graphtik Documentation

Release src: 5.2.0, git: v5.2.0

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

Feb 28, 2020

Contents

(src: 5.2.0, git: v5.2.0 , Feb 28, 2020)

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 *requirements* 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.**FunctionalOperation**(*fn: Callable, name, needs: Union[Collection[T_co], str, None] = None, provides: Union[Collection[T_co], str, None] = None, aliases: Mapping[KT, VT_co] = None, *, parents: Tuple = None, rescheduled=None, endured=None, parallel=None, marshalled=None, returns_dict=None, node_props: Mapping[KT, VT_co] = None*)

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

provides
Value names this operation provides (including aliases/sideffects).

real_provides
Value names the underlying function provides (without aliases, with(!) sideffects).

FIXME: *real_provides* not sure what it does with sideffects

Tip: Use `operation()` builder class to build instances of this class instead.

__call__(**args*, ***kwargs*)
Call self as a function.

__init__(*fn: Callable, name, needs: Union[Collection[T_co], str, None] = None, provides: Union[Collection[T_co], str, None] = None, aliases: Mapping[KT, VT_co] = None, *, parents: Tuple = None, rescheduled=None, endured=None, parallel=None, marshalled=None, returns_dict=None, node_props: Mapping[KT, VT_co] = None*)
Build a new operation out of some function and its requirements.

Parameters

- **name** – a name for the operation (e.g. *'conv1'*, *'sum'*, etc..); it will be prefixed by *parents*.
- **needs** – Names of input data objects this operation requires.
- **provides** – Names of the **real output values** the underlying function provides (without *aliases*, with(!) sideffects)
- **aliases** – an optional mapping of real *provides* to additional ones, togetherher comprising this operations *provides*.
- **parents** – a tuple wth the names of the parents, prefixing *name*, but also kept for equality/hash check.
- **rescheduled** – If true, underlying *callable* may produce a subset of *provides*, and the *plan* must then *reschedule* after the operation has executed. In that case, it makes more sense for the *callable* to *returns_dict*.
- **endured** – If true, even if *callable* fails, solution will *reschedule*; ignored if *endurance* enabled globally.
- **parallel** – execute in *parallel*
- **marshalled** – If true, operation will be *marshalled* while computed, along with its *inputs* & *outputs*. (usefull when run in *parallel* with a *process pool*).
- **returns_dict** – if true, it means the *fn* returns a dictionary with all *provides*, and no further processing is done on them (i.e. the returned output-values are not zipped with *provides*)
- **node_props** – added as-is into NetworkX graph

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** – the input values with which to feed the computation.

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

1.1.1 The `operation` builder factory

There is a better way to instantiate an `FunctionalOperation` than simply constructing it: use the `operation` builder class:

class graphtik.**operation**(*fn: Callable = None, *, name=None, needs: Union[Collection[T_co], str, None] = None, provides: Union[Collection[T_co], str, None] = None, aliases: Mapping[KT, VT_co] = None, rescheduled=None, endured=None, parallel=None, marshalled=None, returns_dict=None, node_props: Mapping[KT, VT_co] = 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** – Names of input data objects this operation requires. These should correspond to the `args` of `fn`.
- **provides** – 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 case a dictionary with as many elements must be returned
- **aliases** – an optional mapping of *provides* to additional ones
- **rescheduled** – If true, underlying *callable* may produce a subset of *provides*, and the *plan* must then *reschedule* after the operation has executed. In that case, it makes more sense for the *callable* to *returns_dict*.
- **endured** – If true, even if *callable* fails, solution will *reschedule*. ignored if *endurance* enabled globally.
- **parallel** – execute in *parallel*
- **marshalled** – If true, operation will be *marshalled* while computed, along with its *inputs* & *outputs*. (usefull when run in *parallel* with a *process pool*).
- **returns_dict** – if true, it means the *fn* *returns dictionary* with all *provides*, and no further processing is done on them (i.e. the returned output-values are not zipped with *provides*)
- **node_props** – added as-is into NetworkX graph

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')
```

Tip: Remember to call once more the builder class at the end, to get the actual operation instance.

__call__(*fn: Callable = None, *, name=None, needs: Union[Collection[T_co], str, None] = None, provides: Union[Collection[T_co], str, None] = None, aliases: Mapping[KT, VT_co] = None, rescheduled=None, endured=None, parallel=None, marshalled=None, returns_dict=None, node_props: Mapping[KT, VT_co] = None*) → graphtik.op.FunctionalOperation

This enables `operation` to act as a decorator or as a functional operation, for example:

```
@operator(name='myadd1', needs=['a', 'b'], provides=['c'])
def myadd(a, b):
    return a + b
```

or:

```
def myadd(a, b):
    return a + b
operator(name='myadd1', needs=['a', 'b'], provides=['c'])(myadd)
```

Parameters **fn** (*function*) – The function to be used by this `operation`.

Returns Returns an operation class that can be called as a function or composed into a computation graph.

withset(**, fn: Callable = None, name=None, needs: Union[Collection[T_co], str, None] = None, provides: Union[Collection[T_co], str, None] = None, aliases: Mapping[KT, VT_co] = None, rescheduled=None, endured=None, parallel=None, marshalled=None, returns_dict=None, node_props: Mapping[KT, VT_co] = None*) → graphtik.op.operation

See *operation* for arguments here.

1.1.2 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
```

1.1.3 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 constant 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.

1.1.4 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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_op3')

>>> graphop = compose('two_pows_graph', pow_op1, pow_op2, pow_op3)
>>> graphop(a=2)
{'a': 2, 'a_squared': 4, 'a_cubed': 1}
```

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.

1.1.5 Modifiers on *operation needs* and *provides*

Modifiers change the behavior of specific *needs* or *provides*.

The *needs* and *provides* annotated with *modifiers* designate, for instance, *optional* function arguments, or "ghost" *sideffects*.

class graphtik.modifiers.**arg**

Annotate a *needs* to map from its name in the *inputs* to a different argument-name.

Parameters **fn_arg** – The argument-name corresponding to this named-input.

Note: This extra mapping argument is needed either for *optionals* or for functions with keywords-only arguments (like `def func(*, foo, bar): ...`), since *inputs*' are normally fed into functions by-position, not by-name.

Example:

In case the name of the function arguments is different from the name in the *inputs* (or just because the name in the *inputs* is not a valid argument-name), you may *map* it with the 2nd argument of *arg* (or *optional*):

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

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

```
>>> graph = compose('mygraph',
...      operation(name='myadd',
...                needs=['a', arg("name-in-inputs", "b")],
...                provides="sum")(myadd)
... )
>>> graph
NetworkOperation('mygraph', needs=['a', 'name-in-inputs'], provides=['sum'], x1␣
↪ops:
  +--FunctionalOperation(name='myadd',
                         needs=['a',
                         arg('name-in-inputs'-->'b')],
                         provides=['sum'],
                         fn='myadd'))
>>> graph.compute({"a": 5, "name-in-inputs": 4})['sum']
9
```

class graphtik.modifiers.**optional**

Annotate *optionals* *needs* corresponding to *defaulted* op-function arguments, ...

received only if present in the *inputs* (when operation is invocated). The value of an optional is passed as a keyword argument to the underlying function.

Example:

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

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

Annotate `b` as optional argument (and notice it's default value `0`):

```
>>> graph = compose('mygraph',
...     operation(name='myadd',
...               needs=["a", optional("b")],
...               provides="sum")(myadd)
... )
>>> graph
NetworkOperation('mygraph',
                 needs=['a', optional('b')],
                 provides=['sum'],
                 x1 ops:
...
```

The graph works both with and without `c` provided in the inputs:

```
>>> graph(a=5, b=4)['sum']
9
>>> graph(a=5)
{'a': 5, 'sum': 5}
```

class graphtik.modifiers.**sideffect**

sideffects dependencies participates in the graph but not exchanged with functions.

Both *needs* & *provides* may be designated as *sideffects* using this modifier. They work as usual while solving the graph (*compilation*) but they do not interact with the *operation*'s function; specifically:

- input sideffects must exist in the *inputs* for an operation to kick-in;
- input sideffects are NOT fed into the function;
- output sideffects are NOT expected from the function;
- output sideffects are stored in the *solution*.

Their purpose is to describe operations that modify the internal state of some of their arguments ("side-effects").

Example:

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('mygraph', needs=['df', 'sideffect(df.b)'],
                 provides=['sideffect(df.sum)'], x1 ops:
  +--FunctionalOperation(name='addcolumns', needs=['df', 'sideffect(df.b)'],␣
↪provides=['sideffect(df.sum)'], fn='addcolumns'))
```

```
>>> df = pd.DataFrame({'a': [5, 0], 'b': [2, 1]})    # doctest: +SKIP
>>> graph({'df': df})['df']                          # doctest: +SKIP
        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']    # doctest: +SKIP
        a       b       sum
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, if requested in *outputs*.

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*, like:

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

---

class graphtik.modifiers.**vararg**

Annotate *optionals* *needs* to be fed as op-function's `*args` when present in inputs.

See also:

Consult also the example test-case in: `test/test_op.py:test_varargs()`, in the full sources of the project.

Example:

```
>>> 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('mygraph',
                 needs=['a', optional('b'), optional('c')],
                 provides=['sum'],
                 x1 ops:
  +--FunctionalOperation(name='addall', needs=['a', vararg('b'), vararg('c')],␣
↪provides=['sum'], fn='addall'))
```

The graph works with and without any of `b` or `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**

Like *vararg*, naming an *optional* *iterable* value in the inputs.

See also:

Consult also the example test-case in: `test/test_op.py:test_varargs()`, in the full sources of the project.

Example:

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

```
>>> def enlist(a, *b):
...     return [a] + list(b)
```

```
>>> graph = compose('mygraph',
...     operation(name='enlist', needs=['a', varargs('b')],
...     provides='sum')(enlist)
... )
>>> graph
NetworkOperation('mygraph',
                 needs=['a', optional('b')],
                 provides=['sum'],
                 x1 ops:
  +--FunctionalOperation(name='enlist', needs=['a', varargs('b')], provides=['sum
↪'], fn='enlist'))
```

The graph works with or without *b* in the inputs:

```
>>> graph(a=5, b=[2, 20])['sum']
[5, 2, 20]
>>> graph(a=5)
```

(continues on next page)

(continued from previous page)

```
{'a': 5, 'sum': [5]}
>>> graph(a=5, b=0xBAD)
Traceback (most recent call last):
...
graphtik.base.MultiValueError: Failed preparing needs:
    1. Expected needs[varargs('b')] to be non-str iterables!
    +++inputs: {'a': 5, 'b': 2989}
    +++FunctionalOperation(name='enlist', needs=['a', varargs('b')], provides=[
↪'sum'], fn='enlist')
```

Attention: To avoid user mistakes, it does not accept strings (though iterables):

```
>>> graph(a=5, b="mistake")
Traceback (most recent call last):
...
graphtik.base.MultiValueError: Failed preparing needs:
    1. Expected needs[varargs('b')] to be non-str iterables!
    +++inputs: {'a': 5, 'b': 'mistake'}
    +++FunctionalOperation(name='enlist', needs=['a', varargs('b')], provides=[
↪'sum'], fn='enlist')
```

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, outputs: Union[Collection[T_co], str, None] = None, rescheduled=None, endured=None, parallel=None, marshalled=None, merge=False, node_props=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).

- **rescheduled** – applies *rescheduled* to all contained *operations*
- **endured** – applies *endurance* to all contained *operations*
- **parallel** – mark all contained *operations* to be executed in *parallel*
- **marshalled** – mark all contained *operations* to be *marshalled* (usefull when run in *parallel* with a *process pool*).
- **node_props** – added as-is into NetworkX graph, to provide for filtering by `NetworkOperation.withset()`.

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('merged_graph',
                 needs=['a', 'b', 'ab', 'a_minus_ab', 'c'],
                 provides=['ab', 'a_minus_ab', 'abs_a_minus_ab_cubed', 'cab'],
                 x4 ops:
...
```

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=solution)      # annotate graph with solution values
```

... or for the last ...:

```
solution.plot(...)
```

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

The same *Plotter.plot()* method applies also for:

- *NetworkOperation*
- *Network*
- *ExecutionPlan*
- *Solution*

each one capable to producing diagrams with increasing complexity. Whenever possible, the top-level `plot()` methods will delegate to the ones below; specifically, the *netop* keeps a transient reference to the last *plan*. BUT the *plan* does not hold such a reference, you have to plot the *solution*.

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 the network:

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

If you want all details, plot the solution:

```
solution.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 following 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:
...     pprint(ex.jetsam)
{'aliases': None,
 'args': {'kwargs': {}, 'positional': [None], 'varargs': []},
 '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'], x1 steps:
  +--FunctionalOperation(name='screamer', needs=['a'], provides=['foo'], fn='scream
↪')),
 'provides': None,
 'results_fn': None,
 'results_op': None,
 'solution': {'a': None},
 'task': OpTask(FunctionalOperation(name='screamer', needs=['a'], provides=['foo'],␣
↪fn='scream'), sol_keys=['a'])}
```

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 meaningful 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 function, if any

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

solution an instance of *Solution*, contains *inputs* & *outputs* till the error happened; note that *Solution.executed* contain the list of executed *operations* so far.

Of course 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 incomplete, you may not always avoid one.

1.3.3 Execution internals

Compile & *execute* network graphs of operations.

1.4 Architecture

COMPUTE

computation

The definition & execution of networked operation is split in 1+2 phases:

- *COMPOSITION*
- *COMPILATION*
- *EXECUTION*

... it is constrained by these IO data-structures:

- *operation*(s) (with *needs* & *provides* for each one)
- given *inputs*
- asked *outputs*

... populates these low-level data-structures:

- *network graph* (COMPOSE time)
- *execution dag* (COMPILE time)
- *execution steps* (COMPILE time)
- *solution* (EXECUTE time)

... and utilizes these main classes:

graphtik.op.FunctionalOperation
graphtik.netop.NetworkOperation
graphtik.network.Network
graphtik.network.ExecutionPlan
graphtik.network.Solution

compose

COMPOSITION The *phase* where *operation*s are constructed and grouped into *netop*s and corresponding *network*s.

Tip:

- Use *operation()* builder class to construct FunctionalOperation instances.
- Use *compose()* factory to prepare the *net* internally, and build *NetworkOperation* instances.

compile

COMPILATION The *phase* where the *Network* creates a new *execution plan* by *pruning* all *graph* nodes into a subgraph *dag*, and deriving the *execution steps*.

execute

EXECUTION

sequential The *phase* where the *ExecutionPlan* calls the underlying functions of all *operation*s contained in *execution steps*, with *inputs*/*outputs* taken from the *solution*.

Currently there are 2 ways to execute:

- *sequential*
- *parallel*, with a multiprocessing.ProcessPool

Plans may abort their execution by setting the *abort run* global flag.

parallel

parallel execution

execution pool

task *Execute operation*s *in parallel*, with a *thread pool* or *process pool* (instead of *sequential*). Operations and *netop* are marked as such on construction, or enabled globally from *configurations*.

Note that *sideffects* are not expected to function with *process pools*, certainly not when *marshalling* is enabled.

process pool When the multiprocessing.Pool() class is used for *parallel* execution, the *task*s must be communicated to/from the worker process, which requires pickling, and that may fail. With pickling failures you may try *marshalling* with *dill* library, and see if that helps.

Note that *sideffects* are not expected to function at all. certainly not when *marshalling* is enabled.

thread pool When the multiprocessing.dummy.Pool() class for *parallel* execution, the *task*s are run *in process*, so no *marshalling* is needed.

marshalling Pickling *parallel operation*s and their *inputs*/*outputs* using the dill module. It is *configured* either globally with *set_marshal_tasks()* or set with a flag on each operation / *netop*.

Note that *sideffects* do not work when this is enabled.

configurations The functions controlling *compile* & *execution* globally are defined in *config* module; they underlying global data are stored in contextvars.ContextVar instances, to allow for nested control.

All *boolean* configuration flags are **tri-state** (None, False, True), allowing to "force" all operations, when they are not set to the None value. All of them default to None (false).

graph

network graph The Network.graph (currently a DAG) contains all FunctionalOperation and _DataNode nodes of some *netop*.

They are layed out and connected by repeated calls of *Network._append_operation()* by Network constructor.

This graph is then *pruned* to extract the *dag*, and the *execution steps* are calculated, all ingredients for a new `ExecutionPlan`.

dag

execution dag

solution dag There are 2 *directed-acyclic-graphs* instances used:

- the `ExecutionPlan.dag`, in the *execution plan*, which contains the *pruned* nodes, used to decide the *execution steps*;
- the `Solution.dag` in the *solution*, which derives the *canceled operations* due to *rescheduled*/failed operations upstream.

steps

execution steps The `ExecutionPlan.steps` contains a list of the operation-nodes only from the *dag*, topologically sorted, and interspersed with *instruction steps* needed to *compute* the asked *outputs* from the given *inputs*.

It is built by `Network._build_execution_steps()` based on the subgraph *dag*.

The only *instruction* step is for performing *evictions*.

evictions The `_EvictInstruction` *steps* erase items from *solution* as soon as they are not needed further down the dag, to reduce memory footprint while computing.

solution A `Solution` instance created internally by `NetworkOperation.compute()` to hold the values both *inputs* & *outputs*, and the status of *executed* operations. It is based on a `collections.ChainMap`, to keep one dictionary for each *operation* executed +1 for inputs.

The results of the last operation executed "wins" in the final *outputs* produced, BUT while executing, the *needs* of each operation receive the *solution* values in **reversed order**, that is, the 1st operation result (or given input) wins for some *needs* name.

Rational:

> During execution we want stability (the same input value used by all operations), and that is most important when consuming input values - otherwise, we would use (possibly *overwritten* and thus changing)) intermediate ones.
>
> But at the end we want to affect the calculation results by adding operations into some *netop* - furthermore, it wouldn't be very useful to get back the given inputs in case of *overwrites*.

overwrites Values in the *solution* that have been written by more than one *operations*, accessed by `Solution.overwrites`:

net

network the `Network` contains a *graph* of *operations* and can *compile* an *execution plan* or *prune* a cloned *network* for given *inputs*/*outputs*/*node predicate*.

plan

execution plan Class `ExecutionPlan` perform the *execution* phase which contains the *dag* and the *steps*.

Compiled *execution plans* are cached in `Network._cached_plans` across runs with (*inputs*, *outputs*, *predicate*) as key.

inputs The named input values that are fed into an *operation* (or *netop*) through `Operation.compute()` method according to its *needs*.

These values are either:

- given by the user to the outer *netop*, at the start of a *computation*, or

- derived from *solution* using *needs* as keys, during intermediate *execution*.

outputs The dictionary of computed values returned by an *operation* (or a *netop*) matching its *provides*, when method `Operation.compute()` is called.

Those values are either:

- retained in the *solution*, internally during *execution*, keyed by the respective *provide*, or
- returned to user after the outer *netop* has finished *computation*.

When no specific outputs requested from a *netop*, the `NetworkOperation.compute()` method returns all intermediate *inputs* along with the *outputs*, that is, no *evictions* happens.

An *operation* may return *partial outputs*.

returns dictionary When an operation is marked with this flag, the underlying function is not expected to return a sequence but a dictionary; hence, no "zipping" of outputs/provides takes place.

operation Either the abstract notion of an action with specified *needs* and *provides*, or the concrete wrapper `FunctionalOperation` for arbitrary functions (any `callable`).

netop

network operation The `NetworkOperation` class holding a *network* of *operation*s.

needs A list of names of the compulsory/*optionals* values or *sideffects* an *operation*'s underlying callable requires to exist in the *inputs* during *execution*.

provides A list of names for the values produced when the *operation*'s underlying callable executes, to be stored into the *solution* during *execution*.

modifiers Annotations on specific arguments of *needs* and/or *provides* such as *optionals* & *sideffects* (see `graphtik.modifiers` module).

optionals *Needs* corresponding either:

- to function arguments-with-defaults (annotated with `optional`), or
- to `*args` (annotated with `vararg` & `varargs`),

that do not hinder execution of the *operation* if absent from *inputs*.

sideffects Fictive *needs* or *provides* not consumed/produced by the underlying function of an *operation*, annotated with `sideffect`. A *sideffect* participates in the *compilation* of the graph, and is updated into the *solution*, but is never given/asked to/from functions.

prune

pruning A subphase of *compilation* performed by method `Network._prune_graph()`, which extracts a subgraph *dag* that does not contain any *unsatisfied operation*s.

It topologically sorts the *graph*, and *prunes* based on given *inputs*, asked *outputs*, *node predicate* and *operation needs* & *provides*.

unsatisfied operation The core of *pruning* & *rescheduling*, performed by `network._unsatisfied_operations()` function, which collects all *operation*s that fall into any of these 2 cases:

- they have *needs* that do not correspond to any of the given *inputs* or the intermediately *compute*d *outputs* of the *solution*;
- all their *provides* are NOT needed by any other operation, nor are asked as *outputs*.

reschedule

rescheduling

partial outputs

partial operation

canceled operation The partial *pruning* of the *solution*'s dag during *execution*. It happens when any of these 2 conditions apply:

- an *operation* is marked with the `FunctionalOperation.rescheduled` attribute, which means that its underlying *callable* may produce only a subset of its *provides* (*partial outputs*);
- *endurance* is enabled, either globally (in the *configurations*), or for a specific *operation*.

the *solution* must then *reschedule* the remaining operations downstream, and possibly *cancel* some of those (assigned in `Solution.canceled`).

endurance Keep executing as many *operation*s as possible, even if some of them fail. Endurance for an operation is enabled if `set_endure_operations()` is true globally in the *configurations* or if `FunctionalOperation.endurance` is true.

You may interrogate `Solution.executed` to discover the status of each executed operations or call `scream_if_incomplete()`.

predicate

node predicate A callable(op, node-data) that should return true for nodes to be included in *graph* during *compilation*.

abort run A global *configurations* flag that when set with `abort_run()` function, it halts the execution of all currently or future *plan*s.

It is reset automatically on every call of `NetworkOperation.compute()` (after a successful intermediate *compilation*), or manually, by calling `reset_abort()`.

1.5 API Reference

`graphtik`	Lightweight *computation* graphs for Python.
`graphtik.op`	About *operation* nodes (but not net-ops to break cycle).
`graphtik.modifiers`	*Modifiers* change the behavior of specific *needs* or *provides*.
`graphtik.netop`	About *network operation*s (those based on graphs)
`graphtik.network`	*Compile* & *execute* network graphs of operations.
`graphtik.plot`	
`graphtik.config`	*Configurations* for network execution, and utilities on them.
`graphtik.base`	Generic or specific utilities

1.5.1 Module: *op*

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

class graphtik.op.**Operation**
An abstract class representing an action with `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** – the input values with which to feed the computation.

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

`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 early.

1.5.2 Module: *modifiers*

Modifiers change the behavior of specific *needs* or *provides*.

The *needs* and *provides* annotated with *modifiers* designate, for instance, *optional* function arguments, or "ghost" *sideffects*.

class `graphtik.modifiers.`**`arg`**
Annotate a *needs* to map from its name in the *inputs* to a different argument-name.

Parameters **`fn_arg`** – The argument-name corresponding to this named-input.

Note: This extra mapping argument is needed either for *optionals* or for functions with keywords-only arguments (like `def func(*, foo, bar): ...`), since *inputs*' are normally fed into functions by-position, not by-name.

Example:

In case the name of the function arguments is different from the name in the *inputs* (or just because the name in the *inputs* is not a valid argument-name), you may *map* it with the 2nd argument of *arg* (or *optional*):

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

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

```
>>> graph = compose('mygraph',
...      operation(name='myadd',
...                needs=['a', arg("name-in-inputs", "b")],
...                provides="sum")(myadd)
... )
>>> graph
NetworkOperation('mygraph', needs=['a', 'name-in-inputs'], provides=['sum'], x1␣
↪ops:
  +--FunctionalOperation(name='myadd',
                         needs=['a',
                         arg('name-in-inputs'-->'b')],
                         provides=['sum'],
                         fn='myadd'))
>>> graph.compute({"a": 5, "name-in-inputs": 4})['sum']
9
```

class `graphtik.modifiers.`**`optional`**
Annotate *optionals* *needs* corresponding to *defaulted* op-function arguments, ...

received only if present in the *inputs* (when operation is invocated). The value of an optional is passed as a keyword argument to the underlying function.

Example:

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

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

Annotate `b` as optional argument (and notice it's default value `0`):

```
>>> graph = compose('mygraph',
...     operation(name='myadd',
...                needs=["a", optional("b")],
...                provides="sum")(myadd)
... )
>>> graph
NetworkOperation('mygraph',
                 needs=['a', optional('b')],
                 provides=['sum'],
                 x1 ops:
...
```

The graph works both with and without `c` provided in the inputs:

```
>>> graph(a=5, b=4)['sum']
9
>>> graph(a=5)
{'a': 5, 'sum': 5}
```

class graphtik.modifiers.**sideffect**

sideffects dependencies participates in the graph but not exchanged with functions.

Both *needs* & *provides* may be designated as *sideffects* using this modifier. They work as usual while solving the graph (*compilation*) but they do not interact with the *operation*'s function; specifically:

- input sideffects must exist in the *inputs* for an operation to kick-in;
- input sideffects are NOT fed into the function;
- output sideffects are NOT expected from the function;
- output sideffects are stored in the *solution*.

Their purpose is to describe operations that modify the internal state of some of their arguments ("side-effects").

Example:

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)
```

(continues on next page)

(continued from previous page)

```
... )
>>> graph
NetworkOperation('mygraph', needs=['df', 'sideffect(df.b)'],
                 provides=['sideffect(df.sum)'], x1 ops:
  +--FunctionalOperation(name='addcolumns', needs=['df', 'sideffect(df.b)'],␣
↪provides=['sideffect(df.sum)'], fn='addcolumns'))
```

```
>>> df = pd.DataFrame({'a': [5, 0], 'b': [2, 1]})    # doctest: +SKIP
>>> graph({'df': df})['df']                          # doctest: +SKIP
        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']    # doctest: +SKIP
        a       b       sum
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, if requested in *outputs*.

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*, like:

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

---

class graphtik.modifiers.**vararg**

Annotate *optionals* *needs* to be fed as op-function's `*args` when present in inputs.

See also:

Consult also the example test-case in: `test/test_op.py:test_varargs()`, in the full sources of the project.

Example:

```
>>> 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('mygraph',
                 needs=['a', optional('b'), optional('c')],
                 provides=['sum'],
                 x1 ops:
  +--FunctionalOperation(name='addall', needs=['a', vararg('b'), vararg('c')],␣
↪provides=['sum'], fn='addall'))
```

The graph works with and without any of `b` or `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**

Like *vararg*, naming an *optional iterable* value in the inputs.

See also:

Consult also the example test-case in: `test/test_op.py:test_varargs()`, in the full sources of the project.

Example:

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

```
>>> def enlist(a, *b):
...     return [a] + list(b)
```

```
>>> graph = compose('mygraph',
...     operation(name='enlist', needs=['a', varargs('b')],
...     provides='sum')(enlist)
... )
>>> graph
NetworkOperation('mygraph',
                 needs=['a', optional('b')],
                 provides=['sum'],
                 x1 ops:
  +--FunctionalOperation(name='enlist', needs=['a', varargs('b')], provides=['sum
↪'], fn='enlist'))
```

(continues on next page)

(continued from previous page)

The graph works with or without *b* in the inputs:

```
>>> graph(a=5, b=[2, 20])['sum']
[5, 2, 20]
>>> graph(a=5)
{'a': 5, 'sum': [5]}
>>> graph(a=5, b=0xBAD)
Traceback (most recent call last):
...
graphtik.base.MultiValueError: Failed preparing needs:
    1. Expected needs[varargs('b')] to be non-str iterables!
    +++inputs: {'a': 5, 'b': 2989}
    +++FunctionalOperation(name='enlist', needs=['a', varargs('b')], provides=[
↪'sum'], fn='enlist')
```

Attention: To avoid user mistakes, it does not accept strings (though iterables):

```
>>> graph(a=5, b="mistake")
Traceback (most recent call last):
...
graphtik.base.MultiValueError: Failed preparing needs:
    1. Expected needs[varargs('b')] to be non-str iterables!
    +++inputs: {'a': 5, 'b': 'mistake'}
    +++FunctionalOperation(name='enlist', needs=['a', varargs('b')], provides=[
↪'sum'], fn='enlist')
```

1.5.3 Module: *netop*

About *network operation*s (those based on graphs)

class graphtik.netop.**NetworkOperation**(*operations*, *name*, ***, *outputs=None*, *predicate: Callable[[Any, Mapping[KT, VT_co]], bool] = None*, *rescheduled=None*, *endured=None*, *parallel=None*, *marshalled=None*, *merge=None*, *node_props=None*)

An operation that can *compute* a network-graph of operations.

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

compile(*inputs=None*, *outputs=<UNSET>*, *predicate: Callable[[Any, Mapping[KT, VT_co]], bool] = <UNSET>*) → graphtik.network.ExecutionPlan

Produce a *plan* for the given args or *outputs*/*predicate* narrowed earlier.

Parameters

- **named_inputs** – a string or a list of strings that should be fed to the *needs* of all operations.
- **outputs** – A string or a list of strings with all data asked to compute. If `None`, all possible intermediate outputs will be kept. If not given, those set by a previous call to `withset()` or cstor are used.

- **predicate** – Will be stored and applied on the next *compute()* or *compile()*. If not given, those set by a previous call to *withset()* or cstor are used.

Returns the *execution plan* satisfying the given *inputs*, *outputs* & *predicate*

Raises **ValueError** –

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

  Unknown output nodes: ...

- If solution does not contain any operations, with msg:

  Unsolvable graph: ...

- If given *inputs* mismatched plan's `needs`, with msg:

  Plan needs more inputs...

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

  Impossible outputs...

compute(*named_inputs: Mapping[KT, VT_co], outputs: Union[Collection[T_co], str, None] = <UNSET>, predicate: Callable[[Any, Mapping[KT, VT_co]], bool] = <UNSET>*) → graphtik.network.Solution

Compile a plan & *execute* the graph, sequentially or parallel.

> **Attention:** If intermediate *compilation* is successful, the "global *abort run* flag is reset before the *execution* starts.

Parameters

- **named_inputs** – A maping of names –> values that will be fed to the *needs* of all operations. Cloned, not modified.
- **outputs** – A string or a list of strings with all data asked to compute. If `None`, all intermediate data will be kept.

Returns The *solution* which contains the results of each operation executed +1 for inputs in separate dictionaries.

Raises **ValueError** –

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

  Unknown output nodes: ...

- If plan does not contain any operations, with msg:

  Unsolvable graph: ...

- If given *inputs* mismatched plan's `needs`, with msg:

  Plan needs more inputs...

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

  Impossible outputs...

See also *Operation.compute()*.

last_plan = None

The execution_plan of the last call to compute(), stored as debugging aid.

name = None
The name for the new netop, used when nesting them.

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

predicate = None
The *node predicate* is a 2-argument callable(op, node-data) that should return true for nodes to include; if None, all nodes included.

withset(*outputs: Union[Collection[T_co], str, None] = <UNSET>, predicate: Callable[[Any, Mapping[KT, VT_co]], bool] = <UNSET>, *, name=None, rescheduled=None, endured=None, parallel=None, marshalled=None*) → graphtik.netop.NetworkOperation
Return a copy with a network pruned for the given *needs* & *provides*.

Parameters

- **outputs** – Will be stored and applied on the next *compute()* or *compile()*. If not given, the value of this instance is conveyed to the clone.
- **predicate** – Will be stored and applied on the next *compute()* or *compile()*. If not given, the value of this instance is conveyed to the clone.
- **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.
- **rescheduled** – applies *reschedule*d to all contained *operations*
- **endured** – applies *endurance* to all contained *operations*
- **parallel** – mark all contained *operations* to be executed in *parallel*
- **marshalled** – mark all contained *operations* to be *marshalled* (usefull when run in *parallel* with a *process pool*).

Returns A narrowed netop clone, which **MIGHT be empty!***

Raises **ValueError** –

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

  Unknown output nodes: ...

graphtik.netop.**compose**(*name, op1, *operations, outputs: Union[Collection[T_co], str, None] = None, rescheduled=None, endured=None, parallel=None, marshalled=None, merge=False, node_props=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).
- **rescheduled** – applies *reschedule*d to all contained *operations*
- **endured** – applies *endurance* to all contained *operations*
- **parallel** – mark all contained *operations* to be executed in *parallel*
- **marshalled** – mark all contained *operations* to be *marshalled* (usefull when run in *parallel* with a *process pool*).
- **node_props** – added as-is into NetworkX graph, to provide for filtering by *NetworkOperation.withset()*.

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*.

1.5.4 Module: *network*

Compile & *execute* network graphs of operations.

exception graphtik.network.**AbortedException**

Raised from Network when `abort_run()` is called, and contains the solution ...

with any values populated so far.

__module__ = 'graphtik.network'

__weakref__

list of weak references to the object (if defined)

class graphtik.network.**ExecutionPlan**

A pre-compiled list of operation steps that can *execute* for the given inputs/outputs.

It is the result of the network's *compilation* phase.

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

net

The parent *Network*

needs

An `iset` with the input names needed to exist in order to produce all *provides*.

provides

An `iset` with the outputs names produces when all *inputs* are given.

dag

The regular (not broken) *pruned* subgraph of net-graph.

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.

__abstractmethods__ = frozenset()

__dict__ = mappingproxy({'__module__': 'graphtik.network', '__doc__': "\n A pre-comp

__module__ = 'graphtik.network'

__repr__()
Return a nicely formatted representation string

_abc_impl = <_abc_data object>

_build_pydot(***kws*)

_check_if_aborted(*solution*)

_execute_sequential_method(*solution: graphtik.network.Solution*)
This method runs the graph one operation at a time in a single thread

Parameters **solution** – must contain the input values only, gets modified

_execute_thread_pool_barrier_method(*solution: graphtik.network.Solution*)
This method runs the graph using a parallel pool of thread executors. You may achieve lower total latency if your graph is sufficiently sub divided into operations using this method.

Parameters **solution** – must contain the input values only, gets modified

_handle_task(*future*, *op*, *solution*) → None
Un-dill parallel task results (if marshalled), and update solution / handle failure.

_prepare_tasks(*operations*, *solution*, *pool*, *global_parallel*, *global_marshal*) → Union[Future, graphtik.network._OpTask, bytes]
Combine ops+inputs, apply *marshalling*, and submit to *execution pool* (or not) ...

based on global/pre-op configs.

execute(*named_inputs*, *outputs=None*, *, *name=''*) → graphtik.network.Solution

Parameters

- **named_inputs** – 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** – If not None, they are just checked if possible, based on `provides`, and scream if not.

Returns The *solution* which contains the results of each operation executed +1 for inputs in separate dictionaries.

Raises **ValueError** –

- If plan does not contain any operations, with msg:

  Unsolvable graph: ...

- If given *inputs* mismatched plan's `needs`, with msg:

  Plan needs more inputs...

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

  Impossible outputs...

validate(*inputs: Union[Collection[T_co], str, None]*, *outputs: Union[Collection[T_co], str, None]*)
Scream on invalid inputs, outputs or no operations in graph.

Raises `ValueError` –

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

  Unsolvable graph: ...

- If given *inputs* mismatched plan's `needs`, with msg:

  Plan needs more inputs...

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

  Impossible outputs...

exception graphtik.network.**IncompleteExecutionError**

Raised by `scream_if_incomplete()` when *netop* operations were canceled/failed.

The exception contains 3 arguments:

1. the causal errors and conditions (1st arg),
2. the list of collected exceptions (2nd arg), and
3. the solution instance (3rd argument), to interrogate for more.

__module__ = 'graphtik.network'

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

class graphtik.network.**Network**(**operations*, *graph=None*)

A graph of operations that can *compile* an execution plan.

needs

the "base", all data-nodes that are not produced by some operation

provides

the "base", all data-nodes produced by some operation

__abstractmethods__ = frozenset()

__init__(**operations*, *graph=None*)

Parameters

- **operations** – to be added in the graph
- **graph** – if None, create a new.

Raises `ValueError` – if dupe operation, with msg:

Operations may only be added once, ...

__module__ = 'graphtik.network'

__repr__()

Return repr(self).

_abc_impl = <_abc_data object>

_append_operation(*graph*, *operation: graphtik.op.Operation*)

Adds the given operation and its data requirements to the network graph.

- Invoked during constructor only (immutability).

- Identities are based on the name of the operation, the names of the operation's needs, and the names of the data it provides.
- Adds needs, operation & provides, in that order.

Parameters

- **graph** – the *networkx* graph to append to
- **operation** – operation instance to append

_apply_graph_predicate(*graph*, *predicate*)

_build_execution_steps(*pruned_dag*, *inputs: Collection[T_co]*, *outputs: Optional[Collection[T_co]]*) → List[T]

Create the list of operation-nodes & *instructions* evaluating all

operations & instructions needed a) to free memory and b) avoid overwritting given intermediate inputs.

Parameters

- **pruned_dag** – The original dag, pruned; not broken.
- **outputs** – outp-names to decide whether to add (and which) evict-instructions

Instances of `_EvictInstructions` are inserted in *steps* between operation nodes to reduce the memory footprint of solutions while the computation is running. An evict-instruction is inserted whenever a *need* is not used by any other *operation* further down the DAG.

_build_pydot(***kws*)

_cached_plans = None

Speed up `compile()` call and avoid a multithreading issue(?) that is occuring when accessing the dag in networkx.

_prune_graph(*inputs: Union[Collection[T_co], str, None]*, *outputs: Union[Collection[T_co], str, None]*, *predicate: Callable[[Any, Mapping[KT, VT_co]], bool] = None*) → Tuple[<sphinx.ext.autodoc.importer._MockObject object at 0x7f20fcd01160>, Collection[T_co], Collection[T_co], Collection[T_co]]

Determines what graph steps need to run to get to the requested outputs from the provided inputs: - Eliminate steps that are not on a path arriving to requested outputs; - Eliminate unsatisfied operations: partial inputs or no outputs needed; - consolidate the list of needs & provides.

Parameters

- **inputs** – The names of all given inputs.
- **outputs** – The desired output names. This can also be `None`, in which case the necessary steps are all graph nodes that are reachable from the provided inputs.
- **predicate** – the *node predicate* is a 2-argument callable(op, node-data) that should return true for nodes to include; if None, all nodes included.

Returns

a 3-tuple with the *pruned_dag* & the needs/provides resolved based on the given inputs/outputs (which might be a subset of all needs/outputs of the returned graph).

Use the returned *needs/provides* to build a new plan.

Raises **ValueError** –

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

  Unknown output nodes: ...

_topo_sort_nodes(*dag*) → List[T]
Topo-sort dag respecting operation-insertion order to break ties.

compile(*inputs: Union[Collection[T_co], str, None] = None, outputs: Union[Collection[T_co], str, None] = None, predicate=None*) → graphtik.network.ExecutionPlan
Create or get from cache an execution-plan for the given inputs/outputs.

See *_prune_graph()* and *_build_execution_steps()* for detailed description.

Parameters

- **inputs** – A collection with the names of all the given inputs. If *None*', all inputs that lead to given *outputs* are assumed. If string, it is converted to a single-element collection.
- **outputs** – A collection or the name of the output name(s). If *None*', all reachable nodes from the given *inputs* are assumed. If string, it is converted to a single-element collection.
- **predicate** – the *node predicate* is a 2-argument callable(op, node-data) that should return true for nodes to include; if None, all nodes included.

Returns the cached or fresh new *execution plan*

Raises **ValueError** –

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

  Unknown output nodes: ...

- If solution does not contain any operations, with msg:

  Unsolvable graph: ...

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

  Plan needs more inputs...

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

  Impossible outputs...

class graphtik.network.**Solution**(*plan*, *input_values*)
Collects outputs from operations, preserving *overwrites*.

plan
the plan that produced this solution

executed
A dictionary with keys the operations executed, and values their status:

- no key: not executed yet
- value None: execution ok
- value Exception: execution failed

canceled
A sorted set of *canceled operation*s due to upstream failures.

finalized
a flag denoting that this instance cannot accept more results (after the *finalized()* has been invoked)

__abstractmethods__ = frozenset()

__delitem__(*key*)

__init__(*plan*, *input_values*)
Initialize a ChainMap by setting *maps* to the given mappings. If no mappings are provided, a single empty dictionary is used.

__module__ = 'graphtik.network'

__repr__()
Return repr(self).

_abc_impl = <_abc_data object>

_build_pydot(***kws*)
delegate to network

finalize()
invoked only once, after all ops have been executed

is_failed(*op*)

operation_executed(*op*, *outputs*)
Invoked once per operation, with its results.

It will update *executed* with the operation status and if *outputs* were partials, it will update *canceled* with the unsatisfied ops downstream of *op*.

Parameters

- **op** – the operation that completed ok
- **outputs** – The names of the *outputs* values the op‘ actually produced, which may be a subset of its *provides*. Sideffects are not considered.

operation_failed(*op*, *ex*)
Invoked once per operation, with its results.

It will update *executed* with the operation status and the *canceled* with the unsatisfied ops downstream of *op*.

overwrites
The data in the solution that exist more than once.

A “virtual” property to a dictionary with keys the names of values that exist more than once, and values, all those values in a list, ordered:

- before `finished()`, as computed;
- after `finished()`, in reverse.

scream_if_incomplete()
Raise a *IncompleteExecutionError* when *netop* operations failed/canceled.

class graphtik.network.**_DataNode**
Dag node naming a data-value produced or required by an operation.

__module__ = 'graphtik.network'

__repr__()
Return repr(self).

__slots__ = ()

class graphtik.network.**_EvictInstruction**
A step in the ExecutionPlan to evict a computed value from the *solution*.

It's a step in *ExecutionPlan.steps* for the data-node *str* that frees its data-value from *solution* after it is no longer needed, to reduce memory footprint while computing the graph.

__module__ = 'graphtik.network'

__repr__()
Return repr(self).

__slots__ = ()

class graphtik.network.**_OpTask**(*op*, *sol*, *solid*)
Mimic concurrent.futures.Future for *sequential* execution.

This intermediate class is needed to solve pickling issue with process executor.

__call__()
Call self as a function.

__init__(*op*, *sol*, *solid*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'graphtik.network'

__repr__()
Return repr(self).

__slots__ = ('op', 'sol', 'result', 'solid')

get()
Call self as a function.

logname = 'graphtik.network'

marshalled()

op

result

sol

solid

graphtik.network.**_do_task**(*task*)
Un-dill the *simpler* *_OpTask* & Dill the results, to pass through pool-processes.

See https://stackoverflow.com/a/24673524/548792

graphtik.network.**_optionalized**(*graph*, *data*)
Retain optionality of a *data* node based on all *needs* edges.

graphtik.network.**_unsatisfied_operations**(*dag, inputs: Collection[T_co]*) → List[T]
Traverse topologically sorted dag to collect un-satisfied operations.

Unsatisfied operations are those suffering from ANY of the following:

- **They are missing at least one compulsory need-input.** Since the dag is ordered, as soon as we're on an operation, all its needs have been accounted, so we can get its satisfaction.
- **Their provided outputs are not linked to any data in the dag.** An operation might not have any output link when _prune_graph() has broken them, due to given intermediate inputs.

Parameters

- **dag** – a graph with broken edges those arriving to existing inputs
- **inputs** – an iterable of the names of the input values

Returns a list of unsatisfied operations to prune

graphtik.network.**_yield_datanodes**(*nodes*)
May scan dag nodes.

graphtik.network.**collect_requirements**(*graph*) → Tuple[<sphinx.ext.autodoc.importer._MockObject object at 0x7f20fcf746a0>, <sphinx.ext.autodoc.importer._MockObject object at 0x7f20fcf746a0>]
Collect & split datanodes in (possibly overlapping) *needs/provides*.

graphtik.network.**is_endure_operations**() → Optional[bool]

graphtik.network.**is_marshal_tasks**() → Optional[bool]

graphtik.network.**is_parallel_tasks**() → Optional[bool]

graphtik.network.**is_reschedule_operations**() → Optional[bool]

graphtik.network.**is_skip_evictions**() → Optional[bool]

graphtik.network.**yield_ops**(*nodes*)
May scan (preferably) `plan.steps` or dag nodes.

1.5.5 Module: *plot*

Plotting of graphtik graphs.

graphtik.plot.**build_pydot**(*graph*, *steps=None*, *inputs=None*, *outputs=None*, *solution=None*, *title=None*, *node_props=None*, *edge_props=None*, *clusters=None*, *legend_url='https://graphtik.readthedocs.io/en/latest/_images/GraphtikLegend.svg'*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f20fd103b38>
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: Mapping[KT, VT_co] = None*, *arch_url='https://graphtik.readthedocs.io/en/latest/arch.html'*)
Generate a legend for all plots (see *Plotter.plot()* for args)

Parameters **arch_url** – the url to the architecture section explaining *graphtik* glossary.

See *render_pydot()* for the rest arguments.

graphtik.plot.**render_pydot**(*dot: <sphinx.ext.autodoc.importer._MockObject object at 0x7f20fd103eb8>*, *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).

  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%"})
```

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

1.5.6 Module: *config*

Configurations for network execution, and utilities on them.

graphtik.config.**abort_run**()
Sets the *abort run* global flag, to halt all currently or future executing plans.

This global flag is reset when any *NetworkOperation.compute()* is executed, or manually, by calling *reset_abort()*.

graphtik.config.**evictions_skipped**(*enabled*)
Like *set_skip_evictions()* as a context-manager to reset old value.

graphtik.config.**get_execution_pool**() → Optional[Pool]
Get the process-pool for *parallel* plan executions.

graphtik.config.**is_abort**()
Return *True* if networks have been signaled to stop *execution*.

graphtik.config.**is_endure_operations**() → Optional[bool]
see *set_endure_operations()*

graphtik.config.**is_marshal_tasks**() → Optional[bool]
see *set_marshal_tasks()*

graphtik.config.**is_parallel_tasks**() → Optional[bool]
see *set_parallel_tasks()*

graphtik.config.**is_reschedule_operations**() → Optional[bool]
see *set_reschedule_operations()*

graphtik.config.**is_skip_evictions**() → Optional[bool]
see *set_skip_evictions()*

graphtik.config.**operations_endured**(*enabled*)
Like *set_endure_operations()* as a context-manager to reset old value.

graphtik.config.**operations_reschedullled**(*enabled*)
Like *set_reschedule_operations()* as a context-manager to reset old value.

graphtik.config.**reset_abort**()
Reset the *abort run* global flag, to permit plan executions to proceed.

graphtik.config.**set_endure_operations**(*enabled*)
Enable/disable globally *endurance* to keep executing even if some operations fail.

Parameters **enable** –

- If None (default), respect the flag on each operation;
- If true/false, force it for all operations.

Returns a "reset" token (see ContextVar.set())

.

graphtik.config.**set_execution_pool**(*pool: Optional[Pool]*)
Set the process-pool for *parallel* plan executions.

You may have to :also func:*set_marshal_tasks()* to resolve pickling issues.

graphtik.config.**set_marshal_tasks**(*enabled*)
Enable/disable globally *marshalling* of *parallel* operations, ...

inputs & outputs with dill, which might help for pickling problems.

Parameters **enable** –

- If None (default), respect the respective flag on each operation;
- If true/false, force it for all operations.

Returns a "reset" token (see ContextVar.set())

graphtik.config.**set_parallel_tasks**(*enabled*)
Enable/disable globally *parallel* execution of operations.

Parameters **enable** –

- If None (default), respect the respective flag on each operation;
- If true/false, force it for all operations.

Returns a "reset" token (see ContextVar.set())

graphtik.config.**set_reschedule_operations**(*enabled*)
Enable/disable globally *rescheduling* for operations returning only *partial outputs*.

Parameters **enable** –

- If None (default), respect the flag on each operation;
- If true/false, force it for all operations.

Returns a "reset" token (see `ContextVar.set()`)

.

`graphtik.config.`**`set_skip_evictions`**(*enabled*)
When true, disable globally *evictions*, to keep all intermediate solution values, ...

regardless of asked outputs.

Returns a "reset" token (see `ContextVar.set()`)

`graphtik.config.`**`tasks_in_parallel`**(*enabled*)
Like *set_parallel_tasks()* as a context-manager to reset old value.

`graphtik.config.`**`tasks_marshalled`**(*enabled*)
Like *set_marshal_tasks()* as a context-manager to reset old value.

1.5.7 Module: *base*

Generic or specific utilities

exception `graphtik.base.`**`MultiValueError`**

`graphtik.base.`**`NO_RESULT = <NO_RESULT>`**
When an operation function returns this special value, it implies operation has no result at all, (otherwise, it would have been a single result, `None`).'

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 bound 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"). It extracts more infos from a *Solution* instance, such as, if *solution* has an `executed` attribute, operations contained in it are drawn as "filled".
- **`title`** – an optional string to display at the bottom of the graph
- **`node_props`** – an optional nested dict of Graphviz attributes for certain nodes
- **`edge_props`** – an optional nested dict of Graphviz 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).
- **legend_url** – a URL to the *graphtik* legend; if it evaluates to false, none is added.

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(jupyter_render={"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.

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 `pydot.Dot` object,␣
↪renderable in Jupyter
>>> print(dot)
digraph G {
    URL="https://graphtik.readthedocs.io/en/latest/_images/GraphtikLegend.svg
↪";
    fontname=italic;
    label=netop;
    a [fillcolor=wheat, shape=invhouse, style=filled, tooltip=1];
...
```

class graphtik.base.**Token**(**args*)
Guarantee equality, not(!) identity, across processes.

hashid

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'salvage_vars'.

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

- Any attributes attached on this manager are attached as a new dict on the raised exception as new `jetsam` attribute 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")
    raise
```

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.

1.6 Graphtik Changelog

1.6.1 TODOs

See #1.

1.6.2 GitHub Releases

https://github.com/pygraphkit/graphtik/releases

1.6.3 Changelog

v5.2.0 (27 Feb 2020, @ankostis): Map *needs* inputs –> args, SPELLCHECK

- FEAT(modifiers): *optionals* and new modifier `arg` can now fetch values from *inputs* into differently-named arguments of operation functions.
  - refact: decouple *varargs* from *optional* modifiers hierarchy.

- REFACT(OP): preparation of NEEDS –> function-args happens *once* for each argument, allowing to report all errors at once.
- feat(base): +MultiValueError exception class.
- DOC(modifiers,arch): modifiers were not included in "API reference", nor in the glossary sections.
- FIX: spell-check everything, and add all custom words in the *VSCode* settings file `.vscode.settings.json`.

v5.1.0 (22 Jan 2020, @ankostis): accept named-tuples/objects *provides*

- ENH(OP): flag *returns_dict* handles also *named-tuples* & *objects* (`__dict__`).

v5.0.0 (31 Dec 2019, @ankostis): Method–>Parallel, all configs now per op flags; Screaming Solutions on fails/partials

- BREAK(NETOP): `compose(method="parallel") --> compose(parallel=None/False/True)` and DROP `netop.set_execution_method(method)`; *parallel* now also controlled with the global *`set_parallel_tasks()` configurations* function.
  - feat(jetsam): report *task* executed in raised exceptions.
- break(netop): rename `netop.narrowed() --> withset()` toi mimic `Operation` API.
- break: rename flags:
  - `reschedule --> rescheduleD`
  - `marshal --> marshalLED`.
- break: rename global configs, as context-managers:
  - `marshal_parallel_tasks --> tasks_marshalled`
  - `endure_operations --> operations_endured`
- FIX(net, plan,.TC): global skip *evictions* flag were not fully obeyed (was untested).
- FIX(OP): revamped zipping of function *outputs* with expected *provides*, for all combinations of rescheduled, `NO_RESULT` & *returns dictionary* flags.
- configs:
  - refact: extract configs in their own module.
  - refact: make all global flags tri-state (`None, False, True`), allowing to "force" operation flags when not *None*. All default to `None` (false).
- ENH(net, sol, logs): include a "solution-id" in revamped log messages, to facilitate developers to discover issues when multiple *netops* are running concurrently. Heavily enhanced log messages make sense to the reader of all actions performed.
- ENH(plot): set toolltips with `repr(op)` to view all operation flags.
- FIX(TCs): close process-pools; now much more TCs for parallel combinations of threaded, process-pool & marshalled.
- ENH(netop,net): possible to abort many netops at once, by resetting abort flag on every call of *`NetworkOperation.compute()`* (instead of on the first stopped *netop*).

- FEAT(SOL): *scream_if_incomplete()* will raise the new *IncompleteExecutionError* exception if failures/partial-outs of endured/rescheduled operations prevented all operations to complete; exception message details causal errors and conditions.
- feat(build): +''all'' extras.
- FAIL: x2 multi-threaded TCs fail spuriously with "inverse dag edges":
  - `test_multithreading_plan_execution()`
  - `test_multi_threading_computes()`

  both marked as `xfail`.

v4.4.1 (22 Dec 2019, @ankostis): bugfix debug print

- fix(net): had forgotten a debug-print on every operation call.
- doc(arch): explain *parallel* & the need for *marshalling* with process pools.

v4.4.0 (21 Dec 2019, @ankostis): RESCHEDULE for PARTIAL Outputs, on a per op basis

- [x] dynamic Reschedule after operations with partial outputs execute.
- [x] raise after jetsam.
- [x] plots link to legend.
- [x] refact netop
- [x] endurance per op.
- [x] endurance/reschedule for all netop ops.
- [x] merge _Rescheduler into Solution.
- [x] keep order of outputs in Solution even for parallels.
- [x] keep solution layers ordered also for parallel.
- [x] require user to create & enter pools.
- [x] FIX pickling THREAD POOL –>Process.

Details

- FIX(NET): keep Solution's insertion order also for PARALLEL executions.
- FEAT(NET, OP): *reschedule*d operations with partial outputs; they must have `FunctionalOperation.reschedule` set to true, or else they will fail.
- FEAT(OP, netop): specify *endurance*/*reschedule* on a per operation basis, or collectively for all operations grouped under some *netop*.
- REFACT(NETOP):
  - feat(netop): new method *NetworkOperation.compile()*, delegating to same-named method of *network*.
  - drop(net): method `Net.narrowed()`; remember *netop.narrowed(outputs+predicate)* and apply them on *netop.compute()* & `netop.compile()`.
    - PROS: cache narrowed plans.

        * CONS: cannot review network, must review plan of (new) *netop.compile()*.
    - drop(netop): *inputs* args in *narrowed()* didn't make much sense, leftover from "unvarying netops"; but exist ni *netop.compile()*.
    - refact(netop): move net-assembly from compose() –> NetOp cstor; now reschedule/endured/merge/method args in cstor.
- NET,OP,TCs: FIX PARALLEL POOL CONCURRENCY
    - Network:
        * feat: +marshal +_OpTask
        * refact: plan._call_op –> _handle_task
        * enh: Make *abort run* variable a *shared-memory* `Value`.
    - REFACT(OP,.TC): not a namedtuple, breaks pickling.
    - ENH(pool): Pool
    - FIX: compare Tokens with *is* –> ==, or else, it won't work for sub-processes.
    - TEST: x MULTIPLE TESTS
        * +4 tags: parallel, thread, proc, marshal.
        * many uses of exemethod.
- FIX(build): PyPi README check did not detect forbidden `raw` directives, and travis auto-deployments were failing.
- doc(arch): more terms.

v4.3.0 (16 Dec 2019, @ankostis): Aliases

- FEAT(OP): support "aliases" of *provides*, to avoid trivial pipe-through operations, just to rename & match operations.

v4.2.0 (16 Dec 2019, @ankostis): ENDURED Execution

- FEAT(NET): when *`set_endure_operations()`* configuration is set to true, a *netop* will keep on calculating solution, skipping any operations downstream from failed ones. The *solution* eventually collects all failures in `Solution.failures` attribute.
- ENH(DOC,plot): Links in Legend and *Architecture* Workflow SVGs now work, and delegate to *architecture* terms.
- ENH(plot): mark *overwrites*, *failed* & *canceled* in `repr()` (see *endurance*).
- refact(conf): fully rename configuration operation `skip_evictions`.
- REFACT(jetsam): raise after jetsam in situ, better for Readers & Linters.
- enh(net): improve logging.

v4.1.0 (13 Dec 2019, @ankostis): ChainMap Solution for Rewrites, stable TOPOLOGICAL sort

- FIX(NET): TOPOLOGICALLY-sort now break ties respecting operations insertion order.

- ENH(NET): new *Solution* class to collect all computation values, based on a collections.ChainMap to distinguish outputs per operation executed:
  - ENH(NETOP): compute() return *Solution*, consolidating:
    - *overwrites*,
    - executed operations, and
    - the generating *plan*.
  - drop(net): _PinInstruction class is not needed.
  - drop(netop): *overwrites_collector* parameter; now in Solution.overwrites().
  - ENH(plot): Solution is also a *Plotter*; e.g. use sol.plot(...)`.
- DROP(plot): *executed* arg from plotting; now embedded in *solution*.
- ENH(PLOT.jupyter,doc): allow to set jupyter graph-styling selectively; fix instructions for jupyter cell-resizing.
- fix(plan): time-keeping worked only for sequential execution, not parallel. Refactor it to happen centrally.
- enh(NET,.TC): Add PREDICATE argument also for compile().
- FEAT(DOC): add GLOSSARY as new *Architecture* section, linked from API HEADERS.

v4.0.1 (12 Dec 2019, @ankostis): bugfix

- FIX(plan): plan.repr() was failing on empty plans.
- fix(site): minor badge fix & landing diagram.

v4.0.0 (11 Dec 2019, @ankostis): NESTED merge, revert v3.x Unvarying, immutable OPs, "color" nodes

- BREAK/ENH(NETOP): MERGE NESTED NetOps by collecting all their operations in a single Network; now children netops are not pruned in case some of their *needs* are unsatisfied.
  - feat(op): support multiple nesting under other netops.
- BREAK(NETOP): REVERT Unvarying NetOps+base-plan, and narrow Networks instead; netops were too rigid, code was cumbersome, and could not really pinpoint the narrowed *needs* always correctly (e.g. when they were also *provides*).
  - A *netop* always narrows its *net* based on given *inputs/outputs*. This means that the *net* might be a subset of the one constructed out of the given operations. If you want all nodes, don't specify *needs/provides*.
  - drop 3 *ExecutionPlan* attributes: plan, needs, plan
  - drop *recompile* flag in Network.compute().
  - feat(net): new method Network.narrowed() clones and narrows.
  - Network() cstor accepts a (cloned) graph to support narrowed() methods.
- BREAK/REFACT(OP): simplify hierarchy, make *Operation* fully abstract, without name or requirements.
  - enh: make FunctionalOperation IMMUTABLE, by inheriting from class:*.namedtuple*.
- refact(net): consider as netop *needs* also intermediate data nodes.
- FEAT(#1, net, netop): support pruning based on arbitrary operation attributes (e.g. assign "colors" to nodes and solve a subset each time).

- enh(netop): `repr()` now counts number of contained operations.
- refact(netop): rename `netop.narrow() --> narrowed()`
- drop(netop): don't topologically-sort sub-networks before merging them; might change some results, but gives control back to the user to define nets.

v3.1.0 (6 Dec 2019, @ankostis): cooler `prune()`

- break/refact(NET): scream on `plan.execute()` (not `net.prune()`) so as calmly solve *needs* vs *provides*, based on the given *inputs*/*outputs*.
- FIX(ot): was failing when plotting graphs with ops without *fn* set.
- enh(net): minor fixes on assertions.

v3.0.0 (2 Dec 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 demanded, 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 `narrowed()` 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 interrogate *needs* & *provides* for a given graph. It accepts *None inputs* & *outputs* to auto-derive 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

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.

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.

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.

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 functools.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 supposed 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 BUILDING 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!

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: pruning 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 data nodes, 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

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 intermediate 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 when building it; thus, execute methods don't need outputs anymore.

- ENH(yahoo#26): "Pin* input values that may be overwritten 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 outputs"* 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, split 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 frequently met during debugging session or failed TCs (inspired by @syamajala's 309338340).
- enh: Sped up parallel/multithread 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

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).

1.2.2 (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

1.2.1 (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.

1.2.0 (Feb 13, 2018, @huyng)

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

1.1.0 (Nov 9, 2017, @huyng)

Update setup.py

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

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

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

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

- Remove 'packaging' as dependency

1.0.1 (Aug 24, 2016)

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:

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

... and plot the results (if in *jupyter*, no need to create the file):

```
>>> solution.plot('graphop.svg')
```

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

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Y