Introduction to GNURadio

What is GNU Radio?

GNU Radio is a free & open-source software development toolkit that provides signal processing blocks to implement software radios.

Why do I want it?

- It is free and open source
- Customized blocks can be made
- can be used with to create software-defined radios, or without hardware in a simulation-like environment

What exactly does GNU Radio do?

- GNU Radio performs signal processing
- has blocks/elements of its own
- has method of connecting these blocks
- manages how data is passed from one block to another
- missing blocks can be created and added

How to use GNU Radio?

1. Using existing features of GNU Radio:
- GNU Radio companion
- Tools and utility program

2. Creating new customized elements/blocks:
- Extend GNU Radio

Working with GNU Radio

- Python Blocks
- OOT modules make the actual apps functionality
- How to add OOTs
- How to add Python blocks with gr_modtool and how to code them
- How to add GRC bindings for block

Pre-requisites

- Ubuntu Linux OS 12.04 or 14.04
- Basic linux commands
- Working Installation of GNU Radio 3.6.4.2 or later
- Familiar with Python or C++

What is a Block?

Block/element in Sandhi consists of:

- XML file
- Python file

Work Function

A Work Function does the following:

- reads inputs
- processes
- writes outputs

IO Signatures

IO Signatures denotes:

- The number of input ports
- The number of output ports
- The item size of each port

Block Type

- Synchronous Blocks
- Decimation Blocks
- Interpolation Blocks
- General Blocks

Synchronous Block (1:1)

input items = output items

- Can have any number of inputs or outputs
- When a sync block has zero inputs, its called a **Source**
- When a sync block has zero outputs, its called a **Sink**

Decimation Block (N:1)

input items = output items*decimation

The decimation block is another type of fixed rate block where the number of input items is a fixed multiple of the number of output items.

Modifying the Python Block File

1. Import the libraries

import numpy
from gnuradio import gr

Interpolation Block

output items = input items*decimation

The interpolation block is another type of fixed rate block where the number of output items is a fixed multiple of the number of input items.

Basic Block

- The basic block provides no relation between the number of input items and the number of output items
- All other blocks are just simplifications of the basic block
- Users should choose to inherit from basic block when the other blocks are not suitable

Hierarchical Block

- Hierarchical blocks are blocks that are made up of other blocks
- They instantiate the other GNU Radio blocks (or other hierarchical blocks) and connect them together
- A hierarchical block has a “connect” function for this purpose

GRAS block

- GRAS is the application scheduler of Sandhi.
- It enables Sandhi to have:
  - closed-loop flowgraphs
  - dispatch threads
  - handle thread synchronization

Top Block

- Main data structure of a GNU Radio flowgraph
- All blocks are connected under this block
- It has the functions that control the running of the flowgraph

Member functions of Top Block

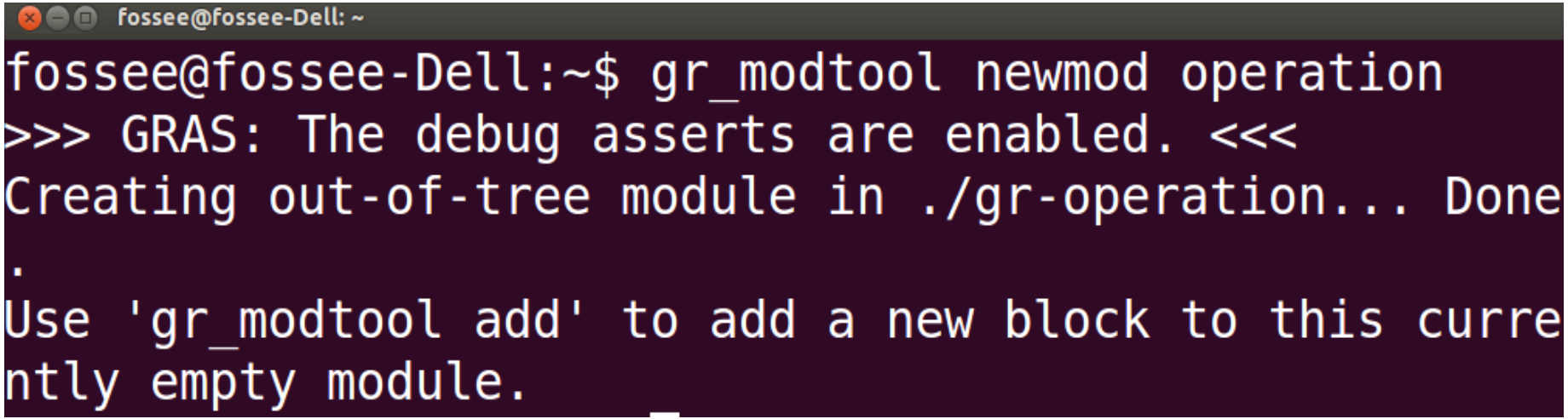
- start(N): starts the flow graph running with N as the maximum noutput_items any block can receive.
- stop(): stops the top block
- wait(): blocks until top block is finished
- run(N): a blocking start(N) (calls start then wait)

Continued..

- lock(): locks the flowgraph so we can reconfigure it
- unlock(): unlocks and restarts the flowgraph

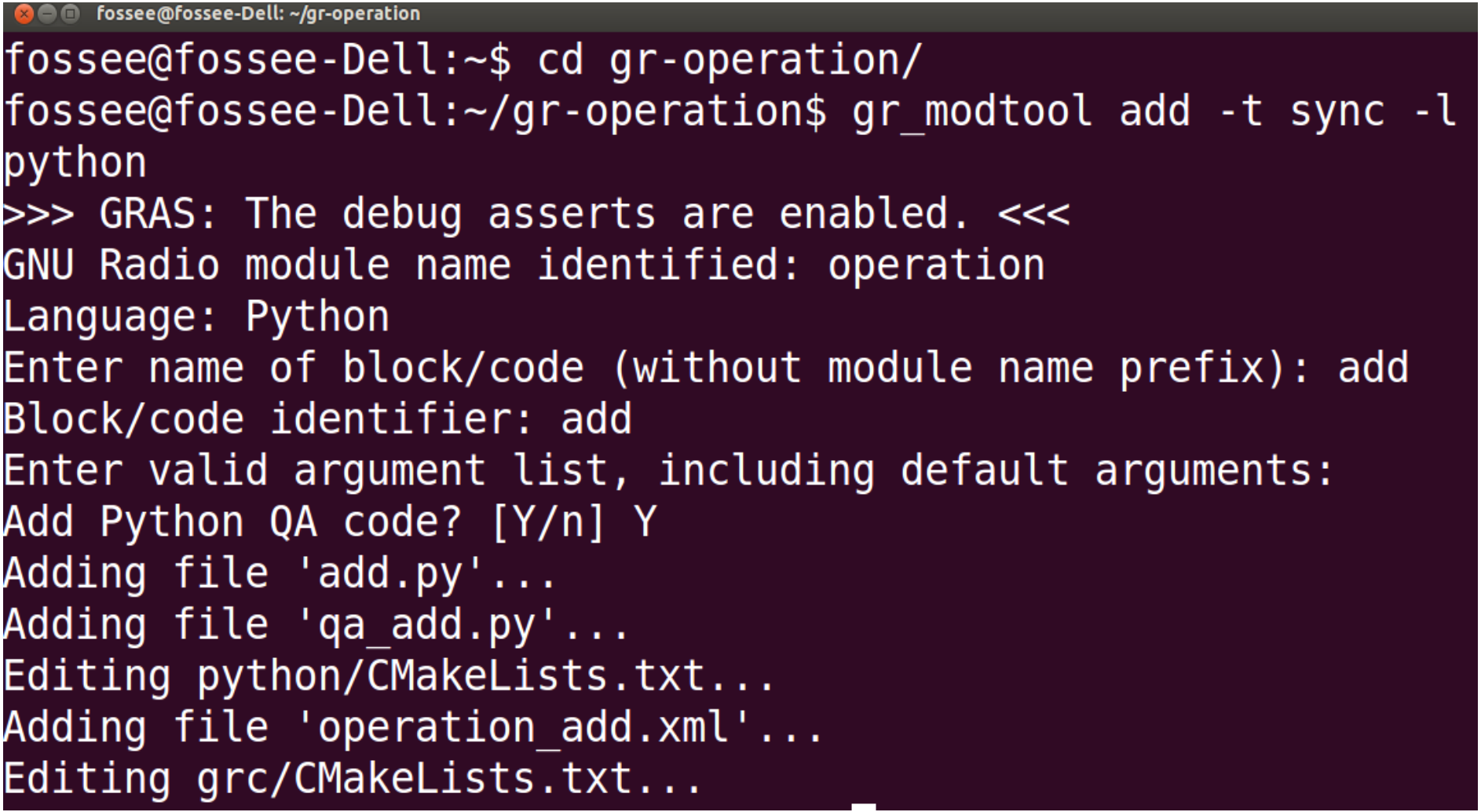
Block creation in Python

1. Setting up a new module:
   a. **gr_modtool newmod <*module name*>**



Continued...

2. Specifying block type
    a. **gr_modtool <*block name*> -t sync -l python**


fossee@fossee-Dell: ~/gr-operation
fossee@fossee-Dell:~$ cd gr-operation/
fossee@fossee-Dell:~/gr-operation$ gr_modtool add -t sync -l
python
>>> GRAS: The debug asserts are enabled. <<<
GNU Radio module name identified: operation
Language: Python
Enter name of block/code (without module name prefix): add
Block/code identifier: add
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] Y
Adding file 'add.py'...
Adding file 'qa_add.py'...
Editing python/CMakeLists.txt...
Adding file 'operation_add.xml'...
Editing grc/CMakeLists.txt...

Code explanation

1. Import libraries:

import numpy
from gnuradio import gr

NumPy is the fundamental package for scientific computing in Python

Continued..

2. Declaration of constructor:

def __init__(self, <*block name*>):
gr.sync_block.__init__(self,
name="<*python file name*>",
in_sig=[<+numpy.float+>],
out_sig=[<+numpy.float+>])

Continued...

- For eg consider, **in_sig=[(numpy.float32,4), numpy.float32]**
  - **(numpy.float32,4) -** one for vectors of 4 floats
  - **numpy.float32 -** scalars
- If in_sig has nothing then it becomes a **Source** block
- if out_sig has nothing it becomes a **Sink** block

Continued...

3. Work function- where the actual processing happens

```
def work(self, input_items, output_items):
    in0 = input_items[0]
    out = output_items[0]
    # <+signal processing here+>
    out[:] = in0
    return len(output_items[0])
```

Continued...

in0 = input_items[0]
out = output_items[0]

- **in0 -** simply store the input in a variable
- **out -** simply store the output in a variable

QA Tests

1. Import libraries:

from gnuradio import gr, gr_unittest
from gnuradio import blocks
from <python file name> import <python file name>

- **gr_unittest** - checks approximate equality of tuples of float and complex numbers

Continued…

2. Test function:

```
def test_001_t (self):
        # set up fg
        self.tb.run ()
        # check data
```

Continued…

For eg:

```
src_data = (0, 1, -2, 5.5, -0.5)
expected_result = (0, 2, -4, 11, -1)
src = blocks.vector_source_f (src_data)
mult = multiply_py_ff (2)
snk = blocks.vector_sink_f ()
```

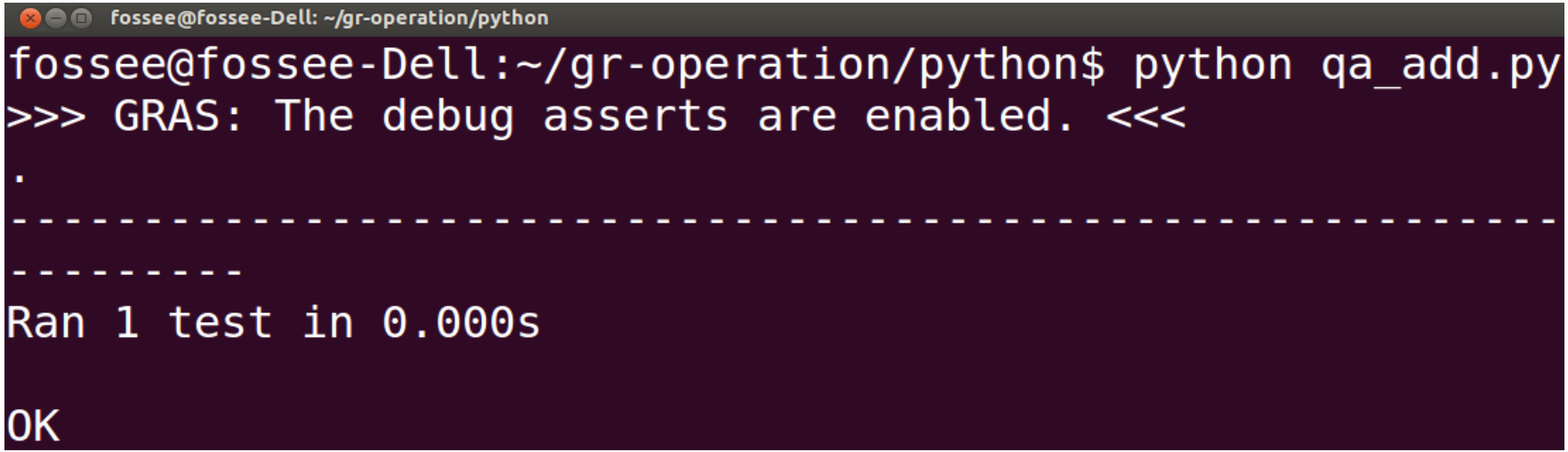
Continued…

```
self.tb.connect (src, mult)
self.tb.connect (mult, snk)
self.tb.run ()
result_data = snk.data ()
self.assertFloatTuplesAlmostEqual
(expected_result, result_data, 6)
```

Continued…

We can then go to the python directory and run:

python <*python qa file name*>.py



XML Files

1. Declaration:

<name><*python file name*></name>
<key><*module n*ame>_<*python file name*></key>
<category><*module name*></category>
<import>import <*module name*></import>
<make><*module name*>.<*python file name*> ($multiple)</make>

Continued…

2. Defining parameters:

```
<param>
    <name>...</name>
    <key>...</key>
    <type>...</type>
  </param>
```

Continued…

3. Source or Sink declaration:

<source or sink>

<name>in</name>
<type><!-- e.g. int, float, complex, byte, short, xxx_vector, ...--></type>
</source or sink>

Installing Python Blocks

To install the block into GRC:

1. Create a build directory called "build".
2. Inside the build directory, we can then run a series of commands:
    a. **cmake ../**
    b. **make**
    c. **sudo make install**
    d. **sudo ldconfig**