MDF Documentation

Release 1.0

Tony Roberts

July 15, 2015

Contents

MDF is a Python package that aims to make writing code simpler, particularly for code that is intended to be run multiple times with varying inputs.

CHAPTER 1

Introduction

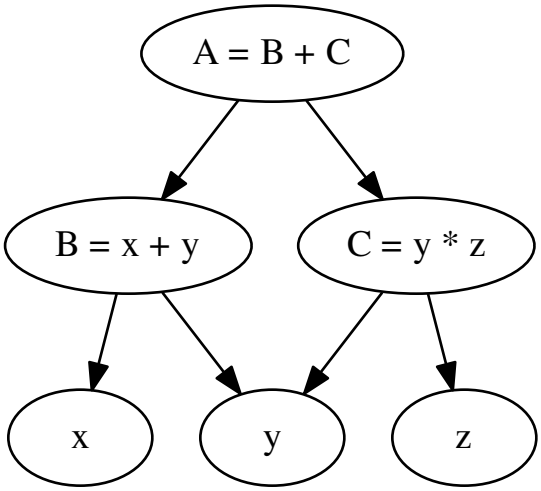
Most programmers are familiar with code written as functions that take arguments that subsequently call more functions that take more arguments. In order to call a function you need to know what arguments to provide, and in addition that function needs to know what arguments to provide to functions it calls and possibly add them to its own argument list to be passed in.

MDF turns this around and provides a way of expressing code as a *directed acylical graph*, or *DAG* for short. Each node in the graph depends on other nodes, which ultimately may depend on a set of terminal nodes that represent the input values to the graph or sub-graph.

This is best explained via a brief example. Consider this set of simple Python functions:

```
def A(x, y, z):
    return B(x, y) + C(y, z)

def B(x, y):
    return x + y

def C(y, z):
    return y * z
```

This can be expressed as the following graph:



In the first case evaluating `A(1, 2, 3)` is simply a case of calling the function. In the second case, we evaluate the node `A` under the condition `x=1`, `y=2` and `z=3`.

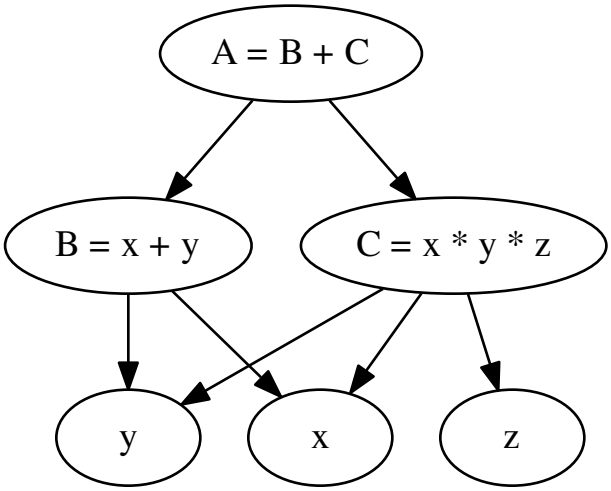
Consider that now the definiation of `C` is not quite right, it's decided it should actually be:

```
def C(x, y, z):
    return x * y * z
```

Because of this change to `C`, now `A` has to changed. If it has required another argument then `A` would also have to have its arguments changed and and functions calling `A` would have to be updated:

```
def A(x, y, z):
    return B(x, y) + C(x, y, z)

def B(x, y):
    return x + y

def C(x, y, z):
    return x * y * z
```

In the graph based approach node `C` can be changed without impact on `A`, and provided all dependents of `C` exist in the graph there is no change required to the calling code.



Now consider that you want to evaluate `A` for a set of `z`. Using the traditional approach you could call `A(x, y, z)` for each `z` in the required set. Now assume that you also want to collect the values of `C` for that set of `z` as well as `A`. At this point you have a few choices. You might call `A` and `C` for each `z`, you might refactor `A` to take `C` as an argument so you only have to compute `C` once for each `z`, or you might refactor `A` to take a list as an argument and accumulate the results of C in that.

Let's assume you end up with something like this:

```
def A(x, y, c):
    return B(x, y) + c

def B(x, y):
    return x + y

def C(x, y, z):
    return x * y * z

x = 1
y = 2
for z in range(100):
    c = C(x, y, z)
    a = A(a, y, c)
    print a, c
```

Now you notice that computing `B` every time is expensive and unnecessary. You refactor out `B` to improve performance:

```
def A(b, c):
    return b + c

def B(x, y):
    return x + y

def C(x, y, z):
    return x * y * z
```

```
# set the static variables x and y
x = 1
y = 2

# pre-compute b as it doesn't vary with z
b = B(x, y)

# compute A and C for each z
for z in range(100):
    c = C(x, y, z)
    a = A(b, c)
    print a, c
```

This is a contrived example, but even so it's starting to feel untidy and every other reference of `A` and `B` needs to be refactored.

In contrast doing the same evaluation using *mdf* is quite straightforward. Below is some example code using *mdf* that does the same as the code above. Don't worry that some terms referenced in this code have not been mentioned yet, this is just to give an idea of how this code can be written:

```
from mdf import MDFContext, varnode, evalnode,

# varnode creates nodes that have values in a context
x = varnode()
y = varnode()
z = varnode()

# @evalnode declares that these functions are nodes in our DAG
@evalnode
def A():
    return B() + C()

@evalnode
def B():
    return x() + y()

@evalnode
def C():
    return x() + y() + z()

# contexts are covered later in these docs, but essentially it's where the
# values for the nodes are kept
ctx = MDFContext()

# set the values for x and y in the context
ctx[x] = 1
ctx[y] = 2

# compute A and C for each z_i
for z_i in range(100):
    ctx[z] = z_i

    # getting the values from the context evaluates them and returns the results
    print ctx[A], ctx[C]
```

Nodes are evaluated lazily and only re-computed when their dependecies have been updated. This means that in the example above `B` is only calculated once as `x` and `y` aren't changed.

If `B` was modified so it was dependent on `z`, or any other changes for that matter, the code above wouldn't have to

be changed outside of the definition of the actual node being changed. In the more traditional version changes in a function may require corresponding changes to the calling code which is not always immediately obvious.

CHAPTER 2

Nodes

The nodes that form the DAG are declared as normal python functions, decorated with one of the node decorators such as *evalnode()*.

Nodes are callable objects that take no arguments. Calling them either invokes the node function or returns the previous cached result if no dependencies of the node have changed.

Nodes may only be called from other node functions. Calling a node outside of a node function will result in an error. Evaluating a node outside of another node function must be done via a context object.

Dependencies between nodes are discovered at run-time as the nodes are evaluated. The context keeps track of what node is currently being evaluated and as that node references other nodes it adds the edges to the DAG. If a node is conditionally evaluated from another node function, that dependency is only discovered once that condition is met and the branch evaluating the other node is executed.

Nodes are evaluated from other within other node functions (or any function called by a node function) by calling them:

```
from mdf import evalnode

@evalnode
def node_function():
    """
    this function is actually a node because of the
    use of the @evalnode decorator
    """
    # to evaluate other nodes they just need to be called
    value = another_node()

    # do some calculation
    result = ...
    return result

@evalnode
def another_node():
    # do some calculation possibly involving other nodes
    return result
```

2.1 Time Dependent Nodes

Nodes are marked as requiring re-calculation whenever any of their dependencies are modified. They are later lazily evaluated as required.

There's a builtin node *now()* that behaves in a more specialized way and allows node valuations to evolve over time.

When the *now()* node is advanced, which can be done manually via *MDFContext.set_date()*, all the nodes dependent on time are marked as requiring re-calculation but additionally they are marked that the reason they require re-calculation is because time has moved forwards.

evalnode() nodes can be generators instead of regular functions (generators *yield* values rather than *return* a single value). When a generator is used *mdf* will advance the generator of the node each time the *now()* node is advanced. This allows state to be maintained between valuations:

```
from mdf import MDFContext, evalnode, now
from datetime import datetime, timedelta

@evalnode
def time_dependent_node():
    """
    a simple node whose value is dependent on 'now'
    """
    # returns 0, 1, 2, ... for the current weekday
    return now().weekday()

@evalnode
def incrementally_updated_node():
    """
    the value of this node is the sum of another node
    """
    # the first value will simply be time_dependent_node
    todays_value = time_dependent_node()
    yield todays_value

    # when the date is advanced this generator is continued
    # until the next yield

    while True:
        # yield today's value + the value evaluated previously
        prev_value = todays_value
        todays_value = time_dependent_node()
        yield todays_value + prev_value

# create the context with an initial date
date = datetime(2011, 9, 2)
ctx = MDFContext(date)

# get the value of incrementally_updated_node
x = ctx[incrementally_updated_node]
# x is now 4 (Friday)

# advance the date one day
date += timedelta(days=1)

# set the date on the context to be invoked (this causes the
# incrementally_updated_node generator to be advanced)
ctx.set_date(date)

# get the value of incrementally_updated_node
x = ctx[incrementally_updated_node]
# x is now 9 : 4 (Friday) + 5 (Saturday) = 9
```

This is a simple example, but the same methods can be used to build more complex nodes that perform incrementally

calculated time-dependent nodes.

If time is ever moved backwards by calling *MDFContext.set_date()* then the current state of the time dependent nodes is discarded and the initial state will be re-evaluated by restarting the generators.

2.1.1 Filtering

For nodes that update incrementally with time sometimes it's useful to be able to specify whether the update should be called or not for a particular date rather than have to check inside the update function.

For example, some values might only need updating on valid business days but the context might be stepped through all calendar dates for a date range:

```
from mdf import evalnode

def my_node_filter():
    # the filtered evalnode will only be advanced on business days
    if my_is_valid_business_day_function():
        return True
    return False

@evalnode(filter=my_node_filter)
def my_node():
    yield some_initial_value
    while True:
        do_some_update_calculation(...)
        yield updated_value
```

The filter could be a node instead of a function. This is convenient if you need to apply the same filter to multiple nodes as it won't be re-calculated more than necessary.

To make it easy to get a filter relating to a specific series of data there's a function *filternode()* to create a node that returns `True` when the current date is in the index of that data, or `False` otherwise. This makes it simpler to perform calculations at the frequency of the underlying data.

2.2 Queue Nodes

Queue nodes are a specialized time-dependent node. The value of the node is a double ended queue (see `collections.deque`) of values. A double ended queue is used as it supports efficient appending and popping to both sides of the queue. Queues can also be used to construct numpy arrays and regular python lists.

The node function is called each time the node *now()* is advanced and the result is appended to the queue. The value of the node is the queue itself, which should be regarded as immutable.

Below is an example that uses a queue to get a delayed value:

```
from mdf import evalnode, queuenode

@queuenode
def some_value_queue():
    # do some calcuations
    return result

@evalnode
def delayed_value():
    values = some_value_queue() # type is collections.deque
    if len(values) < 5:
```

```
        return np.nan

    # return the value calculated 4 timesteps ago
    # (the item at -1 is the value for now)
    return values[-5]
```

Queue nodes can be bounded so they don't grow indefinetely. This is done by setting the size of the queue. Once the queue reaches that size older items will be popped off the queue. The size can be specified as either an integer value or as a callable object (e.g. function or node) which can be useful if the size if a function of another node. Once the queue is created the size is fixed for that context:

```
# keep at most 5 values
@queuenode(size=5)
def some_value_queue():
    # do some calcuations
    return result

#
# or calculate the size as a function (or node)
#

def get_queue_size():
    return 5

@queuenode(size=get_queue_size)
def some_other_value_queue():
    # do some calcuations
    return result
```

Because queue nodes are a specialization of the eval node, they may also be filtered in the same way. If a filter is applied only when the filter returns True will values be calculated and appended to the queue.

2.3 Other Node Types

While eval nodes can be used to calculate any type of value, commonly used valuation types can be packaged as other node types for convenience. Currently the list of these specialized nodes is quite small, but as more use-cases are presented it's reasonable to expect this list to grow.

Because these nodes are all specializations of the eval node, they may also be filtered in the same way. If a filter is applied only when the filter returns True will values be calculated or updated.

2.3.1 Delay Node

The delay node type is closely related to the queue node type. The *delaynode()* node type delays the value returned for a number of timesteps that can be specified as the `periods` parameter to that function:

```
from mdf import evalnode, delaynode

@delaynode(periods=10)
def a_delayed_value():
    return some_value

@evalnode
def some_other_value():
```

```
    x = a_delayed_value() # this is the valued returned by a_delayed_value as it
                          # was 10 timesteps ago
```

The value of the node before the number of periods has elapsed can be set using the `initial_value` parameter. The node's value will be this until enough timesteps have elapsed. By default the initial value is `None`.

The function decorated with *`delaynode()`* may be called when the node is evaluated if it hasn't already been called for the current timestep or if any of its dependencies have changed. This can be a problem if attempting to set up a recursive relationship such as:

```
@delaynode(periods=1, initial_value=0)
def delayed_a():
    return a()

@evalnode
def a():
    return 1 + delayed_a()
```

Even though the value for `delayed_a()` should be available before `a()` is evaluated this still results in an infinite recursion as evaluating `delayed_a()` will result in a recursive call to `a()`.

To solve this problem delaynodes may optionally be lazily evaluated by setting the `lazy` kwarg to True:

```
@delaynode(periods=1, initial_value=0, lazy=True)
def delayed_a():
    return a()
```

This is not the default because dependencies are discovered at run-time and so delaying evaluation of a node will result in dependencies being added in a later timestep that alter the structure of the DAG. When using shifted contexts this can be a problem. If `mdf` thinks that a node can use a parent context of a shifted context, and then later the dependencies change that break that assumption a `ConditionalDependencyError` will be thrown.

The way to fix a problem with conditional dependencies is to make them unconditional. In the case of delayed nodes this can be done by making the `initial_value` an `eval_node()` that has the same dependencies (or at least the ones that are sensitive to the shift) as the delayed node function.

2.3.2 NaN Sum Node

The *`nansumnode()`* node type calculates the sum of the values returned by its function as *`now()`* is advanced. Values that are NaN are excluded from the sum:

```
from mdf import evalnode, nansumnode

@nansumnode
def some_value():
    return some_value

@evalnode
def sum_of_some_value():
    value_sum = some_value() # this is the sum of 'some_value' for all time steps so far
```

2.3.3 Cumulative Product Node

The *`cumprodnode()`* node type calculates the cumulative product of the values returned by its function as *`now()`* is advanced:

```
from mdf import evalnode, cumprodnode

@cumprodnode
def some_value():
    return some_value

@evalnode
def sum_of_some_value():
    value_prod = some_value()  # this is the cumulative product of 'some_value'
                                                    # for all time steps so far
```

2.3.4 Apply Node

The *applynode()* node type applies an arbitrary function to the value returned by the node function. You can optionally supply additional args and kwargs that will be passed in to the function; if any of these arguments are nodes then they will be evaluated and the result will be passed in.

For example, to add the values of existing nodes A and B:

```
A_plus_B = A.apply(operator.add, args=(B,))
```

Or you can get the node:

```
A_plus_B_node = A.applynode(operator.add, args=(B,))
```

And then chain apply additional nodes to it, such as a cumulative product:

```
smoothed_A_plus_B = A_plus_B_node.cumprod(...)
```

NB: Unlike most other node types the applynode shouldn't be used as a decorator, but instead should only be used via the method syntax for node types (see nodetype_method_syntax, below).

2.4 Method Syntax For Node Types

Creating a new node for simple operations on an existing node can make code look bloated and difficult to follow.

For this reason every node type is also exposed as methods on all other nodes. This is a syntactic helper and the end result is exactly the same as if a new node using the node type decorator was used.

This is best illustrated by example:

```
from mdf import evalnode, cumprod
from random import random

@evalnode
def random_value():
    while True:
        yield random()
```

If we wanted to compute the cumulative product of this random value you could do it by creating a new node using the `cumprod()` decorator:

```
@cumprodnode(half_life=10)
def cumulative_product_of_random_value():
    return random_value()
```

But if there are many nodes this can become a bit awkward. Using the method syntax the same thing can be achieved as follows:

```
@evalnode
def some_other_node():
    ewam_of_random_value = random_value.cumprod(half_life=10)

    # do some more calculation
    return result
```

When the *cumprod* method on the *random_value* node is called an internal node is created for that cumulative product calculation. Each subsequent time it's called that internal node is re-used and so the effect is exactly the same as if the cumprod node was created explicitly.

All of the standard node types have corresponding methods, and custom node types can optionally expose themselves as methods.

In addition, there is also a method that returns the internal implicitly created node. This allows for chaining, e.g.:

```
@evalnode
def some_other_node():
    ewam_of_random_value_node = random_value.cumprodnode(half_life=10)
    delayed_cumprod = ewam_of_random_value_node.delay(periods=10, initial_value=0)

    # do some more calculation
    return result
```

Or more simply:

```
@evalnode
def some_other_node():
    delayed_cumprod = random_value.cumprodnode(half_life=10).delay(periods=10, initial_value=0)

    # do some more calculation
    return result
```

CHAPTER 3

Class Nodes

Node definitions may also be applied to python classes. An *evalnode()* when declared on a class applies to either a `classmethod` or a `staticmethod`.

Nodes may not apply to instance methods as state should be maintained in the DAG and not in class instances, and so nodes only make sense in the context of class methods and static methods.

```
from mdf import MDFContext, evalnode, varnode

class MyClass(object):

    # varnodes may be declared as class attributes
    a = varnode(default=10)
    b = varnode(default=20)

    # evalnodes may be declared using class methods
    @evalnode
    def function_of_a_and_b(cls):
        # as this is a classmethod it has access to other nodes on
        # the class, the same way as it can access any other class
        # attributes or methods.
        return cls.a() * cls.b()

    @evalnode
    def example_node(cls):
        return cls.function_of_a_and_b() * 5
```

Class nodes are referenced and used in exactly the same way as non-class nodes:

```
ctx = MDFContext()

# class nodes are attributes of their class and are referenced in the normal way
ctx[MyClass.a] = 5
ctx[MyClass.b] = 10

print clx[MyClass.example_node]
```

Class nodes behave just like normal python class methods with respect to inheritance. Class nodes may be overridden or re-used by subclasses in the same way as class methods:

```
class MyDerivedClass(MyClass):

    @evalnode
    def function_of_a_and_b(cls):
        return cls.a() + b()
```

```
print ctx[MyDerivedClass.example_node]
```

In the example above, `MyDerivedClass.example_node()` is a node inherited from the base class `MyClass` which calls `cls.function_of_a_and_b()`. As that was overridden in the derived class that overridden implementation will be used when evaluating `MyDerivedClass.example_node()`.

Class *evalnode()* nodes (and derived node types) are bound to their owning classes. This means that when a node is referenced by one class it is a distinct node from when the same node definition is referenced from a derived class. In the example above both `MyClass` and `MyDerivedClass` have `example_node()` nodes. Even though it is only declared on `MyClass` and inherited by `MyDerivedClass` `MyClass.example_node()` and `MyDerivedClass.example_node()` are different nodes in the DAG.

varnode() nodes are not bound in the same way as they are just class attributes and so `MyClass.a()` and `MyDerivedClass.a()` actually refer to exactly the same object, and so they are the same nodes in the DAG. It is possible to override class attributes in Python though, and so *varnode()* nodes may be overriden in the same way.

CHAPTER 4

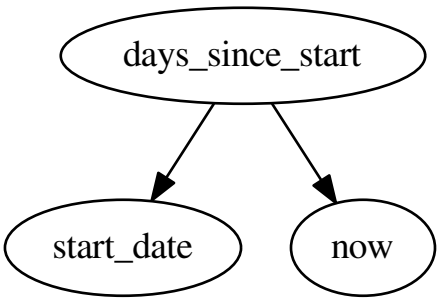
Contexts

As seen in the *Introduction* *mdf* deals with nodes which are structured into a directed acyclical graph (DAG) which are then lazily evaluated.

The structure of the code is what defines the DAG, but on its own the DAG has no values, only the capability of computing values. The context (`MDFContext`) is what contains the values at the nodes in the DAG and so a node can only be said to have a value *in a context*.

There is one special node that is always present and is set when the context is constructed, `now()`. This node represents the current time for all computations in the context, and all values that are time-dependent reference this node.

Multiple contexts can be constructed, and the same nodes can be evaluated in these different contexts to get diffent results with different starting values. Here's a quick example




```
from mdf import MDFContext, evalnode, now
from datetime import datetime

start_date = varnode()

@evalnode
def days_since_start():
    delta = now() - start_date()
    return delta.days

# create two contexts
```

```
ctx1 = MDFContext()
ctx2 = MDFContext()

# set the date on each context (this is used by the 'now' node)
d = datetime(2011, 8, 9)
ctx1.set_date(d)
ctx2.set_date(d)

# set different start dates on the contexts
ctx1[start_date] = datetime(1970, 1, 1)
ctx2[start_date] = datetime(2000, 1, 1)

# the node days_since_start has a different value in ctx1 and ctx2
days1 = ctx1[days_since_start] # = 15195
days2 = ctx1[days_since_start] # = 4238
```

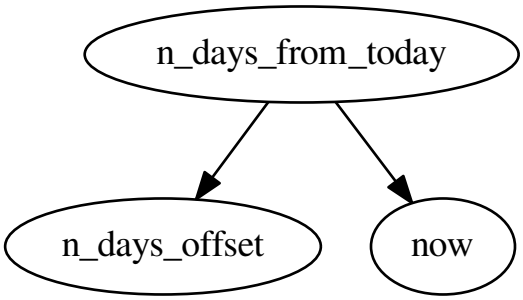
4.1 Setting Values

Values are set for nodes in a context using the *MDFContext.set_value()* method or more commonly by indexing into the context with the node:

```
A = varnode()
ctx = MDFContext()

# set the value of A in ctx
ctx[A] = 100

# get the value A in ctx
a = ctx[A] # a == 100
```

Any node type can have a value set to it. Once a value is set for a node in a context it is fixed until it's changed again. Even if a node has dependencies (e.g. an *evalnode()*) and its dependencies are changed the node won't be re-calculated if a value has explicitly been set.

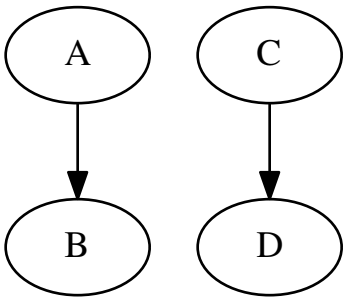
Once a context has been created and any initial values have been set any available nodes can be evaluated. Node values can also be set in the context and when any dependent nodes are re-evaluated they will reflect those changes



```
from mdf import MDFContext, evalnode, varnode, now
from datetime import datetime, timedelta

n_days_offset = varnode("n_days_offset")

@evalnode
def n_days_from_today():
    offset = n_days_offset()
    return now() + timedelta(days=offset)

# create the context and set the date
ctx = MDFContext()
ctx.set_date(datetime(2011, 8, 9))

# set the value of n_days_offset
ctx[n_days_offset] = 1

print ctx[n_days_from_today] # datetime.datetime(2011, 8, 10, 0, 0)

# update n_days_offset
ctx[n_days_offset] = 100

print ctx[n_days_from_today] # datetime.datetime(2011, 11, 17, 0, 0)
```

4.2 Overriding Nodes

As well as being able to set a node's value in a context it is also possible to override a node itself in a context. For example, suppose you had the following graph

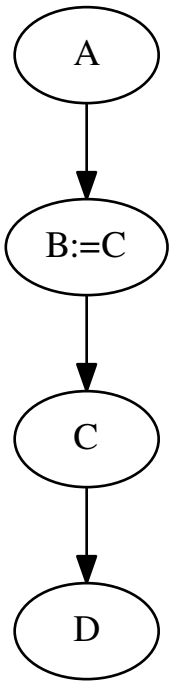


It's possible to override node `B` with node `C` using the *`MDFContext.set_override()`* method, or by setting the value on the context for one node with another node:

```
from mdf import MDFContext, evalnode, varnode

B = varnode(default=10)
D = varnode(default=20)

@evalnode
```

```
def A():
   return B() * 5

@evalnode
def C():
   return D() * 10

ctx = MDFContext()

a = ctx[A] # a == 50

# override B with C
ctx[B] = C

a = ctx[A] # a == (B overriden with C) * 5
           #   == (D * 10) * 5 == 1000

# A now depends on C which depends on D
# so changing D changes A
ctx[D] = 2

a = ctx[A] # a == 100
```

The resulting graph (where := denotes overriden with) looks like this:



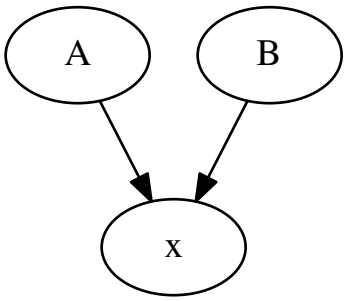
Any type of node can be overriden by any other type of node. This means that you can override entire sub-graphs or add new sub-graphs where previously there was just a single *varnode()*.

Overriding nodes can be useful for unit testing. If you have a node that you don't want to evaluate as part of a unit test it can be overriden with a mock node.

4.3 Shifted Contexts

As mentioned above, a node only has a value in a context. In some situations it's useful to be able to evaluate a node in a context given another node is set to some value *without* modifying the context.

Consider the following DAG



Suppose you want to evaluate `A` for all `x in [1, 2, 3, 4, 5]` but you don't want to actually affect any other values in the context. You could do that by *shifting* the context and evaluating `A` on the shifted context:

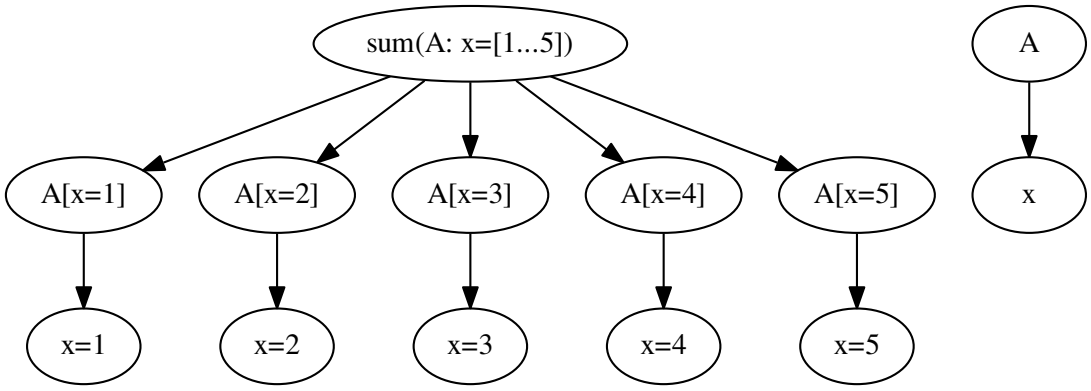
```
from mdf import MDFContext, evalnode, varnode
from datetime import datetime

x = varnode()

@evalnode
def A():
    return x() * 2

@evalnode
def B():
    return x() * 3

ctx = MDFContext()

# set some value for 'x'
ctx[x] = 100

print ctx[A] # 200
print ctx[B] # 300

# calculate A[x=1,2,3,4,5] without modifying the context
for i in [1, 2, 3, 4, 5]:
    shifted_ctx = ctx.shift({x : i})
    print shifted_ctx[A] # 2, 4, 6, 8, 10
```

```
print ctx[A] # nothing's changed, still 200
print ctx[B] # nothing's changed, still 300
```

Shifting a context creates a new context with the shifted value set to a new value, but the shifted context is linked to the original context.

All values not dependent on the shifted value are still shared between the contexts. If you change one in one context it changes in all related contexts. This also means that the cached calculated values are also shared and so shifting a context can be more efficient than simply cloning it.

mdf provides a function `shift()` for use within a node function. It returns the values of a node in multiple shifted contexts and can be used to create new sub-graphs in the DAG, for example:



This relatively complicated looking DAG can be written by shifting `A` by `x`:

```
from mdf import varnode, evalnode, shift

x = varnode()

@evalnode
def A():
    return x() * 2

@evalnode
def sum_of_A():
    all_As = shift(A, x, [1,2,3,4,5])
    return sum(all_As)
```

This allows for code that behaves like sub-routines and loops but retains the DAG struture.

Shifting works with overriding nodes also. If the shift value is actually a node instance then the shifted node will be overriden by that node in the shifted context.

CHAPTER 5

Back-Testing and Scenario Analysis

5.1 Back-Testing

In the context of mdf, a back test is nothing more than evaluating a node or collection of nodes for a range of dates. Its possible to do this with a simple for loop:

```
import mdf
import pandas as pa
from datetime import datetime

# non-existent module used just for illustration in this example
import example_nodes

# get the range of dates used for the back test
date_range = pa.DateRange(datetime(2000, 1, 1), datetime(2011, 9, 6))

# create the context and set some initial values
ctx = mdf.MDFContext()
ctx[example_nodes.an_example_var_node] = 5

# evaluate a node for every date in date_range
results = []
for date in date_range:
    ctx.set_date(date)
    value = ctx[example_nodes.an_example_evalnode]
    results.append(value)

# now we can create a dataframe or plot the values etc.
```

To simplify this *mdf* provides several higher level functions that do this for you:

5.1.1 build_dataframe

```
# evaluate a node and return a dataframe of the results
# the column name is the node name
df = mdf.build_dataframe(date_range, example_nodes.an_example_evalnode)

# or supply a list of nodes
df = mdf.build_dataframe(date_range,
                                                    [example_nodes.an_example_evalnode,
```

```
                                    example_nodes.another_example_evalnode]
                                ctx=ctx)
```

See *build_dataframe()*

5.1.2 plot

```
# plot is the same as build_dataframe but it plots the results
mdf.plot(date_range,
         example_nodes.an_example_evalnode,
         ctx=ctx)
```

See *plot()*

5.1.3 to_csv

```
# to_csv is the same as build_dataframe but it writes the results to a csv file
fh = open("myfile.csv")
df = mdf.to_csv(fh,
                date_range,
                example_nodes.an_example_evalnode,
                ctx=ctx)
```

See `to_csv()`

5.1.4 get_final_values

```
# get_final_values steps through all the dates in the range but only returns
# the final result
value = mdf.get_final_values(date_range,
                                  example_nodes.an_example_evalnode,
                                  ctx=ctx)

values = mdf.get_final_values(date_range,
                                   [example_nodes.an_example_evalnode,
                                    example_nodes.another_example_evalnode],
                                   ctx=ctx)
```

See *get_final_values()*

5.1.5 run

run() is the most general of the back testing functions, and in fact is used by all the other functions.

Instead of producing a particular output format it simply advances the context's date through the given date range and calls the callables. The callables are responsible for evaluating any nodes and collecting the results.

Several callbable object classes are provided for use with *run()*:

```
# DataFrameBuilder can be used for collecting values into a dataframe
df_builder = mdf.DataFrameBuilder([example_nodes.an_example_evalnode,
                                   example_nodes.another_example_evalnode])

# CSVWriter can be used for writing values to a csv file
```

```
csv_builder = mdf.CSVWriter("myfile.csv",
                                [example_nodes.an_example_evalnode,
                                example_nodes.another_example_evalnode])

# or you can use custom functions as well
def my_func(ctx):
    print ctx.get_date(), ctx[example_nodes.an_example_evalnode]

# they're all run in one go using run
mdf.run(date_range, [df_builder, csv_builder, my_func], ctx=ctx)
    # you can then get the dataframe
    df = df_builder.get_dataframe(ctx)
```

See the API docs for *DataFrameBuilder*, *CSVWriter* and *run()* for more information.

5.2 Scenario Analysis

As calculations done using *mdf* are constructed as a DAG all the inputs are accessible, and this lends itself very convieniently to doing scenario analysis.

Using *mdf* it's possible to run a back-test for a date range with multiple sets of input parameters simultaneously. By using shifted contexts to achieve this anything not depedendent on the input parameters being varied intermediate calculations can be shared, potentially making the overall run-time less than if the code was run N times with the different input paramters.

Rather than creating all the shifted contexts and iterating through the date range each time you want to run a scenario the *run()* function may be used:

```
# calculate example_nodes.an_example_evalnode for a range of different
# values for example_nodes.an_example_var_node

# each scenario is specified as a 'shift' dictionary which is a dictionary
# of nodes to shifted values. In this case only one node is shifted but it
# could be multiple.

shifts = [
    {example_nodes.an_example_var_node : 1},
    {example_nodes.an_example_var_node : 2},
    {example_nodes.an_example_var_node : 3},
    {example_nodes.an_example_var_node : 4},
]

# create a dataframe builder to collect the results of the scenarios
df_builder = mdf.DataFrameBuilder(example_nodes.an_example_evalnode)

# run all the scenarios
mdf.run(date_range, [df_builder], shifts=shifts, ctx=ctx)

# df_builder.dataframes is now a list of dataframes, one for each shift set
```

5.2.1 scenario

For the cases where you want to calculate the final result of a node after iterating through a range of dates you can use the *scenario()* function.

scenario() takes two nodes to be varied and two lists of values for those nodes.

The value of the result node should be a scalar value and the result is returned as a 2d numpy array:

```
# artificial example just for illustration
a = varnode()
b = varnode()

@evalnode
def X():
    return a() + b()

# calculate the value of X for a in [1, 2, 3, 4] and b in [10, 20, 30, 40]
a_values = [1, 2, 3, 4]
b_values = [10, 20, 30, 40]

results = mdf.scenario(date_range,
                               X,
                               a, a_values,
                               b, b_values,
                               ctx=ctx)
```

5.2.2 plot_surface

plot_surface() works in the same way as *scenario()* except that the result is plotted as a 3d graph as well as returned as a numpy array:

```
# plots the results of each shift as a 3d surface
results = mdf.plot_surface(date_range,
                                  X,
                                  a, a_values,
                                  b, b_values,
                                  ctx=ctx)
```

5.2.3 heatmap

`heatmap()` works in the same way as *scenario()* except that the result is plotted as a heatmap as well as returned as a numpy array:

```
# plots the results of each shift as a heatmap
results = mdf.heatmap(date_range,
                              X,
                              a, a_values,
                              b, b_values,
                              ctx=ctx)
```

CHAPTER 6

Interactive use of MDF in IPython

MDF has a number of 'magic' functions that make it easier to use interactively in the IPython environment. IPython 'magic' functions are ones that are invoked with a `%` before the name and are not available outside of IPython as they intended only for interactive work.

To access the magic functions you first need to import everything from the *mdf.lab* module:

```
from mdf.lab import *
```

This will print some brief text telling you that to get more help you need to use the `%mdf_help` magic function:

```
%mdf_help
```

Once *mdf.lab* is imported an 'ambient' context is created, so you can evaluate nodes without specifying any particular context - as you would inside a node function. For example:

```
from random import random

# define a new node
@evalnode
def rand():
    while True:
        yield random()

# call it and it will be evaluated in the ambient context
rand() # returns a random number
```

Values of nodes in the current ambient context can be accessed in the normal way by calling the nodes, and can be set using the `value` property

```
In [1]: x = varnode(default=1)

In [2]: x()
Out[5]: 1

In[3]: x.value = 2

In [4]: x()
Out[4]: 2
```

As time-dependent nodes are important in mdf it is easy to set and advance the current date (the *mdf.now* node) in IPython using the magic functions `%mdf_now` and `%mdf_advance`

```
# get the current time set in the ambient context (this is just the same as calling 'now()')
In [5]: %mdf_now
```

```
Out[5]: datetime.datetime(2012, 5, 2, 0, 0)

# set the current date
In [6]: %mdf_now 2005-01-01
Out[6]: datetime.datetime(2005, 1, 1, 0, 0)

# advance the date
In [7]: %mdf_advance

In [8]: %mdf_now
Out[9]: datetime.datetime(2005, 1, 3, 0, 0)
```

Notice that the magic functions understand dates as literals so there's no need to construct datetime objects.

`%mdf_advance` optionally takes some nodes and returns the values of those nodes after the timestep. This can be useful for tracking values as you step through a few timesteps

```
In [9]: %mdf_advance rand
Out[9]: 0.67507466071023625

In [10]: %mdf_advance rand now
Out[10]: [0.67751015258066294, datetime.datetime(2005, 1, 5, 0, 0)]
```

6.1 Working with Timeseries

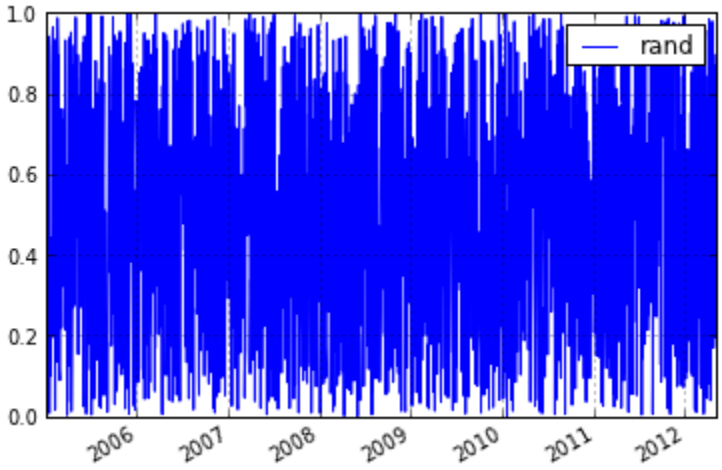
Nodes can be evaluated over a time range to produce time series of results, which can be plotted, stored as pandas DataFrames or exported to Excel.

The main functions used to construct timeseries of results are:

- `%mdf_df` for creating dataframes
- `%mdf_plot` for plotting using matplotlib
- `%mdf_xl` for exporting to Excel

All these functions take two dates (start and end) followed by a list of nodes (note the use of `T` as a shortcut for today)

```
In [11]: %mdf_plot 2005-01-01 T rand
```



Nodes can be defined interactively either by writing new functions or simply using the nodetype method syntax (nodetype_method_syntax) to build up series of operation quickly

```
In [12]: %mdf_plot 2005-01-01 T rand.cumprodnode() rand.nansumnode()
```

This could also be written as

```
In [13]: a = rand.cumprodnode()

In [14]: b = rand.nansumnode()

In [15]: %mdf_plot 2005-01-01 T a b
```

In addition to these functions there is also `%mdf_dfs` which returns a list of dataframes (one for each node) and `%mdf_wp` which returns a widepanel constructed from dataframes for each node. These can be useful when evaluating multiple nodes at the same time but when you don't want the results to get merged into a single dataframe.

6.2 Working with Data

Most often data is loaded as a pandas DataFrame, Series or WidePanel. To use those effectively in mdf we normally define a node that is the 'current' row or item from that dataset. As time advances that node updates to reveal the data from the underlying structure.

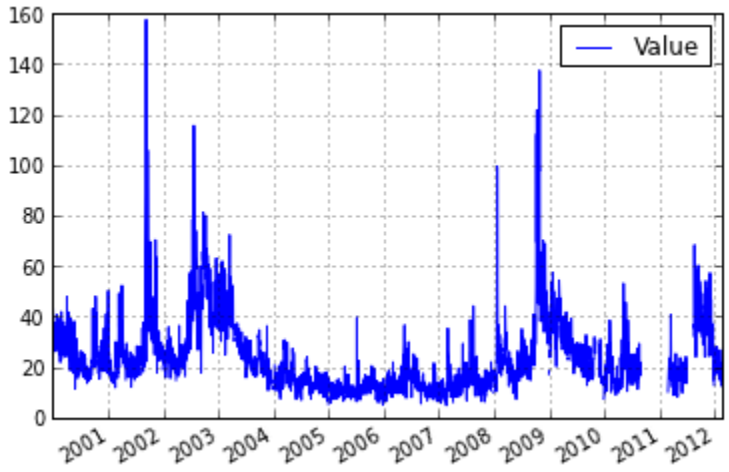
The *`datanode()`* function can be used to construct such a node from any DataFrame, WidePanel or series that is indexed by date.

The following example shows how to access data in a pandas DataFrame:

```
    from mdf import datanode
    import pandas as pa

# load some data
df = pa.DataFrame.from_csv("data_file.csv")

# create a node whose value is the row from the dataframe for 'now'
df_node = datanode("x", df)
```

This `df_node` node is like any other mdf node

```
In [18]: %mdf_plot 2000-01-01 T df_node
```



6.3 Applying Functions

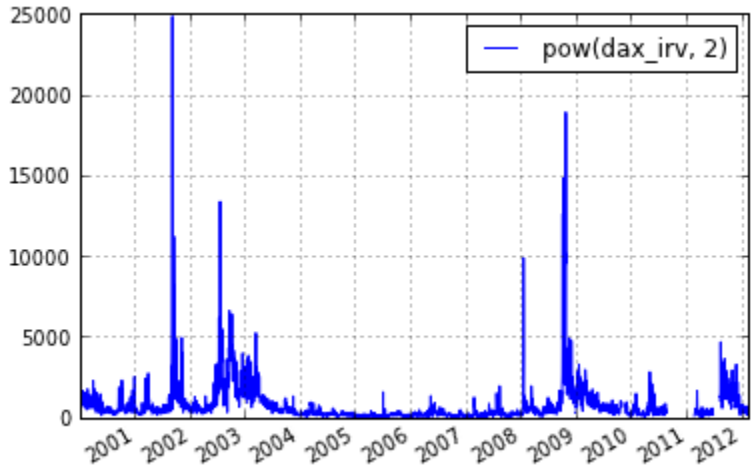
Usually when writing code using mdf new nodes are written whenever a value derived from other nodes is required

```
@evalnode
def df_node_sq():
    return math.pow(df_node(), 2)
```

For trivial functions such as the one above it can be inconvenient to have to write these for each desired node.

The *applynode()* function can be used to create new nodes that apply a function to other nodes. The above example can be re-written as follows:

```
In [19]: df_node_sq = applynode(math.pow, df_node, 2)

In [20]: %mdf_plot 2000-01-01 T df_node_sq
```



6.4 Accessing the Context and Shifting

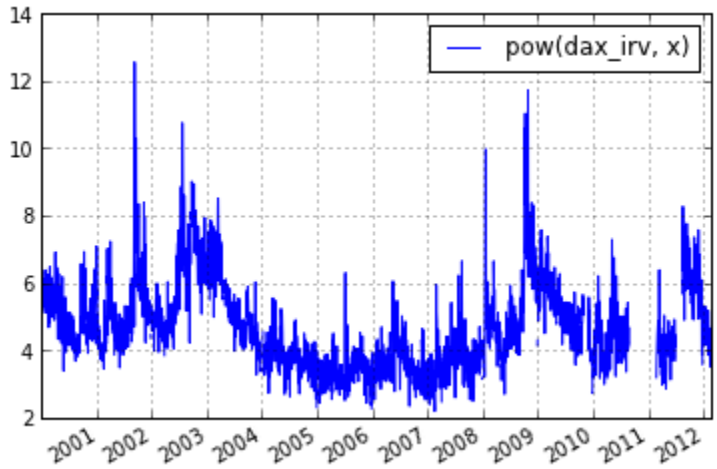
When you want to evaluate a node with another node set to a specific value or overriden you use a shifted context (see *Shifted Contexts*).

You can get and set the current ambient context using the `%mdf_ctx` magic function. This allows you to get the current context, create a shifted context and then set that shifted context as the current context.

```
In [21]: ctx = %mdf_ctx

In [22]: x = varnode(default=1)

In [23]: shifted_ctx = ctx.shift({x : 2})

In [24]: %mdf_ctx shifted_ctx
Out[24]: <ctx 1: 2012-05-03 [x=2] at 123373200>

In [25]: x()
Out[25]: 2
```

The functions `%mdf_df`, `%mdf_plot` and `%mdf_xl` also take an optional set of shifts. This makes it easy to get results from a shift without having the get, shift and set the current context.

```
In [26]: df_node_pow = applynode(math.pow, df_node, x)

In [27]: %mdf_plot 2000-01-01 T df_node_pow [x=0.5]
```

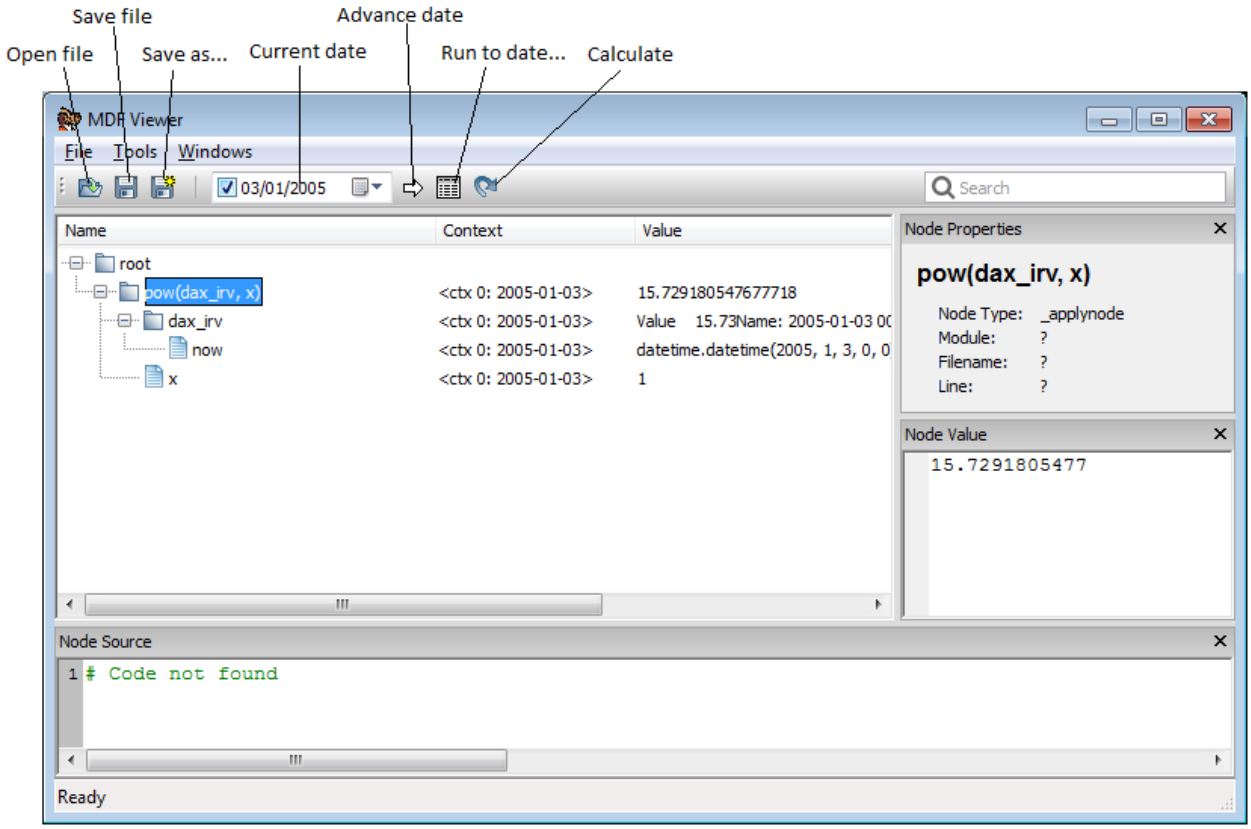


6.5 Using the MDF Viewer

The mdf viewer can be used explore the dependencies between nodes and plot or export values over time.

The mdf viewer can be opened from ipython with the magic command `%mdf_show`.

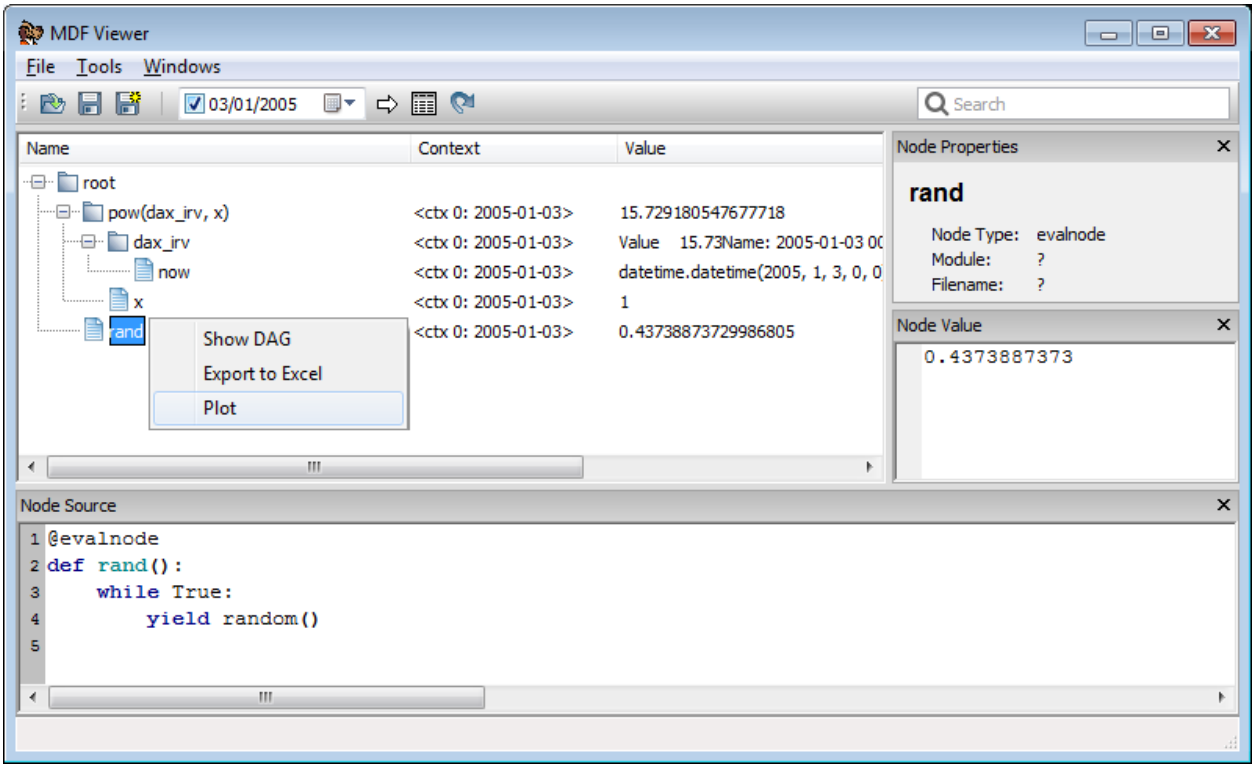
```
In [28]: %mdf_show df_node_pow
```



More nodes can be added to the open viewer using the same command.

```
In [29]: %mdf_show rand
```

To plot or export nodes select the nodes you want (use Shift or Ctrl to select multiple nodes) and then right click and select `plot` from the context menu. The same context menu may be used to export values to Excel or to render a graphical representation of the graph (requires Graphviz to be installed).



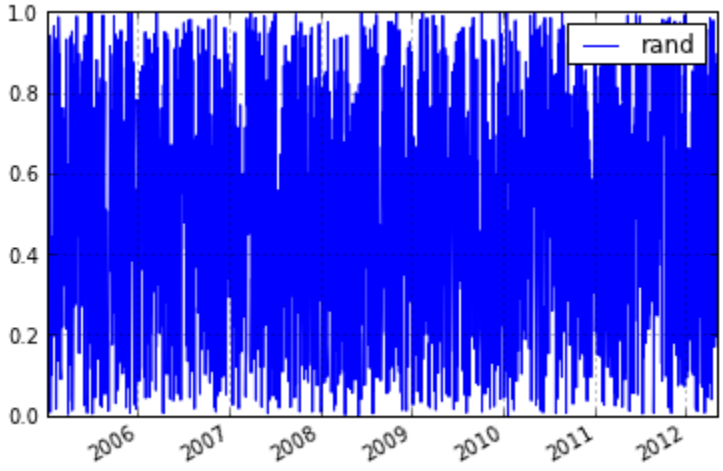
Once the viewer is open you can select one or more nodes and then use the magic command `%mdf_selected` to get your current selection in your IPython session

```
In [29]: %mdf_selected
Out[29]:
[(<ctx 0: 2005-02-11 at 115387248>,
  <<type 'mdf.nodes.MDFEvalNode'> [name=rand] at 0x30fee48>)]
```

This returns a list of contexts and node objects that correspond to what is selected in the viewer.

The magic functions `%mdf_plot`, `%mdf_df` and `%mdf_xl` can also be used to plot or get results for the currently selected nodes. To use the currently selected nodes don't specify any nodes at all on the command line.

```
%mdf_plot 2005-01-01 T
```



CHAPTER 7

API Reference

7.1 Nodes Types

mdf.**varnode**(*[name] [, default] [, category]*)

Creates a simple *MDFNode* that can have a value assigned to it in a context.

It may also take a default value that will be used if no specific value is set for the node in a context.

A varnode may be explicitly named using the name argument, or if left as None the variable name the node is being assigned to will be used.

```
my_varnode = varnode(default=100)
```

mdf.**evalnode**(*func [, filter] [, category]*)

Decorator for creating an *MDFNode* whose value is determined by calling the function func.

func should be a function or generator that takes no arguments and returns or yields the current value of the node.

If *func* is a generator instead of a function it will be advanced as the *now()* node is advanced. This can be used to calculate accumulated values and maintain internal state over evaluations:

```
@evalnode
def some_function():
    # Set initial value of 'accum' to 0
    accum = 0

    # yield the initial value of 'accum' (0 in this case)
    yield accum

    while True:
        accum += 1
        # yield the updated value of 'accum' on each evaluation
        yield accum
```

Yield essentially bookmarks the current execution point, and returns the supplied value (accum in the above example). When the node is evaluated again, execution will resume at the bookmark and continue until the next yield statement is encountered. By using an infinite while loop the node can be evaluated any number of times.

The above example can actually be shortened to:

```
@evalnode
def some_function():
    # Set initial value of 'accum' to 0
```

```
    accum = 0

    while True:
        yield accum
        accum += 1
```

In this case, yield will first return the initial value of 'accum'. On subsequent evaluations, the incrementation step will be also executed and the node will produce the updated value.

filter may be used in the case when *func* is a generator to prevent the node valuation being advanced on every timestep. If supplied, it should be a function or node that returns True if the node should be advanced for the current timestep or False otherwise.

mdf.**queuenode** (*func [, size] [, filter] [, category]*)
Decorator for creating an *MDFNode* that accumulates values in a *collections.deque* each time the context's date is advanced.

The values that are accumulated are the results of the function *func*. *func* is a node function and takes no arguments.

If *size* is specified the queue will grow to a maximum of that size and then values will be dropped off the queue (FIFO).

size may either be a value or a callable (i.e. a function or a node):

```
@queuenode(size=10)
def node():
    return x
```

or:

```
# could be an evalnode also
queue_size = varnode("queue_size", 10)

@queunode(size=queue_size)
def node():
    return x
```

or using the nodetype method syntax (see nodetype_method_syntax):

```
@evalnode
def some_value():
    return ...

@evalnode
def node():
    return some_value.queue(size=5)
```

mdf.**delaynode** (*func [, periods] [, initial_value] [, lazy] [, filter] [, category]*)
Decorator for creating an *MDFNode* that delays values for a number of periods corresponding to each time the context's date is advanced.

The values that are delayed are the results of the function *func*. *func* is a node function and takes no arguments.

`periods` is the number of timesteps to delay the value by.

`initial_value` is the value of the node to be used before the specified number of periods have elapsed.

periods, *initial_value* and *filter* can either be values or callable objects (e.g. a node or a function):

```
@delaynode(periods=5)
def node():
    return x
```

or:

```
# could be an evalnode also
periods = varnode("periods", 5)

@delaynode(periods=periods)
def node():
    return x
```

If `lazy` is True the node value is calculated after any calling nodes have returned. This allows nodes to call delayed version of themselves without ending up in infinite recursion.

The default for `lazy` is False as in most cases it's not necessary and can cause problems because the dependencies aren't all discovered when the node is first evaluated.

e.g.:

```
@delaynode(periods=10)
def node():
    return some_value
```

or using the nodetype method syntax (see nodetype_method_syntax):

```
@evalnode
def some_value():
    return ...

@evalnode
def node():
    return some_value.delay(periods=5)
```

mdf.**nansumnode**(*func [, filter] [, category]*)

Decorator that creates an *MDFNode* that maintains the *nansum* of the result of *func*.

Each time the context's date is advanced the value of this node is calculated as the nansum of the previous value and the new value returned by *func*.

e.g.:

```
@nansumnode
def node():
    return some_value
```

or using the nodetype method syntax (see nodetype_method_syntax):

```
@evalnode
def some_value():
    return ...

@evalnode
def node():
    return some_value.nansum()
```

mdf.**cumprodnode**(*func [, filter] [, category]*)

Decorator that creates an *MDFNode* that maintains the cumulative product of the result of *func*.

Each time the context's date is advanced the value of this node is calculated as the previous value muliplied by the new value returned by *func*.

e.g.:

```
@cumprodnode
def node():
    return some_value
```

or using the nodetype method syntax (see nodetype_method_syntax):

```
@evalnode
def some_value():
    return ...

@evalnode
def node():
    return some_value.cumprod()
```

TODO: That node needs a test for the argument skipna, since it is not entirely clear what it should do if the first value is na. It would be nice to be able to specify an initial value.

mdf.**ffillnode**(*func*[, *initial_value*])

Decorator that creates an *MDFNode* that returns the current result of the decoratored function forward filled from the previous value where the current value is NaN.

The decorated function may return a float, pandas Series or numpy array.

e.g.:

```
@ffillnode
def node():
    return some_value
```

or using the nodetype method syntax (see nodetype_method_syntax):

```
@evalnode
def some_value():
    return ...

@evalnode
def node():
    return some_value.ffill()
```

mdf.**rowiternode**(*func [, index_node=now] [, missing_value=np.nan] [, filter] [, category]*)

Decorator that creates an *MDFNode* that returns the current row of item of a pandas DataFrame, WidePanel or Series returned by the decoratored function.

What row is considered current depends on the *index_node* parameter, which by default is *now*.

missing_value may be specified as the value to use when the index_node isn't included in the data's index. The default is NaN.

delay can be a number of timesteps to delay the index_node by, effectively shifting the data.

ffill causes the value to get forward filled if True, default is False.

e.g.:

```
@rowiternode
def datarow_node():
    # construct a dataframe indexed by date
    return a_dataframe

@evalnode
```

```
    def another_node():
        # the rowiternode returns the row from the dataframe
        # for the current date 'now'
        current_row = datarow_node()
```

or using the nodetype method syntax (see nodetype_method_syntax):

```
    @evalnode
    def dataframe_node():
        # construct a dataframe indexed by date
        return a_dataframe

    @evalnode
    def another_node():
        # get the row from dataframe_node for the current_date 'now'
        current_row = dataframe_node.rowiter()
```

mdf.**returnsnode** (*func [, filter] [, category]*)

Decorator that creates an *MDFNode* that returns the returns of a price series.

NaN prices are filled forward. If there is a NaN price at the beginning of the series, we set the return to zero. The decorated function may return a float, pandas Series or numpy array.

e.g.:

```
    @returnsnode
    def node():
        return some_price
```

or using the nodetype method syntax (see nodetype_method_syntax):

```
    @evalnode
    def some_price():
        return ...

    @evalnode
    def node():
        return some_price.returns()
```

The value at any timestep is the return for that timestep, so the methods ideally would be called 'return', but that's a keyword and so returns is used.

mdf.**applynode** (*func, [, args=()] [, kwargs={}] [, category]*)

Return a new mdf node that applies *func* to the value of the node that is passed in. Extra *args* and *kwargs* can be passed in as values or nodes.

Unlike most other node types this shouldn't be used as a decorator, but instead should only be used via the method syntax for node types, (see nodetype_method_syntax) e.g.:

```
    A_plus_B_node = A.applynode(operator.add, args=(B,))
```

mdf.**lookaheadnode** (*func, periods [, offset=pa.datetools.BDay()] [, filter] [, category]*)

Node type that creates an *MDFNode* that returns a pandas Series of values of the underlying node for a sequence of dates in the future.

Unlike most other node types this shouldn't be used as a decorator, but instead should only be used via the method syntax for node types, (see nodetype_method_syntax) e.g.:

```
    future_values = some_node.lookahead(periods=10)
```

This would get the next 10 values of `some_node` after the current date. Once evaluated it won't be re-evaluated as time moves forwards; it's always the first set of future observations. It is intended to be used sparingly for seeding moving average calculations or other calculations that need some initial value based on the first few samples of another node.

The dates start with the current context date (i.e. *now()*) and is incremented by the optional argument *offset* which defaults to weekdays (see `pandas.datetools.BDay`).

Parameters

- **periods** (*int*) – the total number of observations to collect, excluding any that are ignored due to any filter being used.
- **offset** – date offset object (e.g. datetime timedelta or pandas date offset) to use to increment the date for each sample point.
- **filter** – optional node that if specified should evaluate to True if an observation is to be included, or False otherwise.

7.2 Node Factory Functions

mdf.**datanode**(*[name=None,] data [, index_node] [, missing_value] [, delay] [, name] [,filter] [,category]*)
Return a new mdf node for iterating over a dataframe, panel or series.

data is indexed by another node *index_node*, (default is *now()*), which can be any node that evaluates to a value that can be used to index into *data*.

If the *index_node* evaluates to a value that is not present in the index of the *data* then *missing_value* is returned.

missing_value can be a scalar, in which case it will be converted to the same row format used by the data object with the same value for all items.

delay can be a number of timesteps to delay the index_node by, effectively shifting the data.

ffill causes the value to get forward filled if True, default is False.

data may either be a data object itself (DataFrame, WidePanel or Series) or a node that evaluates to one of those types.

e.g.:

```
df = pa.DataFrame({"A" : range(100)}, index=date_range)
df_node = datanode(data=df)

ctx[df_node] # returns the row from df where df == ctx[now]
```

A datanode may be explicitly named using the name argument, or if left as None the variable name the node is being assigned to will be used.

mdf.**filternode**(*[name=None,] data [, index_node] [, delay] [, name] [,filter] [,category]*)
Return a new mdf node for using as a filter for other nodes based on the index of the data object passed in (DataFrame, Series or WidePanel).

The node value is True when the index_node (default=now) is in the index of the data, and False otherwise.

This can be used to easily filter other nodes so that they operate at the same frequency of the underlying data.

delay can be a number of timesteps to delay the index_node by, effectively shifting the data.

A filternode may be explicitly named using the name argument, or if left as None the variable name the node is being assigned to will be used.

7.3 Custom Node Types

mdf.**nodetype**(*func*)

decorator for creating a custom node type:

```
#
# create a new node type 'new_node_type'
#
@nodetype
def new_node_type(value, fast, slow):
    return (value + fast) * slow

#
# use the new type to create a node
#
@new_node_type(fast=1, slow=10)
def my_node():
    return some_value

# ctx[my_node] returns new_node_type(value=my_node(), fast=1, slow=10)
```

The node type function takes the value of the decorated node and any other keyword arguments that may be supplied when the node is created.

The node type function may be a plain function, in which case it is simply called for every evaluation of the node, or it may be a co-routine in which case it is sent the new value for each iteration:

```
@nodetype
def nansumnode(value):
    accum = 0.
    while True:
        accum = np.nansum([value, accum])
        value = yield accum

@nansumnode
def my_nansum_node():
    return some_value
```

The kwargs passed to the node decorator may be values (as shown above) or nodes which will be evaluated before the node type function is called.

Nodes defined using the @nodetype decorator may be applied to classmethods as well as functions and also support the standard node kwargs 'filter' and 'category'.

Node types may also be used to add methods to the MDFNode class (See nodetype_method_syntax):

```
@nodetype(method="my_nodetype_method")
def my_nodetype(value, scale=1):
    return value * scale

@evalnode
def x():
    return ...

@my_nodetype(scale=10)
def y():
    return x()
```

```
    # can be re-written as:
    y = x.my_nodetype_method(scale=10)
```

7.4 Pre-defined Nodes

mdf.**now**()
Pre-defined node present in every context that always evaluates to the date set on the context.

See *MDFContext.get_date()* and *MDFContext.set_date()*.

7.5 Functions

mdf.**shift**(*node, target [, values] [, shift_sets]*)
This function is for use inside node functions.

Applies shifts to the current context for each shift specified and returns the value of 'node' with each of the shifts applied.

If target and values are specified 'target' is a node to apply a series of shifts to, specified by 'values'.

If shifts_sets is specified, 'shift_sets' is a list of nodes to values dictionaries, each one specifying a shift.

If the same shift set dictionaries are used several times *ShiftSet* objects may be used instead which will be slightly faster. See *make_shift_set()*.

Returns a list of the results of evaluating node for each of the shifted contexts in the same order as values or shift_sets.

See *MDFContext.shift()* for more details about shifted contexts.

mdf.**run**(*date_range [, callbacks=[]] [, values={}] [, shifts=None] [, filter=None] [, ctx=None]*)
creates a context and iterates through the dates in the date range updating the context and calling the callbacks for each date.

If the context needs some initial values set they can be passed in the values dict or as kwargs.

For running the same calculation but with different inputs shifts can be set to a list of dictionaries of (node -> value) shifts.

If shifts is not None and num_processes is greater than 0 then that many child processes will be spawned and the shifts will be processed in parallel.

Any time-dependent nodes are reset before starting by setting the context's date to datetime.min (after applying time zone information if available).

mdf.**plot**(*date_range, nodes [, labels=None] [, values={}] [, filter=None] [, ctx=None]*)
evaluates a list of nodes for each date in date_range and plots the results using matplotlib.

mdf.**build_dataframe**(*date_range, nodes [, labels=None] [, values={}] [, filter=None] [, ctx=None]*)
evaluates a list of nodes for each date in date_range and returns a dataframe of results

mdf.**get_final_values**(*date_range, nodes [, labels=None] [, values={}] [, filter=None] [, ctx=None]*)
evaluates a list of nodes for each date in date_range and returns a list of final values in the same order as nodes.

mdf.**scenario**(*date_range, result_node, x_node, x_shifts, y_node, y_shifts [, values={}] [, filter=None] [, ctx=None] [, dtype=float]*)
evaluates a single result_node for each date in date_range and gets its final value for each shift in x_shifts and y_shifts.

x_shifts and y_shifts are values for x_node and y_node respectively.

result_node should evaluate to a single float, and the result is a 2d nparray

mdf.**plot_surface**(*date_range, result_node, x_node, x_shifts, y_node, y_shifts [, values={}] [, filter=None] [, ctx=None] [, dtype=float]*)
evaluates a single result_node for each date in date_range and gets its final value for each shift in x_shifts and y_shifts.

x_shifts and y_shifts are values for x_node and y_node respectively.

result_node should evaluate to a single float.

The results are plotted as a 3d graph and returned as a 2d numpy array.

mdf.**make_shift_set**(*shift_set_dict*)
Return a 'ShiftSet' object that encapsulates the information required to get a shifted context.

This can be used to pass to the *shift()* function instead of a dictionary for better performance when regularly shifting by the same thing.

7.6 Classes

7.6.1 MDFContext

class mdf.**MDFContext**
Nodes on their own don't have values, they are just the things that can calculate values.

Nodes only have values *in a context*.

Contexts can be thought of as containers for values of nodes.

__init__(*now*)
Initializes a new context with *now()* set to now (datetime).

save(*filename*, *bat_filename=None*, *start_date=None*, *end_date=None*)
Write the context and its state, including all shifted contexts and node states, to a binary file.

The resulting file can be re-loaded using *MDFContext.load()*.

If filename endswith .zip or .bz2 or .gz the data will be compressed. The *MDFContext.load()* method is able to load these compressed files.

Parameters

- **filename** – filename of the output file, or an open file handle.
- **start_date** – datetime used as an optional argument to start the mdf viewer in the .bat file.
- **end_date** – datetime used as an optional argument to start the mdf viewer in the .bat file.

static load(*filename*)
Load a context from a file and return a new MDFContext with the same state as the context that was saved (i.e. all the same shifted contexts and node values).

Parameters **filename** – filename of the file to load or an open file handle.

get_date()
returns the current date set on this context.

This is equivalent to getting the value of the *now()* node in this context.

set_date(*date*)
sets the current date set on this context.

This updates the value of the *now()* node in this context and also calls the update functions for any previously evaluated time-dependent nodes in this context.

get_value(*node*)
returns the value of the node in this context

set_value(*node*, *value*)
Sets a value of a node in the context.

set_override(*node*, *value*)
Sets an override for a node in this context.

__getitem__(*node*)
Gets a node value in the context.

See *get_value()*.

__setitem__(*node*, *value*)
Sets a node value in the context. If `value` is an `MDFNode` it is applied as an override.

See *set_value()* and *set_override()*.

shift(*shift_set*, *cache_context=True*)
create a new context linked to this context, but with nodes set to specific values.

`shift_set` is a dictionary of nodes to values. The returned shifted context will have each node in the dictionary set to its corresponding value.

If the same shift set is used several times a *ShiftSet* object may be used instead of a dictionary which will be slightly faster. See *make_shift_set()*.

If a value is an MDFNode then it will be applied as an override to the target node.

If a context has already been created with this shift that existing context is returned instead.

Shifted contexts are read-only.

If cache_context is True the shifted context will be cached and if *shift()* is called again with the same target and value the cached context will be returned. The exception to this is if `target` is *now()*, in which case a new shifted context is returned each time.

to_dot(*filename=None*, *nodes=None*, *colors={}*, *all_contexts=True*, *max_depth=None*, *rankdir="LR"*)
constructs a .dot graph from the nodes that have a value in this context and writes it to filename if not None.

colors can be used to override any of the colors used to color the graph. the defaults are:

```
defaults = {
    "node"      :   "white",
    "nownode"   :   "darkorchid1",
    "queuenode" :   "darksalmon",
    "varnode"   :   "deepskyblue",
    "shiftnode" :   "gold",
    "headnode"  :   "olivedrab3",
    "edge"      :   "black",
    "nowedge"   :   "darkorchid4",
    "varedge"   :   "deepskyblue4",
    "shiftedge" :   "gold4",
    "context"   :   "grey90",
    "module0"   :   "grey81",
```

```
            "module1"    :    "grey72"
        }
```

If all_contexts is true it will look for head nodes in all contexts, otherwise only this context will be used.

If max_depth is not None the graph will be truncated so that all nodes are at most max_depth levels deep from the root node(s).

rankdir sets how the graph is ordered when rendered. Possible values are:

- "TB" : top to bottom
- "LR" : left to right
- "BT" : bottom to top
- "RL" : right to left

returns a pydot.Graph object

7.6.2 MDFNode

class mdf.**MDFNode**

Nodes should be viewed as opaque objects and not instanciated through anything other than the decorators provided.

They are callable objects and should be called from inside other node functions.

When called they are evaluated in the current context and value is returned. If called multiple times a cached value is returned unless the node has been marked as requiring re-evaluation by one of its depenedencies changing.

7.6.3 MDFEvalNode

class mdf.**MDFEvalNode**

Sub-class of *MDFNode* for nodes that are evaluated rather than plain value storing nodes.

This is an opaque type and shouldn't be used to construct nodes. Instead use the node type decorators.

7.6.4 CSVWriter

class mdf.**CSVWriter**(*fh*, *nodes*, *columns=None*)

callable object that appends values to a csv file For use with mdf.run

__init__(*fh*, *nodes*[, *columns=None*])

Writes node values to a csv file for each date.

'fh' may be a file handle, or a filename, or a node.

If fh is a node it will be evaluated for each context used and is expected to evaluate to the filename or file handle to write the results to.

7.6.5 DataFrameBuilder

class mdf.**DataFrameBuilder**(*nodes*, *contexts=None*, *dtype=<type 'object'>*, *sparse_fill_value=None*, *filter=False*, *start_date=None*)

__init__(*nodes*[, *labels=None*])
Constructs a new DataFrameBuilder.

dtype and sparse_fill_value can be supplied as hints to the data type that will be constructed and whether or not to try and create a sparse data frame.

If *filter* is True and the nodes are filtered then only values where all the filters are True will be returned.

NB. the labels parameter is currently not supported

clear()

get_dataframe([*ctx=None*])

dataframes
all dataframes created by this builder (one per context)

dataframe

plot([*show=True*])
plots all collected dataframes and shows, if show=True

7.6.6 FinalValueCollector

class mdf.**FinalValueCollector**(*nodes*)
callable object that collects the final values for a set of nodes. For use with mdf.run

__init__(*nodes*)

clear()
clears all previously collected values

get_values([*ctx=None*])
returns the collected values for a context

get_dict([*ctx=None*])
returns the collected values as a dict keyed by the nodes

values
returns the values for the last context

CHAPTER 8

Implementation Details

The section of the documentation is an overview of the internal design and implementation of MDF. It should not be necessary to read or understand this section in order to use MDF.

8.1 Building MDF

8.1.1 Use of Cython

Cython is used extensively throughout MDF. It is used in such a way, however, that it's still possible to call MDF code *without* compiling it to allow easier debugging of the MDF internals.

All of the implementation code is in plain `.py` files with additional type information given using `cython.declare()` for local variables and `.pxd` files for class and module attribute type information. The modules are compiled into separate extensions, and type information is shared between the compilation units via the `.pxd` files.

There is significant overhead calling a Python function compared with calling a C function. Even when Cythoned methods are Cythoned using the `cpdef` keyword so they can be called both from Python or from C there is still additional overhead when the method is called compared with calling a plain C function. This is why the Cythoned classed have some `cdef` private methods with corresponding `cpdef` public methods. Internally the `cdef` methods should always be called unless it's expected that the method may be overridden by a Python subclass.

Whenever making changes to MDF the code must be profiled before and after as what look like small changes can drastically change the performance due to the number of times some of the internal functions and methods get called.

8.1.2 Compiling MDF

Building MDF requires Cython version 0.16 or later to be installed.

As with any Cythoned package it also requires a distutils compatible C compiler (e.g. Visual Studio 2008 for Windows or GCC for linux).

MDF can be compiled *in-place* to allow you to test changes and use your local check out of MDF in the usual Python way using setup.py:

```
python setup.py build_ext --inplace
```

Or to build the egg:

```
python setup.py bdist_egg
```

Current setuptools and Cython (0.16) doesn't pick up changes to `.pxd` files correctly when determining dependencies, and so changes to `.pxd` files won't trigger a recompilation of all affected object files. If this happens the easiest work around is to simply delete the generated `.c` files that will be in the same folder as their corresponding `.py` files.

8.1.3 Running/Debugging Non-Cythoned MDF

To be able to import the MDF modules without first compiling you need to have Cython installed. Cython is not required to import the modules after compilation.

There are various places where it has not been possible to have the pure-python version of the code identical to the Cythonized code. These cases are clearly commented, and in order to use MDF without compiling it you need to first un-comment these bits of code.

Follow the instructions in the comments regarding Cython in the following files:

- nodes.py
- context.py
- cqueue.py
- ctx_pickle.py

Although you will be able to use MDF in its uncompiled state for debugging its internals you will find it will be orders of magnitude slower than the compiled version.

8.1.4 Debugging the Compiled Cython Code

In setup.py there is a module variable `cdebug`. Set this to `True` to enable the compiler and linker flags to build debug versions of the extensions. This only sets the Visual Studio debug flags, and for GCC you should change them to whatever flags you required (usually `-g` is sufficient).

Once you have rebuilt the extensions with the debug flags set you can now attach a debugger to a running python instance in order to debug the cythoned MDF functions.

8.1.5 Profiling MDF

MDF includes its own counters and timers for profiling code run using MDF. This should be used to identify hotspots in user code. See `MDFContext.ppstats()`.

If necessary further profiling can be done using cPython. A visual profiler such as RunSnakeRun or kCacheGrind can be useful for understanding the cProfiler output. kCacheGrind provides more detail than RunSnakeRun but requires the cProfile output to be converted to a calltree using the *pyprof2calltree* pacakge.

Profiling MDF Internals

cProfile can also be used to profile the MDF internal Cythoned functions and methods. In setup.py there is a module variable `cython_profile`. Set this to `True` to enable profiling of Cythoned code.

Once you have rebuilt the extensions with the profile flag set you can now use cProfile to profile an application using MDF with the time spent in Cythoned functions included.

The addition of the profiling code to every Cythoned function adds significant overhead as many of the functions have been optimised to be very lightweight and may be inlined by the compiler. Adding the profiling code bloats these functions and distorts the running time significantly and it can therefore be very hard to determine accurate timings and hotspots using this method.

It is better to use a non-invasive statistical profiler such as Intel VTune or Sleepy (see 'Very Sleepy' for Windows). To profile MDF with one of these profilers you will need to build MDF with debug symbols using the `cdebug` setting in setup.py. For more accurate profiling you may want to build with compiler optimisations as well as debug symbols, in which case remove `/Od` from the compiler flags.

Profiling with a statistical profiler requires a little more knowledge and intuition about how the Python code is translated to C by Cython. The Cython code is annotated with the original Python code which makes this much easier, and browsing the code around the functions of interest before looking at the profiling results will make understanding the results of the profiling simpler. This is by far the best way to get a proper feel for where time is being spent inside MDF though and it is worth persevering.

8.2 Source Code Overview

8.2.1 context.py

context.py contains (almost) everything to do with *MDFContext*. It also includes a class `MDFNodeBase` from while *MDFNode* is derived. It's done this way with the node base class in context.py so that the context code can call C methods on the cythoned MDFNode objects without having to cimport `nodes.pxd` as that would result in a circular dependency.

context.py defines the following classes:

- *MDFContext*
- `MDFNodeBase`
- `ShiftSet`

and the public API functions:

- *shift()*
- `get_nodes()`
- *make_shift_set()*

8.2.2 nodes.py

nodes.py defines the following classes:

- *MDFNode*
- `MDFVarNode`
- *MDFEvalNode*
- `MDFTimeNode`
- `MDFIterator`
- `MDFIteratorFactory`
- `MDFIteratorFactory`
- `MDFCallable`

and the public API functions:

- *varnode()*
- `vargroup()`

- *evalnode()*
- *now()*

These classes are almost always only used internally. The API functions return instances of the node classes and so it's almost never necessary to refer to any of these classes outside of MDF.

NodeState is the per context state associated with a particular node. This isn't exposed outside of the Cythoned code and for external use. If it's referenced at all should be considered an opaque type (hence not appearing in the API reference).

8.2.3 cqueue.py

cqueue is an implementation of a double ended queue, like collections.deque. Although it's a queue it's specialised to represent a stack efficiently. Underlying it is a normal python list, and as items are pushed on it grows as it runs out of space. It keeps an index to the start and end of the queue and so popping items off either end simply means moving these indexes.

Because the items aren't reshuffled popping items off the left and adding to the right would result in more and more memory being allocated, but constantly pushing and popping the right is much faster than a deque, which is what this container is used for.

This could probably be improved by writing it in plain C rather than using a Python list, but at the time of writing it was sufficiently faster than collections.deque for what it's being used for that it wasn't optimised further.

8.2.4 nodetypes.py

nodetypes.py is where the *nodetype()* decorator is defined, and also all the various classes that are necessary for implementing more general node types derived from *MDFEvalNode*:

- MDFCustomNode
- MDFCustomNodeIterator
- MDFCustomNodeIteratorFactory
- MDFCustomNodeMethod
- MDFCustomNodeDecorator

MDFCustomNode is derived from *MDFEvalNode* and is constructed with an additional function, iterator or generator that converts the result of the evalnode to whatever the specific nodetype should return.

MDFCustomNodeDecorator is what's returned by the *nodetype()* decorator, which is itself a decorator. It converts whatever function (or generator or iterator class) it decorates into a node type decorator.

All the built-in node type decorators use the *nodetype()* and subclasses of MDFIterator to achieve the various different calculations.

MDFCustomNodeMethod is what's used to add the methods to all *MDFEvalNode* instances (see nodetype_method_syntax). It's a callable object that when called creates or fetches a previously created node. The returned node is the node it was called on wrapped with a node type.

8.2.5 runner.py

runner.py is where the various functions for running an MDF graph over time and extracting values, including parallel computation of scenarios.

8.2.6 to_dot.py

to_dot.py implements the *MDFContext.to_dot()* method. This uses `pydot` and `Graphviz` to render the graph as an image in a variety of different formats (.dot, .svg, .png, e.t.c.).

8.2.7 parser.py

Parsing code for use by the magic ipython functions and also for parsing Python code to find the left hand side of node assignments for defaulting node names, e.g.:

```
a_var_node = varnode()
```

The parser looks at the callstack and parses the line of source code to get `"a_var_node"` to use as the name for this node.

8.2.8 ctx_pickle.py

All the pickling code for contexts and nodes is separated out from the main files, and is implemented as functions in this file. These functions are imported from node.py and context.py and called from the various `__reduce__` methods on the associated classes.

8.2.9 builders sub-package

The builders sub-package is where all the provided callable objects intended to be used with *run()* are.

8.2.10 io sub-package

The methods *MDFContext.save()* and *MDFContext.load()* are implemented in this sub-package. The serialisation is done via pickling in ctx_pickle.py, but this sub-package can also read and write compressed files.

8.2.11 pylab sub-package

The pylab sub-package is where all the various magic ipython function are. This should only be imported for interactive use not from scripts as it depends on IPython.

8.2.12 regression sub-package

Regression testing is done by evaluating nodes in two different processes started from different virtualenvs. The code in this package manages starting the processes in the correct virtualenvs and collecting the values of the nodes over time in both child processes.

The data collection is done by a specialized builder, `DataFrameDiffer`, but other differ could be written by sub-classing `Differ`.

The interprocess communication is done using Pyro and the proxy objects from the remote sub-package.

8.2.13 remote sub-package

remote contains code shared between the various parallel processing components of MDF for creating subprocesses and the Pryo server and objects used for interprocess communication.

It also contains custom Pyro serialisation functions that autmatically compress data larger than a certain size on the fly using bz2.

8.2.14 tests sub-package

Unit tests.

8.3 Node Evaluation

Nodes only have values in a context, so it makes sense that to get a nodes value it's evaluated by starting with the context. Indexing into the context with a node (see *MDFContext.__getitem__()*) calls *MDFContext.get_value()*, which is a public API method. This calls an internal C method `MDFContext._get_node_value()` which is where the work is actually done.

Once inside a node evaluation to avoid passing the current context around to every node call to allow the nodes to evaluate other nodes, there's the notion of a currently active context. The currently active context is stored in a dictionary that maps thread id to the MDFContext (_current_contexts in context.py). See also `_get_current_context()`.

8.3.1 Dependencies

Although it's common to talk about one node being dependent on another, actually the dependencies are between a *node in a context* and another *node in a context*. The contexts need not be the same as one node can call another node in another context (when shifting, for example).

Dependency tracking is facilitated by `MDFContext._get_node_value()` keeping track of the current node being evaluated (and the context it's being evaluated in). These are kept in a stack (actually it's a queue - but conceptually it's a stack) so to find the node that's calling the current node being evaluated it just needs to look at the last item in the stack. Before dropping into the node evaluation itself the current node and context are pushed onto the stack.

All dependencies are discovered at runtime by observing which nodes call other nodes. This can either be directly, or a node may call a function that then calls other nodes. Before anything is evaluated MDF knows nothing about the dependencies between nodes.

Each context has its own node evaluation stack and so to find the node and context calling the current node and context the stack belonging to the previous context is examined. A restriction is that the same context can't be used from different threads concurrently, but different contexts can be used in different threads because the current context is effectively a per thread variable (although it's in a dict rather than TLS).

Once discovering what the calling node and context is (if any), the current node is pushed onto the current context's node evaluation stack and `MDFNodeBase.get_value()` is called to retrieve or calculate the node value in the context. After the node has been evaluated the node is popped off the evaluation stack and a dependency is established between the previous node and context and the current node and context by calling `MDFNodeBase._add_dependency()`.

As the dependencies are context dependent the relationships are stored on the `NodeState` object associated with the node and context pair.

`MDFNodeBase._add_dependency()` is written such that re-calling it for the same node and context is fast, and so it's called everytime `MDFContext._get_node_value()` is called regardless of whether the dependency has been discovered previously or not.

8.3.2 Node evaluation

As mentioned above the entry point for evaluating a node is *MDFContext.__getitem__()* or *MDFContext.get_value()*, which in turn calls the internal C method `MDFContext._get_node_value()`. This ultimately calls `MDFNodeBase.get_value()`, which each derived node class implements.

Varnodes

varnodes are the simplest type of node. Getting their value just involves looking to see if the `NodeState` has a value for the current context and return that. If there is no value in the `NodeState` the default value for the node is returned if there is one, or an exception is raised.

Evalnodes

Eval nodes wrap a function, generator or `MDFIterator`. To get their value the wrapped function is called and the result is cached on the `NodeState` for the context the node is being evaluated in.

Once a node has been evaluated once it is marked as not needing to be re-evaluated. If the node is evaluated again the previous result is returned as long as none of the dependencies of the node have changed.

Dirty flags

Whether the node needs evaluating or not is determined by the `dirty_flags` on the `NodeState` object for the node in each context. When the node is evaluated these flags are cleared to indicate the node isn't dirty and the previously cached value can be used. When any node is changed the `MDFNode.set_dirty()` method is called which marks the node as dirty and then marks all the nodes calling this node as dirty if they are not already flagged dirty. The dirty flags propagate all the way through the graph for all nodes and context pairs that depend on the node in the context being dirtied.

Generators and iterators

If an evalnode wraps a generator or iterator then it is advanced each time the context's date (*now()*) is advanced.

The dirty flags mentioned previously are a bit field. One of these bits is reserverd for updates to time (*now()*). When the time changes all nodes that are dependent on *now()* have the `DIRTY_FLAGS.TIME` bit in their dirty flags set. In addition, any generators or iterators not dependent on *now()* also have the `DIRTY_FLAGS.TIME` bit in their dirty flags set (as well as any dependent nodes, as explained in the previous section).

When the evalnode is evaluated and *only* the `DIRTY_FLAGS.TIME` bit is set then, if the node is a generator or iterator, instead of completely re-evaluating the node the previously instantiated iterator is advanced. The iterator is stored on the `NodeState` for the node and context.

Because generators and iterators need to be advanced on *every* timestep, regardless of whether their value is actually used on any particular timestep, `MDFContext.set_data()` evaluates all of them after marking them as dirty (just with the `DIRTY_FLAGS.TIME` bit). Other nodes could get the value of an iterator node once and then not look at the value for a number of timesteps; this would cause problems as the iterator wouldn't have been steped through the intermediate timesteps and so would have the wrong value, and so evaluating them in the *MDFContext.set_date()* prevents that problem from occurring.

8.3.3 Shifted contexts

Shifted contexts are created by shifting any other context (i.e. shifted or non-shifted) via the *MDFContext.shift()* method.

Shift operations are commutative and associative, i.e:

```
# commutative
ctx.shift({a: x}).shift({b: y}) == ctx.shift({b: y}).shift({a: x})

# associative
ctx.shift({a: x, b: y}).shift({c: z}) == ctx.shift({a: x}).shift({b: y, c: z})
```

Shifted contexts are all stored in a flat structure on the **root** context. Shifted contexts have a parent context, but that parent is *always the root context*. This flat structure is what facilitates the commutative and associative properties of the shift operation. There is a method `MDFContext.is_shift_of()` to determine if one context is a shift of another. This is determined by looking at the intersection of the two contexts' shift sets rather than having an explicit hierachy of contexts.

Each shifted context is keyed by its `shift_set`, which is the dictionary of nodes to their values in the shifted context. Any shift resulting in the same net shift set returns the same shifted context.

Node evaluation in shifted contexts

When evaluating nodes in a shifted context the naive approach would be to just evaluate that node and all its dependencies in that shifted context. This however would cause problems where varnodes are set in the root context (or other shifted contexts that the context the node is being evaluated in is itself a shift of), because the value wouldn't be available in the shifted context. It would also be inefficient as nodes would potentially be evaluated multiple times when they are needed in different contexts, even if the value in each case would be the same because the node doesn't depend on some or all of the shifted nodes.

For these reasons there is the concept of the *alt context*. The alt context is a property of a node and a context, and is the least shifted context the node can be evaluated in that will have the same result as if it were evaluted in the orignal context.

For example:

```
a = varnode()
b = varnode()

@evalnode
def foo():
    return a()


ctx = MDFContext()
shifted_a = ctx.shift({a : 1})
shifted_b = shifted_a.shift({b : 2})

shifted_b[foo] == shifted_a[foo]
```

Here evaluating `foo` in a context where `b` is shifted has no effect, but shifting `a` does because `foo` depends on `a`. Therefore the *alt context* for `shifted_b[foo]` is `shifted_a`.

Even if the context `shifted_a` hadn't been explicitly created as above the alt context would still be that context, i.e. a context with shift set `{a : 1}`.

Determining the alt context

For varnodes getting the alt context from a shifted context is simple. If the shift set of the shifted context includes the varnode then the alt context is the parent (root) context shifted by the varnode and the shift value. All other shifts are irrelevant for the varnode and so are ignored when getting the alt context.

Evalnodes are a bit tricker as they have to be evaluated once before the dependencies are known. The first time round they are evaluated in the context they're called in. Before setting the value of the node in the `NodeState` however the dependencies are analysed and the least shifted context (alt context) is determined by checking the dependencies of all the called nodes and the shift set of the original context. All the dependencies and state of the node in the original context is transfered to the newly discovered alt context.

Once the alt context for a node and context has been determined it's cached in the `NodeState`. If a dependency changes then that cached value is cleared and it will be re-determined the next time it's needed. If a node has conditional dependencies (i.e. new dependencies are discovered after the initial evaluation) that can cause the alt context to change, and this causes an an exception to be raised. Allowing the alt context to change part way through an evaluation would be danergous as there could be accumulated state in the original alt context that wouldn't be consistent in the new alt context.

8.4 Node Types

Other node types (e.g. *queuenode()*, *ffillnode()*) are built on top of *MDFEvalNode*. The basic concept is that a `nodetype` does the job of a normal evalnode but then calls a second function on the result of that to transform it in some way.

The second function that does that transformation is referred to as the `node type function` and the inner function that gets evaluated to provide the input to the `node type function` is the `node function`.

Here's an exampple of a very simple custom node type that will help illustrate how node types work:

```
@nodetype
def node_type_function(value):
    return value * 2

@node_type_function
def node_function():
    return x
```

When evaluating the node `node_function` the order of execution is to first call the python function `node_function` and then to pass the result of that to the python function `node_type_function`. The result of `node_type_function` becomes the value of the node `node_function`.

To understand how this works it helps to talk about what types are used and what the result of these decorators is.

`nodetype` returns an instance of a `MDFCustomNodeDecorator`. This instance keeps a reference to the decorated function. It's a callable object and its call method works as a decorator.

So, the decorated `node_type_function` is an instance of `MDFCustomNodeDecorator`. When used as a decorator as `@node_type_function` on another function it returns an instance of `MDFCustomNode`. This `MDFCustomNode` is a subclass of *MDFEvalNode* as is instantiated with the node function and keeps a reference to the `node type function`. `MDFCustomNode` differs from *MDFEvalNode* by calling its node type function after doing the normal evaluation of the node function. The result of this is what gets returned as the final value of the node.

Things get a little more complicated when adding arguments to the node type, for example:

```
@nodetype
def node_type_function_2(value, multipler):
    return value * multiplier

@node_type_function_2(multiplier=2)
def node_function_2():
    return x
```

To pass the arguments from the node instantiation (when @node_type_function_2 is called with node_function_2) to the node type function `node_type_function_2` the args have to be stored by the `MDFCustomNode` instance. This is exactly what happens, and when the custom node is evaluated it calls the node type function with these stored arguments.

If any of the arguments are nodes they automatically get evaluated before being passed to the node type function. Occasionally it is necessary to pass nodes in as arguments. `MDFCustomNode` checks a class property `node_kwargs` and doesn't automatically evaluate any args in that list. By subclassing `MDFCustomNode` this can be set for specific node types.

8.4.1 Generators and iterators

MDFIterator

`MDFIterator` is a base class that is recognized by the MDF toolkit as being an iterator and is treated in exactly the same way as a generator. The reason for using an `MDFIterator` instead of a generator is that `MDFIterator` instances may be pickleable whereas generators are not.

Node type functions may also be a generator or iterator (`MDFIterator`). If they are then the first time the node type function is called it will be called with the node function results and all the arguments from when the node of that type was created (e.g. `muliplier` in the example from the previous section).

`next()` is then called to get the initial value. Subsequent values are obtained by advancing the iterator sending in the new node function result by the `send()` method of the generator or `MDFIterator` instance.

For this to work correctly the node function that is used to initialize the base class *`MDFEvalNode`* depends on whether the node type function or the node function is a generator or not. If either of them are then the function called by the underlying MDFEvalNode code must return an iterator that will work in the same way as if the final node was a generator. This is done using another class, `MDFIteratorFactory`. This is another callable that when called returns a `MDFIterator`, which the underlying *`MDFEvalNode`* code understands and treats like a generator.

8.4.2 Method syntax for node types

When a node type is registered using the *`nodetype()`* decorator a method name can be specified. This automatically adds two new methods to *`MDFNode`* (and any existing instances) - one that returns a node of the node type and one that returns the value of a node of that node type.

Both methods work in exactly the same way. A new `MDFCustomNode` instance is constructed with the node type function and using the node the method is called on as the node function. If the same method is called again with the same arguments on the same node then it returns the node constructed previously.

It works by creating the new methods when the *`nodetype()`* decorator is called. The methods are actually instances of `MDFCustomNodeMethod` which is yet another callable class. Calling that creates the new node or fetches it if it was created already.

The new instances of `MDFCustomNodeMethod` are added to `MDFNode._additional_attrs_`, which is checked in `MDFNode.__getattr__` allowing for new attributes to be added dynamically to the cythoned class.

8.5 Class nodes

Nodes can be declared as properties of classes as well as modules. These nodes may take a single argument, which will be the class the node is being called on. If they take no arguments they behave the same way as a normal node.

Class nodes have to be aware of the class they're defined on and the class they're bound to (accessed from). Consider the following:

```
class A(object):

    @evalnode
    def foo(cls):
        return "Declared in A, called on %s" % cls.__name__

class B(A):

    @evalnode
    def foo(cls):
        return "Overridden in B (%s)" % super(B, cls).foo()
```

In one sense this code just declares two nodes, `A.foo` and `B.foo`. That's a slight simplication of what's actually going on though; if there were only two then super(B, cls).foo() would have to evaluate `A.foo()` which would return `Declared in A, called on A`. So, there are actually threee [1] nodes, `A:A.foo`, `B:B.foo()` and `B:A.foo()`.

What the code above declares are *unbound* nodes. That is, nodes that have no knowledge of the classes they belong to. When they are *accessed* from the class (i.e. `A.foo` accesses foo from A) they are then bound to the class at that point. Binding creates a new node that includes everything from the original unbound node definition and information about the class the new node is bound to.

MDFEvalNode is a descriptor and so the process of binding nodes to a class is done by `MDFEvalNode.__get__()`. This is called whenever a node is accessed as a property of a class. All evalnodes have a dictionary of classes to bound versions of themselves. When accessing a node multiple times on the same class the same bound node is returned each time. If no bound node exists for the unbound node and class then a new node is created.

To bind the node to a class the function that it references must also be bound (to create a staticmethod or classmethod) and the new node is created using that bound method. As keyword arguments to the node may also reference unbound functions or nodes those too need to be bound. `MDFEvalNode._bind()` handles binding any additional functions or nodes and may be overridden by any subclasses requiring additional objects to also be bound. The helper method `MDFEvalNode._bind_function` is used to create bound versions of individual functions, methods and other callable object types.

[1] More accurately there are three *bound* nodes and two *unbound* nodes, but the unbound nodes aren't accessible outside of the class definition.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

m

Symbols

A

B

C

D

E

F

G

L

M

N

P

Q

R

S

T

V