simplified node and cluster creation and handles cluster communication. In addition, since it has very simple concepts and its python usage allows developers to immediately get to work without having to take much time to learn the framework.

Installation

The tarmac framework comes in a compressed file that contains the dependencies for your DINOPS application. In order to begin working with tarmac, you first need to get the latest build of the framework and extract it onto your project folder.

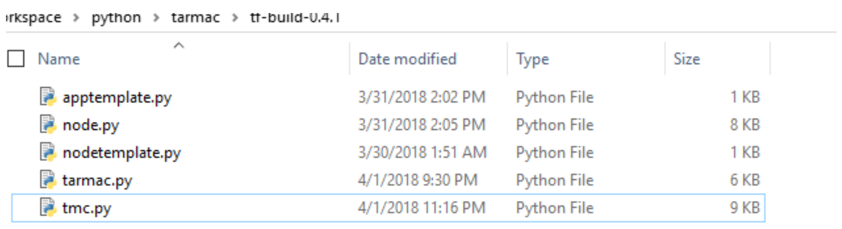


tf-0.0.1.rar	3/23/2018 9:03 AM	WinRAR archive	11 KB
tf-0.0.1-2.rar	3/23/2018 4:02 AM	WinRAR archive	10 KB
tf-0.0.1-33.rar	3/23/2018 9:03 AM	WinRAR archive	11 KB
tf-build-0.1.05.rar	3/23/2018 4:18 AM	WinRAR archive	3 KB
tf-build-0.2.3.rar	3/29/2018 9:02 PM	WinRAR archive	1 KB
tf-build-0.2.4.rar	3/30/2018 12:17 AM	WinRAR archive	6 KB
tf-build-0.3.9.rar	4/1/2018 9:24 PM	WinRAR archive	6 KB
tf-build-0.4.1.rar	4/1/2018 11:17 PM	WinRAR archive	6 KB

Open
Open with Atom
Take Ownership/Restore Rights
Open with WinRAR
Extract files...
Extract Here
Extract to tf-build-0.4.1\
Scan with Windows Defender...
Open with

Once you have the project files extracted to your folder, your project folder should look like this. Make sure it contains the files

- Node.py
- Nodetemplate.py
- Apptemplate.py
- Tmc.py
- Tarmac.py



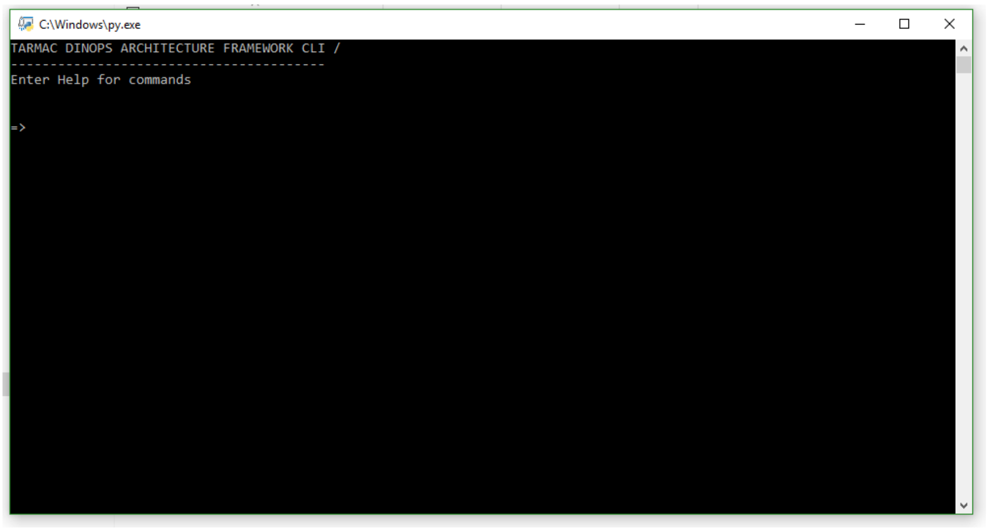
rkspace › python › tarmac › tf-build-0.4.1

Name	Date modified	Type	Size
apptemplate.py	3/31/2018 2:02 PM	Python File	1 KB
node.py	3/31/2018 2:05 PM	Python File	8 KB
nodetemplate.py	3/30/2018 1:51 AM	Python File	1 KB
tarmac.py	4/1/2018 9:30 PM	Python File	6 KB
tmc.py	4/1/2018 11:16 PM	Python File	9 KB

Usage

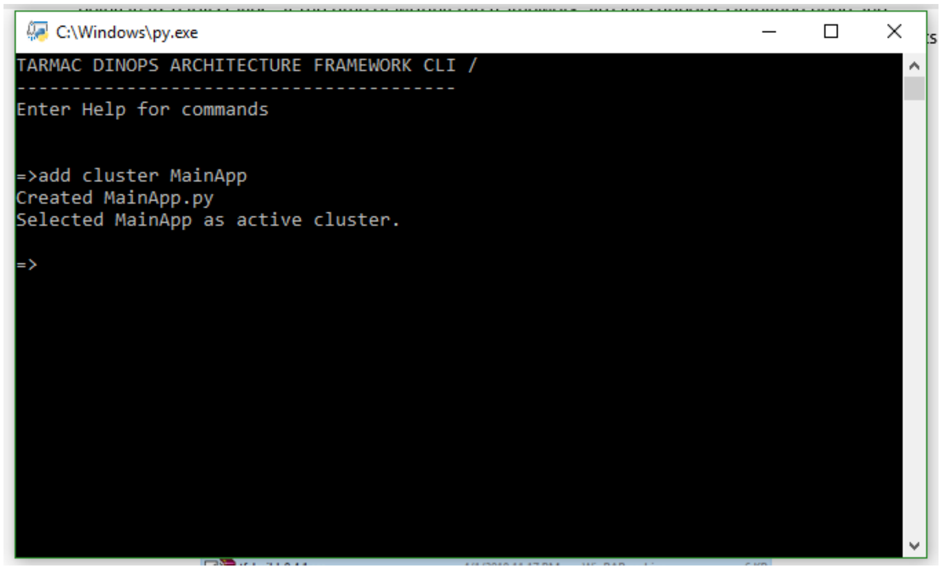
We are going to build a simple tarmac application that creates the string ‘Good Morning Hello World’ via nodes, in order to see how it works.

To build your application, you need to start with launching the tarmac CLI. Run the tmc.py script and the CLI should start running.



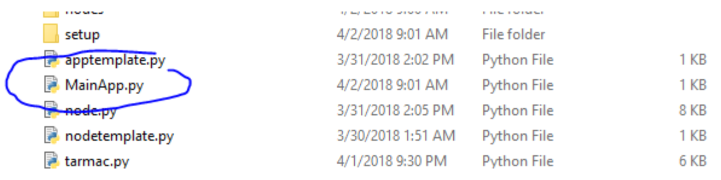
```
TARMAC DINOPS ARCHITECTURE FRAMEWORK CLI /
----------------------------------------
Enter Help for commands


=>
```

First of all, we need to initialize the application by creating one cluster for the current device to run. We can do that by using the command ‘add cluster [cluster name]’. This command adds a cluster to the project, and additionally checks if the project has been initialized. If it hasn’t already, it creates the folders ‘nodes’ and ‘setup’ which have functions we’ll see.



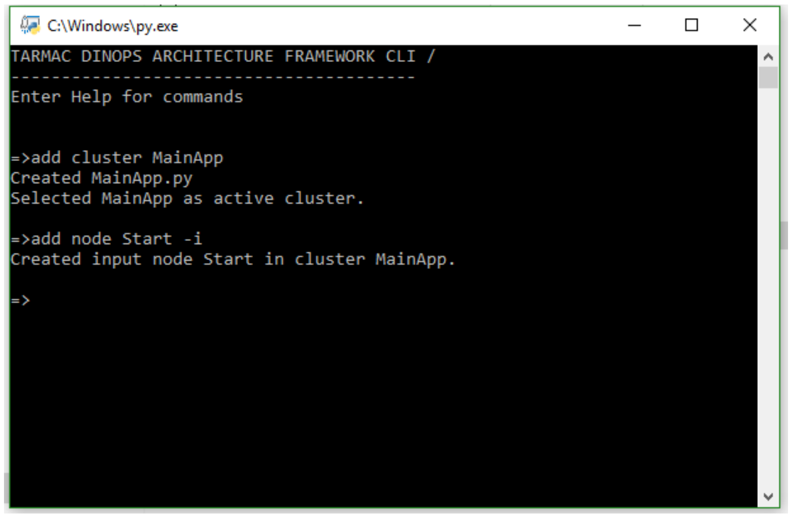
```
TARMAC DINOPS ARCHITECTURE FRAMEWORK CLI /
----------------------------------------
Enter Help for commands


=>add cluster MainApp
Created MainApp.py
Selected MainApp as active cluster.

=>
```

Now if we look in our project folder we can see that a python script by the name of our cluster has been created. This is where we write the initialization logic of our application or identify and pass in the data we want to stream through our node system.



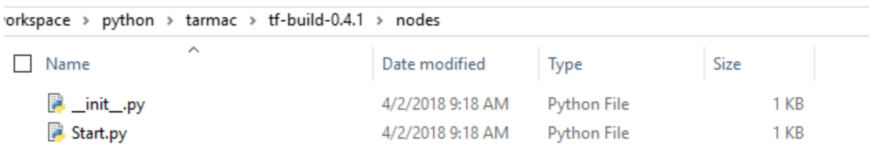
```
setup              4/2/2018 9:01 AM    File folder
apptemplate.py     3/31/2018 2:02 PM   Python File   1 KB
MainApp.py         4/2/2018 9:01 AM    Python File   1 KB
node.py            3/31/2018 2:05 PM   Python File   8 KB
nodetemplate.py    3/30/2018 1:51 AM   Python File   1 KB
tarmac.py          4/1/2018 9:30 PM    Python File   6 KB
```

So now we have a cluster, but no nodes in it. We need at least two nodes to make a functioning tarmac application. Go back to the CLI and enter the command ‘add node Start -i’. This command adds a node by the name of start to our application, and the ‘-i’ declares the node as an input node, or a node that is eligible for data entry through our cluster.



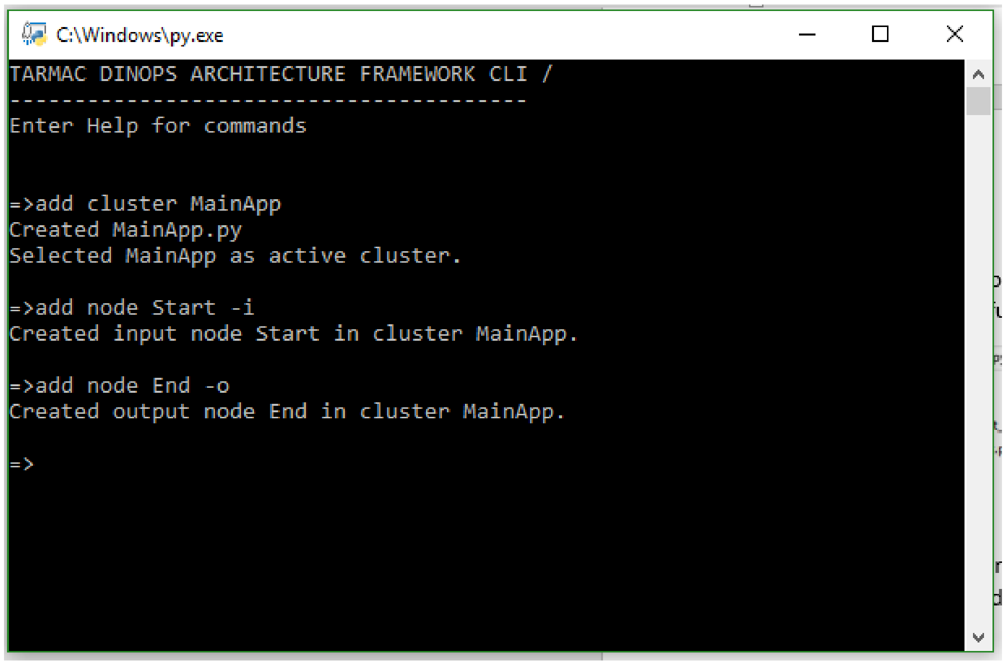
```
TARMAC DINOPS ARCHITECTURE FRAMEWORK CLI /
----------------------------------------
Enter Help for commands


=>add cluster MainApp
Created MainApp.py
Selected MainApp as active cluster.

=>add node Start -i
Created input node Start in cluster MainApp.

=>
```

If we look in the nodes folder we can see that a file named Start.py has been created. The python scripts in which we specify the functions of our nodes will be created in this folder.



Name	Date modified	Type	Size
__init__.py	4/2/2018 9:18 AM	Python File	1 KB
Start.py	4/2/2018 9:18 AM	Python File	1 KB

We also need an output node. Enter the command ‘add node End -o’ and another python script will be created with the title End.py. This is the node from which the cluster retrieves data, and the ‘-o’ flags it as an output node.

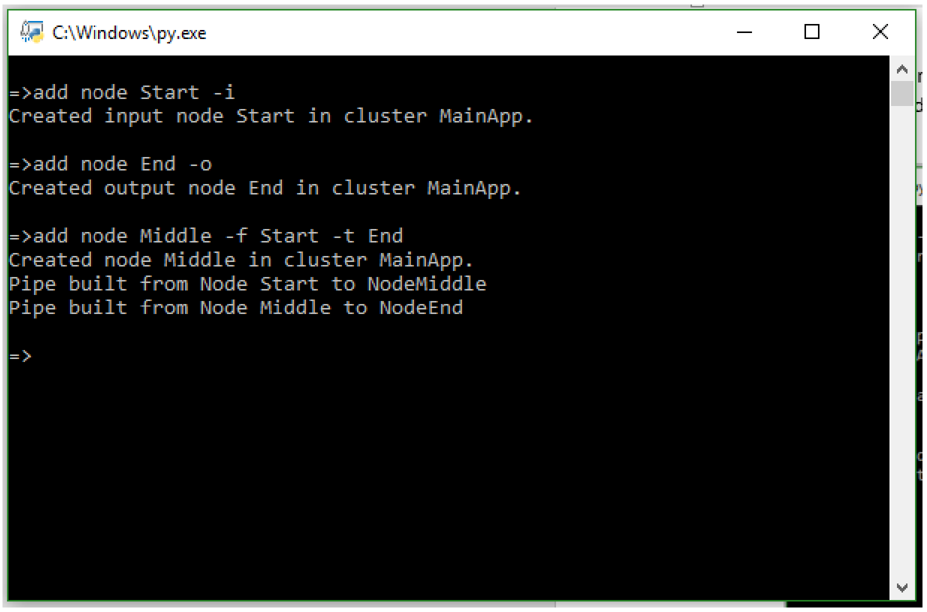


C:\Windows\py.exe

```
TARMAC DINOPS ARCHITECTURE FRAMEWORK CLI /
----------------------------------------
Enter Help for commands


=>add cluster MainApp
Created MainApp.py
Selected MainApp as active cluster.

=>add node Start -i
Created input node Start in cluster MainApp.

=>add node End -o
Created output node End in cluster MainApp.

=>
```

However, two nodes is still too minimal to display a proper DINOPS application with, so let us add one node that serves as a middle point between these two nodes.

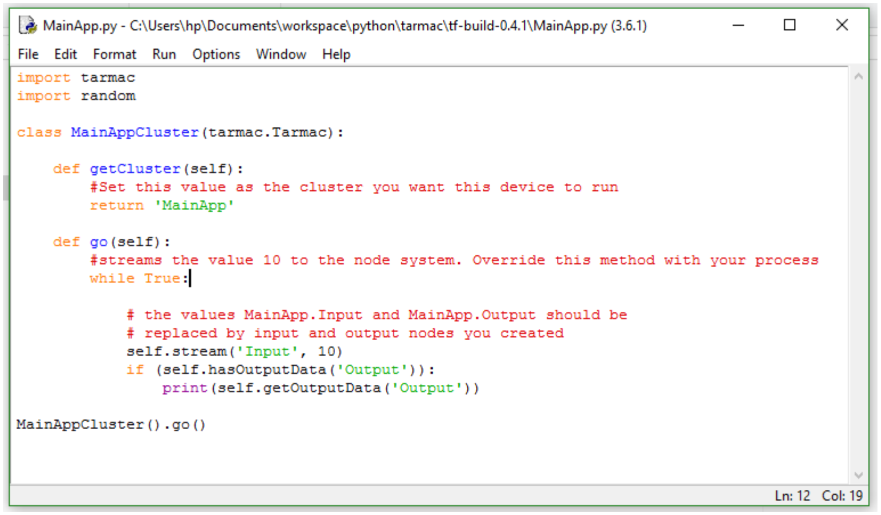
Entering the command ‘add node Middle -f Start -t End’ creates a node mapped between the start and end nodes. The tag ‘-f Start’ signifies that the node should receive data *from* the node Start, while ‘-t End’ notifies the End node that it should receive data from the new Middle Node, or that the Middle node should pass data *to* the End node.



C:\Windows\py.exe

```
=>add node Start -i
Created input node Start in cluster MainApp.

=>add node End -o
Created output node End in cluster MainApp.

=>add node Middle -f Start -t End
Created node Middle in cluster MainApp.
Pipe built from Node Start to NodeMiddle
Pipe built from Node Middle to NodeEnd

=>
```

Once nodes have been put in place and connected, it is time to write the code inside the clusters. Beginning with the cluster, we open the Mainapp.py script and change the code within.


MainApp.py - C:\Users\hp\Documents\workspace\python\tarmac\tf-build-0.4.1\MainApp.py (3.6.1)
File Edit Format Run Options Window Help
Ln: 12 Col: 19

```
import tarmac
import random

class MainAppCluster(tarmac.Tarmac):

    def getCluster(self):
        #Set this value as the cluster you want this device to run
        return 'MainApp'

    def go(self):
        #streams the value 10 to the node system. Override this method with your process
        while True:

            # the values MainApp.Input and MainApp.Output should be
            # replaced by input and output nodes you created
            self.stream('Input', 10)
            if (self.hasOutputData('Output')):
                print(self.getOutputData('Output'))

MainAppCluster().go()
```

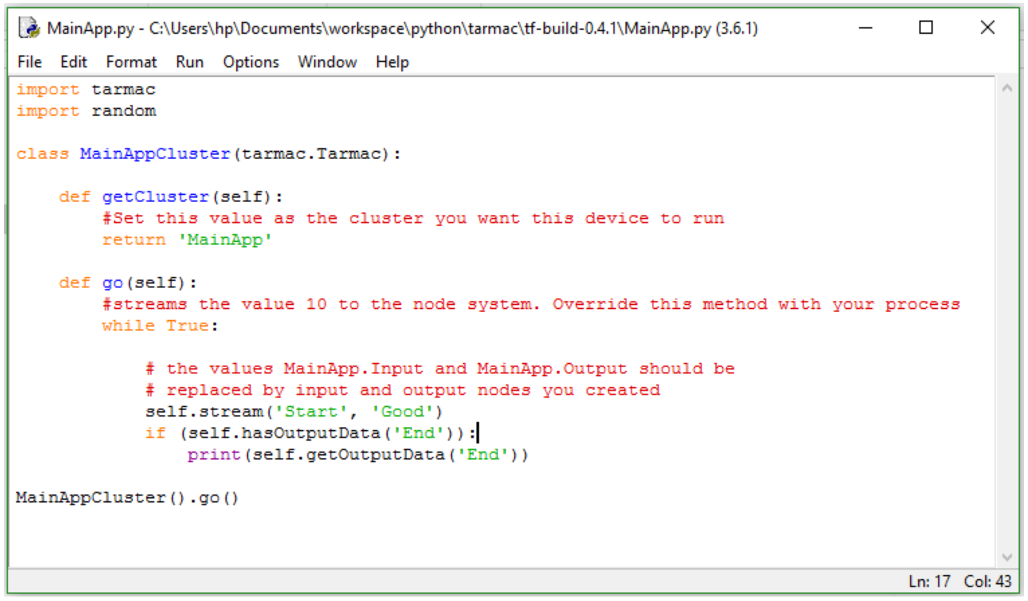
All clusters in a tarmac application inherit from the class Tarmac from the tarmac module. A cluster script should contain a such a class with two main methods overridden. The getCluster method should return the name of the cluster, and the go method should contain the system initialization and stream feed logic. Example statements are already provided, in which the function self.stream is called.

Self.stream passes data to the specified input node. In the example it is passing the number 10 to the node ‘Input’ continuously in an infinite loop.

hasOutputData checks if there is any data that has been processed through the node system and is awaiting pickup. Here it is checking the node ‘Output’.

getOutputData returns the data processed by the node system.

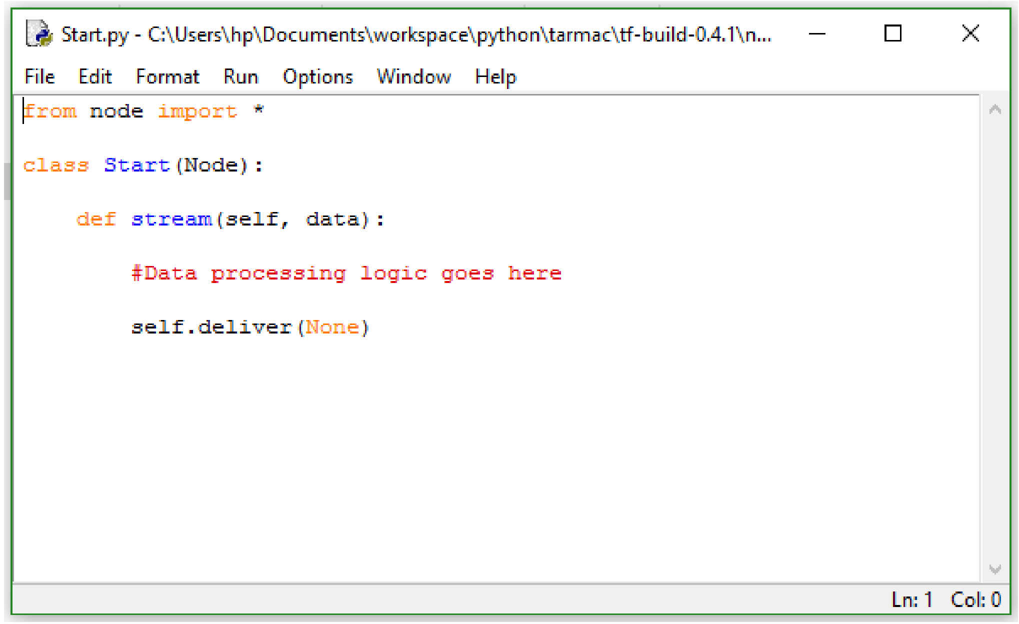
We can modify the above code so that we specify the input and output nodes as Start and End. Then we set the data to be passed into the node system to the string ‘Good’.


MainApp.py - C:\Users\hp\Documents\workspace\python\tarmac\tf-build-0.4.1\MainApp.py (3.6.1)
File Edit Format Run Options Window Help
Ln: 17 Col: 43

```
import tarmac
import random

class MainAppCluster(tarmac.Tarmac):

    def getCluster(self):
        #Set this value as the cluster you want this device to run
        return 'MainApp'

    def go(self):
        #streams the value 10 to the node system. Override this method with your process
        while True:

            # the values MainApp.Input and MainApp.Output should be
            # replaced by input and output nodes you created
            self.stream('Start', 'Good')
            if (self.hasOutputData('End')):
                print(self.getOutputData('End'))

MainAppCluster().go()
```

This scripts continuously enters the string ‘Good’ into the node system, checks if the system has supplied an output and then prints it. Now we edit the node scripts to retrieve and do some operations with this data.

Opening the start node we get:

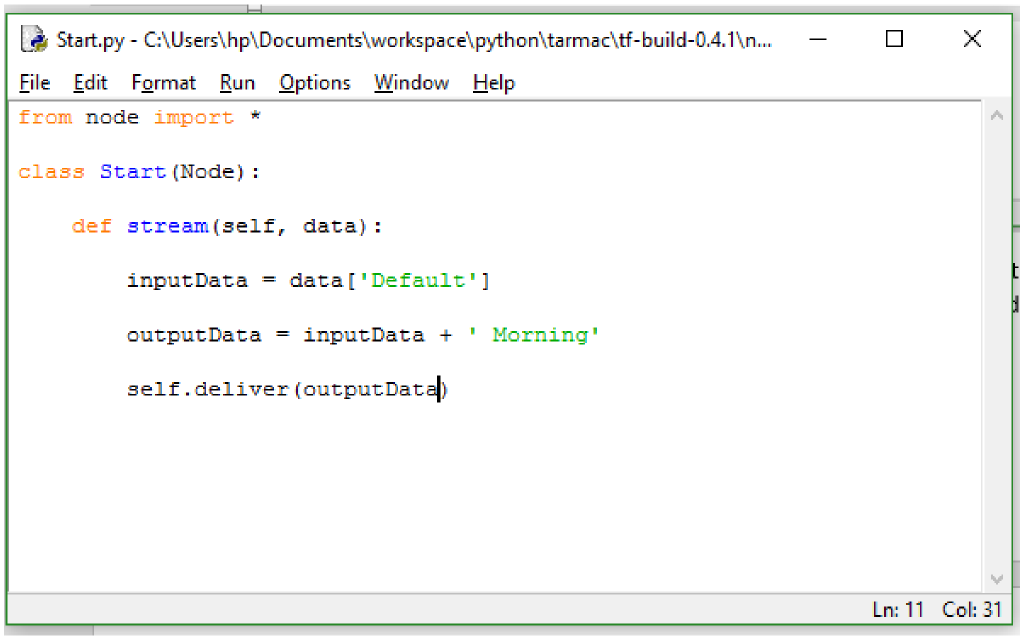

Start.py - C:\Users\hp\Documents\workspace\python\tarmac\tf-build-0.4.1\n...
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
from node import *

class Start(Node):

    def stream(self, data):

        #Data processing logic goes here

        self.deliver(None)
```

A node script contains class that inherits from the Node class and has the stream function overridden. The stream function is the function that is called when new data is received at the node, so the node logic is implemented here.

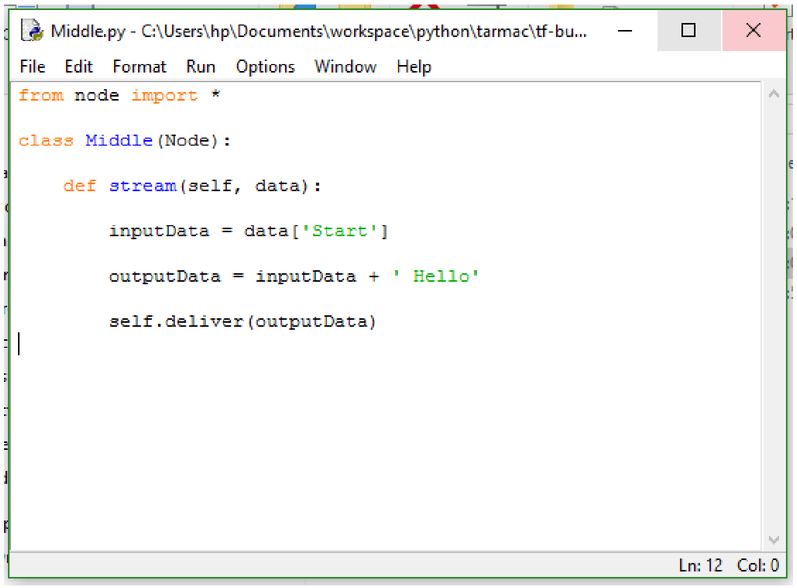
The parameter data is a dictionary of the node name to node data that is passed to the current node. Since this is an input node and the data supplied is not from a node, the input data is retrieved by the key 'Default'.

Once all the computation is completed, the results are passed through the function self.deliver(), which automatically passes the data to the nodes that are waiting for it.

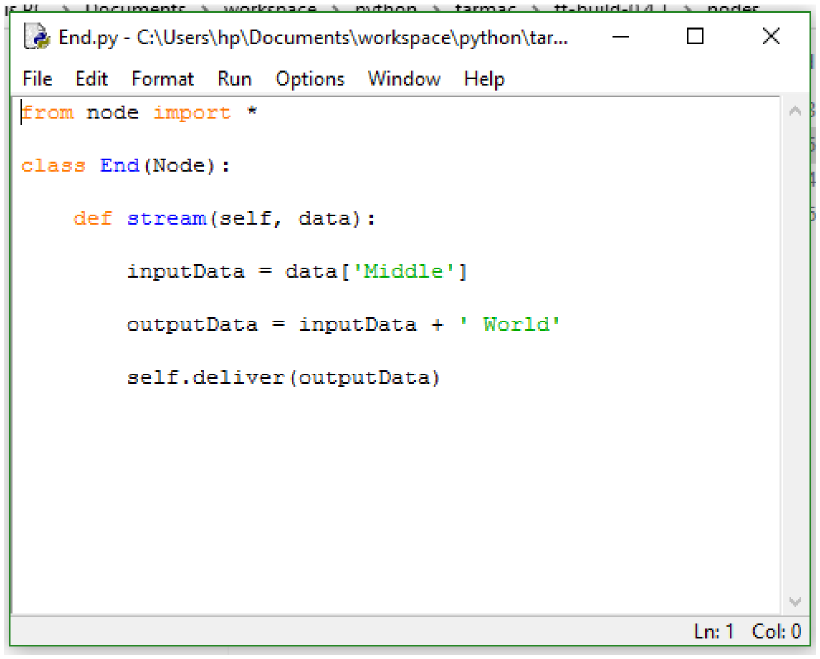

Start.py - C:\Users\hp\Documents\workspace\python\tarmac\tf-build-0.4.1\n...
File Edit Format Run Options Window Help
Ln: 11 Col: 31

```
from node import *

class Start(Node):

    def stream(self, data):

        inputData = data['Default']

        outputData = inputData + ' Morning'

        self.deliver(outputData)
```

Repeat this for the nodes Middle and End, for each node adding the word that comes next to build the final strings, each node receiving the data of the previous node in line, doing some processing and delivering the output.

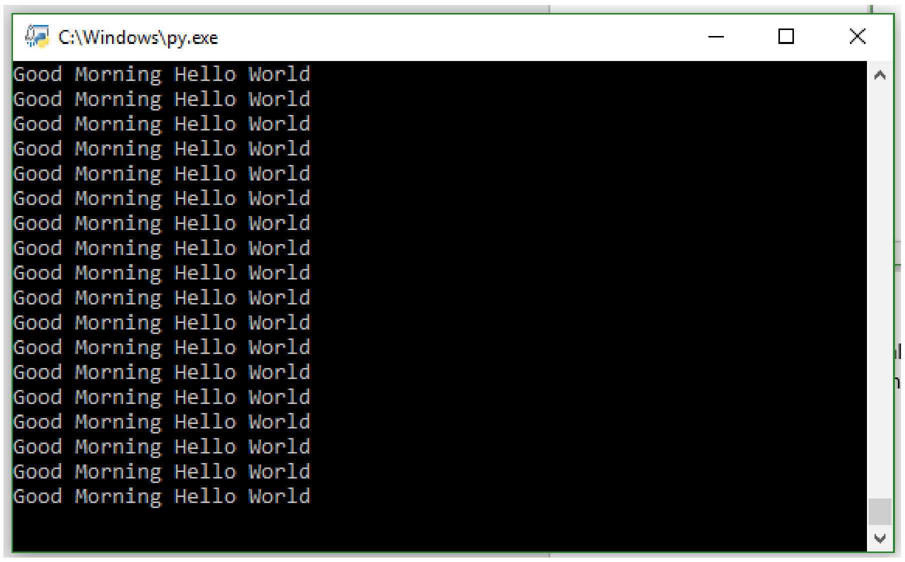

Middle.py - C:\Users\hp\Documents\workspace\python\tarmac\tf-bu...
File Edit Format Run Options Window Help
Ln: 12 Col: 0

```
from node import *

class Middle(Node):

    def stream(self, data):

        inputData = data['Start']

        outputData = inputData + ' Hello'

        self.deliver(outputData)
```


End.py - C:\Users\hp\Documents\workspace\python\tar...
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
from node import *

class End(Node):

    def stream(self, data):

        inputData = data['Middle']

        outputData = inputData + ' World'

        self.deliver(outputData)
```

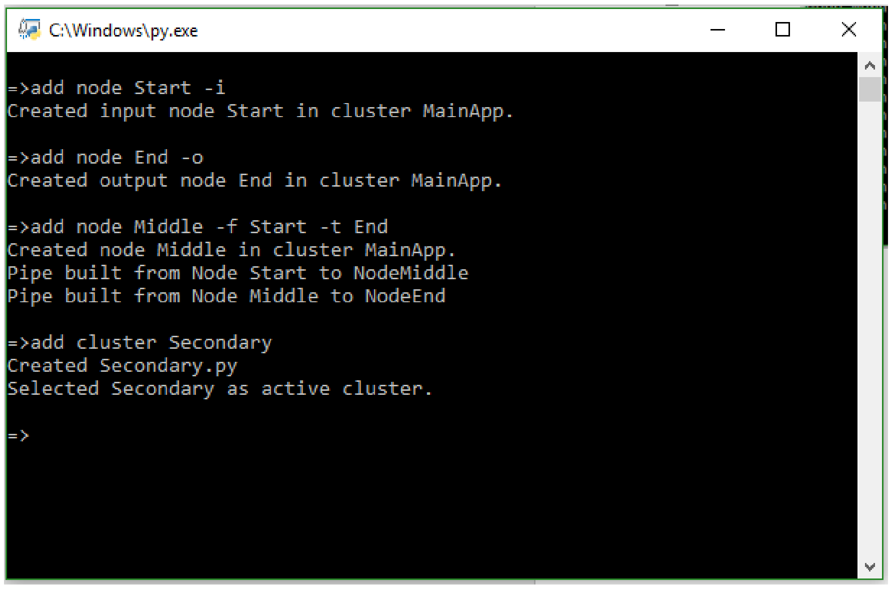
That is it. Our minimal hello world application is complete. If you now run MainApp.py, you will get this output, a display of the string that is being made every iteration of the system.



```
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
Good Morning Hello World
```

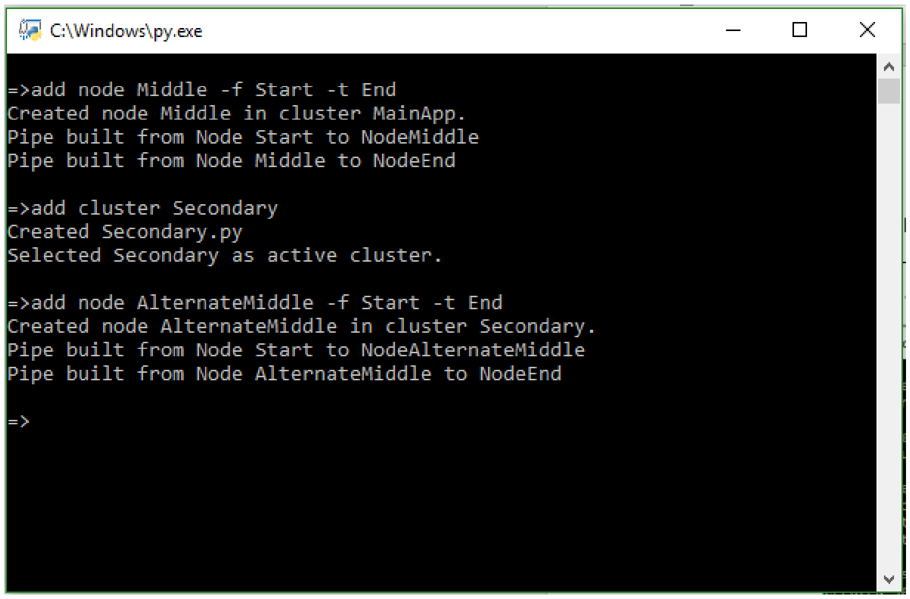
Node System With Multiple Clusters

Let’s add another cluster to our application. Go to the CLI and enter the command ‘add cluster Secondary’. This creates a new cluster script in the project folder.



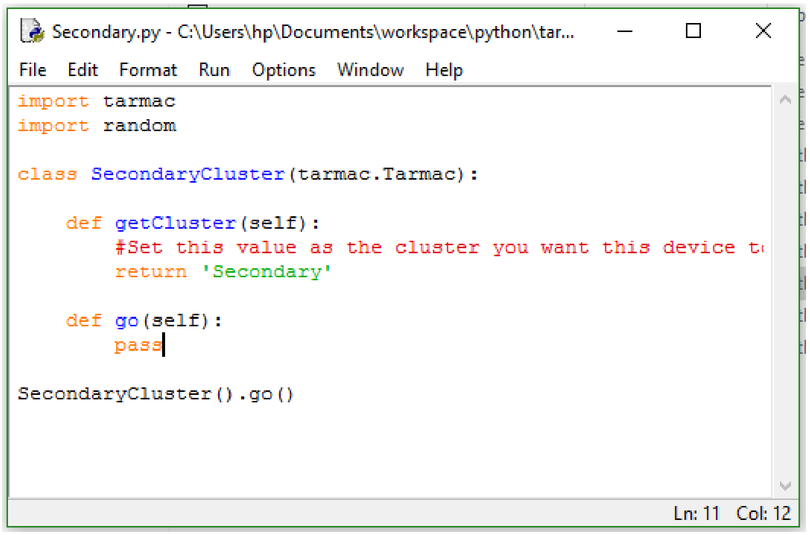
```
=>add node Start -i
Created input node Start in cluster MainApp.

=>add node End -o
Created output node End in cluster MainApp.

=>add node Middle -f Start -t End
Created node Middle in cluster MainApp.
Pipe built from Node Start to NodeMiddle
Pipe built from Node Middle to NodeEnd

=>add cluster Secondary
Created Secondary.py
Selected Secondary as active cluster.

=>
```

The CLI has the last created cluster in memory so that all added nodes will be added into the newly created cluster. So entering the command ‘add node AlternateMiddle -f Start -t End’ will create a node that is just like the middle node, but that exists on a different cluster.



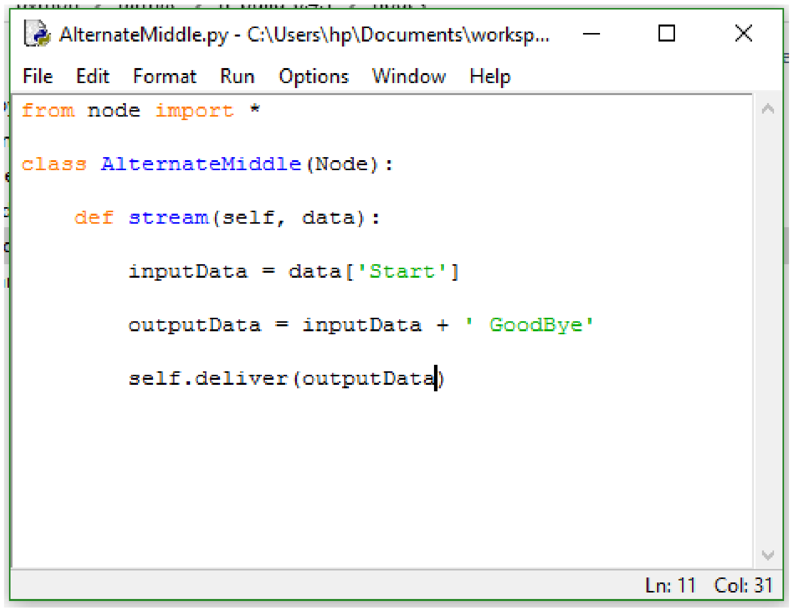
```
=>add node Middle -f Start -t End
Created node Middle in cluster MainApp.
Pipe built from Node Start to NodeMiddle
Pipe built from Node Middle to NodeEnd

=>add cluster Secondary
Created Secondary.py
Selected Secondary as active cluster.

=>add node AlternateMiddle -f Start -t End
Created node AlternateMiddle in cluster Secondary.
Pipe built from Node Start to NodeAlternateMiddle
Pipe built from Node AlternateMiddle to NodeEnd

=>
```

Our new cluster should not have any feeding logic added to it, which the framework automatically adds to the cluster script on creation as we have seen before. Edit the cluster script and clean out the go function, preferably with **pass**.


Secondary.py - C:\Users\hp\Documents\workspace\python\tar...
File Edit Format Run Options Window Help
Ln: 11 Col: 12

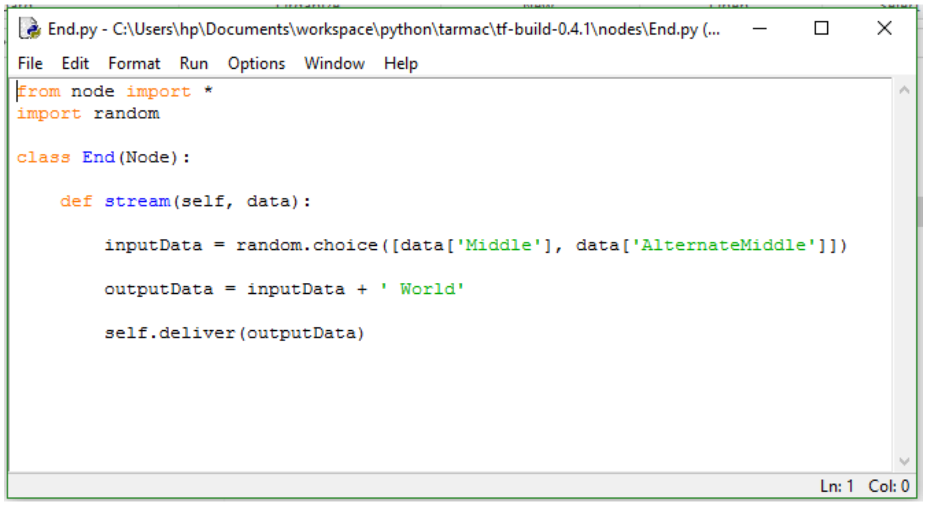
```
import tarmac
import random

class SecondaryCluster(tarmac.Tarmac):

    def getCluster(self):
        #Set this value as the cluster you want this device t
        return 'Secondary'

    def go(self):
        pass

SecondaryCluster().go()
```

Currently our AlternateMiddle function is routed, but has no logic added to it. Since it is receiving data from the Start node, it will be the string ‘Good Morning ’. Instead of adding ‘hello’ to the end, lets append ‘Goodbye’.


AlternateMiddle.py - C:\Users\hp\Documents\worksp...
File Edit Format Run Options Window Help
Ln: 11 Col: 31

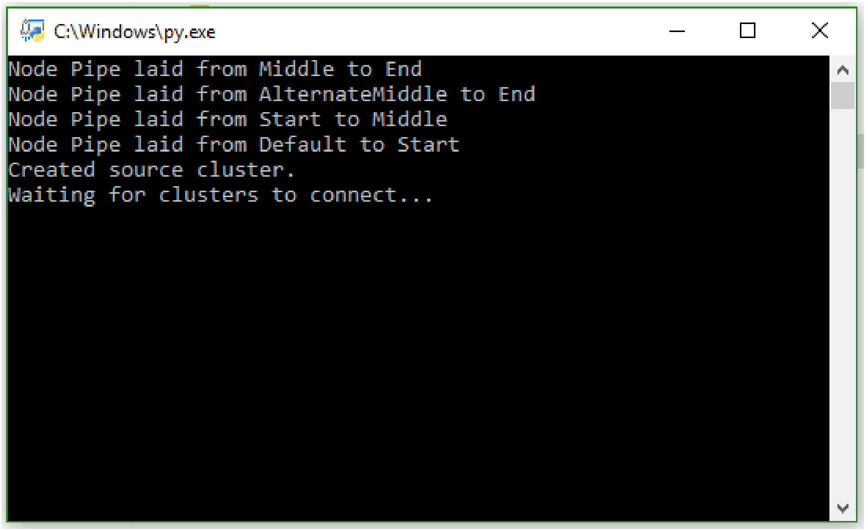
```
from node import *

class AlternateMiddle(Node):

    def stream(self, data):

        inputData = data['Start']

        outputData = inputData + ' GoodBye'

        self.deliver(outputData)
```

Now the End node in the MainApp cluster is set to receive data from two nodes, Middle and AlternateMiddle but it is only using the data from Middle. Let us make it so that it randomly selects one string between the two inputs and passes it on.


End.py - C:\Users\hp\Documents\workspace\python\tarmac\tf-build-0.4.1\nodes\End.py (...
File Edit Format Run Options Window Help
Ln: 1 Col: 0

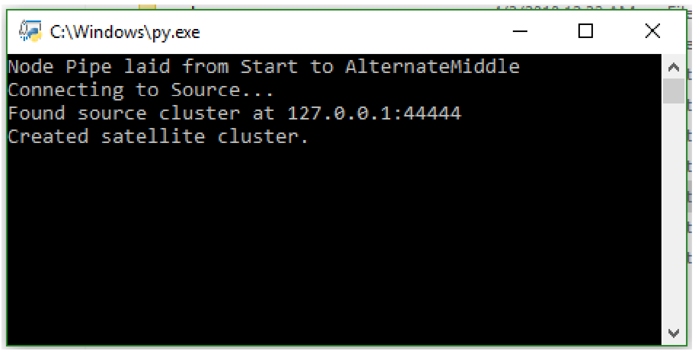
```python
from node import *
import random

class End(Node):

    def stream(self, data):

        inputData = random.choice([data['Middle'], data['AlternateMiddle']])

        outputData = inputData + ' World'

        self.deliver(outputData)
```

The nodes and clusters are complete. Launching the MainApp.py script now should give this window.



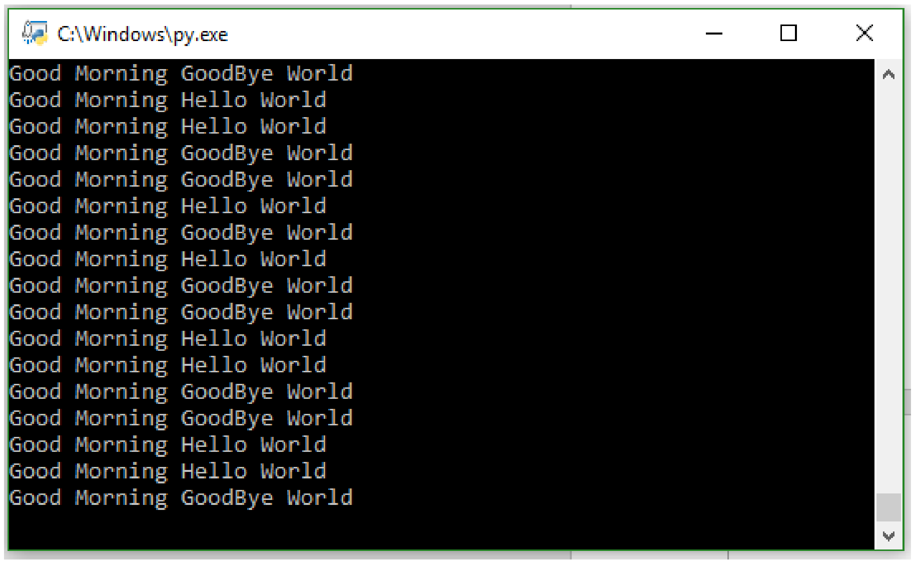
```
Node Pipe laid from Middle to End
Node Pipe laid from AlternateMiddle to End
Node Pipe laid from Start to Middle
Node Pipe laid from Default to Start
Created source cluster.
Waiting for clusters to connect...
```

That is because our application now is a multi-cluster application. The main cluster is waiting for the Secondary cluster to be launched so that the nodes that communicate with the nodes in Secondary can pass and receive their data. Launching the Secondary.py script should give this window.



```
Node Pipe laid from Start to AlternateMiddle
Connecting to Source...
Found source cluster at 127.0.0.1:44444
Created satellite cluster.
```

Meanwhile the MainApp.py should sense the availability of the cluster and start the computations. As you can see it is randomly displaying between 'Good Morning Hello World' and 'Good Morning Goodbye World'.



```
Good Morning GoodBye World
Good Morning Hello World
Good Morning Hello World
Good Morning GoodBye World
Good Morning GoodBye World
Good Morning Hello World
Good Morning GoodBye World
Good Morning Hello World
Good Morning GoodBye World
Good Morning GoodBye World
Good Morning Hello World
Good Morning Hello World
Good Morning GoodBye World
Good Morning GoodBye World
Good Morning Hello World
Good Morning Hello World
Good Morning GoodBye World
```

Multiple Device Cluster

In order to run the Secondary cluster from a different networked device, open up the structure.json file in the setup folder and set IP address of the object in clusters with name Secondary to the IP of the device that would be running the Secondary script. Then copy the project folder onto the device and run the Secondary script while the MainApp script is launched and waiting.