graphtik Documentation

Release src: 3.0.0, git: v3.0.0

Yahoo Vision and Machine Learning Team: Huy Nguyen, Arel Cor

Dec 02, 2019

Contents

(src: 3.0.0, git: v3.0.0 , Dec 02, 2019)

It's a DAG all the way down!

CHAPTER 1

Lightweight computation graphs for Python

Graphtik is an an understandable and lightweight Python module for building and running ordered graphs of computations. The API posits a fair compromise between features and complexity, without precluding any. It can be used as is to build machine learning pipelines for data science projects. It should be extendable to act as the core for a custom ETL engine or a workflow-processor for interdependent files and processes.

Graphtik sprang from Graphkit to experiment with Python 3.6+ features.

1.1 Operations

At a high level, an operation is a node in a computation graph. Graphtik uses an *Operation* class to abstractly represent these computations. The class specifies the *requirments* for a function to participate in a computation graph; those are its input-data **needs**, and the output-data it **provides**.

The FunctionalOperation provides a lightweight wrapper around an arbitrary function to define those specifications.

class graphtik.op.**Operation**(*name*, *needs=None*, *provides=None*)
An abstract class representing a data transformation by *compute()*.

compute(*named_inputs*, *outputs=None*)
Compute (optional) asked *outputs* for the given *named_inputs*.

It is called by *Network*. End-users should simply call the operation with *named_inputs* as kwargs.

Parameters **named_inputs** (*list*) – A list of Data objects on which to run the layer's feed-forward computation.

Returns list Should return a list values representing the results of running the feed-forward computation on inputs.

There is a better way to instantiate an FunctionalOperation than simply constructing it, and we'll get to it later. First off, though, here's the specifications for the *operation* classes:

class graphtik.op.**FunctionalOperation**(*fn: Callable*, *name*, *needs=None*, *provides=None*, ***, *returns_dict=None*)

An Operation performing a callable (ie function, method, lambda).

Use *operation()* factory to build instances of this class instead.

__init__(*fn: Callable*, *name*, *needs=None*, *provides=None*, ***, *returns_dict=None*)

Create a new layer instance. Names may be given to this layer and its inputs and outputs. This is important when connecting layers and data in a Network object, as the names are used to construct the graph.

Parameters

- **name** (*str*) – The name the operation (e.g. conv1, conv2, etc..)
- **needs** (*list*) – Names of input data objects this layer requires.
- **provides** (*list*) – Names of output data objects this provides.

compute(*named_inputs*, *outputs=None*) → dict

Compute (optional) asked *outputs* for the given *named_inputs*.

It is called by *Network*. End-users should simply call the operation with *named_inputs* as kwargs.

Parameters **named_inputs** (*list*) – A list of Data objects on which to run the layer's feed-forward computation.

Returns list Should return a list values representing the results of running the feed-forward computation on inputs.

__call__(**args*, ***kwargs*)

Call self as a function.

1.1.1 Operations are just functions

At the heart of each operation is just a function, any arbitrary function. Indeed, you can instantiate an operation with a function and then call it just like the original function, e.g.:

```
>>> from operator import add
>>> from graphtik import operation

>>> add_op = operation(name='add_op', needs=['a', 'b'], provides=['a_plus_b'])(add)

>>> add_op(3, 4) == add(3, 4)
True
```

1.1.2 Specifying graph structure: provides and needs

Of course, each operation is more than just a function. It is a node in a computation graph, depending on other nodes in the graph for input data and supplying output data that may be used by other nodes in the graph (or as a graph output). This graph structure is specified via the provides and needs arguments to the operation constructor. Specifically:

- provides: this argument names the outputs (i.e. the returned values) of a given operation. If multiple outputs are specified by provides, then the return value of the function comprising the operation must return an iterable.
- needs: this argument names data that is needed as input by a given operation. Each piece of data named in needs may either be provided by another operation in the same graph (i.e. specified in the provides

argument of that `operation`), or it may be specified as a named input to a graph computation (more on graph computations *here*).

When many operations are composed into a computation graph (see *Graph Composition* for more on that), Graphtik matches up the values in their `needs` and `provides` to form the edges of that graph.

Let's look again at the operations from the script in *Quick start*, for example:

```
>>> from operator import mul, sub
>>> from functools import partial
>>> from graphtik import compose, operation

>>> # Computes |a|^p.
>>> def abspow(a, p):
...   c = abs(a) ** p
...   return c

>>> # Compose the mul, sub, and abspow operations into a computation graph.
>>> graphop = compose("graphop",
...    operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
...    operation(name="sub1", needs=["a", "ab"], provides=["a_minus_ab"])(sub),
...    operation(name="abspow1", needs=["a_minus_ab"], provides=["abs_a_minus_ab_cubed
↪"])
...    (partial(abspow, p=3))
... )
```

Tip: Notice the use of `functools.partial()` to set parameter `p` to a contant value.

The `needs` and `provides` arguments to the operations in this script define a computation graph that looks like this (where the oval are *operations*, squares/houses are *data*):

Tip: See *Plotting* on how to make diagrams like this.

1.1.3 Instantiating operations

There are several ways to instantiate an `operation`, each of which might be more suitable for different scenarios.

Decorator specification

If you are defining your computation graph and the functions that comprise it all in the same script, the decorator specification of `operation` instances might be particularly useful, as it allows you to assign computation graph structure to functions as they are defined. Here's an example:

```
>>> from graphtik import operation, compose

>>> @operation(name='foo_op', needs=['a', 'b', 'c'], provides='foo')
... def foo(a, b, c):
...   return c * (a + b)

>>> graphop = compose('foo_graph', foo)
```

Functional specification

If the functions underlying your computation graph operations are defined elsewhere than the script in which your graph itself is defined (e.g. they are defined in another module, or they are system functions), you can use the functional specification of `operation` instances:

```
>>> from operator import add, mul
>>> from graphtik import operation, compose

>>> add_op = operation(name='add_op', needs=['a', 'b'], provides='sum')(add)
>>> mul_op = operation(name='mul_op', needs=['c', 'sum'], provides='product')(mul)

>>> graphop = compose('add_mul_graph', add_op, mul_op)
```

The functional specification is also useful if you want to create multiple `operation` instances from the same function, perhaps with different parameter values, e.g.:

```
>>> from functools import partial

>>> def mypow(a, p=2):
...     return a ** p

>>> pow_op1 = operation(name='pow_op1', needs=['a'], provides='a_squared')(mypow)
>>> pow_op2 = operation(name='pow_op2', needs=['a'], provides='a_cubed
↪')(partial(mypow, p=3))

>>> graphop = compose('two_pows_graph', pow_op1, pow_op2)
```

A slightly different approach can be used here to accomplish the same effect by creating an operation "builder pattern":

```
>>> def mypow(a, p=2):
...     return a ** p

>>> pow_op_factory = operation(mypow, needs=['a'], provides='a_squared')

>>> pow_op1 = pow_op_factory(name='pow_op1')
>>> pow_op2 = pow_op_factory.withset(name='pow_op2', provides='a_cubed
↪')(partial(mypow, p=3))
>>> pow_op3 = pow_op_factory(lambda a: 1, name='pow_op0')

>>> graphop = compose('two_pows_graph', pow_op1, pow_op2, pow_op3)
>>> graphop(a=2)
{'a': 2, 'a_cubed': 8, 'a_squared': 4}
```

Note: You cannot call again the factory to overwrite the *function*, you have to use either the `fn=` keyword with `withset()` method or call once more.

1.1.4 Modifiers on `operation` inputs and outputs

Certain modifiers are available to apply to input or output values in `needs` and `provides`, for example, to designate optional inputs, or "ghost" sideffects inputs & outputs. These modifiers are available in the `graphtik.modifiers` module:

Optionals

class graphtik.modifiers.**optional**

An optional need signifies that the function's argument may not receive a value.

Only input values in `needs` may be designated as optional using this modifier. An `operation` will receive a value for an optional need only if if it is available in the graph at the time of its invocation. The `operation`'s function should have a defaulted parameter with the same name as the opetional, and the input value will be passed as a keyword argument, if it is available.

Here is an example of an operation that uses an optional argument:

```
>>> from graphtik import operation, compose, optional

>>> def myadd(a, b, c=0):
...     return a + b + c
```

Designate c as an optional argument:

```
>>> graph = compose('mygraph',
...      operation(name='myadd', needs=['a', 'b', optional('c')], provides='sum
↪')(myadd)
... )
>>> graph
NetworkOperation(name='mygraph',
                 needs=['a', 'b', optional('c')],
                 provides=['sum'])
```

The graph works with and without *c* provided as input:

```
>>> graph(a=5, b=2, c=4)['sum']
11
>>> graph(a=5, b=2)
{'a': 5, 'b': 2, 'sum': 7}
```

Varargs

class graphtik.modifiers.**vararg**

Like *optional* but feeds as ONE OF the `*args` into the function (instead of `**kwargs`).

For instance:

```
>>> from graphtik import operation, compose, vararg

>>> def addall(a, *b):
...     return a + sum(b)
```

Designate *b* & *c* as an *vararg* arguments:

```
>>> graph = compose('mygraph',
...      operation(name='addall', needs=['a', vararg('b'), vararg('c')],
...      provides='sum')(addall)
... )
>>> graph
NetworkOperation(name='mygraph',
                 needs=['a', optional('b'), optional('c')],
                 provides=['sum'])
```

The graph works with and without any of *b* and *c* inputs:

```
>>> graph(a=5, b=2, c=4)['sum']
11
>>> graph(a=5, b=2)
{'a': 5, 'b': 2, 'sum': 7}
>>> graph(a=5)
{'a': 5, 'sum': 5}
```

class graphtik.modifiers.**varargs**
An optional like *vararg* feeds as MANY `*args` into the function (instead of `**kwargs`).

Read also the example test-case in: `test/test_op.py:test_varargs()`

Sideffects

class graphtik.modifiers.**sideffect**
A sideffect data-dependency participates in the graph but never given/asked in functions.

Both inputs & outputs in `needs` & `provides` may be designated as *sideffects* using this modifier. *Sideffects* work as usual while solving the graph but they do not interact with the `operation`'s function; specifically:

- input sideffects are NOT fed into the function;
- output sideffects are NOT expected from the function.

Their purpose is to describe operations that modify the internal state of some of their arguments ("side-effects"). A typical use case is to signify columns required to produce new ones in pandas dataframes:

```
>>> from graphtik import operation, compose, sideffect

>>> # Function appending a new dataframe column from two pre-existing ones.
>>> def addcolumns(df):
...     df['sum'] = df['a'] + df['b']
```

Designate *a*, *b* & *sum* column names as an sideffect arguments:

```
>>> graph = compose('mygraph',
...     operation(
...         name='addcolumns',
...         needs=['df', sideffect('df.b')],  # sideffect names can be anything
...         provides=[sideffect('df.sum')])(addcolumns)
... )
>>> graph
NetworkOperation(name='mygraph', needs=['df', 'sideffect(df.b)'],
                 provides=['sideffect(df.sum)'])

>>> df = pd.DataFrame({'a': [5, 0], 'b': [2, 1]})
>>> graph({'df': df})['df']
        a       b
0       5       2
1       0       1
```

We didn't get the `sum` column because the *b* sideffect was unsatisfied. We have to add its key to the inputs (with _any_ value):

```
>>> graph({'df': df, sideffect("df.b"): 0})['df']
        a       b       sum
```

(continues on next page)

(continued from previous page)

```
0       5       2       7
1       0       1       1
```

Note that regular data in *needs* and *provides* do not match same-named *sideffects*. That is, in the following operation, the `prices` input is different from the `sideffect(prices)` output:

```
>>> def upd_prices(sales_df, prices):
...     sales_df["Prices"] = prices
```

```
>>> operation(fn=upd_prices,
...           name="upd_prices",
...           needs=["sales_df", "price"],
...           provides=[sideffect("price")])
operation(name='upd_prices', needs=['sales_df', 'price'],
          provides=['sideffect(price)'], fn='upd_prices')
```

Note: An `operation` with *sideffects* outputs only, have functions that return no value at all (like the one above). Such operation would still be called for their side-effects.

Tip: You may associate sideffects with other data to convey their relationships, simply by including their names in the string - in the end, it's just a string - but no enforcement will happen from *graphtik*.

```
>>> sideffect("price[sales_df]")
'sideffect(price[sales_df])'
```

1.2 Graph Composition

Graphtik's `compose` factory handles the work of tying together `operation` instances into a runnable computation graph.

1.2.1 The `compose` factory

For now, here's the specification of `compose`. We'll get into how to use it in a second.

`graphtik.`**`compose`**(*name*, *op1*, **operations*, *needs=None*, *provides=None*, *merge=False*, *method=None*, *overwrites_collector=None*) → graphtik.netop.NetworkOperation

Composes a collection of operations into a single computation graph, obeying the `merge` property, if set in the constructor.

Parameters

- **`name`** (*`str`*) – A optional name for the graph being composed by this object.
- **`op1`** – syntactically force at least 1 operation
- **`operations`** – Each argument should be an operation instance created using `operation`.
- **`merge`** (*`bool`*) – If `True`, this compose object will attempt to merge together `operation` instances that represent entire computation graphs. Specifically, if one of the `operation`

instances passed to this `compose` object is itself a graph operation created by an earlier use of `compose` the sub-operations in that graph are compared against other operations passed to this `compose` instance (as well as the sub-operations of other graphs passed to this `compose` instance). If any two operations are the same (based on name), then that operation is computed only once, instead of multiple times (one for each time the operation appears).

- **`method`** – either *parallel* or None (default); if `"parallel"`, launches multi-threading. Set when invoking a composed graph or by `set_execution_method()`.
- **`overwrites_collector`** – (optional) a mutable dict to be fillwed with named values. If missing, values are simply discarded.

Returns Returns a special type of operation class, which represents an entire computation graph as a single operation.

Raises **`ValueError`** – If the *net'* cannot produce the asked *outputs* from the given *inputs*.

1.2.2 Simple composition of operations

The simplest use case for `compose` is assembling a collection of individual operations into a runnable computation graph. The example script from *Quick start* illustrates this well:

```
>>> from operator import mul, sub
>>> from functools import partial
>>> from graphtik import compose, operation

>>> # Computes |a|^p.
>>> def abspow(a, p):
...     c = abs(a) ** p
...     return c

>>> # Compose the mul, sub, and abspow operations into a computation graph.
>>> graphop = compose("graphop",
...     operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
...     operation(name="sub1", needs=["a", "ab"], provides=["a_minus_ab"])(sub),
...     operation(name="abspow1", needs=["a_minus_ab"], provides=["abs_a_minus_ab_cubed
↪"])
...     (partial(abspow, p=3))
... )
```

The call here to `compose()` yields a runnable computation graph that looks like this (where the circles are operations, squares are data, and octagons are parameters):

1.2.3 Running a computation graph

The graph composed in the example above in *Simple composition of operations* can be run by simply calling it with a dictionary argument whose keys correspond to the names of inputs to the graph and whose values are the corresponding input values. For example, if `graph` is as defined above, we can run it like this:

```
# Run the graph and request all of the outputs.
>>> out = graphop(a=2, b=5)
>>> out
{'a': 2, 'b': 5, 'ab': 10, 'a_minus_ab': -8, 'abs_a_minus_ab_cubed': 512}
```

Producing a subset of outputs

By default, calling a graph-operation on a set of inputs will yield all of that graph's outputs. You can use the `outputs` parameter to request only a subset. For example, if `graphop` is as above:

```
# Run the graph-operation and request a subset of the outputs.
>>> out = graphop.compute({'a': 2, 'b': 5}, outputs="a_minus_ab")
>>> out
{'a_minus_ab': -8}
```

When using `outputs` to request only a subset of a graph's outputs, Graphtik executes only the `operation` nodes in the graph that are on a path from the inputs to the requested outputs. For example, the `abspow1` operation will not be executed here.

Short-circuiting a graph computation

You can short-circuit a graph computation, making certain inputs unnecessary, by providing a value in the graph that is further downstream in the graph than those inputs. For example, in the graph-operation we've been working with, you could provide the value of `a_minus_ab` to make the inputs `a` and `b` unnecessary:

```
# Run the graph-operation and request a subset of the outputs.
>>> out = graphop(a_minus_ab=-8)
>>> out
{'a_minus_ab': -8, 'abs_a_minus_ab_cubed': 512}
```

When you do this, any `operation` nodes that are not on a path from the downstream input to the requested outputs (i.e. predecessors of the downstream input) are not computed. For example, the `mul1` and `sub1` operations are not executed here.

This can be useful if you have a graph-operation that accepts alternative forms of the same input. For example, if your graph-operation requires a `PIL.Image` as input, you could allow your graph to be run in an API server by adding an earlier `operation` that accepts as input a string of raw image data and converts that data into the needed `PIL.Image`. Then, you can either provide the raw image data string as input, or you can provide the `PIL.Image` if you have it and skip providing the image data string.

1.2.4 Adding on to an existing computation graph

Sometimes you will have an existing computation graph to which you want to add operations. This is simple, since `compose` can compose whole graphs along with individual `operation` instances. For example, if we have `graph` as above, we can add another operation to it to create a new graph:

```
>>> # Add another subtraction operation to the graph.
>>> bigger_graph = compose("bigger_graph",
...     graphop,
...     operation(name="sub2", needs=["a_minus_ab", "c"], provides="a_minus_ab_minus_c
↪")(sub)
... )

>>> # Run the graph and print the output.
>>> sol = bigger_graph.compute({'a': 2, 'b': 5, 'c': 5}, outputs=["a_minus_ab_minus_c
↪"])
>>> sol
{'a_minus_ab_minus_c': -13}
```

This yields a graph which looks like this (see *Plotting*):

```
>>> bigger_graph.plot('bigger_example_graph.svg', solution=sol)
```

1.2.5 More complicated composition: merging computation graphs

Sometimes you will have two computation graphs—perhaps ones that share operations—you want to combine into one. In the simple case, where the graphs don't share operations or where you don't care whether a duplicated operation is run multiple (redundant) times, you can just do something like this:

```
combined_graph = compose("combined_graph", graph1, graph2)
```

However, if you want to combine graphs that share operations and don't want to pay the price of running redundant computations, you can set the `merge` parameter of `compose()` to `True`. This will consolidate redundant `operation` nodes (based on `name`) into a single node. For example, let's say we have `graphop`, as in the examples above, along with this graph:

```
>>> # This graph shares the "mul1" operation with graph.
>>> another_graph = compose("another_graph",
...     operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
...     operation(name="mul2", needs=["c", "ab"], provides=["cab"])(mul)
... )
```

We can merge `graphop` and `another_graph` like so, avoiding a redundant `mul1` operation:

```
>>> merged_graph = compose("merged_graph", graphop, another_graph, merge=True)
>>> print(merged_graph)
NetworkOperation(name='merged_graph',
                 needs=['a', 'b', 'c'],
                 provides=['ab', 'a_minus_ab', 'abs_a_minus_ab_cubed', 'cab'])
```

This `merged_graph` will look like this:

As always, we can run computations with this graph by simply calling it:

```
>>> merged_graph.compute({'a': 2, 'b': 5, 'c': 5}, outputs=["cab"])
{'cab': 50}
```

1.3 Plotting and Debugging

1.3.1 Plotting

For *Errors & debugging* it is necessary to visualize the graph-operation. You may plot the original plot and annotate on top the *execution plan* and solution of the last computation, calling methods with arguments like this:

```
netop.plot(show=True)                # open a matplotlib window
netop.plot("netop.svg")            # other supported formats: png, jpg, pdf, ...
netop.plot()                         # without arguments return a pydot.DOT object
netop.plot(solution=out)             # annotate graph with solution values
```

Fig. 1: The legend for all graphtik diagrams, generated by *legend()*.

The same *Plotter.plot()* method applies for *NetworkOperation*, *Network* & *ExecutionPlan*, each one capable to produce diagrams with increasing complexity. Whenever possible, the top-level `plot()` methods delegates to the ones below.

For instance, when a net-operation has just been composed, plotting it will come out bare bone, with just the 2 types of nodes (data & operations), their dependencies, and the sequence of the execution-plan.

But as soon as you run it, the net plot calls will print more of the internals. Internally it delegates to `ExecutionPlan.plot()` of *NetworkOperation.last_plan* attribute, which *caches* the last run to facilitate debugging. If you want the bare-bone diagram, plot network:

```
netop.net.plot(...)
```

Note: For plots, Graphviz program must be in your PATH, and `pydot` & `matplotlib` python packages installed. You may install both when installing `graphtik` with its `plot` extras:

```
pip install graphtik[plot]
```

Tip: The pydot.Dot instances returned by *Plotter.plot()* are rendered directly in *Jupyter/IPython* notebooks as SVG images.

You may increase the height of the SVG cell output with something like this:

```
netop.plot(jupyter_render={"svg_element_styles": "height: 600px; width: 100%"})
```

Check *default_jupyter_render* for defaults.

1.3.2 Errors & debugging

Graphs may become arbitrary deep. Launching a debugger-session to inspect deeply nested stacks is notoriously hard

As a workaround, when some operation fails, the original exception gets annotated with the folllowing properties, as a debug aid:

```
>>> from graphtik import compose, operation
>>> from pprint import pprint
```

```
>>> def scream(*args):
...     raise ValueError("Wrong!")
```

```
>>> try:
...     compose("errgraph",
...             operation(name="screamer", needs=['a'], provides=["foo"])(scream)
...     )(a=None)
... except ValueError as ex:
```

(continues on next page)

(continued from previous page)

```
...     pprint(ex.jetsam)
{'args': {'args': [None], 'kwargs': {}},
 'executed': set(),
 'network': Network(
    +--a
    +--FunctionalOperation(name='screamer', needs=['a'], provides=['foo'], fn='scream
↪')
    +--foo),
 'operation': FunctionalOperation(name='screamer', needs=['a'], provides=['foo'], fn=
↪'scream'),
 'outputs': None,
 'plan': ExecutionPlan(needs=['a'], provides=['foo'], steps:
  +--FunctionalOperation(name='screamer', needs=['a'], provides=['foo'], fn='scream
↪')),
 'provides': None,
 'results_fn': None,
 'results_op': None,
 'solution': {'a': None}}
```

In interactive *REPL* console you may use this to get the last raised exception:

```
import sys

sys.last_value.jetsam
```

The following annotated attributes *might* have meaningfull value on an exception:

network the innermost network owning the failed operation/function

plan the innermost plan that executing when a operation crashed

operation the innermost operation that failed

args either the input arguments list fed into the function, or a dict with both `args` & `kwargs` keys in it.

outputs the names of the outputs the function was expected to return

provides the names eventually the graph needed from the operation; a subset of the above, and not always what has been declared in the operation.

fn_results the raw results of the operation's fuction, if any

op_results the results, always a dictionary, as matched with operation's *provides*

executed` a set with the operation nodes & instructions executed till the error happened.

Ofcourse you may use many of the above "jetsam" values when plotting.

Note: The *Plotting* capabilities, along with the above annotation of exceptions with the internal state of plan/operation often renders a debugger session unnecessary. But since the state of the annotated values might be incomple, you may not always avoid one.

1.3.3 Execution internals

Network-based computation of operations & data.

The execution of network *operations* is splitted in 2 phases:

COMPILE: prune unsatisfied nodes, sort dag topologically & solve it, and derive the *execution steps* (see below) based on the given *inputs* and asked *outputs*.

EXECUTE: sequential or parallel invocation of the underlying functions of the operations with arguments from the `solution`.

Computations are based on 5 data-structures:

`Network.graph` A `networkx` graph (yet a DAG) containing interchanging layers of `Operation` and `_DataNode` nodes. They are layed out and connected by repeated calls of `add_OP()`.

The computation starts with `prune()` extracting a *DAG subgraph* by *pruning* its nodes based on given inputs and requested outputs in `compute()`.

`ExecutionPlan.dag` An directed-acyclic-graph containing the *pruned* nodes as build by `prune()`. This pruned subgraph is used to decide the `ExecutionPlan.steps` (below). The containing `ExecutionPlan.steps` instance is cached in `_cached_plans` across runs with inputs/outputs as key.

`ExecutionPlan.steps` It is the list of the operation-nodes only from the dag (above), topologically sorted, and interspersed with *instruction steps* needed to complete the run. It is built by `_build_execution_steps()` based on the subgraph dag extracted above. The containing `ExecutionPlan.steps` instance is cached in `_cached_plans` across runs with inputs/outputs as key.

The *instructions* items achieve the following:

- **`_EvictInstruction`: evicts items from *solution* as soon as** they are not needed further down the dag, to reduce memory footprint while computing.
- **`_PinInstruction`: avoid overwritting any given intermediate** inputs, and still allow their providing operations to run (because they are needed for their other outputs).

var solution a local-var in `compute()`, initialized on each run to hold the values of the given inputs, generated (intermediate) data, and output values. It is returned as is if no specific outputs requested; no data-eviction happens then.

arg overwrites The optional argument given to `compute()` to colect the intermediate *calculated* values that are overwritten by intermediate (aka "pinned") input-values.

1.4 API Reference

1.4.1 Package: *graphtik*

Lightweight computation graphs for Python.

1.4.2 Module: *base*

Mostly utilities

class graphtik.base.**Plotter**

Classes wishing to plot their graphs should inherit this and ...

implement property `plot` to return a "partial" callable that somehow ends up calling *`plot.render_pydot()`* with the *graph* or any other args binded appropriately. The purpose is to avoid copying this function & documentation here around.

plot(*filename=None, show=False, jupyter_render: Union[None, Mapping[KT, VT_co], str] = None, **kws*)

Entry-point for plotting ready made operation graphs.

Parameters

- **filename** (*str*) – Write diagram into a file. Common extensions are `.png .dot .jpg .jpeg .pdf .svg` call `plot.supported_plot_formats()` for more.
- **show** – If it evaluates to true, opens the diagram in a matplotlib window. If it equals *-1*, it plots but does not open the Window.
- **inputs** – an optional name list, any nodes in there are plotted as a "house"
- **outputs** – an optional name list, any nodes in there are plotted as an "inverted-house"
- **solution** – an optional dict with values to annotate nodes, drawn "filled" (currently content not shown, but node drawn as "filled")
- **executed** – an optional container with operations executed, drawn "filled"
- **title** – an optional string to display at the bottom of the graph
- **node_props** – an optional nested dict of Grapvhiz attributes for certain nodes
- **edge_props** – an optional nested dict of Grapvhiz attributes for certain edges
- **clusters** – an optional mapping of nodes –> cluster-names, to group them
- **jupyter_render** – a nested dictionary controlling the rendering of graph-plots in Jupyter cells, if *None*, defaults to `jupyter_render` (you may modify it in place and apply for all future calls).

Returns

a pydot.Dot instance (for for API reference visit: https://pydotplus.readthedocs.io/reference.html#pydotplus.graphviz.Dot)

Tip: The `pydot.Dot` instance returned is rendered directly in *Jupyter/IPython* notebooks as SVG images.

You may increase the height of the SVG cell output with something like this:

```
netop.plot(svg_element_styles="height: 600px; width: 100%")
```

Check *default_jupyter_render* for defaults.

Note that the *graph* argument is absent - Each Plotter provides its own graph internally; use directly *render_pydot()* to provide a different graph.

NODES:

oval function

egg subgraph operation

house given input

inversed-house asked output

polygon given both as input & asked as output (what?)

square intermediate data, neither given nor asked.

red frame evict-instruction, to free up memory.

blue frame pinned-instruction, not to overwrite intermediate inputs.

filled data node has a value in *solution* OR function has been executed.

thick frame function/data node in execution *steps*.

ARROWS

solid black arrows dependencies (source-data *need*-ed by target-operations, sources-operations *provides* target-data)

dashed black arrows optional needs

blue arrows sideffect needs/provides

wheat arrows broken dependency (`provide`) during pruning

green-dotted arrows execution steps labeled in succession

To generate the **legend**, see *`legend()`*.

Sample code:

```
>>> from graphtik import compose, operation
>>> from graphtik.modifiers import optional
>>> from operator import add
```

```
>>> netop = compose("netop",
...     operation(name="add", needs=["a", "b1"], provides=["ab1"])(add),
...     operation(name="sub", needs=["a", optional("b2")], provides=["ab2
↪"])(lambda a, b=1: a-b),
...     operation(name="abb", needs=["ab1", "ab2"], provides=["asked"])(add),
... )
```

```
>>> netop.plot(show=True);                  # plot just the graph in a␣
↪matplotlib window # doctest: +SKIP
>>> inputs = {'a': 1, 'b1': 2}
>>> solution = netop(**inputs)              # now plots will include the␣
↪execution-plan
```

```
>>> netop.plot('plot1.svg', inputs=inputs, outputs=['asked', 'b1'],␣
↪solution=solution);           # doctest: +SKIP
>>> dot = netop.plot(solution=solution);   # just get the `pydoit.Dot` object,
↪ renderable in Jupyter
>>> print(dot)
digraph G {
  fontname=italic;
  label=netop;
  a [fillcolor=wheat, shape=invhouse, style=filled, tooltip=1];
...
```

graphtik.base.**aslist**(*i*, *argname*, *allowed_types=<class 'list'>*)
Utility to accept singular strings as lists, and None –> [].

graphtik.base.**astuple**(*i*, *argname*, *allowed_types=<class 'tuple'>*)

graphtik.base.**jetsam**(*ex*, *locs*, **salvage_vars*, *annotation='jetsam'*, ***salvage_mappings*)
Annotate exception with salvaged values from locals() and raise!

Parameters

- **ex** – the exception to annotate

- **locs** – `locals()` from the context-manager's block containing vars to be salvaged in case of exception

  ATTENTION: wrapped function must finally call `locals()`, because *locals* dictionary only reflects local-var changes after call.

- **annotation** – the name of the attribute to attach on the exception
- **salvage_vars** – local variable names to save as is in the salvaged annotations dictionary.
- **salvage_mappings** – a mapping of destination-annotation-keys –> source-locals-keys; if a *source* is callable, the value to salvage is retrieved by calling `value(locs)`. They take precendance over'salvae_vars'.

Raises any exception raised by the wrapped function, annotated with values assigned as atrributes on this context-manager

- Any attrributes attached on this manager are attached as a new dict on the raised exception as new `jetsam` attrribute with a dict as value.
- If the exception is already annotated, any new items are inserted, but existing ones are preserved.

Example:

Call it with managed-block's `locals()` and tell which of them to salvage in case of errors:

```
try:
    a = 1
    b = 2
    raise Exception()
exception Exception as ex:
    jetsam(ex, locals(), "a", b="salvaged_b", c_var="c")
```

And then from a REPL:

```
import sys
sys.last_value.jetsam
{'a': 1, 'salvaged_b': 2, "c_var": None}
```

** Reason:**

Graphs may become arbitrary deep. Debugging such graphs is notoriously hard.

The purpose is not to require a debugger-session to inspect the root-causes (without precluding one).

Naively salvaging values with a simple try/except block around each function, blocks the debugger from landing on the real cause of the error - it would land on that block; and that could be many nested levels above it.

1.4.3 Module: *op*

About operation nodes (but not net-ops to break cycle).

class graphtik.op.**FunctionalOperation**(*fn: Callable*, *name*, *needs=None*, *provides=None*, *, *returns_dict=None*)

An Operation performing a callable (ie function, method, lambda).

Use *operation()* factory to build instances of this class instead.

compute(*named_inputs*, *outputs=None*) → dict

Compute (optional) asked *outputs* for the given *named_inputs*.

It is called by *Network*. End-users should simply call the operation with *named_inputs* as kwargs.

Parameters **named_inputs** (*list*) – A list of `Data` objects on which to run the layer's feed-forward computation.

Returns list Should return a list values representing the results of running the feed-forward computation on `inputs`.

class graphtik.op.**Operation**(*name*, *needs=None*, *provides=None*)
An abstract class representing a data transformation by *compute()*.

compute(*named_inputs*, *outputs=None*)
Compute (optional) asked *outputs* for the given *named_inputs*.

It is called by *Network*. End-users should simply call the operation with *named_inputs* as kwargs.

Parameters **named_inputs** (*list*) – A list of `Data` objects on which to run the layer's feed-forward computation.

Returns list Should return a list values representing the results of running the feed-forward computation on `inputs`.

class graphtik.op.**operation**(*fn: Callable = None*, ***, *name=None*, *needs=None*, *provides=None*, *returns_dict=None*)
A builder for graph-operations wrapping functions.

Parameters

- **fn** (*function*) – The function used by this operation. This does not need to be specified when the operation object is instantiated and can instead be set via `__call__` later.
- **name** (*str*) – The name of the operation in the computation graph.
- **needs** (*list*) – Names of input data objects this operation requires. These should correspond to the `args` of `fn`.
- **provides** (*list*) – Names of output data objects this operation provides. If more than one given, those must be returned in an iterable, unless *returns_dict* is true, in which cae a dictionary with as many elements must be returned
- **returns_dict** (*bool*) – if true, it means the *fn* returns a dictionary with all *provides*, and no further processing is done on them.

Returns when called, it returns a *FunctionalOperation*

Example:

This is an example of its use, based on the "builder pattern":

```
>>> from graphtik import operation

>>> opb = operation(name='add_op')
>>> opb.withset(needs=['a', 'b'])
operation(name='add_op', needs=['a', 'b'], provides=[], fn=None)
>>> opb.withset(provides='SUM', fn=sum)
operation(name='add_op', needs=['a', 'b'], provides=['SUM'], fn='sum')
```

You may keep calling `withset()` till you invoke a final `__call__()` on the builder; then you get the actual *FunctionalOperation* instance:

```
>>> # Create `Operation` and overwrite function at the last moment.
>>> opb(sum)
FunctionalOperation(name='add_op', needs=['a', 'b'], provides=['SUM'], fn='sum')
```

withset(***, *fn=None*, *name=None*, *needs=None*, *provides=None*, *returns_dict=None*) → graphtik.op.operation

graphtik.op.**reparse_operation_data**(*name*, *needs*, *provides*)
Validate & reparse operation data as lists.

As a separate function to be reused by client code when building operations and detect errors aearly.

1.4.4 Module: *netop*

About network-operations (those based on graphs)

class graphtik.netop.**NetworkOperation**(*net*, *name*, *, *inputs=None*, *outputs=None*, *method=None*, *overwrites_collector=None*)
An Operation performing a network-graph of other operations.

Tip: Use *compose()* factory to prepare the *net* and build instances of this class.

---

compute(*named_inputs*, *outputs=None*, *recompile=None*) → dict
Solve & execute the graph, sequentially or parallel.

It see also *Operation.compute()*.

Parameters

- **named_inputs** (*dict*) – A maping of names –> values that must contain at least the compulsory inputs that were specified when the plan was built (but cannot enforce that!). Cloned, not modified.
- **outputs** – a string or a list of strings with all data asked to compute. If you set this variable to None, all data nodes will be kept and returned at runtime.
- **recompile** –
  - if *False*, uses fixed *plan*;
  - if true, recompiles a temporary plan from network;
  - if *None*, assumed true if *outputs* given (is not *None*).

  In all cases, the *:attr:'last_plan* is updated.

Returns a dictionary of output data objects, keyed by name.

Raises **ValueError** –

- If given *inputs* mismatched *plan.needs*, with msg:

  Plan needs more inputs...
- If *outputs* asked do not exist in network, with msg:

  Unknown output nodes: ...
- If *outputs* asked cannot be produced by the *graph*, with msg:

  Impossible outputs...
- If cannot produce any *outputs* from the given *inputs*, with msg:

  Unsolvable graph: ...

inputs = None
The inputs names (possibly *None*) used to compile the *plan*.

last_plan = None
The execution_plan of the last call to compute(), stored as debugging aid.

method = None
set execution mode to single-threaded sequential by default

narrow(*inputs: Collection[T_co] = None*, *outputs: Collection[T_co] = None*, *name=None*) → graphtik.netop.NetworkOperation
Return a copy with a network pruned for the given *needs* & *provides*.

Parameters

- **inputs** – a collection of inputs that must be given to `compute()`; a WARNing is issued for any irrelevant arguments. If *None*, they are collected from the `net`. They become the *needs* of the returned *netop*.
- **outputs** – a collection of outputs that will be asked from `compute()`; RAISES if those cannnot be satisfied. If *None*, they are collected from the `net`. They become the *provides* of the returned *netop*.
- **name** – the name for the new netop:
  - if *None*, the same name is kept;
  - if True, a distinct name is devised:

    ```
    <old-name>-<uid>
    ```

  - otherwise, the given *name* is applied.

Returns a cloned netop with a narrowed plan

Raises **ValueError** –

- If *outputs* asked do not exist in network, with msg:

  Unknown output nodes: ...

- If *outputs* asked cannot be produced by the *graph*, with msg:

  Impossible outputs...

- If cannot produce any *outputs* from the given *inputs*, with msg:

  Unsolvable graph: ...

outputs = None
The outputs names (possibly *None*) used to compile the `plan`.

overwrites_collector = None

plan = None
The *narrowed plan* enforcing unvarying *needs* & *provides* when `compute()` called with `recompile=False` (default is `recompile=None`, which means, recompile only if *outputs* given).

set_execution_method(*method*)
Determine how the network will be executed.

Parameters **method** (*str*) – If "parallel", execute graph operations concurrently using a threadpool.

set_overwrites_collector(*collector*)
Asks to put all *overwrites* into the *collector* after computing

An "overwrites" is intermediate value calculated but NOT stored into the results, becaues it has been given also as an intemediate input value, and the operation that would overwrite it MUST run for its other results.

Parameters **collector** – a mutable dict to be fillwed with named values

graphtik.netop.**compose**(*name*, *op1*, **operations*, *needs=None*, *provides=None*, *merge=False*, *method=None*, *overwrites_collector=None*) → graphtik.netop.NetworkOperation

Composes a collection of operations into a single computation graph, obeying the `merge` property, if set in the constructor.

Parameters

- **name** (*str*) – A optional name for the graph being composed by this object.
- **op1** – syntactically force at least 1 operation
- **operations** – Each argument should be an operation instance created using `operation`.
- **merge** (*bool*) – If `True`, this compose object will attempt to merge together `operation` instances that represent entire computation graphs. Specifically, if one of the `operation` instances passed to this `compose` object is itself a graph operation created by an earlier use of `compose` the sub-operations in that graph are compared against other operations passed to this `compose` instance (as well as the sub-operations of other graphs passed to this `compose` instance). If any two operations are the same (based on name), then that operation is computed only once, instead of multiple times (one for each time the operation appears).
- **method** – either *parallel* or None (default); if `"parallel"`, launches multi-threading. Set when invoking a composed graph or by *set_execution_method()*.
- **overwrites_collector** – (optional) a mutable dict to be fillwed with named values. If missing, values are simply discarded.

Returns Returns a special type of operation class, which represents an entire computation graph as a single operation.

Raises **ValueError** – If the *net'* cannot produce the asked *outputs* from the given *inputs*.

1.4.5 Module: *network*

Network-based computation of operations & data.

The execution of network *operations* is splitted in 2 phases:

COMPILE: prune unsatisfied nodes, sort dag topologically & solve it, and derive the *execution steps* (see below) based on the given *inputs* and asked *outputs*.

EXECUTE: sequential or parallel invocation of the underlying functions of the operations with arguments from the `solution`.

Computations are based on 5 data-structures:

Network.graph A `networkx` graph (yet a DAG) containing interchanging layers of `Operation` and `_DataNode` nodes. They are layed out and connected by repeated calls of `add_OP()`.

The computation starts with `prune()` extracting a *DAG subgraph* by *pruning* its nodes based on given inputs and requested outputs in `compute()`.

ExecutionPlan.dag An directed-acyclic-graph containing the *pruned* nodes as build by `prune()`. This pruned subgraph is used to decide the `ExecutionPlan.steps` (below). The containing `ExecutionPlan.steps` instance is cached in `_cached_plans` across runs with inputs/outputs as key.

ExecutionPlan.steps It is the list of the operation-nodes only from the dag (above), topologically sorted, and interspersed with *instruction steps* needed to complete the run. It is built by `_build_execution_steps()`

based on the subgraph dag extracted above. The containing `ExecutionPlan.steps` instance is cached in `_cached_plans` across runs with inputs/outputs as key.

The *instructions* items achieve the following:

- **`_EvictInstruction`: evicts items from *solution* as soon as** they are not needed further down the dag, to reduce memory footprint while computing.
- **`_PinInstruction`: avoid overwritting any given intermediate** inputs, and still allow their providing operations to run (because they are needed for their other outputs).

var solution a local-var in `compute()`, initialized on each run to hold the values of the given inputs, generated (intermediate) data, and output values. It is returned as is if no specific outputs requested; no data-eviction happens then.

arg overwrites The optional argument given to `compute()` to colect the intermediate *calculated* values that are overwritten by intermediate (aka "pinned") input-values.

exception graphtik.network.**AbortedException**
Raised from the Network code when `abort_run()` is called.

graphtik.network.**_execution_configs = <ContextVar name='execution_configs' default={'execut**
Global configurations for all (nested) networks in a computaion run.

class graphtik.network.**Network**(**operations*)
Assemble operations & data into a directed-acyclic-graph (DAG) to run them.

Variables

- **needs** – the "base", all data-nodes that are not produced by some operation
- **provides** – the "base", all data-nodes produced by some operation

class graphtik.network.**ExecutionPlan**
The result of the network's compilation phase.

Note the execution plan's attributes are on purpose immutable tuples.

Variables

- **net** – The parent *Network*
- **needs** – A tuple with the input names needed to exist in order to produce all *provides*.
- **provides** – A tuple with the outputs names produces when all *inputs* are given.
- **dag** – The regular (not broken) *pruned* subgraph of net-graph.
- **broken_edges** – Tuple of broken incoming edges to given data.
- **steps** – The tuple of operation-nodes & *instructions* needed to evaluate the given inputs & asked outputs, free memory and avoid overwritting any given intermediate inputs.
- **evict** – when false, keep all inputs & outputs, and skip prefect-evictions check.

1.4.6 Module: *plot*

Plotting graphtik graps

graphtik.plot.**build_pydot**(*graph*, *steps=None*, *inputs=None*, *outputs=None*, *solution=None*, *executed=None*, *title=None*, *node_props=None*, *edge_props=None*, *clusters=None*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f688418cb70>
Build a *Graphviz* out of a Network graph/steps/inputs/outputs and return it.

See *Plotter.plot()* for the arguments, sample code, and the legend of the plots.

graphtik.plot.**default_jupyter_render = {'svg_container_styles': '', 'svg_element_styles'**
A nested dictionary controlling the rendering of graph-plots in Jupyter cells,

as those returned from *Plotter.plot()* (currently as SVGs). Either modify it in place, or pass another one in the respective methods.

The following keys are supported.

Parameters

- **svg_pan_zoom_json** – arguments controlling the rendering of a zoomable SVG in Jupyter notebooks, as defined in https://github.com/ariutta/svg-pan-zoom#how-to-use if *None*, defaults to string (also maps supported):

```
"{controlIconsEnabled: true, zoomScaleSensitivity: 0.4, fit: true}"
```

- **svg_element_styles** – mostly for sizing the zoomable SVG in Jupyter notebooks. Inspect & experiment on the html page of the notebook with browser tools. if *None*, defaults to string (also maps supported):

```
"width: 100%; height: 300px;"
```

- **svg_container_styles** – like *svg_element_styles*, if *None*, defaults to empty string (also maps supported).

graphtik.plot.**legend**(*filename=None*, *show=None*, *jupyter_render: Optional[Mapping[KT, VT_co]] = None*)
Generate a legend for all plots (see *Plotter.plot()* for args)

graphtik.plot.**render_pydot**(*dot: <sphinx.ext.autodoc.importer._MockObject object at 0x7f688418ccc0>*, *filename=None*, *show=False*, *jupyter_render: str = None*)
Plot a *Graphviz* dot in a matplotlib, in file or return it for Jupyter.

Parameters

- **dot** – the pre-built *Graphviz* `pydot.Dot` instance
- **filename** (*str*) – Write diagram into a file. Common extensions are `.png .dot .jpg .jpeg .pdf .svg` call `plot.supported_plot_formats()` for more.
- **show** – If it evaluates to true, opens the diagram in a matplotlib window. If it equals *-1*, it returns the image but does not open the Window.
- **jupyter_render** – a nested dictionary controlling the rendering of graph-plots in Jupyter cells. If *None*, defaults to *default_jupyter_render* (you may modify those in place and they will apply for all future calls).

Returns the matplotlib image if `show=-1`, or the *dot*.

See *Plotter.plot()* for sample code.

graphtik.plot.**supported_plot_formats**() → List[str]
return automatically all *pydot* extensions

1.5 Graphtik Changelog

1.5.1 TODO

See #1.

1.5.2 v3.0.0 (27 Nov 2019, @ankostis): UNVARYING NetOperations, narrowed, API refact

- NetworkOperations:
  - BREAK(NET): RAISE if the graph is UNSOLVABLE for the given *needs* & *provides*! (see "raises" list of *compute()*).
  - BREAK: `NetworkOperation.__call__()` accepts solution as keyword-args, to mimic API of `Operation.__call__()`. `outputs` keyword has been dropped.

    Tip: Use *NetworkOperation.compute()* when you ask different *outputs*, or set the `recompile` flag if just different *inputs* are given.

    Read the next change-items for the new behavior of the `compute()` method.

  - UNVARYING NetOperations:
    - BREAK: calling method *NetworkOperation.compute()* with a single argument is now *UNVARYING*, meaning that all *needs* are demaned, and hence, all *provides* are produced, unless the `recompile` flag is true or `outputs` asked.
    - BREAK: net-operations behave like regular operations when nested inside another netop, and always produce all their *provides*, or scream if less *inputs* than *needs* are given.
    - ENH: a newly created or cloned netop can be *narrow()*ed to specific *needs* & *provides*, so as not needing to pass *outputs* on every call to *compute()*.
    - feat: implemented based on the new "narrowed" *NetworkOperation.plan* attribute.
  - FIX: netop *needs* are not all *optional* by default; optionality applied only if all underlying operations have a certain need as optional.
  - FEAT: support function `**args` with 2 new modifiers *vararg* & *varargs*, acting like *optional* (but without feeding into underlying functions like keywords).
  - BREAK(yahoo#12): simplify `compose` API by turning it from class –> function; all args and operations are now given in a single `compose()` call.
  - REFACT(net, netop): make Network IMMUTABLE by appending all operations together, in `NetworkOperation` constructor.
  - ENH(net): public-size `_prune_graph()` –> `Network.prune()` `()` which can be used to interogate *needs* & *provides* for a given graph. It accepts *None inputs* & *outputs* to auto-derrive them.
- FIX(SITE): autodocs *API* chapter were not generated in at all, due to import errors, fixed by using autodoc_mock_imports on *networkx*, *pydot* & *boltons* libs.
- enh(op): polite error-,msg when calling an operation with missing needs (instead of an abrupt `KeyError`).
- FEAT(CI): test also on Python-3.8

1.5.3 v2.3.0 (24 Nov 2019, @ankostis): Zoomable SVGs & more op jobs

- FEAT(plot): render Zoomable SVGs in jupyter(lab) notebooks.
- break(netop): rename execution-method `"sequential" --> None`.
- break(netop): move `overwrites_collector` & `method` args from `netop.__call__()` –> cstor
- refact(netop): convert remaining `**kwargs` into named args, tighten up API.

1.5.4 v2.2.0 (20 Nov 2019, @ankostis): enhance OPERATIONS & restruct their modules

- REFACT(src): split module `nodes.py` –> `op.py` + *netop.py* and move `Operation` from `base.py` –> `op.py`, in order to break cycle of *base(op)* <– *net* <– *netop*, and keep utils only in *base.py*.
- ENH(op): allow Operations WITHOUT any NEEDS.
- ENH(op): allow Operation FUNCTIONS to return directly Dictionaries.
- ENH(op): validate function Results against operation *provides*; *jetsam* now includes *results* variables: `results_fn` & `results_op`.
- BREAK(op): drop unused *Operation._after_init()* pickle-hook; use *dill* instead.
- refact(op): convert `Operation._validate()` into a function, to be called by clients wishing to automate operation construction.
- refact(op): replace `**kwargs` with named-args in class:*FunctionalOperation*, because it allowed too wide args, and offered no help to the user.
- REFACT(configs): privatize `network._execution_configs`; expose more config-methods from base package.

1.5.5 v2.1.1 (12 Nov 2019, @ankostis): global configs

- BREAK: drop Python-3.6 compatibility.
- FEAT: Use (possibly multiple) global configurations for all networks, stored in a `contextvars.ContextVar`.
- ENH/BREAK: Use a (possibly) single *execution_pool* in global-configs.
- feat: add *abort* flag in global-configs.
- feat: add *skip_evictions* flag in global-configs.

1.5.6 v2.1.0 (20 Oct 2019, @ankostis): DROP BW-compatible, Restruct modules/API, Plan perfect evictions

The first non pre-release for 2.x train.

- BRAKE API: DROP Operation's `params` - use funtools.partial() instead.
- BRAKE API: DROP Backward-Compatible `Data` & `Operation` classes,
- BRAKE: DROP Pickle workarounds - expected to use `dill` instead.
- break(jetsam): drop "graphtik_' prefix from annotated attribute

- ENH(op): now `operation()` supported the "builder pattern" with *`operation.withset()`*.
- REFACT: renamed internal package *functional* –> *nodes* and moved classes around, to break cycles easier, (`base` works as suppposed to), not to import early everything, but to fail plot early if `pydot` dependency missing.
- REFACT: move PLAN and `compute()` up, from `Network --> NetworkOperation`.
- ENH(NET): new PLAN BULDING algorithm produces PERFECT EVICTIONS, that is, it gradually eliminates from the solution all non-asked outputs.
  - enh: pruning now cleans isolated data.
  - enh: eviction-instructions are inserted due to two different conditions: once for unneeded data in the past, and another for unused produced data (those not belonging typo the pruned dag).
  - enh: discard immediately irrelevant inputs.
- ENH(net): changed results, now unrelated inputs are not included in solution.
- refact(sideffect): store them as node-attributes in DAG, fix their combination with pinning & eviction.
- fix(parallel): eviction was not working due to a typo 65 commits back!

1.5.7 v2.0.0b1 (15 Oct 2019, @ankostis): Rebranded as *Graphtik* for Python 3.6+

Continuation of yahoo#30 as yahoo#31, containing review-fixes in huyng/graphkit#1.

Network

- FIX: multithreaded operations were failing due to shared `ExecutionPlan.executed`.
- FIX: prunning sometimes were inserting plan string in DAG. (not `_DataNode`).
- ENH: heavily reinforced exception annotations ("jetsam"):
  - FIX: (8f3ec3a) outer graphs/ops do not override the inner cause.
  - ENH: retrofitted exception-annotations as a single dictionary, to print it in one shot (8f3ec3a & 8d0de1f)
  - enh: more data in a dictionary
  - TCs: Add thorough TCs (8f3ec3a & b8063e5).
- REFACT: rename *Delete*–>'Evict', removed *Placeholder* from nadanodes, privatize node-classes.
- ENH: collect "jetsam" on errors and annotate exceptions with them.
- ENH(sideffects): make them always DIFFERENT from regular DATA, to allow to co-exist.
- fix(sideffects): typo in add_op() were mixing needs/provides.
- enh: accept a single string as *outputs* when running graphs.

Testing & other code:

- TCs: *pytest* now checks sphinx-site builds without any warnings.
- Established chores with build services:
  - Travis (and auto-deploy to PyPi),
  - codecov

  - ReadTheDocs

1.5.8 v1.3.0 (Oct 2019, @ankostis): NEVER RELEASED: new DAG solver, better plotting & "sideffect"

Kept external API (hopefully) the same, but revamped pruning algorithm and refactored network compute/compile structure, so results may change; significantly enhanced plotting. The only new feature actually is the `sideffect`` modifier.

Network:

- FIX(yahoo#18, yahoo#26, yahoo#29, yahoo#17, yahoo#20): Revamped DAG SOLVER to fix bad pruning described in yahoo#24 & yahoo#25

  Pruning now works by breaking incoming provide-links to any given intermedediate inputs dropping operations with partial inputs or without outputs.

  The end result is that operations in the graph that do not have all inputs satisfied, they are skipped (in v1.2.4 they crashed).

  Also started annotating edges with optional/sideffects, to make proper use of the underlying `networkx` graph.

- REFACT(yahoo#21, yahoo#29): Refactored Network and introduced `ExecutionPlan` to keep compilation results (the old `steps` list, plus input/output names).

  Moved also the check for when to evict a value, from running the execution-plan, to whenbuilding it; thus, execute methods don't need outputs anymore.

- ENH(yahoo#26): "Pin* input values that may be overriten by calculated ones.

  This required the introduction of the new `_PinInstruction` in the execution plan.

- FIX(yahoo#23, yahoo#22-2.4.3): Keep consistent order of `networkx.DiGraph` and *sets*, to generate deterministic solutions.

  Unfortunately, it non-determinism has not been fixed in < PY3.5, just reduced the frequency of spurious failures, caused by unstable dicts, and the use of subgraphs.

- enh: Mark outputs produced by `NetworkOperation`'s needs as `optional`. TODO: subgraph network-operations would not be fully functional until *"optional outpus"* are dealt with (see yahoo#22-2.5).
- enh: Annotate operation exceptions with `ExecutionPlan` to aid debug sessions,
- drop: methods `list_layers()`/`show layers()` not needed, `repr()` is a better replacement.

Plotting:

- ENH(yahoo#13, yahoo#26, yahoo#29): Now network remembers last plan and uses that to overlay graphs with the internals of the planing and execution:
  - execution-steps & order
  - evict & pin instructions
  - given inputs & asked outputs
  - solution values (just if they are present)

  - "optional" needs & broken links during pruning
- REFACT: Move all API doc on plotting in a single module, splitted in 2 phases, build DOT & render DOT
- FIX(yahoo#13): bring plot writing into files up-to-date from PY2; do not create plot-file if given file-extension is not supported.
- FEAT: path pydot library to support rendering in *Jupyter notebooks*.

Testing & other code:

- Increased coverage from 77% –> 90%.
- ENH(yahoo#28): use `pytest`, to facilitate TCs parametrization.
- ENH(yahoo#30): Doctest all code; enabled many assertions that were just print-outs in v1.2.4.
- FIX: `operation.__repr__()` was crashing when not all arguments had been set - a condition frequtnly met during debugging session or failed TCs (inspired by @syamajala's 309338340).
- enh: Sped up parallel/multihtread TCs by reducing delays & repetitions.

> **Tip:** You need `pytest -m slow` to run those slow tests.

Chore & Docs:

- FEAT: add changelog in `CHANGES.rst` file, containing flowcharts to compare versions `v1.2.4 <--> v1.3..0`.
- enh: updated site & documentation for all new features, comparing with v1.2.4.
- enh(yahoo#30): added "API reference' chapter.
- drop(build): `sphinx_rtd_theme` library is the default theme for Sphinx now.
- enh(build): Add `test` *pip extras*.
- sound: https://www.youtube.com/watch?v=-527VazA4IQ, https://www.youtube.com/watch?v=8J182LRi8sU&t=43s

1.5.9 v1.2.4 (Mar 7, 2018)

- Issues in pruning algorithm: yahoo#24, yahoo#25
- Blocking bug in plotting code for Python-3.x.
- Test-cases without assertions (just prints).

1.5.10 1.2.2 (Mar 7, 2018, @huyng): Fixed versioning

Versioning now is manually specified to avoid bug where the version was not being correctly reflected on pip install deployments

1.5.11 1.2.1 (Feb 23, 2018, @huyng): Fixed multi-threading bug and faster compute through caching of *find_necessary_steps*

We've introduced a cache to avoid computing find_necessary_steps multiple times during each inference call.

This has 2 benefits:

- It reduces computation time of the compute call
- It avoids a subtle multi-threading bug in networkx when accessing the graph from a high number of threads.

1.5.12 1.2.0 (Feb 13, 2018, @huyng)

Added *set_execution_method('parallel')* for execution of graphs in parallel.

1.5.13 1.1.0 (Nov 9, 2017, @huyng)

Update setup.py

1.5.14 1.0.4 (Nov 3, 2017, @huyng): Networkx 2.0 compatibility

Minor Bug Fixes:

- Compatibility fix for networkx 2.0
- *net.times* now only stores timing info from the most recent run

1.5.15 1.0.3 (Jan 31, 2017, @huyng): Make plotting dependencies optional

- Merge pull request yahoo#6 from yahoo/plot-optional
- make plotting dependencies optional

1.5.16 1.0.2 (Sep 29, 2016, @pumpikano): Merge pull request yahoo#5 from yahoo/remove-packaging-dep

- Remove 'packaging' as dependency

1.5.17 1.0.1 (Aug 24, 2016)

1.5.18 1.0 (Aug 2, 2016, @robwhess)

First public release in PyPi & GitHub.

- Merge pull request yahoo#3 from robwhess/travis-build
- Travis build

CHAPTER 2

Quick start

Here's how to install:

```
pip install graphtik
```

OR with dependencies for plotting support (and you need to install Graphviz program separately with your OS tools):

```
pip install graphtik[plot]
```

Here's a Python script with an example Graphtik computation graph that produces multiple outputs (`a * b`, `a - a * b`, and `abs(a - a * b) ** 3`):

```
>>> from operator import mul, sub
>>> from functools import partial
>>> from graphtik import compose, operation

# Computes |a|^p.
>>> def abspow(a, p):
...     c = abs(a) ** p
...     return c
```

Compose the `mul`, `sub`, and `abspow` functions into a computation graph:

```
>>> graphop = compose("graphop",
...     operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
...     operation(name="sub1", needs=["a", "ab"], provides=["a_minus_ab"])(sub),
...     operation(name="abspow1", needs=["a_minus_ab"], provides=["abs_a_minus_ab_cubed
↪"])
...     (partial(abspow, p=3))
... )
```

Run the graph-operation and request all of the outputs:

```
>>> graphop(**{'a': 2, 'b': 5})
{'a': 2, 'b': 5, 'ab': 10, 'a_minus_ab': -8, 'abs_a_minus_ab_cubed': 512}
```

Run the graph-operation and request a subset of the outputs:

```
>>> graphop.compute({'a': 2, 'b': 5}, outputs=["a_minus_ab"])
{'a_minus_ab': -8}
```

As you can see, any function can be used as an operation in Graphtik, even ones imported from system modules!

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

V

W