The TDL Bedside Companion

How To Find Your Way Among The Trees, And Much More Besides

O.M. Smirnov
smirnov@astron.nl

July 23, 2007

Contents

1 Hello World **7**
1.1 Browsers, kernels, trees: a quick primer 8
1.2 Defining a node 9
1.3 Our first "real" tree 11
1.4 How to ask the right question 16

2 Growing Forests **23**

3 The Python Perspective **25**
3.1 Node scopes, stubs and definitions 26
3.2 What really happens during binding 29
3.3 Hiding information 32
3.4 Using Python variables and functions 33
3.5 Node definitions, up close 39
3.6 Using qualifiers 41
3.7 Built-in node arithmetic 44

4 What's In a Name? **49**
4.1 Qualifiers as naming devices 50
4.2 Node scopes 51

5 Node tags and scope searches **57**
5.1 Node tags 58
5.2 Searching for tags 58
5.3 Searching within subtrees and families 59
5.4 More complicated searches 60

6 TDLOptions: Get Your Free GUI Here **61**
6.1 Run-time vs. compile-time options 62

7 Meow: a Case Study **63**
7.1 Jones series 63
7.2 Using contract adapters 65
7.3 Meow and node tags 67

Introduction

blah blah MeqTrees blah blah TDL blah blah

About the demo scripts

With a few exceptions, the demo scripts used here were originally developed for the second MeqTree Workshop (Dwingeloo, January–February 2007.) You can get a copy of all these scripts via our Subversion repository, under

```
Timba/doc/TDL_Companion/Scripts/
```

This directory should contain *all* the scripts used during the workshop, while the book covers only a subset. If you'd like to study the other scripts, then you should also refer to the workshop presentations under

```
Timba/doc/Courses/Workshop2007-Presentations/
```

for additional background material.

Chapter 1

Hello World

It has become a tradition to start any introduction to a programming language with a "Hello World" program. The first Hello World I ever saw was in Kernighan-Richie's classic "The C Programming Language". It is also one of the few Hello World programs that is truly transparent (look up a Java "Hello World" some time if you want an example of truly painful programming...) Unfortunately, a TDL "Hello World" will necessarily tend to be on the obtuse side, since TDL was designed for something other than printing frivolous messages.

Nevertheless, we can try to provide an equivalent (see `1-hello-world.py`):

```
from Timba.TDL import *

def _define_forest (ns,**kwargs):
  ns.hello_world << Meq.Constant(value=0,message="Hello world!");

print "Hello world!"
```

Despite looking a little bit odd (i.e. the "<<" operator is in an unusual context), this script is undoubtedly pure Python. This is because TDL *is* Python. The point well-worth emphasizing:

TDL is Python!!!

If you're not familiar with Python, now is the time to close this book and go read the excellent *"The Python Programming Language"* (???). Once you've done that, I recommend also getting *"The Python Cookbook"* and keeping it somewhere where you're likely to broswe it often, a few pages at a time. Python has many elegant mechanisms for getting complex jobs done quickly, and TDL is designed to take full advantage of them; the *Cookbook* is a great resource for neat and useful little Python tricks.

Now then, what happens if you run this script? Of course, you can just run it through Python from the command line:

```
$ python 1-helloworld.py
Debug: registered context Global=0
Registered verbose context: tdl = 0
Registered verbose context: pixmaps = 0
Registered verbose context: meqds = 0
Registered verbose context: widgets = 0
Registered verbose context: gw = 0
Registered verbose context: octopussy = 0
Registered verbose context: tdlopt = 0
Hello world!
$
```

Given that TDL is Python, the result is rather predictable: the `import` statement pulls in some modules (which result in the debug messages at the top), then there's a function being defined (seemingly to no purpose), and then the `print` statement at the top level produces "Hello world!".

To see the real point of this script, we have to run it via MeqTrees.

1.1 Browsers, kernels, trees: a quick primer

MeqTrees consists of three main components:

- A GUI called the *MeqBrowser* (or simply the *browser* for short.) This is the only piece of software that you will be interacting with directly. The browser is responsible for running TDL scripts, defining trees, controlling trees, and visualizing the results.
- The trees defined inside the browser are passed on to a *kernel* (a.k.a. the meqserver) for execution. The kernel runs as a separate process — potentially, even on a different machine. The kernel is where all the computational heavy lifting occurs. It is meant to be lean, mean and fast, with all the complex eye candy such as visualization passed off to the browser.
- A set of importable TDL modules and frameworks, including those contributed by other users, which your scripts can make use of.

To start the browser, you run the following command:

```
$ meqbrowser.py
```

...and if all goes well, a GUI window will appear.

The first thing the browser needs to do is to start a kernel process, which it offers to do through the dialog in Figure 1.1. Just click OK to start a local process.[1]

[1] The option to run a remote kernel was not fully functional at time of writing, so it was greyed out. If you see it available in your GUI, then your version of MeqTrees is a lot newer than this manual!

Figure 1.1: The “Connect” dialog

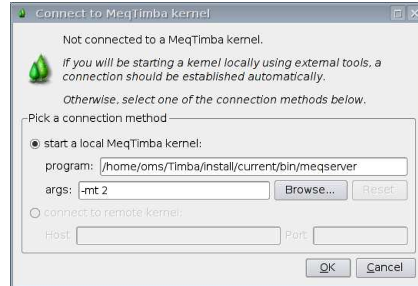
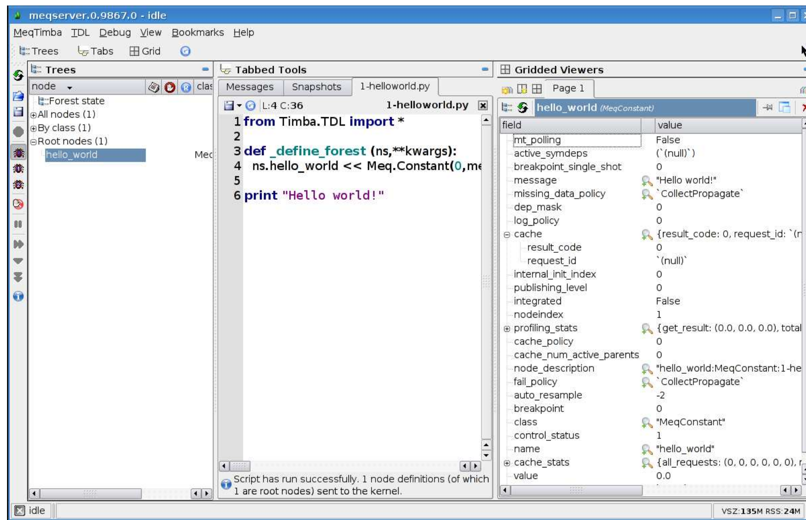


Figure 1.2: The browser, looking at the hello world node



Now we can load our Hello World script. Select the TDL | Load & compile TDL script menu option (or press Ctrl+T). If all goes well, you'll see a message to the effect that “script has run successfully”, and you'll see +Root nodes (1) show up in the “Trees” panel on the left side of the screen. Click on the icon next to the Root nodes label to open it up, then click on the hello_world entry, to see something like Figure 1.2.

1.2 Defining a node

Now, what do we see here? We can see that we've created one *node* called hello_world, of class MeqConstant. We can see its *state record* in the panel on the right. The latter is full of stuff we don't need to worry about at the moment,

but if you look carefully, you'll see a field called value, and a field called message, with values of 0 and "Hello world!", respectively.

This situation was brought about by this bit of code:

```
def _define_forest (ns,**kwargs):
  ns.hello_world << Meq.Constant(value=0,message="Hello world!");
```

Now, `_define_forest()` here is a predefined TDL name. When you run a TDL script inside the browser, it looks for a function by that name, and invokes it (try loading a script without a `_define_forest()`, and observe the results.)

When the browser calls `_define_forest()`, it gives it a parameter called the *node scope* (`ns`). The node scope is a magical little object that is pretty much at the heart of TDL. The `**kwargs`[2] entry serves to catch any optional named parameters which may be supplied by future versions of the browser, and which we needn't concern ourselves with here.

The main purpose of the node scope is to allow us to *name* nodes. We'll be examining scopes in more detail later on; for now, all that matters is that if `ns` is a node scope object, then `ns.`*`name`* declares a node called *"name"*. Since in Python terms everything is an object, `ns.`*`name`* is an object too. We call such an object a *node stub*.

Taken by itself, a node stub is not yet a real node, because we haven't described what kind of a node it is. We do this by *defining* a node. To define a node (i.e. to turn a node stub into a fully-fledged node on the kernel side), we create a *node definition* — in this case, by calling `Meq.Constant()` — and *bind* that definition to a node stub using the Python `<<` operator.[3]

There's some potential for confusion here, so this deserves to be reiterated. Nodes as such only exist on the kernel side of things. On the TDL side, they're represented by node stubs. To define a node on the kernel side, we have to make a node stub on the TDL side, and bind it with a definition. The node stub is the TDL equivalent of a kernel-side node, so sometimes the two terms are used interchangably. There is an important difference though – not all node stubs get turned into nodes. If you don't bind a node stub with a definition, it will remain a "potential" node in TDL, and no kernel-side counterpart will be created. When you do bind a node stub, the result is called an *initialized* node stub – and a "real" node counterpart will be created by the kernel.

To make a node definition, we need, as a minimum, to specify a node class — in this case, calling `Meq.Constant()` specifies a node class of `MeqConstant`. As you'll see in the examples later on, in simple cases the node class by itself is sufficient. Sometimes, though, you want to pass some optional arguments to a node. A `MeqConstant` node needs a value — this is passed in via the `value`

[2]If you're not familiar with the Python "**" syntax, now is the time to close this book again, and go back to *"The Python Programming Language"*.

[3]The normal meaning of `<<` is bitwise shift, which is only applicable to Python integers. TDL redefines this operator for node stub objects, so as to perform binding between a node stub and a node definition.

argument. You'll notice we also pass it a `message` argument. Do constants need messages? They don't — but you can pass in arbitrary arguments when defining a node, and if they're not recognized by the node, they will just sit in the state record for all to see. This is actually a useful feature, as we'll discover later when we look at node tags.

Now, what is the point of this whole Hello World exercise? Actually, there isn't one. As I said before, TDL isn't really meant for writing Hello World programs. If we want to see something interesting, we should go beyond one node, and make us a *tree*.

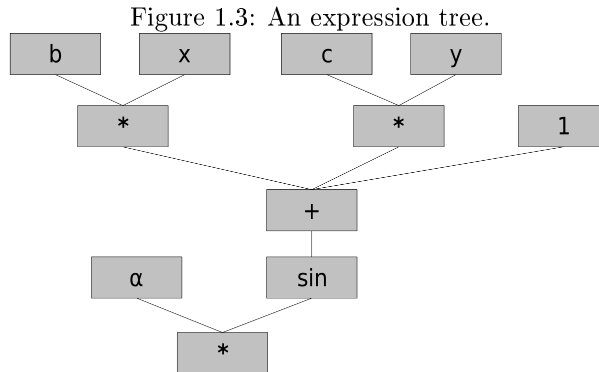
1.3 Our first "real" tree

Nodes can be connected together into a tree structure. If you've already read about or seen something of MeqTrees, then you'll know that the point of a tree is to evaluate a mathematical expression. Any expression can be represented by a tree. For example, the expression

$$f = \alpha \sin(bx + cy + 1)$$

can be represented by the tree in Figure 1.3.

Figure 1.3: An expression tree.



An expression tree[4] has the following features:

- nodes can have a parent-child relationship, indicated by connections on the diagram. *Parent* nodes are shown below their *child* nodes.
- *leaf nodes* are the nodes at the top having no children. A leaf node represents an atomic component of an expression, such as a constant ("1", in this example) or a parameter (α, b, x, c, y).

[4]MeqTrees are not quite *trees* in the pedantic sense, since we allow a child to be shared between multiple parents, and a tree — in the conventional definition of graph theory — doesn't. The correct term for the sort of structures created in MeqTrees is a *directed acyclic graph*. You will agree that it doesn't quite roll off the tongue like "tree" does.

- *functional nodes* have children. A functional node evaluates some kind of a mathematical function on the result(s) of its children, as determined by its *class*. In this diagram we see instances of add, multiply, and sine classes.

- a *root node* has no parents, its result is the result of the expression (f, in this case). Note that f here happens to be a multiplication node. Note also that we can choose to view any node as a "root" node, the result of which is determined by its *subtree* — the tree above it.

Given a tree, it is very straightforward to evaluate its result algorithmically, by starting at the leaf nodes, and propagating their values through to the parents. The power of MeqTrees lies in the fact that practically any expression can be represented by a tree — this is only limited by the availability of node classes for particular mathematical operations — and most tastes are catered for.

1.3.1 Defining trees

To implement the tree for our expression, we need to define multiple nodes, and somehow specify who is a child of whom. But let's start at the top of the tree, and define the leaf nodes first:

```
def _define_forest (ns,**kwargs):
  # this defines a leaf node named "alpha" of class Meq.Constant,
  # initialized with the given value
  # (i.e. the Fine Structure Constant, in appropriate units)
  ns.alpha << Meq.Constant(value=297.61903062068177);
```

This defines an "alpha" node as a constant with the given value. It would be quite tiresome if we always had to define constants in such a roundabout way, so TDL provides a shortcut:

```
  ns.b << 1;
  ns.c << 1;
  ns.x << 1;
  ns.y << 1;
```

Here, instead of creating a node definition, we simply give a numeric value, and bind it to a node stub via the `<<` operator. This is exactly equivalent to defining a `MeqConstant` the verbose way.

Next, we want to define a parent node to compute the bx product. Multiplication is done by the `MeqMultiply` node, so we have to define one and tell it that its childen are "b" and "x". This is easily done:

```
  ns.bx << Meq.Multiply(ns.b,ns.x);
```

To specify a node's children, we simply pass the node stubs `ns.b` and `ns.x` as (unnamed) arguments to the node definition call. In fact, there's an even easier way:

```
ns.cy << ns.c * ns.y;
```

TDL redefines Python's "`*`" operator so that when it is used with node stubs, it automatically produces a definition for a `MeqMultiply` node! The same applies to `+`, `-` (both subtraction and unary negation), `/`, `%` (modulo), and the built-in `abs()` function. This works for single operators and whole expressions, and makes it quite easy to define trees for simple arithmetic. E.g. we could then say:

```
ns.sum << ns.bx + ns.cy + 1;
```

to automatically define a `MeqAdd` node (two, in fact, since two additions are being done), and even another implict `MeqConstant` node to represent the "1". We could have even rewritten the whole thing as:

```
ns.sum << ns.b*ns.x + ns.c*ns.y + 1;
```

A compound expression like this has to implicitly define some intermediate nodes (e.g. for the *bx* and *cy* products). In the meantime, we've only explicitly named the "sum" node. Since every node must have a name, TDL will automatically generate names for the intermediate nodes. If you run the `Intro1/demo1-first-tree.py` script in the browser, and open up the Trees view (starting with Root nodes) on the left, you'll see these auto-generated names show up.

Don't get carried away though, automatic definition only works for basic arithmetic. To compute the sine, we need to define a `MeqSin` node the hard way:

```
ns.sin << Meq.Sin(ns.sum);
```

Of course, Python provides a `math.sin()` function for computing sines, but it would be a mistake to try to use it here. It's important to always realize that

TDL does not compute!!!

That is, the actual taking of the sine happens later, *on the kernel side,* when we evaluate the tree. Over on the TDL side, we're simply defining a sine node to perform this operation. This is why the built-in `math.sin` is inappropriate. If you say something like

```
ns.sin << math.sin(ns.sum);
```

...you'll get back a Python error, since it can't compute the sine of a node stub object. On the other hand, you can always say things like:

```
ns.foo << math.sin(1);
```

This will evaluate $sin(1)$ using Python, the result of which is a numeric value — which then hits the `<<` operator and gets turned into a `MeqConstant` definition.

Getting back to our tree, we finally need to define the root node, "f":

```
ns.f << ns.alpha * ns.sin;
```

or we could roll the sine and the product into a single statement:

```
ns.f << ns.alpha * Meq.Sin(ns.sum)
```

or even roll up the whole tree into a single statement:

```
ns.f << ns.alpha*Meq.Sin(ns.b*ns.x+ns.c*ns.y+1);
```

You'll note that here we're applying the multiplication operator "`*`" to a node stub (`ns.alpha`) and a node defintion (`Meq.Sin()`). TDL recognizes this kind of thing too, and automatically defines an intermediate node for the sine.

This completes our `_define_forest()` function, now it's time to see some results.

1.3.2 Evaluating trees

Now that we've put together our first tree, how do we get MeqTrees to compute the result? We have to tell the kernel to evaluate the tree. This is called *executing the tree*, and is a completely separate step from defining the tree.[5] Oftentimes, you will run a TDL script to define a tree, and then reevaluate that tree many times under different conditions.

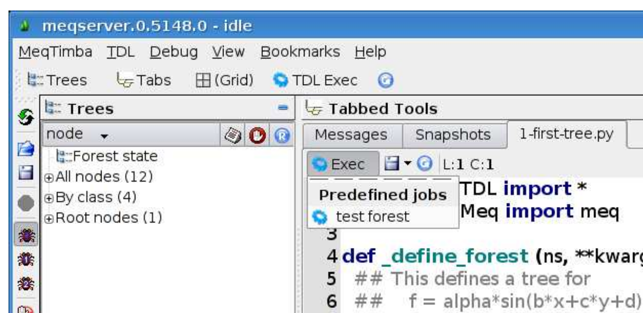
To execute the tree, we need to add an extra bit of code to our script:

```
# at top of file
from Timba.Meq import meq

# ...and somewhere else
def _test_forest (mqs, parent):
  domain = meq.domain(1,10,1,10);
  cells = meq.cells(domain,num_freq=10, num_time=11);
  request = meq.request(cells);
  result = mqs.execute('f',request);
```

[5]The term *running* a tree is occasionally used in place of *executing*. To most programmers, the two words mean pretty much the same. We prefer to *execute* our trees, so as not to confuse it with *running* TDL scripts (which only *define* trees.)

Figure 1.4: The script Exec menu.



Without going into details (of which there will be more than enough later), this function puts together a *request* object, and (with the final statement) sends it to a node of our tree, in this case "f" (it is also perfectly all right to send the request to any node — not just the root node — so as to evaluate its subtree separately.) On the kernel side, the request will be passed up the tree to the leaf nodes, telling them to return their values to their parents, telling the parents to compute their functions and return the results to *their* parents, and so on all the way back to the root node.

As you may have guessed, `_test_forest()` is another one of these predefined function names. How do we invoke this function? If you load and run the `Intro1/demo1-first-tree.py` script in the browser, you'll note a button labelled Exec at the top of the script editor window (There's actually two redundant buttons — the second one is labelled TDL Exec and is situated somewhat higher. The second button is convenient when you've hidden the editor window.) Pressing this button pops open a menu (see Figure 1.4) containing a "test forest" item. Clicking on this item will cause the browser to call our `_test_forest()` function. Basically, any time the browser loads a script with a `_test_forest()` function, it will place a link to it into the Exec menu. Other predefined names are also recognized: we'll see more of them later on.

1.3.3 Results & Bookmarks

Now, clicking on the "test forest" item was probably not very satisfying. You may have seen a message flash by to the effect that "tree has executed successfully", but that's all.

How do we see the result of our calculation? Actually, this isn't something you often need to do in real life, so the browser doesn't try to thrust it in your face. Usually you're more interested in having the result written out somewhere (e.g. to an AIPS++ Measurement Set.) On the other hand, even then one would still like to see something — some intermediate results for example — if only to verify that the tree is acting as expected, or to keep track of the data going by.

These can often be far more interesting than the final result itself.

In fact, MeqTrees goes a lot further than any software system (arguably, all the way), by making available the result of *each and every node* — effectively, each and every intermediate calculation. These can be examined by clicking around in the Trees view, as we'll see somewhat later. Of course, the result of each and every node is way more information than you'll ever need (unless you're debugging a tree that acts funny.) It's far more likely that there's only a handful of "interesting" nodes that you want to look at. For fast access to these, a TDL script can define one or more *bookmarks*, which show up in the Bookmarks menu at the top of the screen.

Now load up `Intro1/demo1-first-tree.py` if you haven't already done so, and run the tree by clicking on the "test forest" item in the Exec drop-down menu. Access the Bookmarks menu, and click on "result of 'f'". This is a bookmark to the root node of our tree. You'll see a panel show up in the Gridded Viewers pane on the right. Depending on how recent your installation of MeqTrees is — the viewer widgets are always a work in progress — this will either be a large green box (indicating a constant value), or a simple message. In any case you should be able to see the answer given by our tree:

$$42.$$

1.4 How to ask the right question

At this point you may be feeling somewhat underwhelmed, having gone through a lot of trouble for such a seemingly simple answer. But if you still remember your classics (???), you should immediately recognize that we simply don't know how to ask the right question yet.

Fortunately, formulating the right question in MeqTrees is not very difficult. The thing is, if you're only interested in scalar calculations, you don't really need to build trees — MathLab, or a Python prompt, or even a calculator is way more convenient. MeqTrees was really designed to work with functions, not scalars.

1.4.1 A world of functions

Most things we deal with in real life (insofar as radioastronomy and such can be considered "real life") are functions of something or other. A typical astronomical image gives flux or brightness a function of x, y (and perhaps ν — frequency). An interferometer observes correlations as functions of time t and frequency ν. Antenna gains can be complex functions of t, ν. Ionospheric phase (as seen by one antenna) is a function of direction and ν.

Now consider our original expression:

$$f = \alpha \sin(bx + cy + 1)$$

If we make x and y functions of time and frequency, then f itself becomes a function of time and frequency:

$$f(t,\nu) = \alpha \sin(bx(t,\nu) + cy(t,\nu) + 1)$$

Can we make a tree to compute f as a function of t, ν? *We already have, pretty much.* Every node in a tree can produce a function instead of a scalar, and the rest will compute the "right" result, regardless. In fact, scalars are just special cases of constant functions.

1.4.2 Representing functions

Of course a function in the mathematical sense is usually defined over a continuous — and thus infinite — space. The conventional way to represent functions in numerical computing, and the one used by MeqTrees, is to limit ourselves to a finite *domain* (e.g. $[t_{begin}, t_{end}] \times [\nu_{begin}, \nu_{end}]$), make a *gridding* of that domain ($\{t_1...t_n\}, \{\nu_1...\nu_m\}$), and then represent a function $f(t,v)$ by an $n \times m$ array of samples $\{f_{ij} = f(t_i,\nu_j)\}$. This of course applies to an arbitrary number of axes, not just two.[6]

To see this in action, load up `Intro1/demo2-improved-tree.py`, run "test forest", and look at the bookmarks.

1.4.3 So what's a function look like?

This script actually builds two trees — one rooted at node "f", the other at "f1". If you load up the "f" bookmark, you should see something like Figure 1.5 (and if you don't, you've probably forgotten to click on "test forest".)

Now, if you look at the plot, you will see that it displays "f" as a two-dimensional image in frequency and time. How did this time/frequency dependence come about? Look at the top of the script, and you'll see that the "x" and "y" nodes are no longer defined as constants:

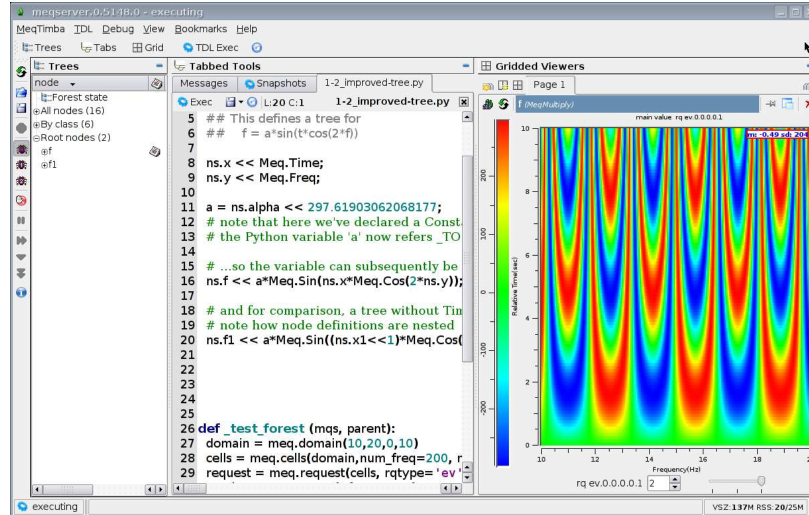
```
ns.x << Meq.Time;
ns.y << Meq.Freq;
```

The `MeqTime` node returns time as a function of time, the `MeqFreq` node likewise (note that when defining a node with no arguments, the `()` is not necessary — but we could have used `Meq.Time()` just as well.) So we have just defined

$$x(t) \equiv t,\ y(\nu) \equiv \nu$$

[6] MeqTrees supports up to 8 axes at a time, the meaning of which may be (re)defined in an arbitrary way. Frequency and time, being the two most popular ones, are the default for axis 1 and 2, and so will be used in most of our examples.

Figure 1.5: Viewing a function in the browser



Looking at the rest of the tree, the expression for f then becomes

$$f(t, \nu) = \alpha \sin(t \cos(2\nu))$$

Note that it's only the "x" and "y" nodes where we define a time/frequency dependence — it propagates through the rest of the tree auto-magically. For comparison, the tree rooted at "f1" is defined as

$$f_1 = \alpha \sin(x_1 \cos(2y_1)),$$

with $x_1 = y_1 = 1$. If you look at the bookmark for "f1", you'll see a constant result as previously.

On a side note, note this syntax:

```
ns.f1 << a*Meq.Sin((ns.x1<<1)*Meq.Cos(2*(ns.y1<<1)))
```

This takes advantage of the fact that the << operator returns the node stub itself as a result. This lets us name and define nodes directly inside an expression. Thus the line above is just a more concise version of the following piece of code:

```
ns.x1 << 1;
ns.y1 << 1;
ns.f1 << a*Meq.Sin(ns.x1*Meq.Cos(2*ns.y1))
```

1.4.4 Intermezzo: assigning nodes to variables

You may have noticed one other interesting twist in this script. At the start of `_define_forest()`, we define the "alpha" node as follows:

```
a = ns.alpha << 297.61903062068177;
```

Recall that the `<<` operator has a return value, namely the node stub itself. The statement above defines the node stub "alpha", and assigns it to the Python variable `a`. We can then reuse `a` anywhere in place of `ns.alpha`, in particular here:

```
ns.f1 << a*Meq.Sin((ns.x1<<1)*Meq.Cos(2*(ns.y1<<1)))
```

As we'll see later, this is a very useful feature of TDL, but it can cause some confusion to beginners, as I saw myself during our workshops. In particular,

```
a = Meq.Constant(value=0);
```

may look like it produces a node, but in fact it doesn't[7], no more than

```
Meq.Constant(value=0);
```

alone by itself does. *To define a real node, you must always do two things: make a node stub, and bind it to a definition with `<<`.*

1.4.5 Looking at other results

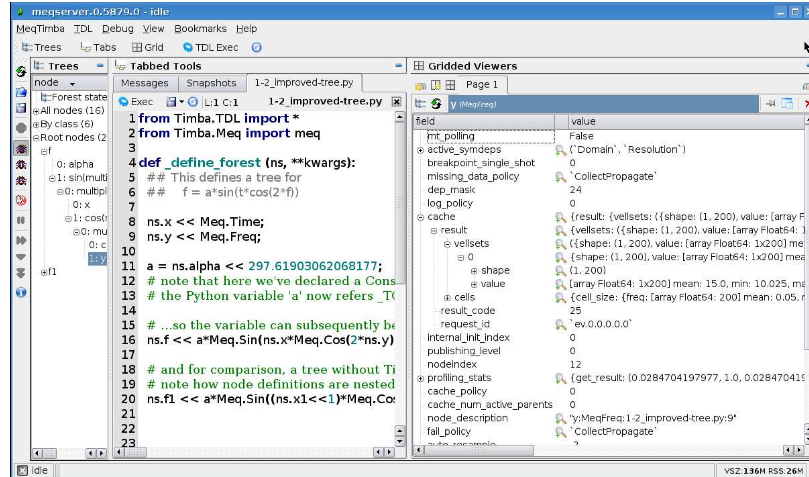
The browser provides a number of ways of looking at nodes and results. These are provided by different *viewer plugins*. The plots we accessed via the bookmarks were produced by the Result Plotter plugin. A more system-level view can be obtained via the Record Browser. If you left-click on a node in the Trees view, it brings up a Record Browser for that node's *state record*. If you right-click on a node, you can select a different viewer via the context menu.

The state record tells you more than you ever want to know about a node. For now, we're only interested in a field called cache.result. If you find node "y" and bring up its state record, then expand the cache.result field, you should see something like Figure 1.6.

The browser makes it so easy to examine the tree that it's easy to forget one important thing: what you're looking at is just a Python representation of something that's going on inside the kernel. When you click on a node, the browser sends a message to the kernel requesting that node's state, and receives a snapshot of the state in reply. We'll return to this distinction later on.

[7] What it actually does will be made clearer in Chapter 3.

Figure 1.6: Looking at a cached result



1.4.5.1 Axes of variability

The thing to note in Figure 1.6 is that the value field of the result is a 1×200 array. This is because the "y" node depends only on frequency, which is the second axis by default. In this case we say that the result has a single axis of variability, frequency. If you look at the "x" node, its value will be a 100×1 array (or a vector of length 100, which is really the same thing as far as MeqTrees are concerned), indicating that time (the first axis) is the only axis of variability. If you click through the cache.results of the *parents* of "y", you'll see the same 1×200 dimensions, until we get to the "multiply..." node upstream, where the "x" and the "y" branches come together. The value of this node is a 100×200 array, indicating a that both time and frequency are now axes of variability.

What this illustrates is that MeqTrees is very economical about representing functions — if a node's result has no variability along a particular axis, its size along that axis will be 1. But results with different axes of variability will be combined together and yield the right answer with the right number of axes.

If you've ever done any numerical code of your own, you've probably written endless `for` loops iterating over times, frequencies, coordinates, and whatnot. MeqTrees takes care of all of this for you, and even lets you go back and painlessly introduce an extra dimension somewhere at the start of the calculation — which in normal numerical code would require restructuring of every subsequent loop.

1.4.6 Domains, Grids, Cells

Where did the magical sizes of 100 and 200 for the time and frequency axes comes from? From this bit of code:

```
def _test_forest (mqs, parent):
  domain = meq.domain(10,20,0,10);
  cells = meq.cells(domain,num_freq=200, num_time=100);
  request = meq.request(cells);
```

The first line constructs a *domain* object, specifying the domain (in t, ν) over which we want to evaluate the tree. In this case t goes from 0 to 10, and ν from 10 to 20. The second line constructs a *cells* object. A cells consists of a domain and a gridding. This particular invocation makes a regular grid of 100 points in t and 200 points in ν, for the given domain.[8] Finally, we put the cells into a *request* object, which will then be passed to the root node of our tree.

1.4.7 Who knows what

"Who knows what" is a very important question in software design. In a well-designed system, each component should have the minimum information necessary to get its job done. If this "segregation of information" is not lovingly maintained, you'll see various assumptions creep in where they don't belong, each one undermining the flexibility of the final system.

We'll be touching on this question again later; in the meantime, let's review who in MeqTrees knew what in the examples we've looked at.

1.4.7.1 The user (i.e., ourselves)

We know all — or do we? Of course we know the script that built our trees, since we wrote it. On the other hand, somebody else could have written the script for us. We can then choose to treat it as a black box. The only thing we, the users, need to know then is to click on "test forest" to run the tree, and click on Bookmarks to look at results. If we're really interested in the internal structure of the tree, we can browse it in the Trees pane, and click on various nodes to discover everything there is to know about them.

There's an important principle here that was designed into MeqTrees from the very beginning — the user is allowed to know everything. Practically all the system internals are completely transparent and can be viewed from the browser. On the other hand, for users who just want a broad overview, there's an extensive system of visualizations and bookmarks to provide just that.

[8] In case you're wondering, it is also possible to specify irregular grids. And of course a domain need not be in frequency and time. We'll see examples of this later.

1.4.7.2 The tree

The tree, in fact, knows nothing, because the tree, as a separate entity, is practically non-existent. A tree is only a collection of nodes that happen to be connected. All the knowledge necessary for running the tree resides in the individual nodes. This is called the *locality principle*.

1.4.7.3 The nodes

What does a node know?

- It knows its class, i.e. what sort of operation it has to perform
- It knows its children
- It has some knowledge of its parents (mostly for housekeeping purposes such as caching)
- It knows its *state*, which can influence its behaviour

What a node *doesn't* know:

- It doesn't (usually) know the type of its children. All it knows is that they return some kind of result.
- It doesn't know anything about the tree, apart from its children or parents.
- It doesn't know anything about domains or grids. The domain and grid comes up in the request from its parent(s). It can make use of that information while it processes the request, but it doesn't usually retain it.

Recap

Here's a short review of what we learned in this chapter:

- MeqTrees consists of a fast kernel for doing the real work (building and running trees), and a graphical browser for looking at things.
- You build trees via TDL scripts. TDL scripts define the structure of trees. TDL is Python with some syntactic sugar thrown in.
- The purpose of a tree is (usually) to evaluate some mathematical expression. Most real-life expression will be functions of something or other, e.g. time, frequency, position, etc. A tree provides an efficient way to evaluate such functions "in bulk".
- You "run" a tree by giving it a domain and a gridding ("cells"), and it returns the value of the expression over that grid.
- With the browser, you can look at trees in great detail, and visualize everything.

Chapter 2

Growing Forests

The previous chapter's examples don't really answer the fundamental question of *What's the point?*

blah blah lots of trees blah blah. Basically some subset of the workshop demos. This may even grow into several chapters.

Chapter 3

The Python Perspective

This chapter aims to provide an in-depth description of how TDL does what it does, and how to make it do what you want. As a way of describing trees, TDL is (I hope) natural enough that you can get a very long way without any real idea of what's happening under the hood. The same of course goes for Python itself, or any other high-level programming language.

To be really proficient in a language, however, requires a solid understanding of the basics. What really happens when you define a node? What's the difference between a node and a Python variable? What happens when you pass a node to a function? What does the node scope *really* do, and why should you care?

If you can answer these questions, what have you gained? At least a few things:

- You'll think of faster ways to accomplish the same result, and with fewer errors
- Obscure error messages will cease to be so mysterious
- You'll find your scripts to be a lot more reusable down the line
- You'll have a far clearer idea of how frameworks like Meow work, and you'll be able to employ them better
- You may even make your own frameworks and modules.

All in all, you'll find yourself doing less actual work in TDL while putting it to much better use, leaving you with more time for getting interesting results!

And there's another good reason. Because *TDL is Python*, getting adept at TDL will automatically make you more proficient in Python, thus honing skills which can be quite useful in the world outside of MeqTrees (as much as we hate to admit that there is such a place...)

As a final warning, this chapter will certainly venture into the touchy territory of *good* (or even — ambitiously — best) *practices*. This is a subject that is

difficult to cover without hitting the occasional preachy note. As the person who is principally to blame for TDL in the first place, and having done probably more work in it — and, crucially, having made more mistakes — than anyone else up till now, I do feel somewhat qualified to preach on the subject. On the other hand, in a project that has always developed through something of a creative anarchy, there's not much place for gospel. Besides, as a language TDL is certainly too young for an anywhere-near-complete body of best practices to have emerged. So please feel free to do things differently and make your own mistakes (try not to repeat too many of mine, though) and refine your own practices. In the words of Captain Barbossa, "the [Pirate] Code is more what you'd call 'guidelines' than actual rules."

Reading the Python class documentation

The Python classes responsible for making TDL work reside in the `Timba.TDLimpl` module. The classes are called `_NodeScope`, `_ClassGen`, `_ClassStub`, `_NodeDef` and `_NodeStub`. You can find a detailed class reference here:

??????????????????????????????

It's hard to make much sense of the class reference without a basic understanding of how the classes work together. The latter is supposed to be provided by this chapter. After reading this chapter, you can go to the class reference to fill in the gaps.

3.1 Node scopes, stubs and definitions

If you recall our Hello World script (`1-helloworld.py`), we can see that the "tree", such as it is, is brought about by this bit of code:

```
def _define_forest (ns,**kwargs):
  ns.hello_world << Meq.Constant(value=0,message="Hello world!");
```

Now, we already know that `_define_forest()` is a predefined TDL name that the browser looks for when we load the script. The *node scope* (`ns`) argument is more precisely known as the *global node scope*. This is a "magic" object that's created by the browser. The purpose of `_define_forest()` is to populate the global node scope with node definitions, which the browser can then pass onto the kernel to construct a tree.

A crucial (for us) feature of Python is that function arguments are always passed by reference. This means that only one copy of the `ns` object actually exists, and any manipulations inside `_define_forest()` are done directly on the original object. Likewise, you can pass `ns` to your own functions, and it will still refer to the same original global node scope.

Note that *assignment in Python is not manipulation.* If your `_define_forest()` says

```
ns = 0;
```

then you haven't done anything to the original node scope object. All you've done is disassociate the local (local to `_define_forest()`, that is) `ns` variable from the node scope object, and associate it with the integer "0". The original node scope is still out there somewhere, you've simply discarded your reference to it. Similarly, saying

```
ns1 = ns;
```

does not make a new node scope object — all it does is give you two local variables that refer to the same object. This is really fundamental to Python, not just TDL.

The main purpose of the node scope is to allow us to *name* nodes. In Python, saying `ns.foo` means "give me the attribute named 'foo' of the object referred to by variable `ns`". An alternative syntax for this operation is `getattr(ns,'foo')`. "Normal" Python objects have a fixed set of attributes defined by their class. The node scope class redefines the get-attribute operation to do node naming. The naming operation works as follows:

- When you try to get the attribute "foo", it checks if a *node stub* called "foo" is already present.
- If it is, it returns that node stub.
- If not, then it creates a new node stub, and returns it.

Note that, once again, Python works by reference. There is only one node stub out there for "foo". If you say:

```
foo = ns.foo;
bar = ns.foo;
```

you now have two local variables, `foo` and `bar`, both referring to the same stub for node "foo".

The node scope class also redefines the `[]` operation to work in exactly the same way. The three expressions

```
ns.foo
getattr(ns,'foo')
ns['foo']
```

all produce exactly the same result. The second and third form allow one twist that the first form doesn't — that is, using *expressions* to "compute" node names. For example,

```
a = 'fo'
ns[a+'o']
```

is a fancy alternative to `ns.foo`. I'm sure creative minds can find a great use for this.

3.1.1 "Real" node scope methods

Note that node scopes also have a few "real" attributes (more precisely, methods) in the conventional Python sense. For example, `ns.Subscope()` invokes an actual method to create a subscope, instead of creating a node stub named "Subscope". Another example is the `ns.Search()` method, which can be used to search node scopes. Both of these operations will be discussed later.

The convention is that all "real" methods of the node scope object have names that start with a capital. To see a complete list of these reserved names, you can refer to the class documentation.

In the unlikely event that you actually do mean to make a node stub named "Subscope" (and are not worried by the ensuing terminology creep — but hey, it's a free country, presumably), you can always do it via the `[]` syntax:

```
ns["Subscope"] << Meq.Confusion(message="I've gone bonkers")
```

3.1.2 Objects and more objects

But what *is* a node stub? First of all, remember that *everything in Python is a first-class object*. So a node stub is an object — as it happens, of the `TDL._NodeStub` class. We'll study the class in more detail later, in the meantime, the crucial question is, what does a node stub really represent? Well, the answer is, it represents a *node name*. The name, by itself, is not yet a complete node — we have to associate it with a *node definition*.

What's a node definition? You guessed it, it's also an object (of the `TDL._NodeDef` class.) What does a node definition really represent? All the information required to create a node, except the name. As a minimum, this includes the *node class* (not to be confused with Python classes![1]), but it may also contain references to child nodes, and optional arguments. Node definition objects are created by calling something like `Meq.`*`Classname`*`(...)`. By itself, a definition is not a complete node either. The `<<` operator is how a node stub (i.e. the name) meets a defintion and becomes a fully-fledged node.

This is worth a summary:

- Every MeqTree node needs a name and a definition. The latter consists of a node class, (optional) references to child nodes, and (optional) keyword arguments.
- The name is unique but the definition isn't. In fact, you can define many identical (in all but name) nodes using the same definition object.
- Taken by itself, a node stub like `ns.foo` is only a "potential" node — it's really just a node name, nothing more.

[1]Node classes are only known to the MeqTree kernel. On the kernel side, they actually correspond to C++ classes. On the TDL side, a "node class" is just a string identifier.

- Likewise, on its own, a node definition object such as that returned by `Meq.Constant()` is also only a "potential" node. It is when the two are *bound* together with the `<<` operator that a complete node emerges.

- Consequently, you may litter your code with node stubs and node definitions that are not bound to anything, and none of these will cause any "real" nodes to be created. The following thoroughly useless bit of code creates an empty tree:

```
def _define_forest (ns,**kwargs):
  a = ns.foo;
  b = Meq.Constant();
```

3.2 What really happens during binding

Let's see how Python handles the statement:

```
ns.foo << Meq.Constant(value=0);
```

First, it evaluates the left-hand side, `ns.foo`. It tries to get the "foo" attribute of the node scope object, and gets back a node stub object.

Next, it evaluates the right-hand side. Here we have another "magic" TDL object called `Meq`. This is a *node class generator* object (`TDL._ClassGen`). We'll take a detailed look inside it in a bit. In the meantime, the expression `Meq.Constant` means simply "get the attribute named 'Constant' of the object referred to by `Meq`." Similarly to how node scopes return node stubs for their attributes, a class generator returns a class stub (`TDL._ClassStub`) when an attribute is accessed. Then, "`(value=0)`" means "treat whatever you got as a function, and call it with a keyword parameter, `value=0`." The end result of the expression is then whatever the function call returned. Which, due to some magic going on inside the class generator and class stub objects, happens to be a node definition object specifying a node class of `MeqConstant`, with an extra `value=0` argument.

Now Python has a node stub object on the left, a node definition on the right, and the "`<<`" operator in between. This operator normally means a bitwise shift, but the `TDL._NodeStub` redefines it to do something else. Because of this, Python calls a special method of the node stub object (called `__lshift__`, if you want to know), with the node definition as an argument. The node stub object then indelibly associates itself with the given definition, and now contains enough information to be later turned into a "real" node. This operation is also called *initializing* a node stub.

For the sake of illustration, here's an equivalent piece of Python code that makes every step explicit:

```
node_stub = ns.foo
# or equivalently: nodestub = getattr(ns,'foo')
class_stub = Meq.Constant
```

```
node_def = class_stub(value=0)
node_stub << node_def
```

3.2.1 Rebinding and reusing nodes

What happens when you refer to `ns.foo` again? Of course you get back a reference to the same node stub, which has now been initialized. We do this all the time when specifying the children of a node. In fact, a little-appreciated feature is that node stubs *need not be initialized before use*, as long as you remember to initialize them later. The following is perfectly legitimate:

```
ns.bar << ns.foo + 1;
ns.foo << Meq.Constant(value=1);
```

If you omit the second line, you'll get back an error message.[2]

What about doing << to an already initialized node stub? This is only allowed if the node definition is the same:

```
ns.foo << Meq.Constant(value=1);
ns.foo << Meq.Constant(value=1);
```

is OK, but

```
ns.foo << Meq.Constant(value=1);
ns.foo << Meq.Constant(value=0);
```

will produce an error. Personally, I consider rebinding to be bad practice. In some complicated cases, figuring out if a node definition is the same or not can be a dicey process, so I'd rather not rely on this feature.

3.2.1.1 The "**" operator: one-time binding

If you have a function for defining some part of the tree, and you call that function repeatedly, you probably want to initialize the nodes only once, rather than try to rebind them every time. TDL provides two convenient mechanisms for this. The first one is the "**" operator. This is similar to "<<", but it means "only bind if not already bound, otherwise ignore." So something like:

```
def make_tree_for_foo (ns):
  return ns.foo ** Meq.Constant(value=1);
```

[2] In the current version of TDL, this error message says something like "child n not initialized". This makes some sense: you have specified "foo" to be a child of "bar", but you forgot to initialize "foo". Still, I've noticed that this error tends to confuse people, so I may change it to something more lucid in the future.

means that `ns.foo` will be initialized as usual the first time the function is called. Subsequent calls will ignore the definition and return the already-initialized `ns.foo`. Note that a common gotcha with the "**" operator is that it has very high precedence (unlike "<<", which is very low). For example, the statement

```
ns.foo << ns.bar + 1
```

is (intuitively enough) equivalent to

```
ns.foo << (ns.bar + 1)
```

due to "<<" having a lower precedence than "+". By contrast,

```
ns.foo ** ns.bar + 1
```

actually corresponds to

```
(ns.foo ** ns.bar) + 1
```

and would probably result in an error! It's therefore a good practice to always parenthesize the right-hand side of the "**":

```
ns.foo ** ( ns.bar + 1 )
```

3.2.1.2 The initialized() method

The "**" operator is handy for one-line definitions, but what is you want to make a whole subtree for "foo", but only once? A useful mechanism is provided by the `initialized()` method of a node stub:

```
def make_tree_for_foo (ns):
  if not ns.foo.initialized():
    # define complicated subtree for foo
  return ns.foo;
```

The first time the function is called, `initialized()` returns `False`, so the body of the `if` statement is executed and "foo" is presumably initialized. For subsequent calls, `initialized()` becomes `True`, so the body is skipped and `ns.foo` is simply returned as is. If you combine it with the handy trick of assigning a node stub to a variable for later reuse (see below), you end up with:

```
def make_tree_for_foo (ns):
  foo = ns.foo;
  if not foo.initialized():
    # define complicated subtree for foo
  return foo;
```

which is my preferred pattern for making define-once functions. I prefer to *name* the top-level node stub only once, and use a local Python variable to refer to it everywhere else. With this pattern, if I later decide to rename the "foo" node to "foo_with_a_twist", I only need to change the first line of the function; with the previous pattern, I would have to edit all the `ns.foo` invocations.

3.2.2 Binding via = (assignment)

It is also possible to bind a node stub to a definition by assigning to a node stub:

```
ns.foo = Meq.Constant(value=0);
```

This is fully equivalent to

```
ns.foo << Meq.Constant(value=0);
```

The only difference is that the << operator returns the node stub itself as its result (see below), while assignment in Python is not an expression. That is, you can't say

```
a = (b=0) + 1;
```

in Python, as tempting as it may be to someone with a C/C++ background. You *can* say things like:

```
a = b = 0;
```

but that's just a shorthand for two separate assignments:

```
a=0; b=0;
```

Thus, the double assignment

```
ns.foo = ns.bar = Meq.Constant(value=0);
```

should initialize both "foo" and "bar" with the same definition.

Personally, I don't like to use assignment with node stubs, since it can be easily confused with assignment to a variable. TDL code is clearer when you use << for all your nodes. Also, something like

```
a = ns.foo = Meq.Constant(value=0);
```

produces something entirely different (and dangerously, unintuitively so) as compared to

```
a = ns.foo << Meq.Constant(value=0);
```

We'll take a closer look at this in section 3.4, when we talk about the use of Python variables to refer to nodes.

3.3 Hiding information

The hiding of information is one of the lesser appreciated virtues of good software design. When a framework appears to be rich in functionality and features, but proves to be clunky and difficult to work with in practice, it is more often than not due to too much information being revealed and/or shared.

A design based on many small components (i.e. functions and classes) that reveal and share minimal information will usually prove to be superior — in the sense of being easier to use and extend — to a functionally equivalent one that is based on a few larger aggregate components that try to do and reveal too much. One reason for this is something called *assumption creep*. When you write code based on an interface, you can't help making extra assumptions about

what goes on behind the scenes, some of which invariably go beyond what the original developer (which may even be yourself) intended. The result of this can be fragile code with a spaghetti-like web of dependencies that is hard to extend but easy to break. Assumption creep can be somewhat mitigated by good coding discipline, but truly disciplined programmers are few and far between (I certainly don't count myself in that category.) A good interface that hides more than it reveals can be a powerful deterrent all by itself. This may seem counterintuitive at first (you'd think "less information" = "more assumptions"), but is none the less true. The key word here is *good* interface. It's certainly possible to hide too much and/or to hide the wrong things. A "many small components" design may be thoroughly useless if the components hide the important things and reveal the trivial ones.

Striking the right balance between hiding and revealing information is a matter of judgement and hands-on experience. Nobody (that I know of) gets it right the first time anyway. My personal preference is to err on the side of caution — expose minimal information in the initial versions, and reveal more in later iterations as required.

The reason I bring the subject up here is that subsequent discussion will feature many examples of information being hidden, in ways both good and bad. The "hiding of information" point of view can (hopefully) provide an additional perspective on what constitutes good and bad TDL and Python.

3.4 Using Python variables and functions

TDL is Python. Everything in Python is a first-class object. I don't get tired of reiterating these two basic principles because they are effectively the two axioms of TDL from which everything else follows.

We may have redefined the `<<` operator in a funny way, but the things on the left- and right-hand side of the `<<` are still Python objects, and they are subject to the same rules as any other Python objects. For example, we can assign them to a Python variable.

With that in mind, try to guess what this bit of code really does:

```
foo = Meq.Constant(value=1);
```

Does this define any nodes? Of course not. All it does is call `Meq.Constant()` to create a node definition object, and associate that object with the Python variable `foo`. Because everything is a first-class object, everything and anything can be assigned to variables for later reuse. In this case, we've assigned a node definition to `foo`, so subsequently we can do:

```
ns.a << foo;
ns.b << foo;
```

...to create nodes named "a" and "b", and bind the same definition to both.

More usefully, a node stub such as `ns.a` is also a first-class object. We could have rewritten the above as:

```
a = ns.a;
a << foo;
```

The first statement here creates a node stub for "a" and assigns it to Python variable `a`, the second statement binds it with the node definition that we had previously assigned to `foo`. We've already seen things like this in a lot of the examples.

3.4.1 Using functions

All Python objects can be passed in and out of functions. Since everything is passed by reference, you really are dealing with the same object both inside the function, and outside, in the calling context. For example:

```
def my_node_definition ():
  return Meq.Constant(value=1);

def _define_forest(ns,**kwargs):
  ns.a << my_node_definition();
```

is a perfectly legitimate way of decoupling the definition of the node, and hiding it away inside a function. I've never found this particular pattern very useful, but you may.

3.4.1.1 Functions that define trees

A far far more important practice is illustrated by this example:

```
def define_tree_to_do_foo(node,a,b):
  node << a + b;
  return node;

def _define_forest(ns,**kwargs):
  define_tree_to_do_foo(ns.a,1,2);
  define_tree_to_do_foo(ns.b,3,4);
  define_tree_to_do_foo(ns.c,ns.a,ns.b);
```

Think carefully about what's happening here!

- We've **hidden** the knowledge of how to build a tree for this particular operation inside a function. This is a good thing — we can change the function later without changing the calling code.

- We don't want the function to worry about *naming* nodes, because we're going to call it many times, to produce differently named nodes. So we name the nodes on the outside, where we create the node stub, and then we pass the stub as an argument to the function. This is a very good thing, since it **hides** the business of naming from the function, and ultimately helps to avoid name clashes.[3]

Essentially, our function has become a *template* for making a tree. We've already seen this at work in the ionosphere example. I've thrown in the `a` and `b` parameters to show how the "template" can be made to depend on additional arguments. Can you guess what tree this script is going to produce? Load it up in the browser (`?????????.py`) to see if you guessed right.

There's an important lesson to be learned from `a` and `b` here. You don't specify parameter types when you define a Python function, and Python does no type checking at call time. So, what *can* you pass for `a` and `b`? The answer is, anything that'll work (as in, execute without an error). In the particular case of `define_tree_for_foo()`, this means anything that can be legally added together and bound to a node stub. The first two times we call `define_tree_for_foo()` with constants for `a` and `b`, and the third time with node stubs — and each time, the function just works as we would intuitively expect it to. This is great, since it makes our function all that more flexible.

Note also the "`return node`" statement at the end. The function will return the same node stub that was passed in. This is a convenience device. We ignore the return value in the example here, but in other contexts it may prove to be handy. For example, we could rewrite the three separate invocations above as one nested statement:

```
define_tree_to_do_foo(ns.c,
  define_tree_to_do_foo(ns.a,1,2),
  define_tree_to_do_foo(ns.b,3,4)
);
```

3.4.1.2 The reverse of the medal

This kind of delayed type checking is a fantastically powerful feature of Python. Used properly, it allows you to write much more flexible and useful code than you ever could with a statically-typed language. Like most power tools, it will happily inflict grievous bodily harm if you don't show it proper respect. Try adding the following line to the code above:

```
define_tree_to_do_foo(ns.d,ns.c,"x");
```

[3] You may wonder what happens if our function needs to name some intermediate nodes. The best way to avoid clashes then is to use subscopes, or to add qualifiers to the node stub that is passed in. This will be discussed in detail later on.

The function call executes fine, but inside the function Python throws an error. The error is obvious: you can't add a node stub to a string. The problem is, the "real" mistake occurs at the point where we call the function — but Python can't detect it until it is executing the function itself. The error message is thus misleading with respect to the actual cause.

That's no big deal in a short script like this, but imagine we were calling a function in somebody else's code, and got the parameter types wrong. We'd see a mysterious Python error within a completely unfamiliar piece of code. Only by looking back through the call stack could we get to the ultimate cause of the error, which is in our own code. Fortunately, the browser makes this easy — every error message shows an associated call stack, which you can quickly click through to see where the error may have arisen.

This is a particular headache if you're trying to write a module or a framework for the benefit of the world at large. In my personal experience, wrong parameter types — and the ensuing confusion of error messages — have been a constant pain to beginning users of Meow. It is possible to partially alleviate this by adding explicit type checking at the entrypoints to your functions, e.g.:

```
def define_tree_to_do_foo(node,a,b):
  if not ( (is_node(a) or isinstance(a,(int,float,complex))) and \
      (is_node(b) or isinstance(b,(int,float,complex))) ):
    raise TypeError,"define_tree_to_do_foo: node stubs or constants expected";
  node << a + b;
```

...but you can see all the signs of a losing battle here. Not only is it ugly and error-prone, but you can easily throw out the baby with the bathwater, reducing your functions to an inflexible, inelegant mess. My recommendation is:

- If you're a developer, do type checking only where it's worth it. That is, if it's not too much of an effort, or if you think particularly bad confusion may ensue (this, of course, is purely a judgement call, and can be difficult to anticipate), then by all means do it, otherwise don't bother.
- If you're a user of somebody else's code, do not be scared of mysterious errors, and always check the call stack before you panic. More often than not, the error is yours — and you can figure out exactly where it is by looking at the point where the call stack leaves your own code.

3.4.2 Using the value of <<

The redefined << operator, like every other Python operator (except assignment!), can have a return value. In TDL, the return value of << on a node stub is the node stub itself. So this bit of code:

```
  a = ns.foo << Meq.Constant(value=1);
```

does three things:

1. Creates a node stub named "foo".
2. Binds it to a node definition returned by `Meq.Constant()`.
3. Assigns the node stub (it being the return value of `<<`) to the Python variable `a`.

Point #3 is especially convenient, as it allows us to use the variable `a` later on when we want to refer to node "foo". I do this all the time, for the reason already mentioned — if I later need to rename the node "foo" to something else, there's only one point in the code that has to be changed. Think of it as another way of hiding information: you name the node "foo" in only one place, where you invoke `ns.foo`. As far as the rest of your code is concerned, the name is hidden behind the "`a`" variable.

In a previous section, I mentioned two similar-looking statements that do something completely different:

```
a = ns.foo << Meq.Constant(value=0);
```

```
a = ns.foo = Meq.Constant(value=0);
```

The first statement is "good": it initializes the node "foo" and assigns the node stub to variable `a`. The second statement does something way more confusing. Due to the way Python handles repeated assignments, it is actually equivalent to two separate assignments:

```
a = Meq.Constant(value=0);
ns.foo = Meq.Constant(value=0);
```

We've initialized node "foo" fine, but the variable `a` now refers to the node definition object, and not the node itself! I find this completely counterintuitive. The lesson here is to avoid multiple assignments in a single statement.

Another way to put the return value of `<<` to good use is through nested node definitions. Since the return value of `<<` is the node stub itself, we can immediately reuse it in another expression:

```
ns.sum << Meq.Add(ns.a<<Meq.Time(),ns.b<<Meq.Freq());
ns.other_sum = (ns.a<<1) + (ns.b<<2);
```

You can nest these definitions to any depth, although it makes for some very unwieldy statements if you take it too far.

Note that all this also applies to the "optional bind" operator, `**`.

3.4.3 Why you can't bind a node to a node

A common gotcha (at least in my experience of the workshops) is trying to bind a node stub to a node stub. The statement:

```
ns.foo << ns.bar
```

is bound to produce an error — the two things are two separate node stubs, so you can't just bind them together. The ensuing error message may be somewhat cryptic: "can't bind node name (operator <<) with argument of type _NodeStub". It is in fact an instance of a more broad class of error: trying to bind a node stub to something illegal. For example, the statement:

```
ns.foo << 'abc';
```

results in a similar error: "can't bind node name (operator <<) with argument of type str".

A more common occurence of the error is via something like:

```
ns.foo << define_tree_to_do_foo (ns.a,1,2);
```

The function already defines and returns a node stub (`ns.a`, in this case), so we can't just bind it to another node stub with `<<`. Usually the intent of a statement like this is to have something like an *identity* operation, i.e., trying to make node "foo" identical to node "bar". Of course, the nodes cannot be made truly identical — they have different names, for starters. You can achieve identity in a mathematical sense though, by creating a `MeqIndentity` node:

```
ns.foo << Meq.Identity(ns.bar)
```

A `MeqIdentity` simply passes along the result of its child as is. This trick may come in handy if you're trying to make a single node appear as a set of nodes (see the discussion on qualifiers below).

3.4.4 Python variables: conclusion

Now, all this may seem trivial, especially if you have a good understanding of Python, but it is really ubiquitous throughout TDL, and so it is vitally important to understand thoroughly. To reiterate:

- Everything in Python is a first-class object
- Node stubs, node definitions (and of course the node scope itself) are Python objects...
- ...and as such may be assigned to variables, passed into functions, and returned from functions.

These three points encapsulate pretty much everything you need to know about manipulating nodes in TDL! If you ever get confused in the future about what's happening in any given piece of code — and if you use a TDL framework like Meow, there will be a lot of this kind of manipulation going on — then you should go back and reread this section.

3.5 Node definitions, up close

We have sort of glossed over the point of how a node definition comes about. In the example above, the invocation of `Meq.Constant(...)` produced a node definition. If this looks to you like a function call, that's because it is. The `Meq` object comes from the `Timba.TDL` module, via the

```
from Timba.TDL import *
```

statement. It is also a "magic" kind of object, called a *class generator*. Any invocation of the form

```
Meq.Classname(arg1=value1,arg2=value2...)
```

produces a node definition for a node of class `MeqClassname`, with the given optional arguments. Note that the `Meq` object is not really aware of what node classes are valid, you could write something like

```
Meq.BadClass(...)
```

and TDL will accept it; but when the tree is sent to the kernel for creation, it will come back with an "Unknown class *BadClass*" error. The "magic" of `Meq` is that calling an arbitrarily-named method of the `Meq` object results in a node definition with the class given by that name.

Likewise, the optional arguments are not enforced — they are blindly inserted into the initial node state record. If an argument has a special meaning for a particular class of node (such as the `value` argument to a `MeqConstant`), then it will have an effect on the node, and will probably be checked for consistency when the node is finally created inside the kernel. Any meaningless arguments (e.g. the `message` we gave to `MeqConstant` in our hello world script) are left to sit and rot in the state record exactly as they came in. This can actually be quite useful, as we will see later on when we look at node tags.

Some node classes are an exception to this rule. In general, if a node class has an overly complicated behaviour (in terms of the available arguments), then the `Meq` object provides some explicit argument checking to make life easier. Some examples of this[4] are the `MeqParm` and `MeqSolver` node classes, which we'll learn about later. Another example is the `Meq.Matrix22()` "shortcut class" which can be used to construct a 2×2 matrix. Matrices are actually created by `MeqComposer` nodes with appropriate arguments; the `Meq.Matrix22()` method provides a convenient shortcut. You can see all available shortcuts here:

```
Timba/PyApps/src/TDL/MeqClasses.py
```

The mechanics of node definitions will become clearer as you see more examples and write some scripts of your own; in the meantime, all you need to remember is that calling any `Meq.Classname()` usually produces a node definition of class "*Classname*".

[4] At time of writing — it is possible that additional explicit definitions will be added to the `Meq` object at a later date.

3.5.1 Nested node definitions and automatic naming

Node definitions may also be nested to any depth. For example, consider the following statement:

```
a = Meq.Add(Meq.Freq,Meq.Multiply(Meq.Time,ns.foo,2));
```

By itself, this does not produce any actual nodes! Instead, it results in the following node definition: *"a node of class MeqAdd, with one child of class MeqFreq, and a second child of class MeqMultiply, itself having one child of class MeqTime, a second child which is node "foo", and a third child which is a MeqConstant with a value of 2."*

To turn this into an actual tree, the definition must be bound to a node stub:

```
ns.bar << a;
```

When binding the definition, TDL creates all required intermediate nodes, and automatically endows them with some [hopefully] reasonable names. If you look at the resulting tree, you will see a bunch of auto-generated nodes with names like "freq", "time", "multiply(time,foo,2)", etc.

What happens if you reuse the definition, e.g. say

ns.bar2 < < a;

later on in the code? The same process occurs all over again. In fact, some of the intermediate nodes may end up being reused. TDL will usually do a reasonable job of this, but it's not always perfect. I therefore recommend not overrelying on auto-naming, and have deliberately left the exact rules unspecified (since they may be revised and/or refined in future versions.) Note also that auto-naming may produce errors related to name conflicts, when similar expressions are used in different contexts. Should this occur, you're advised to split up the offending expression and name the conflicting nodes explicitly.

3.5.2 Automatic constants

TDL will also recognize a Python numeric value where it expects a node definition, and implicitly turn it into a definition like `MeqConstant(value=`x`)`. Integer and float values result in real MeqConstants, complex values result in complex `MeqConstants`.

If a constant node needs to be automatically named, the name is usually derived from the value of the constant. For example, the statement

```
ns.foo << Meq.Add(ns.bar,1.5)
```

will implicitly create a `MeqConstant` node named "c1.5". Constant nodes are reused if possible, so

```
ns.foo2 << Meq.Add(ns.other_bar,1.5)
```

will not create another `MeqConstant` node, but will instead reuse the previously created "c1.5".

Values with many significant digits will produce correspondingly long node names. For example,

```
ns.foo << Meq.Add(ns.bar,math.sin(1))
```

will probably implicitly create a node with the inscrutable name of "c0.8414709...". It's usually a good idea to name such complicated constants explicitly:

```
ns.sin_1 << math.sin(1);
ns.foo << Meq.Add(ns.bar,ns.sin_1);
```

This explicit naming also helps avoid potential name conflicts, which are always possible if two constants have very close values.

3.6 Using qualifiers

Qualifiers are one of the most powerful features of TDL. The example scripts we've seen before have hopefully made it clear how they work on an intuitive level. A piece of code such as

```
for k in TERMS:
  for l in TERMS:
    ns.f(k,l) << Meq.Polar(1,-2*math.pi*(k*x+l*y));
```

defines a series of nodes named "f:k:l" in a loop, for all possible combinations of k, l.

What happens on a more technical level? A statement like

```
ns.f(k,l)
```

invokes the *function call operator* (`__call__`) on the node stub `ns.f`. The function call produces a *new* node stub object, by appending the values of `k` and `l`[5] to the name of the original node stub, separated by "`:`". Note that the resulting node stub has absolutely no relationship to the original `ns.f`, apart from a meaningless (as far as the node stubs know...) similarity of names. The following three expressions all resolve to exactly the same node stub:

```
ns.f(1,2)
ns["f:1:2"]
ns.f(1)(2)
```

The second statement illustrates that qualifiers really are just a naming device. The third statement shows that qualifiers may be chained. It is quite obvious if you think about it: since `ns.f(1)` evaluates to a node stub, you can certainly repeat the qualification process by applying "`(2)`" to it, and — due to the way qualification works — the result is the same as saying `ns.f(1,2)` in the first place.

[5] Or more strictly, their string representation. Any Python object *x* has a string representation, which may be obtained by calling `str(x)`. For non-trivial types, this may be something unexpected. One exception is if an object defines a `name` attribute, then that name is used as the qualifier. Otherwise, it's a good idea to stick to numbers and strings for your qualifiers.

Qualifiers may also contain arbitrary keywords. For example, `ns.f(1,x=1,y=2)` will result in a node stub named "f:1:x=1:y=2", as will `ns.f(1)(x=1)(y=2)`. I have never found keyword qualifiers to be particularly useful, but some people swear by them.

3.6.1 Unqualified and underqualified nodes

After a loop like the one above, what does the `ns.f` node stub refer to? The answer is, nothing in particular — it is still an uninitialized node stub. When you build the tree, there won't be a node named "f" in it. You could also define it to something completely unrelated:

```
ns.f << 0
```

would make it a `MeqConstant` node. Since there's only a nominal relationship between `ns.f` and `ns.f(k,l)`, this is a perfectly legitimate thing to do.

On the other hand, `ns.f` can play a significant role in our code, since we know that we can qualify it to get at any node in the k, l series. Note that the knowledge of this is vested with us — we know that we can qualify `ns.f` with two numbers to get at one of the nodes in a series, because that's the way our script was written. The `ns.f` node itself has no knowledge of this. We could say something like

```
foobar = ns.f('foo','bar')
```

and get back a node stub named "f:foo:bar", with no-one the wiser. The `ns.f` node stub doesn't know (nor should it) whether "f:foo:bar" refers to a valid & initialized node or not. An error can only arise later, if we happen to use `foobar` in a context where a valid node is expected. For example,

```
ns.sum << foobar + ns.f(1,2)
```

will produce an error in the "sum" node, complaining that the first child has not been initialized.

Still, the `ns.f` node stub is an extremely useful object, because we can use it to refer to all the "f:k:l" nodes in bulk, as long as we remember to qualify it properly. When used like this, `ns.f` is called an *unqualified node.* This can be a very powerful technique, as illustrated by `Intro1/demo4-more_quals.py`:

```
def sum_series (sumnode,fnodes):
  sumnode << Meq.Add(*[fnodes(k,l) for k in TERMS for l in TERMS]);

def sum_sq_series (sumnode,fnodes):
  sumnode << Meq.Add(*[Meq.Sqr(fnodes(k,l)) for k in TERMS for l in TERMS]);

def _define_forest (ns, **kwargs):
  # (some code skipped)
  for k in TERMS:
    for l in TERMS:
      ns.f(k,l) << Meq.Polar(1,-2*math.pi*(k*x+l*y));
```

```
  sum_series(ns.sum,ns.f);
  sum_sq_series(ns.sum_sq,ns.f);
```

The `sum_series()` function implements summation over a k, l series. It has no knowledge of the series itself — all it knows is that it gets an `fnodes` object that it can qualify with a k, l pair (with k, l within some known limits) to get at the individual terms of the series. This is a good way of hiding information. In the context, we say that `sum_series()` expects an *unqualified node* for its `fnodes` parameter.[6]

On a similar note, what does `ns.f(1)` refer to? On the one hand, it's just a node stub named "f:1", which is not initialized to a valid node anywhere in our code. On the other hand, it's a means to refer to all the nodes named "f:1:l", as a series. In this latter context we call it an *underqualified* node, because we can append an extra qualifier to make it refer to a "real" (i.e. valid & initialized) node. We could write a function to sum over a one-dimensional series:

```
def sum_1d_series (sumnode,fnodes):
  sumnode << Meq.Add(*[fnodes(k) for k in TERMS]);
```

and then invoke it to make sums over slices over l through "f:k:l", for a given k:

```
  sum_1d_series(ns.sum1,ns.f(1));
  sum_1d_series(ns.sum2,ns.f(2));
```

3.6.2 Design by contract

You'd be right to wonder if we could somehow use the same function to sum over slices in the other direction, i.e. over all k's for a given l. The qualifier syntax only allows us to append qualifiers; there's no easy prepend feature. Python's absense of type checking comes to the rescue here. If you look at the code for `sum_1d_series()`, you can see that there's nothing in it that actually

[6] If the syntax

```
*[fnodes(k,l) for k in TERMS for l in TERMS]
```

looks unfamiliar, then you're really missing out on some great labour-saving Python tricks. In brief, the "`[`*expr* `for` *x* `in` *L [*`for` *y* `in` *M ...]* `]`" syntax is called *list comprehension*. It evaluates *expr* in a loop over all values of L (and M, if a second `for` clause is specified, etc.), while replacing all occurences of x (and y, etc.) in *expr* by the corresponding list elements. The result of this is a list composed of the values of *expr*.

The "`*`" before the list is a completely independent operation. This causes the function to be invoked with parameters composed from the given list. For example,

```
L = [1,2,3];
M = [10,20,30];
foo(*[x+y for x in L for y in M])
```

will call the function `foo` as `foo(11,12,13,21,22,23,31,32,33)`.

List comprehension (and the "`*`" parameter list syntax) is a very important technique to understand thoroughly, since it is ubiquitous throughout our TDL scripts. It really *does* make life easier!

requires the `fnodes` argument to be a node stub. All the function really requires is that `fnodes` be something that can be qualified with a single index (k) to yield a valid node stub. If we had a function that took a single argument — the missing k qualifier — and returned a fully-qualified node stub, we could pass that function as the `fnodes` parameter, and `sum_1d_series()` would do exactly the right thing. Python's *lambda* syntax makes it easy to construct such a function on-the-fly:

```
sum_1d_series(ns.sum1,lambda k:ns.f(k,1));
```

This defines an anonymous function[7] , with one argument, `k`, returning `ns.f(k,1)`. Inside `sum_1d_series()`, this lambda-function then behaves in exactly the same way that an underqualified node stub would have.

This technique is an illustration of something called *design by contract*. Instead of specifying that our `sum_1d_series()` function takes an argument of type node stub ("design by type"), we only specify that the parameter should implement some kind of behaviour — that it behaves "like" a node stub w.r.t. qualification. This behaviour requirement is called a *contract*. In this particular case, the contract on `fnodes` is that it supports the function call operator, and that `fnodes(k)` returns a valid node stub.

Design by contract is made possible by Python's late type-checking mechanism. It is a very powerful technique: by not tying our interfaces to specific types, it allows us to keep them all that more flexible. On the other hand, it carries with it all the risks discussed in section 3.4.1.2. These can be mitigated by making sure that your interfaces are built around a limited set of fairly simple and well-defined contracts. The contract on `fnodes`, above, is a good example, and is used extensively in frameworks like Meow (which we will look at in a subsequent chapter).

3.7 Built-in node arithmetic

Built-in arithmetic removes some of the tedium of putting together complicated expressions. A statement like:

```
ns.sum << (ns.foo + ns.bar)/2;
```

is a lot easier on the eye than its verbose equivalent:

```
ns.sum << Meq.Divide(Meq.Add(ns.foo,ns.bar),Meq.Constant(2));
```

Of course, the above statement doesn't really perform any arithmetic. Remember, *TDL does not compute*. Instead, the statement "unrolls" into a little tree definition for computing the indicated expression.

The advantages of built-in node arithmetic are obvious: they make TDL code easier to write and more intuitive to read. The dangers are somewhat more subtle:

[7] The term *lambda* comes from functional programming, which ultimately got it from the λ operation in functional calculus.

- the intermediate nodes of an expression will receive automatic names, which can be hard to interpret for the user.
- it is not always obvious which parts of an expression are "true" arithmetic that is evaluated in Python, and which ones correspond to tree declarations.
- since intermediate objects are almost always involved, misuse of arithmetic can lead to some pretty cryptic error messages.

This section aims to provide a basic understanding of how node arithmetic *really* works, which will (hopefully) help you to avoid the disadvantages listed above.

3.7.1 Arithmetic recognized by TDL

TDL recognizes the following Python operations, translating them into appropriate node definitions:

- Standard binary arithmetic:

  `+` translates into a `MeqAdd` node

  `-` translates into a `MeqSubtract` node

  `*` translates into a `MeqMultiply` node

  `/` and `//` both translate into a `MeqDivide` node. Note that Python uses `//` for "floor division"; in TDL both version of the operator have the same meaning, that of regular division.

  `%` ("modulo division") translates into a `MeqFMod` node

- Unary minus ("`-`") translates intop a `MeqNegate` node
- The built-in `abs()` function translates into a `MeqAbs` node.

<u>No other built-ins are recognized!</u> A common beginner's mistake is to think that TDL is not just smart, but omniscient:

```
import math;
ns.foo << math.sin(ns.bar);
```

This fails with a somewhat mysterious error message. Python's `math.sin()` function has no clue what to do with node stub objects, and there's nothing TDL can do about that.

3.7.2 When is TDL arithmetic invoked?

In order for TDL to recognize an arithmetic expression as a tree declaration, some component of that expression *must be either a node stub, or a node definition object.* For example, the expression

```
ns.foo + 1
```

invokes TDL arithmetic, because the object on the left-hand side of the "+" is a node stub. Likewise,

```
2*Meq.Freq()
```

is recognized, because the right-hand side is a node definition. On the other hand, something like

```
1+2
```

is evaluated entirely in Python, since the expression contains no TDL objects.

Of course, the same rules apply when TDL objects are assigned to Python variables. The first two expressions above are equivalent to:

```
a = ns.foo
a + 1
b = Meq.Freq()
2*b
```

Where complicated expressions are involved, they're parsed by Python and TDL is invoked in the appropriate places. For example,

```
ns.foo + 2*(Meq.Freq()+(1+2))
```

translates into

```
Meq.Add(ns.foo,Meq.Multiply(2,Meq.Add(Meq.Freq(),3)))
```

...since "1+2" is immediately evaluated in Python, and the rest is turned into a tree definition.

Note also that binary arithmetic is always binary, and parsed left-to-right (following the usual operator precedence rules). An expression like

```
ns.foo + ns.bar + ns.bar2 + 1
```

translates into the rather inelegant:

```
Meq.Add(Meq.Add(Meq.Add(ns.foo,ns.bar),ns.bar2),Meq.Constant(1))
```

With a long sum like this, you're better off doing all the addition within a single node by explicitly invoking `Meq.Add()`, e.g.:

```
Meq.Add(ns.foo,ns.bar,ns.bar2,1);
```

3.7.3 Intermediate nodes

Once arithmetic operators have been translated into appropriate node classes, the result is no different from a nested node definition. For example,

```
ns.foo + 2*(Meq.Freq()+(1+2))
```

translates into the following nested definition:

```
Meq.Add(ns.foo,Meq.Multiply(2,Meq.Add(Meq.Freq(),3)))
```

From that point on, these definitions are handled exactly as described in section 3.5.1, and are subject to exactly the same caveats.

Chapter 4

What's In a Name?

Naming can be a difficult business — just ask Ford Prefect (???). For various good reasons that we will not go into too deeply, every node in the tree must have a unique name. It would be easy to generate unique names automatically, but these would be largely meaningless to the user. Good, meaningful node names are important, because they make a tree easier to understand. This in turn makes it easier to indentify intermediate values, keep track of what's going on, and locate potential errors.

Real-life trees can have thousands of nodes, and it would be rather tedious if we had to choose names for them all. TDL provides a number of convenient shortcuts. First of all, repeated structures in the tree (such as per-baseline, per-source and per-station branches) can be given different qualifiers. Then there's implicit arithmetic and automatic node naming, which assigns names to intermediate nodes for us. Recall our first "meaningful" tree of section 1.3, which implemented the expression:

$$f = \alpha \sin(bx + cy + 1).$$

In TDL, such a subtree can be defined by a single statement:

```
ns.f << ns.alpha*Meq.Sin(ns.b*ns.x + ns.c*ns.y + 1)
```

...provided we have already defined the `alpha`, `b`, `c`, `x` and `y` nodes. All the intermediates nodes in this expression will be assigned names automatically, based on their class and arguments. If you study these names in the browser, you'll see a clear meaning to them.

Is it a good idea to rely on these automatic names? In general, they tend to make life a lot easier, but, like any good idea, they can be taken too far. For example, the following is perfectly legitimate TDL:

```
ns.f << 297*Meq.Sin(1*Meq.Freq() + 2*Meq.Time() + 1)
```

...but the resulting tree will be much more difficult to understand.

In general, I would recommend the following guidelines:

- If a node represents some well-defined physical or mathematical quantity, name it explicitly.
- If a node represents something you might want to look at separately, name it explicitly.
- Otherwise, allow automatic naming to take care of it.

Striking the right balance here is a matter of feeling and personal preference. Trees that explicitly name every single node are easy to understand, but take a lot more TDL code (which in itself can be difficult to read). Trees that explicitly name barely anything at all may be easy to code, but will be perfectly inscrutable.

Of course, picking the right name is only part of the problem. Imagine you're using a framework or a module that was written by Bob. This module will build a large part of the tree for you, using some naming scheme invented by Bob. The details of this scheme are hidden inside Bob's code, so you could very well end up using the same names for your own, compelely unrelated nodes. All sorts of confusion may ensue, all the worse if Bob (and/or you) is sloppy in his naming, or if both of you are fond of generic names like "x" and "y". Then you get even more ambitious, and try to make use of another module, this one written by Alice, with her own naming scheme. There's now three potential sources of conflicting names. Finally, you pull out all your hair and swear off other people's code for life...

This scenario shows that using good naming schemes, and avoiding naming conflicts, is terribly important — the more so as your scripts get more complex and as you start sharing code with other people. This is a matter of good coding practice. Fortunately, TDL provides a number of tools that, if properly used, let you implement good naming practices will relatively little effort. The purpose of this chapter is to discuss these tools and practices in depth.

4.1 Qualifiers as naming devices

If used properly, qualifiers will help avoid most naming conflicts. We have already seen many examples of this. Some of the more straightforward ones are:

- Using a station index and a station pair to distinguish per-station and per-baseline nodes.
- Using a "source name" qualifier to distinguish nodes related to a particular source. Meow makes heavy use of this, by forcing you to assign a unique name to each "sky component".

4.2 Node scopes

Up till now, we've only dealt with the *global node scope*, a.k.a. the `ns` object. The good news is that we can make use of other node scopes. If you're familiar with C++ namespaces, or just name scopes in general, then this concept should be quite familiar. For example, in the following bit of (rather silly) Python code, the name `foo` appears twice:

```
def a():
  foo = 1;
  return foo;

def b():
  foo = 2;
  return foo;
```

...yet there's no confusion at all between the two `foo`'s, because one resides in the local scope of function `a()`, and the other in the local scope of `b()`. In C++, you could have something like:

```
namespace a
{
  int foo;
}
int foo;

int b ()
{
  int foo = 2;
  return foo + ::foo + a::foo;
}
```

Here we have three separate `foo`s. One is declared in *namespace* a, another in the global scope, and the third in the local scope of function `b()`. It is even possible to explicitly refer to the "other" `foo`s inside of `b()`, by prepending "`::`" and "`a::`" to the name.

TDL provides a similar (though more powerful) mechanism via node scopes. Think of a node scope object as a naming device. The global node scope is the equivalent of global name scope of most programming langauges. If `ns` is the global node scope, then `ns.foo` refers to the name "foo". The name "foo" is global in the sense that anyone else attempting to define a node named "foo" will end up referring to the same name.

4.2.1 Subscopes

If we want make different distinct "foo" nodes, we can make use of subscopes. This bit of TDL code:

```
nsa = ns.Subscope("a");
nsb = ns.Subscope("b");
ns.foo << 1;
nsa.foo << 2;
nsb.foo << 3;
```

will define three nodes named "foo", "a::foo", and "b::foo". The `nsa` and `nsb` objects are called *subscopes.* When you declare node stubs inside a subscope, the name of the subscope is prepended to the node name, separated by "::". Note that this is purely a naming trick. You could declare something like

```
ns1 = ns.Subscope("a");
ns2 = ns.Subscope("a");
ns1.foo << 1;
ns2.foo << 2;
```

and get back a "node redefined" error, because while the `ns1` and `ns2` subscopes are, technically, different Python objects, they name their nodes the same way. Likewise, if `ns` is the global node scope, then `ns['a::foo']` is yet another way to refer to "a::foo". The bottom line is that all node names, no matter how you put them together, still reside within a single global name repository, and as such are subject to potential conflicts. Subscopes are simply a handy way to automatically prepend something to a group of names. This is similar to what happens with qualifiers. For example, if `nsa` is a subscope of `ns` named "a", then

```
nsa.foo(1)
ns['a::foo'](1)
ns['a::foo:1']
```

are simply three different ways of referring to the same name, "a::foo:1". Subscopes and qualifiers are merely naming devices, so it is important that you understand exactly what they do and don't do!

4.2.2 A subscope is just a scope

The power of subscopes lies in the fact that they're entirely indistiguishable, from a program's point of view, from the global node scope. All node scopes are instances of the `TDL.NodeScope` class. When you pass a node scope to a function, the function has no way of knowing whether it's dealing with the global scope or a subscope, because the object "looks and feels" the same way. This is another instance of "good" information hiding.

Consider our Alice/Bob scenario. Say we're trying to make use of Alice's module, `alice.py`, which contains

```
def make_useful_subtree (ns,node,a=1):
```

```
ns.foo << a * Meq.Time();
node << ns.foo + Meq.Freq();
return node;
```

Obviously, Alice is aware of the good practice of letting the caller create and pass in the "output" node stub as the `node` argument. Then, we're also trying to pull in `bob.py`:

```
def make_really_useful_subtree (ns,b=1):
  ns.foo << b*2;
  ns.bar << ns.foo + Meq.Time();
  return ns.bar;
```

Bob's module prefers to declare its own output node stub. When we try to combine the two in our own code, we start out with something like:

```
import alice;
import bob;
x = alice.make_useful_subtree(ns,ns.x,a=1);
y = bob.make_useful_subtree(ns,b=2);
ns.bar << x + y;
```

This produces at least two naming conflicts — both Alice and Bob use "foo", and both Bob and us use "bar". The easiest way to resolve this problem is to put Alice and Bob into separate subscopes:

```
x = alice.make_useful_subtree(ns.Subscope("alice"),ns.x,a=1);
y = bob.make_useful_subtree(ns.Subscope("bob"),b=2);
ns.bar << x + y;
```

All Alice's nodes will now be called "alice::...", all Bob's nodes will be called "bob::...", and we're free to use the global scope for our own nodes without worrying about naming conflicts.

This is great, because it allows us to treat Alice and Bob's modules as black boxes (which we ought to be entitled to), and not worry about how they name their nodes internally.

4.2.3 Qualifying subscopes

When creating a subscope, you can also pass in arbitrary qualifiers:

```
nsa = ns.Subscope("a",1,x=2);
nsa.foo << 1;
```

This creates a subscope named "a:1:x=2", and a node named "a:1:x=2::foo". Again, this is only a naming convenience — we could have achieved exactly the same result by saying:

```
nsa = ns.Subscope("a:1:x=2");
nsa.foo << 1;
```

4.2.4 QualScopes

A *QualScope* is a variation on the subscope theme. You can create a qualscope with an arbitrary set of qualifiers:

```
nsq = ns.QualScope("x",y=1);
nsq.foo << 1;
nsq.bar(2,a=3) << 1;
```

...and the qualscope's qualifiers will be *prepended* to the set of a node's qualifiers when creating nodes within that qualscope. Thus, the example above produces nodes named "foo:x:y=1" and "bar:x:y=1:2:a=3".

Qualscopes are very useful when you know you will be creating a large number of nodes sharing the same qualifiers. For example, classes in the the `Meow.SkyComponent` hierarchy, which represent individual sources on the sky, make themselves a qualscope using the source name as a qualifier. All nodes related to that component are then created within that qualscope, and thus are automatically qualified with the source name.

4.2.5 Nesting scopes

Just like subscopes, qualscopes are also fully-fleged scopes. This means you can do anything with a qualscope that you could with the global node scope, including creating another qualscope or subscope within it. This leads to *nested* scopes:

```
nsa = ns.Subscope("a");
nsq = ns.QualScope("x",y=1);
nsq1 = nsa.QualScope("z",t=2);   # nested qualscope
nsb = nsq1.Subscope("b");        # nested subscope
nsq2 = nsb.QualScope("x",y=2);
nsa.foo << 1;                    # creates "a::foo"
nsq.foo << 1;                    # creates "foo:x:y=1"
nsq1.foo << 1;                   # creates "a::foo:z:t=2"
nsb.foo << 1;                    # creates "a::b::foo:z:t=2"
nsq2.foo << 1;                   # creates "a::b::foo:z:t=2:x:y=2"
```

Having too many nested scopes can look confusing, but you don't generally need to worry about it. The important thing to know is that you can always make a subscope or qualscope from any given scope object. Returning to our Alice/Bob example, if Alice's function were to look like this:

```
def make_useful_subtree (ns,node,a=1):
  qualscope = ns.QualScope(alice=a);
  qualscope.foo << a + 1;
  node << ns.foo + Meq.Freq();
  return node;
```

...it would work perfectly fine whether invoked as

```
  alice.make_useful_subtree(ns,ns.x,a=1);
```

...or as

```
  alice.make_useful_subtree(ns.Subscope("alice"),ns.x,a=1);
```

In the first case the foo node will be named "foo:alice=1", in the second case "alice::foo:alice=1".

The bottom line is, if you're Alice or Bob — writing modules for other people to use — you're free to make sensible use of subscopes and qualscopes within your own module, as long as these are derived from the `ns` object that the user of your code passes in. The user can then decide to "sandbox" your module into a subscope as he or she sees fit; your own subscopes and qualscopes will then nest inside the subscope that the user has selected for you, but it's not something you need to concern yourself with at all.

4.2.6 Scopifying nodes

You can also derive a subscope or a qualscope from a node stub. This is called *scopifying a node.* Calling `node.Subscope(*args,**kw)` creates a subscope based on the full name of the node stub (including any qualifiers), with optional extra qualifiers tacked on:

```
  nsf = ns.foo.Subscope();          # creates subscope "foo::"
  nsf.bar(2) << 1;                  # creates "foo::bar:2"
  nsf1 = ns.foo(1).Subscope();      # creates subscope "foo:1::"
  nsf1.bar << 2;                    # creates "foo:1::bar"
  nsf2 = ns.foo(1).Subscope(2,x=3); # creates subscope "foo:1:2:x=3::"
  nsf2.bar << 3;                    # creates "foo:1:2:x=3::bar"
```

Likewise, `node.QualScope()` creates a qualscope using the node's qualifiers:

```
nsf = ns.foo(1,2).QualScope();
nsf.bar << 1;                   # creates "bar:1:2"
```

Scopifying a node is an especially elegant way to avoid naming conflicts. For example, Alice could rewrite her `make_useful_subtree()` function as follows:

```
def make_useful_subtree (node,a=1):
  ns = node.Subscope();
  ns.foo << a * Meq.Time();
  node << ns.foo + Meq.Freq();
  return node;
```

Note the difference with the previous version — we no longer need to pass in an `ns` object. Instead, the node scope for intermediate nodes is derived from the output node. If we invoke Alice's function as, e.g.:

```
  make_useful_subtree(ns.bar,a=2)
```

her nodes will be named "bar::foo", etc. This simplifies the interface (one less parameter to keep track of), while effectively sandboxing Alice's nodenames.

Chapter 5

Node tags and scope searches

As you start building more and more complicated trees, you will run into the problem of keeping track of the important nodes. It's great that you can pass around a whole subtree just by referring to its root node, and we use it all the time in patterns like:

```
foo = make_subtree_for_foo()
bar = make_subtree_for_bar()
ns.sum << foo + bar;
```

This hides the structure of the subtree inside a function or a module, and is normally a good thing, allowing us to change the functions without worrying about the caller. On the flip side, there may be some "interesting" nodes inside the subtree that we *do* want to know about on the caller side:

- the subtrees may cointain solvable parameters, and we need to know what these are if we want to make a solver tree. Unfortunately, these can be anywhere in the subtree.
- the subtrees may contain nodes of interest for the user, and we may want to present them in a bookmark. For example, the ionosphere trees that we looked at earlier return nodes for applying ionospheric phase, which is all that's needed for most calculations. The user, on the other hand, may want to look at the TEC values from which these phases are derived. These latter are hidden away inside the subtree.

Some of our early TDL code is littered with various more or less elegant kludges designed to work around this problem. For example, you can have your function build up a list of "solvable" nodes, and return that to the caller. These kludges do get the job done, but they complicate the interfaces and make code less portable. Subsequently, we came up with the idea of *node tags* to provide a more elegant solution to the problem.

5.1 Node tags

A tag is just a short string that can be associated with a node. A node can have multiple tags. Tags are specified when defining a node, via a "`tags`" keyword:

```
ns.foo << Meq.Constant(0,tags="constant");
ns.bar << Meq.Constant(1,tags=["constant","my_tag"]);
ns.foobar << Meq.Constant(2,tags="my tag");
```

The first statement defines node "foo" with the single tag "constant" . The second statement defines node "bar", and tags it with "constant" and "my_tag". The third statement defines "foobar", and tags it with "my" and "tag". Do note this last example: tags cannot contain whitespace, so a single `tags` string will always be split up on whitespaces. This is a handy way to assign multiple tags without the clutter of lists (as in example 2).

Apart from this automatic splitting on whitespace, TDL itself does nothing else with tags. If you look inside the state record of a tagged node, you will see a "`tags`" field containing the tags you supplied. Tags are pretty much meaningless to TDL, they are just arbitrary identifiers that you can use to find the nodes later.

5.2 Searching for tags

To find nodes with a certain tag, you simply call the `Search()` method of any node scope object:

```
ns.Search(tags='constant');
```

This returns a Python list (possibly empty) of all the node stubs that have been defined with a "constant" tag. If multiple tags are specified, the result is a logical "AND" operation:

```
ns.Search(tags='jones solvable');
```

returns a list of all node stubs having the tag "jones" AND "solvable". A more verbose version of the same statement would be:

```
ns.Search(tags=['jones','solvable']);
```

You can also use Python regular expressions when specifying a search tag:

```
ns.Search(tags='jon.* solvable');
```

This returns a list of all node stubs having a tag that begins with "jon", and the tag "solvable". You should refer to the Python Library Reference, module "`re`" for details on regular expression syntax. Regexes provide great flexibility in searches. For example, the following will search for tags "jones" OR "solvable":

```
ns.Search(tags='jones|solvable');
```

Tags are away of revealing important information on a tree, while hiding irrelevant details. For example, imagine that Alice has given you a module for

computing beam patterns. You want to use Alice's beams in your tree, and you want to solve for beam parameters. These parameters are sitting somewhere deep in Alice's trees, and you don't really need to know where, you just want to know what they are, so you can feed them to a solver. If Alice has read this manual (and is endowed with a little foresight), then she has probably seen fit to tag her parameters in some meaningful way, e.g. with "beam" and "solvable", and mentioned it in her documentation. Knowing this, you can do a simple search to find all solvable beam parameters, without knowing anything else about Alice's trees.

5.3 Searching within subtrees and families

`ns.Search()` by default searches the entire node scope. A lot of the time this is NOT what you want to do — after all, if you're trying to find Alice's beam solvables, you only want to search within Alice's subtrees, and not anywhere else. You can use the `search()` method of a node stub to limit the search to the subtree starting at a particular node:

```
ns.foo0 << Meq.Constant(0,tags="constant");
ns.foo1 << Meq.Constant(1,tags="constant");
ns.bar << Meq.Constant(2);
ns.foobar << ns.foo0 + ns.bar;
ns.foobar.search(tags="constant");
```

The `search()` method of a node stub behaves like the "global" `ns.Search()`, but the search is limited to the subtree rooted at that node. The search operation above will return "foo0", but not "foo1".

Note that the starting node of a search may also be unqualified. Consider this example:

```
for k in range(10):
  ns.foo(k) << Meq.Constant(k,tags="constant");
  ns.bar(k) << ns.foo(k) + 2;
ns.bar.search(tags="constant");
```

This search will return 10 nodes, from "foo:0" to "foo:9". None of these are actually in bar's subtree – in fact, "bar" itself is not even a real node (remember, it's only an unqualified node stub from which we derive "bar:0" through "bar:9".) A search on "bar", nonetheless, will include all subtrees rooted at all nodes belonging to bar's *family*. The *family* of a node stub consists of all the actual, initialized node stubs that are derived from it via the use of qualifiers.[1] This

[1] To be more precise, a family is recognized merely by name prefix. For example, both `ns.foo(1,2)` and `ns['foo:1:5']` are considered to be in the "foo:1" family, even though neither is directly derived from `ns.foo(1)`. Note also that `ns.foobar` is not considered to be in the "foo" family because "foo" must appear as a complete unqualified name, not as part of a name.

is a very useful feature, since it is a common practice to pass around a whole series of subtrees by referring to an unqualified root node.

If, for whatever reason, you want to limit your search to just the one subtree rooted at a the node, without involving the rest of its family, you can pass the `no_family=True` keyword to `search()`.

As a final note, the `family()` method of a node stub will return a list of node stubs comprising that node's family.

5.4 More complicated searches

The node scope `Search()` method, as well as the node stub `search()`, can also be used to find nodes with particular names and/or of a particular class. For example, the following will return a list of all nodes of class "MeqConstant" or "MeqParm", whose names begin with "foo":

```
ns.Search(name='foo.*',class_name='MeqConstant|MeqParm');
```

As you can see, regexes are supported here as well. When multiple search criteria (from among `name`, `class_name` and `tags`) are specified, the result is a logical AND. For example, this finds all MeqParms tagged as "solvable":

```
ns.Search(class_name='MeqParm',tags='solvable');
```

As a final wrinkle, you may choose to get a list of node names (rather than a list of node stubs) by passing in the `return_names=True` keyword:

```
names = ns.Search(tags='solvable',return_names=True);
```

Of course, the same result can be achieved via a simple list comprehension:

```
names = [ node.name for node in ns.Search(tags='solvable') ];
```

Chapter 6

TDLOptions: Get Your Free GUI Here

Any even remotely useful TDL script must have some degree of flexibility. Most programmers intuitively abhor hard-coded constants, preferring to place them in a separate header, or at least at the top of the source file. The code itself is then [hopefully] flexible enough that one can later change the constants and have everything else "just work". We use the term "constants" in a rather broad sense here, since beside actual numeric constants, these can also manifest themselves as:

- Input and output files and/or Measurement Sets
- Options that influence repeated tree structures (number of stations, baselines, sources, etc.)
- Options that determine what particular operation a forest performs (in other words, that effect conditional compilation of parts of a forest.)
- etc. etc.

The term "options" is therefore preferred to "constants".

The *TDLOptions* mechanism provides a uniform and user-friendly way to make your script expose various "knobs" and "switches". The example scripts that we have looked at perviously have already made use of TDLOptions. For example, a statement like

```
TDLCompileOption('num_ant','Number of antennas',[2,3,4]);
```

at the global level of your script creates a global Python variable named '`num_ant`' which can assume one of the specified values (2, 3 or 4). When loading the script, the user sees a little GUI that allows him to set the "Number of antennas" option to one of the allowed values. Your script is then expected to adapt itself to whichever value was specified.

All TDLOption settings are automatically saved to a file called "`.tdl.conf`", which means that the next time a user loads your script, the previous settings are restored. In addition, every time the user executes a TDL job, a summary of current settings is logged to a file named "`meqtree.log`". When using complicated scripts with many options, this log file provides a convenient "processing history". Also, your script may be invoked in "headless" (noninteractive) mode, e.g. as part of a pipeline, in which case its options may be set up by some other means entirely.

TDLOptions are simple to declare, but they provide all sorts of sophisticated functionality that allows one to build up highly interactive and context-sensitive GUIs. This chapter discusses the subject in detail.

6.1 Run-time vs. compile-time options

to be continued...

Chapter 7

Meow: a Case Study

Meow (Measurement Equation Object frameWork) makes extensive use of all the Python and TDL techniques discussed in the previous chapter. In this chapter, we're going to take an in-depth look at some example scripts that employ Meow, to see how these methods can fit together into a coherent whole.

7.1 Jones series

A Jones term is a 2×2 matrix[1] that describes a signal propagation effect. The word *term* is used because a Jones matrix usually appears as a term of a *measurement equation*. When putting together a measurement equation, we usually have a separate Jones term per antenna, or, if the effect also happens to be direction-dependent, a separate term per antenna and per direction on the sky (which usually corresponds to what we call a "source", since we model the sky by a collection of discrete sources.) A set of Jones terms describing the same effect per antenna and/or per source is called a *Jones series*.

The TDL qualifier mechanism is a godsend when creating and manupulating Jones series. As a working example, let's look at the `ME2/example6-iono.py` script. The measurement equation used here needs a series of Jones terms (the "zeta-Jones", from the ionospheric phase delay ζ), Z_{kp}, one per each source k and antenna p. The trees for these are created by the following call:

```
Zj = iono_model.compute_zeta_jones(ns,sources);
```

The `compute_zeta_jones()` takes a node scope and a list of Meow source objects, and creates a series of Z nodes. These will be created as

```
ns.Z(source_name,p)
```

[1] A Jones matrix can also be represented by a single scalar g, which is mathematically equivalent to the diagonal matrix $G = \begin{pmatrix} g & 0 \\ 0 & g \end{pmatrix}$. Matrix operations in MeqTrees work the same whether the g or G form is used, so we will not dwell any further on the distinction.

where `source_name` is obtained from each source object, and `p` is an antenna index. Ultimately, the nodes are created inside this function from `ME2/iono_geometry.py`:

```
def compute_zeta_jones_from_tecs (ns,tecs,source_list):
  """Creates the Z Jones for ionospheric phase,
  given TECs (per source,   per station).""";
  zeta = ns.Z;
  for src in source_list:
    for p in Context.array.stations():
      zeta(src.name,p) << Meq.Polar(1,-25*Lightspeed*tecs(src.name,p)/Meq.Freq());
  return zeta;
```

If you've followed this chapter until now, it should be pretty clear what's going on here. First we make a node stub, `ns.Z`, and assign it to the variable `zeta`. Then we loop over sources and stations, and define a `zeta(src.name,p)` node for each. Since `zeta` is just a reference to `ns.Z`, this is the same as simply creating `ns.Z(src.name,p)`. Finally, we return the `ns.Z` node stub.

The caller of this function doesn't know how the Z-Jones nodes are named or how they are created. All the caller needs to know is the contract on the return value: "something that can be qualified with a source name k and an antenna index p to obtain the node stub for Z_{kp}".

Now, how do we use the Z terms? Looking at `ME2/example6-iono.py`:

```
  for src in sources:
    allsky.add(src.corrupt(Zj(src.name)));
```

To create a "corrupted" source, we need to apply the Jones terms to a source. We know that a Meow source object implements a `corrupt(J)` method that takes a Jones series as a parameter. The contract on the `J` parameter is "something that can be qualified with an antenna index p to obtain a node stub for the Jones term corresponding to antenna p".[2] Well, this is almost a perfect fit with the previous contract — we know that `Zj` takes two qualifiers, so `Zj(src.name)` produces an underqualified node stub that is exactly the thing we want to pass to `corrupt()`. Inside `corrupt()` there will be a loop over stations, which will end up accessing the individual Jones terms as `J(p)`, which is the same thing as `Zj(src.name)(p)`, which is the same thing as `Zj(src.name,p)`, which is simply `ns.Z(src.name,p)`.[3]

[2] Unless corrupt is called as `corrupt(J,per_station=False)`, in which case it knows that `J` is not a per-antenna series, but a single Jones term.

[3] The sharp-eyed reader will notice that sometimes the example scripts use `src.direction.name` rather than `src.name`. For most intents and purposes, this is the same thing. Every source has a `direction` attribute that has a `name`, which is almost always identical to the source name, since most scripts have a one-to-one correspondence between sources and directions. In more exotic scenarios we can have multiple sources in the same direction, in which case you really want to have a per-direction rather than a per-source Jones term. Even though none of the examples actually realize this scenario, the `src.direction.name` thing has crept in. I apologize for any confusion.

7.1.1 Mea culpa

The `compute_zeta_jones()` function as shown here (and in the demo scripts) has one somewhat glaring design defect. We name the `ns.Z` node inside the function. What if we also use a different piece of code that wants to make a "Z" node? All sorts of confusion ensues.

The more robust appproach to node naming is illustrated by the function in section 3.4:

```
def define_tree_to_do_foo(node,a,b):
  node << a + b;
  return node;
```

Here, we name the node outside the function, and pass in a node stub. This a much better approach than allowing functions to pick their own names, since it gives the caller an opportunity to avoid name clashes. We want to **hide** the business of node naming from functions as much as possible.

In these particular circumstances this is only a minor sin — the letter Z is (hopefully) a well-established term for ionopsheric phase, so it's unlikely that someone else will have another Z of their own. Even then, this sort of thing is to be avoided in a proper design!

7.2 Using contract adapters

The `Zj(src.name)` operation is a simple example of something I call a *contract adapter*. We have one object (`Zj`) which implements an "I can be qualified with a name and a station index to make a Jones term" contract, and another object (the `J` argument to `corrupt()`) that implements an "I can be qualified with a station index to make a Jones term" contract. The "`(src.name)`" operation adapts one contract to match the other. Happily for us, it really is that simple, which is hardly a coincedence: Meow's contracts are designed around TDL's qualification mechanism.

A more somewhat more complicated case arises if we decide to change our ionospheric model (i.e. the innards of `compute_zeta_jones()`) to eliminate directional dependence. The function would then presumably look something like this:

```
def compute_zeta_jones_from_tecs (ns,tecs,source_list):
  """Creates the Z Jones for ionospheric phase,
  given TECs (per source,   per station).""";
  zeta = ns.Z;
  for p in Context.array.stations():
    zeta(p) << Meq.Polar(1,-25*Lightspeed*tecs(p)/Meq.Freq());
  return zeta; # really??
```

The problem here is that we've gone and changed the contract on the function's return value, thus breaking all the scripts that use our model.

There are a few ways around this difficulty. The naive way is to add some redundant per-source terms:

```
for src in source_list:
  for p in Context.array.stations():
    zeta(src.name,p) << Meq.Polar(1,-25*Lightspeed*tecs(p)/Meq.Freq());
return zeta;
```

Here we've created a bunch of subtrees that all repeat the same calculation over and over. This is rather wasteful, especially with a lot of source involved. The second approach is to use `MeqIdentity` nodes:

```
for p in Context.array.stations():
  zeta(p) << Meq.Polar(1,-25*Lightspeed*tecs(p)/Meq.Freq());
  for src in source_list:
    zeta(src.name,p) << Meq.Identity(zeta(p));
```

This works rather better — the tree will have a number of redundant `MeqIdentity` nodes, but since they're effectively a no-op, they will not slow it down in any noticeable way.

Finally, there's the purely Pythonic "contract adapter" approach. We can define a little function that behaves like a node stub. There's a wrong way and a right way to do this. The wrong way goes something like this:

```
for p in Context.array.stations():
  zeta(p) << Meq.Polar(1,-25*Lightspeed*tecs(p)/Meq.Freq());
return lambda x:zeta;
```

The "`lambda x:zeta`" statement creates a function with one argument that returns `zeta` (and ignores the argument). This only half-works with the `ME2/example6-iono.py` script:

- the return value — our lambda function — gets assigned to `Zj.`
- when `Zj` is invoked as `Zj(src.name)`, the lambda function is called, and returns `ns.Z`. This is something that can be qualified with an antenna index inside `corrupt()`, so we're doing good so far.
- but when `Zj` is invoked as `Zj(src.name,p)` a little bit later in the script, the whole thing breaks down since the lambda function expects a single argument.

Our lambda function behaves just like a node stub when invoked as `Zj(src.name)(p)`, but not when invoked as `Zj(src.name,p)`. To get things right in both cases, we need a somewhat more sophisticated adapter function:

```
for p in Context.array.stations():
  zeta(p) << Meq.Polar(1,-25*Lightspeed*tecs(p)/Meq.Freq());
def adapter (name,p=None):
  z = zeta;
  if p is not None:
    z = z(p);
  return z;
return adapter;
```

This shows how you can define and return a "local" function on-the-fly. The return value of `compute_zeta_jones()` is the `adapter` function.[4] When called as `adapter(src.name)`, it returns `zeta`. When called as `adapter(src.name,p)`, it returns `zeta(p)`. This is exactly the behaviour we want.

There's an even more elegant to implement this adapter function. Note that the operation we're trying to perform here can be reformulated in a general way as "ignore the first qualifier, apply the rest". Here's a suitably general implementation:

```
def adapter (first_qual,*other_quals):
  return zeta(*other_quals);
```

In this particular scenario, it is really only a matter of taste whether you go for the `MeqIdentity` nodes approach, or the (more Pythonic) contract adapter approach. In my opinion, the latter is more elegant, and is an essential technique to master, as it really comes into its own once you get into more complicated territory. To summarize:

- Python encourages you to consider objects in terms of their behaviour rather than their type. Using this wisely leads to more flexible interfaces and cleaner code.
- The essence of design by contract is to think in terms of behaviour, not type. For example, rather than thinking of the return value of a function as an unqualified node, think of it as "a thingy that can be qualified by a source name and a station index to derive a node".
- Python makes it easy to write little adapters that make one object behave like another.

7.3 Meow and node tags

**** to be continued ***

[4]Note that the name `adapter` is local to `compute_zeta_jones()` and is not seen anywhere outside that function. Other functions can define their own local things called `adapter` without any confusion.