# OutOfTreeModules

From GNU Radio

**Extending GNU Radio with own functionality and blocks**

This article borrows heavily from the original (but very outdated) "How to write a block?" written by Eric Blossom.

## Contents



# What is an out-of-tree module?

An out-of-tree module is a GNU Radio component that does not live within the GNU Radio source tree. Typically, if you want to extend GNU Radio with your own functions and blocks, such a module is what you create (i.e. you wouldn't usually add stuff to the actual GNU Radio source tree unless you're planning to submit it to the devs for upstream integration). This allows you to maintain the code yourself and have additional functionality alongside the main code.

A lot of OOT projects are hosted at CGRAN (http://cgran.org) -- the Comprehensive GNU Radio Archive Network. CGRAN projects are all available through our tool PyBOMBS (http://gnuradio.org/pybombs). In fact, when you add your project to the PyBOMBS recipe repo (https://github.com/gnuradio/gr-etcetera), it will automatically update the CGRAN website. For more information about setting up your project and adding it to PyBOMBS, see our tutorial on Configuring OOT Projects. This will help you craft your CMake files for best chances of installation, edit the MANIFEST file of your project to properly display information on CGRAN, and create the recipe file for adding the project to PyBOMBS.

For example of such a module is the spectral estimation toolbox (http://cgran.org/pages/gr-specest.html), which extends GNU Radio with spectral estimation features. When installed, you have more blocks available (e.g. in the GNU Radio companion) which behave just like the rest of GNU Radio; however, the developers are different people.

# Tools and resources at my disposal

There are a couple of tools, scripts, and documents that are available as 3rd-party programs or as part of GNU Radio.

## gr_modtool - The swiss army knife of module editing

When developing a module, there's a lot of boring, monotonous work involved: boilerplate code, makefile editing, etc. gr_modtool is a script which aims to help with all these things by automatically editing makefiles, using templates, and doing as much work as possible for the developer such that you can jump straight into the DSP coding.

Note that gr_modtool makes a lot of assumptions on what the code looks like. The more your module is custom and has specific changes, the less useful gr_modtool will be, but it is probably the best place to start with any new module or block.

gr_modtool is now available in the GNU Radio source tree and is installed by default.

## Developer resources on the wiki

Most important is definitely the block coding guide. While this is written for the GNU Radio main tree, this should also be applied to all modules. Specifically, have a look at the naming conventions!

If you're reading this, you're most likely familiar with all the GNU Radio jargon, but just in case you're not, have a peak at the core concepts tutorial. This contains definitive must-knows. Also, the tutorial on writing Python applications explains a lot of the key features.

## CMake, make, etc.

GNU Radio uses CMake as a build system. Building a module, therefore, requires you to have cmake installed, and whatever build manager you prefer (most often this is 'make', but you could also be using Eclipse or MS Visual Studio).

# Tutorial 1: Creating an out-of-tree module

In the following tutorials, we will use an out-of-tree module called **howto**. The first step is to create this module.

With gr_modtool, this is dead easy. Just point your command line wherever you want your new module directory (this should be outside the GNU Radio source tree!), and go:

```
% gr_modtool newmod howto
Creating out-of-tree module in ./gr-howto... Done.
Use 'gr_modtool add' to add a new block to this currently empty module.
```

If all went well, you now have a new directory called `gr-howto` in which we will work for the other tutorials.

# Structure of a module

Let's jump straight into the gr-howto module and see what it's made up of:

```
gr-howto % ls
apps  cmake  CMakeLists.txt  docs  examples  grc  include  lib  python  swig
```

It consists of several subdirectories. Anything that will be written in C++ (or C, or any language that is not Python) is put into `lib/`. For C++ files, we usually have headers which are put into `include/` (if they are to be exported) or also in `lib/` (if they're only relevant during compile time, but are not installed later, such as `_impl.h` files. You'll see what that is in the next tutorial).

Of course, Python stuff goes into the `python/` directory. This includes unit tests (which are not installed) and parts of the Python module which are installed.

You probably know already that GNU Radio blocks are available in Python even if they were written in C++. This is done by the help of SWIG, the simplified wrapper and interface generator, which automatically creates glue code to make this possible. SWIG needs some instructions on how to do this, which are put into the `swig/` subdirectory. Unless doing something extra clever with your block, you will not need to go into the `swig/` directory; gr_modtool handles all of that for us.

If you want your blocks to be available in the GNU Radio companion, the graphical UI for GNU Radio, you need to add XML descriptions of the blocks and put them into `grc/`.

For documentation, `docs/` contains some instructions on how to extract documentation from the C++ files and Python files (we use Doxygen and Sphinx for this) and also make sure they're available as docstrings in Python. Of course, you can add custom documentation here as well.

The `apps/` subdir contains any complete applications (both for GRC and standalone executables) which are installed to the system alongside with the blocks.

The directory, `examples/` can be used to save (guess what) examples, which are a great addendum to documentation because other developers can simply look straight at the code to see how your blocks are used.

The build system brings some baggage along, as well: the `CMakeLists.txt` file (one of which is present in every subdirectory) and the `cmake/` folder. You can ignore the latter for now, as it brings along mainly instructions for CMake on how to find GNU Radio libraries etc. The CMakeLists.txt files need to be edited a lot in order to make

sure your module builds correctly.

But one step at a time! Now, let's move on to our next tutorial.

# Tutorial 2: Writing a block (square_ff) in C++

For our first example, we'll create a block that computes the square of its single float input. This block will accept a single float input stream and produce a single float output stream, i.e., for every incoming float item, we output one float item which is the square of that input item.

Following the naming conventions, the block will be called `square_ff` because it has float inputs, float outputs.

We are going to arrange that this block, as well as the others that we write in this article, end up in the `howto` Python module. This will allow us to access it from Python like this:

```
import howto
sqr = howto.square_ff()
```

## Creating the files

First step is to create empty files for the block and edit the CMakeLists.txt files.
Again, `gr_modtool` does the job. On the command line, go to the `gr-howto` directory and enter:

```
gr-howto % gr_modtool add -t general -l cpp square_ff
GNU Radio module name identified: howto
Block/code identifier: square_ff
Language: C++
Please specify the copyright holder:
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [y/N]
Adding file 'square_ff_impl.h'...
Adding file 'square_ff_impl.cc'...
Adding file 'square_ff.h'...
Editing swig/howto_swig.i...
Adding file 'qa_square_ff.py'...
Editing python/CMakeLists.txt...
Adding file 'howto_square_ff.xml'...
Editing grc/CMakeLists.txt...
```

On the command line, we specify that we're adding a block, its type is 'general' (because we don't know what block types are, yet) and it is called `square_ff`. The block should be created in C++ and it currently has no specified copyright holder (by default, gr-module author is the copyright holder). `gr_modtool` then asks you if your block takes any arguments (it doesn't, so we leave that empty), whether or not we want QA code for Python (yes, we do) and for C++ (no, we don't right now).

Now, have another look at the different CMakeLists.txt files and see what `gr_modtool` did. You can also see a lot of new files, which now have to be edited if we want the block to work.

## Test Driven Programming

We could just start banging out the C++ code, but being highly evolved modern programmers, we're going to write the test code first. After all, we do have a good spec for the behavior: take a single stream of floats as the input and produce a single stream of floats as the output. The output should be the square of the input.

How hard could this be? Turns out that this is easy! So, we open `python/qa_square_ff.py`, which we edit to look like this:

```
from gnuradio import gr, gr_unittest
from gnuradio import blocks
import howto_swig as howto

class qa_square_ff (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    def test_001_square_ff(self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = blocks.vector_source_f(src_data)
        sqr = howto.square_ff()
        dst = blocks.vector_sink_f()
        self.tb.connect(src, sqr)
        self.tb.connect(sqr, dst)
        self.tb.run()
        result_data = dst.data()
        self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)

if __name__ == '__main__':
    gr_unittest.run(qa_square_ff, "qa_square_ff.xml")
```

gr_unittest is an extension to the standard Python module unittest. gr_unittest adds support for checking approximate equality of tuples of float and complex numbers. Unittest uses Python's reflection mechanism to find all methods that start with test_ and runs them. Unittest wraps each call to test_* with matching calls to setUp and tearDown. See the Python unittest documentation (http://docs.python.org/2/library/unittest.html) for details.

When we run the test, gr_unittest.main is going to invoke setUp, test_001_square_ff, and tearDown, in that order.

`test_001_square_ff` builds a small graph that contains three nodes. `blocks.vector_source_f(src_data)` will source the elements of `src_data` and then say that it's finished. `howto.square_ff` is the block we're testing. `blocks.vector_sink_f` gathers the output of `howto.square_ff`.

The `run()` method runs the graph until all the blocks indicate they are finished. Finally, we check that the result of executing `square_ff` on `src_data` matches what we expect.

Note that such a test is usually called before installing the module. This means that we need some trickery to be able to load the blocks when testing. CMake takes care of most things by changing PYTHONPATH appropriately. Also, we import `howto_swig` instead of `howto` in this file.

In order for CMake to actually know this test exists, `gr_modtool` modified `python/CMakeLists.txt` with these lines:

```
########################################################################
# Handle the unit tests
########################################################################
include(GrTest)

set(GR_TEST_TARGET_DEPS gnuradio-howto)
set(GR_TEST_PYTHON_DIRS ${CMAKE_BINARY_DIR}/swig)
GR_ADD_TEST(qa_square_ff ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/qa_square_ff.py)
```

## The C++ code (part 1)

Now that we've got a test case, let's write the C++ code. All signal processing blocks are derived from `gr::block` or one of its subclasses. Go check out the block documentation (http://gnuradio.org/doc/doxygen/classgr_1_1block.html) on the Doxygen-generated manual.

`gr_modtool` already provided us with three files that define the block: `lib/square_ff_impl.h`, `lib/square_ff_impl.cc` and `include/howto/square_ff.h`. All we have to do is modify to them to do our bidding. After you've finished with this tutorial *please* read and understand the Blocks Coding Guide (https://wiki.gnuradio.org/index.php/BlocksCodingGuide) to find out how these files are structured and why!

First of all, we have a look at our header files. Because the block we're writing is so simple, we don't have to actually change them (the header file in `include/` is often quite complete after running `gr_modtool`, unless we need to add some public methods such as mutator methods, i.e., getters and setters). That leaves us with `lib/square_ff_impl.cc`.

`gr_modtool` hints at where you have to change code by adding `<++>` symbols.
Let's go through these one at a time:

```
square_ff_impl::square_ff_impl()
  : gr::block("square_ff",
              gr::io_signature::make(1, 1, sizeof (float)), // input signature
              gr::io_signature::make(1, 1, sizeof (float))) // output signature
{
    // empty constructor
}
```

The constructor itself is empty, as the squaring block has no need to set up anything.

The only interesting portion is the definition of the input and output signatures: At the input, we have 1 port that allows float inputs. The output port is the same.

```
void
square_ff_impl::forecast (int noutput_items, gr_vector_int &ninput_items_required)
{
  ninput_items_required[0] = noutput_items;
}
```

`forecast()` is a function which tells the scheduler how many input items are required to produce `noutput_items` output items. In this case, they're the same. The index 0 indicates that this is for the first port, but we only have one any way. This is generally the case for `forecast` in a lot of blocks. For examples, you can look at how `gr::block`, `gr::sync_block`, `gr::sync_decimator`, and `gr::sync_interpolator` define the default forecast functions to account for things like rate changes and history.

Finally, there's `general_work()`, which is pure virtual in `gr::block`, so we definitely need to override that. `general_work()` is the method that does the actual signal processing:

```
int
square_ff_impl::general_work (int noutput_items,
                              gr_vector_int &ninput_items,
                              gr_vector_const_void_star &input_items,
                              gr_vector_void_star &output_items)
{
  const float *in = (const float *) input_items[0];
  float *out = (float *) output_items[0];

  for(int i = 0; i < noutput_items; i++) {
    out[i] = in[i] * in[i];
  }
```

```
    // Tell runtime system how many input items we consumed on
    // each input stream.
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
  }
```

There is one pointer to the input- and one pointer to the output buffer, respectively, and a for-loop which copies the square of the input buffer to the output buffer.

## Using CMake

If you've never used CMake before, this is a good time to give it a try. The typical workflow of a CMake-based project as seen from the command line is this (if using PyBOMBS, first read **Build Tree vs. Install Tree**):

```
$ mkdir build    # We're currently in the module's top directory
$ cd build/
$ cmake ../      # Tell CMake that all its config files are one dir up
$ make           # And start building (should work after the previous section)
```

#### Build Tree vs. Install Tree

When you run cmake, you usually run it in a separate directory (e.g. `build/`). This is the build tree. The path to the install tree is `$prefix/lib/$pythonversion/dist-packages`, where `$prefix` is whatever you specified to CMake during configuration (usually `/usr/local/`) with the `-DCMAKE_INSTALL_PREFIX` switch. (Note: different versions of Python will either use site-packages or dist-packages; dist-packages is the newer way and most likely for newer OSes and installations.)

If you installed GNU Radio using PyBOMBS, the install tree is located in the `target/` directory set during the initial PyBOMBS configuration. Make sure to add the -DCMAKE_INSTALL_PREFIX switch for CMake, so that it will correctly locate your GNU Radio installation. The command should look similar to this:

```
$ cmake -DCMAKE_INSTALL_PREFIX= ../ #  should be the configured PyBOMBS target
```

Now we have a new directory `build/` in our module's directory. All the compiling etc. is done in here, so the actual source tree is not littered with temporary files. If we change any CMakeLists.txt files, we should re-run `cmake ../` (although in truth, cmake detects these changes and reruns automatically when you next run `make`). During compilation, the libraries are copied into the build tree. Only during installation, files are installed to the install tree, thus making our blocks available to GNU Radio apps.

We write our applications such that they access the code and libraries in the install tree. On the other hand, we want our test code to run on the build tree, where we can detect problems before installation.

## Let's try that -- running `make test`

Because we wrote the QA code before the C++ code, we can immediately see if what we did was correct.

We use `make test` to run our tests (run this from the `build/` subdirectory, after calling `cmake` and `make`). This invokes a shell script which sets up the PYTHONPATH environment variable so that our tests use the build tree versions of our code and libraries. It then runs all files which have names of the form qa_*.py and reports the

overall success or failure.

There is quite a bit of behind-the-scenes action required to use the non-installed versions of our code (look at the `cmake/` directory for a cheap thrill.)

If you completed the `square_ff` block, this should work fine:

```
gr-howto/build % make test
Running tests...
Test project /home/braun/tmp/gr-howto/build
    Start 1: test_howto
1/2 Test #1: test_howto .......................   Passed    0.01 sec
    Start 2: qa_square_ff
2/2 Test #2: qa_square_ff .....................   Passed    0.38 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.39 sec
```

If something fails during the tests, we can dig a little deeper. When we run `make test`, we're actually invoking the CMake program `ctest`, which has a number of options we can pass to it for more detailed information. Say we forgot to multiply `in[i] * in[i]` and so aren't actually squaring the signal. If we just run `make test` or even just `ctest`, we would get this:

```
gr-howto/build  $ ctest
Test project /home/braun/tmp/gr-howto/build
    Start 1: test_howto
1/2 Test #1: test_howto .......................   Passed    0.02 sec
    Start 2: qa_square_ff
2/2 Test #2: qa_square_ff .....................***Failed    0.21 sec

50% tests passed, 1 tests failed out of 2

Total Test time (real) =   0.23 sec

The following tests FAILED:
      2 - qa_square_ff (Failed)
Errors while running CTest
```

To find out what happened with our qa_square_ff test, we run `ctest -V -R square`. The '-V' flag gives us verbose output and the '-R' flag is a regex expression to only run those tests which match.

```
gr-howto/build  $ ctest -V -R square
UpdateCTestConfiguration  from :/home/braun/tmp/gr-howto/build/DartConfiguration.tcl
UpdateCTestConfiguration  from :/home/bruan/tmp/gr-howto/build/DartConfiguration.tcl
Test project /home/braun/tmp/gr-howto/build
Constructing a list of tests
Done constructing a list of tests
Checking test dependency graph...
Checking test dependency graph end
test 2
    Start 2: qa_square_ff

2: Test command: /bin/sh "/home/bruan/tmp/gr-howto/build/python/qa_square_ff_test.sh"
2: Test timeout computed to be: 9.99988e+06
2: F
2: ======================================================================
2: FAIL: test_001_t (__main__.qa_square_ff)
2: ----------------------------------------------------------------------
2: Traceback (most recent call last):
2:   File "/home/braun/tmp/gr-howto/python/qa_square_ff.py", line 44, in test_001_t
2:     self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)
2:   File "/opt/gr/lib/python2.7/dist-packages/gnuradio/gr_unittest.py", line 90, in assertFloatTuplesAlmostEqual
2:     self.assertAlmostEqual (a[i], b[i], places, msg)
```

```
2: AssertionError: 9 != -3.0 within 6 places
2:
2: ----------------------------------------------------------------------
2: Ran 1 test in 0.002s
2:
2: FAILED (failures=1)
1/1 Test #2: qa_square_ff .....................***Failed    0.21 sec

0% tests passed, 1 tests failed out of 1

Total Test time (real) =   0.21 sec

The following tests FAILED:
      2 - qa_square_ff (Failed)
Errors while running CTest
```

This tells us that "9 != -3.0" because we expected the output to be (-3)^2 = 9 but really got the input of -3. We can use this information to go back and fix our block until the tests pass.

We can also put in debug print statements into our QA code on failures, like printing out `expected_result` and `result_data` to compare them to better understand the problem.

## More C++ code (but better) - Subclasses for common patterns

`gr::block` allows tremendous flexibility with regard to the consumption of input streams and the production of output streams. Adroit use of `forecast()` and `consume()` (see below) allows variable rate blocks to be built. It is possible to construct blocks that consume data at different rates on each input and produce output at a rate that is a function of the contents of the input data.

On the other hand, it is very common for signal processing blocks to have a fixed relationship between the input rate and the output rate. Many are 1:1, while others have 1:N or N:1 relationships. You must have thought the same thing in the `general_work()` function of the previous block: if the number of items consumed is identical the number of items produced, why do I have to tell GNU Radio the exact same number twice?

Another common requirement is the need to examine more than one input sample to produce a single output sample. This is orthogonal to the relationship between input and output rate. For example, a non-decimating, non-interpolating FIR filter needs to examine N input samples for each output sample it produces, where N is the number of taps in the filter. However, it only consumes a single input sample to produce a single output. We call this concept "history", but you could also think of it as "look-ahead".

- `gr::sync_block`

gr::sync_block is derived from gr::block and implements a 1:1 block with optional history. Given that we know the input to output rate, certain simplifications are possible. From the implementor's point-of-view, the primary change is that we define a `work()` method instead of `general_work()`. `work()` has a slightly different calling sequence; it omits the unnecessary `ninput_items` parameter, and arranges for `consume_each()` to be called on our behalf.

Let's add another block which derives from `gr::sync_block` and call it `square2_ff`. First, we edit `qa_square_ff.py` to add another test:

```
    def test_002_square2_ff(self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = blocks.vector_source_f(src_data)
        sqr = howto.square2_ff()
        dst = blocks.vector_sink_f()
        self.tb.connect(src, sqr, dst)
        self.tb.run()
```

```
       result_data = dst.data()
       self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)
```

You can see it's the exact same test as before except for the use of `square2_ff`.

Then, we use `gr_modtool` to add the block files, skipping the QA code (because we already have that):

```
gr-howto % gr_modtool add -t sync -l cpp square2_ff
GNU Radio module name identified: howto
Block/code identifier: square2_ff
Language: C++
Please specify the copyright holder:
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] n
Add C++ QA code? [Y/n] n
Adding file 'square2_ff_impl.h'...
Adding file 'square2_ff_impl.cc'...
Adding file 'square2_ff.h'...
Editing swig/howto_swig.i...
Adding file 'howto_square2_ff.xml'...
Editing grc/CMakeLists.txt...
```

The constructor in `square2_ff_impl.cc` is done the same way as before, except for the parent class being `gr::sync_block`.

```
    square2_ff_impl::square2_ff_impl()
      : gr::sync_block("square2_ff",
                       gr::io_signature::make(1, 1, sizeof (float)),
                       gr::io_signature::make(1, 1, sizeof (float)))
    {}

// [...] skip some lines ...

    int
    square2_ff_impl::work(int noutput_items,
                          gr_vector_const_void_star &input_items,
                          gr_vector_void_star &output_items)
    {
      const float *in = (const float *) input_items[0];
      float *out = (float *) output_items[0];

      for(int i = 0; i < noutput_items; i++) {
        out[i] = in[i] * in[i];
      }

      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

The `work` function is the real difference (also, we don't have a `forecast()` function any more). We'll look at it in greater detail in the next section.

This gives us fewer things to worry about and less code to write. If the block requires history greater than 1, call `set_history()` in the constructor or any time the requirement changes.

`gr::sync_block` provides a version of `forecast` that handles the history requirement.

- `gr::sync_decimator`

`gr::sync_decimator` is derived from `gr::sync_block` and implements a N:1 block with optional history.

- `gr::sync_interpolator`

`gr::sync_interpolator` is derived from `gr::sync_block` and implements a 1:N block with optional history.

With this knowledge it should be clear that `howto_square_ff` should be a `gr::sync_block` with no history.

Now, go back into our build directory and run `make`. Because `gr_modtool` added the `square2_ff` block to the necessary CMakeLists.txt files, `cmake` is automatically rerun for us and followed by `make`.

Again, running `make test` will spawn a test run with of `qa_square_ff.py` which should not fail.

## Inside the `work()` function

If you're using a sync block (including decimator and interpolator), this is how the skeleton code looks like produced by gr_modtool:

```
int
my_block_name::work(int noutput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)
{
  const float *in = (const float *) input_items[0];
  float *out = (float *) output_items[0];

  // Do <+signal processing+>

  // Tell runtime system how many output items we produced.
  return noutput_items;
}
```

So, given history, vectors, multiple input ports etc., is this really all you need? Yes, it is! Because sync blocks have a fixed output to input rate, all you need to know is the number of output items, and you can calculate how many input items are available.

Example - the adder block: source:gr-blocks/lib/add_XX_impl.cc.t

This block has an unknown number of inputs and variable vector lengths. The number of connected ports can be checked by `input_items.size()` and `output_items.size()`. The outer `for` loop, which goes over all the available items, goes up to `noutput_items*d_vlen`. The number of output items is identical to the number of input items because it is a sync block, and you can trust GNU Radio to have this number of items available. In this case, one item is a vector of samples, but we want to add the individual samples, so the for loop considers that.

Example - interpolation in gr::blocks::unpack_k_bits_bb: source:gr-blocks/lib/unpack_k_bits_bb_impl.cc

This is a block which picks apart bytes and produces the individual bits. Again, it is unknown at compile time how many bits per byte there are. However, there's a fixed number of output items per input item, so we can simply divide `noutput_items/d_k` to get the correct number of input items. It will always be correct because GNU Radio knows the input to output ratio and will make sure that `noutput_items` is always a multiple of this integer ratio.

Example - history in source:gr-digital/lib/diff_phasor_cc_impl.cc

If you use history of length k, GNU Radio will keep k-1 entries of the input buffer instead of discarding them. This means that if GNU Radio tells you the input buffer has N items, it actually has N+k-1 items you may use.

Consider the example above. We need one previous item, so history is set to k=2. If you inspect the for loop closely, you'll find that out of `noutput_items` items, `noutput_items+1` items are actually read. This is possible because there is an extra item in the input buffer from the history.

After consuming `noutput_items` items, the last entry is not discarded and will be available for the next call of `work()`.

## Help! My test fails!

Congratulations! If your test fails, your QA code has already paid for itself. Obviously, you want to fix everything before you continue.

You can use the command `ctest -V` (instead of `make test`, again, all in your `build/` subdirectory) to get all the output from the tests. You can also use `ctest -V -R REGEX` to only run tests that match REGEX, if you have many tests and want to narrow it down. If you can't figure out the problem from the output of your QA code, put in `print` statements and show intermediary results. If you need more info on debugging blocks, check out the debugging tutorial.

## Making your blocks available in GRC

You can now install your module, but it will not be available in GRC. That's because `gr_modtool` can't create valid XML files before you've even written a block. The XML code generated when you call `gr_modtool add` is just some skeleton code.

Once you've finished writing the block, `gr_modtool` has a function to help you create the XML code for you. For the howto example, you can invoke it on the `square2_ff` block by calling

```
gr-howto % gr_modtool makexml square2_ff
GNU Radio module name identified: howto
Warning: This is an experimental feature. Don't expect any magic.
Searching for matching files in lib/:
Making GRC bindings for lib/square2_ff_impl.cc...
Overwrite existing GRC file? [y/N] y
```

Note that `gr_modtool add` creates an invalid GRC file, so we can overwrite that.

In most cases, `gr_modtool` can't figure out all the parameters by itself and you will have to edit the appropriate XML file by hand. The GRC wiki site has a description available.

In this case, because the block is so simple, the XML is actually valid. Have a look at `grc/howto_square2_ff.xml`:

```
<block>
  <name>Square ff</name>
  <key>howto_square_ff</key>
  <category>[HOWTO]</category>
  <import>import howto</import>
  <make>howto.square_ff()</make>
  <sink>
    <name>in</name>
    <type>float</type>
  </sink>
  <source>
    <name>out</name>
    <type>float</type>
  </source>
</block>
```

Perhaps you want to change the autogenerated name to something nicer.

If you do a `make install` from the build directory, you can use the block in GRC. If GRC is already running, you can hit the "Reload Blocks" button in the GRC toolbar; it's a blue circular arrow on the right-hand side. You should now see a "HOWTO" category in the block tree.

## There's more: additional `gr::block`-methods

If you've read the gr::block documentation (http://gnuradio.org/doc/doxygen/classgr_1_1block.html) (which you should have), you'll have noticed there are a great number of methods available to configure your block.

Here's some of the more important ones:

#### `set_history()`

If your block needs a history (i.e., something like an FIR filter), call this in the constructor. GNU Radio then makes sure you have the given number of 'old' items available.

The smallest history you can have is 1, i.e., for every output item, you need 1 input item. If you choose a larger value, N, this means your output item is calculated from the current input item and from the N-1 previous input items.

The scheduler takes care of this for you. If you set the history to length N, the first N items in the input buffer include the N-1 previous ones (even though you've already consumed them).

#### `forecast()`

The system needs to know how much data is required to ensure validity in each of the input arrays. As stated before, the `forecast()` method provides this information, and you must therefore override it anytime you write a `gr::block` derivative (for sync blocks, this is implicit).

The default implementation of `forecast()` says there is a 1:1 relationship between `noutput_items` and the requirements for each input stream. The size of the items is defined by `gr::io_signature::make` in the constructor of `gr::block`. The sizes of the input and output items can of course differ; this still qualifies as a 1:1 relationship. Of course, if you had this relationship, you wouldn't want to use a `gr::block`!

```
// default implementation:  1:1
void
gr::block::forecast(int noutput_items,
                    gr_vector_int &ninput_items_required)
{
  unsigned ninputs = ninput_items_required.size ();
  for(unsigned i = 0; i < ninputs; i++)
    ninput_items_required[i] = noutput_items;
}
```

Although the 1:1 implementation worked for `square_ff`, it wouldn't be appropriate for interpolators, decimators, or blocks with a more complicated relationship between `noutput_items` and the input requirements. That said, by deriving your classes from gr::sync_block, gr::sync_interpolator or gr::sync_decimator instead of gr::block, you can often avoid implementing forecast.

#### `set_output_multiple()`

When implementing your `general_work()` routine, it's occasionally convenient to have the run time system ensure that you are only asked to produce a number of output items that is a multiple of some particular value. This might occur if your algorithm naturally applies to a fixed sized block of data. Call `set_output_multiple` in your constructor to specify this requirement. The default output multiple is 1.

### Finalizing your work and installing

First, go through this checklist:

- Have you written one or more blocks, including QA codes?
- Does `make test` pass?
- Are there GRC bindings available (if that's what you want)?

In that case, you can go ahead and install your module. On a Linux machine, this would mean going back to the build directory and calling `make install`:

```
$ cd build/
$ make install # or sudo make install
```

With Ubuntu, you may have to call `ldconfig` as well:

```
$ sudo ldconfig
```

Otherwise, you'll get an error message that the library you just installed cannot be found.

## Other types of blocks

### Sources and sinks

Sources and sinks are derived from `gr::sync_block`. The only thing different about them is that sources have no inputs and sinks have no outputs. This is reflected in the `gr::io_signature::make` that are passed to the `gr::sync_block` constructor. Take a look at [source:gr-blocks/lib/file_source_impl.cc file_source.{h,cc}] and file_sink_impl.{h,cc} for some very straight-forward examples. See also the tutorial on writing Python applications.

### Hierarchical blocks

For the concept of hierarchical blocks, see this. Of course, they can also be written in C++. *`gr_modtool` supports skeleton code for hierarchical blocks both in Python and C++.*

```
~/gr-howto % gr_modtool.py add -t hier -l cpp hierblockcpp_ff
GNU Radio module name identified: howto
Block/code identifier: hierblockcpp_ff
Language: C++
Please specify the copyright holder:
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [y/N]
Adding file 'hierblockcpp_ff_impl.h'...
Adding file 'hierblockcpp_ff_impl.cc'...
Adding file 'hierblockcpp_ff.h'...
Editing swig/howto_swig.i...
```

```
Adding file 'howto_hierblockcpp_ff.xml'...
Editing grc/CMakeLists.txt...
```

Using the `-l python` switch creates such a block in Python.

# Everything at one glance: Cheat sheet for editing modules/components:

Here's a quick list for all the steps necessary to build blocks and out-of-tree modules:

1. Create (do this once per module): `gr_modtool create MODULENAME`
2. Add a block to the module: `gr_modtool add BLOCKNAME`
3. Create a build directory: `mkdir build/`
4. Invoke the make process: `cd build && cmake <OPTIONS> ../ && make` (Note that you only have to call cmake if you've changed the CMake files)
5. Invoke the testing: `make test` or `ctest` or `ctest -V` for more verbosity
6. Install (only when everything works and no tests fail): `sudo make install`
7. Ubuntu users: reload the libs: `sudo ldconfig`
8. Delete blocks from the source tree: `gr_modtool rm REGEX`
9. Disable blocks by removing them from the CMake files: `gr_modtool disable REGEX`

# Tutorial 3: Writing a signal processing block in Python

**Note:** Writing signal processing blocks in Python comes with a performance penalty. The most common cause for using Python to write blocks is because you want to quickly prototype something without having to argue with C++.

From the previous tutorials, you already know about blocks and how they work. Lets go through things a bit quicker, and code another squaring block in pure Python, which shall be called `square3_ff()`.

## Adding the test case

So, first of all, we add another test case by editing `qa_square_ff.py`. Leaving out the test cases for the other two blocks, the QA file now looks like this:

```
from gnuradio import gr, gr_unittest
from gnuradio import blocks
import howto_swig
from square3_ff import square3_ff

class qa_square_ff (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    # [...] Skipped the other test cases

    def test_003_square3_ff (self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = blocks.vector_source_f (src_data)
        sqr = square3_ff ()
        dst = blocks.vector_sink_f ()
        self.tb.connect (src, sqr)
        self.tb.connect (sqr, dst)
        self.tb.run ()
```

```
        result_data = dst.data ()
        self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)

if __name__ == '__main__':
    gr_unittest.main ()
```

The actual test case looks **exactly** like the previous ones did, only replacing the block definition with `square3_ff()`. The only other difference is in the import statements: We are now importing a module called `square3_ff` from which we pull the new block.

## Adding the block code

Having put the unit test in place, we add a file called `square3_ff.py` into the `python/` directory using `gr_modtool`:

```
gr-howto % gr_modtool add -t sync -l python square3_ff
GNU Radio module name identified: howto
Block/code identifier: square3_ff
Language: Python
Please specify the copyright holder:
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] n
Adding file 'square3_ff.py'...
Adding file 'howto_square3_ff.xml'...
Editing grc/CMakeLists.txt...
```

Remember not to add any QA files as we're using the existing one. Next, edit the new file `python/square3_ff.py`. It should look a bit like this:

```
import numpy
from gnuradio import gr

class square3_ff(gr.sync_block):
    " Squaring block "
    def __init__(self):
        gr.sync_block.__init__(
            self,
            name = "square3_ff",
            in_sig = [numpy.float32], # Input signature: 1 float at a time
            out_sig = [numpy.float32], # Output signature: 1 float at a time
        )

    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0] * input_items[0] # Only works because numpy.array
        return len(output_items[0])
```

Some things should immediately stick out:

- The block class is derived from `gr.sync_block`, just like the C++ version was derived from gr::sync_block
- It has a constructor where the name and input/output signatures are set and a `work()` function

However, there are some major differences to the C++ version:

- The input and output signatures are simply defined as a list. Every element contains the item size of that port. So in this case, there is one port per input and one port per output and each has an item size of `numpy.float32` (a single-precision float). If you want a port to operate on vectors, define a tuple, e.g. [(numpy.float32, 4), numpy.float32] means there are two ports: The first one is for vectors of 4 floats, the second is for scalar floats.
- When assigning vectors to `output_items`, remember to use the `[:]` operator. This makes sure Python doesn't rebind the variables or does something clever but guarantees that the data is properly copied

- `input_items` and `output_items` are numpy arrays, which is why we can do the very simple element-wise multiplication the way it's done here (instead of a list comprehension)
- No recompiling is necessary for the `make test` (faster development cycles, yay!)

### Other types of Python blocks

Just like the C++ variant, there are four types of blocks in Python:

- `gr.sync_block`
- `gr.decim_block`
- `gr.interp_block`
- `gr.basic_block` - The Python version of `gr::block`

Like their C++ versions, these blocks have `forecast()`, `work()`, and `general_work()` methods you can override. The difference is, the argument list for the work functions is always as shown in the previous example:

```
def work(self, input_items, output_items):
# Do stuff

def general_work(self, input_items, output_items):
# Do stuff
```

The number of input/output items is obtained through `len(input_items[PORT_NUM])`.

### More examples

Check out the QA code for the Python blocks for some good examples:

- gr-blocks/python/blocks/qa_block_gateway.py (https://github.com/gnuradio/gnuradio/blob/master/gr-blocks/python/blocks/qa_block_gateway.py)

## Debugging blocks

Debugging GNU Radio is available as a separate tutorial.