schedula Documentation

Release 0.1.19

Vincenzo Arcidiacono

Jun 05, 2018

Table of Contents

release 0.1.19

date 2018-06-05 13:00:00

repository https://github.com/vinci1it2000/schedula

pypi-repo https://pypi.org/project/schedula/

docs http://schedula.readthedocs.io/

wiki https://github.com/vinci1it2000/schedula/wiki/

download http://github.com/vinci1it2000/schedula/releases/

keywords scheduling, dispatch, dataflow, processing, calculation, dependencies, scientific, engineering, simulink, graph theory

developers

- Vincenzo Arcidiacono <vincenzo.arcidiacono@ext.jrc.ec.europa.eu>
- Kostis Anagnostopoulos <konstantinos.anagnostopoulos@ext.jrc.ec.europa.eu>

license EUPL 1.1+

CHAPTER 1

What is schedula?

Schedula implements a intelligent function scheduler, which selects and executes functions. The order (workflow) is calculated from the provided inputs and the requested outputs. A function is executed when all its dependencies (i.e., inputs, input domain) are satisfied and when at least one of its outputs has to be calculated.

Note: Schedula is performing the runtime selection of the **minimum-workflow** to be invoked. A workflow describes the overall process - i.e., the order of function execution - and it is defined by a directed acyclic graph (DAG). The **minimum-workflow** is the DAG where each output is calculated using the shortest path from the provided inputs. The path is calculated on the basis of a weighed directed graph (data-flow diagram) with a modified Dijkstra algorithm.

CHAPTER 2

Installation

To install it use (with root privileges):

```
$ pip install schedula
```

Or download the last git version and use (with root privileges):

```
$ python setup.py install
```

What is schedula?

Schedula implements a intelligent function scheduler, which selects and executes functions. The order (workflow) is calculated from the provided inputs and the requested outputs. A function is executed when all its dependencies (i.e., inputs, input domain) are satisfied and when at least one of its outputs has to be calculated.

Note: Schedula is performing the runtime selection of the **minimum-workflow** to be invoked. A workflow describes the overall process - i.e., the order of function execution - and it is defined by a directed acyclic graph (DAG). The **minimum-workflow** is the DAG where each output is calculated using the shortest path from the provided inputs. The path is calculated on the basis of a weighed directed graph (data-flow diagram) with a modified Dijkstra algorithm.

Installation

To install it use (with root privileges):

```
$ pip install schedula
```

Or download the last git version and use (with root privileges):

```
$ python setup.py install
```

Why may I use schedula?

Imagine we have a system of interdependent functions - i.e. the inputs of a function are the output for one or more function(s), and we do not know which input the user will provide and which output will request. With a normal

scheduler you would have to code all possible implementations. I'm bored to think and code all possible combinations of inputs and outputs from a model.

Solution

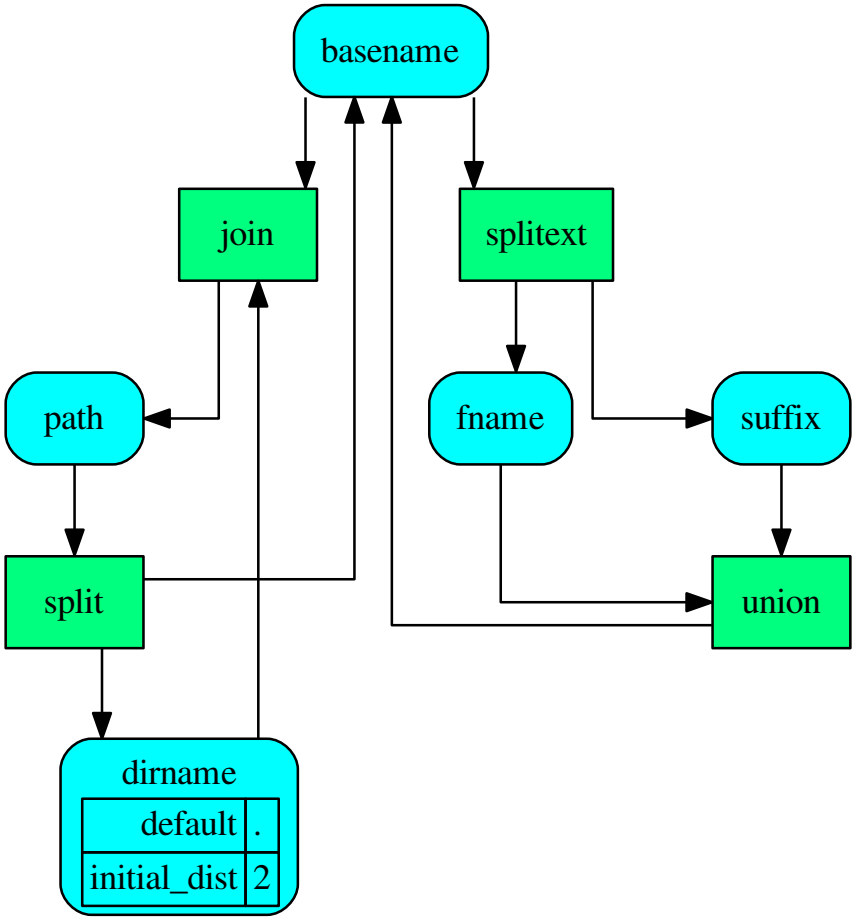
Schedula allows to write a simple model (`Dispatcher()`) with just the basic functions, then the `Dispatcher()` will select and execute the proper functions for the given inputs and the requested outputs. Moreover, schedula provides a flexible framework for structuring code. It allows to extract sub-models from a bigger one.

Note: A successful application is CO_2MPAS, where schedula has been used to model an entire vehicle.

Very simple example

Let's assume that we have to extract some filesystem attributes and we do not know which inputs the user will provide. The code below shows how to create a `Dispatcher()` adding the functions that define your system. Note that with this simple system the maximum number of inputs combinations is 31 ($(2^n - 1)$, where n is the number of data).

```
>>> import schedula
>>> import os.path as osp
>>> dsp = schedula.Dispatcher()
>>> dsp.add_data(data_id='dirname', default_value='.', initial_dist=2)
'dirname'
>>> dsp.add_function(function=osp.split, inputs=['path'],
...                  outputs=['dirname', 'basename'])
'split'
>>> dsp.add_function(function=osp.splitext, inputs=['basename'],
...                  outputs=['fname', 'suffix'])
'splitext'
>>> dsp.add_function(function=osp.join, inputs=['dirname', 'basename'],
...                  outputs=['path'])
'join'
>>> dsp.add_function(function_id='union', function=lambda *a: ''.join(a),
...                  inputs=['fname', 'suffix'], outputs=['basename'])
'union'
```



dmap

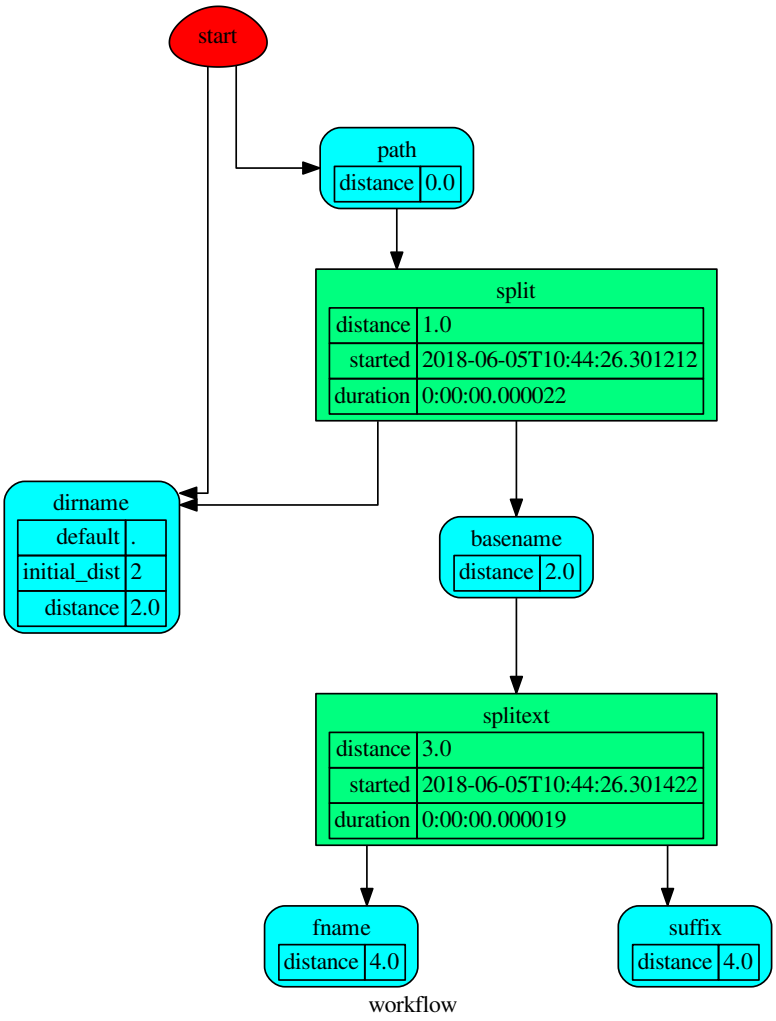
Tip: You can explore the diagram by clicking on it.

Note: For more details how to created a `Dispatcher()` see: `add_data()`, `add_function()`, `add_dispatcher()`, *`SubDispatch()`*, *`SubDispatchFunction()`*, *`SubDispatchPipe()`*, and *`DFun()`*.

The next step to calculate the outputs would be just to run the `dispatch()` method. You can invoke it with just the inputs, so it will calculate all reachable outputs:

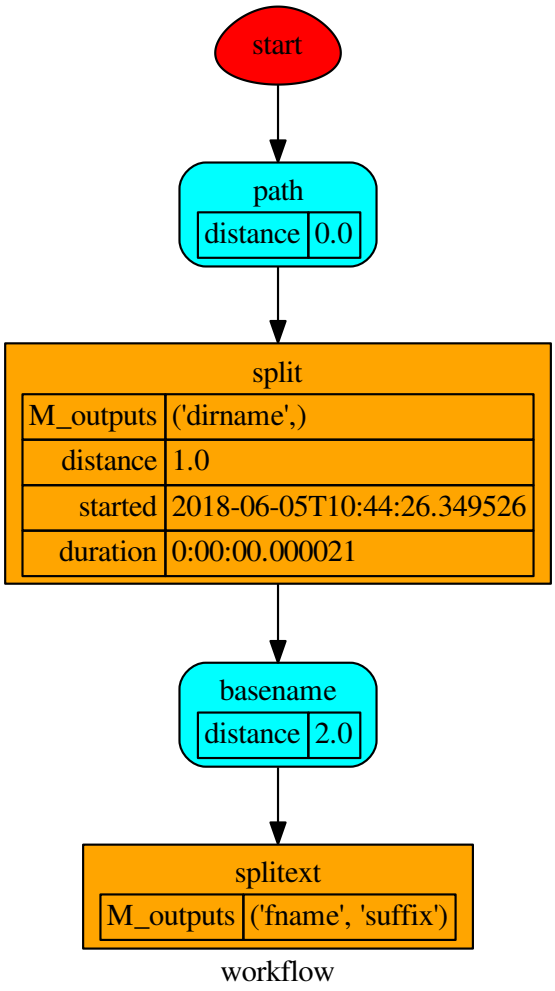
```
>>> inputs = {'path': 'schedula/_version.py'}
>>> o = dsp.dispatch(inputs=inputs)
>>> o
Solution([('path', 'schedula/_version.py'),
          ('basename', '_version.py'),
```

```
        ('dirname', 'schedula'),
        ('fname', '_version'),
        ('suffix', '.py')])
```



or you can set also the outputs, so the dispatch will stop when it will find all outputs:

```
>>> o = dsp.dispatch(inputs=inputs, outputs=['basename'])
>>> o
Solution([('path', 'schedula/_version.py'), ('basename', '_version.py')])
```



workflow

Advanced example (circular system)

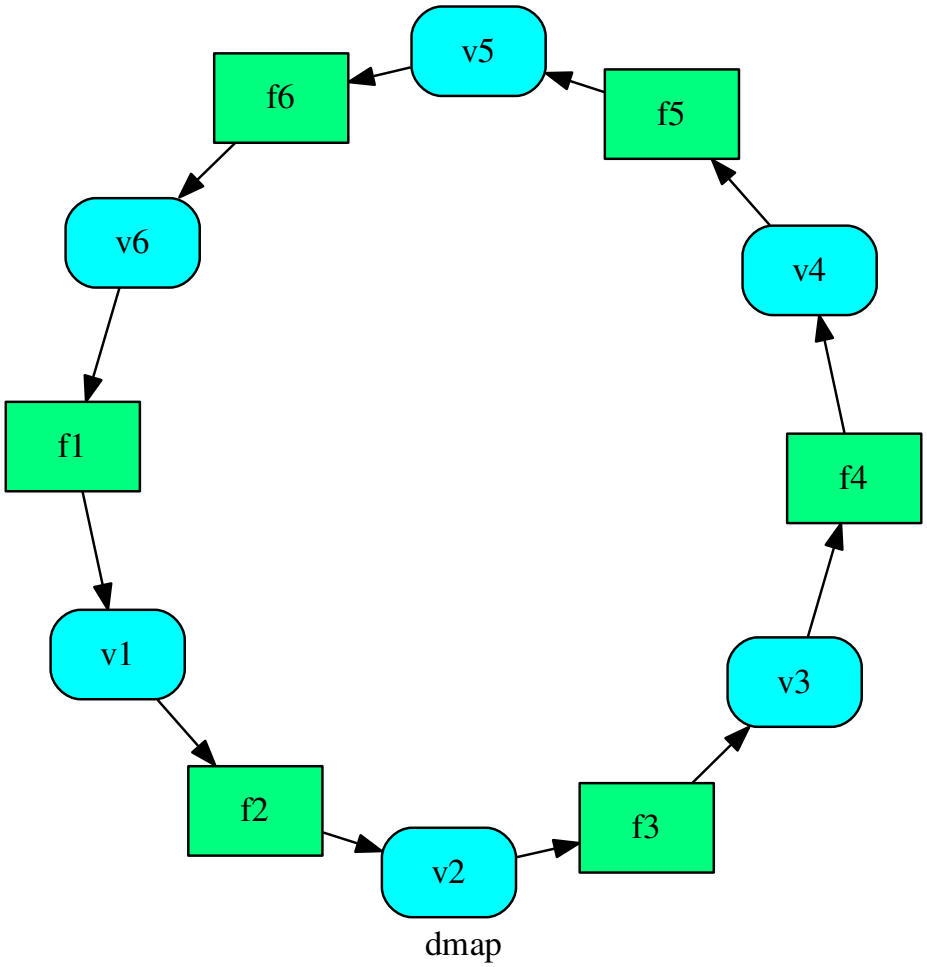
Systems of interdependent functions can be described by "graphs" and they might contains **circles**. This kind of system can not be resolved by a normal scheduler.

Suppose to have a system of sequential functions in circle - i.e., the input of a function is the output of the previous function. The maximum number of input and output permutations is $(2^n - 1)^2$, where n is the number of functions. Thus, with a normal scheduler you have to code all possible implementations, so $(2^n - 1)^2$ functions (IMPOSSIBLE!!!).

Schedula will simplify your life. You just create a `Dispatcher()`, that contains all functions that link your data:

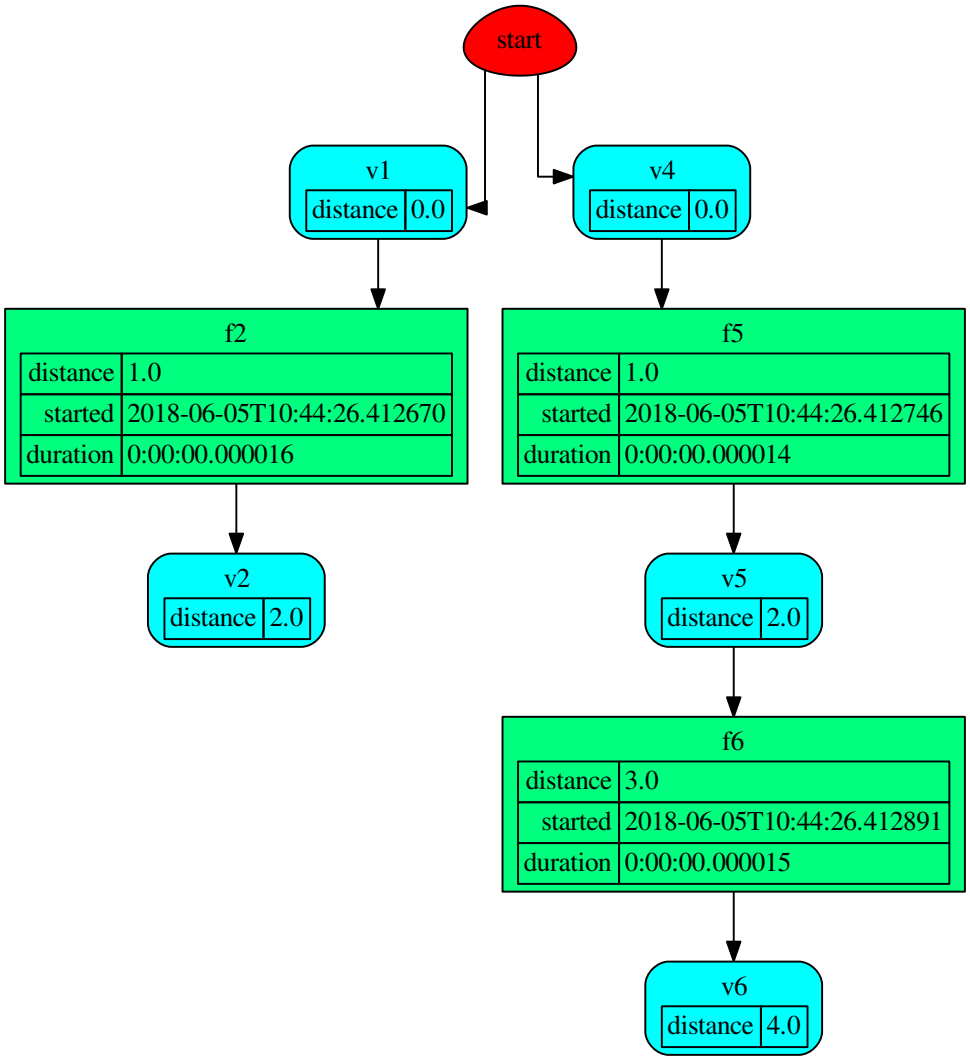
```
>>> import schedula
>>> dsp = schedula.Dispatcher()
>>> plus, minus = lambda x: x + 1, lambda x: x - 1
>>> n = j = 6
```

```
>>> for i in range(1, n + 1):
...     func = plus if i < (n / 2 + 1) else minus
...     f = dsp.add_function('f%d' % i, func, ['v%d' % j], ['v%d' % i])
...     j = i
```



Then it will handle all possible combination of inputs and outputs ($(2^n-1)^2$) just invoking the `dispatch()` method, as follows:

```
>>> out = dsp.dispatch(inputs={'v1': 0, 'v4': 1}, outputs=['v2', 'v6'])
>>> out
Solution([('v1', 0), ('v4', 1), ('v2', 1), ('v5', 0), ('v6', -1)])
```

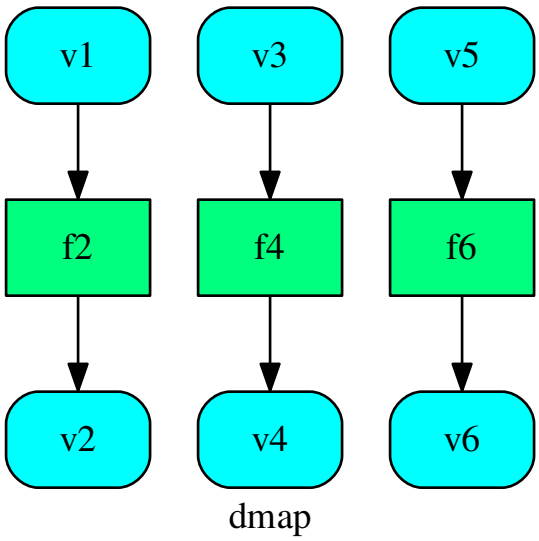


workflow

Sub-system extraction

Schedula allows to extract sub-models from a model. This could be done with the `shrink_dsp()` method, as follows:

```
>>> sub_dsp = dsp.shrink_dsp(('v1', 'v3', 'v5'), ('v2', 'v4', 'v6'))
```



Note: For more details how to extract a sub-model see: `get_sub_dsp()`, `get_sub_dsp_from_workflow()`, *`SubDispatch()`*, *`SubDispatchFunction()`*, and *`SubDispatchPipe()`*.

Next moves

Things yet to do include a mechanism to allow the execution of functions in parallel.

API Reference

The core of the library is composed from the following modules: It contains a comprehensive list of all modules and classes within schedula.

Docstrings should provide sufficient understanding for any individual function.

Modules:

`dispatcher`	It provides Dispatcher class.
`utils`	It contains utility classes and functions.
`ext`	It provides sphinx extensions.

dispatcher

It provides Dispatcher class.

Classes

Dispatcher	It provides a data structure to process a complex system of functions.

Dispatcher

class **Dispatcher**(*dmap=None*, *name=''*, *default_values=None*, *raises=False*, *description=''*, *stopper=None*)

It provides a data structure to process a complex system of functions.

The scope of this data structure is to compute the shortest workflow between input and output data nodes.

Variables ***stopper*** – A semaphore (`threading.Event`) to abort the dispatching.

Tip: Remember to set *stopper* to *False* before dispatching ;-)

A workflow is a sequence of function calls.

****

Example:

As an example, here is a system of equations:

$b - a = c$

$log(c) = d_{from-log}$

$d = (d_{from-log} + d_{initial-guess})/2$

that will be solved assuming that $a = 0$, $b = 1$, and $d_{initial-guess} = 4$.

Steps

Create an empty dispatcher:

```
>>> dsp = Dispatcher(name='Dispatcher')
```

Add data nodes to the dispatcher map:

```
>>> dsp.add_data(data_id='a')
'a'
>>> dsp.add_data(data_id='c')
'c'
```

Add a data node with a default value to the dispatcher map:

```
>>> dsp.add_data(data_id='b', default_value=1)
'b'
```

Add a function node:

```
>>> def diff_function(a, b):
...     return b - a
...
>>> dsp.add_function('diff_function', function=diff_function,
...                  inputs=['a', 'b'], outputs=['c'])
'diff_function'
```

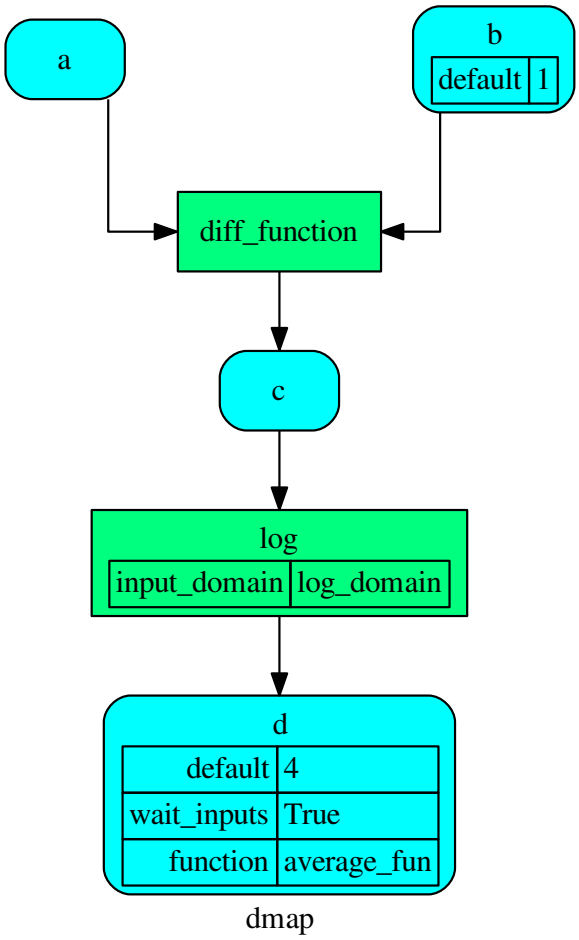
Add a function node with domain:

```
>>> from math import log
...
>>> def log_domain(x):
...     return x > 0
...
>>> dsp.add_function('log', function=log, inputs=['c'], outputs=['d'],
...                  input_domain=log_domain)
'log'
```

Add a data node with function estimation and callback function.

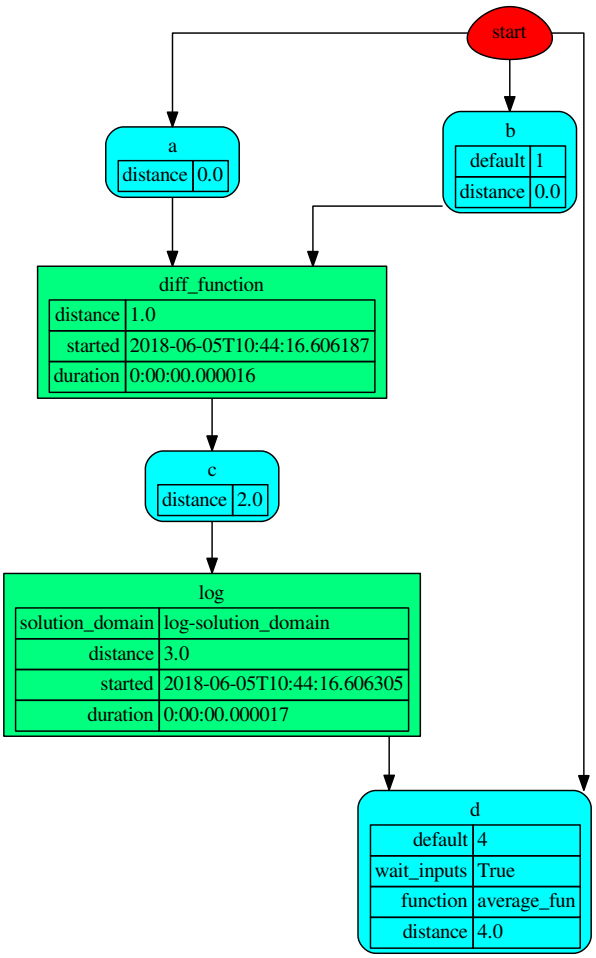
- function estimation: estimate one unique output from multiple estimations.
- callback function: is invoked after computing the output.

```
>>> def average_fun(kwargs):
...     '''
...     Returns the average of node estimations.
...
...     :param kwargs:
...         Node estimations.
...     :type kwargs: dict
...
...     :return:
...         The average of node estimations.
...     :rtype: float
...     '''
...
...     x = kwargs.values()
...     return sum(x) / len(x)
...
>>> def callback_fun(x):
...     print('(log(1) + 4) / 2 = %.1f' % x)
...
>>> dsp.add_data(data_id='d', default_value=4, wait_inputs=True,
...              function=average_fun, callback=callback_fun)
'd'
```



Dispatch the function calls to achieve the desired output data node *d*:

```
>>> outputs = dsp.dispatch(inputs={'a': 0}, outputs=['d'])
(log(1) + 4) / 2 = 2.0
>>> outputs
Solution([('a', 0), ('b', 1), ('c', 1), ('d', 2.0)])
```



workflow

Methods

__init__	Initializes the dispatcher.
add_data	Add a single data node to the dispatcher.
add_dispatcher	Add a single sub-dispatcher node to dispatcher.
add_from_lists	Add multiple function and data nodes to dispatcher.
add_function	Add a single function node to dispatcher.
copy	Returns a copy of the Dispatcher.
copy_structure	
dispatch	Evaluates the minimum workflow and data outputs of the dispatcher model from given inputs.
get_node	Returns a sub node of a dispatcher.
get_sub_dsp	Returns the sub-dispatcher induced by given node and edge bunches.

Continued on next page

Table 2.3 – continued from previous page

get_sub_dsp_from_workflow	Returns the sub-dispatcher induced by the workflow from sources.
plot	Plots the Dispatcher with a graph in the DOT language with Graphviz.
search_node_description	
set_data_remote_link	Set a remote link of a data node in the dispatcher.
set_default_value	Set the default value of a data node in the dispatcher.
shrink_dsp	Returns a reduced dispatcher.
web	Creates a dispatcher Flask app.

__init__

Dispatcher.**__init__**(*dmap=None*, *name=''*, *default_values=None*, *raises=False*, *description=''*, *stopper=None*)

Initializes the dispatcher.

Parameters

- **dmap** (*networkx.DiGraph, optional*) – A directed graph that stores data & functions parameters.
- **name** (*str, optional*) – The dispatcher's name.
- **default_values** (*dict[str, dict], optional*) – Data node default values. These will be used as input if it is not specified as inputs in the ArciDispatch algorithm.
- **raises** (*bool, optional*) – If True the dispatcher interrupt the dispatch when an error occur, otherwise it logs a warning.
- **description** (*str, optional*) – The dispatcher's description.
- **stopper** (*threading.Event, optional*) – A semaphore to abort the dispatching.

add_data

Dispatcher.**add_data**(*data_id=None*, *default_value=empty*, *initial_dist=0.0*, *wait_inputs=False*, *wildcard=None*, *function=None*, *callback=None*, *remote_links=None*, *description=None*, *filters=None*, ***kwargs*)

Add a single data node to the dispatcher.

Parameters

- **data_id** (*str, optional*) – Data node id. If None will be assigned automatically ('unknown<%d>') not in dmap.
- **default_value** (*T, optional*) – Data node default value. This will be used as input if it is not specified as inputs in the ArciDispatch algorithm.
- **initial_dist** (*float, int, optional*) – Initial distance in the ArciDispatch algorithm when the data node default value is used.
- **wait_inputs** (*bool, optional*) – If True ArciDispatch algorithm stops on the node until it gets all input estimations.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.

- **function** (*callable, optional*) – Data node estimation function. This can be any function that takes only one dictionary (key=function node id, value=estimation of data node) as input and return one value that is the estimation of the data node.
- **callback** (*callable, optional*) – Callback function to be called after node estimation. This can be any function that takes only one argument that is the data node estimation output. It does not return anything.
- **remote_links** (*list[[str, Dispatcher]], optional*) – List of parent or child dispatcher nodes e.g., [[dsp_id, dsp], ...].
- **description** (*str, optional*) – Data node's description.
- **filters** (*list[function], optional*) – A list of functions that are invoked after the invocation of the main function.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Data node id.

Return type str

See also:

add_function(), *add_dispatcher()*, *add_from_lists()*

****

Example:

Add a data to be estimated or a possible input data node:

```
>>> dsp.add_data(data_id='a')
'a'
```

Add a data with a default value (i.e., input data node):

```
>>> dsp.add_data(data_id='b', default_value=1)
'b'
```

Create a data node with function estimation and a default value.

•function estimation: estimate one unique output from multiple estimations.

•default value: is a default estimation.

```
>>> def min_fun(kwargs):
...     '''
...     Returns the minimum value of node estimations.
...
...     :param kwargs:
...         Node estimations.
...     :type kwargs: dict
...
...     :return:
...         The minimum value of node estimations.
...     :rtype: float
...     '''
...
...     return min(kwargs.values())
...
>>> dsp.add_data(data_id='c', default_value=2, wait_inputs=True,
```

```
...                 function=min_fun)
'c'
```

Create a data with an unknown id and return the generated id:

```
>>> dsp.add_data()
'unknown'
```

add_dispatcher

Dispatcher.**add_dispatcher**(*dsp*, *inputs*, *outputs*, *dsp_id=None*, *input_domain=None*, *weight=None*, *inp_weight=None*, *description=None*, *include_defaults=False*, ***kwargs*)

Add a single sub-dispatcher node to dispatcher.

Parameters

- **dsp** (*Dispatcher | dict[str, list]*) – Child dispatcher that is added as sub-dispatcher node to the parent dispatcher.
- **inputs** (*dict[str, str | list[str]] | tuple[str] | (str, ..., dict[str, str | list[str]])*) – Inputs mapping. Data node ids from parent dispatcher to child sub-dispatcher.
- **outputs** (*dict[str, str | list[str]] | tuple[str] | (str, ..., dict[str, str | list[str]])*) – Outputs mapping. Data node ids from child sub-dispatcher to parent dispatcher.
- **dsp_id** (*str, optional*) – Sub-dispatcher node id. If None will be assigned as <dsp.name>.
- **input_domain** (*(dict) -> bool, optional*) – A function that checks if input values satisfy the function domain. This can be any function that takes the a dictionary with the inputs of the sub-dispatcher node and returns True if input values satisfy the domain, otherwise False.

  Note: This function is invoked every time that a data node reach the sub-dispatcher node.

- **weight** (*float, int, optional*) – Node weight. It is a weight coefficient that is used by the dispatch algorithm to estimate the minimum workflow.
- **inp_weight** (*dict[str, int | float], optional*) – Edge weights from data nodes to the sub-dispatcher node. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **description** (*str, optional*) – Sub-dispatcher node's description.
- **include_defaults** (*bool, optional*) – If True the default values of the sub-dispatcher are added to the current dispatcher.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Sub-dispatcher node id.

Return type str

See also:

add_data(), *add_function()*, *add_from_lists()*

****

Example:

Create a sub-dispatcher:

```
>>> sub_dsp = Dispatcher()
>>> sub_dsp.add_function('max', max, ['a', 'b'], ['c'])
'max'
```

Add the sub-dispatcher to the parent dispatcher:

```
>>> dsp.add_dispatcher(dsp_id='Sub-Dispatcher', dsp=sub_dsp,
...                    inputs={'A': 'a', 'B': 'b'},
...                    outputs={'c': 'C'})
'Sub-Dispatcher'
```

Add a sub-dispatcher node with domain:

```
>>> def my_domain(kwargs):
...     return kwargs['C'] > 3
...
>>> dsp.add_dispatcher(dsp_id='Sub-Dispatcher with domain',
...                    dsp=sub_dsp, inputs={'C': 'a', 'D': 'b'},
...                    outputs={'c': 'E'}, input_domain=my_domain)
'Sub-Dispatcher with domain'
```

add_from_lists

Dispatcher.**add_from_lists**(*data_list=None*, *fun_list=None*, *dsp_list=None*)

Add multiple function and data nodes to dispatcher.

Parameters

- **data_list** (*list[dict], optional*) – It is a list of data node kwargs to be loaded.
- **fun_list** (*list[dict], optional*) – It is a list of function node kwargs to be loaded.
- **dsp_list** (*list[dict], optional*) – It is a list of sub-dispatcher node kwargs to be loaded.

Returns

- Data node ids.
- Function node ids.
- Sub-dispatcher node ids.

Return type (list[str], list[str], list[str])

See also:

add_data(), *add_function()*, *add_dispatcher()*

****

Example:

Define a data list:

```
>>> data_list = [
...     {'data_id': 'a'},
...     {'data_id': 'b'},
...     {'data_id': 'c'},
... ]
```

Define a functions list:

```
>>> def func(a, b):
...     return a + b
...
>>> fun_list = [
...     {'function': func, 'inputs': ['a', 'b'], 'outputs': ['c']}
... ]
```

Define a sub-dispatchers list:

```
>>> sub_dsp = Dispatcher(name='Sub-dispatcher')
>>> sub_dsp.add_function(function=func, inputs=['e', 'f'],
...                      outputs=['g'])
'func'
>>>
>>> dsp_list = [
...     {'dsp_id': 'Sub', 'dsp': sub_dsp,
...      'inputs': {'a': 'e', 'b': 'f'}, 'outputs': {'g': 'c'}},
... ]
```

Add function and data nodes to dispatcher:

```
>>> dsp.add_from_lists(data_list, fun_list, dsp_list)
(['a', 'b', 'c'], ['func'], ['Sub'])
```

add_function

Dispatcher.**add_function**(*function_id=None*, *function=None*, *inputs=None*, *outputs=None*, *input_domain=None*, *weight=None*, *inp_weight=None*, *out_weight=None*, *description=None*, *filters=None*, ***kwargs*)

Add a single function node to dispatcher.

Parameters

- **function_id** (*str, optional*) – Function node id. If None will be assigned as <fun.__name__>.
- **function** (*callable, optional*) – Data node estimation function.
- **inputs** (*list, optional*) – Ordered arguments (i.e., data node ids) needed by the function.
- **outputs** (*list, optional*) – Ordered results (i.e., data node ids) returned by the function.
- **input_domain** (*callable, optional*) – A function that checks if input values satisfy the function domain. This can be any function that takes the same inputs of the

function and returns True if input values satisfy the domain, otherwise False. In this case the dispatch algorithm doesn't pass on the node.

- **weight** (*float, int, optional*) – Node weight. It is a weight coefficient that is used by the dispatch algorithm to estimate the minimum workflow.
- **inp_weight** (*dict[str, float | int], optional*) – Edge weights from data nodes to the function node. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **out_weight** (*dict[str, float | int], optional*) – Edge weights from the function node to data nodes. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **description** (*str, optional*) – Function node's description.
- **filters** (*list[function], optional*) – A list of functions that are invoked after the invocation of the main function.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Function node id.

Return type str

See also:

add_data(), *add_dispatcher()*, *add_from_lists()*

****

Example:

Add a function node:

```
>>> def my_function(a, b):
...     c = a + b
...     d = a - b
...     return c, d
...
>>> dsp.add_function(function=my_function, inputs=['a', 'b'],
...                  outputs=['c', 'd'])
'my_function'
```

Add a function node with domain:

```
>>> from math import log
>>> def my_log(a, b):
...     return log(b - a)
...
>>> def my_domain(a, b):
...     return a < b
...
>>> dsp.add_function(function=my_log, inputs=['a', 'b'],
...                  outputs=['e'], input_domain=my_domain)
'my_log'
```

copy

Dispatcher.**copy**()

Returns a copy of the Dispatcher.

Returns A copy of the Dispatcher.

Return type *Dispatcher*

Example:

```
>>> dsp = Dispatcher()
>>> dsp is dsp.copy()
False
```

copy_structure

Dispatcher.**copy_structure**(***kwargs*)

dispatch

Dispatcher.**dispatch**(*inputs=None*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *wildcard=False*, *no_call=False*, *shrink=False*, *rm_unused_nds=False*, *select_output_kw=None*, *_wait_in=None*, *stopper=None*)

Evaluates the minimum workflow and data outputs of the dispatcher model from given inputs.

Parameters

- **inputs** (*dict[str, T], list[str], iterable, optional*) – Input data values.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.
- **no_call** (*bool, optional*) – If True data node estimation function is not used and the input values are not used.
- **shrink** (*bool, optional*) – If True the dispatcher is shrink before the dispatch.

  See also:

  shrink_dsp()
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **select_output_kw** (*dict, optional*) – Kwargs of selector function to select specific outputs.
- **_wait_in** (*dict, optional*) – Override wait inputs.

- **stopper** (*threading.Event, optional*) – A semaphore to abort the dispatching.

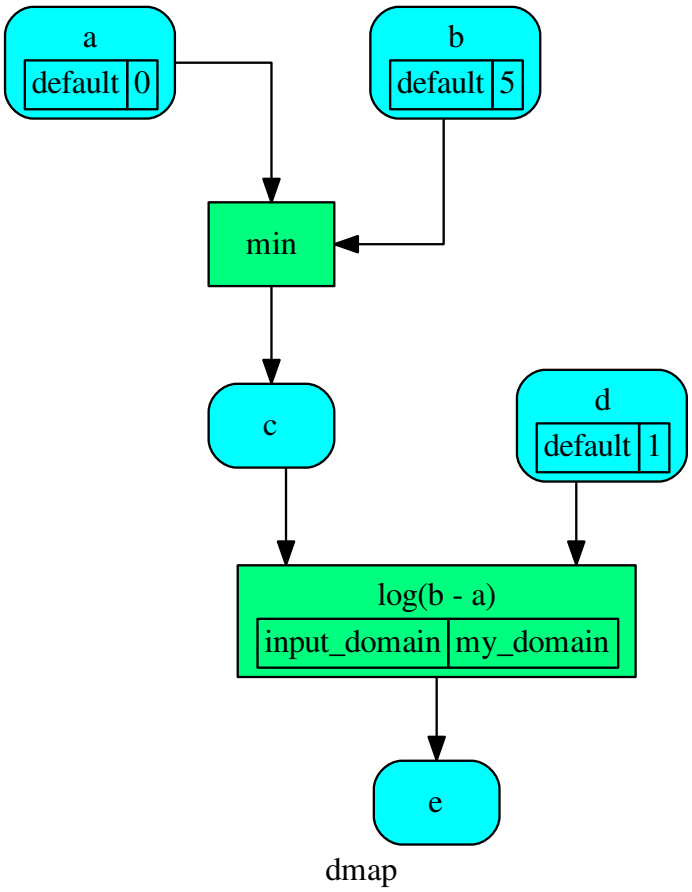
Returns Dictionary of estimated data node outputs.

Return type *schedula.utils.sol.Solution*

****

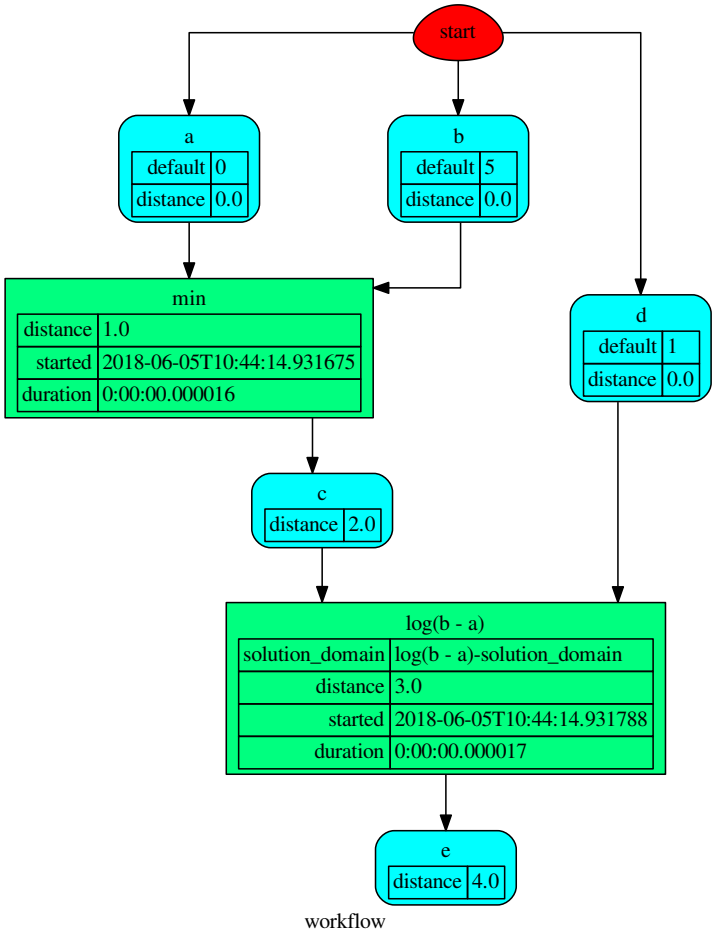
Example:

A dispatcher with a function $log(b - a)$ and two data a and b with default values:



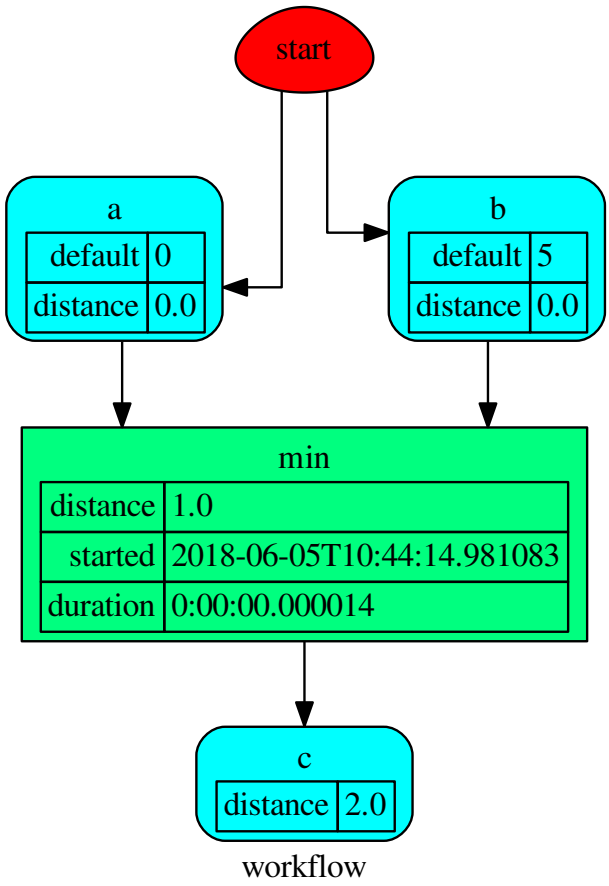
Dispatch without inputs. The default values are used as inputs:

```
>>> outputs = dsp.dispatch()
>>> outputs
Solution([('a', 0), ('b', 5), ('d', 1), ('c', 0), ('e', 0.0)])
```



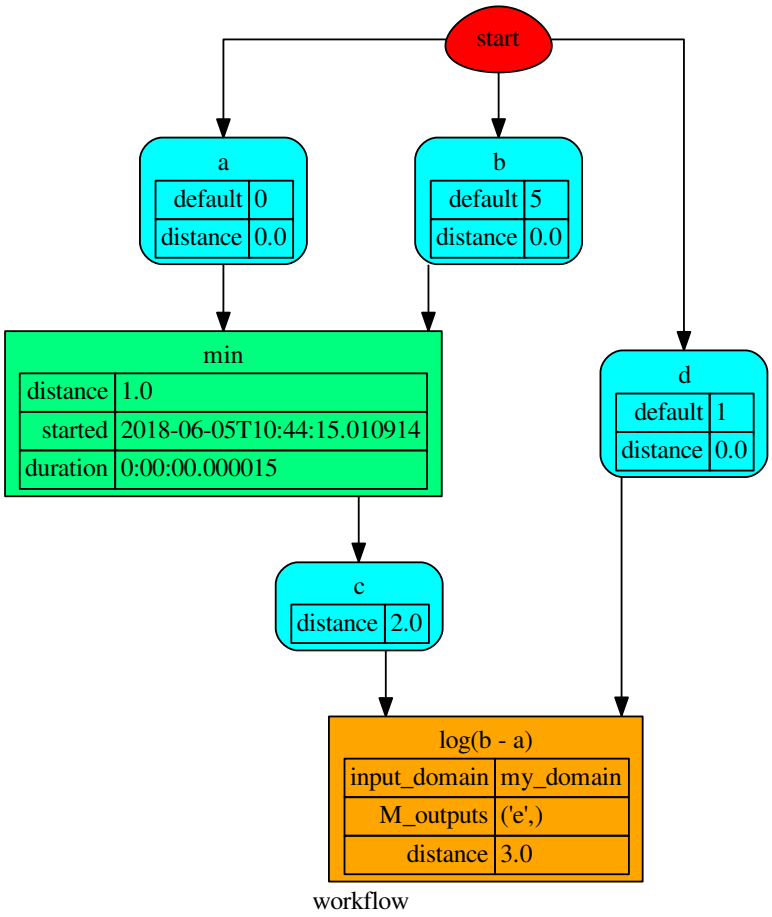
Dispatch until data node *c* is estimated:

```
>>> outputs = dsp.dispatch(outputs=['c'])
>>> outputs
Solution([('a', 0), ('b', 5), ('c', 0)])
```



Dispatch with one inputs. The default value of *a* is not used as inputs:

```
>>> outputs = dsp.dispatch(inputs={'a': 3})
>>> outputs
Solution([('a', 3), ('b', 5), ('d', 1), ('c', 3)])
```



get_node

Dispatcher.**get_node**(**node_ids*, ***, *node_attr=none*)

Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  – for data node: its output, and if not exists, all its attributes.

  – for function and sub-dispatcher nodes: the 'function' attribute.

  When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

When 'output', returns the data node output.

When 'default_value', returns the data node default value.

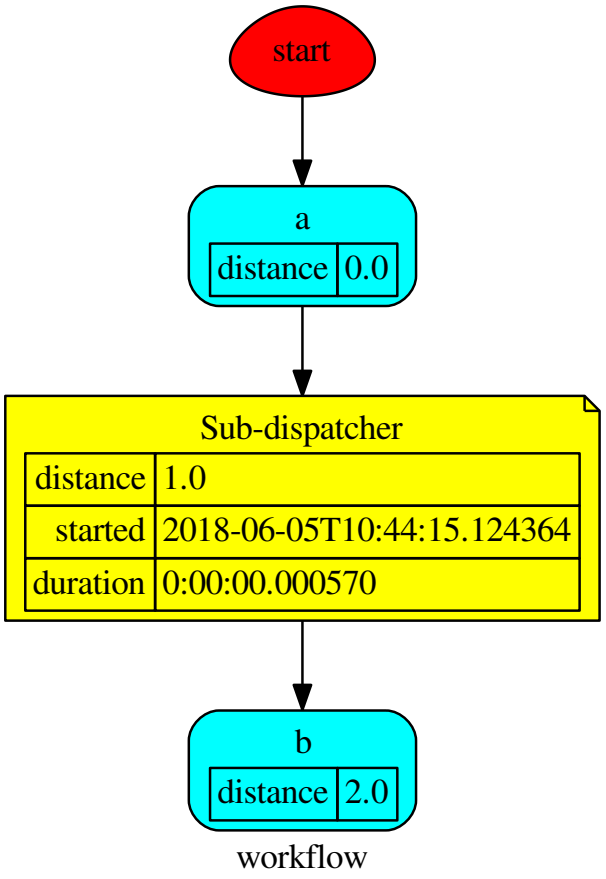
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

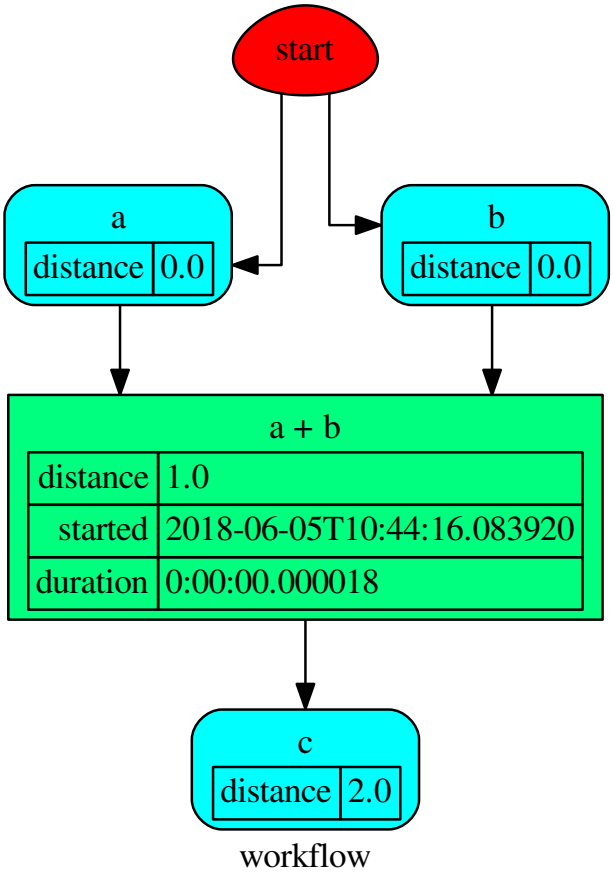
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



get_sub_dsp

Dispatcher.**get_sub_dsp**(*nodes_bunch*, *edges_bunch=None*)

Returns the sub-dispatcher induced by given node and edge bunches.

The induced sub-dispatcher contains the available nodes in nodes_bunch and edges between those nodes, excluding those that are in edges_bunch.

The available nodes are non isolated nodes and function nodes that have all inputs and at least one output.

Parameters

- **nodes_bunch** (*list[str], iterable*) – A container of node ids which will be iterated through once.
- **edges_bunch** (*list[(str, str)], iterable, optional*) – A container of edge ids that will be removed.

Returns A dispatcher.

Return type *Dispatcher*

See also:

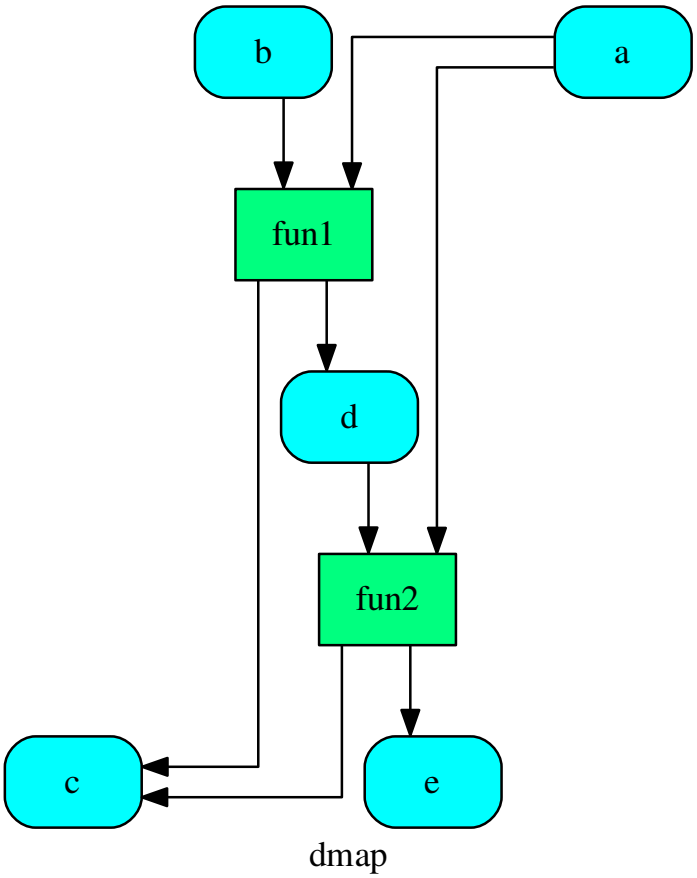
get_sub_dsp_from_workflow()

Note: The sub-dispatcher edge or node attributes just point to the original dispatcher. So changes to the node or edge structure will not be reflected in the original dispatcher map while changes to the attributes will.

****

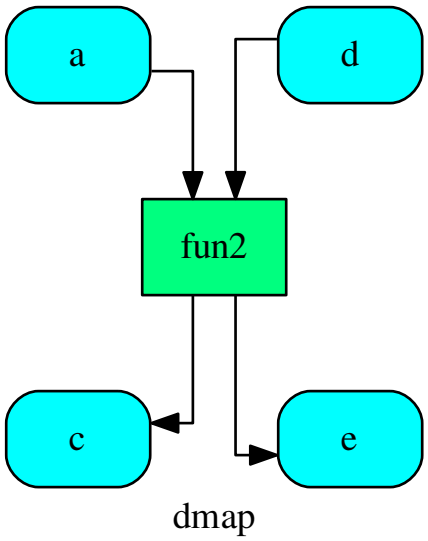
Example:

A dispatcher with a two functions *fun1* and *fun2*:



Get the sub-dispatcher induced by given nodes bunch:

```
>>> sub_dsp = dsp.get_sub_dsp(['a', 'c', 'd', 'e', 'fun2'])
```



get_sub_dsp_from_workflow

Dispatcher.**get_sub_dsp_from_workflow**(*sources*, *graph=None*, *reverse=False*, *add_missing=False*, *check_inputs=True*, *blockers=None*, *wildcard=False*, *_update_links=True*)

Returns the sub-dispatcher induced by the workflow from sources.

The induced sub-dispatcher of the dsp contains the reachable nodes and edges evaluated with breadth-first-search on the workflow graph from source nodes.

Parameters

- **sources** (*list[str], iterable*) – Source nodes for the breadth-first-search. A container of nodes which will be iterated through once.
- **graph** (*networkx.DiGraph, optional*) – A directed graph where evaluate the breadth-first-search.
- **reverse** (*bool, optional*) – If True the workflow graph is assumed as reversed.
- **add_missing** (*bool, optional*) – If True, missing function' inputs are added to the sub-dispatcher.
- **check_inputs** (*bool, optional*) – If True the missing function' inputs are not checked.
- **blockers** (*set[str], iterable, optional*) – Nodes to not be added to the queue.

- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.
- **_update_links** (*bool, optional*) – If True, it updates remote links of the extracted dispatcher.

Returns A sub-dispatcher.

Return type *Dispatcher*

See also:

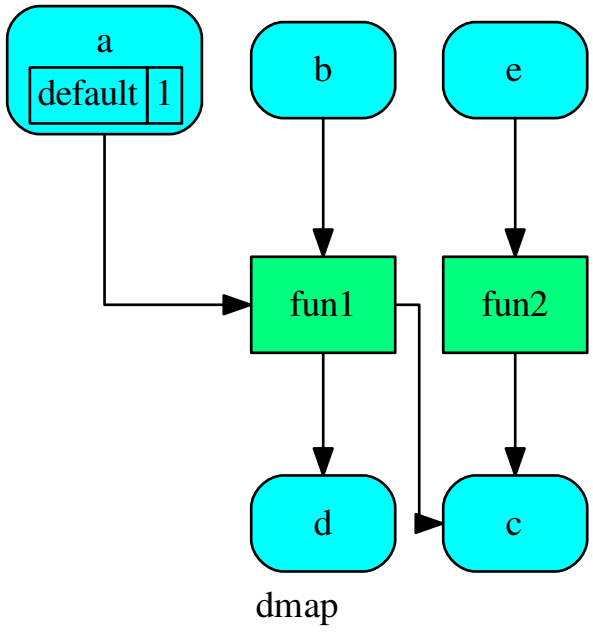
get_sub_dsp()

Note: The sub-dispatcher edge or node attributes just point to the original dispatcher. So changes to the node or edge structure will not be reflected in the original dispatcher map while changes to the attributes will.

****

Example:

A dispatcher with a function *fun* and a node *a* with a default value:

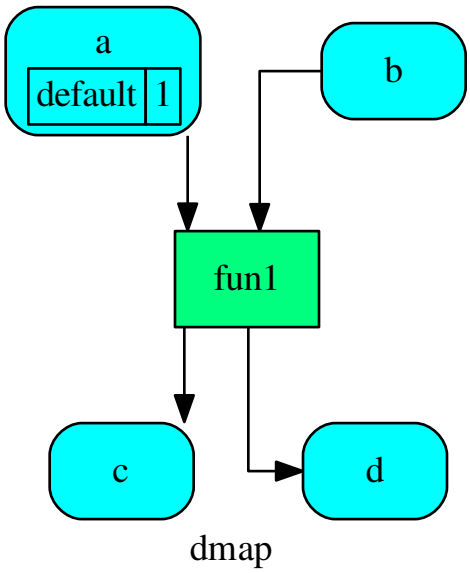


Dispatch with no calls in order to have a workflow:

```
>>> o = dsp.dispatch(inputs=['a', 'b'], no_call=True)
```

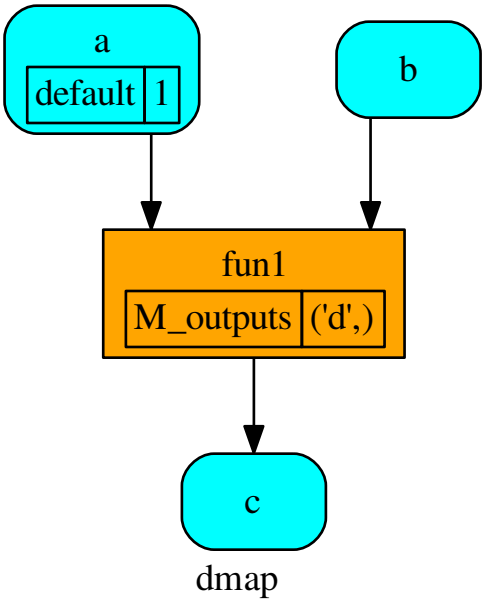
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> sub_dsp = dsp.get_sub_dsp_from_workflow(['a', 'b'])
```



Get sub-dispatcher from a workflow output *c*:

```
>>> sub_dsp = dsp.get_sub_dsp_from_workflow(['c'], reverse=True)
```



plot

Dispatcher.**plot**(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)

Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **workflow** (*bool, optional*) – If True the latest solution will be plotted, otherwise the dmap.
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.

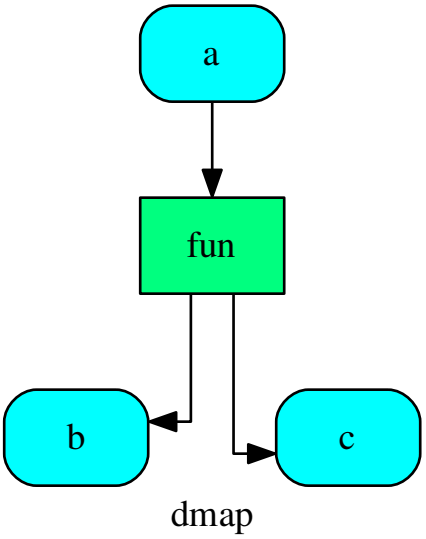
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



search_node_description

Dispatcher.**search_node_description**(*node_id*, *what='description'*)

set_data_remote_link

Dispatcher.**set_data_remote_link**(*data_id*, *remote_link=empty*, *is_parent=True*)

Set a remote link of a data node in the dispatcher.

Parameters

- **data_id** (*str*) – Data node id.
- **remote_link** (*[str, Dispatcher], optional*) – Parent or child dispatcher and its node id (id, dsp).
- **is_parent** (*bool*) – If True the link is inflow (parent), otherwise is outflow (child).

set_default_value

Dispatcher.**set_default_value**(*data_id*, *value=empty*, *initial_dist=0.0*)

Set the default value of a data node in the dispatcher.

Parameters

- **data_id** (*str*) – Data node id.
- **value** (*T, optional*) – Data node default value.

Note: If *EMPTY* the previous default value is removed.

- **initial_dist** (*float, int, optional*) – Initial distance in the ArciDispatch algorithm when the data node default value is used.

****

Example:

A dispatcher with a data node named *a*:

```
>>> dsp = Dispatcher(name='Dispatcher')
...
>>> dsp.add_data(data_id='a')
'a'
```

Add a default value to *a* node:

```
>>> dsp.set_default_value('a', value='value of the data')
>>> list(sorted(dsp.default_values['a'].items()))
[('initial_dist', 0.0), ('value', 'value of the data')]
```

Remove the default value of *a* node:

```
>>> dsp.set_default_value('a', value=EMPTY)
>>> dsp.default_values
{}
```

shrink_dsp

Dispatcher.**shrink_dsp**(*inputs=None*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *wildcard=True*)

Returns a reduced dispatcher.

Parameters

- **inputs** (*list[str], iterable, optional*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.

Returns A sub-dispatcher.

Return type *Dispatcher*

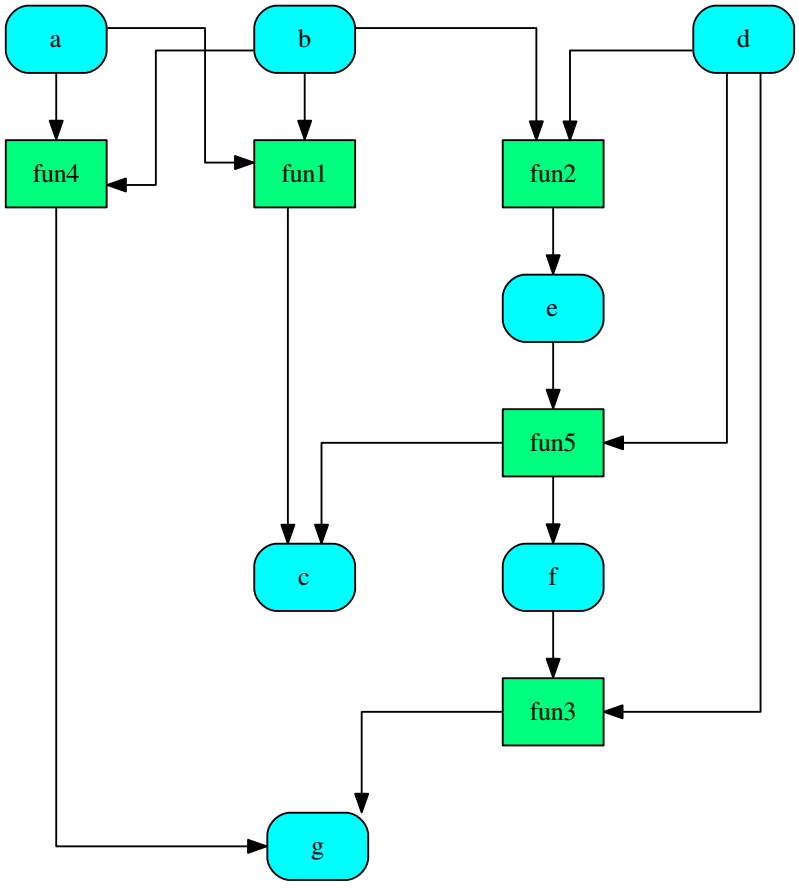
See also:

dispatch()

****

Example:

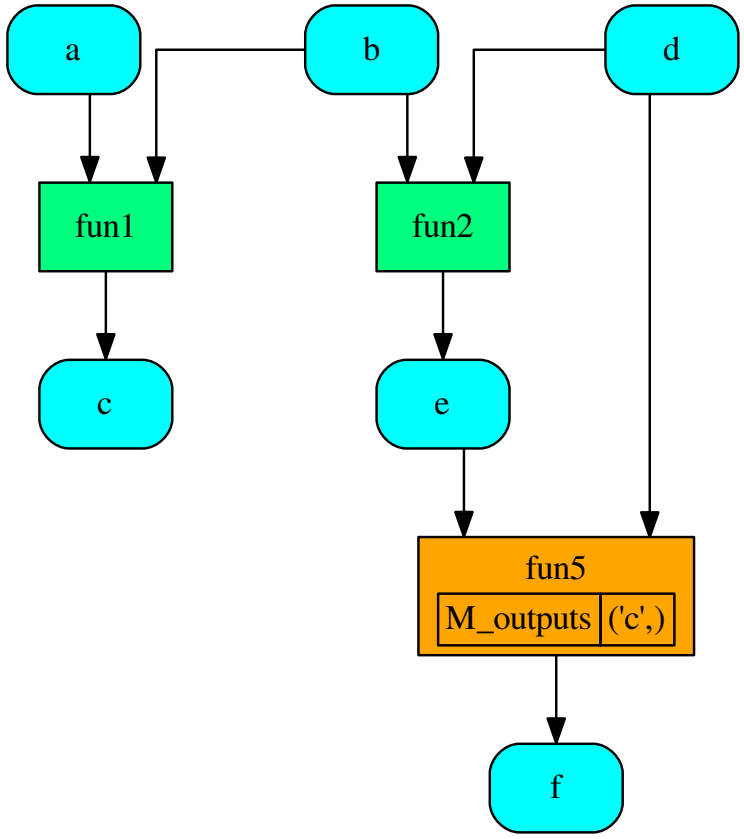
A dispatcher like this:



dmap

Get the sub-dispatcher induced by dispatching with no calls from inputs *a*, *b*, and *c* to outputs *c*, *e*, and *f*:

```
>>> shrink_dsp = dsp.shrink_dsp(inputs=['a', 'b', 'd'],
...                             outputs=['c', 'f'])
```



dmap

web

Dispatcher.**web**(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

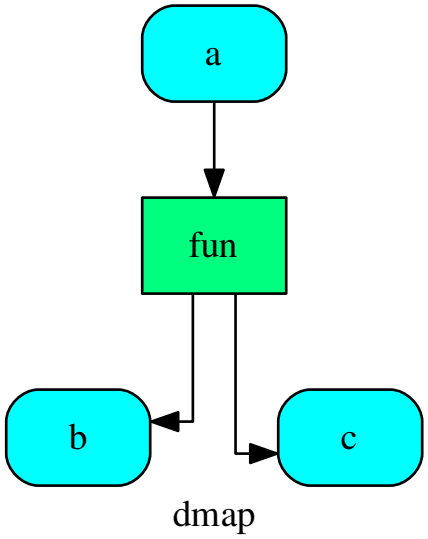
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

__init__(*dmap=None*, *name=''*, *default_values=None*, *raises=False*, *description=''*, *stopper=None*)

Initializes the dispatcher.

Parameters

- **dmap** (*networkx.DiGraph, optional*) – A directed graph that stores data & functions parameters.
- **name** (*str, optional*) – The dispatcher's name.
- **default_values** (*dict[str, dict], optional*) – Data node default values. These will be used as input if it is not specified as inputs in the ArciDispatch algorithm.
- **raises** (*bool, optional*) – If True the dispatcher interrupt the dispatch when an error occur, otherwise it logs a warning.
- **description** (*str, optional*) – The dispatcher's description.
- **stopper** (*threading.Event, optional*) – A semaphore to abort the dispatching.

Attributes

data_nodes	Returns all data nodes of the dispatcher.
function_nodes	Returns all function nodes of the dispatcher.
sub_dsp_nodes	Returns all sub-dispatcher nodes of the dispatcher.

data_nodes

Dispatcher.**data_nodes**
Returns all data nodes of the dispatcher.

Returns All data nodes of the dispatcher.

Return type dict[str, dict]

function_nodes

Dispatcher.**function_nodes**
Returns all function nodes of the dispatcher.

Returns All data function of the dispatcher.

Return type dict[str, dict]

sub_dsp_nodes

Dispatcher.**sub_dsp_nodes**
Returns all sub-dispatcher nodes of the dispatcher.

Returns All sub-dispatcher nodes of the dispatcher.

Return type dict[str, dict]

dmap = None
The directed graph that stores data & functions parameters.

name = None
The dispatcher's name.

nodes = None
The function and data nodes of the dispatcher.

default_values = None
Data node default values. These will be used as input if it is not specified as inputs in the ArciDispatch algorithm.

weight = None
Weight tag.

raises = None
If True the dispatcher interrupt the dispatch when an error occur.

stopper = <threading.Event object>
Stopper to abort the dispatcher execution.

solution = None
Last dispatch solution.

counter = None
Counter to set the node index.

add_data(*data_id=None*, *default_value=empty*, *initial_dist=0.0*, *wait_inputs=False*, *wildcard=None*, *function=None*, *callback=None*, *remote_links=None*, *description=None*, *filters=None*, ***kwargs*)
Add a single data node to the dispatcher.

Parameters

- **data_id** (*str, optional*) – Data node id. If None will be assigned automatically ('unknown<%d>') not in dmap.
- **default_value** (*T, optional*) – Data node default value. This will be used as input if it is not specified as inputs in the ArciDispatch algorithm.
- **initial_dist** (*float, int, optional*) – Initial distance in the ArciDispatch algorithm when the data node default value is used.
- **wait_inputs** (*bool, optional*) – If True ArciDispatch algorithm stops on the node until it gets all input estimations.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.
- **function** (*callable, optional*) – Data node estimation function. This can be any function that takes only one dictionary (key=function node id, value=estimation of data node) as input and return one value that is the estimation of the data node.
- **callback** (*callable, optional*) – Callback function to be called after node estimation. This can be any function that takes only one argument that is the data node estimation output. It does not return anything.
- **remote_links** (*list[[str, Dispatcher]], optional*) – List of parent or child dispatcher nodes e.g., [[dsp_id, dsp], ...].
- **description** (*str, optional*) – Data node's description.
- **filters** (*list[function], optional*) – A list of functions that are invoked after the invocation of the main function.

- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Data node id.

Return type str

See also:

add_function(), *add_dispatcher()*, *add_from_lists()*

****

Example:

Add a data to be estimated or a possible input data node:

```
>>> dsp.add_data(data_id='a')
'a'
```

Add a data with a default value (i.e., input data node):

```
>>> dsp.add_data(data_id='b', default_value=1)
'b'
```

Create a data node with function estimation and a default value.

•function estimation: estimate one unique output from multiple estimations.

•default value: is a default estimation.

```
>>> def min_fun(kwargs):
...     '''
...     Returns the minimum value of node estimations.
...
...     :param kwargs:
...         Node estimations.
...     :type kwargs: dict
...
...     :return:
...         The minimum value of node estimations.
...     :rtype: float
...     '''
...
...     return min(kwargs.values())
...
>>> dsp.add_data(data_id='c', default_value=2, wait_inputs=True,
...              function=min_fun)
'c'
```

Create a data with an unknown id and return the generated id:

```
>>> dsp.add_data()
'unknown'
```

add_function(*function_id=None*, *function=None*, *inputs=None*, *outputs=None*, *input_domain=None*, *weight=None*, *inp_weight=None*, *out_weight=None*, *description=None*, *filters=None*, ***kwargs*)

Add a single function node to dispatcher.

Parameters

- **function_id** (*str, optional*) – Function node id. If None will be assigned as <fun.__name__>.
- **function** (*callable, optional*) – Data node estimation function.
- **inputs** (*list, optional*) – Ordered arguments (i.e., data node ids) needed by the function.
- **outputs** (*list, optional*) – Ordered results (i.e., data node ids) returned by the function.
- **input_domain** (*callable, optional*) – A function that checks if input values satisfy the function domain. This can be any function that takes the same inputs of the function and returns True if input values satisfy the domain, otherwise False. In this case the dispatch algorithm doesn't pass on the node.
- **weight** (*float, int, optional*) – Node weight. It is a weight coefficient that is used by the dispatch algorithm to estimate the minimum workflow.
- **inp_weight** (*dict[str, float | int], optional*) – Edge weights from data nodes to the function node. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **out_weight** (*dict[str, float | int], optional*) – Edge weights from the function node to data nodes. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **description** (*str, optional*) – Function node's description.
- **filters** (*list[function], optional*) – A list of functions that are invoked after the invocation of the main function.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Function node id.

Return type str

See also:

add_data(), *add_dispatcher()*, *add_from_lists()*

****

Example:

Add a function node:

```
>>> def my_function(a, b):
...     c = a + b
...     d = a - b
...     return c, d
...
>>> dsp.add_function(function=my_function, inputs=['a', 'b'],
...                  outputs=['c', 'd'])
'my_function'
```

Add a function node with domain:

```
>>> from math import log
>>> def my_log(a, b):
...     return log(b - a)
...
```

```
>>> def my_domain(a, b):
...     return a < b
...
>>> dsp.add_function(function=my_log, inputs=['a', 'b'],
...                  outputs=['e'], input_domain=my_domain)
'my_log'
```

add_dispatcher(*dsp*, *inputs*, *outputs*, *dsp_id=None*, *input_domain=None*, *weight=None*, *inp_weight=None*, *description=None*, *include_defaults=False*, ***kwargs*)

Add a single sub-dispatcher node to dispatcher.

Parameters

- **dsp** (*Dispatcher | dict[str, list]*) – Child dispatcher that is added as sub-dispatcher node to the parent dispatcher.
- **inputs** (*dict[str, str | list[str]] | tuple[str] | (str, ..., dict[str, str | list[str]])*) – Inputs mapping. Data node ids from parent dispatcher to child sub-dispatcher.
- **outputs** (*dict[str, str | list[str]] | tuple[str] | (str, ..., dict[str, str | list[str]])*) – Outputs mapping. Data node ids from child sub-dispatcher to parent dispatcher.
- **dsp_id** (*str, optional*) – Sub-dispatcher node id. If None will be assigned as <dsp.name>.
- **input_domain** (*(dict) -> bool, optional*) – A function that checks if input values satisfy the function domain. This can be any function that takes the a dictionary with the inputs of the sub-dispatcher node and returns True if input values satisfy the domain, otherwise False.

  Note: This function is invoked every time that a data node reach the sub-dispatcher node.

- **weight** (*float, int, optional*) – Node weight. It is a weight coefficient that is used by the dispatch algorithm to estimate the minimum workflow.
- **inp_weight** (*dict[str, int | float], optional*) – Edge weights from data nodes to the sub-dispatcher node. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **description** (*str, optional*) – Sub-dispatcher node's description.
- **include_defaults** (*bool, optional*) – If True the default values of the sub-dispatcher are added to the current dispatcher.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Sub-dispatcher node id.

Return type str

See also:

add_data(), *add_function()*, *add_from_lists()*

****

Example:

Create a sub-dispatcher:

```
>>> sub_dsp = Dispatcher()
>>> sub_dsp.add_function('max', max, ['a', 'b'], ['c'])
'max'
```

Add the sub-dispatcher to the parent dispatcher:

```
>>> dsp.add_dispatcher(dsp_id='Sub-Dispatcher', dsp=sub_dsp,
...                    inputs={'A': 'a', 'B': 'b'},
...                    outputs={'c': 'C'})
'Sub-Dispatcher'
```

Add a sub-dispatcher node with domain:

```
>>> def my_domain(kwargs):
...     return kwargs['C'] > 3
...
>>> dsp.add_dispatcher(dsp_id='Sub-Dispatcher with domain',
...                    dsp=sub_dsp, inputs={'C': 'a', 'D': 'b'},
...                    outputs={'c': 'E'}, input_domain=my_domain)
'Sub-Dispatcher with domain'
```

add_from_lists(*data_list=None*, *fun_list=None*, *dsp_list=None*)

Add multiple function and data nodes to dispatcher.

Parameters

- **data_list** (*list[dict], optional*) – It is a list of data node kwargs to be loaded.
- **fun_list** (*list[dict], optional*) – It is a list of function node kwargs to be loaded.
- **dsp_list** (*list[dict], optional*) – It is a list of sub-dispatcher node kwargs to be loaded.

Returns

- Data node ids.
- Function node ids.
- Sub-dispatcher node ids.

Return type (list[str], list[str], list[str])

See also:

add_data(), *add_function()*, *add_dispatcher()*

****

Example:

Define a data list:

```
>>> data_list = [
...     {'data_id': 'a'},
...     {'data_id': 'b'},
...     {'data_id': 'c'},
... ]
```

Define a functions list:

```
>>> def func(a, b):
...     return a + b
...
>>> fun_list = [
...     {'function': func, 'inputs': ['a', 'b'], 'outputs': ['c']}
... ]
```

Define a sub-dispatchers list:

```
>>> sub_dsp = Dispatcher(name='Sub-dispatcher')
>>> sub_dsp.add_function(function=func, inputs=['e', 'f'],
...                      outputs=['g'])
'func'
>>>
>>> dsp_list = [
...     {'dsp_id': 'Sub', 'dsp': sub_dsp,
...      'inputs': {'a': 'e', 'b': 'f'}, 'outputs': {'g': 'c'}},
... ]
```

Add function and data nodes to dispatcher:

```
>>> dsp.add_from_lists(data_list, fun_list, dsp_list)
(['a', 'b', 'c'], ['func'], ['Sub'])
```

set_default_value(*data_id*, *value=empty*, *initial_dist=0.0*)

Set the default value of a data node in the dispatcher.

Parameters

- **data_id** (*str*) – Data node id.
- **value** (*T, optional*) – Data node default value.

  Note: If *EMPTY* the previous default value is removed.

- **initial_dist** (*float, int, optional*) – Initial distance in the ArciDispatch algorithm when the data node default value is used.

****

Example:

A dispatcher with a data node named *a*:

```
>>> dsp = Dispatcher(name='Dispatcher')
...
>>> dsp.add_data(data_id='a')
'a'
```

Add a default value to *a* node:

```
>>> dsp.set_default_value('a', value='value of the data')
>>> list(sorted(dsp.default_values['a'].items()))
[('initial_dist', 0.0), ('value', 'value of the data')]
```

Remove the default value of *a* node:

```
>>> dsp.set_default_value('a', value=EMPTY)
>>> dsp.default_values
{}
```

set_data_remote_link(*data_id*, *remote_link=empty*, *is_parent=True*)
Set a remote link of a data node in the dispatcher.

Parameters

- **data_id** (*str*) – Data node id.
- **remote_link** (*[str, Dispatcher], optional*) – Parent or child dispatcher and its node id (id, dsp).
- **is_parent** (*bool*) – If True the link is inflow (parent), otherwise is outflow (child).

get_sub_dsp(*nodes_bunch*, *edges_bunch=None*)
Returns the sub-dispatcher induced by given node and edge bunches.

The induced sub-dispatcher contains the available nodes in nodes_bunch and edges between those nodes, excluding those that are in edges_bunch.

The available nodes are non isolated nodes and function nodes that have all inputs and at least one output.

Parameters

- **nodes_bunch** (*list[str], iterable*) – A container of node ids which will be iterated through once.
- **edges_bunch** (*list[(str, str)], iterable, optional*) – A container of edge ids that will be removed.

Returns A dispatcher.

Return type *Dispatcher*

See also:

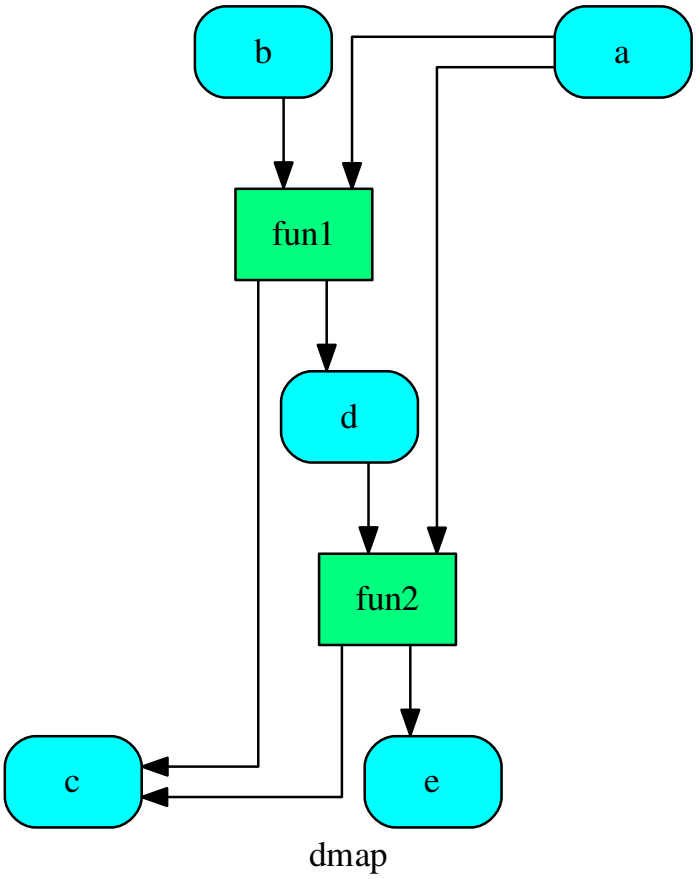
get_sub_dsp_from_workflow()

Note: The sub-dispatcher edge or node attributes just point to the original dispatcher. So changes to the node or edge structure will not be reflected in the original dispatcher map while changes to the attributes will.

****

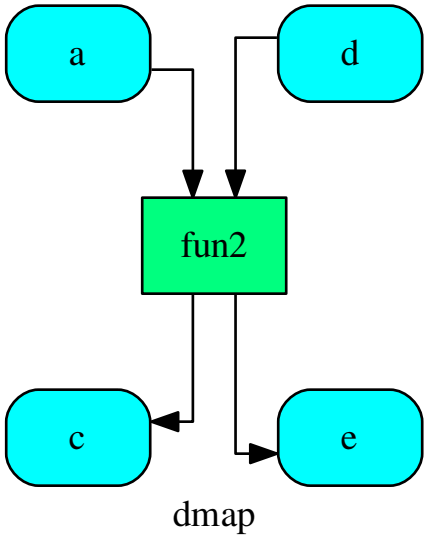
Example:

A dispatcher with a two functions *fun1* and *fun2*:



Get the sub-dispatcher induced by given nodes bunch:

```
>>> sub_dsp = dsp.get_sub_dsp(['a', 'c', 'd', 'e', 'fun2'])
```



get_sub_dsp_from_workflow(*sources*, *graph=None*, *reverse=False*, *add_missing=False*, *check_inputs=True*, *blockers=None*, *wildcard=False*, *_update_links=True*)

Returns the sub-dispatcher induced by the workflow from sources.

The induced sub-dispatcher of the dsp contains the reachable nodes and edges evaluated with breadth-first-search on the workflow graph from source nodes.

Parameters

- **sources** (*list[str], iterable*) – Source nodes for the breadth-first-search. A container of nodes which will be iterated through once.
- **graph** (*networkx.DiGraph, optional*) – A directed graph where evaluate the breadth-first-search.
- **reverse** (*bool, optional*) – If True the workflow graph is assumed as reversed.
- **add_missing** (*bool, optional*) – If True, missing function' inputs are added to the sub-dispatcher.
- **check_inputs** (*bool, optional*) – If True the missing function' inputs are not checked.
- **blockers** (*set[str], iterable, optional*) – Nodes to not be added to the queue.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.
- **_update_links** (*bool, optional*) – If True, it updates remote links of the extracted dispatcher.

Returns A sub-dispatcher.

Return type *Dispatcher*

See also:

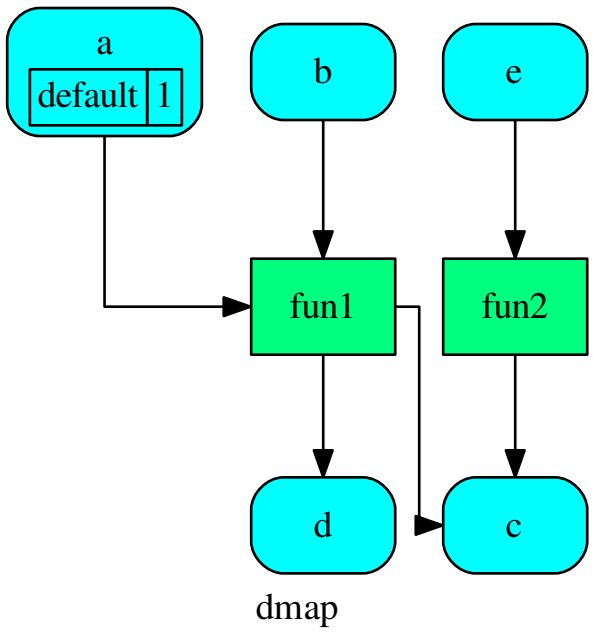
get_sub_dsp()

Note: The sub-dispatcher edge or node attributes just point to the original dispatcher. So changes to the node or edge structure will not be reflected in the original dispatcher map while changes to the attributes will.

****

Example:

A dispatcher with a function *fun* and a node *a* with a default value:

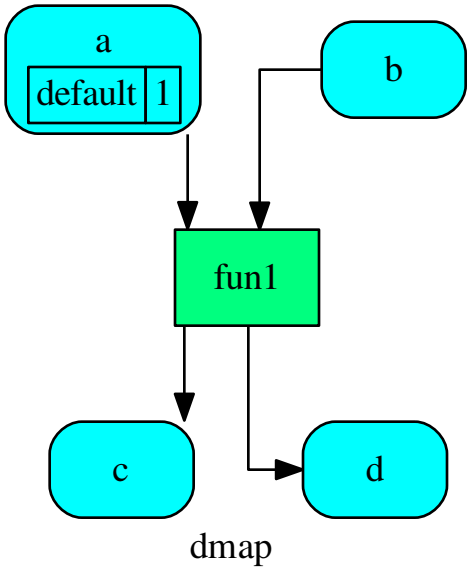


Dispatch with no calls in order to have a workflow:

```
>>> o = dsp.dispatch(inputs=['a', 'b'], no_call=True)
```

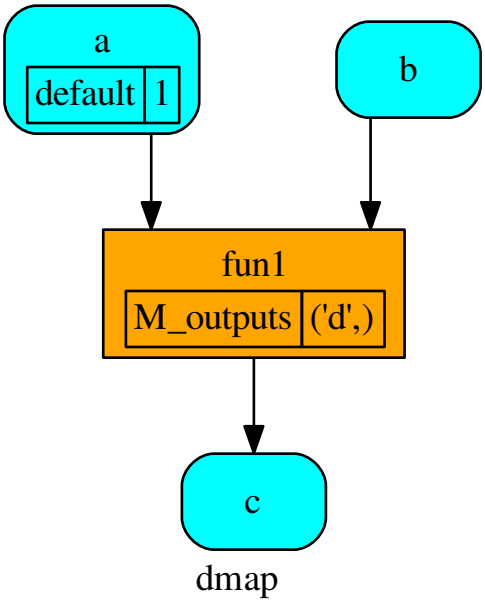
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> sub_dsp = dsp.get_sub_dsp_from_workflow(['a', 'b'])
```



Get sub-dispatcher from a workflow output *c*:

```
>>> sub_dsp = dsp.get_sub_dsp_from_workflow(['c'], reverse=True)
```



data_nodes
Returns all data nodes of the dispatcher.

Returns All data nodes of the dispatcher.

Return type dict[str, dict]

function_nodes
Returns all function nodes of the dispatcher.

Returns All data function of the dispatcher.

Return type dict[str, dict]

sub_dsp_nodes
Returns all sub-dispatcher nodes of the dispatcher.

Returns All sub-dispatcher nodes of the dispatcher.

Return type dict[str, dict]

copy()
Returns a copy of the Dispatcher.

Returns A copy of the Dispatcher.

Return type *Dispatcher*

Example:

```
>>> dsp = Dispatcher()
>>> dsp is dsp.copy()
False
```

dispatch(*inputs=None*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *wildcard=False*, *no_call=False*, *shrink=False*, *rm_unused_nds=False*, *select_output_kw=None*, *_wait_in=None*, *stopper=None*)

Evaluates the minimum workflow and data outputs of the dispatcher model from given inputs.

Parameters

- **inputs** (*dict[str, T], list[str], iterable, optional*) – Input data values.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.
- **no_call** (*bool, optional*) – If True data node estimation function is not used and the input values are not used.
- **shrink** (*bool, optional*) – If True the dispatcher is shrink before the dispatch.

  See also:

  shrink_dsp()
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **select_output_kw** (*dict, optional*) – Kwargs of selector function to select specific outputs.
- **_wait_in** (*dict, optional*) – Override wait inputs.
- **stopper** (*threading.Event, optional*) – A semaphore to abort the dispatching.

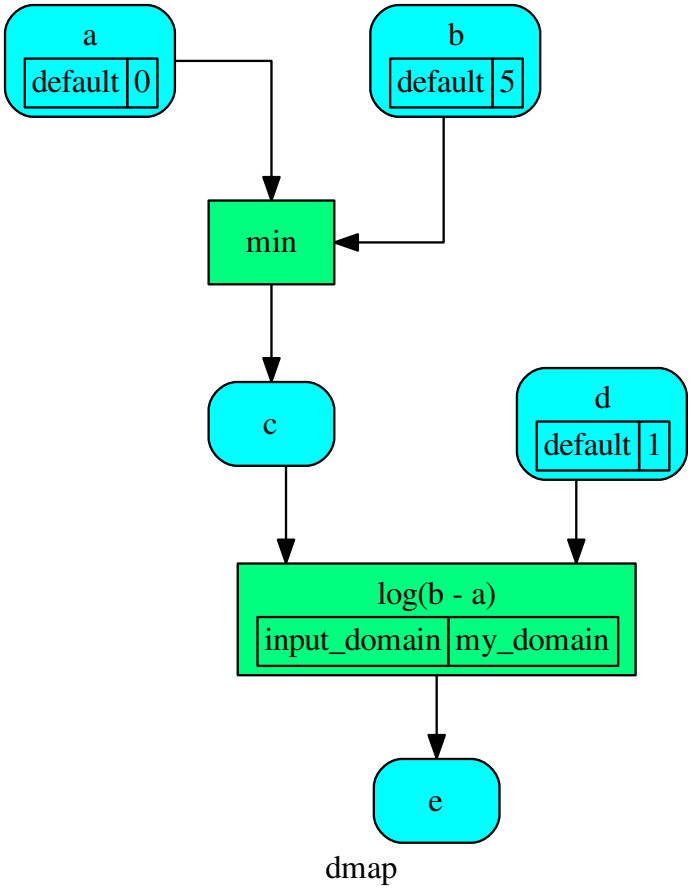
Returns Dictionary of estimated data node outputs.

Return type *schedula.utils.sol.Solution*

****

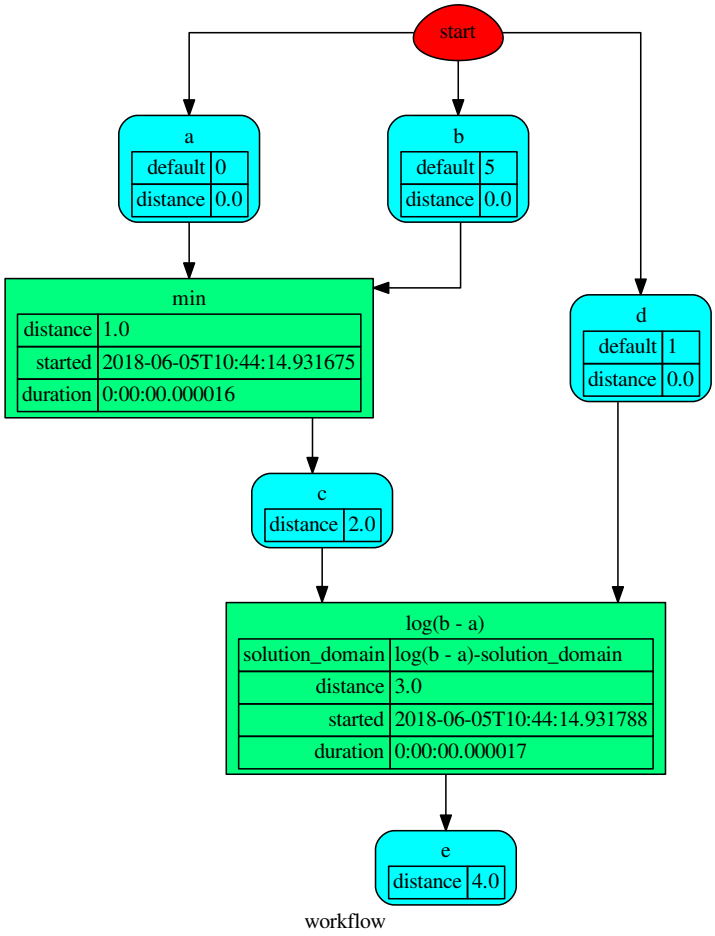
Example:

A dispatcher with a function $log(b - a)$ and two data a and b with default values:



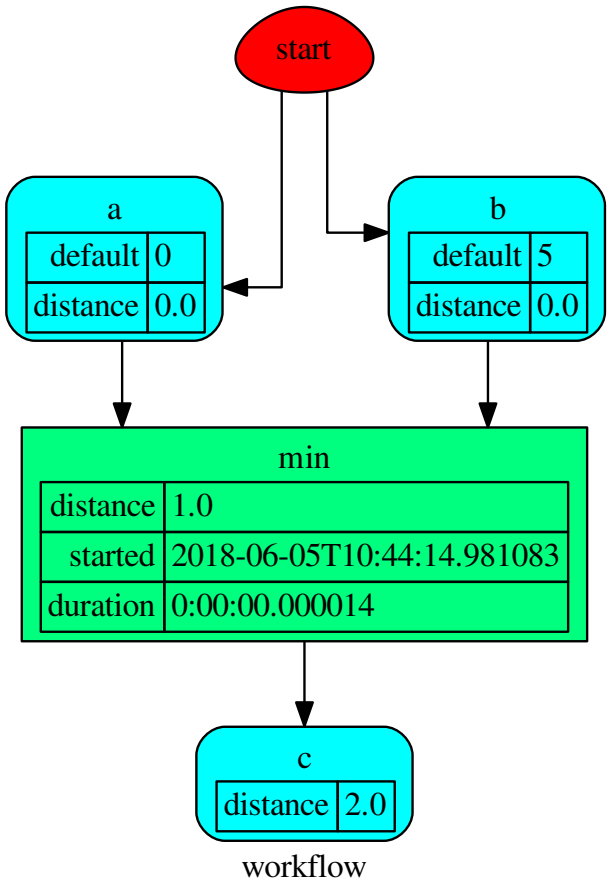
Dispatch without inputs. The default values are used as inputs:

```
>>> outputs = dsp.dispatch()
>>> outputs
Solution([('a', 0), ('b', 5), ('d', 1), ('c', 0), ('e', 0.0)])
```



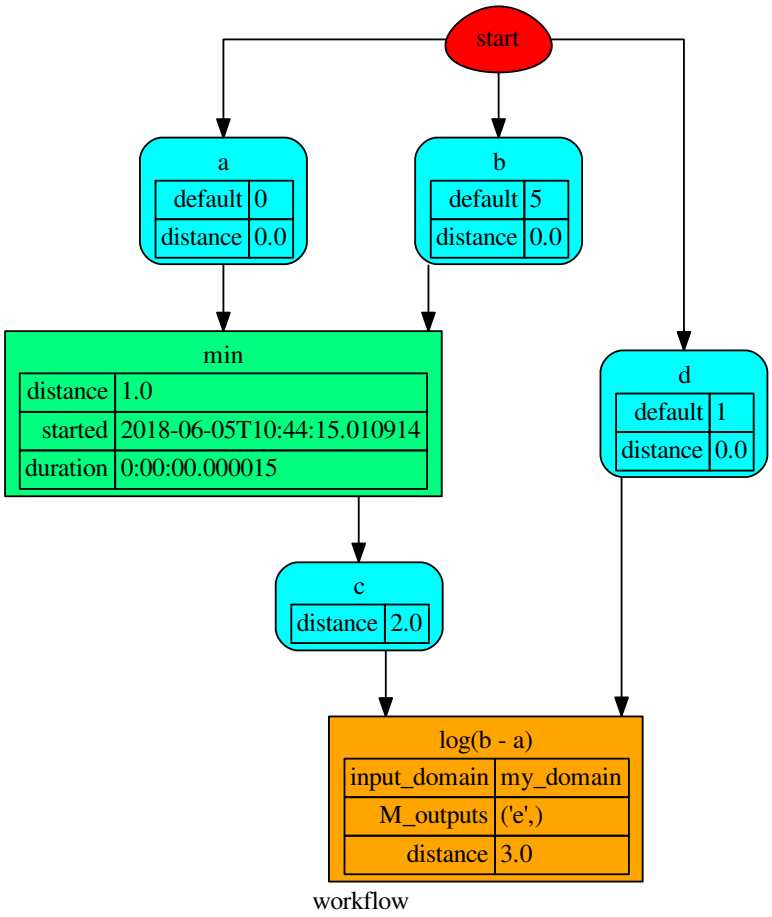
Dispatch until data node *c* is estimated:

```
>>> outputs = dsp.dispatch(outputs=['c'])
>>> outputs
Solution([('a', 0), ('b', 5), ('c', 0)])
```



Dispatch with one inputs. The default value of *a* is not used as inputs:

```
>>> outputs = dsp.dispatch(inputs={'a': 3})
>>> outputs
Solution([('a', 3), ('b', 5), ('d', 1), ('c', 3)])
```



shrink_dsp(*inputs=None*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *wildcard=True*)

Returns a reduced dispatcher.

Parameters

- **inputs** (*list[str], iterable, optional*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.

Returns A sub-dispatcher.

Return type *Dispatcher*

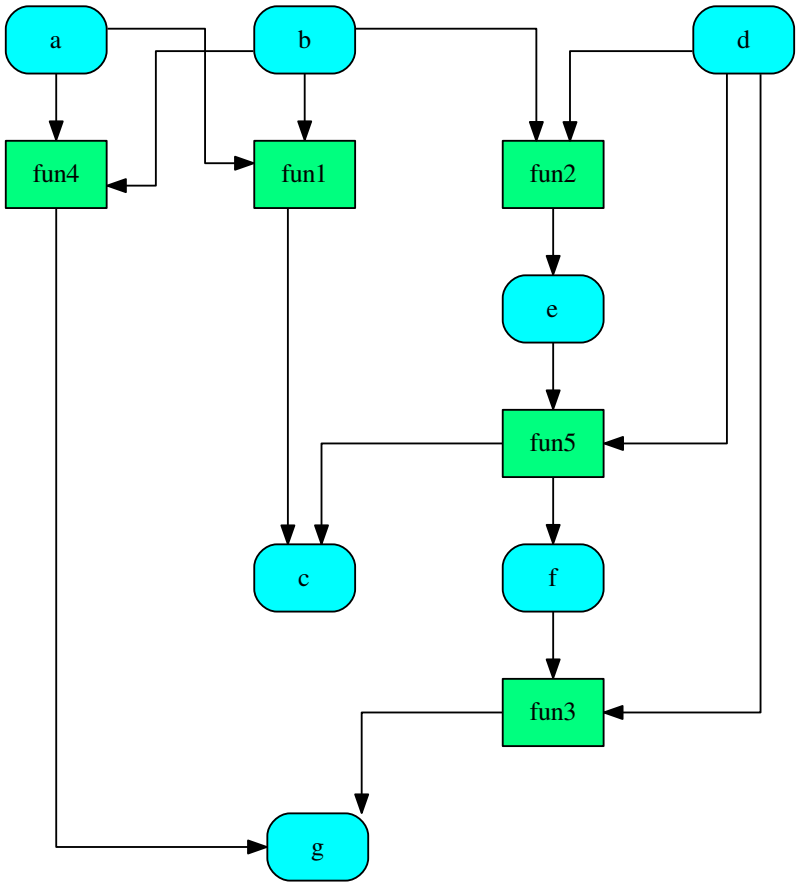
See also:

dispatch()

****

Example:

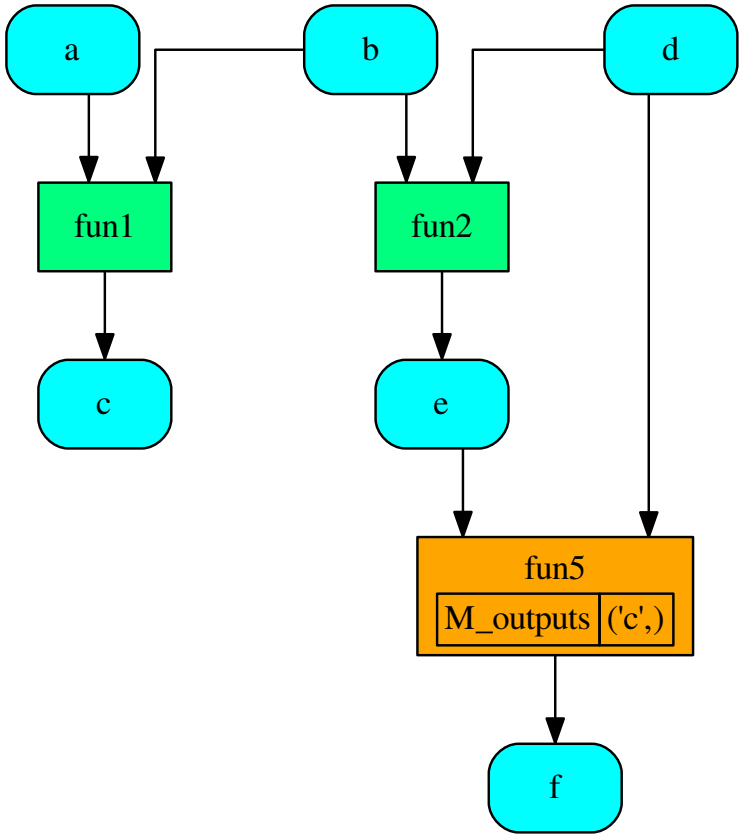
A dispatcher like this:



dmap

Get the sub-dispatcher induced by dispatching with no calls from inputs *a*, *b*, and *c* to outputs *c*, *e*, and *f*:

```
>>> shrink_dsp = dsp.shrink_dsp(inputs=['a', 'b', 'd'],
...                             outputs=['c', 'f'])
```



dmap

utils

It contains utility classes and functions.

The utils module contains classes and functions of general utility used in multiple places throughout *schedula*. Some of these are graph-specific algorithms while others are more python tricks.

The utils module is composed of eleven submodules to make organization clearer. The submodules are fairly different from each other, but the main uniting theme is that all of these submodules are not specific to a particularly schedula application.

Note: The `utils` module is composed of submodules that can be accessed separately. However, they are all also included in the base module. Thus, as an example, schedula.utils.gen.Token and schedula.utils.Token are different names for the same class (Token). The schedula.utils.Token usage is preferred as this allows the internal organization to be changed if it is deemed necessary.

Sub-Modules:

alg	It contains basic algorithms, numerical tricks, and data processing tasks.
base	It provides a base class to for dispatcher objects.
cst	It provides constants data node ids and values.
des	It provides tools to find data, function, and sub-dispatcher node description.
drw	It provides functions to plot dispatcher map and workflow.
dsp	It provides tools to create models with the `Dispatcher()`.
exc	Defines the dispatcher exception.
gen	It contains classes and functions of general utility.
io	It provides functions to read and save a dispatcher from/to files.
sol	It contains a comprehensive list of all modules and classes within dispatcher.
web	It provides functions to build a flask app from a dispatcher.

alg

It contains basic algorithms, numerical tricks, and data processing tasks.

Functions

add_edge_fun	Returns a function that adds an edge to the *graph* checking only the out node.
add_func_edges	Adds function node edges.
get_full_pipe	Returns the full pipe of a dispatch run.
get_sub_node	Returns a sub node of a dispatcher.
get_unused_node_id	Finds an unused node id in *graph*.
remove_edge_fun	Returns a function that removes an edge from the *graph*.
remove_links	
replace_remote_link	Replaces or removes remote links.

add_edge_fun

add_edge_fun(*graph*)

Returns a function that adds an edge to the *graph* checking only the out node.

Parameters **graph** (*networkx.classes.digraph.DiGraph*) – A directed graph.

Returns A function that adds an edge to the *graph*.

Return type callable

add_func_edges

add_func_edges(*dsp*, *fun_id*, *nodes_bunch*, *edge_weights=None*, *input=True*, *data_nodes=None*)

Adds function node edges.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **fun_id** (*str*) – Function node id.
- **nodes_bunch** (*iterable*) – A container of nodes which will be iterated through once.
- **edge_weights** (*dict, optional*) – Edge weights.
- **input** (*bool, optional*) – If True the nodes_bunch are input nodes, otherwise are output nodes.
- **data_nodes** (*list*) – Data nodes to be deleted if something fail.

Returns List of new data nodes.

Return type list

get_full_pipe

get_full_pipe(*sol*, *base=()*)
Returns the full pipe of a dispatch run.

Parameters

- **sol** (*schedula.utils.Solution*) – A Solution object.
- **base** (*tuple[str]*) – Base node id.

Returns Full pipe of a dispatch run.

Return type *DspPipe*

get_sub_node

get_sub_node(*dsp*, *path*, *node_attr='auto'*, *solution=none*, *_level=0*, *_dsp_name=none*)
Returns a sub node of a dispatcher.

Parameters

- **dsp** (*schedula.Dispatcher | SubDispatch*) – A dispatcher object or a sub dispatch function.
- **path** (*tuple, str*) – A sequence of node ids or a single node id. Each id identifies a sub-level node.
- **node_attr** (*str | None*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  – for data node: its output, and if not exists, all its attributes.

  – for function and sub-dispatcher nodes: the 'function' attribute.
- **solution** (*schedula.utils.Solution*) – Parent Solution.
- **_level** (*int*) – Path level.
- **_dsp_name** (*str*) – dsp name to show when the function raise a value error.

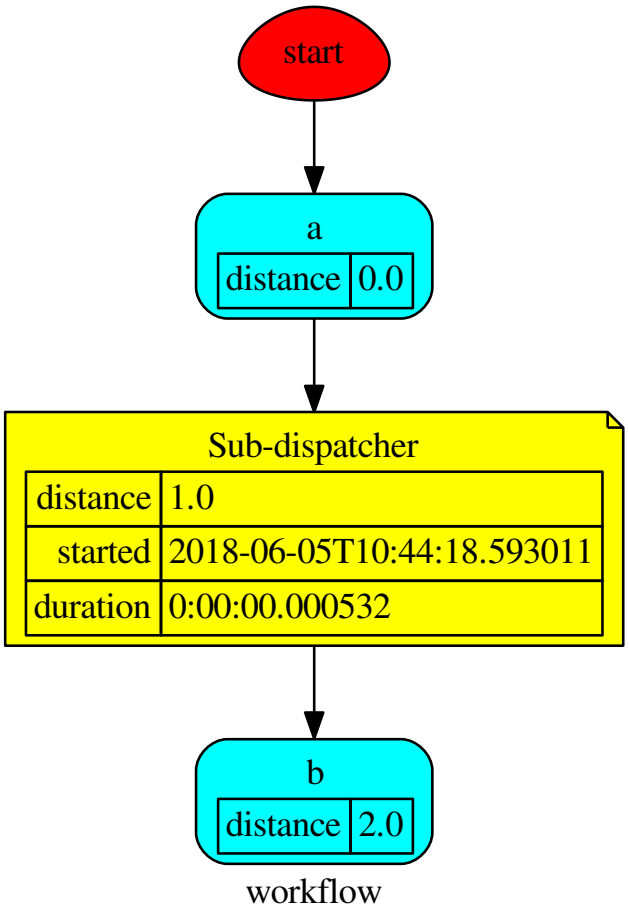
Returns A sub node of a dispatcher and its path.

Return type dict | object, tuple[str]

Example:

```
>>> from schedula import Dispatcher
>>> s_dsp = Dispatcher(name='Sub-dispatcher')
>>> def fun(a, b):
...     return a + b
...
>>> s_dsp.add_function('a + b', fun, ['a', 'b'], ['c'])
'a + b'
>>> dispatch = SubDispatch(s_dsp, ['c'], output_type='dict')
>>> dsp = Dispatcher(name='Dispatcher')
>>> dsp.add_function('Sub-dispatcher', dispatch, ['a'], ['b'])
'Sub-dispatcher'
```

```
>>> o = dsp.dispatch(inputs={'a': {'a': 3, 'b': 1}})
...
```

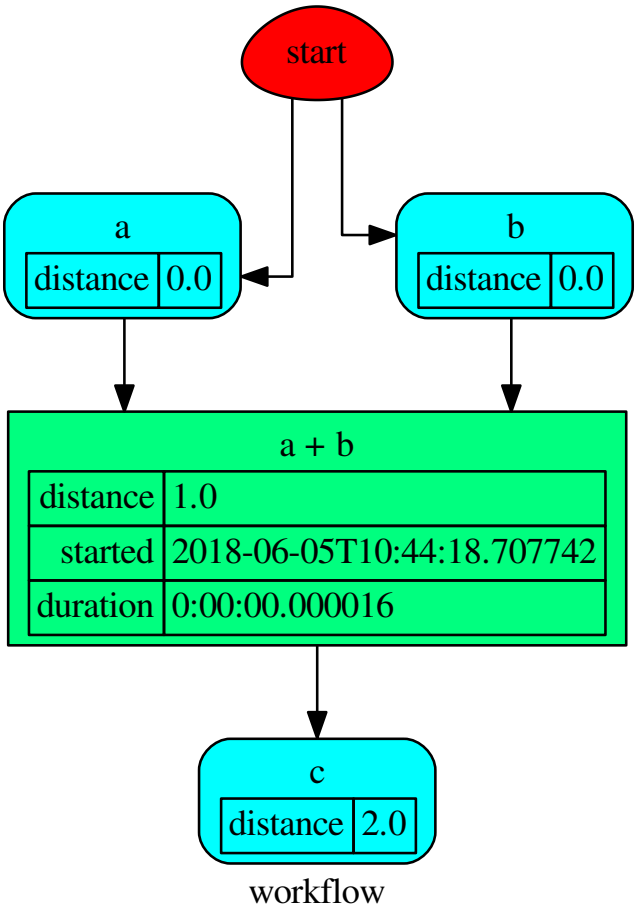


Get the sub node ‘c’ output or type:

```
>>> get_sub_node(dsp, ('Sub-dispatcher', 'c'))
(4, ('Sub-dispatcher', 'c'))
>>> get_sub_node(dsp, ('Sub-dispatcher', 'c'), node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

Get the sub-dispatcher output:

```
>>> sol, p = get_sub_node(dsp, ('Sub-dispatcher',), node_attr='output')
>>> sol, p
(Solution([('a', 3), ('b', 1), ('c', 4)]), ('Sub-dispatcher',))
```



get_unused_node_id

get_unused_node_id(*graph*, *initial_guess='unknown'*, *_format='{}<%d>'*)

Finds an unused node id in *graph*.

Parameters

- **graph** (*networkx.classes.digraph.DiGraph*) – A directed graph.
- **initial_guess** (*str, optional*) – Initial node id guess.
- **_format** (*str, optional*) – Format to generate the new node id if the given is already used.

Returns An unused node id.

Return type str

remove_edge_fun

remove_edge_fun(*graph*)

Returns a function that removes an edge from the *graph*.

..note:: The out node is removed if this is isolate.

Parameters **graph** (*networkx.classes.digraph.DiGraph*) – A directed graph.

Returns A function that remove an edge from the *graph*.

Return type callable

remove_links

remove_links(*dsp*)

replace_remote_link

replace_remote_link(*dsp*, *nodes_bunch*, *link_map*)

Replaces or removes remote links.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher with remote links.
- **nodes_bunch** (*iterable*) – A container of nodes which will be iterated through once.
- **link_map** (*dict*) – A dictionary that maps the link keys ({old link: new link}

Classes

DspPipe

DspPipe

class DspPipe

Methods

clear

Continued on next page

Table 2.8 – continued from previous page

copy	
fromkeys	If not specified, the value defaults to None.
get	
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last==False).
pop	value. If key is not found, d is returned if given, otherwise KeyError
popitem	Pairs are returned in LIFO order if last is true or FIFO order if false.
setdefault	
update	
values	

clear

DspPipe.**clear**() → None. Remove all items from od.

copy

DspPipe.**copy**() → a shallow copy of od

fromkeys

DspPipe.**fromkeys**(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get

DspPipe.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items

DspPipe.**items**()

keys

DspPipe.**keys**()

move_to_end

DspPipe.**move_to_end**()
Move an existing element to the end (or beginning if last==False).

Raises KeyError if the element does not exist. When last=True, acts like a fast version of self[key]=self.pop(key).

pop

DspPipe.**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem

DspPipe.**popitem**() → (k, v), return and remove a (key, value) pair. Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault

DspPipe.**setdefault**(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

update

DspPipe.**update**()

values

DspPipe.**values**()

__init__()
Initialize self. See help(type(self)) for accurate signature.

base

It provides a base class to for dispatcher objects.

Classes

Base

Base

class Base

Methods

get_node	Returns a sub node of a dispatcher.

Continued on next page

Table 2.10 – continued from previous page

plot	Plots the Dispatcher with a graph in the DOT language with Graphviz.
search_node_description	
web	Creates a dispatcher Flask app.

get_node

Base.**get_node**(**node_ids*, ***, *node_attr=none*)

Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  – for data node: its output, and if not exists, all its attributes.

  – for function and sub-dispatcher nodes: the 'function' attribute.

  When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

  When 'output', returns the data node output.

  When 'default_value', returns the data node default value.

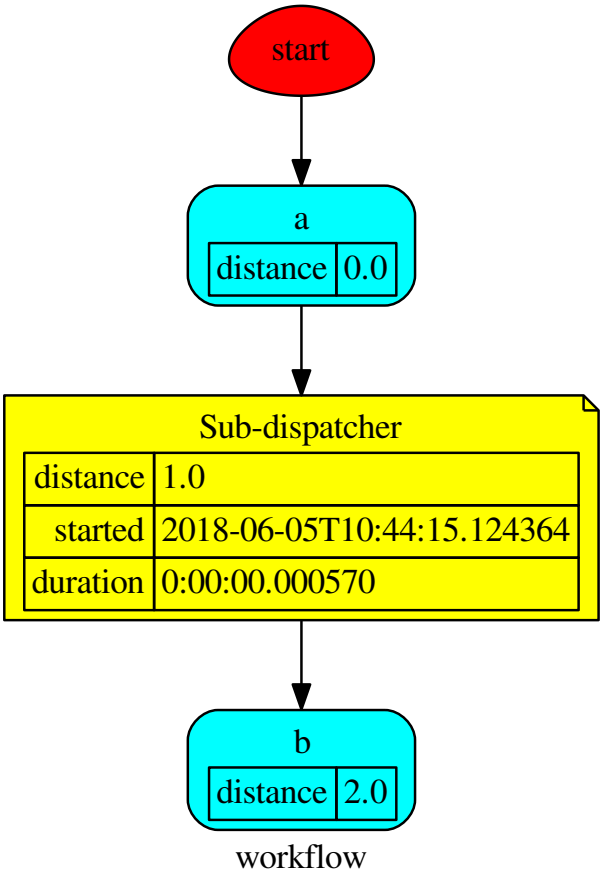
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

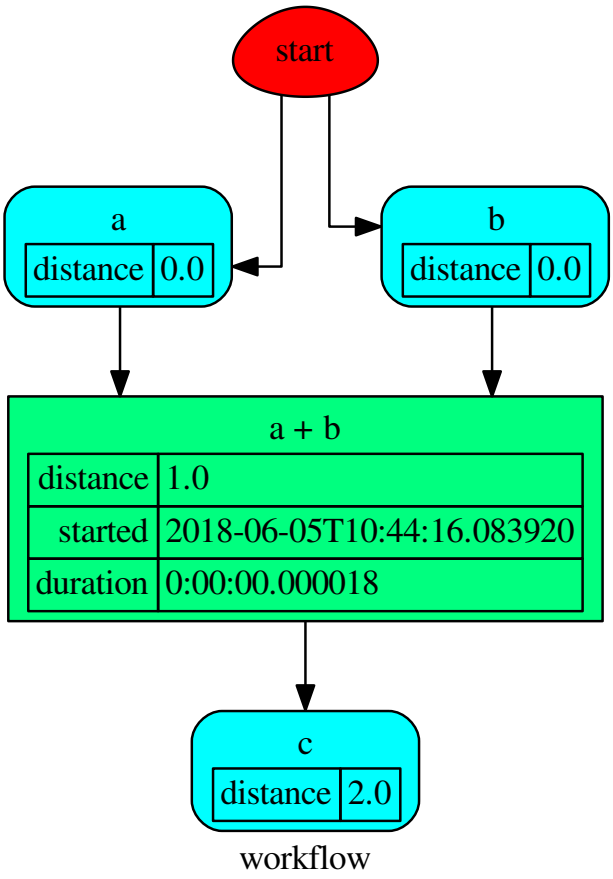
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



plot

`Base.`**`plot`**(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)

Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **`workflow`** (*`bool, optional`*) – If True the latest solution will be plotted, otherwise the dmap.
- **`view`** (*`bool, optional`*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **`edge_data`** (*`tuple[str], optional`*) – Edge attributes to view.
- **`node_data`** (*`tuple[str], optional`*) – Data node attributes to view.
- **`node_function`** (*`tuple[str], optional`*) – Function node attributes to view.

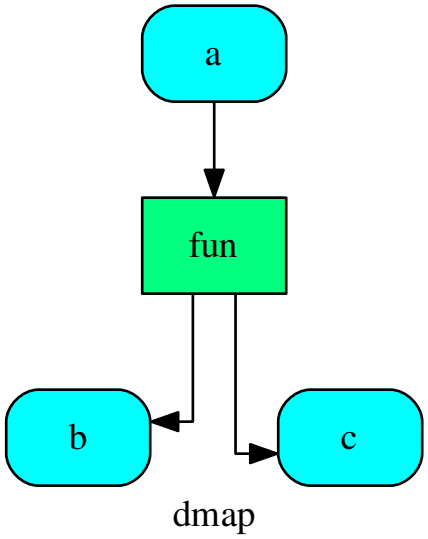
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



search_node_description

`Base.`**`search_node_description`**(*node_id*, *what='description'*)

web

`Base.`**`web`**(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

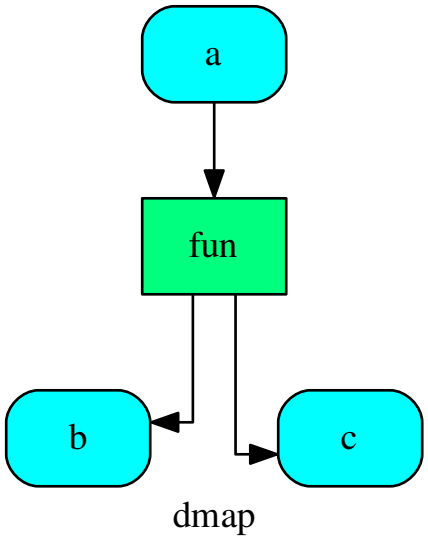
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

__init__()
Initialize self. See help(type(self)) for accurate signature.

web(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)
Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

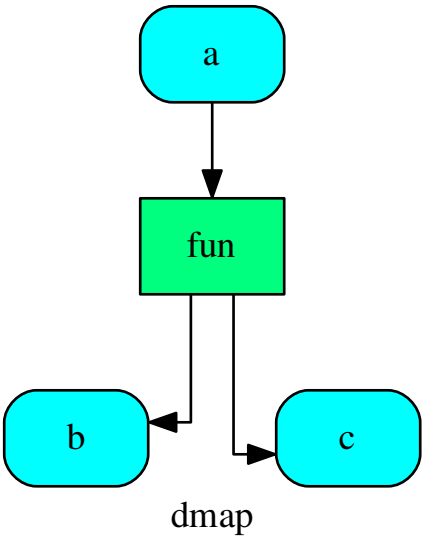
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
```

```
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

plot(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)

Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **workflow** (*bool, optional*) – If True the latest solution will be plotted, otherwise the dmap.
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?

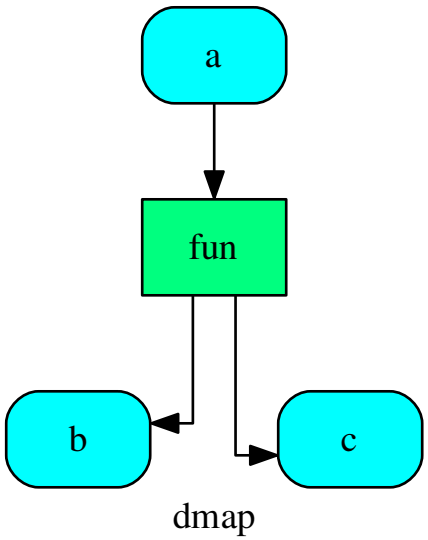
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



get_node(**node_ids*, ***, *node_attr=none*)

Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

- for data node: its output, and if not exists, all its attributes.
- for function and sub-dispatcher nodes: the 'function' attribute.

When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

When 'output', returns the data node output.

When 'default_value', returns the data node default value.

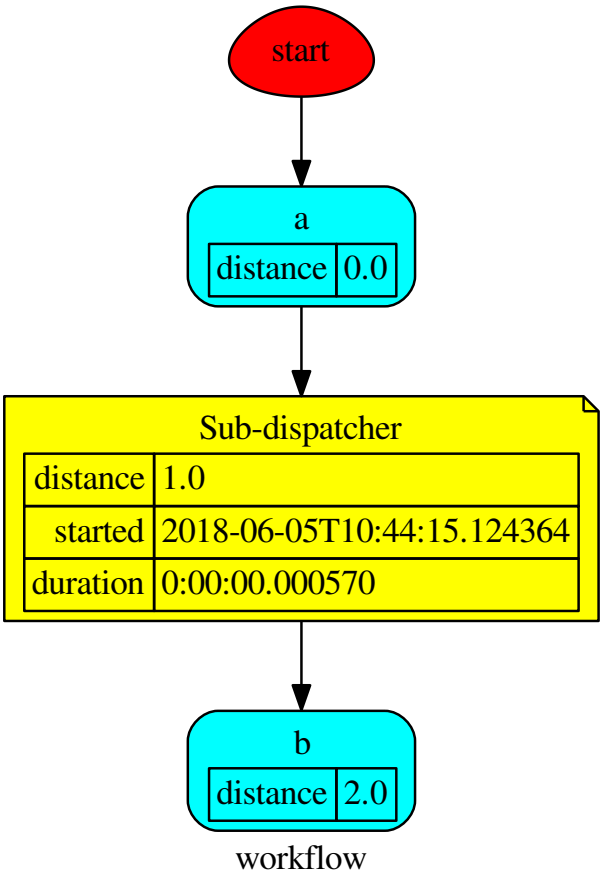
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

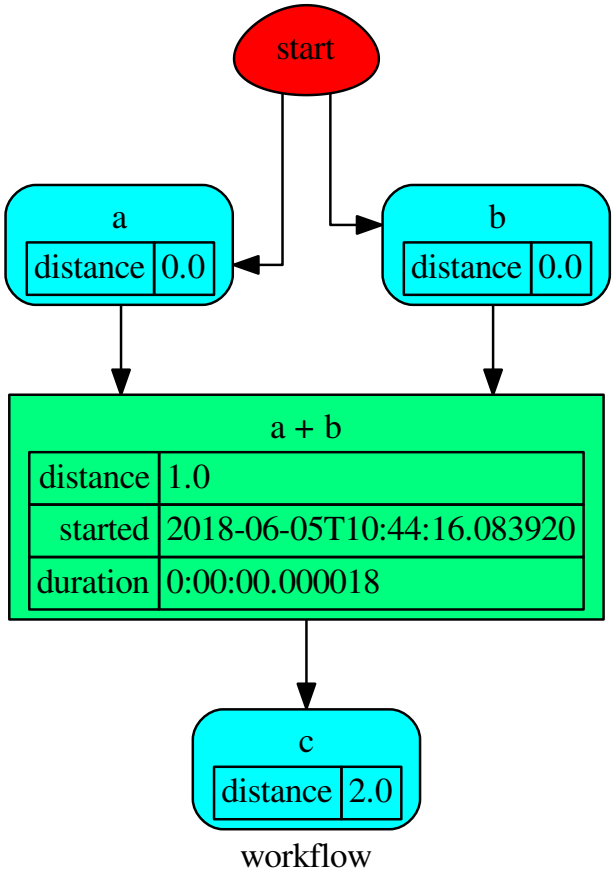
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



cst

It provides constants data node ids and values.

EMPTY = empty

It is used set and unset empty values.

See also:

`set_default_value()`

START = start
Starting node that identifies initial inputs of the workflow.

See also:

`dispatch()`

NONE = none
Fake value used to set a default value to call functions without arguments.

See also:

`add_function()`

SINK = sink
Sink node of the dispatcher that collects all unused outputs.

See also:

`add_data()`, `add_function()`, `add_dispatcher()`

END = end
Ending node of SubDispatcherFunction.

See also:

SubDispatchFunction()

SELF = self
Self node of the dispatcher, it is a node that contains the dispatcher.

PLOT = plot
Plot node, it is a node that plot the dispatcher solution. .. note:: you can pass the *kwargs* of `_DspPlot` .. seealso:: `add_data()`, `add_function()`, `add_dispatcher()`

des

It provides tools to find data, function, and sub-dispatcher node description.

Functions

get_attr_doc
get_link
get_summary
search_node_description

get_attr_doc

get_attr_doc(*doc*, *attr_name*, *get_param=True*, *what='description'*)

get_link

get_link(**items*)

get_summary

get_summary(*doc*)

search_node_description

search_node_description(*node_id*, *node_attr*, *dsp*, *what='description'*)

drw

It provides functions to plot dispatcher map and workflow.

Functions

add_header
autoplot_callback
autoplot_function
basic_app
before_request
cached_view
jinja2_format
render_output
run_server
site_view
uncpath
update_filenames
valid_filename

add_header

add_header(*filepath*, *header*)

autoplot_callback

autoplot_callback(*res*)

autoplot_function

autoplot_function(*kwargs*)

basic_app

basic_app(*root_path*, *cleanup=None*, *shutdown=None*, ***kwargs*)

before_request

before_request()

cached_view

cached_view(*node*, *directory*, *context*, *rendered*, *header*)

jinja2_format

jinja2_format(*source*, *context=None*, ***kw*)

render_output

render_output(*out*, *pformat*)

run_server

run_server(*app*, *options*)

site_view

site_view(*app*, *node*, *context*, *generated_files*, *rendered*, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*)

uncpath

uncpath(*p*)

update_filenames

update_filenames(*node*, *filenames*)

valid_filename

valid_filename(*item*, *filenames*, *ext=None*)

Classes

FolderNode
Site
SiteFolder

Continued on next page

Table 2.13 – continued from previous page

SiteIndex
SiteMap
SiteNode

FolderNode

class FolderNode (*folder*, *node_id*, *attr*, ***options*)

Methods

__init__
dot
href
items
parent_ref
render_funcs
render_size
style
yield_attr

__init__

FolderNode.**__init__** (*folder*, *node_id*, *attr*, ***options*)

dot

FolderNode.**dot** (*context=None*)

href

FolderNode.**href** (*context*, *link_id*)

items

FolderNode.**items** ()

parent_ref

FolderNode.**parent_ref** (*context*, *text*, *attr=None*)

render_funcs

FolderNode.**render_funcs** ()

render_size

FolderNode.**render_size**(*out*)

style

FolderNode.**style**()

yield_attr

FolderNode.**yield_attr**(*name*)

__init__(*folder*, *node_id*, *attr*, ***options*)

Attributes

counter
edge_data
max_lines
max_width
node_data
node_function
node_map
node_styles
re_node
title
type

counter

FolderNode.**counter** = <method-wrapper '__next__' of itertools.count object>

edge_data

FolderNode.**edge_data** = ('?', 'inp_id', 'out_id', 'weight')

max_lines

FolderNode.**max_lines** = 5

max_width

FolderNode.**max_width** = 200

node_data

FolderNode.**node_data** = (‘-‘, ‘.tooltip’, ‘!default_values’, ‘wait_inputs’, ‘+function|solution’, ‘weight’, ‘remote_links

node_function

FolderNode.**node_function** = (‘-‘, ‘.tooltip’, ‘+input_domain|solution_domain’, ‘weight’, ‘+filters|solution_filters’, ‘

node_map

FolderNode.**node_map** = {‘’: (‘dot’, ‘table’), ‘*’: (‘link’,), ‘+’: (‘dot’, ‘table’), ‘!’: (‘dot’, ‘table’), ‘.’: (‘dot’,), ‘-‘: (), ‘?’

node_styles

FolderNode.**node_styles** = {‘info’: {start: {‘fillcolor’: ‘red’, ‘shape’: ‘egg’, ‘label’: ‘start’}, plot: {‘fillcolor’: ‘gold’,

re_node

FolderNode.**re_node** = regex.Regex(‘^([.*+!]?)(\\w+)(?>\\|(\\w+))?$’, flags=regex.V0)

title

FolderNode.**title**

type

FolderNode.**type**

counter = <method-wrapper ‘__next__’ of itertools.count object>

Site

class Site(*sitemap*, *host='localhost'*, *port=0*, *delay=0.1*, *until=30*, ***kwargs*)

Methods

__init__
app
get_port
run
shutdown_site
wait_server

__init__

Site.**__init__**(*sitemap*, *host='localhost'*, *port=0*, *delay=0.1*, *until=30*, ***kwargs*)

app

Site.**app**()

get_port

Site.**get_port**(*host=None*, *port=None*, ***kw*)

run

Site.**run**(***options*)

shutdown_site

static Site.**shutdown_site**(*url*)

wait_server

Site.**wait_server**(*elapsed=0*)

__init__(*sitemap*, *host='localhost'*, *port=0*, *delay=0.1*, *until=30*, ***kwargs*)

Attributes

url

url

Site.**url**

SiteFolder

class SiteFolder(*item*, *dsp*, *graph*, *obj*, *name=''*, *workflow=False*, *digraph=None*, ***options*)

Methods

__init__
dot
view

__init__

SiteFolder.**__init__**(*item*, *dsp*, *graph*, *obj*, *name=''*, *workflow=False*, *digraph=None*, ***options*)

dot

SiteFolder.**dot**(*context=None*)

view

SiteFolder.**view**(*filepath*, *context=None*, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*)

__init__(*item*, *dsp*, *graph*, *obj*, *name=''*, *workflow=False*, *digraph=None*, ***options*)

Attributes

counter
digraph
ext
filename
inputs
label_name
name
outputs
title
view_id

counter

SiteFolder.**counter** = <method-wrapper '__next__' of itertools.count object>

digraph

SiteFolder.**digraph** = {'graph_attr': {}, 'edge_attr': {}, 'format': 'svg', 'node_attr': {'style': 'filled'}, 'body': {'splin

ext

SiteFolder.**ext** = 'html'

filename

SiteFolder.**filename**

inputs

SiteFolder.**inputs**

label_name

SiteFolder.**label_name**

name

SiteFolder.**name**

outputs

SiteFolder.**outputs**

title

SiteFolder.**title**

view_id

SiteFolder.**view_id**

counter = <method-wrapper '__next__' of itertools.count object>

SiteIndex

class SiteIndex(*sitemap*, *node_id='index'*)

Methods

__init__
render
view

__init__

SiteIndex.**__init__**(*sitemap*, *node_id='index'*)

render

SiteIndex.**render**(*context*, **args*, ***kwargs*)

view

SiteIndex.**view**(*filepath*, **args*, ***, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*, ***kwargs*)

__init__(*sitemap*, *node_id='index'*)

Attributes

counter
ext
filename
name
title
view_id

counter

SiteIndex.**counter** = <method-wrapper '__next__' of itertools.count object>

ext

SiteIndex.**ext** = 'html'

filename

SiteIndex.**filename**

name

SiteIndex.**name**

title

SiteIndex.**title**

view_id

SiteIndex.**view_id**

SiteMap

class SiteMap

Methods

__init__	
add_items	
app	
clear	
copy	
fromkeys	If not specified, the value defaults to None.
get	
get_dsp_from	
get_sol_from	
index_filenames	
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last==False).
pop	value. If key is not found, d is returned if given, otherwise KeyError
popitem	Pairs are returned in LIFO order if last is true or FIFO order if false.
render	
rules	
setdefault	
site	
update	
values	

__init__

SiteMap.**__init__**()

add_items

SiteMap.**add_items**(*item*, *workflow=False*, *depth=-1*, ***options*)

app

SiteMap.**app**(*root_path=None*, *depth=-1*, *index=True*, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*, ***kw*)

clear

SiteMap.**clear**() → None. Remove all items from od.

copy

SiteMap.**copy**() → a shallow copy of od

fromkeys

SiteMap.**fromkeys**(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get

SiteMap.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_dsp_from

static SiteMap.**get_dsp_from**(*item*)

get_sol_from

static SiteMap.**get_sol_from**(*item*)

index_filenames

SiteMap.**index_filenames**()

items

SiteMap.**items**()

keys

SiteMap.**keys**()

move_to_end

SiteMap.**move_to_end**()
Move an existing element to the end (or beginning if last==False).

Raises KeyError if the element does not exist. When last=True, acts like a fast version of self[key]=self.pop(key).

pop

SiteMap.**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem

SiteMap.**popitem**() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

render

SiteMap.**render**(*depth=-1*, *directory='static'*, *view=False*, *index=True*, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*)

rules

SiteMap.**rules**(*depth=-1*, *index=True*)

setdefault

SiteMap.**setdefault**(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

site

SiteMap.**site**(*root_path=None*, *depth=-1*, *index=True*, *view=False*, ***kw*)

update

SiteMap.**update**()

values

SiteMap.**values**()

__init__()

Attributes

include_folders_as_filenames
nodes
options

include_folders_as_filenames

SiteMap.**include_folders_as_filenames** = True

nodes

SiteMap.**nodes**

options

SiteMap.**options** = {'node_function', 'edge_data', 'max_lines', 'max_width', 'node_styles', 'digraph', 'node_data'}

SiteNode

class SiteNode (*folder*, *node_id*, *item*, *obj*)

Methods

__init__
render
view

__init__

SiteNode.**__init__** (*folder*, *node_id*, *item*, *obj*)

render

SiteNode.**render** (**args*, ***kwargs*)

view

SiteNode.**view** (*filepath*, **args*, ***, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*, ***kwargs*)

__init__ (*folder*, *node_id*, *item*, *obj*)

Attributes

counter
ext
filename
name
title
view_id

counter

SiteNode.**counter** = <method-wrapper '__next__' of itertools.count object>

ext

SiteNode.**ext** = 'html'

filename

SiteNode.**filename**

name

SiteNode.**name**

title

SiteNode.**title**

view_id

SiteNode.**view_id**

counter = <method-wrapper '__next__' of itertools.count object>

dsp

It provides tools to create models with the `Dispatcher()`.

Functions

add_function	Decorator to add a function to a dispatcher.
are_in_nested_dicts	Nested keys are inside of nested-dictionaries.
bypass	Returns the same arguments.
combine_dicts	Combines multiple dicts in one.
combine_nested_dicts	Merge nested-dictionaries.
get_nested_dicts	Get/Initialize the value of nested-dictionaries.
kk_dict	Merges and defines dictionaries with values identical to keys.
map_dict	Returns a dict with new key values.
map_list	Returns a new dict.
parent_func	
replicate_value	Replicates *n* times the input value.

Continued on next page

Table 2.26 – continued from previous page

selector	Selects the chosen dictionary keys from the given dictionary.
stack_nested_keys	Stacks the keys of nested-dictionaries into tuples and yields a list of k-v pairs.
stlp	Converts a string in a tuple.
summation	Sums inputs values.

add_function

add_function(*dsp*, *inputs_kwargs=False*, *inputs_defaults=False*, ***kw*)
Decorator to add a function to a dispatcher.

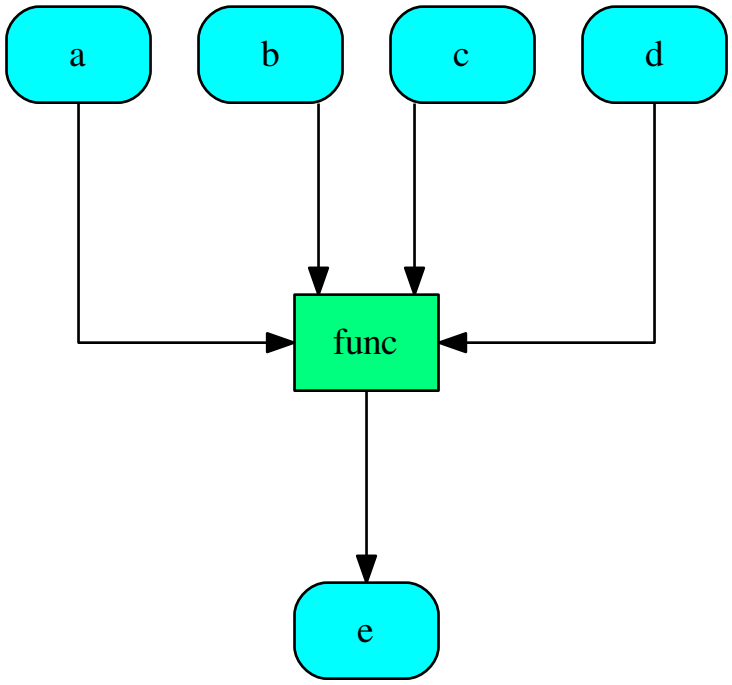
Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher.
- **inputs_kwargs** (*bool*) – Do you want to include kwargs as inputs?
- **inputs_defaults** (*bool*) – Do you want to set default values?
- **kw** (*dict*) – See :func:~‘schedula.Dispatcher.add_function‘.

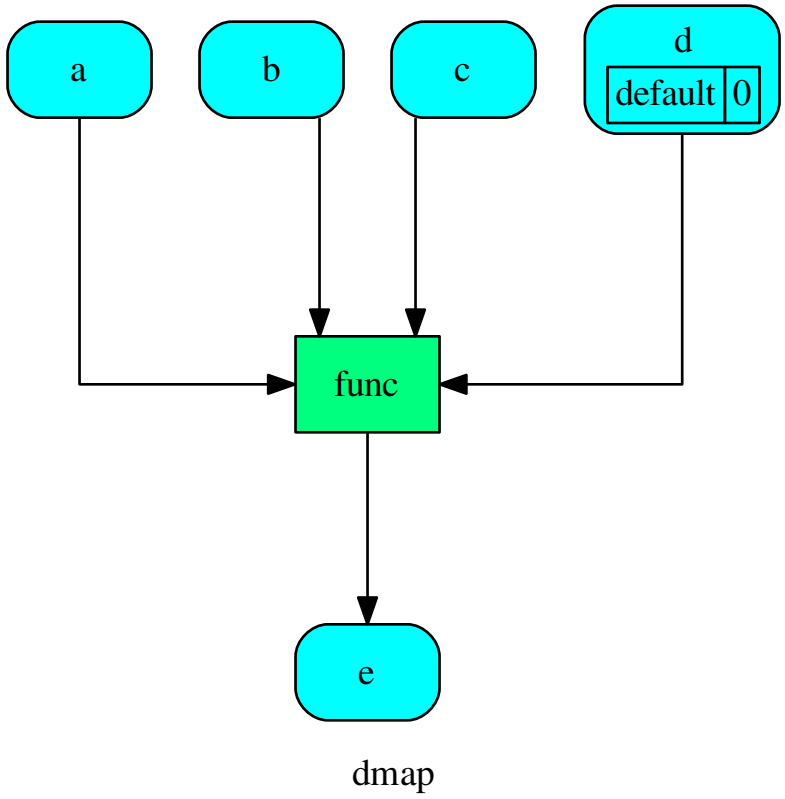
Returns Decorator.

Return type callable

Example:



dmap



dmap

are_in_nested_dicts

are_in_nested_dicts(*nested_dict*, **keys*)
Nested keys are inside of nested-dictionaries.

Parameters

- **nested_dict** (*dict*) – Nested dictionary.
- **keys** (*object*) – Nested keys.

Returns True if nested keys are inside of nested-dictionaries, otherwise False.

Return type bool

bypass

bypass(**inputs*, ***, *copy=False*)
Returns the same arguments.

Parameters

- **inputs** (*T*) – Inputs values.
- **copy** (*bool, optional*) – If True, it returns a deepcopy of input values.

Returns Same input values.

Return type (T, ...), T

Example:

```
>>> bypass('a', 'b', 'c')
('a', 'b', 'c')
>>> bypass('a')
'a'
```

combine_dicts

combine_dicts(**dicts*, ***, *copy=False*, *base=None*)

Combines multiple dicts in one.

Parameters

- **dicts** (*dict*) – A sequence of dicts.
- **copy** (*bool, optional*) – If True, it returns a deepcopy of input values.
- **base** (*dict, optional*) – Base dict where combine multiple dicts in one.

Returns A unique dict.

Return type dict

Example:

```
>>> sorted(combine_dicts({'a': 3, 'c': 3}, {'a': 1, 'b': 2}).items())
[('a', 1), ('b', 2), ('c', 3)]
```

combine_nested_dicts

combine_nested_dicts(**nested_dicts*, ***, *depth=-1*, *base=None*)

Merge nested-dictionaries.

Parameters

- **nested_dicts** (*dict*) – Nested dictionaries.
- **depth** (*int, optional*) – Maximum keys depth.
- **base** (*dict, optional*) – Base dict where combine multiple dicts in one.

Returns Combined nested-dictionary.

Return type dict

get_nested_dicts

get_nested_dicts(*nested_dict*, **keys*, ***, *default=None*, *init_nesting=<class 'dict'>*)

Get/Initialize the value of nested-dictionaries.

Parameters

- **nested_dict** (*dict*) – Nested dictionary.
- **keys** (*object*) – Nested keys.
- **default** (*callable, optional*) – Function used to initialize a new value.
- **init_nesting** (*callable, optional*) – Function used to initialize a new intermediate nesting dict.

Returns Value of nested-dictionary.

Return type generator

kk_dict

kk_dict(**kk*, ***adict*)

Merges and defines dictionaries with values identical to keys.

Parameters

- **kk** (*object | dict, optional*) – A sequence of keys and/or dictionaries.
- **adict** (*dict, optional*) – A dictionary.

Returns Merged dictionary.

Return type dict

Example:

```
>>> sorted(kk_dict('a', 'b', 'c').items())
[('a', 'a'), ('b', 'b'), ('c', 'c')]

>>> sorted(kk_dict('a', 'b', **{'a-c': 'c'}).items())
[('a', 'a'), ('a-c', 'c'), ('b', 'b')]

>>> sorted(kk_dict('a', {'b': 'c'}, 'c').items())
[('a', 'a'), ('b', 'c'), ('c', 'c')]

>>> sorted(kk_dict('a', 'b', **{'b': 'c'}).items())
Traceback (most recent call last):
 ...
ValueError: keyword argument repeated
```

map_dict

map_dict(*key_map*, **dicts*, ***, *copy=False*, *base=None*)

Returns a dict with new key values.

Parameters

- **key_map** (*dict*) – A dictionary that maps the dict keys ({old key: new key}
- **dicts** (*dict*) – A sequence of dicts.
- **copy** (*bool, optional*) – If True, it returns a deepcopy of input values.
- **base** (*dict, optional*) – Base dict where combine multiple dicts in one.

Returns A unique dict with new key values.

Return type dict

Example:

```
>>> d = map_dict({'a': 'c', 'b': 'd'}, {'a': 1, 'b': 1}, {'b': 2})
>>> sorted(d.items())
[('c', 1), ('d', 2)]
```

map_list

map_list(*key_map*, **inputs*, ***, *copy=False*, *base=None*)

Returns a new dict.

Parameters

- **key_map** (*list[str | dict | list]*) – A list that maps the dict keys ({old key: new key}
- **inputs** (*iterable | dict | int | float | list | tuple*) – A sequence of data.
- **copy** (*bool, optional*) – If True, it returns a deepcopy of input values.
- **base** (*dict, optional*) – Base dict where combine multiple dicts in one.

Returns A unique dict with new values.

Return type dict

Example:

```
>>> key_map = [
...     'a',
...     {'a': 'c'},
...     [
...         'a',
...         {'a': 'd'}
...     ]
... ]
>>> inputs = (
...     2,
...     {'a': 3, 'b': 2},
...     [
...         1,
...         {'a': 4}
...     ]
... )
>>> d = map_list(key_map, *inputs)
>>> sorted(d.items())
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

parent_func

parent_func(*func*, *input_id=None*)

replicate_value

replicate_value(*value*, *n=2*, *copy=True*)

Replicates *n* times the input value.

Parameters

- **n** (*int*) – Number of replications.
- **value** (*T*) – Value to be replicated.
- **copy** (*bool*) – If True the list contains deep-copies of the value.

Returns A list with the value replicated *n* times.

Return type list

Example:

```
>>> from functools import partial
>>> fun = partial(replicate_value, n=5)
>>> fun({'a': 3})
({'a': 3}, {'a': 3}, {'a': 3}, {'a': 3}, {'a': 3})
```

selector

selector(*keys*, *dictionary*, *copy=False*, *output_type='dict'*, *allow_miss=False*)

Selects the chosen dictionary keys from the given dictionary.

Parameters

- **keys** (*list, tuple, set*) – Keys to select.
- **dictionary** (*dict*) – A dictionary.
- **copy** (*bool*) – If True the output contains deep-copies of the values.
- **output_type** – Type of function output:
  - 'list': a list with all values listed in *keys*.
  - 'dict': a dictionary with any outputs listed in *keys*.
  - **'values': if output length == 1 return a single value otherwise a** tuple with all values listed in *keys*.

    type output_type str, optional
- **allow_miss** (*bool*) – If True it does not raise when some key is missing in the dictionary.

Returns A dictionary with chosen dictionary keys if present in the sequence of dictionaries. These are combined with *combine_dicts()*.

Return type dict

Example:

```
>>> from functools import partial
>>> fun = partial(selector, ['a', 'b'])
>>> sorted(fun({'a': 1, 'b': 2, 'c': 3}).items())
[('a', 1), ('b', 2)]
```

stack_nested_keys

stack_nested_keys(*nested_dict*, *key=()*, *depth=-1*)

Stacks the keys of nested-dictionaries into tuples and yields a list of k-v pairs.

Parameters

- **nested_dict** (*dict*) – Nested dictionary.
- **key** (*tuple, optional*) – Initial keys.
- **depth** (*int, optional*) – Maximum keys depth.

Returns List of k-v pairs.

Return type generator

stlp

stlp(*s*)

Converts a string in a tuple.

summation

summation(**inputs*)

Sums inputs values.

Parameters **inputs** (*int, float*) – Inputs values.

Returns Sum of the input values.

Return type int, float

Example:

```
>>> summation(1, 3.0, 4, 2)
10.0
```

Classes

DFun	A 3-tuple `(out,fun,**kwds)`, used to prepare a list of calls to `Dispatcher.add_function()`.
DispatchPipe	It converts a `Dispatcher()` into a function.
NoSub	Class for avoiding to add a sub solution to the workflow.
SubDispatch	It dispatches a given `Dispatcher()` like a function.
SubDispatchFunction	It converts a `Dispatcher()` into a function.
SubDispatchPipe	It converts a `Dispatcher()` into a function.
add_args	Adds arguments to a function (left side).

DFun

class DFun(*out*, *fun*, *inputs=None*, ***kwds*)

A 3-tuple `(out,fun,**kwds)`, used to prepare a list of calls to

`Dispatcher.add_function()`.

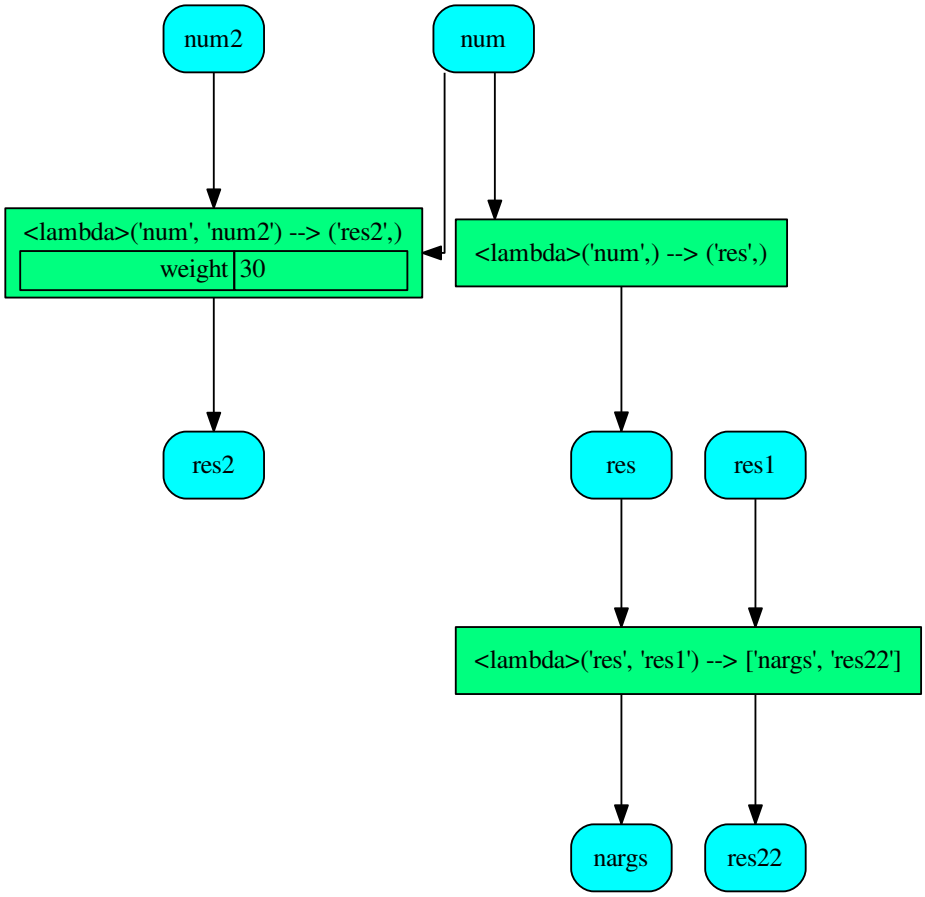
The workhorse is the `addme()` which delegates to `Dispatcher.add_function()`:

- `out`: a scalar string or a string-list that, sent as *output* arg,
- `fun`: a callable, sent as *function* args,
- `kwds`: any keywords of `Dispatcher.add_function()`.
- Specifically for the 'inputs' argument, if present in *kwds*, use them (a scalar-string or string-list type, possibly empty), else inspect function; in any case wrap the result in a tuple (if not already a list-type).

Note: Inspection works only for regular args, no `*args,**kwds` supported, and they will fail late, on `addme()`, if no *input* or *inp* defined.

Example:

```
>>> dfuns = [
...     DFun('res', lambda num: num * 2),
...     DFun('res2', lambda num, num2: num + num2, weight=30),
...     DFun(out=['nargs', 'res22'],
...          fun=lambda *args: (len(args), args),
...          inputs=('res', 'res1')
...     )]
>>> dfuns
[DFun('res', <function <lambda> at 0x...>, ),
 DFun('res2', <function <lambda> at 0x...>, weight=30),
 DFun(['nargs', 'res22'], <function <lambda> at 0x...>,
      inputs=('res', 'res1'))]
>>> from schedula import Dispatcher
>>> dsp = Dispatcher()
>>> DFun.add_dfuns(dfuns, dsp)
```



dmap

Methods

__init__
add_dfuns
addme
copy
inspect_inputs

__init__

DFun.**__init__**(*out*, *fun*, *inputs=None*, ***kwds*)

add_dfuns

classmethod `DFun.`**`add_dfuns`**(*dfuns*, *dsp*)

addme

`DFun.`**`addme`**(*dsp*)

copy

`DFun.`**`copy`**()

inspect_inputs

`DFun.`**`inspect_inputs`**()

`__init__`(*out*, *fun*, *inputs=None*, ***kwds*)

DispatchPipe

class `DispatchPipe`(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *no_domain=True*, *wildcard=True*)

It converts a `Dispatcher()` into a function.

This function takes a sequence of arguments as input of the dispatch.

Returns A function that executes the pipe of the given *dsp*, updating its workflow.

Return type callable

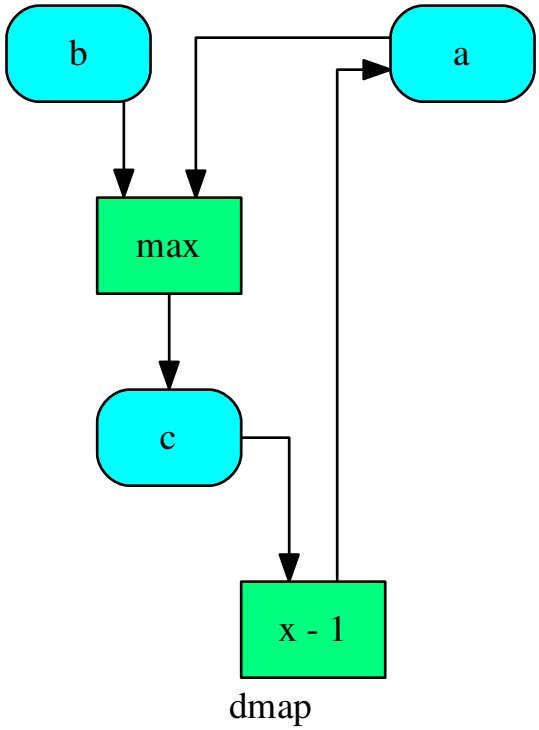
Note: This wrapper is not thread safe, because it overwrite the solution.

See also:

`dispatch()`, `shrink_dsp()`

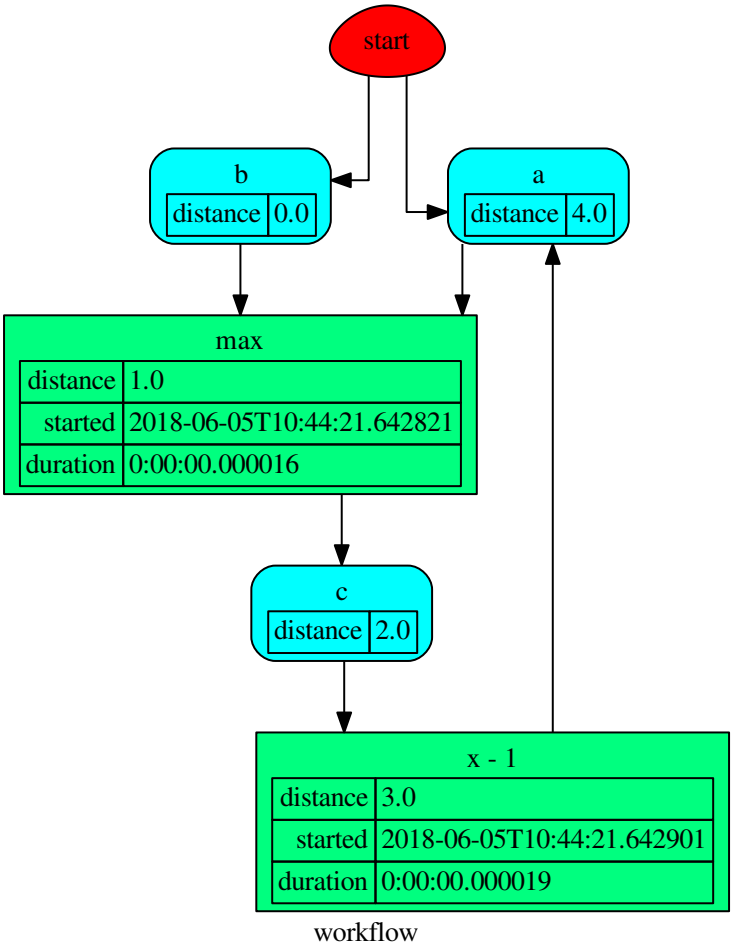
Example:

A dispatcher with two functions *max* and *min* and an unresolved cycle (i.e., *a* –> *max* –> *c* –> *min* –> *a*):



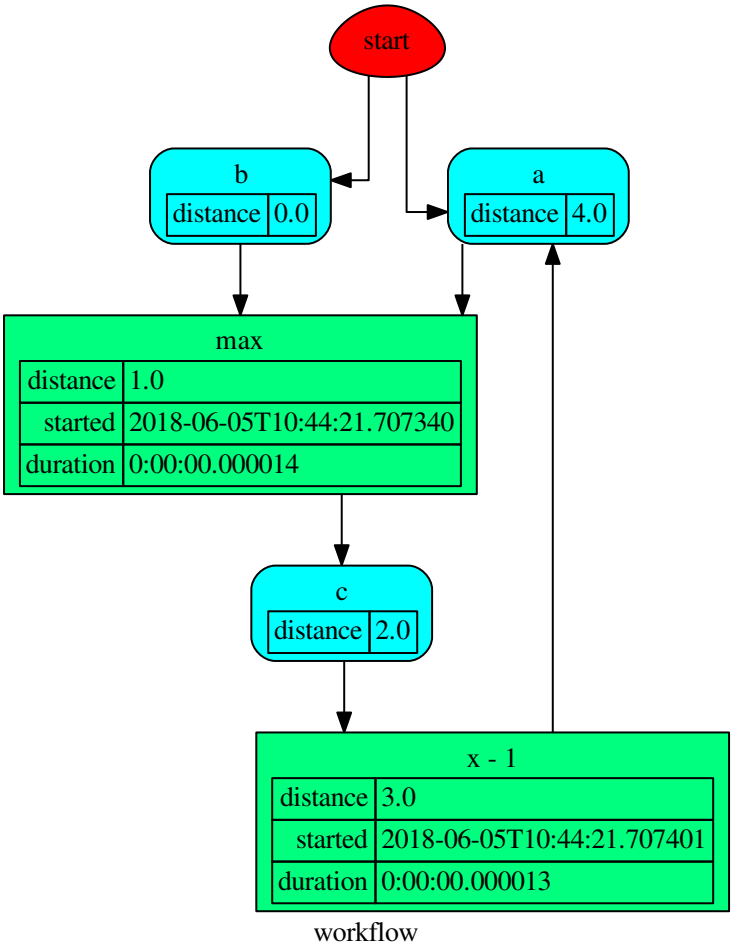
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```
>>> fun = DispatchPipe(dsp, 'myF', ['a', 'b'], ['a'])
>>> fun.__name__
'myF'
>>> fun(2, 1)
1
```



The created function raises a ValueError if un-valid inputs are provided:

```
>>> fun(1, 0)
0
```



Methods

__init__	Initializes the Sub-dispatch Function.
copy	
get_node	Returns a sub node of a dispatcher.
parse_inputs	
plot	Plots the Dispatcher with a graph in the DOT language with Graphviz.
search_node_description	
web	Creates a dispatcher Flask app.

__init__

DispatchPipe.**__init__**(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *no_domain=True*, *wildcard=True*)

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **function_id** (*str*) – Function name.
- **inputs** (*list[str], iterable*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.

copy

DispatchPipe.**copy**()

get_node

DispatchPipe.**get_node**(**node_ids*, ***, *node_attr=none*)

Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  – for data node: its output, and if not exists, all its attributes.

  – for function and sub-dispatcher nodes: the 'function' attribute.

  When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

  When 'output', returns the data node output.

  When 'default_value', returns the data node default value.

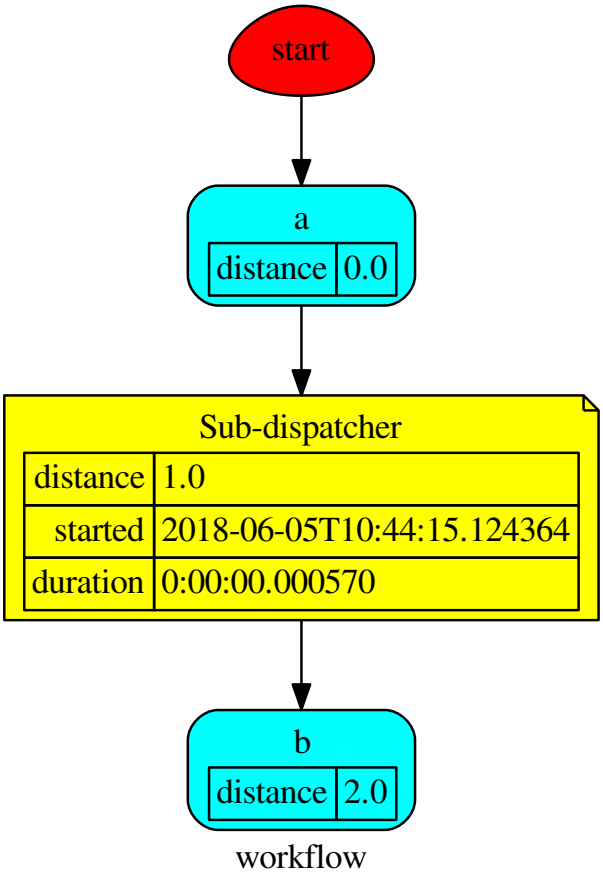
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

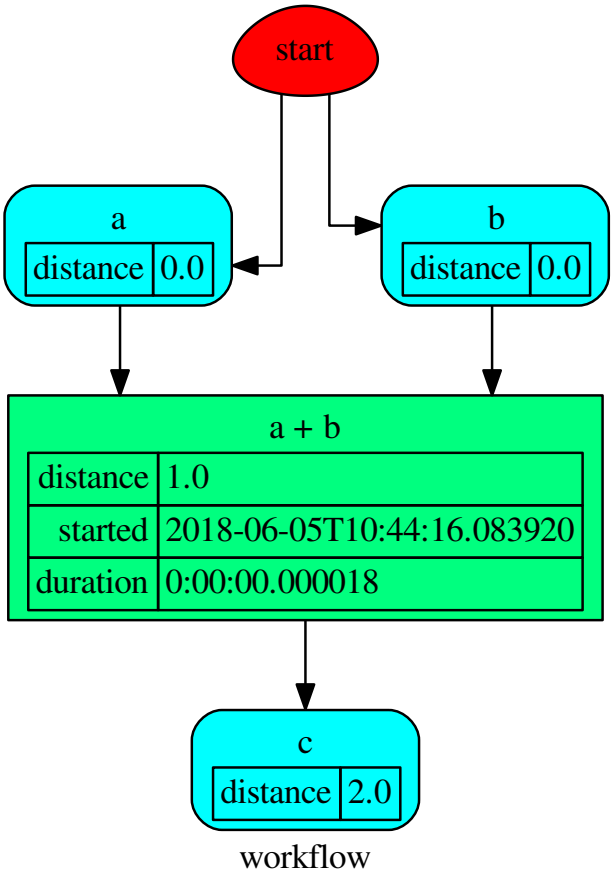
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



parse_inputs

DispatchPipe.**parse_inputs**(*valid_keyword*, **args*, ***kwargs*)

plot

DispatchPipe.**plot**(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)

Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **workflow** (*bool, optional*) – If True the latest solution will be plotted, otherwise the dmap.

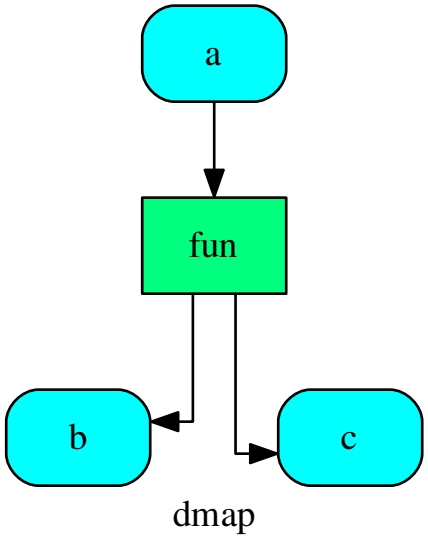
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



search_node_description

DispatchPipe.**search_node_description**(*node_id*, *what='description'*)

web

DispatchPipe.**web**(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

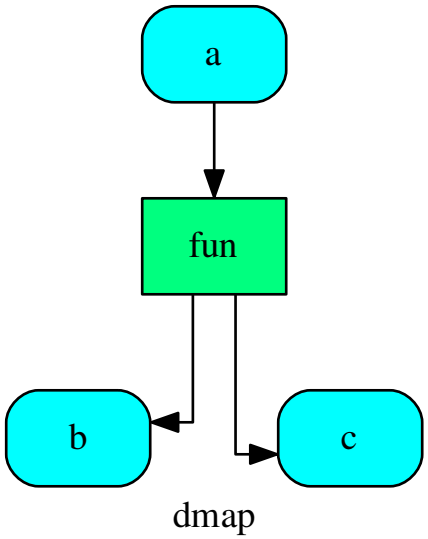
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

__init__(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *no_domain=True*, *wildcard=True*)

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **function_id** (*str*) – Function name.

- **inputs** (*list[str], iterable*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.

NoSub

class NoSub

Class for avoiding to add a sub solution to the workflow.

__init__()

Initialize self. See help(type(self)) for accurate signature.

SubDispatch

class SubDispatch(*dsp*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *wildcard=False*, *no_call=False*, *shrink=False*, *rm_unused_nds=False*, *output_type='all'*)

It dispatches a given `Dispatcher()` like a function.

This function takes a sequence of dictionaries as input that will be combined before the dispatching.

Returns A function that executes the dispatch of the given `Dispatcher()`.

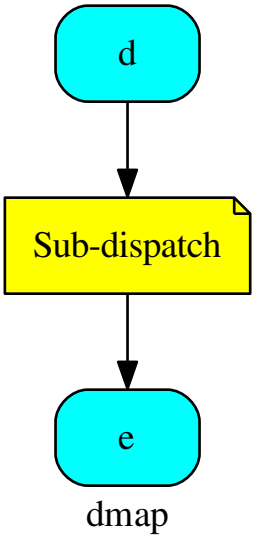
Return type callable

See also:

`dispatch()`, *combine_dicts()*

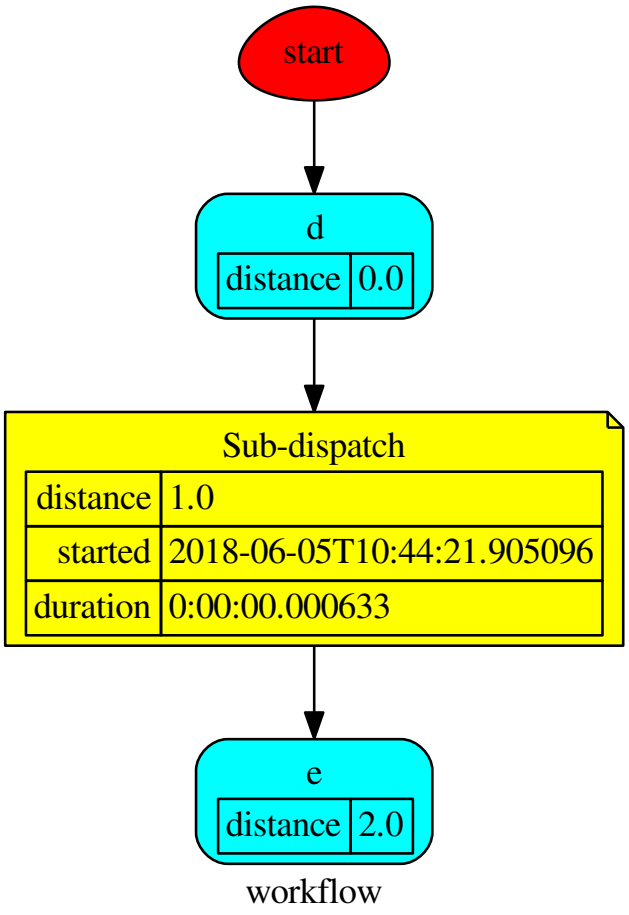
Example:

```
>>> from schedula import Dispatcher
>>> sub_dsp = Dispatcher(name='Sub-dispatcher')
...
>>> def fun(a):
...     return a + 1, a - 1
...
>>> sub_dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dispatch = SubDispatch(sub_dsp, ['a', 'b', 'c'], output_type='dict')
>>> dsp = Dispatcher(name='Dispatcher')
>>> dsp.add_function('Sub-dispatch', dispatch, ['d'], ['e'])
'Sub-dispatch'
```



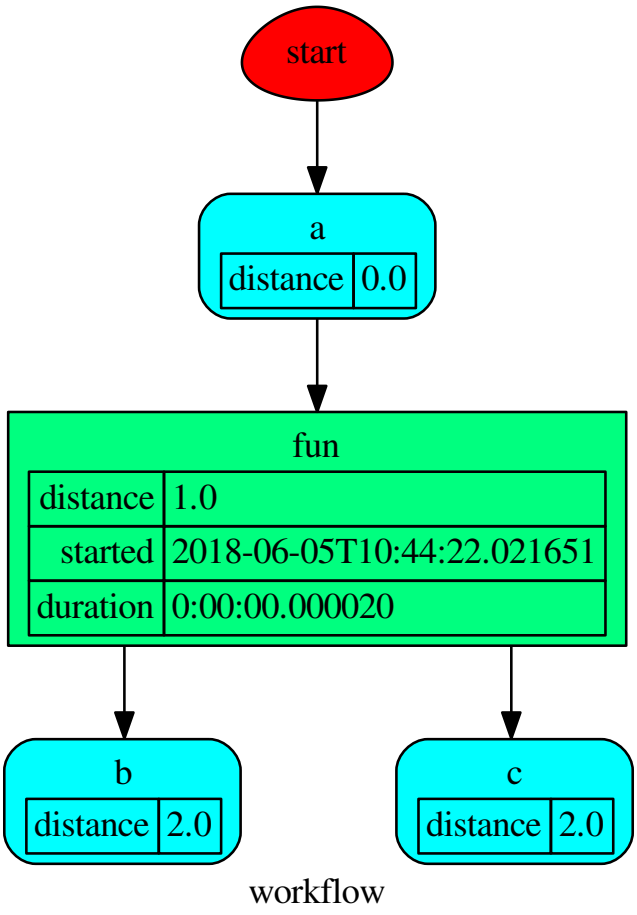
The Dispatcher output is:

```
>>> o = dsp.dispatch(inputs={'d': {'a': 3}})
```



while, the Sub-dispatch is:

```
>>> sol = o.workflow.node['Sub-dispatch']['solution']
>>> sol
Solution([('a', 3), ('b', 4), ('c', 2)])
>>> sol == o['e']
True
```



Methods

`__init__`	Initializes the Sub-dispatch.
`copy`	
`get_node`	Returns a sub node of a dispatcher.
`plot`	Plots the Dispatcher with a graph in the DOT language with Graphviz.
`search_node_description`	
`web`	Creates a dispatcher Flask app.

__init__

`SubDispatch.__init__`(*dsp*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *wildcard=False*, *no_call=False*, *shrink=False*, *rm_unused_nds=False*, *output_type='all'*)

Initializes the Sub-dispatch.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **outputs** (*list[str], iterable*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.
- **no_call** (*bool, optional*) – If True data node estimation function is not used.
- **shrink** (*bool, optional*) – If True the dispatcher is shrink before the dispatch.
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **output_type** (*str, optional*) – Type of function output:
  - 'all': a dictionary with all dispatch outputs.
  - 'list': a list with all outputs listed in *outputs*.
  - 'dict': a dictionary with any outputs listed in *outputs*.

copy

SubDispatch.**copy**()

get_node

SubDispatch.**get_node**(**node_ids*, *, *node_attr=none*)

Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  - for data node: its output, and if not exists, all its attributes.
  - for function and sub-dispatcher nodes: the 'function' attribute.

  When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

  When 'output', returns the data node output.

  When 'default_value', returns the data node default value.

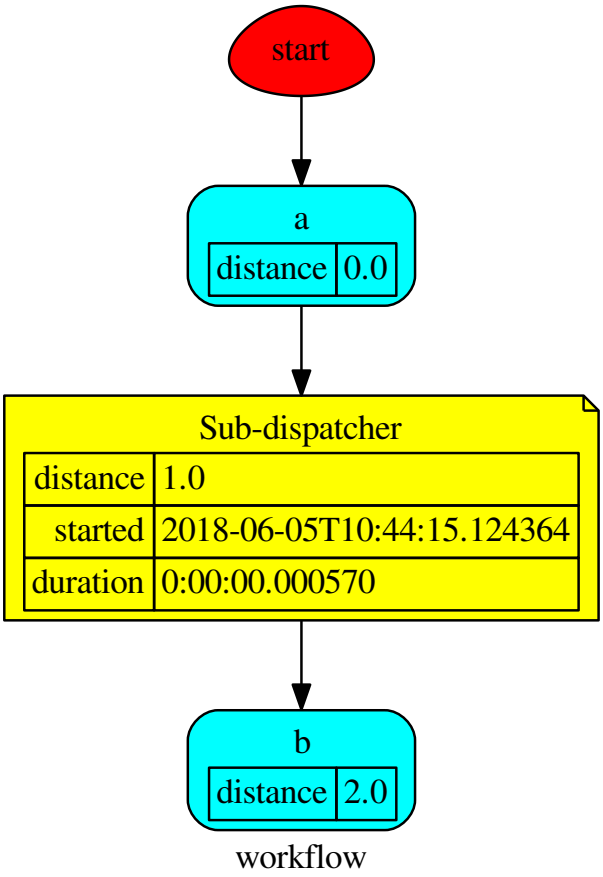
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

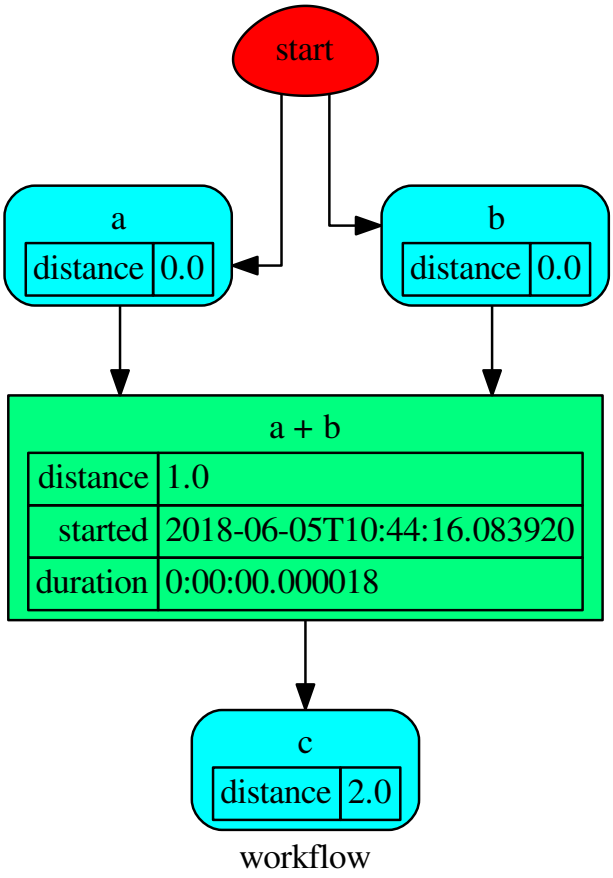
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



plot

SubDispatch.**plot**(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)

Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **workflow** (*bool, optional*) – If True the latest solution will be plotted, otherwise the dmap.
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.

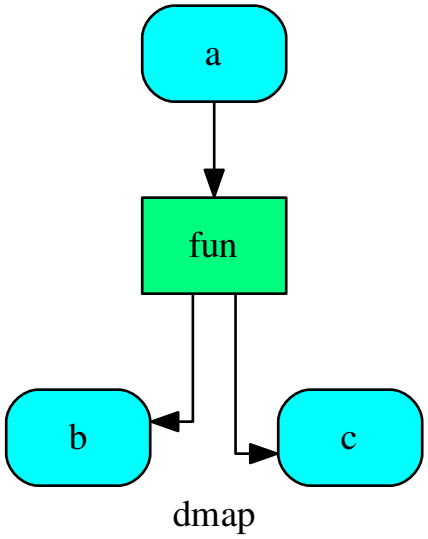
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



search_node_description

SubDispatch.**search_node_description**(*node_id*, *what='description'*)

web

SubDispatch.**web**(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

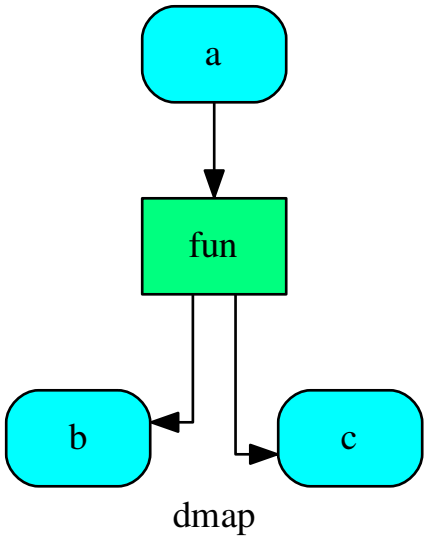
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

__init__(*dsp*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *wildcard=False*, *no_call=False*, *shrink=False*, *rm_unused_nds=False*, *output_type='all'*)
Initializes the Sub-dispatch.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **outputs** (*list[str], iterable*) – Ending data nodes.

- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.
- **wildcard** (*bool, optional*) – If True, when the data node is used as input and target in the ArciDispatch algorithm, the input value will be used as input for the connected functions, but not as output.
- **no_call** (*bool, optional*) – If True data node estimation function is not used.
- **shrink** (*bool, optional*) – If True the dispatcher is shrink before the dispatch.
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **output_type** (*str, optional*) – Type of function output:
  - 'all': a dictionary with all dispatch outputs.
  - 'list': a list with all outputs listed in *outputs*.
  - 'dict': a dictionary with any outputs listed in *outputs*.

SubDispatchFunction

class **SubDispatchFunction**(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *shrink=True*, *wildcard=True*)

It converts a `Dispatcher()` into a function.

This function takes a sequence of arguments or a key values as input of the dispatch.

Returns A function that executes the dispatch of the given *dsp*.

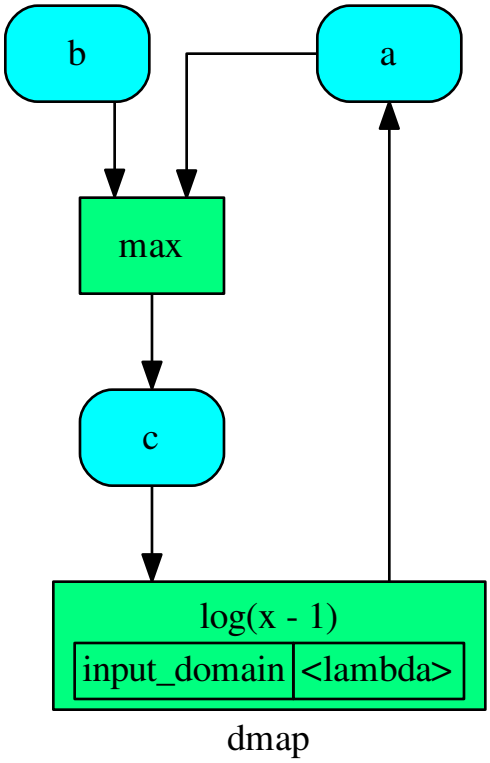
Return type callable

See also:

`dispatch()`, `shrink_dsp()`

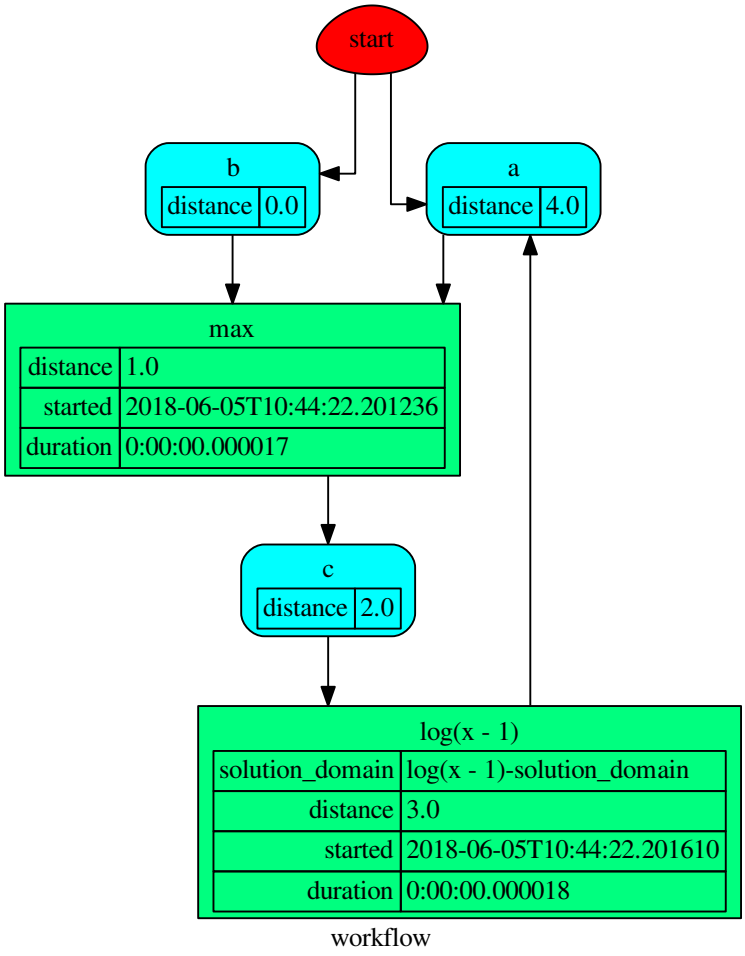
Example:

A dispatcher with two functions *max* and *min* and an unresolved cycle (i.e., *a* –> *max* –> *c* –> *min* –> *a*):



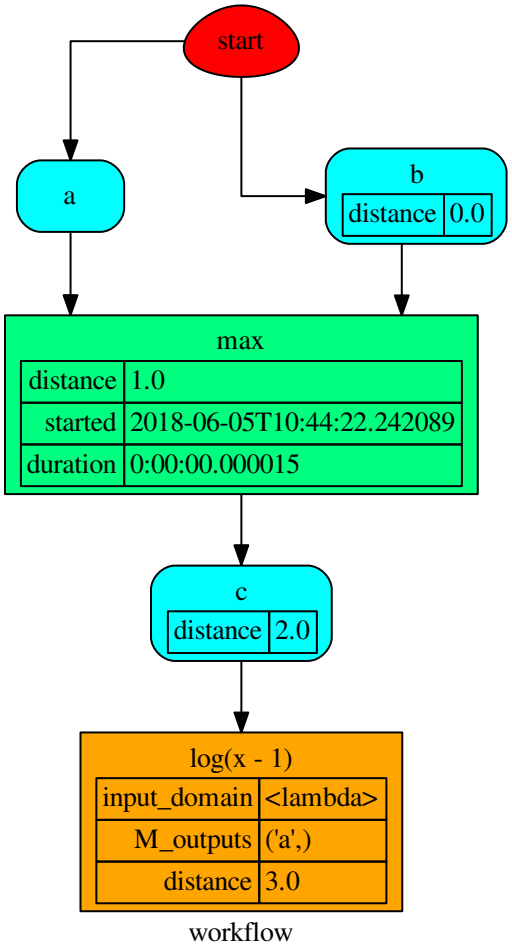
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```
>>> fun = SubDispatchFunction(dsp, 'myF', ['a', 'b'], ['a'])
>>> fun.__name__
'myF'
>>> fun(2, 1)
0.0
```



The created function raises a ValueError if un-valid inputs are provided:

```
>>> fun(1, 0)
Traceback (most recent call last):
...
DispatcherError:
  Unreachable output-targets: ...
  Available outputs: ...
```



Methods

__init__	Initializes the Sub-dispatch Function.
copy	
get_node	Returns a sub node of a dispatcher.
parse_inputs	
plot	Plots the Dispatcher with a graph in the DOT language with Graphviz.
search_node_description	
web	Creates a dispatcher Flask app.

__init__

SubDispatchFunction.**__init__**(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *shrink=True*, *wildcard=True*)

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **function_id** (*str*) – Function name.
- **inputs** (*list[str], iterable*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.

copy

SubDispatchFunction.**copy**()

get_node

SubDispatchFunction.**get_node**(**node_ids*, ***, *node_attr=none*)

Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  – for data node: its output, and if not exists, all its attributes.

  – for function and sub-dispatcher nodes: the 'function' attribute.

  When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

  When 'output', returns the data node output.

  When 'default_value', returns the data node default value.

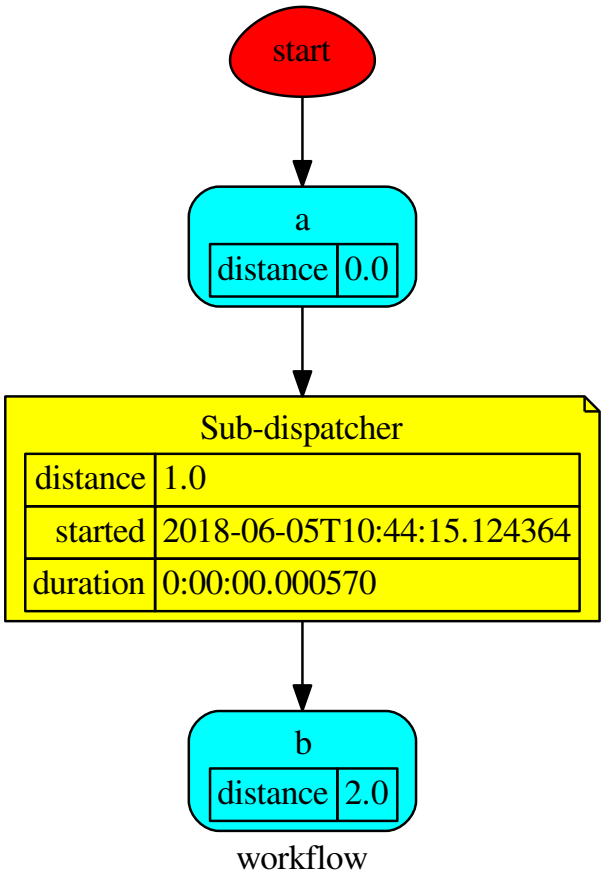
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

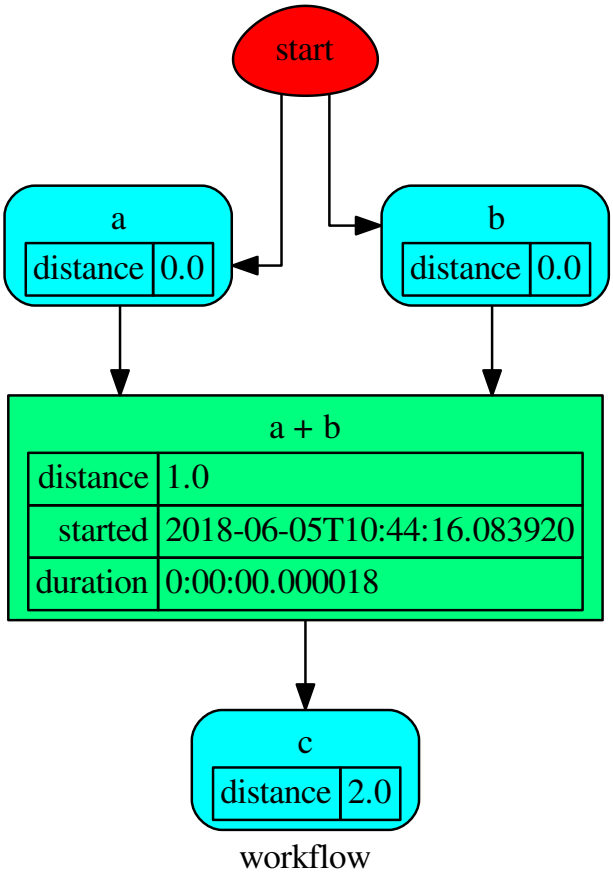
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



parse_inputs

`SubDispatchFunction.`**`parse_inputs`**(*valid_keyword*, **args*, ***kwargs*)

plot

`SubDispatchFunction.`**`plot`**(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)

Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **`workflow`** (*`bool, optional`*) – If True the latest solution will be plotted, otherwise the dmap.

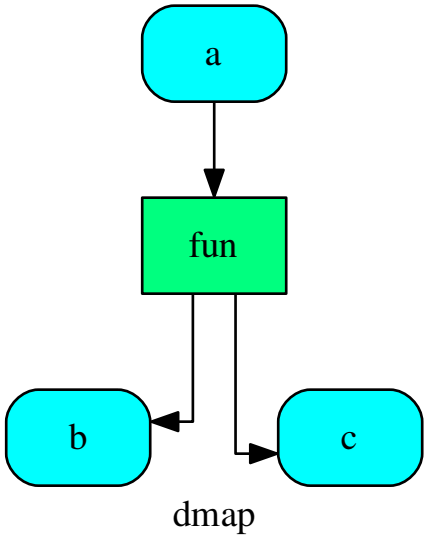
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



search_node_description

SubDispatchFunction.**search_node_description**(*node_id*, *what='description'*)

web

SubDispatchFunction.**web**(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

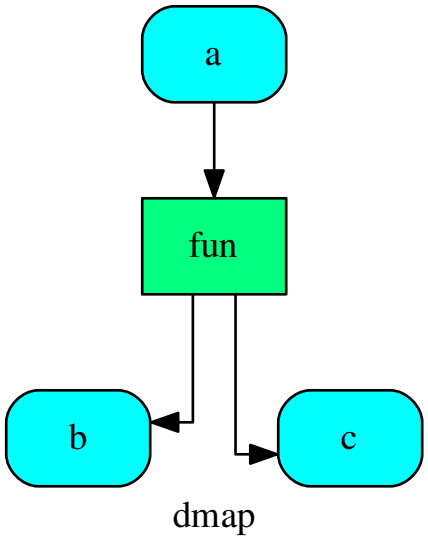
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

__init__(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *shrink=True*, *wildcard=True*)
Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **function_id** (*str*) – Function name.

- **inputs** (*list[str], iterable*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.

SubDispatchPipe

class SubDispatchPipe(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *no_domain=True*, *wildcard=True*)

It converts a `Dispatcher()` into a function.

This function takes a sequence of arguments as input of the dispatch.

Returns A function that executes the pipe of the given *dsp*.

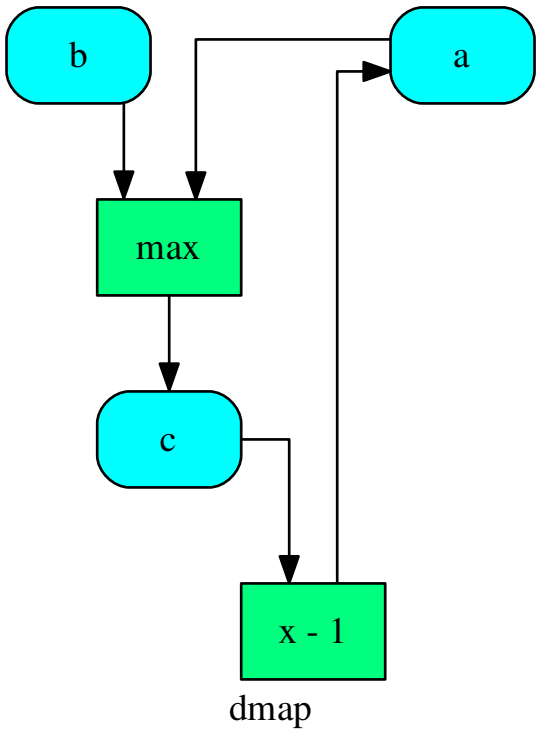
Return type callable

See also:

`dispatch()`, `shrink_dsp()`

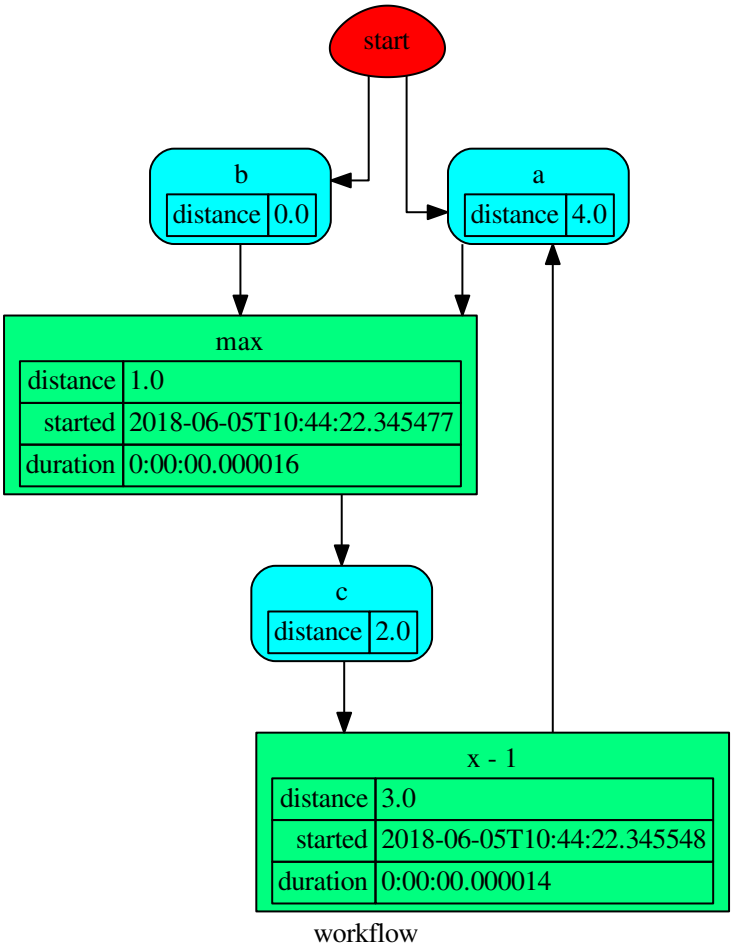
Example:

A dispatcher with two functions *max* and *min* and an unresolved cycle (i.e., *a* –> *max* –> *c* –> *min* –> *a*):



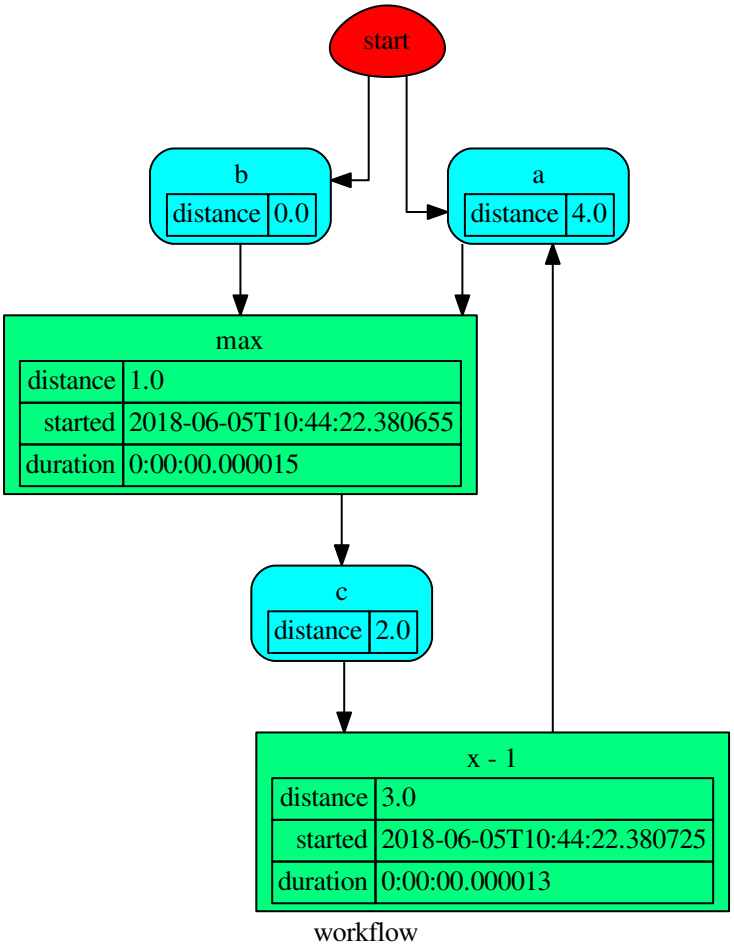
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```
>>> fun = SubDispatchPipe(dsp, 'myF', ['a', 'b'], ['a'])
>>> fun.__name__
'myF'
>>> fun(2, 1)
1
```



The created function raises a ValueError if un-valid inputs are provided:

```
>>> fun(1, 0)
0
```



Methods

__init__	Initializes the Sub-dispatch Function.
copy	
get_node	Returns a sub node of a dispatcher.
parse_inputs	
plot	Plots the Dispatcher with a graph in the DOT language with Graphviz.
search_node_description	
web	Creates a dispatcher Flask app.

__init__

SubDispatchPipe.**__init__**(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *no_domain=True*, *wildcard=True*)

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **function_id** (*str*) – Function name.
- **inputs** (*list[str], iterable*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.

copy

SubDispatchPipe.**copy**()

get_node

SubDispatchPipe.**get_node**(**node_ids*, ***, *node_attr=none*)

Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  – for data node: its output, and if not exists, all its attributes.

  – for function and sub-dispatcher nodes: the 'function' attribute.

  When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

  When 'output', returns the data node output.

  When 'default_value', returns the data node default value.

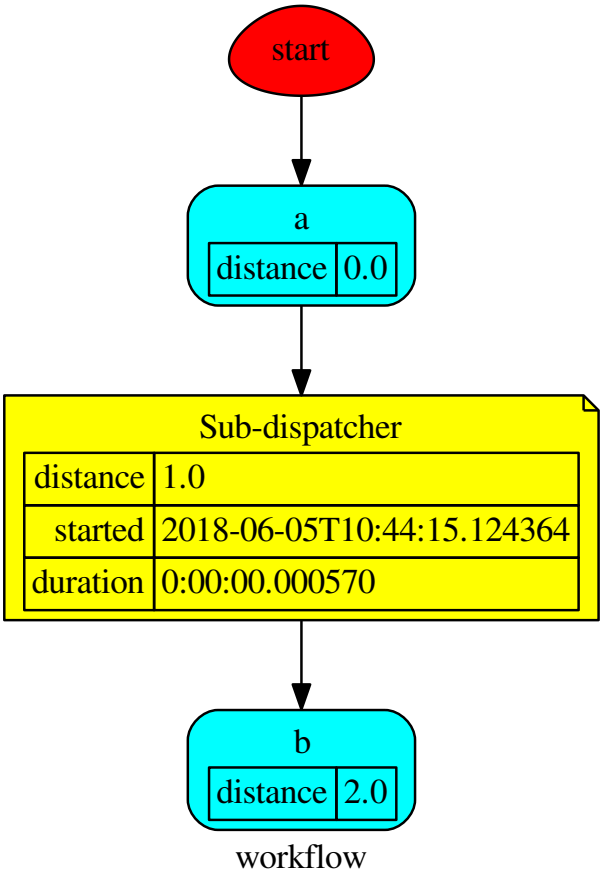
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

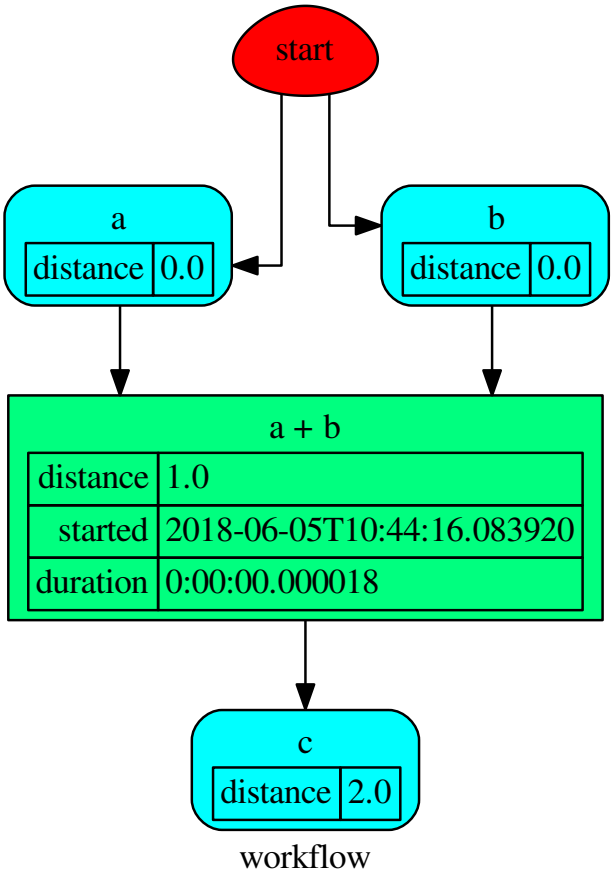
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



parse_inputs

SubDispatchPipe.**parse_inputs**(*valid_keyword*, **args*, ***kwargs*)

plot

SubDispatchPipe.**plot**(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)

Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **workflow** (*bool, optional*) – If True the latest solution will be plotted, otherwise the dmap.

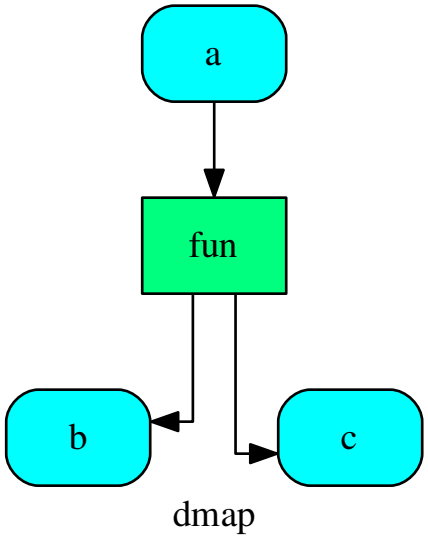
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



search_node_description

SubDispatchPipe.**search_node_description**(*node_id*, *what='description'*)

web

SubDispatchPipe.**web**(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

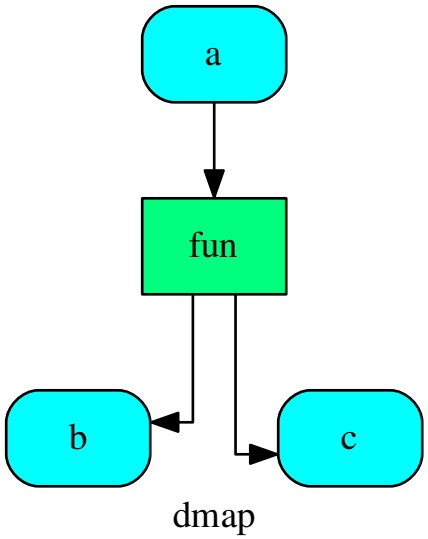
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

__init__(*dsp*, *function_id*, *inputs*, *outputs=None*, *cutoff=None*, *inputs_dist=None*, *no_domain=True*, *wildcard=True*)

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **function_id** (*str*) – Function name.

- **inputs** (*list[str], iterable*) – Input data nodes.
- **outputs** (*list[str], iterable, optional*) – Ending data nodes.
- **cutoff** (*float, int, optional*) – Depth to stop the search.
- **inputs_dist** (*dict[str, int | float], optional*) – Initial distances of input data nodes.

add_args

class add_args (*func*, *n=1*, *callback=None*)

Adds arguments to a function (left side).

Parameters

- **func** (*callable*) – Function to wrap.
- **n** (*int*) – Number of unused arguments to add to the left side.

Returns Wrapped function.

Return type callable

Example:

```
>>> def original_func(a, b):
...     '''Doc'''
...     return a + b
>>> func = add_args(original_func, n=2)
>>> func.__name__, func.__doc__
('original_func', 'Doc')
>>> func(1, 2, 3, 4)
7
```

Methods

__init__

__init__

add_args.**__init__** (*func*, *n=1*, *callback=None*)

__init__ (*func*, *n=1*, *callback=None*)

exc

Defines the dispatcher exception.

Exceptions

DispatcherAbort
DispatcherError

DispatcherAbort

exception **DispatcherAbort**(**args*, ***, *sol=None*, ***kwargs*)

DispatcherError

exception **DispatcherError**(**args*, ***, *sol=None*, ***kwargs*)

gen

It contains classes and functions of general utility.

These are python-specific utilities and hacks - general data-processing or numerical operations.

Functions

counter	Return a object whose .__call__() method returns consecutive values.
pairwise	A sequence of overlapping sub-sequences.

counter

counter(*start=0*, *step=1*)

Return a object whose .__call__() method returns consecutive values.

Parameters

- **start** (*int, float, optional*) – Start value.
- **step** (*int, float, optional*) – Step value.

pairwise

pairwise(*iterable*)

A sequence of overlapping sub-sequences.

Parameters **iterable** (*iterable*) – An iterable object.

Returns A zip object.

Return type zip

Example:

```
>>> list(pairwise([1, 2, 3, 4, 5]))
[(1, 2), (2, 3), (3, 4), (4, 5)]
```

Classes

Token	It constructs a unique constant that behaves like a string.

Token

class Token

It constructs a unique constant that behaves like a string.

Example:

```
>>> s = Token('string')
>>> s
string
>>> s == 'string'
False
>>> s == Token('string')
False
>>> {s: 1, Token('string'): 1}
{string: 1, string: 1}
>>> s.capitalize()
'String'
```

Methods

capitalize	Return a capitalized version of S, i.e.
casefold	Return a version of S suitable for caseless comparisons.
center	Return S centered in a string of length width.
count	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
encode	Encode S using the codec registered for encoding.
endswith	Return True if S ends with the specified suffix, False otherwise.
expandtabs	Return a copy of S where all tab characters are expanded using spaces.
find	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
format	Return a formatted version of S, using substitutions from args and kwargs.
format_map	Return a formatted version of S, using substitutions from mapping.
index	Like S.find() but raise ValueError when the substring is not found.
isalnum	Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.
isalpha	Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.
isdecimal	Return True if there are only decimal characters in S, False otherwise.
isdigit	Return True if all characters in S are digits and there is at least one character in S, False otherwise.
isidentifier	Return True if S is a valid identifier according to the language definition.
islower	Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

Continued on next page

Table 2.37 – continued from previous page

isnumeric	Return True if there are only numeric characters in S, False otherwise.
isprintable	Return True if all characters in S are considered printable in repr() or S is empty, False otherwise.
isspace	Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.
istitle	Return True if S is a titlecased string and there is at least one character in S, i.e.
isupper	Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.
join	Return a string which is the concatenation of the strings in the iterable.
ljust	Return S left-justified in a Unicode string of length width.
lower	Return a copy of the string S converted to lowercase.
lstrip	Return a copy of the string S with leading whitespace removed.
maketrans	Return a translation table usable for str.translate().
partition	Search for the separator sep in S, and return the part before it, the separator itself, and the part after it.
replace	Return a copy of S with all occurrences of substring old replaced by new.
rfind	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
rindex	Like S.rfind() but raise ValueError when the substring is not found.
rjust	Return S right-justified in a string of length width.
rpartition	Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it.
rsplit	Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front.
rstrip	Return a copy of the string S with trailing whitespace removed.
split	Return a list of the words in S, using sep as the delimiter string.
splitlines	Return a list of the lines in S, breaking at line boundaries.
startswith	Return True if S starts with the specified prefix, False otherwise.
strip	Return a copy of the string S with leading and trailing whitespace removed.
swapcase	Return a copy of S with uppercase characters converted to lowercase and vice versa.
title	Return a titlecased version of S, i.e.
translate	Return a copy of the string S in which each character has been mapped through the given translation table.
upper	Return a copy of S converted to uppercase.

Continued on next page

Table 2.37 – continued from previous page

zfill	Pad a numeric string S with zeros on the left, to fill a field of the specified width.

capitalize

Token.**capitalize**() → str
Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

casefold

Token.**casefold**() → str
Return a version of S suitable for caseless comparisons.

center

Token.**center**(*width*[, *fillchar*]) → str
Return S centered in a string of length width. Padding is done using the specified fill character (default is a space)

count

Token.**count**(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode

Token.**encode**(*encoding='utf-8'*, *errors='strict'*) → bytes
Encode S using the codec registered for encoding. Default encoding is 'utf-8'. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith

Token.**endswith**(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs

Token.**expandtabs**(*tabsize=8*) → str
Return a copy of S where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.

find

Token.**find**(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format

Token.**format**(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map

Token.**format_map**(*mapping*) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index

Token.**index**(*sub*[, *start*[, *end*]]) → int
Like S.find() but raise ValueError when the substring is not found.

isalnum

Token.**isalnum**() → bool
Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

isalpha

Token.**isalpha**() → bool
Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

isdecimal

Token.**isdecimal**() → bool
Return True if there are only decimal characters in S, False otherwise.

isdigit

Token.**isdigit**() → bool
Return True if all characters in S are digits and there is at least one character in S, False otherwise.

isidentifier

Token.**isidentifier**() → bool
Return True if S is a valid identifier according to the language definition.

Use keyword.iskeyword() to test for reserved identifiers such as “def” and “class”.

islower

Token.**islower**() → bool
Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

isnumeric

Token.**isnumeric**() → bool
Return True if there are only numeric characters in S, False otherwise.

isprintable

Token.**isprintable**() → bool
Return True if all characters in S are considered printable in repr() or S is empty, False otherwise.

isspace

Token.**isspace**() → bool
Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

istitle

Token.**istitle**() → bool
Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper

Token.**isupper**() → bool
Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

join

Token.**join**(*iterable*) → str
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

ljust

Token.**ljust**(*width*[, *fillchar*]) → str
Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

lower

Token.**lower**() → str
Return a copy of the string S converted to lowercase.

lstrip

Token.**lstrip**([*chars*]) → str
Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead.

maketrans

Token.**maketrans**()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition

Token.**partition**(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

replace

Token.**replace**(*old*, *new*[, *count*]) → str
Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind

Token.**rfind**(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex

Token.**rindex**(*sub*[, *start*[, *end*]]) → int
Like S.rfind() but raise ValueError when the substring is not found.

rjust

Token.**rjust**(*width*[, *fillchar*]) → str
Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space).

rpartition

Token.**rpartition**(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit

Token.**rsplit**(*sep=None*, *maxsplit=-1*) → list of strings
Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.

rstrip

Token.**rstrip**([*chars*]) → str
Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead.

split

Token.**split**(*sep=None*, *maxsplit=-1*) → list of strings
Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

splitlines

Token.**splitlines**([*keepends*]) → list of strings
Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

startswith

Token.**startswith**(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip

Token.**strip**([*chars*]) → str
Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead.

swapcase

Token.**swapcase**() → str
Return a copy of S with uppercase characters converted to lowercase and vice versa.

title

Token.**title**() → str
Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

translate

Token.**translate**(*table*) → str
Return a copy of the string S in which each character has been mapped through the given translation table. The table must implement lookup/indexing via __getitem__, for instance a dictionary or list, mapping Unicode ordinals to Unicode ordinals, strings, or None. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper

Token.**upper**() → str
Return a copy of S converted to uppercase.

zfill

Token.**zfill**(*width*) → str
Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

__init__()
Initialize self. See help(type(self)) for accurate signature.

io

It provides functions to read and save a dispatcher from/to files.

Functions

load_default_values	Load Dispatcher default values in Python pickle format.
load_dispatcher	Load Dispatcher object in Python pickle format.
load_map	Load Dispatcher map in Python pickle format.
save_default_values	Write Dispatcher default values in Python pickle format.
save_dispatcher	Write Dispatcher object in Python pickle format.
save_map	Write Dispatcher graph object in Python pickle format.

load_default_values

load_default_values(*dsp*, *path*)

Load Dispatcher default values in Python pickle format.

Pickles are a serialized byte stream of a Python object. This format will preserve Python objects used as nodes or edges.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **path** (*str, file*) – File or filename to write. File names ending in .gz or .bz2 will be uncompressed.

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher()
>>> dsp.add_data('a', default_value=1)
'a'
>>> dsp.add_function(function=max, inputs=['a', 'b'], outputs=['c'])
'max'
>>> save_default_values(dsp, file_name)

>>> dsp = Dispatcher(dmap=dsp.dmap)
>>> load_default_values(dsp, file_name)
>>> dsp.dispatch(inputs={'b': 3})['c']
3
```

load_dispatcher

load_dispatcher(*path*)

Load Dispatcher object in Python pickle format.

Pickles are a serialized byte stream of a Python object. This format will preserve Python objects used as nodes or edges.

Parameters **path** (*str, file*) – File or filename to write. File names ending in .gz or .bz2 will be uncompressed.

Returns A dispatcher that identifies the model adopted.

Return type schedula.Dispatcher

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher()
>>> dsp.add_data('a', default_value=1)
'a'
>>> dsp.add_function(function=max, inputs=['a', 'b'], outputs=['c'])
'max'
>>> save_dispatcher(dsp, file_name)

>>> dsp = load_dispatcher(file_name)
>>> dsp.dispatch(inputs={'b': 3})['c']
3
```

load_map

load_map(*dsp*, *path*)

Load Dispatcher map in Python pickle format.

Parameters

- **dsp** (*schedula.schedula.Dispatcher*) – A dispatcher that identifies the model to be upgraded.
- **path** (*str, file*) – File or filename to write. File names ending in .gz or .bz2 will be uncompressed.

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher()
>>> dsp.add_function(function=max, inputs=['a', 'b'], outputs=['c'])
'max'
>>> save_map(dsp, file_name)

>>> dsp = Dispatcher()
>>> load_map(dsp, file_name)
>>> dsp.dispatch(inputs={'a': 1, 'b': 3})['c']
3
```

save_default_values

save_default_values(*dsp*, *path*)

Write Dispatcher default values in Python pickle format.

Pickles are a serialized byte stream of a Python object. This format will preserve Python objects used as nodes or edges.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **path** (*str, file*) – File or filename to write. File names ending in .gz or .bz2 will be compressed.

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher()
>>> dsp.add_data('a', default_value=1)
'a'
>>> dsp.add_function(function=max, inputs=['a', 'b'], outputs=['c'])
'max'
>>> save_default_values(dsp, file_name)
```

save_dispatcher

save_dispatcher(*dsp*, *path*)

Write Dispatcher object in Python pickle format.

Pickles are a serialized byte stream of a Python object. This format will preserve Python objects used as nodes or edges.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **path** (*str, file*) – File or filename to write. File names ending in .gz or .bz2 will be compressed.

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher()
>>> dsp.add_data('a', default_value=1)
'a'
>>> dsp.add_function(function=max, inputs=['a', 'b'], outputs=['c'])
'max'
>>> save_dispatcher(dsp, file_name)
```

save_map

save_map(*dsp*, *path*)

Write Dispatcher graph object in Python pickle format.

Pickles are a serialized byte stream of a Python object. This format will preserve Python objects used as nodes or edges.

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model adopted.
- **path** (*str, file*) – File or filename to write. File names ending in .gz or .bz2 will be compressed.

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher()
>>> dsp.add_function(function=max, inputs=['a', 'b'], outputs=['c'])
'max'
>>> save_map(dsp, file_name)
```

sol

It contains a comprehensive list of all modules and classes within dispatcher.

Docstrings should provide sufficient understanding for any individual function.

Classes

Solution

Solution

class **Solution**(*dsp=None*, *inputs=None*, *outputs=None*, *wildcard=False*, *cutoff=None*, *inputs_dist=None*, *no_call=False*, *rm_unused_nds=False*, *wait_in=None*, *no_domain=False*, *_empty=False*, *index=(-1,)*, *stopper=None*)

Methods

__init__	
clear	
copy	
copy_structure	
fromkeys	If not specified, the value defaults to None.
get	
get_node	Returns a sub node of a dispatcher.
get_sub_dsp_from_workflow	
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last==False).
plot	Plots the Dispatcher with a graph in the DOT language with Graphviz.
pop	value. If key is not found, d is returned if given, otherwise KeyError
popitem	Pairs are returned in LIFO order if last is true or FIFO order if false.
run	
search_node_description	
setdefault	
update	
values	
web	Creates a dispatcher Flask app.

__init__

Solution.**__init__**(*dsp=None*, *inputs=None*, *outputs=None*, *wildcard=False*, *cutoff=None*, *inputs_dist=None*, *no_call=False*, *rm_unused_nds=False*, *wait_in=None*, *no_domain=False*, *_empty=False*, *index=(-1,)*, *stopper=None*)

clear

Solution.**clear**() → None. Remove all items from od.

copy

Solution.**copy**() → a shallow copy of od

copy_structure

Solution.**copy_structure**(***kwargs*)

fromkeys

Solution.**fromkeys**(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get

Solution.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_node

Solution.**get_node**(**node_ids*, *, *node_attr=none*)
Returns a sub node of a dispatcher.

Parameters

- **node_ids** (*str*) – A sequence of node ids or a single node id. The id order identifies a dispatcher sub-level.
- **node_attr** (*str, None, optional*) – Output node attr.

  If the searched node does not have this attribute, all its attributes are returned.

  When 'auto', returns the "default" attributes of the searched node, which are:

  – for data node: its output, and if not exists, all its attributes.

  – for function and sub-dispatcher nodes: the 'function' attribute.

  When 'description', returns the "description" of the searched node, searching also in function or sub-dispatcher input/output description.

  When 'output', returns the data node output.

  When 'default_value', returns the data node default value.

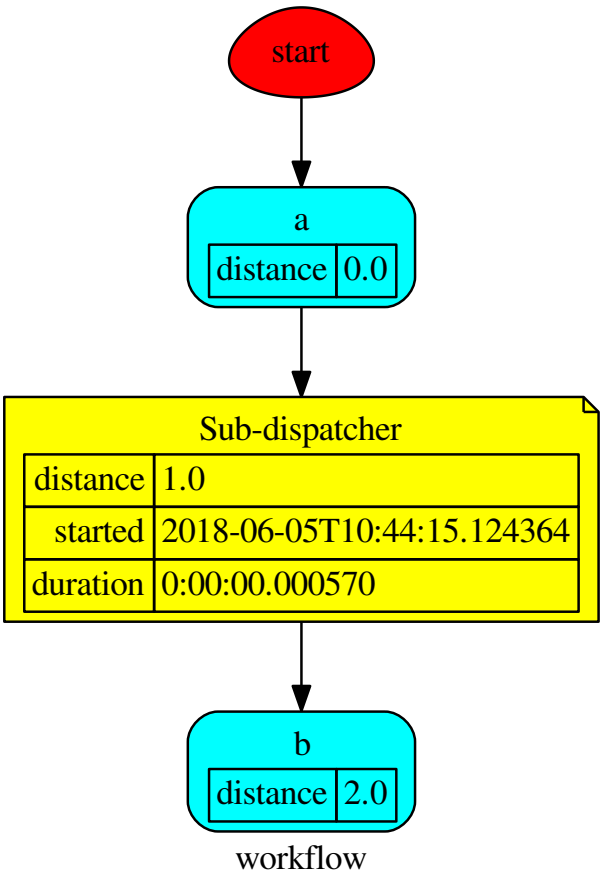
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

Return type (T, (str, ...))

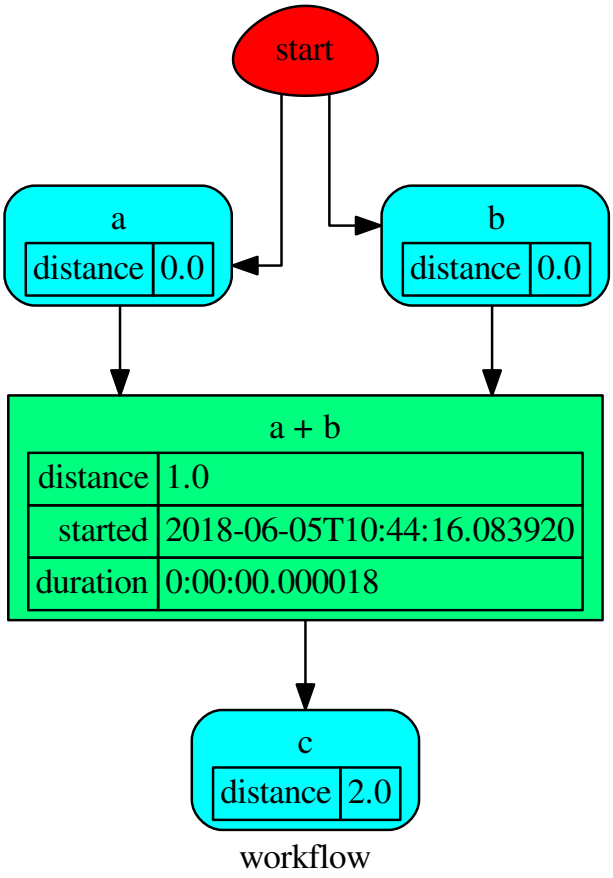
Example:



Get the sub node output:

```
>>> d.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> d.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))
```

```
>>> sub_dsp, sub_dsp_id = d.get_node('Sub-dispatcher')
```



get_sub_dsp_from_workflow

`Solution.`**`get_sub_dsp_from_workflow`**(*sources*, *reverse=False*, *add_missing=False*, *check_inputs=True*)

items

`Solution.`**`items`**()

keys

`Solution.`**`keys`**()

move_to_end

Solution.**move_to_end**()
Move an existing element to the end (or beginning if last==False).

Raises KeyError if the element does not exist. When last=True, acts like a fast version of self[key]=self.pop(key).

plot

Solution.**plot**(*workflow=None*, *view=True*, *depth=-1*, *name=none*, *comment=none*, *format=none*, *engine=none*, *encoding=none*, *graph_attr=none*, *node_attr=none*, *edge_attr=none*, *body=none*, *node_styles=none*, *node_data=none*, *node_function=none*, *edge_data=none*, *max_lines=none*, *max_width=none*, *directory=None*, *sites=None*, *index=False*)
Plots the Dispatcher with a graph in the DOT language with Graphviz.

Parameters

- **workflow** (*bool, optional*) – If True the latest solution will be plotted, otherwise the dmap.
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict, optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **index** (*bool, optional*) – Add the site index as first page?

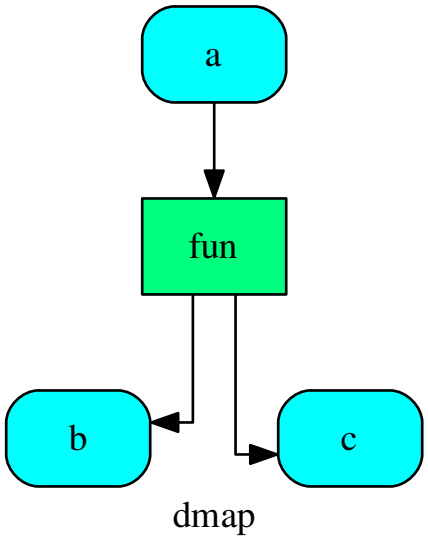
- **max_lines** (*int, optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int, optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

Returns A SiteMap.

Return type *schedula.utils.drw.SiteMap*

Example:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



pop

`Solution.`**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem

Solution.**popitem**() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

run

Solution.**run**()

search_node_description

Solution.**search_node_description**(*node_id*, *what='description'*)

setdefault

Solution.**setdefault**(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

update

Solution.**update**()

values

Solution.**values**()

web

Solution.**web**(*depth=-1*, *node_data=none*, *node_function=none*, *directory=None*, *sites=None*, *run=True*)
Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[~schedula.utils.drw.Site], optional*) – A set of *Site()* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

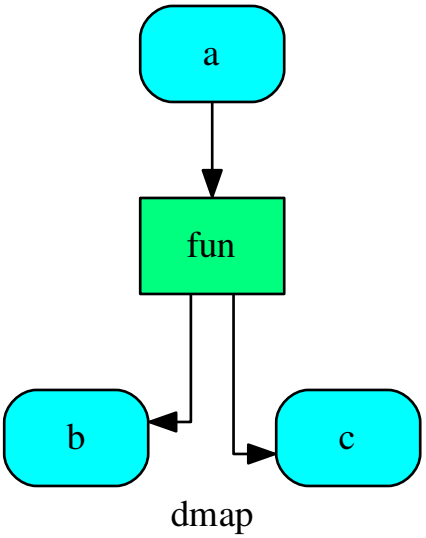
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

```
>>> from schedula import Dispatcher
>>> dsp = Dispatcher(name='Dispatcher')
>>> def fun(a):
...     return a + 1, a - 1
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
```



You can create a web server with the following steps:

```
>>> webmap = dsp.web()
>>> print("Starting...\n"); site = webmap.site().run(); site
Starting...
Site(WebMap([(Dispatcher, WebMap())]), host='localhost', ...)
>>> import requests
>>> url = '%s/%s/%s' % (site.url, dsp.name, fun.__name__)
>>> requests.post(url, json={'args': (0,)}).json()['return']
[1, -1]
>>> site.shutdown()  # Remember to shutdown the server.
True
```

Note: When *Site()* is garbage collected the server is shutdown automatically.

__init__(*dsp=None*, *inputs=None*, *outputs=None*, *wildcard=False*, *cutoff=None*, *inputs_dist=None*, *no_call=False*, *rm_unused_nds=False*, *wait_in=None*, *no_domain=False*, *_empty=False*, *index=(-1,)*, *stopper=None*)

Attributes

full_name	Returns the full node id.
pipe	

full_name

Solution.**full_name**
Returns the full node id.

Returns Full node id.

Return type tuple[str]

pipe

Solution.**pipe**

full_name
Returns the full node id.

Returns Full node id.

Return type tuple[str]

web

It provides functions to build a flask app from a dispatcher.

Functions

func_handler

func_handler

func_handler(*func*)

Classes

FolderNodeWeb
WebFolder
WebMap
WebNode

FolderNodeWeb

class FolderNodeWeb(*folder*, *node_id*, *attr*, ***options*)

Methods

__init__
dot
href
items
parent_ref
render_funcs
render_size
style
yield_attr

__init__

FolderNodeWeb.**__init__**(*folder*, *node_id*, *attr*, ***options*)

dot

FolderNodeWeb.**dot**(*context=None*)

href

FolderNodeWeb.**href**(*context*, *link_id*)

items

FolderNodeWeb.**items**()

parent_ref

FolderNodeWeb.**parent_ref**(*context*, *text*, *attr=None*)

render_funcs

FolderNodeWeb.**render_funcs**()

render_size

FolderNodeWeb.**render_size**(*out*)

style

FolderNodeWeb.**style**()

yield_attr

FolderNodeWeb.**yield_attr**(*name*)

__init__(*folder*, *node_id*, *attr*, ***options*)

Attributes

counter
edge_data
max_lines
max_width
node_data
node_function
node_map
node_styles
re_node
title
type

counter

FolderNodeWeb.**counter** = <method-wrapper '__next__' of itertools.count object>

edge_data

FolderNodeWeb.**edge_data** = ()

max_lines

FolderNodeWeb.**max_lines** = 5

max_width

FolderNodeWeb.**max_width** = 200

node_data

FolderNodeWeb.**node_data** = ()

node_function

FolderNodeWeb.**node_function** = ('+function',)

node_map

FolderNodeWeb.**node_map** = {'': ('dot', 'table'), '*': ('link',), '+': ('dot', 'table'), '!': ('dot', 'table'), '.': ('dot',), '-': ()

node_styles

FolderNodeWeb.**node_styles** = {'info': {start: {'fillcolor': 'red', 'shape': 'egg', 'label': 'start'}, plot: {'fillcolor': 'g

re_node

FolderNodeWeb.**re_node** = regex.Regex('^([.*+!]?)(\\w+)(?>\\|(\\w+))?$', flags=regex.V0)

title

FolderNodeWeb.**title**

type

FolderNodeWeb.**type**

WebFolder

class **WebFolder**(*item*, *dsp*, *graph*, *obj*, *name=''*, *workflow=False*, *digraph=None*, ***options*)

Methods

__init__
dot
view

__init__

WebFolder.**__init__**(*item*, *dsp*, *graph*, *obj*, *name=''*, *workflow=False*, *digraph=None*, ***options*)

dot

WebFolder.**dot**(*context=None*)

view

WebFolder.**view**(*filepath*, *context=None*, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*)

__init__(*item*, *dsp*, *graph*, *obj*, *name=''*, *workflow=False*, *digraph=None*, ***options*)

Attributes

counter
digraph
ext
filename
inputs
label_name
name
outputs
title
view_id

counter

WebFolder.**counter** = <method-wrapper '__next__' of itertools.count object>

digraph

WebFolder.**digraph** = {'graph_attr': {}, 'edge_attr': {}, 'format': 'svg', 'node_attr': {'style': 'filled'}, 'body': {'spline

ext

WebFolder.**ext** = ''

filename

WebFolder.**filename**

inputs

WebFolder.**inputs**

label_name

WebFolder.**label_name**

name

WebFolder.**name**

outputs

WebFolder.**outputs**

title

WebFolder.**title**

view_id

WebFolder.**view_id**

WebMap

class WebMap

Methods

__init__	
add_items	
app	
clear	
copy	
fromkeys	If not specified, the value defaults to None.
get	
get_dsp_from	
get_sol_from	
index_filenames	
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last==False).
pop	value. If key is not found, d is returned if given, otherwise KeyError
popitem	Pairs are returned in LIFO order if last is true or FIFO order if false.
render	
rules	
setdefault	
site	
site_index	
update	
values	

__init__

WebMap.**__init__**()

add_items

WebMap.**add_items**(*item*, *workflow=False*, *depth=-1*, ***options*)

app

WebMap.**app**(*root_path=None*, *depth=-1*, ***kwargs*)

clear

WebMap.**clear**() → None. Remove all items from od.

copy

WebMap.**copy**() → a shallow copy of od

fromkeys

WebMap.**fromkeys**(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

get

WebMap.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_dsp_from

WebMap.**get_dsp_from**(*item*)

get_sol_from

WebMap.**get_sol_from**(*item*)

index_filenames

WebMap.**index_filenames**()

items

WebMap.**items**()

keys

WebMap.**keys**()

move_to_end

WebMap.**move_to_end**()
Move an existing element to the end (or beginning if last==False).

Raises KeyError if the element does not exist. When last=True, acts like a fast version of self[key]=self.pop(key).

pop

WebMap.**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem

WebMap.**popitem**() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

render

WebMap.**render**(**args*, ***kwargs*)

rules

WebMap.**rules**(*depth=-1*, *index=True*)

setdefault

WebMap.**setdefault**(*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

site

WebMap.**site**(*root_path=None*, *depth=-1*, *index=True*, *view=False*, ***kw*)

site_index

WebMap.**site_index**(**args*, ***kwargs*)

update

WebMap.**update**()

values

WebMap.**values**()

__init__()

Attributes

include_folders_as_filenames
nodes
options

include_folders_as_filenames

WebMap.**include_folders_as_filenames** = False

nodes

WebMap.**nodes**

options

WebMap.**options** = {'node_function', 'edge_data', 'max_lines', 'max_width', 'node_styles', 'digraph', 'node_data'}

WebNode

class WebNode (*folder*, *node_id*, *item*, *obj*)

Methods

__init__
render
view

__init__

WebNode.**__init__** (*folder*, *node_id*, *item*, *obj*)

render

WebNode.**render** (**args*, ***kwargs*)

view

WebNode.**view**(*filepath*, **args*, ***, *header='\n \n <input type="button" VALUE="Back"\n onClick="window.history.back()">\n <input type="button" VALUE="Forward"\n onClick="window.history.forward()">\n \n'*, ***kwargs*)

__init__(*folder*, *node_id*, *item*, *obj*)

Attributes

counter
ext
filename
name
title
view_id

counter

WebNode.**counter** = **<method-wrapper '__next__' of itertools.count object>**

ext

WebNode.**ext** = **''**

filename

WebNode.**filename**

name

WebNode.**name**

title

WebNode.**title**

view_id

WebNode.**view_id**

ext

It provides sphinx extensions.

Extensions:

autosummary	It is a patch to shpinx.ext.autosummary.
dispatcher	

autosummary

It is a patch to shpinx.ext.autosummary.

Functions

generate_autosummary_docs
get_members
process_generate_options
setup

generate_autosummary_docs

generate_autosummary_docs(*sources*, *output_dir=None*, *suffix='.rst'*, *warn=<function _simple_warn>*, *info=<function _simple_info>*, *base_path=None*, *builder=None*, *template_dir=None*)

get_members

get_members(*obj*, *typ*, *include_public=()*, *imported=False*)

process_generate_options

process_generate_options(*app*)

setup

setup(*app*)

dispatcher

Functions

setup

setup

setup(*app*)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

B

C

D

E

F

G

J

K

L

M

N

P

R

S

T

U

V

W