Ioman Documentation

Release 0.3.0

Ed Parcell

May 25, 2023

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

1.1 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.

1.2 Installation Guide

1.2.1 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.

1.2.2 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')
```

1.3 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.

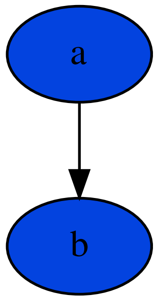
1.3.1 Creating and Running a Computation

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)
```

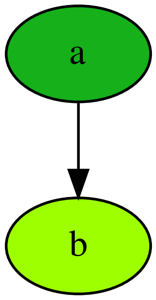
Loman's computations have a method `draw` which lets us easily see a visualization of the computation we just created:

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



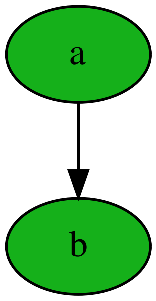
Loman gives us a quick and easy way to visualize our computations as a graph data structure. Each node of the graph is a colored oval, representing an input value or calculated value, and each edge (line) shows where the calculation of one node depends on another. The graph above shows us that node **b** 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.

1.3.2 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
```

1.3.3 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
```

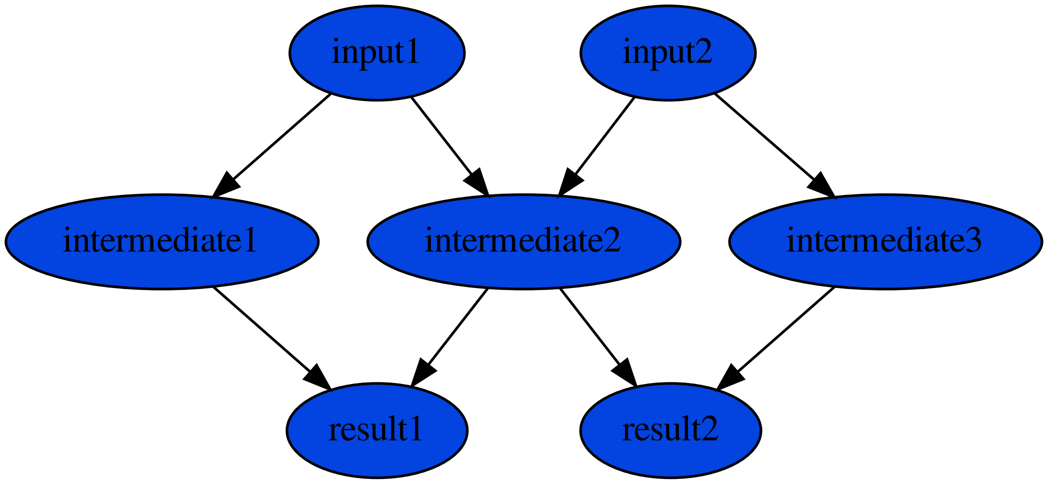
Finally, the function supplied to **add_node** can have `*args` or `**kwargs` arguments. When this is done, the `args` and `kwds` provided to **add_node** control what will be placed in `*args` or `**kwargs`:

```
>>> 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}
```

1.3.4 Controlling Computation

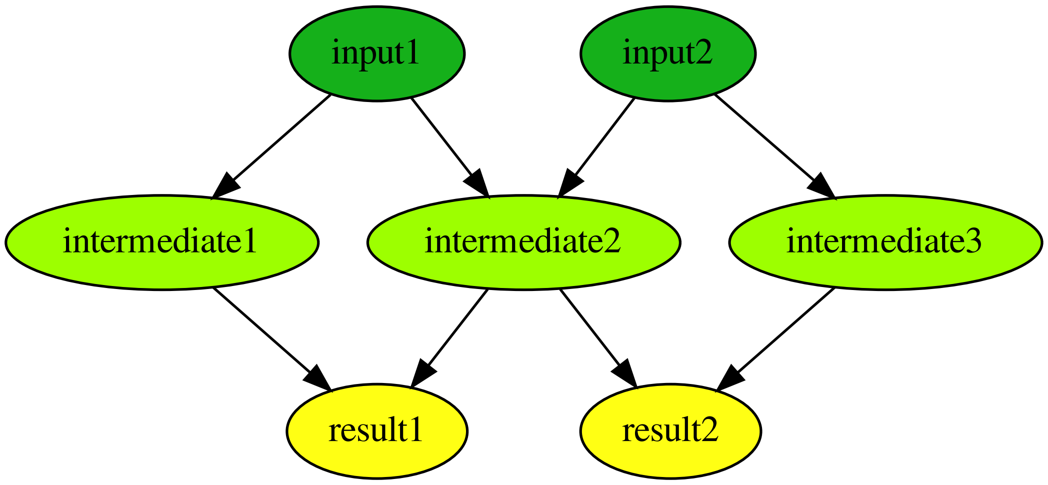
For these examples, we define a more complex Computation:

```
>>> 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**:

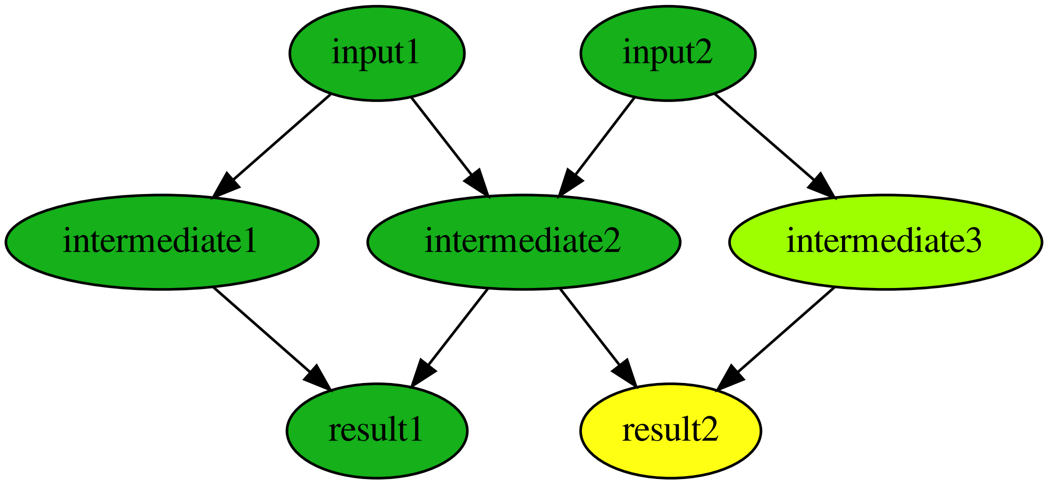
```
>>> comp.insert('input1, 1)
>>> comp.insert('input2', 2)
>>> comp.draw()
```



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:

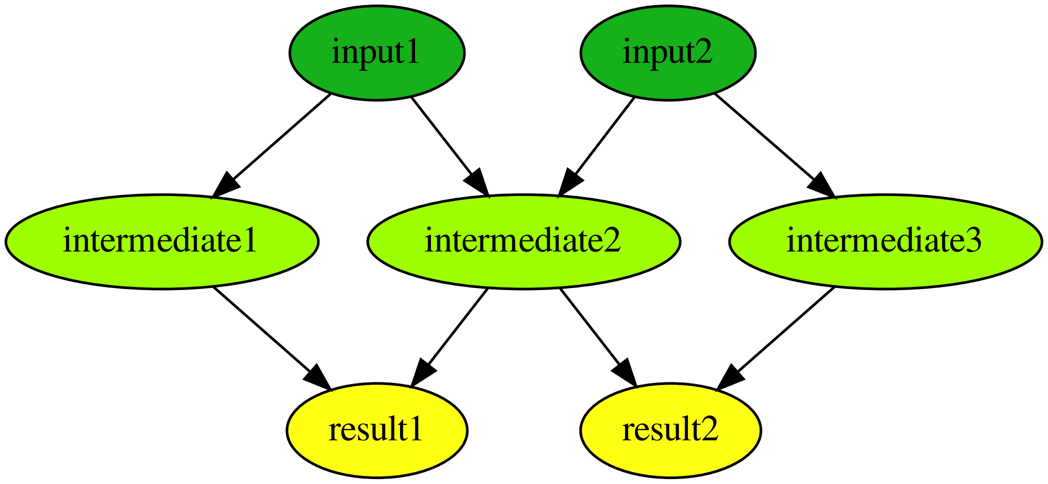
```
>>> comp.compute('result1')
>>> comp.v.result1
5
>>> comp.draw()
```



1.3.5 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()
```



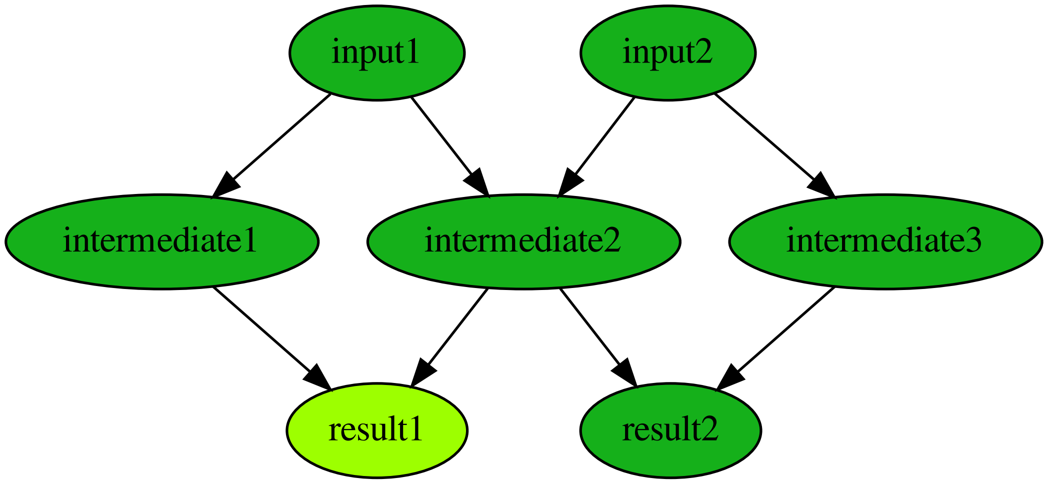
And again we can ask Loman to calculate nodes in the computation, and give us results. Here we calculate all nodes:

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

1.3.6 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):

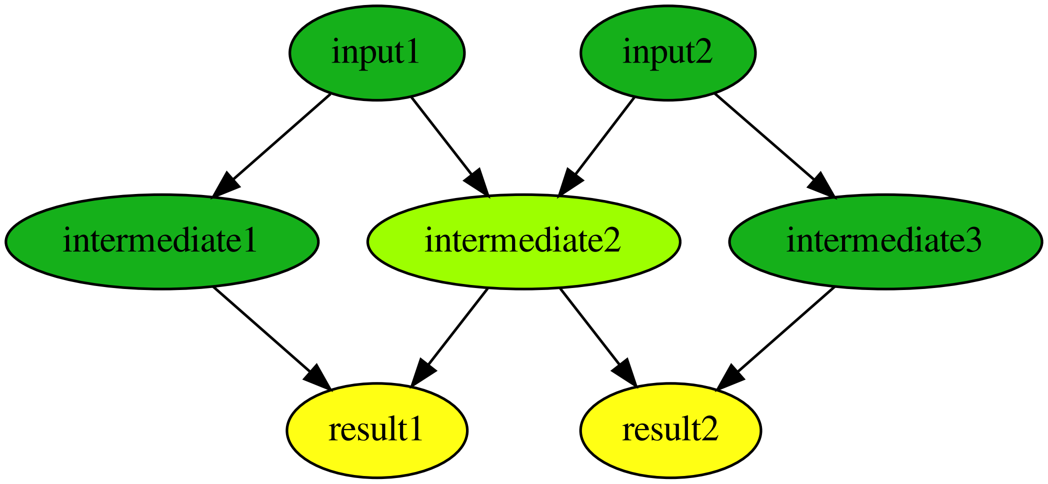
```
>>> comp.insert('intermediate2', 100)
>>> comp.compute('result2')
>>> comp.v.result2
106
>>> comp.draw()
```



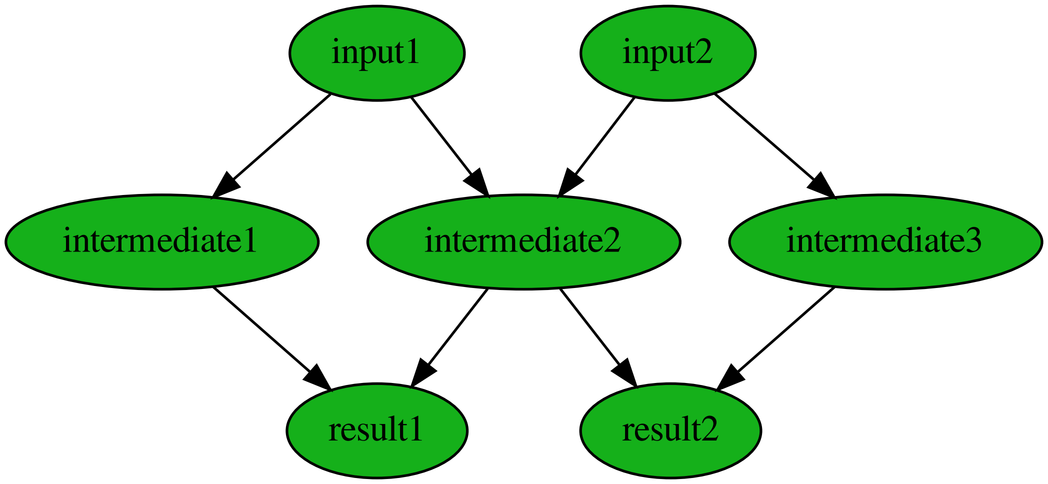
1.3.7 Changing calculations

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()
```

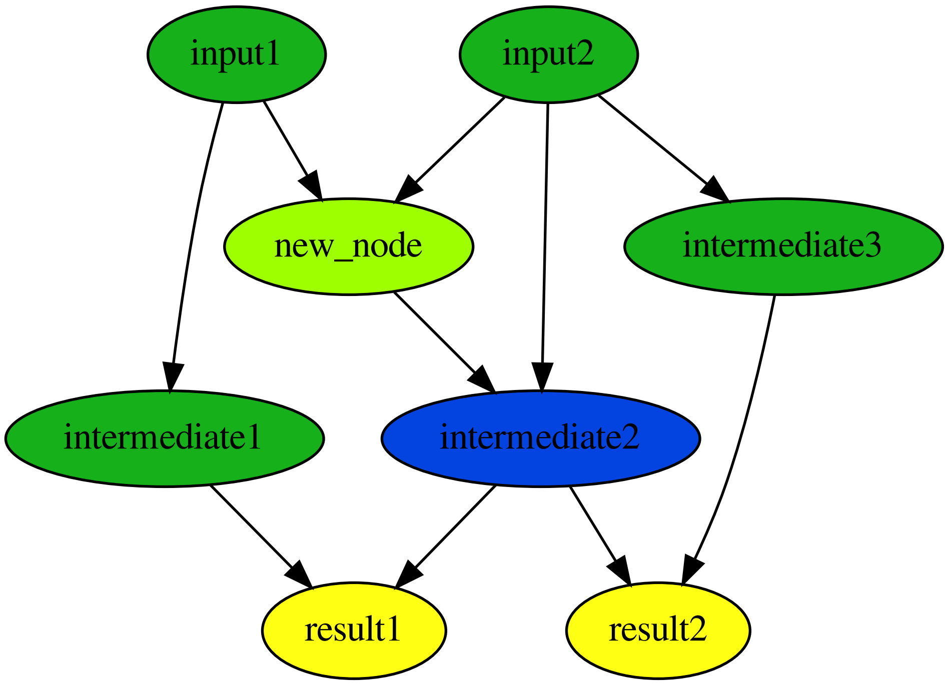


```
>>> comp.v.result1
18
>>> comp.v.result2
20
```

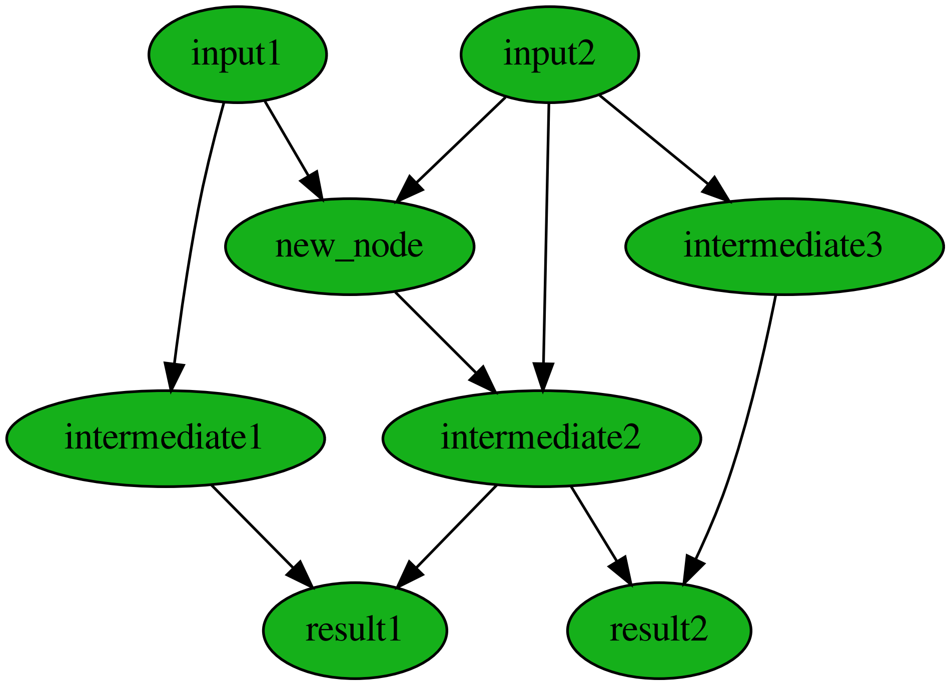
1.3.8 Adding new nodes

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()
```

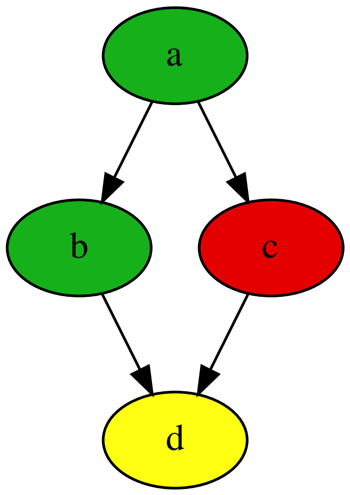


```
>>> comp.v.result1
13.0
>>> comp.v.result2
15.0
```

1.3.9 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:

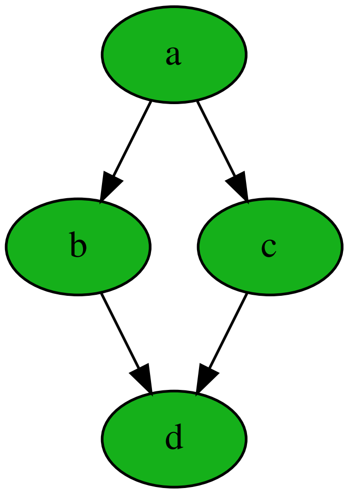
```
>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', lambda a: a + 1)
>>> comp.add_node('c', lambda a: a / 0) # This will cause an exception
>>> comp.add_node('d', lambda b, c: b + c)
>>> comp.compute_all()
>>> comp.draw()
```



```
>>> 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:

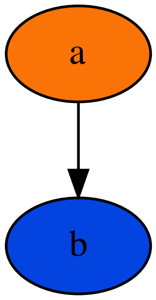
```
>>> comp.add_node('c', lambda a: a / 1)
>>> comp.compute_all()
>>> comp.draw()
```



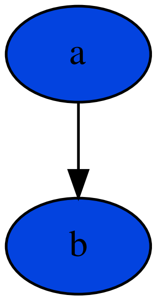
1.3.10 Missing upstream nodes

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:

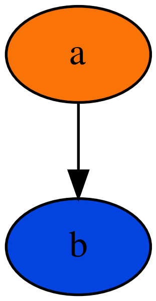
```
>>> comp = Computation()
>>> comp.add_node('b', lambda a: a)
>>> comp.draw()
```



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



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



1.3.11 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.

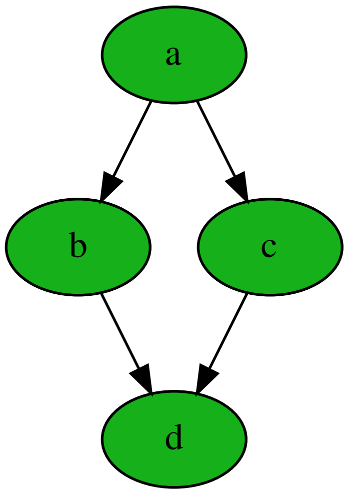
For more recommendations on how to use Loman in various contexts, you are invited to read the next section, *Strategies for using Loman in the Real World*.

1.4 Advanced Features

1.4.1 Viewing node inputs and outputs

Loman computations contain methods to see what nodes are inputs to any node, and what nodes a given node is itself an input to. First, let's define a simple Computation:

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



We can find the inputs of a node using the `get_inputs` method, or the `i` attribute (which works similarly to the `v` and `s` attributes to access value and state):

```
>>> comp.get_inputs('b')
['a']
>>> comp.get_inputs('d')
['c', 'b']
>>> comp.i.d # Attribute-style access
['c', 'b']
>>> comp.i['d'] # Dictionary-style access
['c', 'b']
>>> comp.i[['b', 'd']] # Multiple dictionary-style accesses:
[['a'], ['c', 'b]]
```

We can also use `get_original_inputs` to find the inputs of the entire Computation (or a subset of it):

```
>>> comp.get_original_inputs()
['a']
>>> comp.get_original_inputs(['b']) # Just the inputs used to compute b
['a']
```

To find what a node feeds into, there are `get_outputs`, the `o` attribute and `get_final_outputs` (although as intermediate nodes are often useful, this latter is less useful):

```
>>> comp.get_outputs('a')
['b', 'c']
>>> comp.o.a
['b', 'c']
>>> comp.o[['b', 'c']]
[['d'], ['d']]
>>> comp.get_final_outputs()
['d']
```

Finally, these can be used with the `v` accessor to quickly see all the input values to a given node:

```
>>> {n: comp.v[n] for n in comp.i.d}
{'c': 2, 'b': 2}
```

1.4.2 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
```

1.4.3 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
```

1.4.4 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)
```

(continues on next page)

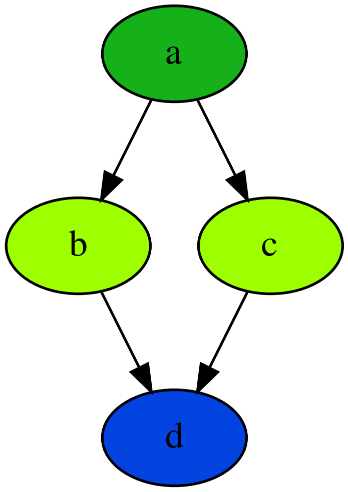
(continued from previous page)

```
... 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()
```



1.4.5 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'}
```

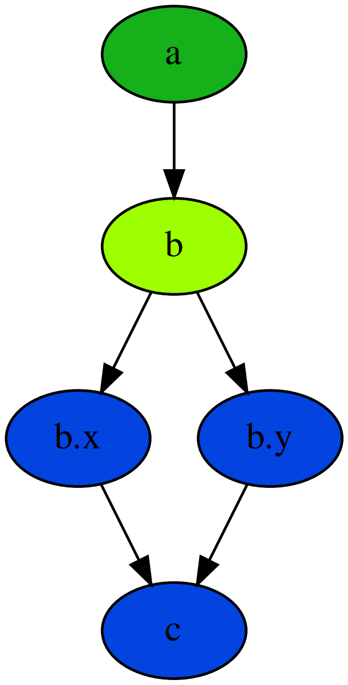
By design, no error is thrown if a tag is added to a node that already has that tag, nor if a tag is cleared from a node that does not have that tag.

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).

1.4.6 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:

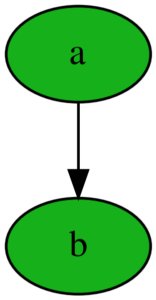
```
>>> Coordinate = namedtuple('Coordinate', ['x', 'y'])
>>> comp = Computation()
>>> comp.add_node('a', value=1)
>>> comp.add_node('b', lambda a: Coordinate(a+1, a+2))
>>> comp.add_named_tuple_expansion('b', Coordinate)
>>> comp.add_node('c', lambda *args: sum(args), args=['b.x', 'b.y'])
>>> comp.compute_all()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



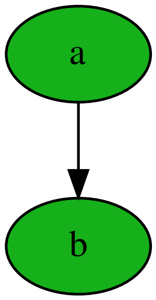
1.4.7 Serializing computations

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()
```



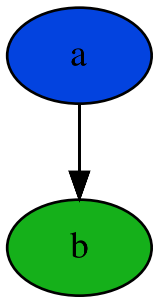
```
>>> comp.to_dict()
{'a': 1, 'b': 2}
>>> comp.write_dill('foo.dill')
>>> comp2 = Computation.read_dill('foo.dill')
>>> comp2.draw()
```



```
>>> 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()
```

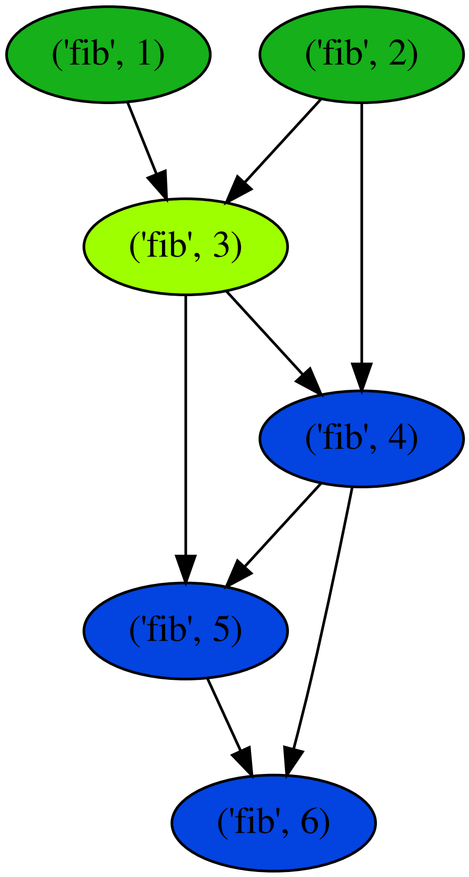


Note: The serialization format is not currently stabilized. While it is convenient to be able to inspect the results of previous calculations, this method should *not* be relied on for long-term storage.

1.4.8 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
```

1.4.9 Repointing Nodes

It is possible to repoint existing nodes to a new node. This can be useful when it is desired to make a small change in one node, without having to recreate all descendant nodes. As an example:

```
>>> from loman import *
>>> comp = Computation()
>>> comp.add_node('a', value = 2)
>>> comp.add_node('b', lambda a: a + 1)
>>> comp.add_node('c', lambda a: 10*a)
>>> comp.compute_all()
>>> comp.v.b
3
>>> comp.v.c
```

(continues on next page)

(continued from previous page)

```
20
>>> comp.add_node('modified_a', lambda a: a*a)
>>> comp.compute_all()
>>> comp.v.a
2
>>> comp.v.modified_a
4
>>> comp.v.b
3
>>> comp.v.c
20
>>> comp.repoint('a', 'modified_a')
>>> comp.compute_all()
>>> comp.v.b
5
>>> comp.v.c
40
```

1.5 Strategies for using Loman in the Real World

1.5.1 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 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.

1.5.2 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.

[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.

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.

1.5.3 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

2.1 API Reference

CHAPTER 3

Developer Guidelines

3.1 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

Index