loman Documentation

Release 0.1.2

Ed Parcell

Apr 29, 2017

Contents

Loman is a Python library to deal with complex dependencies between sets of calculations. You can think of it as make for calculations. By keeping track of the state of your computations, and the dependencies between them, it makes understanding calculation processes easier and allows on-demand full or partial recalculations. This makes it easy to efficiently implement robust real-time and batch systems, as well as providing powerful mechanism for interactive work.

CHAPTER 1

User Guide

Introduction

Loman is a Python library for keeping track of dependencies between elements of a large computation, allowing you to recalculate only the parts that are necessary as new input data arrives, or as you change how certain elements are calculated.

It stems from experience with real-life systems taking data from many independent source. Often systems are implemented using sets of scheduled tasks. This approach is often pragmatic at first, but suffers several drawbacks as the scale of the system increases:

- When failures occur, such as a required file or data set not being in place on time, then downstream scheduled tasks may execute anyway.
- When re-runs are required, typically each step must be manually invoked. Often it is not clear which steps must be re-run, and so operators re-run everything until things look right. A large proportion of the operational overhead of many real-world systems comes from needing enough capacity to improvised re-runs when systems fail.
- As tasks are added, the schedule may be become tight. It may not be clear which items can be moved earlier or later to make room for new tasks.

Other problems occur at the scale of single programs, which are often programmed as a sequential set of steps. Typically any reasonably complex computation will require multiple iterations before it is correct. A limiting factor is the speed at which the programmer can perform these iterations - there are only so many minutes in each day. Often repeatedly pulling large data sets or re-performing lengthy calculations that will not have changed between iterations ends up substantially slowing progress.

Loman aims to provide a solution to both these problems. Computations are represented explicitly as a directed acyclic graph data structures. A graph is a set of nodes, each representing an input value calculated value, and a set of edges (lines) between them, where one value feeds into the calculation of another. This is similar to a flowchart, the calculation tree in Excel, or the dependency graph used in build tools such as make. Loman keeps track of the current state of each node as the user requests certain elements be calculated, inserts new data into input nodes of the graph, or even changes the functions used to perform calculations. This allows analysts, researchers and developers to iterate quickly, making changes to isolated parts of complicated calculations.

Loman can serialize the entire contents of a graph to disk. When failures occur in batch systems a serialized copy of its computations allows for easy inspection of the inputs and intermediates to determine what failed. Once the error is diagnosed, it can be fixed by inserting updated data if available, and only recalculating what was necessary. Or alternatively, input or intermediate data can be directly updated by the operator. In either case, diagnosing errors is as easy as it can be, and recovering from errors is efficient.

Finally, Loman also provides useful capability to real-time systems, where the cadence of inputs can vary widely between input sources, and the computational requirement for different outputs can also be quite different. In this context, Loman allows updates to fast-calculated outputs for every tick of incoming data, but may limit the rate at which slower calculated outputs are produced.

Hopefully this gives a flavor of the type of problem Loman is trying to solve, and whether it will be useful to you. Our aim is that if you are performing a computational task, Loman should be able to provide value to you, and should be as frictionless as possible to use.

Installation Guide

Using Pip

To install Loman, run the following command:

```
$ pip install loman
```

If you don't have pip installed (tisk tisk!), this Python installation guide can guide you through the process.

Dependency on graphviz

Loman uses the graphviz tool, and the Python graphviz library to draw dependency graphs. If you are using Continuum's excellent Anaconda Python distribution (recommended), then you can install them by running these commands:

```
$ conda install graphviz
$ python install graphviz
```

Windows users: Adding the graphviz binary to your PATH

Under Windows, Anaconda's graphviz package installs the graphviz tool's binaries in a subdirectory under the bin directory, but only the bin directory is on the PATH. So we will need to add the subdirectory to the path. To find out where the bin directory is in your installation, use the where command:

```
C:\>where dot
C:\ProgramData\Anaconda3\Library\bin\dot.bat
C:\>dir C:\ProgramData\Anaconda3\Library\bin\graphviz\dot.exe
 Volume in drive C has no label.
 Volume Serial Number is XXXX-XXXX

 Directory of C:\ProgramData\Anaconda3\Library\bin\graphviz

01/03/2017  04:16 PM             7,680 dot.exe
               1 File(s)          7,680 bytes
               0 Dir(s)  xx bytes free
```

You can then add the subdirectory graphviz to your PATH. You can either do this through the Windows Control Panel, or in an interactive session, by running this code:

```
import sys, os
def ensure_path(path):
    paths = os.environ['PATH'].split(';')
    if path not in paths:
        paths.append(path)
        os.environ['PATH'] = ';'.join(paths)
ensure_path(r'C:\ProgramData\Anaconda3\Library\bin\graphviz')
```

Quick Start

In Loman, a computation is represented as a set of nodes. Each node can be either an input node, which must be provided, or a calculation node which can be calculated from input nodes or other calculation nodes. In this quick start guide, we walk through creating computations in Loman, inspecting the results and controlling recalculation.

To keep things simple, the examples will perform simple calculations on integers. Our focus initially is on the dependency between various calculated items, rather than the calculations themselves, which are deliberately trivial. In a real system, it is likely that rather than integers, we would be dealing with more interesting objects such as Pandas DataFrames.

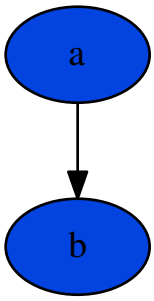
Creating and Running a Computation

Let's start by creating a computation object and adding a couple of nodes to it:

```
>>> comp = Computation()
>>> comp.add_node('a')
>>> comp.add_node('b', lambda a: a + 1)
```

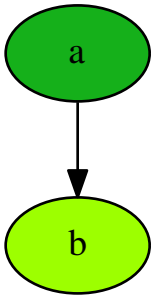
Loman's computations have a method `draw` which lets us easily see a visualization of the computation we just created:

```
>>> comp.draw()
```



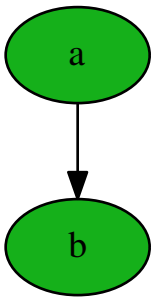
Loman gives us a quick and easy way to visualize our computations as a graph data structure. Each node of the graph is a colored oval, representing an input value or calculated value, and each edge (line) shows where the calculation of one node depends on another. The graph above shows us that node **b** depends on node **a**. Both are colored blue as neither has a value. Let's insert a value into node **a**:

```
>>> comp.insert('a', 1)
>>> comp.draw()
```



Now we see that node **a** is colored dark green, indicating that it is up-to-date, since we just inserted a value. Node **b** is colored light green, indicating it is computable - that is to say that it is not up-to-date, but it can immediately be calculated. Let's do that:

```
>>> comp.compute_all()
```



Now **b** is up-to-date, and is also colored dark green.

Inspecting Nodes

Loman gives us several ways of inspecting nodes. We can use the `value` and `state` methods:

```
>>> comp.value('b')
2
>>> comp.state('b')
<States.UPTODATE: 4>
```

Or we can use `v` and `s` to access values and states with attribute-style access. This method of access works well with the auto-complete feature in IPython and Jupyter Notebook, but it is only able to access nodes with valid alphanumeric names:

```
>>> comp.v.b
2
>>> comp.s.b
<States.UPTODATE: 4>
```

The []-operator provides both the state and value:

```
>>> comp['b']
NodeData(state=<States.UPTODATE: 4>, value=2)
```

The `value` and `state` methods and `v` and `s` accessors can also take lists of nodes, and will return corresponding lists of values and states:

```
>>> comp.value(['a', 'b'])
[1, 2]
>>> comp.state(['a', 'b'])
[<States.UPTODATE: 4>, <States.UPTODATE: 4>]
>>> comp.v[['a', 'b']]
[1, 2]
>> comp.s[['a', 'b']]
[<States.UPTODATE: 4>, <States.UPTODATE: 4>]
```

There are also methods `to_dict()` and `to_df()` which get the values of all the nodes:

```
>>> comp.to_dict()
{'a': 1, 'b': 2}
>>> comp.to_df()
             state  value  is_expansion
a  States.UPTODATE      1           NaN
b  States.UPTODATE      2           NaN
```

More Ways to Define Nodes

In our first example, we used a lambda expression to provide a function to calculate **b**. We can also provide a named function. The name of the function is unimportant. However, the names of the function parameters will be used to determine which nodes should supply inputs to the function:

```
>>> comp = Computation()
>>> comp.add_node('input_node')
>>> def foo(input_node):
...   return input_node + 1
...
>>> comp.add_node('result_node', foo)
>>> comp.insert('input_node', 1)
>>> comp.compute_all()
>>> comp.v.result_node
2
```

We can explicitly specify the mapping from parameter names to node names if we require, using the `kwds` parameter. And a node can depend on more than one input node. Here we have a function of two parameters. The argument to `kwds` can be read as saying "Parameter **a** comes from node **x**, parameter **b** comes from node **y**":

```
>>> comp = Computation()
>>> comp.add_node('x')
>>> comp.add_node('y')
>>> def add(a, b):
...   return a + b
...
>>> comp.add_node('result', add, kwds={'a': 'x', 'b': 'y'})
>>> comp.insert('x', 20)
>>> comp.insert('y', 22)
>>> comp.compute_all()
>>> comp.v.result
42
```

For input nodes, the `add_node` method can optionally take a value, rather than having to separately call the insert method:

```
>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', lambda a: a + 1)
>>> comp.compute_all()
>>> comp.v.result
2
```

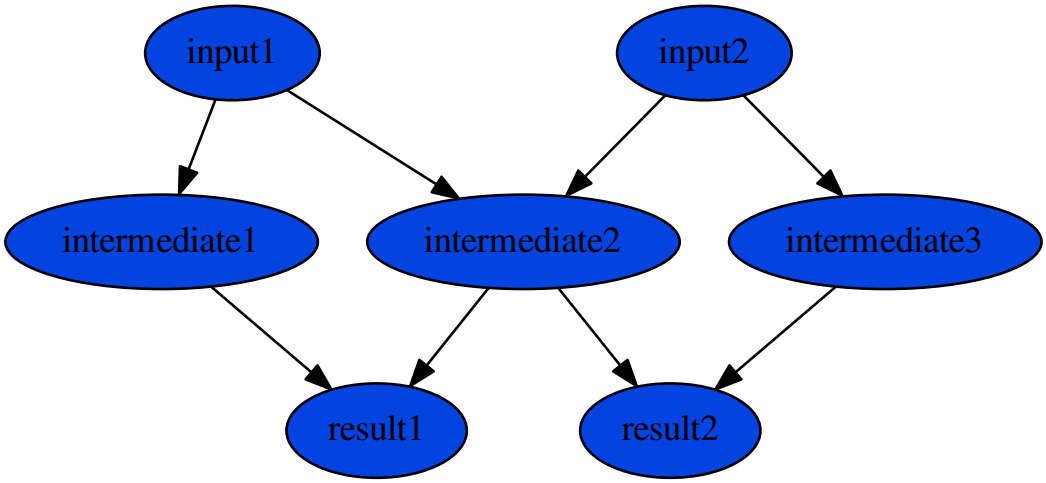
Finally, the function supplied to **add_node** can have `*args` or `**kwargs` arguments. When this is done, the `args` and `kwds` provided to **add_node** control what will be placed in `*args` or `**kwargs`:

```
>>> comp = Computation()
>>> comp.add_node('x', value=1)
>>> comp.add_node('y', value=2)
>>> comp.add_node('z', value=3)
>>> comp.add_node('args', lambda *args: args, args=['x', 'y', 'z'])
>>> comp.add_node('kwargs', lambda **kwargs: kwargs, kwds={'a': 'x', 'b': 'y', 'c': 'z
↪'})
>>> comp.compute_all()
>>> comp.v.args
(1, 2, 3)
>>> comp.v.kwargs
{'a': 1, 'b': 2, 'c': 3}
```

Controlling Computation

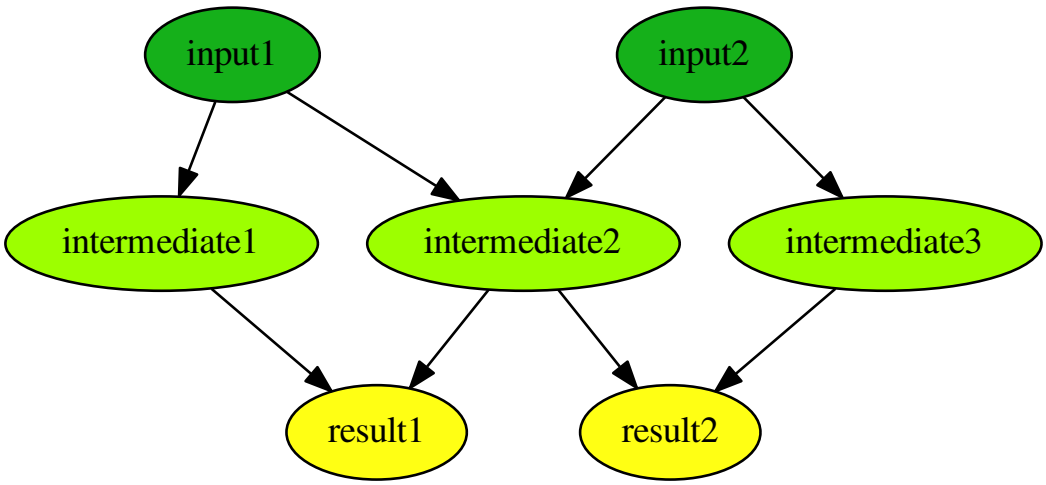
For these examples, we define a more complex Computation:

```
>>> comp = Computation()
>>> comp.add_node('input1')
>>> comp.add_node('input2')
>>> comp.add_node('intermediate1', lambda input1: 2 * input1)
>>> comp.add_node('intermediate2', lambda input1, input2: input1 + input2)
>>> comp.add_node('intermediate3', lambda input2: 3 * input2)
>>> comp.add_node('result1', lambda intermediate1, intermediate2: intermediate1 +␣
↪intermediate2)
>>> comp.add_node('result2', lambda intermediate2, intermediate3: intermediate2 +␣
↪intermediate3)
>>> comp.draw()
```



We insert values into **input1** and **input2**:

```
>>> comp.insert('input1, 1)
>>> comp.insert('input2', 2)
>>> comp.draw()
```

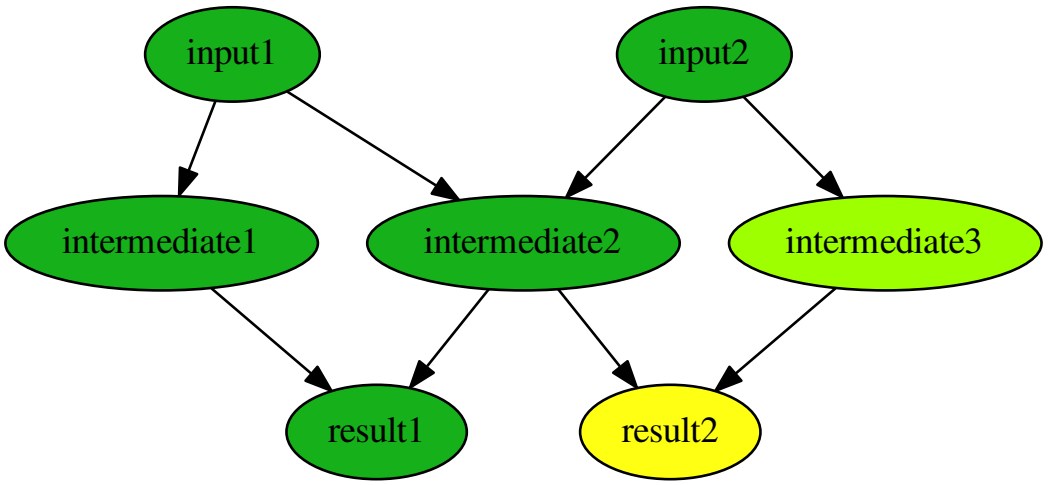


As before, we see that the nodes we have just inserted data for are colored dark green, indicating they are up-to-date. The intermediate nodes are all colored light green, to indicate that they are computable - that is that their immediate upstream nodes are all up-to-date, and so any one of them can be immediately calculated. The result nodes are colored yellow. This means that they are stale - they are not up-to-date, and they cannot be immediately calculated without

first calculating some nodes that they depend on.

We saw before that we can use the `compute_all` method to calculate nodes. We can also specify exactly which nodes we would like calculated using the `compute` method. This method will calculate any upstream dependencies that are not up-to-date, but it will not calculate nodes that do not need to be calculated. For example, if we request the **result1** be calculated, **intermediate1** and **intermedate2** will be calculated first, but **intermediate3** and **result2** will not be calculated:

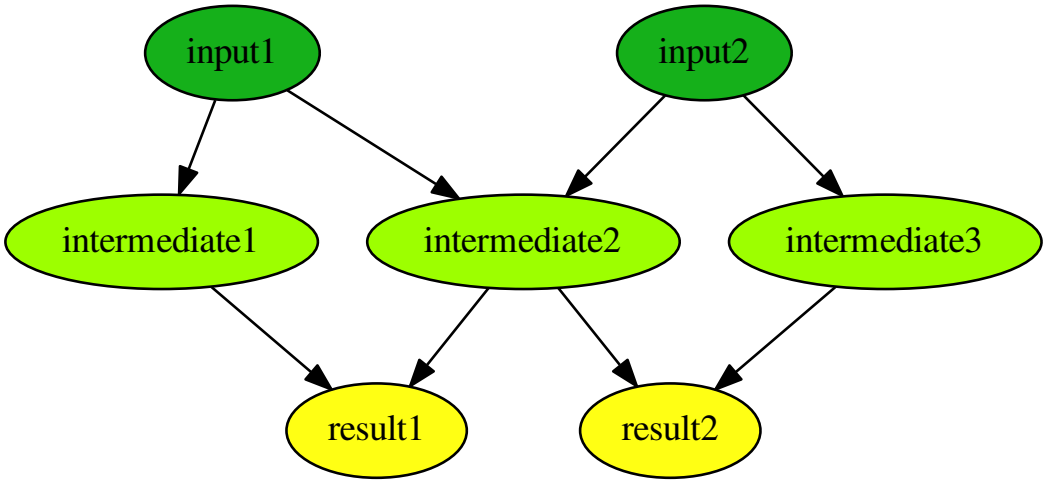
```
>>> comp.compute('result1')
>>> comp.v.result1
5
>>> comp.draw()
```



Inserting new data

Often, in real-time systems, updates will come periodically for one or more of the inputs to a computation. We can insert this updated data into a computation and Loman will corresponding mark any downstream nodes as stale or computable i.e. no longer up-to-date. Continuing from the previous example, we insert a new value into **input1**:

```
>>> comp.insert('input1', 2)
>>> comp.draw()
```



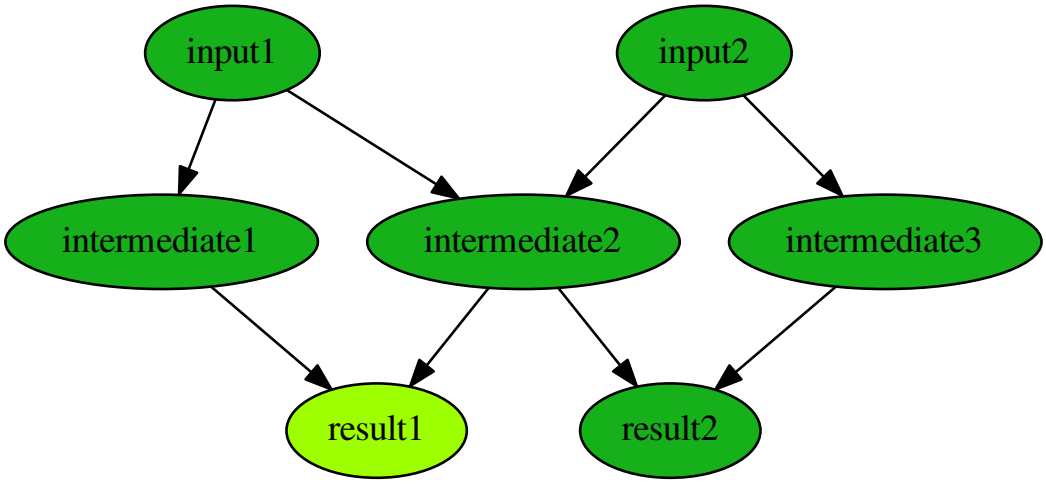
And again we can ask Loman to calculate nodes in the computation, and give us results. Here we calculate all nodes:

```
>>> comp.compute_all()
>>> comp.v.result1
8
```

Overriding calculation nodes

In fact, we are not restricted to inserting data into input nodes. It is perfectly possible to use the `insert` method to override the value of a calculated node also. The overridden value will remain in place until the node is recalculated (which will happen after one of its upstreams is updated causing it to be marked stale, or when it is explicitly marked as stale, and then recalculated). Here we override **intermediate2** and calculate **result2** (note that **result1** is not recalculated, because we didn't ask anything that required it to be):

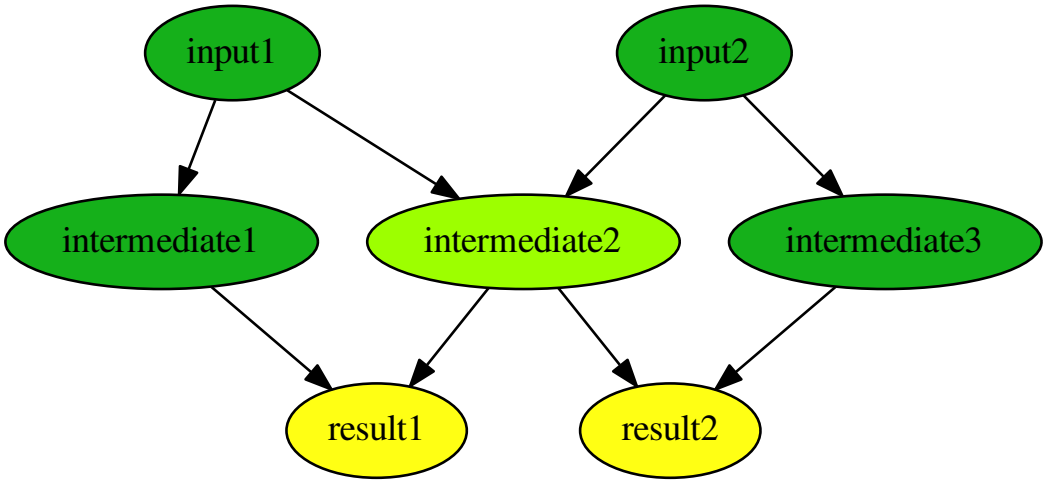
```
>>> comp.insert('intermediate2', 100)
>>> comp.compute('result2')
>>> comp.v.result2
106
>>> comp.draw()
```



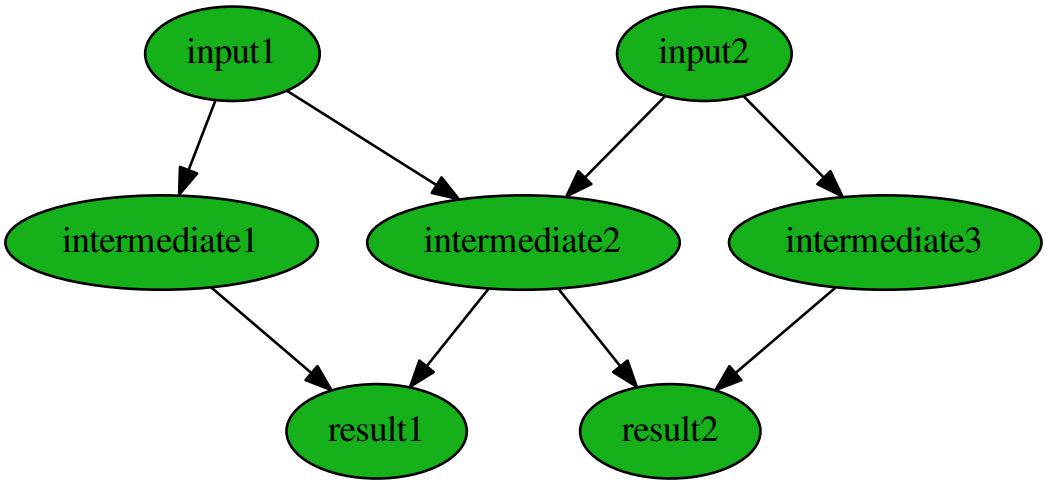
Changing calculations

As well as inserting data into nodes, we can update the computation they perform by re-adding the node. Node states get updated appropriately automatically. For example, continuing from the previous example, we can change how **intermediate2** is calculated, and we see that nodes **intermediate2**, **result1** and **result2** are no longer marked up-to-date:

```
>>> comp.add_node('intermediate2', lambda input1, input2: 5 * input1 + 2 * input2)
>>> comp.draw()
```




```
>>> comp.compute_all()
>>> comp.draw()
```

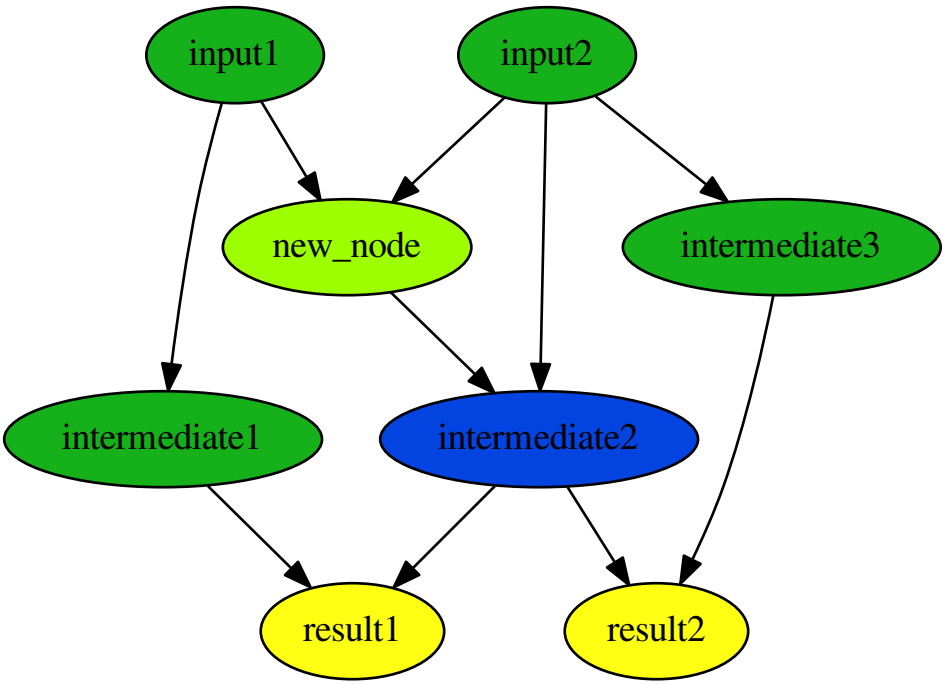



```
>>> comp.v.result1
18
>>> comp.v.result2
20
```

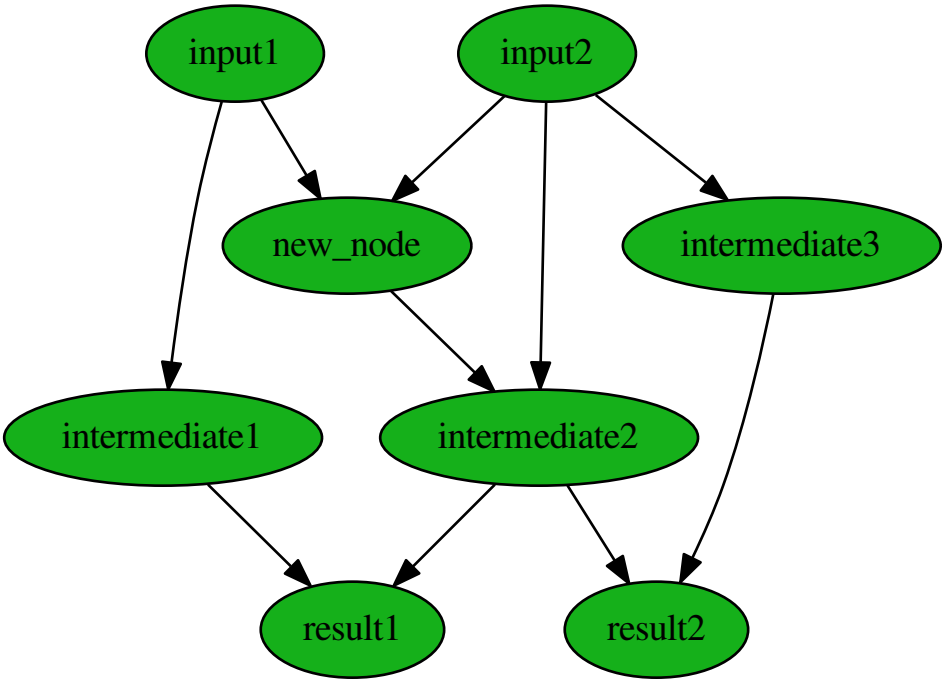
Adding new nodes

We can even add new nodes, and change the dependencies of existing calculations. So for example, we can create a new node called **new_node**, and have **intermediate2** depend on that, rather than **input1**. It's confusing when I describe it with words, but Loman's visualization helps us keep tabs on everything - that's its purpose:

```
>>> comp.add_node('new_node', lambda input1, input2: input1 / input2)
>>> comp.add_node('intermediate2', lambda new_nod, input2: 5 * new_nod + 2 * input2)
>>> comp.draw()
```




```
>>> comp.compute_all()
>>> comp.draw()
```

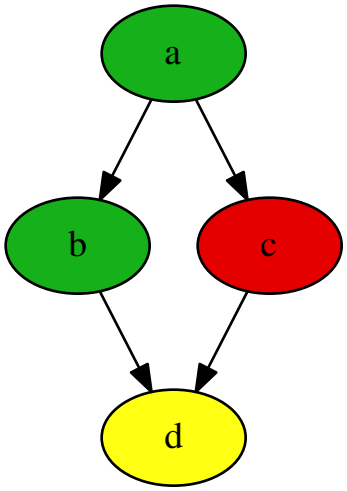



```
>>> comp.v.result1
13.0
>>> comp.v.result2
15.0
```

Error-handling

If trying to calculate a node causes an exception, then Loman will mark its state as error. Loman will also retain the exception and the stacktrace that caused the exception, which can be useful in large codebases. Downstream nodes cannot be calculated of course, but any other nodes that could be calculated will be. This allows us to discover multiple errors at once, avoiding the frustration of lenthgy-run-discover-next-error cycles:

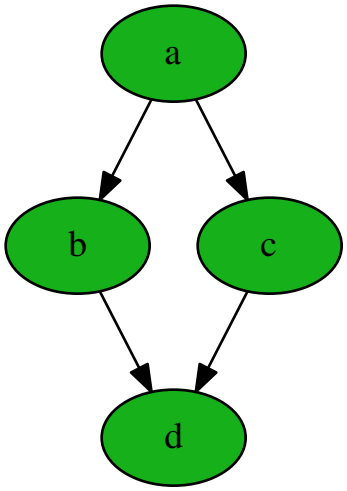
```
>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', lambda a: a + 1)
>>> comp.add_node('c', lambda a: a / 0) # This will cause an exception
>>> comp.add_node('d', lambda b, c: b + c)
>>> comp.compute_all()
>>> comp.draw()
```




```
>>> comp.s.c
<States.ERROR: 5>
>>> comp.v.c.exception
ZeroDivisionError('division by zero')
>>> print(comp.v.c.traceback)
Traceback (most recent call last):
  File "C:\ProgramData\Anaconda3\lib\site-packages\loman\computeengine.py", line 211,␣
↪in _compute_node
  File "<ipython-input-79-028365426246>", line 4, in <lambda>
    comp.add_node('c', lambda a: a / 0) # This will cause an exception
ZeroDivisionError: division by zero
```

We can use Loman's facilities of changing calculations or overriding values to quickly correct errors in-place, and without having to recompute upstreams, or wait to redownload large data-sets:

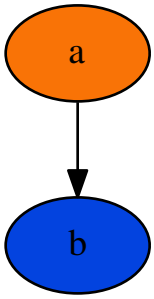
```
>>> comp.add_node('c', lambda a: a / 1)
>>> comp.compute_all()
>>> comp.draw()
```



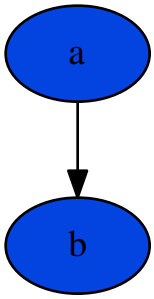
Missing upstream nodes

Loman has a special state, "Placeholder" for missing upstream nodes. This can occur when a node depends on a node that was not created, or when an existing node was deleted, which can be done with the `delete_node` method:

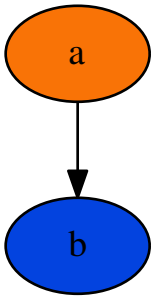
```
>>> comp = Computation()
>>> comp.add_node('b', lambda a: a)
>>> comp.draw()
```




```
>>> comp.s.a
<States.PLACEHOLDER: 0>
>>> comp.add_node('a')
>>> comp.draw()
```




```
>> comp.delete_node('a')
```



A final word

This quickstart is intended to help you understand how to create computations using Loman, how to update inputs, correct errors, and how to control the execution of your computations. The examples here are deliberately contrived to emphasize the dependency structures that Loman lets you create. The actual calculations performed are deliberately simplified for ease of exposition. In reality, nodes are likely to be complex objects, such as Numpy arrays, Pandas DataFrames, or classes you create, and calculation functions are likely to be longer than one line. In fact, we recommend that Loman nodes are fairly coarse grained - you should have a node for each intermediate value in a calculation that you might care to inspect or overide, but not one for each line of sequential program.

For more recommendations on how to use Loman in various contexts, you are invited to read the next section, *Strategies for using Loman in the Real World*.

Advanced Features

Constant Values

When you are using a pre-existing function for a node, and one or more of the parameters takes a constant value, one way is to define a lambda, which fixes the parameter value. For example, below we use a lambda to fix the second parameter passed to the add function:

```
>>> def add(x, y):
...     return x + y

>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', lambda a: add(a, 1))
>>> comp.compute_all()
>>> comp.v.b
2
```

However providing `ConstantValue` objects to the `args` or `kwds` parameters of `add_node`, make this simpler. `C` is an alias for `ConstantValue`, and in the example below, we use that to tell node `b` to calculate by taking parameter `x` from node `a`, and `y` as a constant, `1`:

```
>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', add, kwds={"x": "a", "y": C(1)})
>>> comp.compute_all()
>>> comp.v.b
2
```

Interactive Debugging

As shown in the quickstart section "Error-handling", loman makes it easy to see a traceback for any exceptions that are shown while calculating nodes, and also makes it easy to update calculation functions in-place to fix errors. However, it is often desirable to use Python's interactive debugger at the exact time that an error occurs. To support this, the `calculate` method takes a parameter `raise_exceptions`. When it is `False` (the default), nodes are set to state ERROR when exceptions occur during their calculation. When it is set to `True` any exceptions are not caught, allowing the user to invoke the interactive debugger

```
comp = Computation()
comp.add_node('numerator', value=1)
comp.add_node('divisor', value=0)
comp.add_node('result', lambda numerator, divisor: numerator / divisor)
comp.compute('result', raise_exceptions=True)
```

```
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-38-4243c7243fc5> in <module>()
----> 1 comp.compute('result', raise_exceptions=True)


[... skipped ...]
```

```
<ipython-input-36-c0efbf5b74f7> in <lambda>(numerator, divisor)
      3 comp.add_node('numerator', value=1)
      4 comp.add_node('divisor', value=0)
----> 5 comp.add_node('result', lambda numerator, divisor: numerator / divisor)


ZeroDivisionError: division by zero
```

```
%debug
```

```
> <ipython-input-36-c0efbf5b74f7>(5)<lambda>()
      1 from loman import *
      2 comp = Computation()
      3 comp.add_node('numerator', value=1)
      4 comp.add_node('divisor', value=0)
----> 5 comp.add_node('result', lambda numerator, divisor: numerator / divisor)

ipdb> p numerator
1
ipdb> p divisor
0
```

Creating Nodes Using a Decorator

Loman provide a decorator `@node`, which allows functions to be added to computations. The first parameter is the Computation object to add a node to. By default, it will take the node name from the function, and the names of input nodes from the names of the parameter of the function, but any parameters provided are passed through to `add_node`, including name:

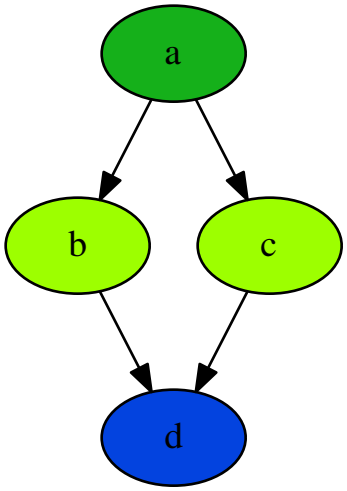
```
>>> from loman import *
>>> comp = Computation()
>>> comp.add_node('a', value=1)

>>> @node(comp)
... def b(a):
...     return a + 1

>>> @node(comp, 'c', args=['a'])
... def foo(x):
...     return 2 * x

>>> @node(comp, kwds={'x': 'a', 'y': 'b'})
... def d(x, y):
...     return x + y

>>> comp.draw()
```



Tagging Nodes

Nodes can be tagged with string tags, either when the node is added, using the `tags` parameter of `add_node`, or later, using the `set_tag` or `set_tags` methods, which can take a single node or a list of nodes:

```
>>> from loman import *
>>> comp = Computation()
>>> comp.add_node('a', value=1, tags=['foo'])
>>> comp.add_node('b', lambda a: a + 1)
>>> comp.set_tag(['a', 'b'], 'bar')
```

Note: Tags beginning and ending with double-underscores ("__[tag]__") are reserved for internal use by Loman.

The tags associated with a node can be inspected using the `tags` method, or the `t` attribute-style accessor:

```
>>> comp.tags('a')
{'__serialize__', 'bar', 'foo'}
>>> comp.t.b
{'__serialize__', 'bar'}
```

Tags can also be cleared with the `clear_tag` and `clear_tags` methods:

```
>>> comp.clear_tag(['a', 'b'], 'foo')
>>> comp.t.a
{'__serialize__', 'bar'}
```

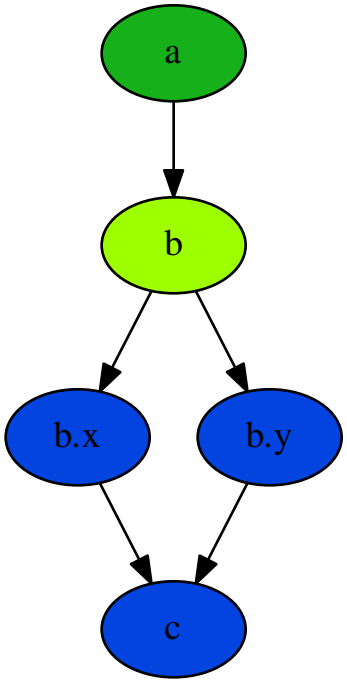
By design, no error is thrown if a tag is added to a node that already has that tag, nor if a tag is cleared from a node that does not have that tag.

In future, it is intended it will be possible to control graph drawing and calculation using tags (for example, by requesting that only nodes with or without certain tags are rendered or calculated).

Automatically expanding named tuples

Often, a calculation will return more than one result. For example, a numerical solver may return the best solution it found, along with a status indicating whether the solver converged. Python introduced namedtuples in version 2.6. A namedtuple is a tuple-like object where each element can be accessed by name, as well as by position. If a node will always contain a given type of namedtuple, Loman has a convenience method `add_named_tuple_expansion` which will create new nodes for each element of a namedtuple, using the naming convention **parent_node.tuple_element_name**. This can be useful for clarity when different downstream nodes depend on different parts of computation result:

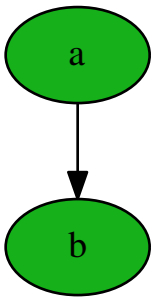
```
>>> Coordinate = namedtuple('Coordinate', ['x', 'y'])
>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', lambda a: Coordinate(a+1, a+2))
>>> comp.add_named_tuple_expansion('b', Coordinate)
>>> comp.add_node('c', lambda *args: sum(args), args=['b.x', 'b.y'])
>>> comp.compute_all()
>>> comp.get_value_dict()
{'a': 1, 'b': Coordinate(x=2, y=3), 'b.x': 2, 'b.y': 3, 'c': 5}
>>> comp.draw()
```



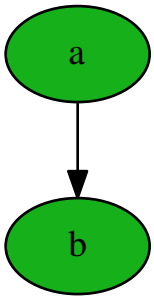
Serializing computations

Loman can serialize computations to disk using the dill package. This can be useful to have a system store the inputs, intermediates and results of a scheduled calculation for later inspection if required:

```
>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', lambda a: a + 1)
>>> comp.compute_all()
>>> comp.draw()
```




```
>>> comp.get_value_dict()
{'a': 1, 'b': 2}
>>> comp.write_dill('foo.dill')
>>> comp2 = Computation.read_dill('foo.dill')
>>> comp2.draw()
```

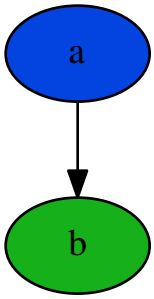



```
>>> comp.get_value_dict()
{'a': 1, 'b': 2}
```

It is also possible to request that a particular node not be serialized, in which case it will have no value, and uninitialized state when it is deserialized. This can be useful where an object is not serializable, or where data is not licensed to be distributed:

```
>>> comp.add_node('a', value=1, serialize=False)
>>> comp.compute_all()
```

```
>>> comp.write_dill('foo.dill')
>>> comp2 = Computation.read_dill('foo.dill')
>>> comp2.draw()
```

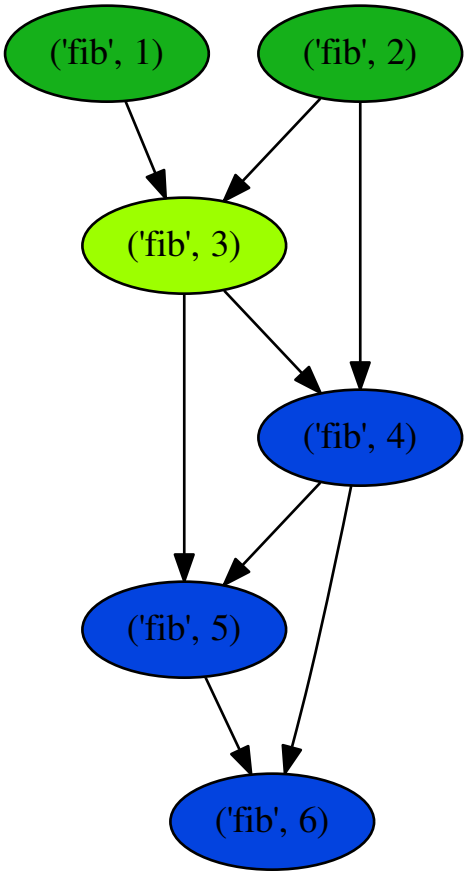


Note: The serialization format is not currently stabilized. While it is convenient to be able to inspect the results of previous calculations, this method should *not* be relied on for long-term storage.

Non-string node names

In the previous example, the nodes have all been given strings as keys. This is not a requirement, and in fact any object that could be used as a key in a dictionary can be a key for a node. As function parameters can only be strings, we have to rely on the `kwds` argument to `add_node` to specify which nodes should be used as inputs for calculation nodes' functions. For a simple but frivolous example, we can represent a finite part of the Fibonacci sequence using tuples of the form `('fib', [int])` as keys:

```
>>> comp = Computation()
>>> comp.add_node(('fib', 1), value=1)
>>> comp.add_node(('fib', 2), value=1)
>>> for i in range(3,7):
...     comp.add_node(('fib', i), lambda x, y: x + y, kwds={'x': ('fib', i - 1), 'y': (
↪'fib', i - 2)})
...
>>> comp.draw()
```




```
>>> comp.compute_all()
>>> comp.value(('fib', 6))
8
```

Strategies for using Loman in the Real World

Fine-grained or Coarse-grained nodes

When using Loman, we have a choice of whether we make each expression in our program a node (very fine-grained), or have one node which executes all the code in our calculation (very coarse-grained) or somewhere in between. Loman is relatively efficient at executing code, but it can never be as efficient as the Python interpreter sequentially running lines of Python. Accordingly, we recommend that you should create a node for each input, result or intermediate value that you might care to inspect, alter, or calculate in a different way.

On the other hand, there is no cost to nodes that are not executed[1], and execution is effectively lazy if you specify

[1] This is not quite true. The method for working out computable nodes has not been optimized, so in fact there is linear cost to adding unused nodes, but this limitation is unnecessary and will be removed in due course.

which nodes you wish to calculate. With this in mind, it can make sense to create large numbers of nodes that import data for example, in the anticipation that it will be useful to have that data to hand at some point, but there is no cost if it is not needed.

Converting existing codebases to use Loman

We typically find that production code tends to have a "main" function which loads data from databases and other systems, coordinates running a few calculations, and then loads the results back into other systems. We have had good experiences transferring the data downloading and calculation parts to Loman.

Typically such a "main" function will have small groups of lines responsible for grabbing particular pieces of input data, or for calling out to specific calculation routine. Each of these groups can be converted to a node in a Loman computation easily. Often, Loman's `kwds` input to `add_node` is useful to martial data into existing functions.

It is often helpful to put the creation of a Loman computation object with uninitialized values into a separate function. Then it is easy to experiment with the computation in an interactive environment such as Jupyter.

The final result is that the "main" function will instantiate a computation object, give it objects for database access, and other inputs, such as run date. Exporting calculated results is not within Loman's scope, so the "main" function will coordinate writing results from the computation object to the same systems as before. It is also useful if the "main" function serializes the computation for later inspection if necessary.

Already, having a concrete visualization of the computation's structure, as well as the ability to access intermediates of the computation through the serialized copy will be great steps ahead of the existing system. Experimenting with adding additional parts to the computation will also be easier, as blank or serialized computations can be used as the basis for this work in an interactive environment.

Finally, it is not necessary to "big bang" existing systems over to a Loman-based solution. Instead, small discrete parts of an existing implementation can be converted, and gradually migrate additional parts inside of the Loman computation as desirable.

Accessing databases and other external systems

To access databases, we recommend SQLAlchemy Core. For each database, we recommend creating two nodes in a computation, one for the engine, and another for the metadata object, and these nodes should not be serialized. Then every data access can use the nodes **engine** and **metadata** as necessary. This is not dissimilar to dependency injection:

```
>>> import sqlalchemy as sa
>>> comp = Computation()
>>> comp.add_node('engine', sa.create_engine(...), serialize=False)
>>> comp.add_node('metadata', lambda engine: sa.MetaData(engine), serialize=False)
>>> def get_some_data(engine, ...):
...   [...]
...
>>> comp.add_node('some_data', get_some_data)
```

Accessing other data sources such as scraped websites, or vendor systems can be accessed similarly. For example, here is code to create a logged in browser under the control of Selenium to scrape data from a website:

```
>>> from selenium import webdriver
>>> comp = Computation()
>>> def get_logged_in_browser():
...   browser = webdriver.Chrome()
...   browser.get('http://somewebsite.com')
...   elem = browser.find_element_by_id('userid')
...   elem.send_keys('user@id.com')
```

```
...     elem = browser.find_element_by_id('password')
...     elem.send_keys('secret')
...     elem = browser.find_element_by_name('_submit')
...     elem.click()
...     return browser
... comp.add_node('browser', get_logged_in_browser)
```

CHAPTER 2

API Reference

API Reference

class loman.**States**

Possible states for a computation node

COMPUTABLE = 3

ERROR = 5

PLACEHOLDER = 0

STALE = 2

UNINITIALIZED = 1

UPTODATE = 4

class loman.**Computation**

add_map_node(*result_node*, *input_node*, *subgraph*, *subgraph_input_node*, *subgraph_output_node*)

Apply a graph to each element of iterable

In turn, each element in the `input_node` of this graph will be inserted in turn into the subgraph's `subgraph_input_node`, then the subgraph's `subgraph_output_node` calculated. The resultant list, with an element or each element in `input_node`, will be inserted into `result_node` of this graph. In this way `add_map_node` is similar to `map` in functional programming.

Parameters

- **result_node** – The node to place a list of results in **this** graph
- **input_node** – The node to get a list input values from **this** graph
- **subgraph** – The graph to use to perform calculation for each element
- **subgraph_input_node** – The node in **subgraph** to insert each element in turn
- **subgraph_output_node** – The node in **subgraph** to read the result for each element

add_named_tuple_expansion(*name*, *namedtuple_type*, *group=None*)
Automatically add nodes to extract each element of a named tuple type

It is often convenient for a calculation to return multiple values, and it is polite to do this a namedtuple rather than a regular tuple, so that later users have same name to identify elements of the tuple. It can also help make a computation clearer if a downstream computation depends on one element of such a tuple, rather than the entire tuple. This does not affect the computation per se, but it does make the intention clearer.

To avoid having to create many boiler-plate node definitions to expand namedtuples, the `add_named_tuple_expansion` method automatically creates new nodes for each element of a tuple. The convention is that an element called 'element', in a node called 'node' will be expanded into a new node called 'node.element', and that this will be applied for each element.

Example:

```
>>> from collections import namedtuple
>>> Coordinate = namedtuple('Coordinate', ['x', 'y'])
>>> comp = Computation()
>>> comp.add_node('c', value=Coordinate(1, 2))
>>> comp.add_named_tuple_expansion('c', Coordinate)
>>> comp.compute_all()
>>> comp.value('c.x')
1
>>> comp.value('c.y')
2
```

Parameters

- **name** – Node to cera
- **namedtuple_type** (*namedtuple class*) – Expected type of the node

add_node(*name*, *func=None*, ***kwargs*)
Adds or updates a node in a computation

Parameters

- **name** – Name of the node to add. This may be any hashable object.
- **func** (*Function, default None*) – Function to use to calculate the node if the node is a calculation node. By default, the input nodes to the function will be implied from the names of the function parameters. For example, a parameter called `a` would be taken from the node called `a`. This can be modified with the `kwds` parameter.
- **args** (*List, default None*) – Specifies a list of nodes that will be used to populate arguments of the function positionally for a calculation node. e.g. If args is `['a', 'b', 'c']` then the function would be called with three parameters, taken from the nodes 'a', 'b' and 'c' respectively.
- **kwds** (*Dictionary, default None*) – Specifies a mapping from parameter name to the node that should be used to populate that parameter when calling the function for a calculation node. e.g. If args is `{'x': 'a', 'y': 'b'}` then the function would be called with parameters named 'x' and 'y', and their values would be taken from nodes 'a' and 'b' respectively. Each entry in the dictionary can be read as "take parameter [key] from node [value]".
- **value** (*default None*) – If given, the value is inserted into the node, and the node state set to UPTODATE.

- **serialize** (*boolean, default True*) – Whether the node should be serialized. Some objects cannot be serialized, in which case, set serialize to False
- **inspect** (*boolean, default True*) – Whether to use introspection to determine the arguments of the function, which can be slow. If this is not set, kwds and args must be set for the function to obtain parameters.
- **group** (*default None*) – Subgraph to render node in
- **tags** (*Iterable*) – Set of tags to apply to node

clear_tag(*name*, *tag*)
Clear tag on a node or nodes. Ignored if tags are not set.

Parameters

- **name** – Node or nodes to clear tags for
- **tag** – Tag to clear

clear_tags(*name*, *tags*)
Clear tags on a node or nodes. Ignored if tags are not set.

Parameters

- **name** – Node or nodes to clear tags for
- **tags** – Tags to clear

compute(*name*, *raise_exceptions=False*)
Compute a node and all necessary predecessors

Following the computation, if successful, the target node, and all necessary ancestors that were not already UPTODATE will have been calculated and set to UPTODATE. Any node that did not need to be calculated will not have been recalculated.

If any nodes raises an exception, then the state of that node will be set to ERROR, and its value set to an object containing the exception object, as well as a traceback. This will not halt the computation, which will proceed as far as it can, until no more nodes that would be required to calculate the target are COMPUTABLE.

Parameters

- **name** – Name of the node to compute
- **raise_exceptions** (*Boolean, default False*) – Whether to pass exceptions raised by node computations back to the caller

compute_all(*raise_exceptions=False*)
Compute all nodes of a computation that can be computed

Nodes that are already UPTODATE will not be recalculated. Following the computation, if successful, all nodes will have state UPTODATE, except UNINITIALIZED input nodes and PLACEHOLDER nodes.

If any nodes raises an exception, then the state of that node will be set to ERROR, and its value set to an object containing the exception object, as well as a traceback. This will not halt the computation, which will proceed as far as it can, until no more nodes are COMPUTABLE.

Parameters **raise_exceptions** (*Boolean, default False*) – Whether to pass exceptions raised by node computations back to the caller

copy()
Create a copy of a computation

The copy is shallow. Any values in the new Computation's DAG will be the same object as this Computation's DAG. As new objects will be created by any further computations, this should not be an issue.

Return type *Computation*

delete_node(*name*)
Delete a node from a computation

When nodes are explicitly deleted with `delete_node`, but are still depended on by other nodes, then they will be set to PLACEHOLDER status. In this case, if the nodes that depend on a PLACEHOLDER node are deleted, then the PLACEHOLDER node will also be deleted.

Parameters **name** – Name of the node to delete. If the node does not exist, a `NonExistentNodeException` will be raised.

draw(*graph_attr=None*, *node_attr=None*, *edge_attr=None*, *show_expansion=False*)
Draw a computation's current state using the GraphViz utility

Parameters

- **graph_attr** – Mapping of (attribute, value) pairs for the graph. For example `graph_attr={'size': '10,8'}` can control the size of the output graph
- **node_attr** – Mapping of (attribute, value) pairs set for all nodes.
- **edge_attr** – Mapping of (attribute, value) pairs set for all edges.
- **show_expansion** – Whether to show expansion nodes (i.e. named tuple expansion nodes) if they are not referenced by other nodes

get_inputs(*name*)
Get a list of the inputs for a node or set of nodes

Parameters **name** – Name or names of nodes to get inputs for

Returns If name is scalar, return a list of upstream nodes used as input. If name is a list, return a list of list of inputs.

insert(*name*, *value*)
Insert a value into a node of a computation

Following insertation, the node will have state UPTODATE, and all its descendents will be COMPUTABLE or STALE.

If an attempt is made to insert a value into a node that does not exist, a `NonExistentNodeException` will be raised.

Parameters

- **name** – Name of the node to add.
- **value** – The value to be inserted into the node.

insert_from(*other*, *nodes=None*)
Insert values into another Computation object into this Computation object

Parameters

- **other** – The computation object to take values from
- **nodes** (*List, default None*) – Only populate the nodes with the names provided in this list. By default, all nodes from the other Computation object that have corresponding nodes in this Computation object will be inserted

insert_many(*name_value_pairs*)
Insert values into many nodes of a computation simultaneously

Following insertation, the nodes will have state UPTODATE, and all their descendents will be COMPUTABLE or STALE. In the case of inserting many nodes, some of which are descendents of others, this ensures that the inserted nodes have correct status, rather than being set as STALE when their ancestors are inserted.

If an attempt is made to insert a value into a node that does not exist, a `NonExistentNodeException` will be raised, and none of the nodes will be inserted.

Parameters **name_value_pairs** (*List of tuples*) – Each tuple should be a pair (name, value), where name is the name of the node to insert the value into.

nodes()
Get a list of nodes in this computation :return: List of nodes

static **read_dill**(*file_*)
Deserialize a computation from a file or file-like object

Parameters **file** (*File-like object, or string*) – If string, writes to a file

set_stale(*name*)
Set the state of a node and all its dependencies to STALE

Parameters **name** – Name of the node to set as STALE.

set_tag(*name*, *tag*)
Set tags on a node or nodes. Ignored if tags are already set.

Parameters

- **name** – Node or nodes to set tag for
- **tag** – Tag to set

set_tags(*name*, *tags*)
Set tags on a node or nodes. Ignored if tags are already set.

Parameters

- **name** – Node or nodes to set tags for
- **tags** – Tags to set

state(*name*)
Get the state of a node

This can also be accessed using the attribute-style accessor `s` if `name` is a valid Python attribute name:

```
>>> comp = Computation()
>>> comp.add_node('foo', value=1)
>>> comp.state('foo')
<States.UPTODATE: 4>
>>> comp.s.foo
<States.UPTODATE: 4>
```

Parameters **name** (*Key or [Keys]*) – Name or names of the node to get state for

tags(*name*)
Get the tags associated with a node

```
>>> comp = Computation()
>>> comp.add_node('a', tags=['foo', 'bar'])
>>> comp.t.a
{'__serialize__', 'bar', 'foo'}
```

Parameters **name** –

Returns

to_df()

Get a dataframe containing the states and value of all nodes of computation

```
>>> comp = loman.Computation()
>>> comp.add_node('foo', value=1)
>>> comp.add_node('bar', value=2)
>>> comp.to_df()
                state  value  is_expansion
bar  States.UPTODATE      2           NaN
foo  States.UPTODATE      1           NaN
```

to_dict()

Get a dictionary containing the values of all nodes of a computation

```
>>> comp = loman.Computation()
>>> comp.add_node('foo', value=1)
>>> comp.add_node('bar', value=2)
>>> comp.to_dict()
{'bar': 2, 'foo': 1}
```

value(*name*)

Get the current value of a node

This can also be accessed using the attribute-style accessor `v` if `name` is a valid Python attribute name:

```
>>> comp = Computation()
>>> comp.add_node('foo', value=1)
>>> comp.value('foo')
1
>>> comp.v.foo
1
```

Parameters **name** (*Key or [Keys]*) – Name or names of the node to get the value of

write_dill(*file_*)

Serialize a computation to a file or file-like object

Parameters **file** (*File-like object, or string*) – If string, writes to a file

CHAPTER 3

Developer Guidelines

Release Checklist

- Check CHANGELOG is up-to-date
- Check Travis CI builds are passing
- Check Read The Docs documentation builds are passing
- Update version string in
  - setup.py
  - docs/conf.py
- Commit updated versions and tag
- Build the tar.gz and wheel: `python setup.py sdist bdist_wheel`
- Upload the tar.gz and wheel: `twine upload dist\loman-x.y.z*`
- Email the community

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index