schedula Documentation

Release 0.3.7

Vincenzo Arcidiacono

Dec 06, 2019

Table of Contents

2019-12-06 15:50:00

https://github.com/vinci1it2000/schedula

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

http://schedula.readthedocs.io/

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

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

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

- Vincenzo Arcidiacono <vincenzo.arcidiacono@ext.jrc.ec.europa.eu>

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
```

2.1 Install extras

Some additional functionality is enabled installing the following extras:

- plot: enables the plot of the Dispatcher model and workflow (see *plot()*).
- web: enables to build a dispatcher Flask app (see *web()*).
- sphinx: enables the sphinx extension directives (i.e., autosummary and dispatcher).
- parallel: enables the parallel execution of Dispatcher model.

To install schedula and all extras, do:

```
$ pip install schedula[all]
```

Note: `plot` extra requires **Graphviz**. Make sure that the directory containing the `dot` executable is on your systems' path. If you have not you can install it from its download page.

2.1.1 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.

2.1.2 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
```

2.1.2.1 Install extras

Some additional functionality is enabled installing the following extras:

- plot: enables the plot of the Dispatcher model and workflow (see *plot()*).
- web: enables to build a dispatcher Flask app (see *web()*).
- sphinx: enables the sphinx extension directives (i.e., autosummary and dispatcher).
- parallel: enables the parallel execution of Dispatcher model.

To install schedula and all extras, do:

```
$ pip install schedula[all]
```

Note: `plot` extra requires **Graphviz**. Make sure that the directory containing the `dot` executable is on your systems' path. If you have not you can install it from its download page.

2.1.3 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.

2.1.3.1 Solution

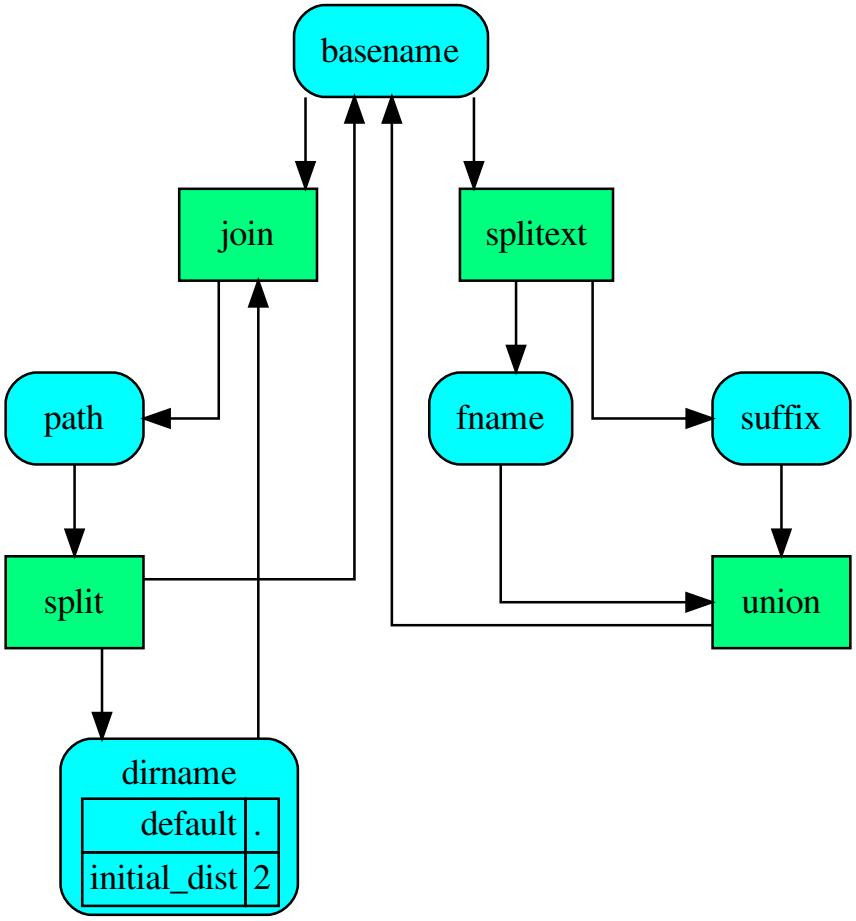
Schedula allows to write a simple model (*Dispatcher*) with just the basic functions, then the *Dispatcher* will select and execute the proper functions for the given inputs and the requested outputs. Moreover, schedula provides a flexible framework for structuring code. It allows to extract sub-models from a bigger one and to run your model asynchronously or in parallel without extra coding.

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

2.1.4 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 as sh
>>> import os.path as osp
>>> dsp = sh.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_func()*, *add_function()*, *add_dispatcher()*, *SubDispatch*, *SubDispatchFunction*, *SubDispatchPipe*, *DispatchPipe*, and DFun.

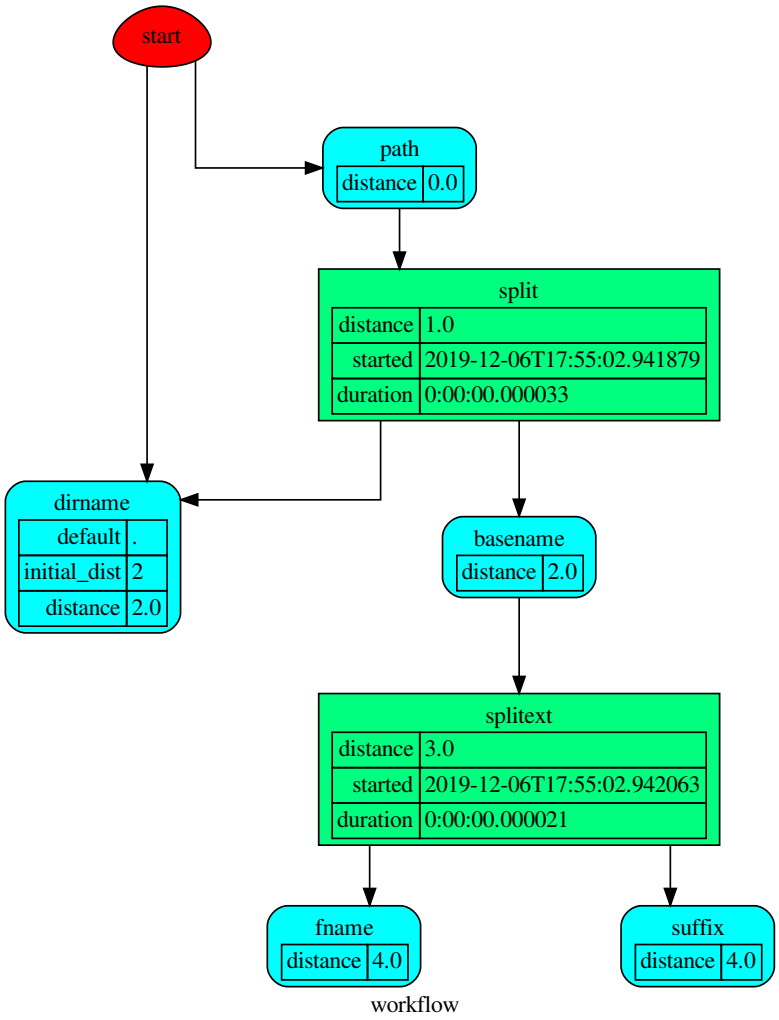
The next step to calculate the outputs would be just to run the *dispatch()* method. You can invoke it with just the inputs, so it will calculate all reachable outputs:

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

(continues on next page)

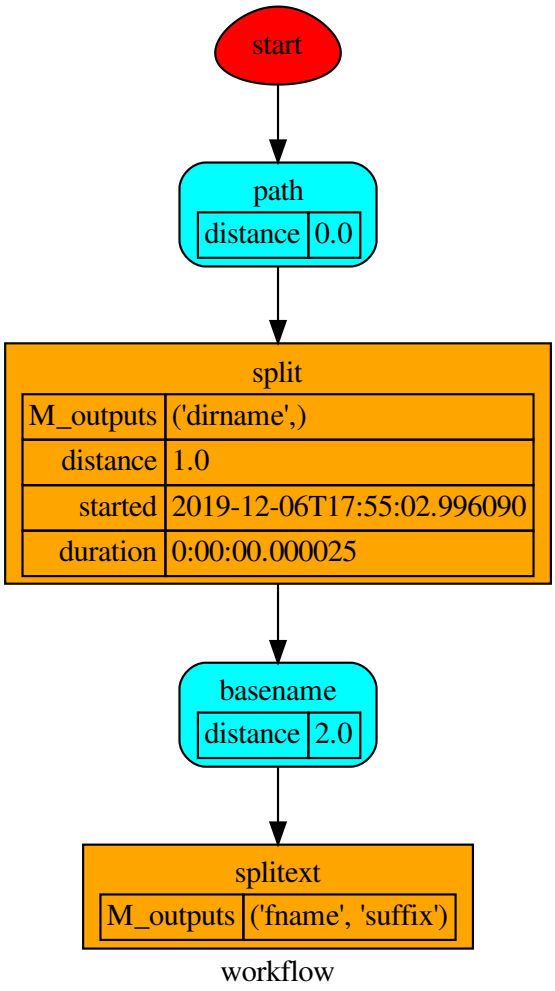
(continued from previous page)

```
        ('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

2.1.5 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!!!).

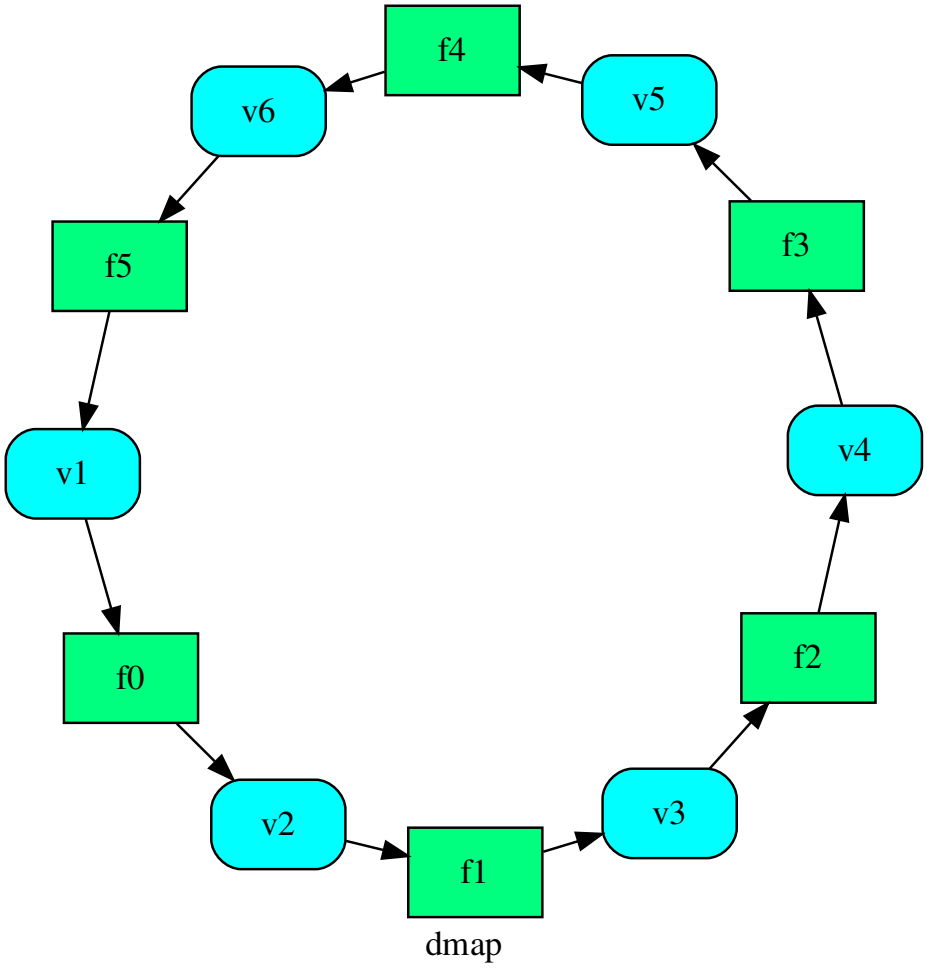
Schedula will simplify your life. You just create a *Dispatcher*, that contains all functions that link your data:

```
>>> import schedula as sh
>>> dsp = sh.Dispatcher()
>>> increment = lambda x: x + 1
>>> for k, (i, j) in enumerate(sh.pairwise([1, 2, 3, 4, 5, 6, 1])):
```

(continues on next page)

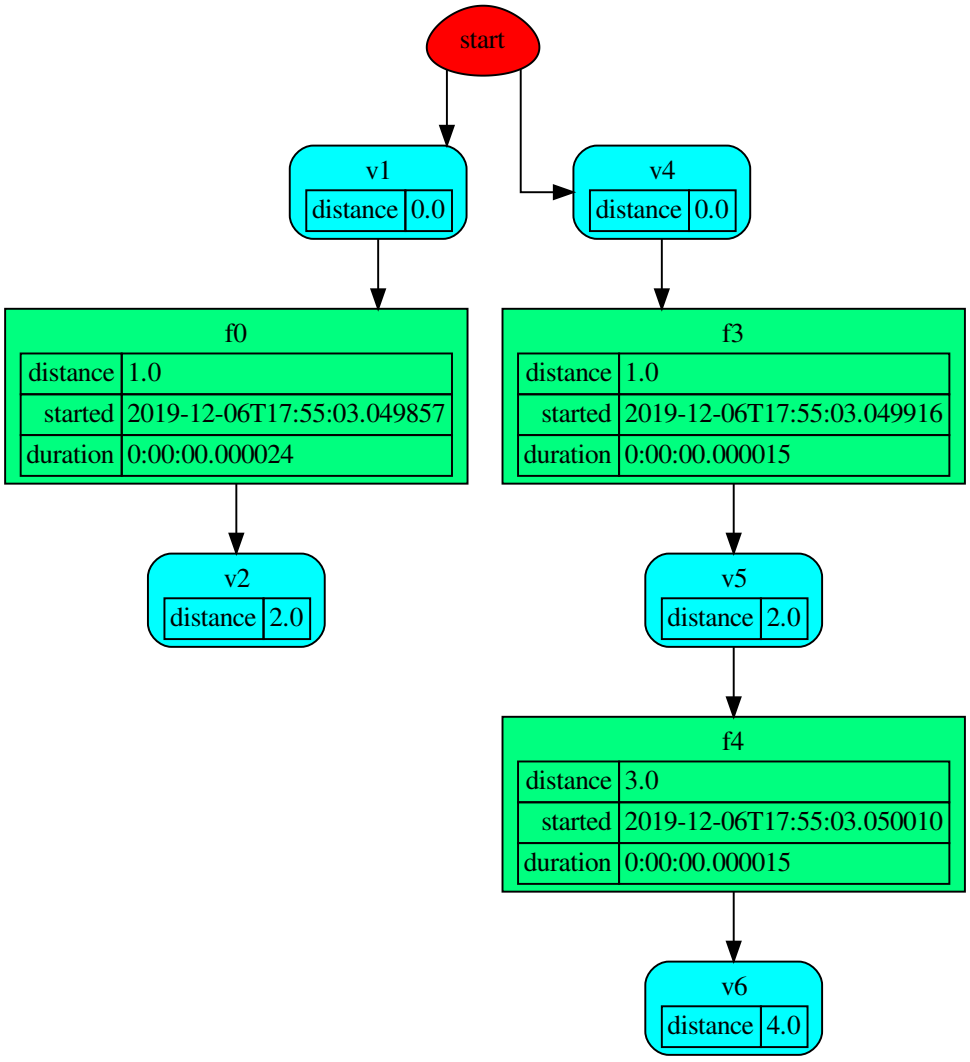
(continued from previous page)

```
...     dsp.add_function('f%d' % k, increment, ['v%d' % i], ['v%d' % j])
'...'
```



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', 2), ('v6', 3)])
```

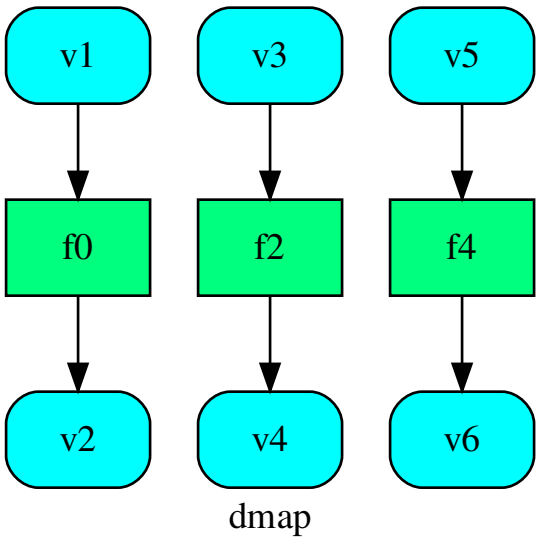


workflow

2.1.5.1 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*, *DispatchPipe*, and *SubDispatchPipe*.

2.1.5.2 Iterated function

Schedula allows to build an iterated function, i.e. the input is recalculated. This could be done easily with the *DispatchPipe*, as follows:

```
>>> func = sh.DispatchPipe(dsp, 'func', ('v1', 'v4'), ('v1', 'v4'))
>>> x = [[1, 4]]
>>> for i in range(6):
...     x.append(func(*x[-1]))
>>> x
[[1, 4], [7, 4], [7, 10], [13, 10], [13, 16], [19, 16], [19, 22]]
```

2.1.6 Asynchronous and Parallel dispatching

When there are heavy calculations which takes a significant amount of time, you want to run your model asynchronously or in parallel. Generally, this is difficult to achieve, because it requires an higher level of abstraction and a deeper knowledge of python programming and the Global Interpreter Lock (GIL). Schedula will simplify again your life. It has four default executors to dispatch asynchronously or in parallel:

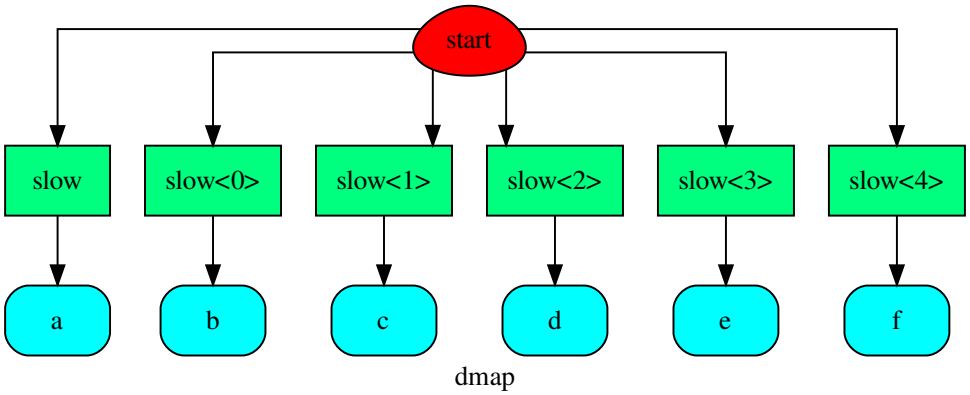
- *async*: execute all functions asynchronously in the same process,
- *parallel*: execute all functions in parallel excluding *SubDispatch* functions,
- *parallel-pool*: execute all functions in parallel using a process pool excluding *SubDispatch* functions,

- *parallel-dispatch*: execute all functions in parallel including `SubDispatch`.

Note: Running functions asynchronously or in parallel has a cost. Schedula will spend time creating / deleting new threads / processes.

The code below shows an example of a time consuming code, that with the concurrent execution it requires at least 6 seconds to run. Note that the *slow* function return the process id.

```
>>> import schedula as sh
>>> dsp = sh.Dispatcher()
>>> def slow():
...     import os, time
...     time.sleep(1)
...     return os.getpid()
>>> for o in 'abcdef':
...     dsp.add_function(function=slow, outputs=[o])
'...'
```



dmap

while using the *async* executor, it lasts a bit more then 1 second:

```
>>> import time
>>> start = time.time()
>>> sol = dsp(executor='async').result()  # Asynchronous execution.
>>> (time.time() - start) < 2  # Faster then concurrent execution.
True
```

all functions have been executed asynchronously, but in the same process:

```
>>> import os
>>> pid = os.getpid()  # Current process id.
>>> {sol[k] for k in 'abcdef'} == {pid}  # Single process id.
True
```

if we use the *parallel* executor all functions are executed in different processes:

```
>>> sol = dsp(executor='parallel').result()  # Parallel execution.
>>> pids = {sol[k] for k in 'abcdef'}  # Process ids returned by `slow`.
```

(continues on next page)

(continued from previous page)

```
>>> len(pids) == 6  # Each function returns a different process id.
True
>>> pid not in pids  # The current process id is not in the returned pids.
True
>>> sorted(sh.shutdown_executors())
['async', 'parallel']
```

2.1.7 Next moves

Things yet to do: utility to transform a dispatcher in a command line tool.

2.1.8 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.

2.1.8.1 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=''*, *executor=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.

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.

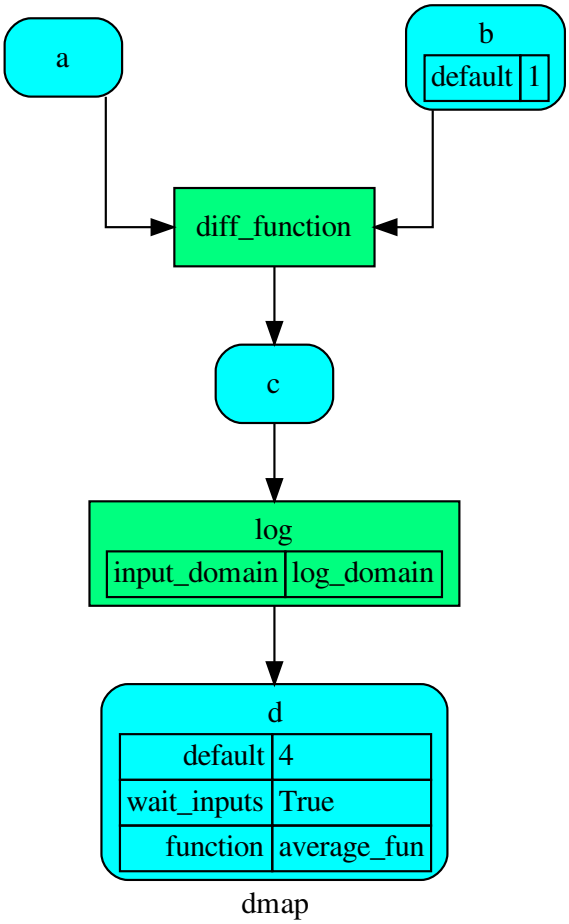
- function estimation: estimate one unique output from multiple estimations.
- callback function: is invoked after computing the output.

```
>>> 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
...     '''
```

(continues on next page)

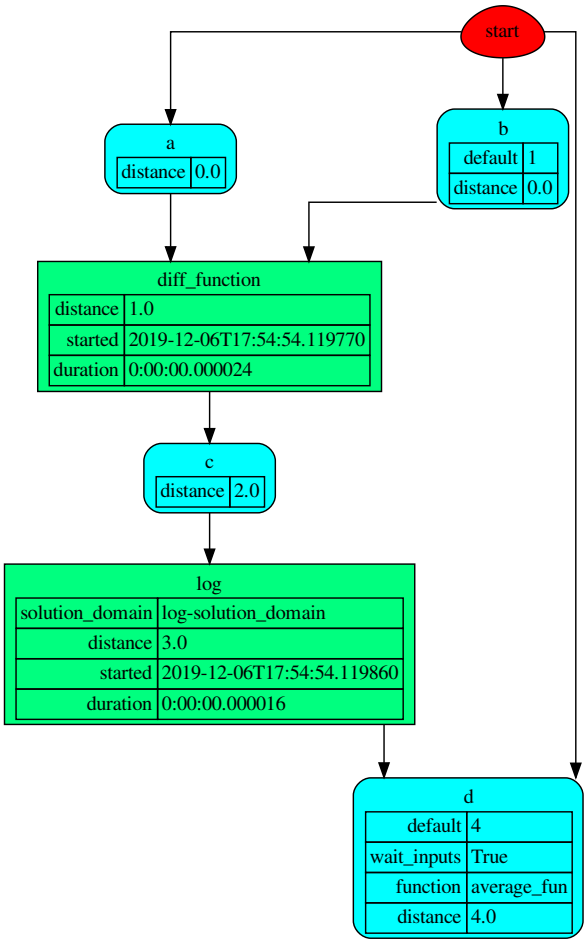
(continued from previous page)

```
...
...     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_func	Add a single function node to dispatcher.
add_function	Add a single function node to dispatcher.
blue	Constructs a BlueDispatcher out of the current object.
copy	Returns a copy of the Dispatcher.
copy_structure	
dispatch	Evaluates the minimum workflow and data outputs of the dispatcher model from given inputs.
extend	Extends Dispatcher calling each deferred operation of given Blueprints.

Continued on next page

Table 3 – continued from previous page

get_node	Returns a sub node of a dispatcher.
get_sub_dsp	Returns the sub-dispatcher induced by given node and edge bunches.
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_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=''*, *executor=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|callable, optional*) – If True the dispatcher interrupt the dispatch when an error occur, otherwise it logs a warning. If a callable is given it will be executed passing the exception to decide to raise or not the exception.
- **description** (*str, optional*) – The dispatcher's description.
- **executor** (*str, optional*) – A pool executor id to dispatch asynchronously or in parallel.

  There are four default Pool executors to dispatch asynchronously or in parallel:

  - *async*: execute all functions asynchronously in the same process,
  - *parallel*: execute all functions in parallel excluding *SubDispatch* functions,
  - *parallel-pool*: execute all functions in parallel using a process pool excluding *SubDispatch* functions,
  - *parallel-dispatch*: execute all functions in parallel including *SubDispatch*.

add_data

Dispatcher.**add_data**(*data_id=None*, *default_value=empty*, *initial_dist=0.0*, *wait_inputs=False*, *wildcard=None*, *function=None*, *callback=None*, *description=None*, *filters=None*, *await_result=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.
- **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.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits data results before assigning them to the solution. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Data node id.

Return type str

See also:

add_func(), *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*, *await_domain=None*, ***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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Sub-dispatcher node id.

Return type str

See also:

add_data(), *add_func()*, *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', 'b'): ('E', 'E1')},
...                    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_func()*, *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_func

Dispatcher.**add_func**(*function*, *outputs=None*, *weight=None*, *inputs_defaults=False*, *inputs_kwargs=False*, *filters=None*, *input_domain=None*, *await_domain=None*, *await_result=None*, *inp_weight=None*, *out_weight=None*, *description=None*, *inputs=None*, *function_id=None*, ***kwargs*)

Add a single function node to dispatcher.

Parameters

- **inputs_kwargs** (*bool*) – Do you want to include kwargs as inputs?
- **inputs_defaults** (*bool*) – Do you want to set default values?
- **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. If None it will take parameters names from function signature.
- **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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.

- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Function node id.

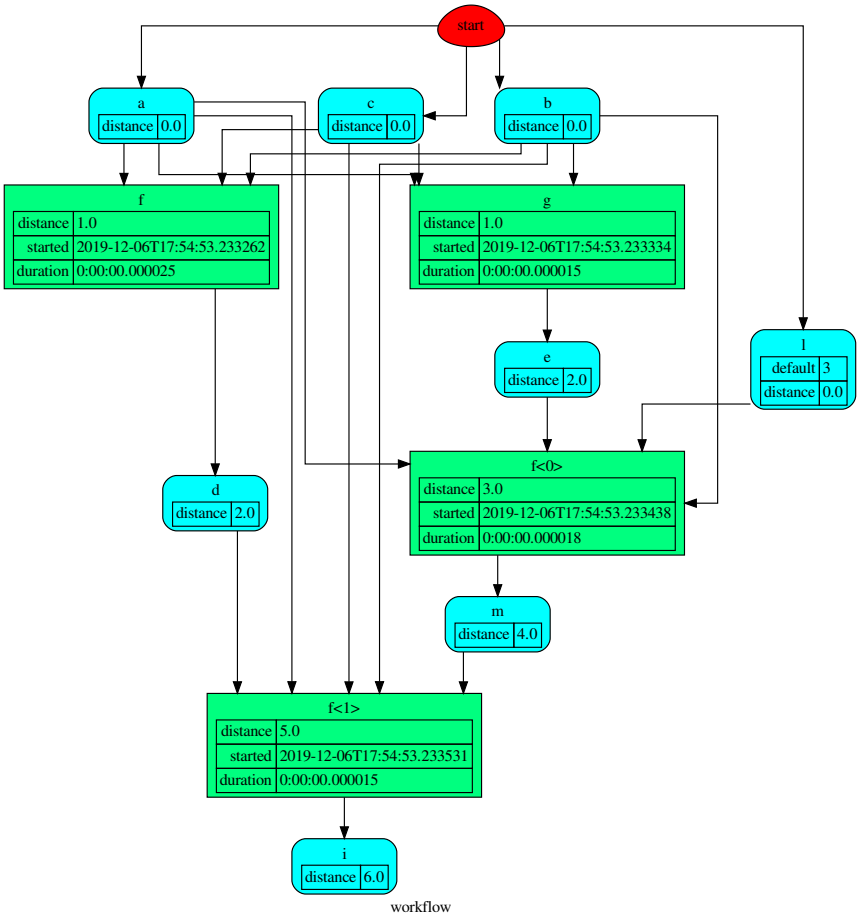
Return type str

See also:

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

Example:

```
>>> import schedula as sh
>>> dsp = sh.Dispatcher(name='Dispatcher')
>>> def f(a, b, c, d=3, m=5):
...     return (a + b) - c + d - m
>>> dsp.add_func(f, outputs=['d'])
'f'
>>> dsp.add_func(f, ['m'], inputs_defaults=True, inputs='beal')
'f<0>'
>>> dsp.add_func(f, ['i'], inputs_kwargs=True)
'f<1>'
>>> def g(a, b, c, *args, d=0):
...     return (a + b) * c + d
>>> dsp.add_func(g, ['e'], inputs_defaults=True)
'g'
>>> sol = dsp({'a': 1, 'b': 3, 'c': 0}); sol
Solution([('a', 1), ('b', 3), ('c', 0), ('l', 3), ('d', 2),
          ('e', 0), ('m', 0), ('i', 6)])
```



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*, *await_domain=None*, *await_result=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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Function node id.

Return type str

See also:

add_data(), *add_func()*, *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)
```

(continues on next page)

(continued from previous page)

```
...
>>> 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'
```

blue

Dispatcher.**blue**(*memo=None*)

Constructs a BlueDispatcher out of the current object.

Parameters **memo** (*dict[T, schedula.utils.blue.Blueprint]*) – A dictionary to cache Blueprints.

Returns A BlueDispatcher of the current object.

Return type *schedula.utils.blue.BlueDispatcher*

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*, *executor=False*, *sol_name=()*)

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:

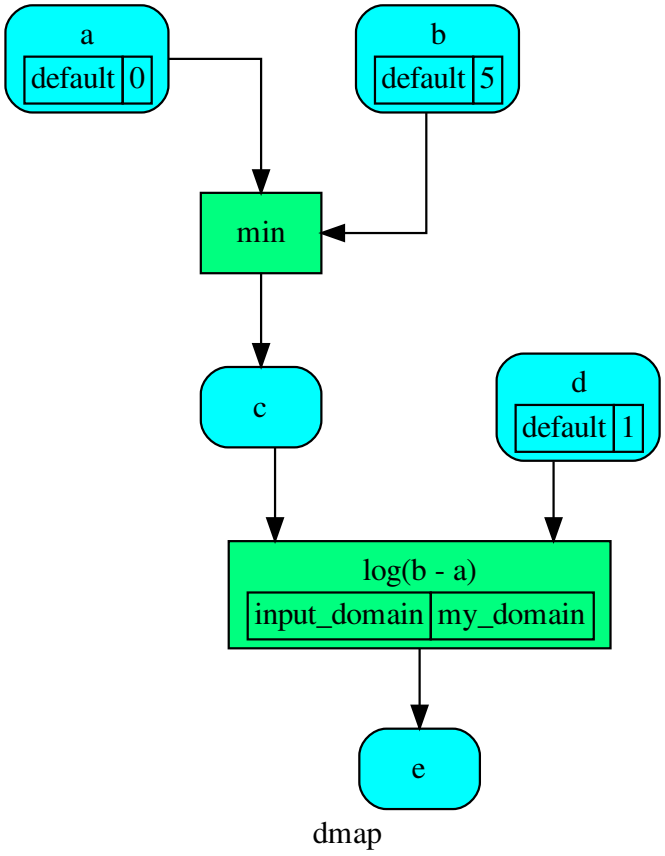
  shrink_dsp()
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **select_output_kw** (*dict, optional*) – Kwargs of selector function to select specific outputs.
- **_wait_in** (*dict, optional*) – Override wait inputs.
- **stopper** (*multiprocess.Event, optional*) – A semaphore to abort the dispatching.
- **executor** (*str, optional*) – A pool executor id to dispatch asynchronously or in parallel.
- **sol_name** (*tuple[str], optional*) – Solution name.

Returns Dictionary of estimated data node outputs.

Return type *schedula.utils.sol.Solution*

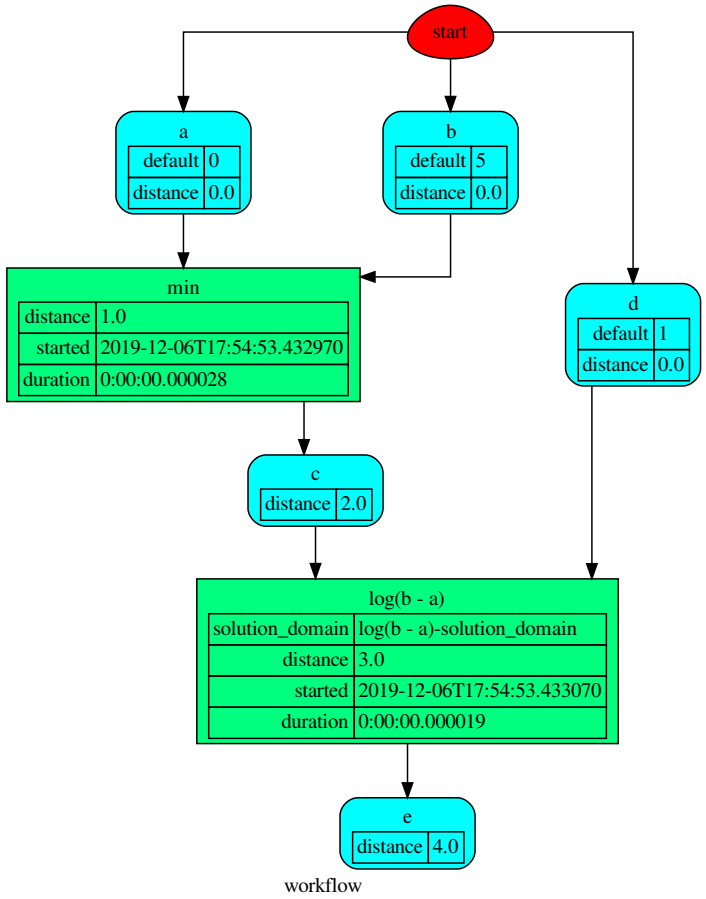
Example:

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



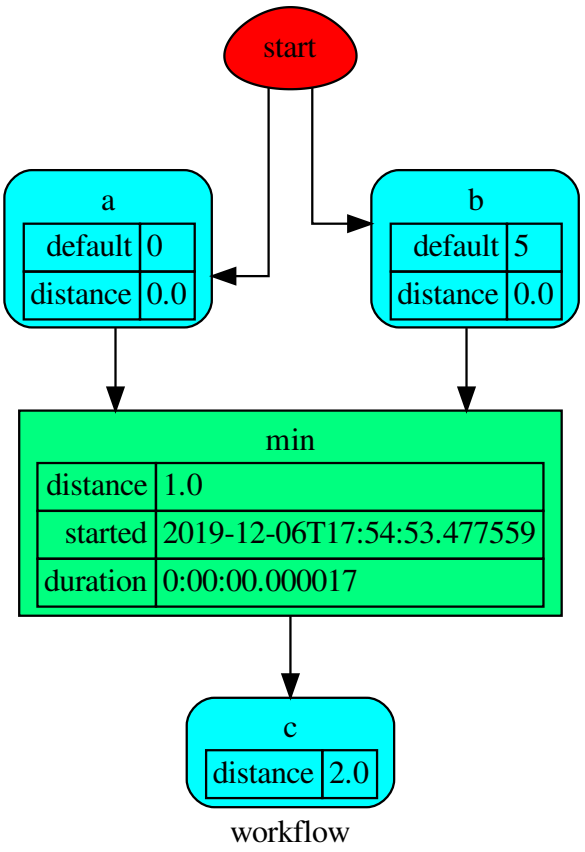
Dispatch without inputs. The default values are used as inputs:

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



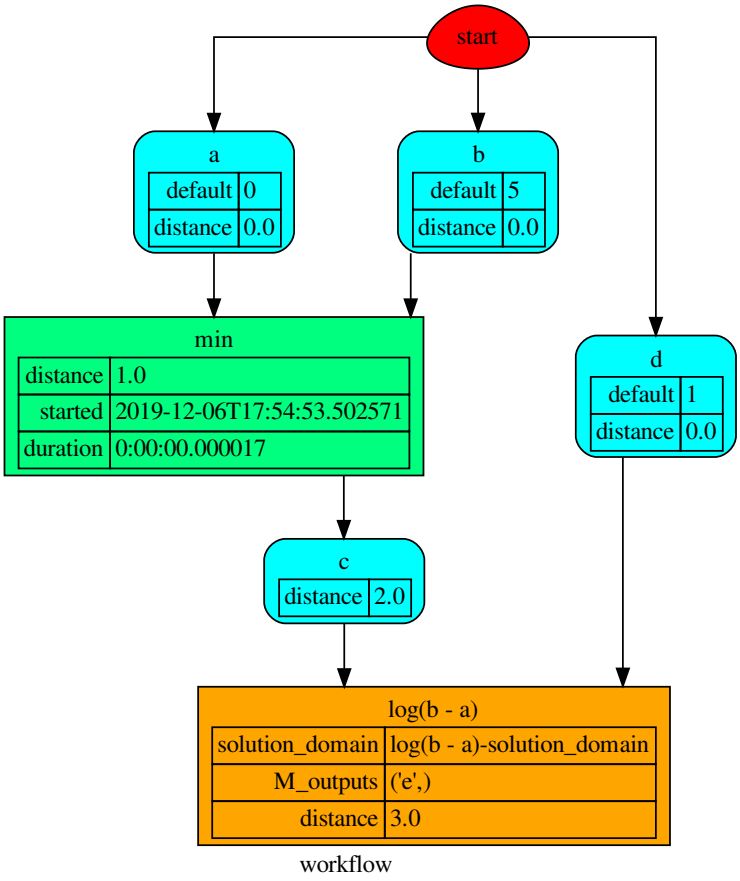
Dispatch until data node *c* is estimated:

```
>>> 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)])
```



workflow

extend

`Dispatcher.`**`extend`**`(*blues, memo=None)`

Extends Dispatcher calling each deferred operation of given Blueprints.

Parameters

- **blues** (*Blueprint | schedula.dispatcher.Dispatcher*) – Blueprints or Dispatchers to extend deferred operations.
- **memo** (*dict[T, schedula.utils.blue.Blueprint|Dispatcher]*) – A dictionary to cache Blueprints and Dispatchers.

Returns Self.

Return type *Dispatcher*

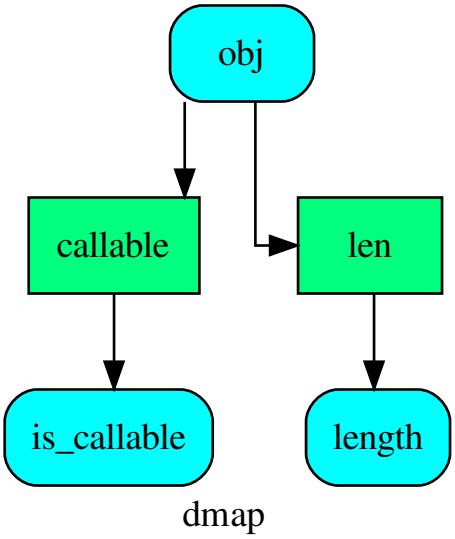
Example:

```
>>> import schedula as sh
>>> dsp = sh.Dispatcher()
```

(continues on next page)

(continued from previous page)

```
>>> dsp.add_func(callable, ['is_callable'])
'callable'
>>> blue = sh.BlueDispatcher().add_func(len, ['length'])
>>> dsp = sh.Dispatcher().extend(dsp, blue)
```



dmap

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.

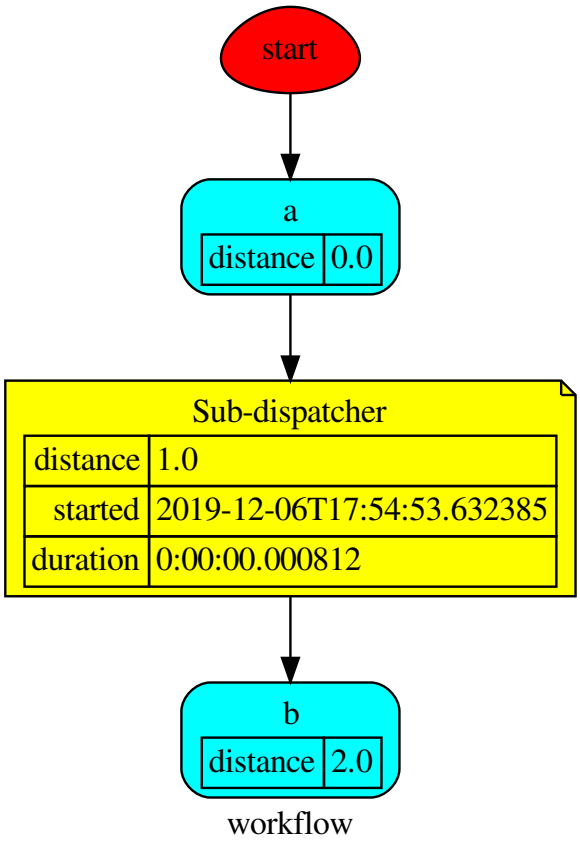
  When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

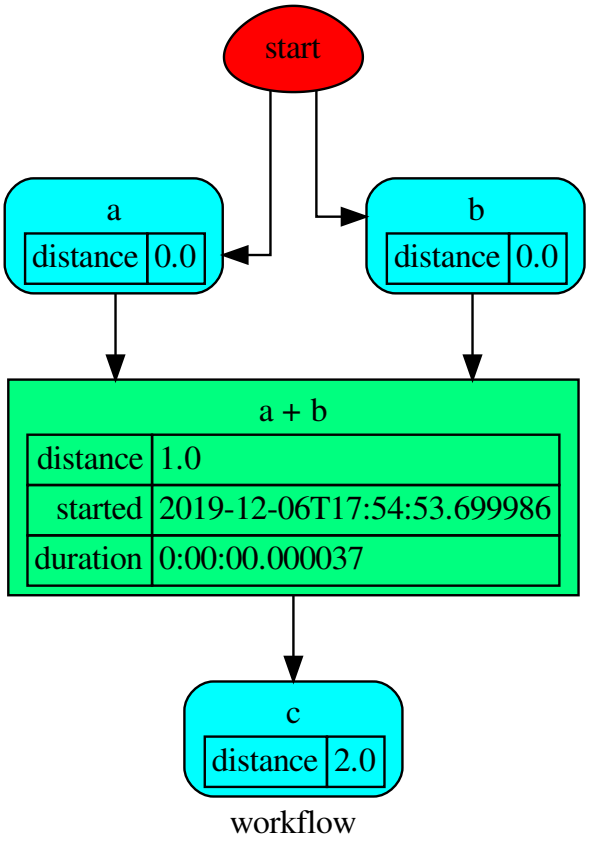
Example:



Get the sub node output:

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

```
>>> sub_dsp, sub_dsp_id = dsp.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*

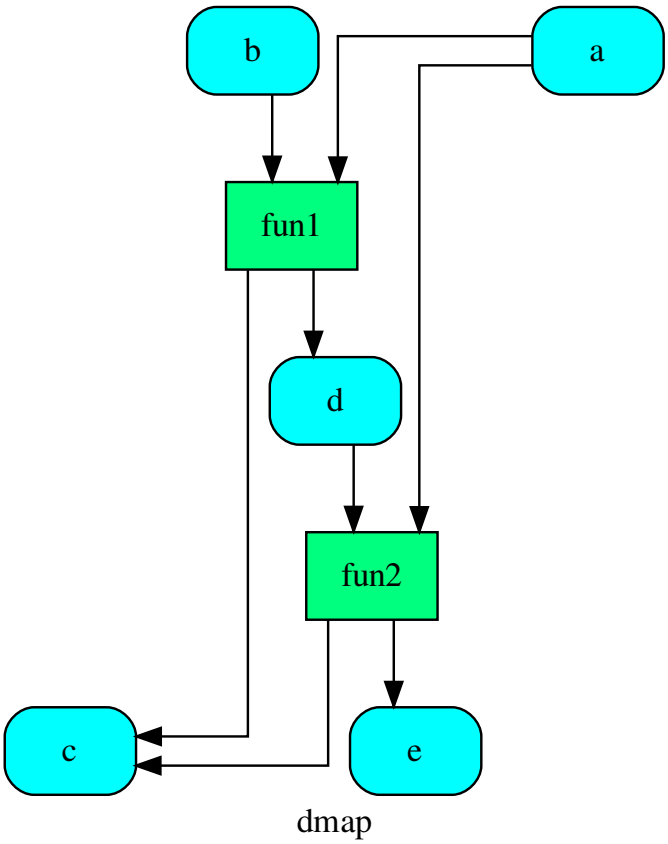
See also:

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.

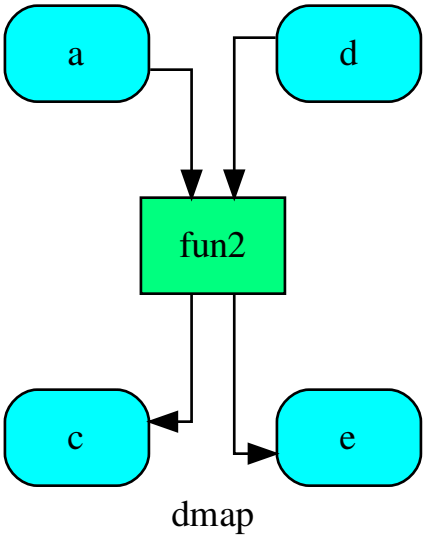
Example:

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*

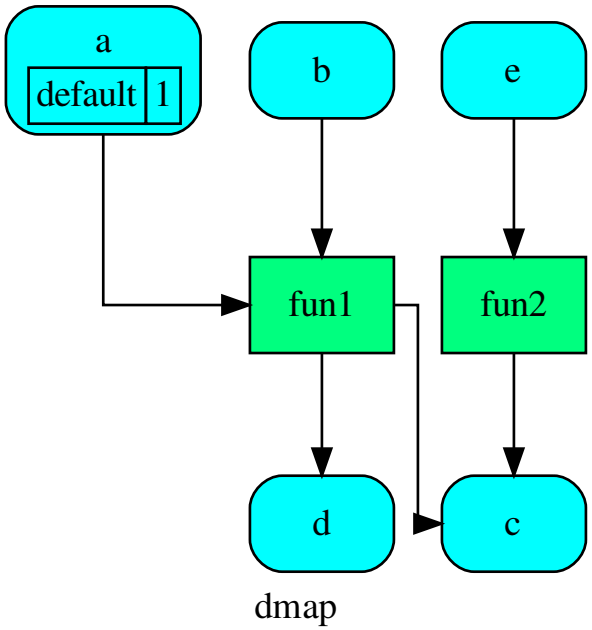
See also:

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:

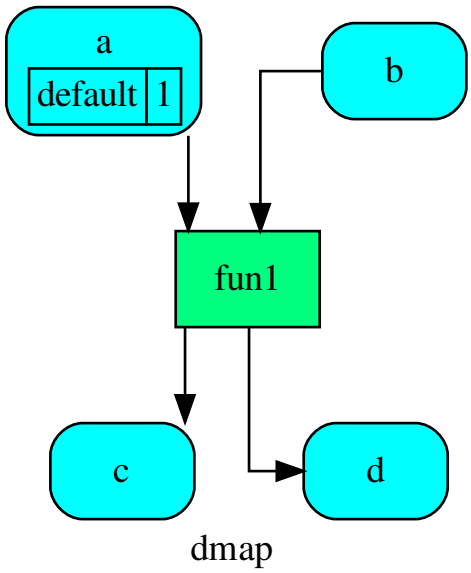


Dispatch with no calls in order to have a workflow:

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

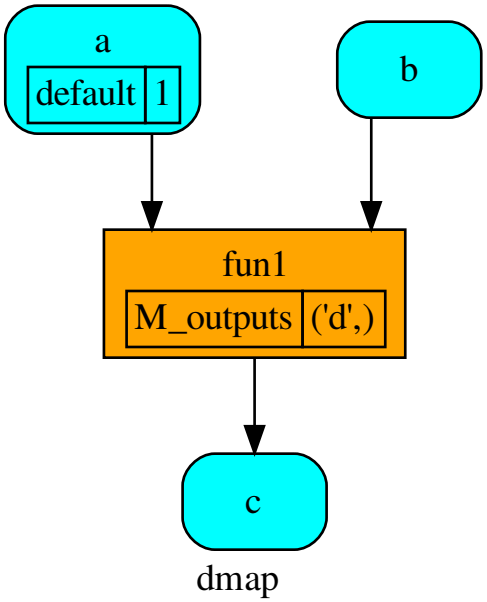
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> 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.

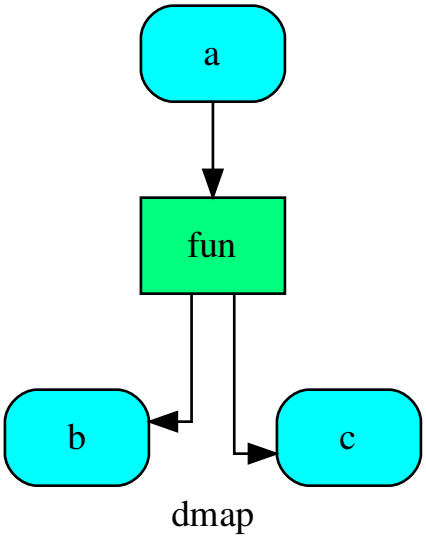
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str, optional*) – (Sub)directory for source saving and rendering.
- **format** (*str, optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **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[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_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')
...
```

(continues on next page)

(continued from previous page)

```
>>> 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.

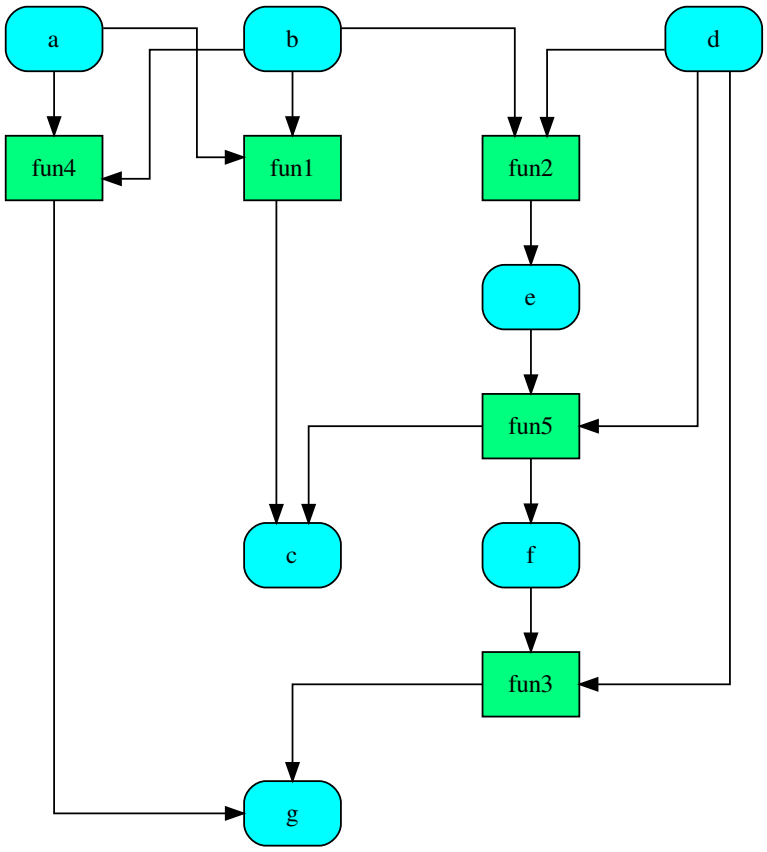
Return type *Dispatcher*

See also:

dispatch()

Example:

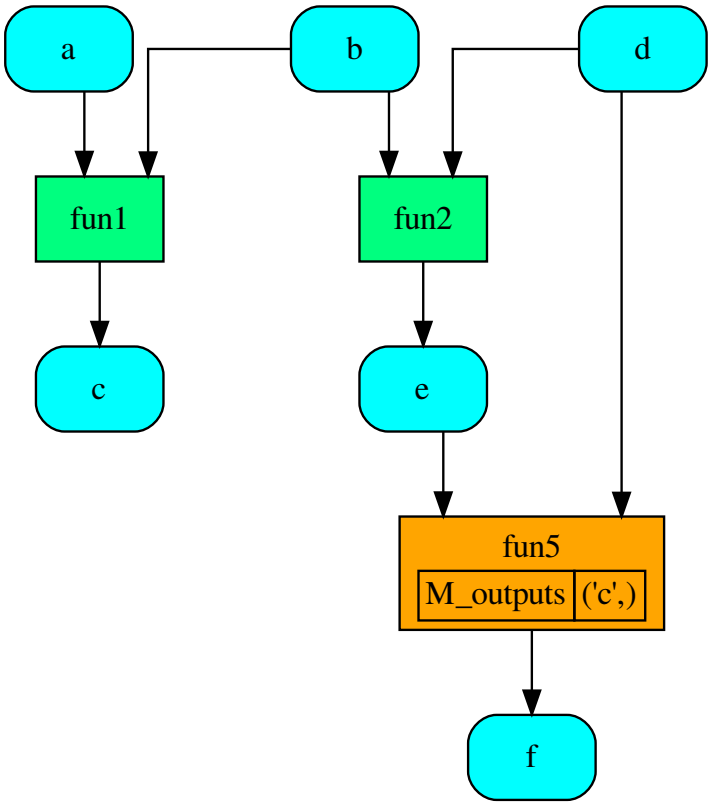
A dispatcher like this:



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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

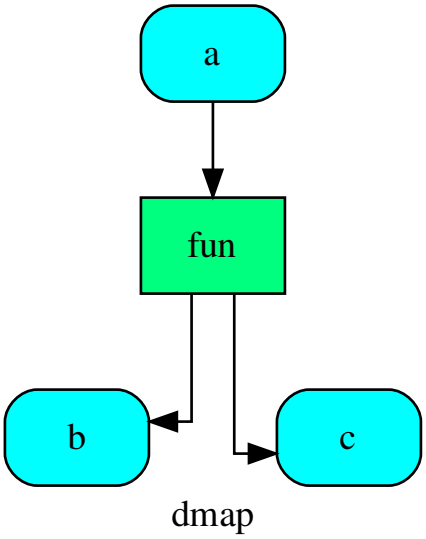
Returns A WebMap.

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=''*, *executor=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|callable, optional*) – If True the dispatcher interrupt the dispatch when an error occur, otherwise it logs a warning. If a callable is given it will be executed passing the exception to decide to raise or not the exception.
- **description** (*str, optional*) – The dispatcher's description.
- **executor** (*str, optional*) – A pool executor id to dispatch asynchronously or in parallel.

  There are four default Pool executors to dispatch asynchronously or in parallel:

  – *async*: execute all functions asynchronously in the same process,

  – *parallel*: execute all functions in parallel excluding *SubDispatch* functions,

  – *parallel-pool*: execute all functions in parallel using a process pool excluding *SubDispatch* functions,

  – *parallel-dispatch*: execute all functions in parallel including *SubDispatch*.

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.

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

executor = None
Pool executor to dispatch asynchronously.

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*, *description=None*, *filters=None*, *await_result=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.
- **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.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits data results before assigning them to the solution. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Data node id.

Return type str

See also:

add_func(), *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,
```

(continues on next page)

(continued from previous page)

```
...                 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*, *await_domain=None*, *await_result=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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.

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

Returns Function node id.

Return type str

See also:

add_data(), *add_func()*, *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_func(*function*, *outputs=None*, *weight=None*, *inputs_defaults=False*, *inputs_kwargs=False*, *filters=None*, *input_domain=None*, *await_domain=None*, *await_result=None*, *inp_weight=None*, *out_weight=None*, *description=None*, *inputs=None*, *function_id=None*, ***kwargs*)

Add a single function node to dispatcher.

Parameters

- **inputs_kwargs** (*bool*) – Do you want to include kwargs as inputs?
- **inputs_defaults** (*bool*) – Do you want to set default values?
- **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. If None it will take parameters names from function signature.
- **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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Function node id.

Return type str

See also:

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

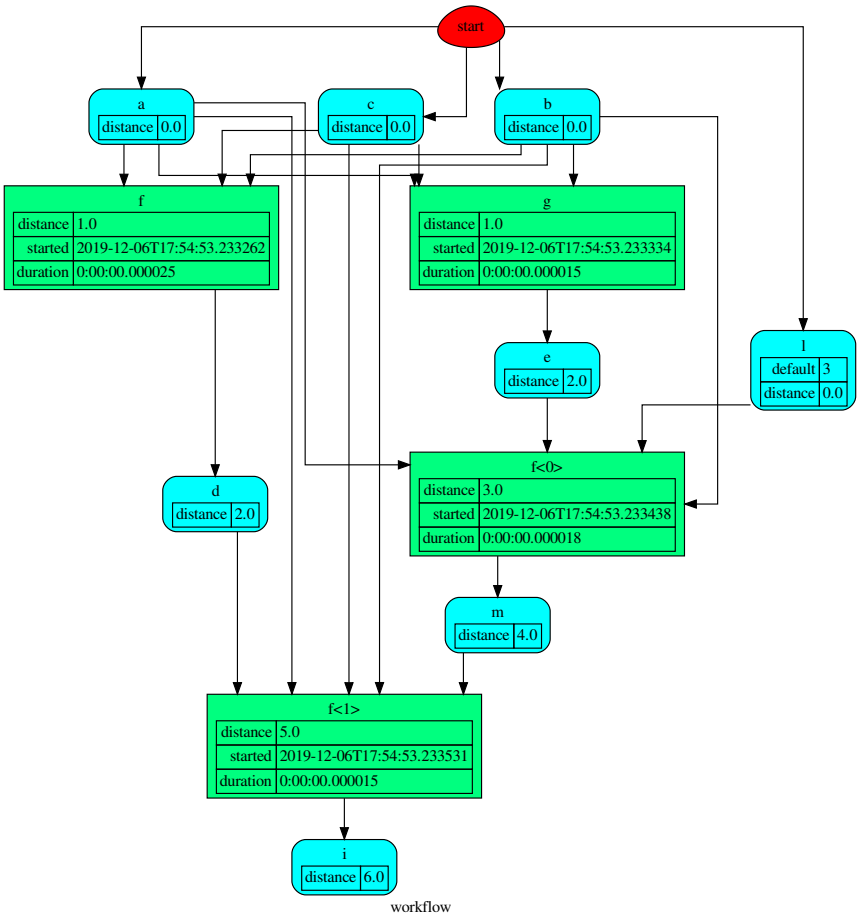
Example:

```
>>> import schedula as sh
>>> dsp = sh.Dispatcher(name='Dispatcher')
>>> def f(a, b, c, d=3, m=5):
...     return (a + b) - c + d - m
>>> dsp.add_func(f, outputs=['d'])
'f'
>>> dsp.add_func(f, ['m'], inputs_defaults=True, inputs='beal')
'f<0>'
>>> dsp.add_func(f, ['i'], inputs_kwargs=True)
'f<1>'
>>> def g(a, b, c, *args, d=0):
...     return (a + b) * c + d
>>> dsp.add_func(g, ['e'], inputs_defaults=True)
```

(continues on next page)

(continued from previous page)

```
'g'
>>> sol = dsp({'a': 1, 'b': 3, 'c': 0}); sol
Solution([('a', 1), ('b', 3), ('c', 0), ('l', 3), ('d', 2),
          ('e', 0), ('m', 0), ('i', 6)])
```



add_dispatcher(*dsp*, *inputs*, *outputs*, *dsp_id=None*, *input_domain=None*, *weight=None*, *inp_weight=None*, *description=None*, *include_defaults=False*, *await_domain=None*, ***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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Sub-dispatcher node id.

Return type str

See also:

add_data(), *add_func()*, *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', 'b'): ('E', 'E1')},
...                    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_func()*, *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
{}
```

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*

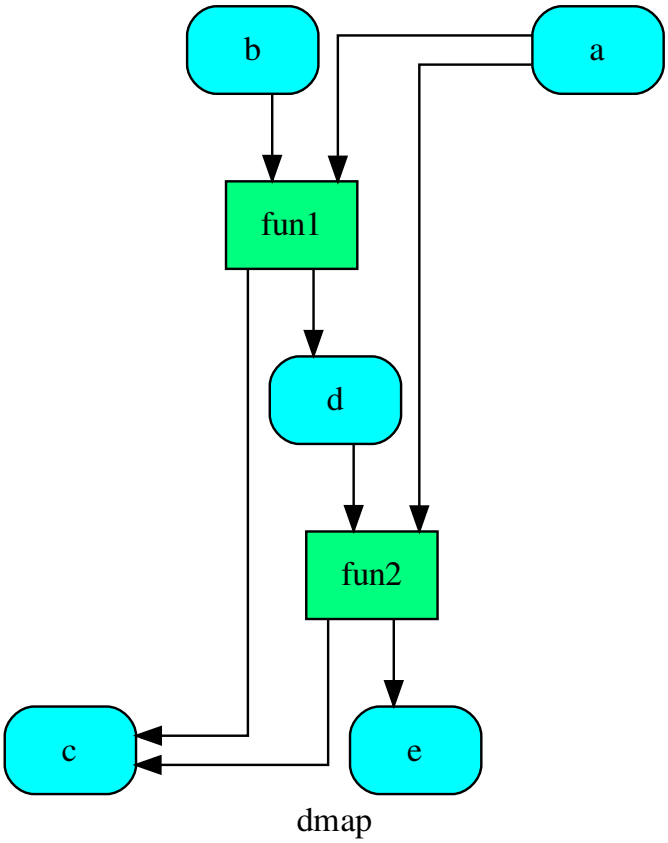
See also:

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.

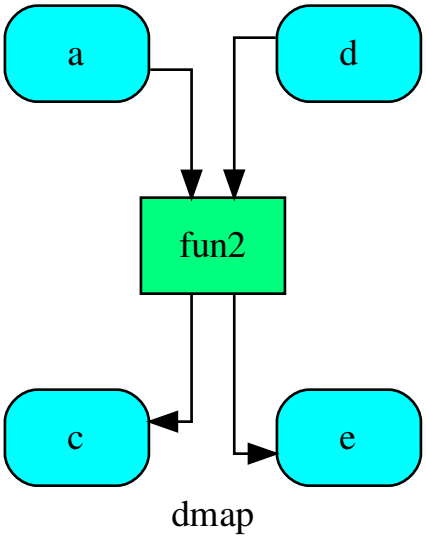
Example:

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*

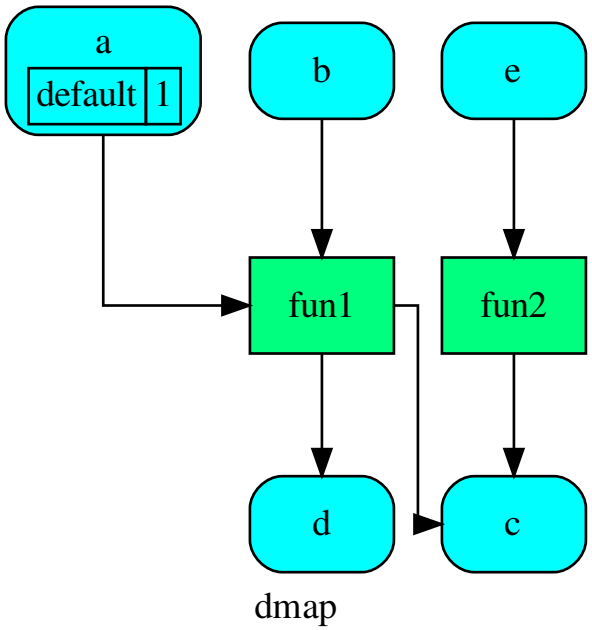
See also:

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:

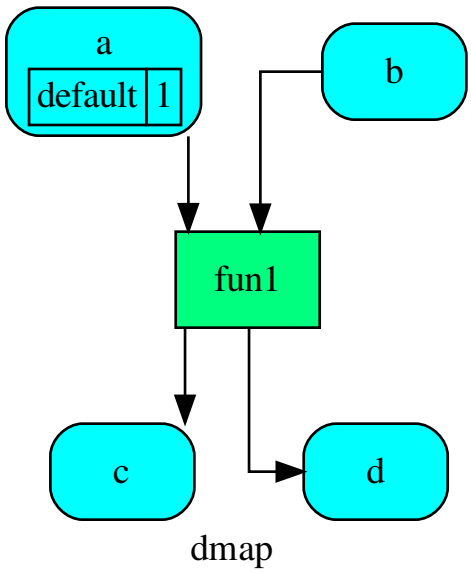


Dispatch with no calls in order to have a workflow:

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

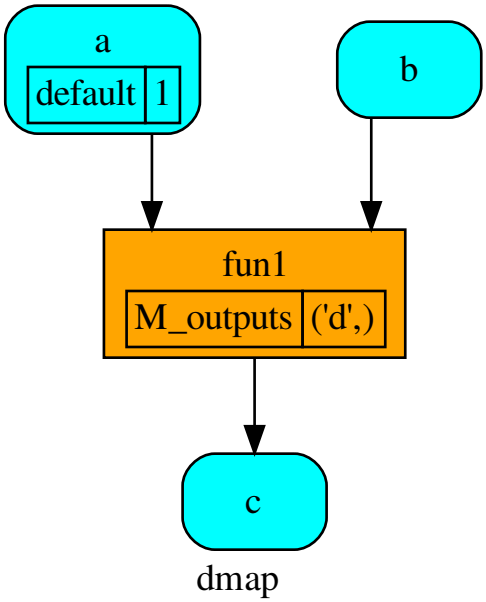
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> 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
```

blue(*memo=None*)

Constructs a BlueDispatcher out of the current object.

Parameters **memo** (*dict[T, schedula.utils.blue.Blueprint]*) – A dictionary to cache Blueprints.

Returns A BlueDispatcher of the current object.

Return type *schedula.utils.blue.BlueDispatcher*

extend(**blues*, *memo=None*)

Extends Dispatcher calling each deferred operation of given Blueprints.

Parameters

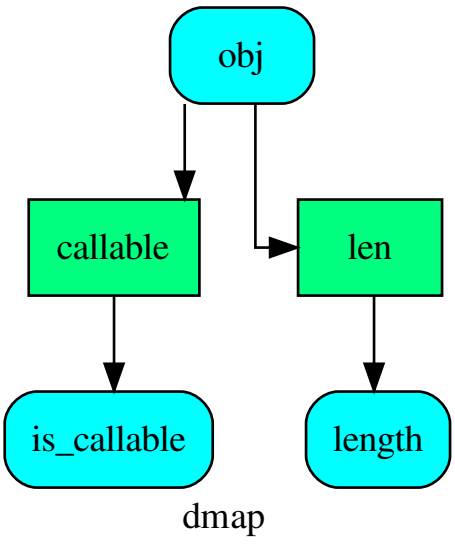
- **blues** (*Blueprint | schedula.dispatcher.Dispatcher*) – Blueprints or Dispatchers to extend deferred operations.
- **memo** (*dict[T, schedula.utils.blue.Blueprint|Dispatcher]*) – A dictionary to cache Blueprints and Dispatchers.

Returns Self.

Return type *Dispatcher*

Example:

```
>>> import schedula as sh
>>> dsp = sh.Dispatcher()
>>> dsp.add_func(callable, ['is_callable'])
'callable'
>>> blue = sh.BlueDispatcher().add_func(len, ['length'])
>>> dsp = sh.Dispatcher().extend(dsp, blue)
```



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*, *executor=False*, *sol_name=()*)

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:

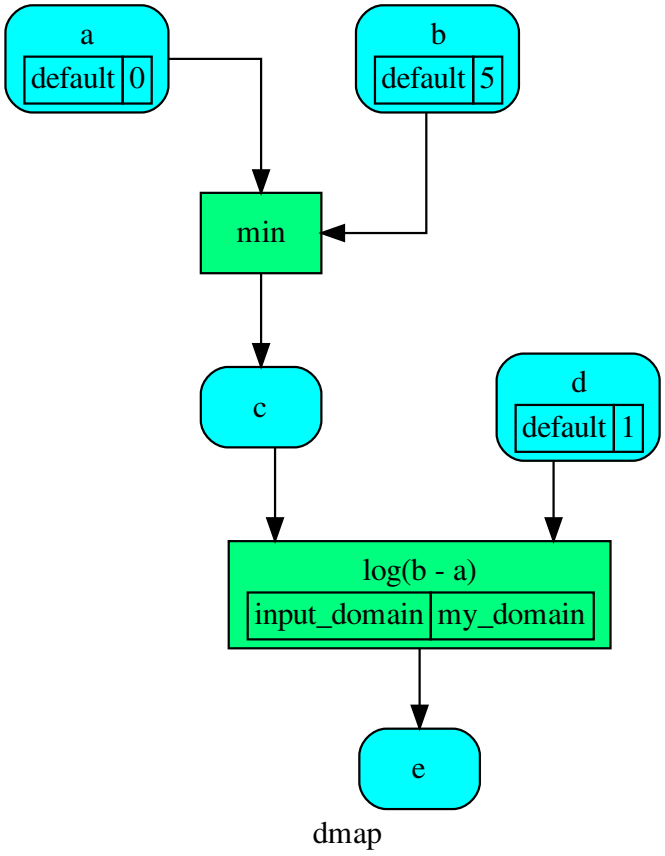
  shrink_dsp()
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **select_output_kw** (*dict, optional*) – Kwargs of selector function to select specific outputs.
- **_wait_in** (*dict, optional*) – Override wait inputs.
- **stopper** (*multiprocess.Event, optional*) – A semaphore to abort the dispatching.
- **executor** (*str, optional*) – A pool executor id to dispatch asynchronously or in parallel.
- **sol_name** (*tuple[str], optional*) – Solution name.

Returns Dictionary of estimated data node outputs.

Return type *schedula.utils.sol.Solution*

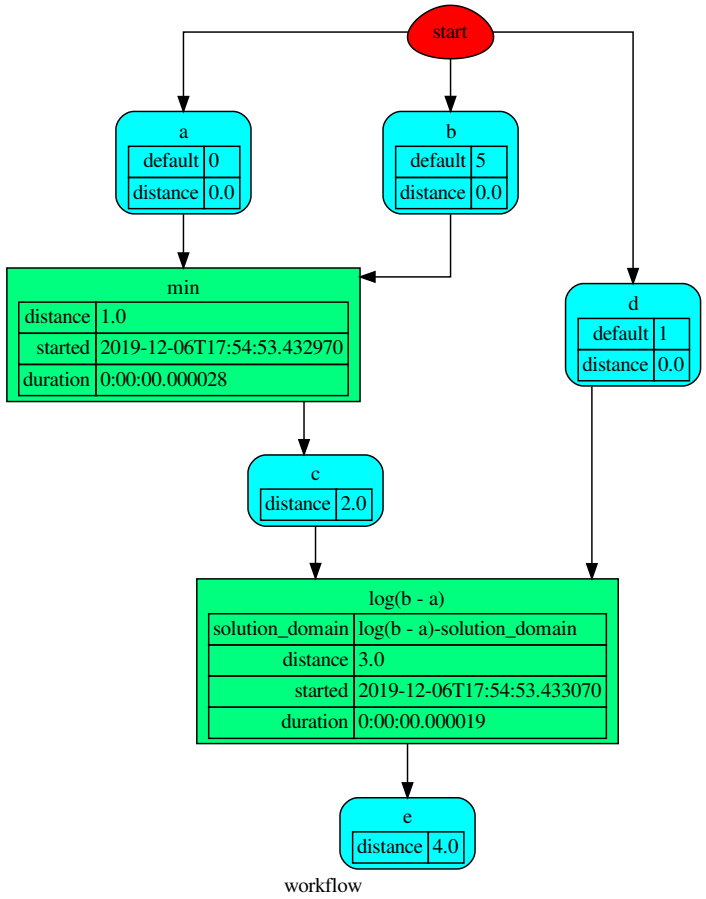
Example:

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



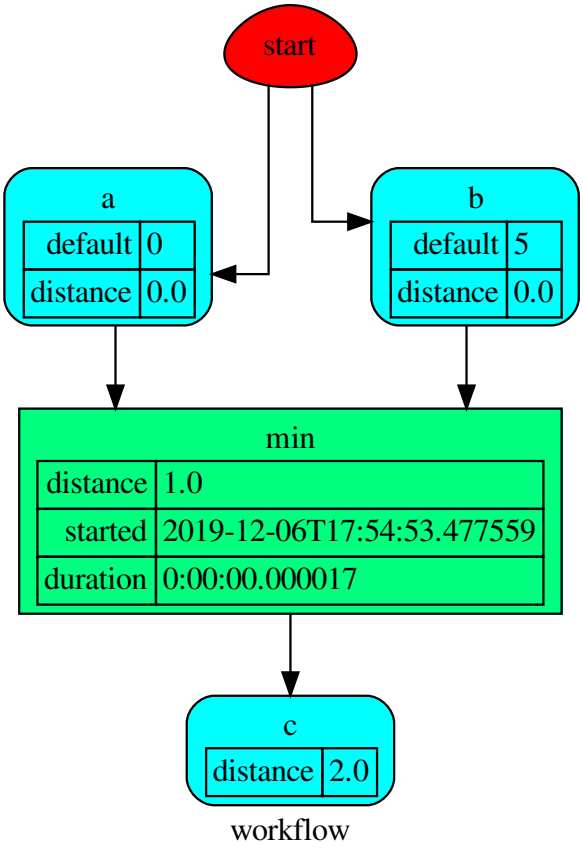
Dispatch without inputs. The default values are used as inputs:

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



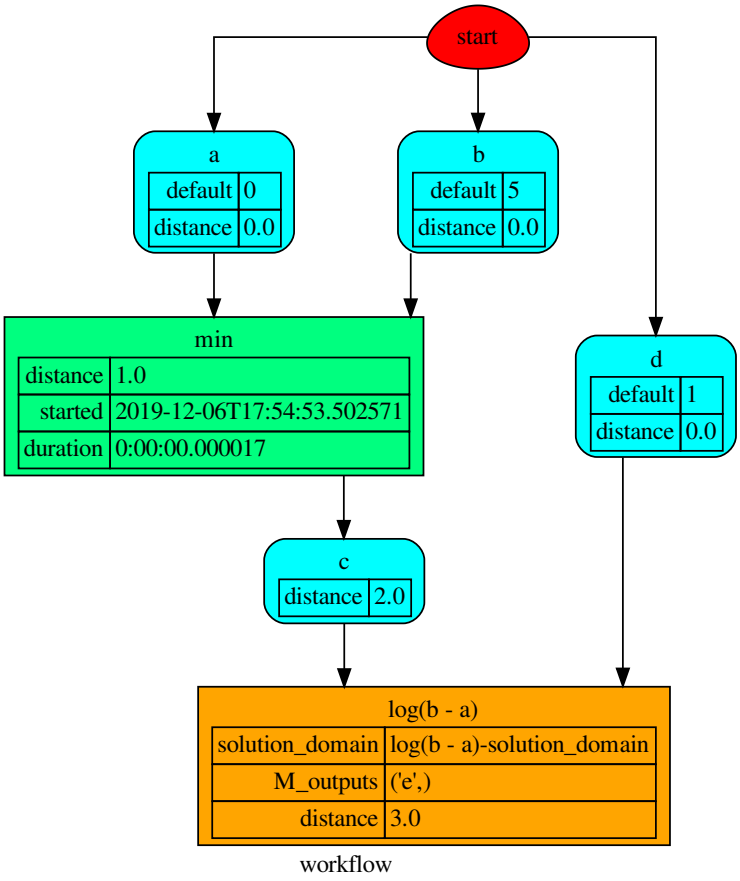
Dispatch until data node *c* is estimated:

```
>>> 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.

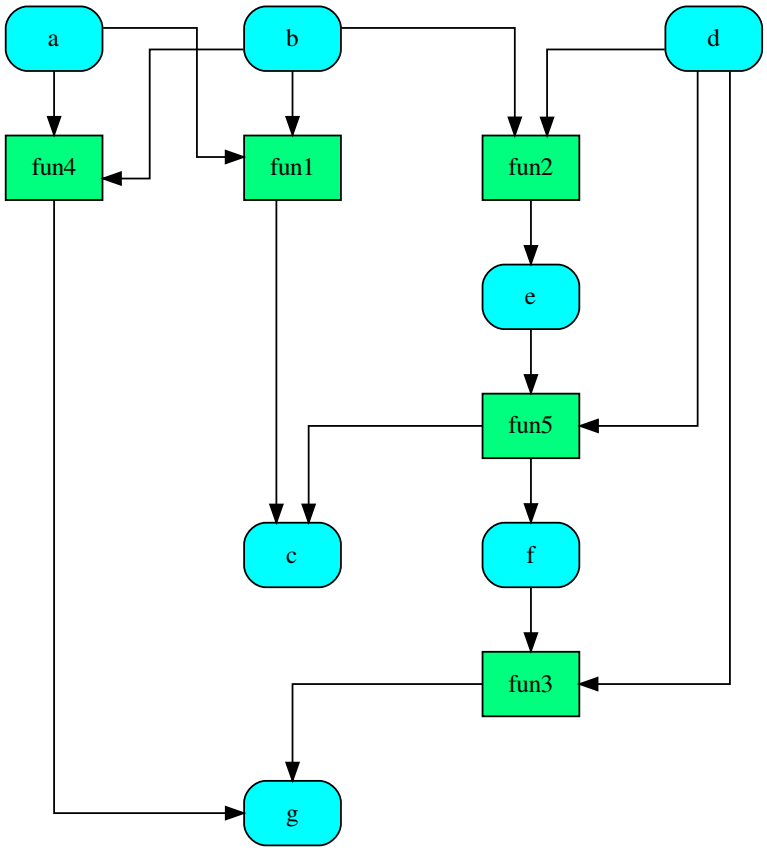
Return type *Dispatcher*

See also:

dispatch()

Example:

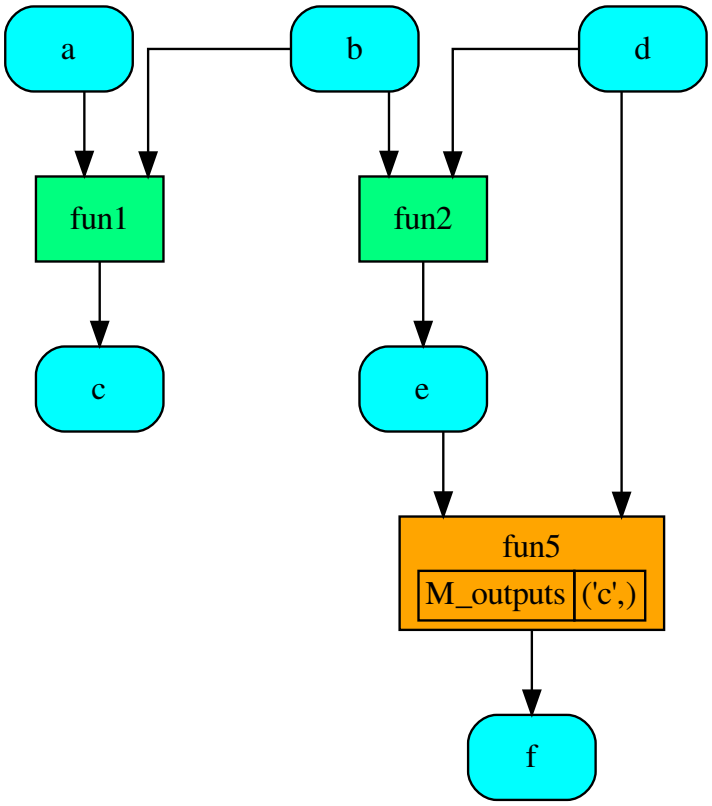
A dispatcher like this:



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

2.1.8.2 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 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.
asy	It contains functions to dispatch asynchronously and in parallel.
base	It provides a base class for dispatcher objects.
blue	It provides a Blueprint class to construct a Dispatcher and SubDispatch 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 provides a solution class for dispatch result.
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*.

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.

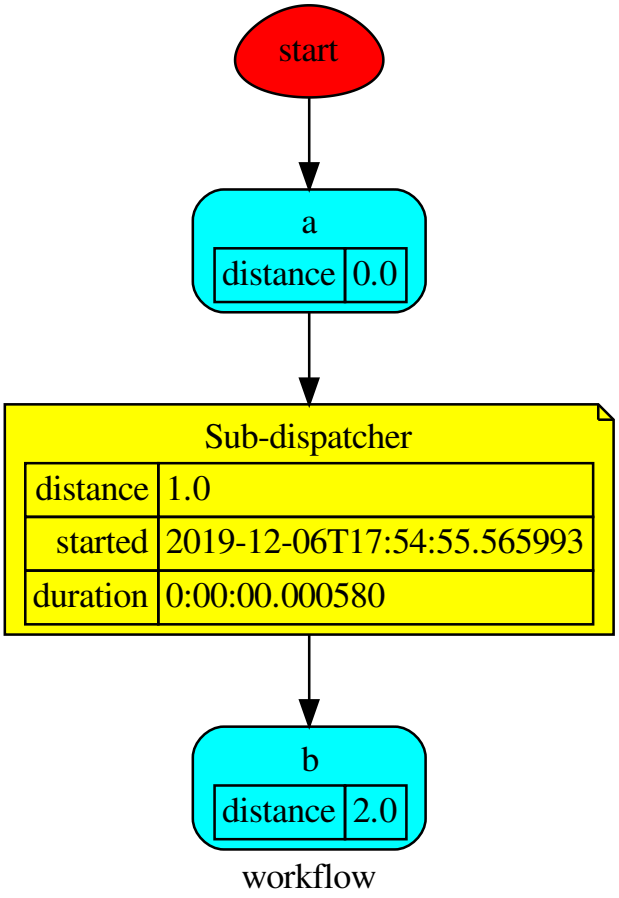
Returns A sub node of a dispatcher and its path.

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}})
...
```

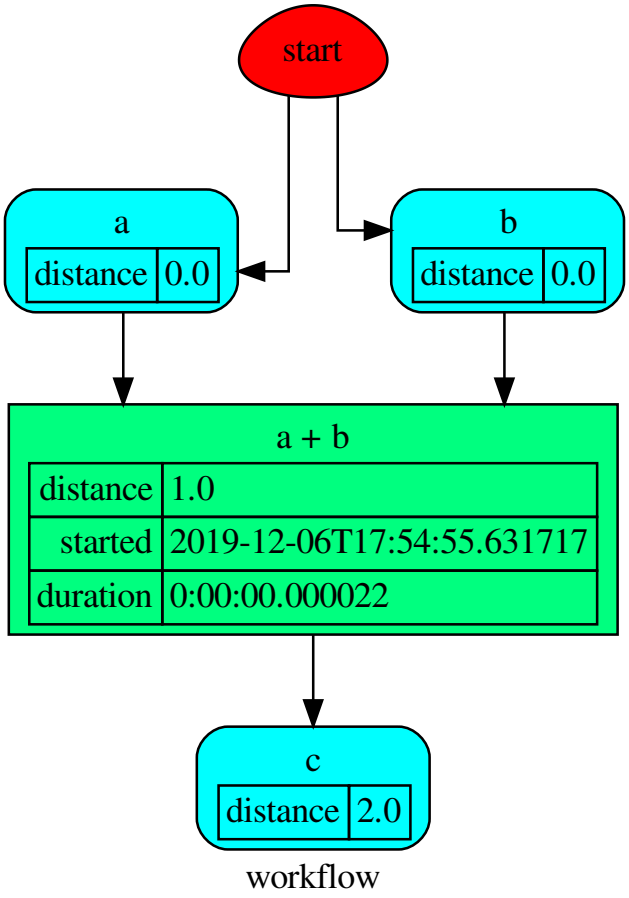


Get the sub node 'c' output or type:

```
>>> 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

Classes

DspPipe

DspPipe

class DspPipe

Methods

clear	
copy	
fromkeys	Create a new ordered dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last is false).
pop	value.
popitem	Remove and return a (key, value) pair from the dictionary.
setdefault	Insert key with a value of default if key is not in the dictionary.
update	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values	

clear

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

copy

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

fromkeys

DspPipe.**fromkeys**()
Create a new ordered dictionary with keys from iterable and values set to value.

get

DspPipe.**get**()
Return the value for key if key is in the dictionary, else default.

items

DspPipe.**items**() → a set-like object providing a view on D's items

keys

DspPipe.**keys**() → a set-like object providing a view on D's keys

move_to_end

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

Raise KeyError if the element does not exist.

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**()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault

DspPipe.**setdefault**()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update

DspPipe.**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

DspPipe.**values**() → an object providing a view on D's values

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

asy

It contains functions to dispatch asynchronously and in parallel.

Functions

async_process	Execute *func(*args)* in an asynchronous parallel process.
async_thread	Execute *sol._evaluate_node* in an asynchronous thread.
await_result	Return the result of a *Future* object.
register_executor	Register a new executor type.
shutdown_executor	Clean-up the resources associated with the Executor.
shutdown_executors	Clean-up the resources of all initialized executors.

async_process

async_process(*funcs*, **args*, *executor=False*, *sol=None*, *callback=None*, ***kw*)
Execute *func(*args)* in an asynchronous parallel process.

Parameters

- **funcs** (*list[callable]*) – Functions to be executed.
- **args** (*tuple*) – Arguments to be passed to first function call.
- **executor** (*str | bool*) – Pool executor to run the function.
- **sol** (schedula.utils.sol.Solution) – Parent solution.
- **callback** (*callable*) – Callback function to be called after all function execution.
- **kw** (*dict*) – Keywords to be passed to first function call.

Returns Functions result.

Return type object

async_thread

async_thread(*sol*, *args*, *node_attr*, *node_id*, **a*, ***kw*)
Execute *sol._evaluate_node* in an asynchronous thread.

Parameters

- **sol** (*schedula.utils.sol.Solution*) – Solution to be updated.
- **args** (*tuple*) – Arguments to be passed to node calls.
- **node_attr** (*dict*) – Dictionary of node attributes.
- **node_id** (*str*) – Data or function node id.
- **a** (*tuple*) – Extra args to invoke *sol._evaluate_node*.
- **kw** (*dict*) – Extra kwargs to invoke *sol._evaluate_node*.

Returns Function result.

Return type concurrent.futures.Future | AsyncList

await_result

await_result(*obj*, *timeout=None*)
Return the result of a *Future* object.

Parameters

- **obj** (*concurrent.futures.Future | object*) – Value object.
- **timeout** (*int*) – The number of seconds to wait for the result if the future isn't done. If None, then there is no limit on the wait time.

Returns Result.

Return type object

Example:

```
>>> from concurrent.futures import Future
>>> fut = Future()
>>> fut.set_result(3)
>>> await_result(fut), await_result(4)
(3, 4)
```

register_executor

register_executor(*name*, *init*)
Register a new executor type.

Parameters

- **name** (*str*) – Executor name.
- **init** (*callable*) – Function to initialize the executor.

shutdown_executor

shutdown_executor(*name*, *wait=True*)
Clean-up the resources associated with the Executor.

Parameters

- **name** (*str*) – Executor name.
- **wait** (*bool*) – If True then shutdown will not return until all running futures have finished executing and the resources used by the executor have been reclaimed.

Returns Shutdown pool executor.
:rtype:dict[concurrent.futures.Future,threading.Thread|multiprocess.Process]

shutdown_executors

shutdown_executors(*wait=True*)

Clean-up the resources of all initialized executors.

Parameters **wait** (*bool*) – If True then shutdown will not return until all running futures have finished executing and the resources used by the executors have been reclaimed.

Returns Shutdown pool executor.

Return type dict[str,dict]

Classes

AsyncList	
Executor	Base Executor
PoolExecutor	General PoolExecutor to dispatch asynchronously and in parallel.
ProcessExecutor	Multi Process Executor
ProcessPoolExecutor	Process Pool Executor
ThreadExecutor	Multi Thread Executor

AsyncList

class AsyncList(*, *future=None*, *n=1*)

Methods

__init__	Initialize self.
append	Append object to the end of the list.
clear	Remove all items from list.
copy	Return a shallow copy of the list.
count	Return number of occurrences of value.
extend	Extend list by appending elements from the iterable.
index	Return first index of value.
insert	Insert object before index.
pop	Remove and return item at index (default last).
remove	Remove first occurrence of value.
reverse	Reverse *IN PLACE*.
sort	Stable sort *IN PLACE*.

__init__

AsyncList.**__init__**(*, *future=None*, *n=1*)
Initialize self. See help(type(self)) for accurate signature.

append

AsyncList.**append**()
Append object to the end of the list.

clear

AsyncList.**clear**()
Remove all items from list.

copy

AsyncList.**copy**()
Return a shallow copy of the list.

count

AsyncList.**count**()
Return number of occurrences of value.

extend

AsyncList.**extend**()
Extend list by appending elements from the iterable.

index

AsyncList.**index**()
Return first index of value.

Raises ValueError if the value is not present.

insert

AsyncList.**insert**()
Insert object before index.

pop

AsyncList.**pop**()
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove

AsyncList.**remove**()
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse

AsyncList.**reverse**()
Reverse *IN PLACE*.

sort

AsyncList.**sort**()
Stable sort *IN PLACE*.

__init__(*, *future=None*, *n=1*)
Initialize self. See help(type(self)) for accurate signature.

Executor

class Executor
Base Executor

Methods

__init__	Initialize self.
shutdown	
submit	

__init__

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

shutdown

Executor.**shutdown**(*wait=True*)

submit

Executor.**submit**(*func*, **args*, ***kwargs*)

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

PoolExecutor

class PoolExecutor(*thread_executor*, *process_executor=None*, *parallel=None*)
General PoolExecutor to dispatch asynchronously and in parallel.

Methods

__init__	**param thread_executor**
process	
process_funcs	
shutdown	
thread	

__init__

PoolExecutor.**__init__**(*thread_executor*, *process_executor=None*, *parallel=None*)

Parameters

- **thread_executor** (ThreadExecutor) – Thread pool executor to dispatch asynchronously.
- **process_executor** (*ProcessExecutor | ProcessPoolExecutor*) – Process pool executor to execute in parallel the functions calls.
- **parallel** (*bool*) – Run *_process_funcs* in parallel.

process

PoolExecutor.**process**(*fn*, **args*, ***kwargs*)

process_funcs

PoolExecutor.**process_funcs**(*name*, *funcs*, **args*, ***kw*)

shutdown

PoolExecutor.**shutdown**(*wait=True*)

thread

PoolExecutor.**thread**(**args*, ***kwargs*)

__init__(*thread_executor*, *process_executor=None*, *parallel=None*)

Parameters

- **thread_executor** (ThreadExecutor) – Thread pool executor to dispatch asynchronously.
- **process_executor** (*ProcessExecutor | ProcessPoolExecutor*) – Process pool executor to execute in parallel the functions calls.
- **parallel** (*bool*) – Run *_process_funcs* in parallel.

ProcessExecutor

class ProcessExecutor

Multi Process Executor

Methods

__init__	Initialize self.
shutdown	
submit	

__init__

ProcessExecutor.**__init__**()

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

shutdown

ProcessExecutor.**shutdown**(*wait=True*)

submit

ProcessExecutor.**submit**(*func*, **args*, ***kwargs*)

__init__()

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

ProcessPoolExecutor

class ProcessPoolExecutor

Process Pool Executor

Methods

__init__	Initialize self.
shutdown	
submit	

__init__

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

shutdown

ProcessPoolExecutor.**shutdown**(*wait=True*)

submit

ProcessPoolExecutor.**submit**(*func*, **args*, ***kwargs*)

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

ThreadExecutor

class ThreadExecutor
Multi Thread Executor

Methods

__init__	Initialize self.
shutdown	
submit	

__init__

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

shutdown

ThreadExecutor.**shutdown**(*wait=True*)

submit

ThreadExecutor.**submit**(*func*, **args*, ***kwargs*)

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

base

It provides a base class for dispatcher objects.

Classes

Base	Base class for dispatcher objects.

Base

class Base
Base class for dispatcher objects.

Methods

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.

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.

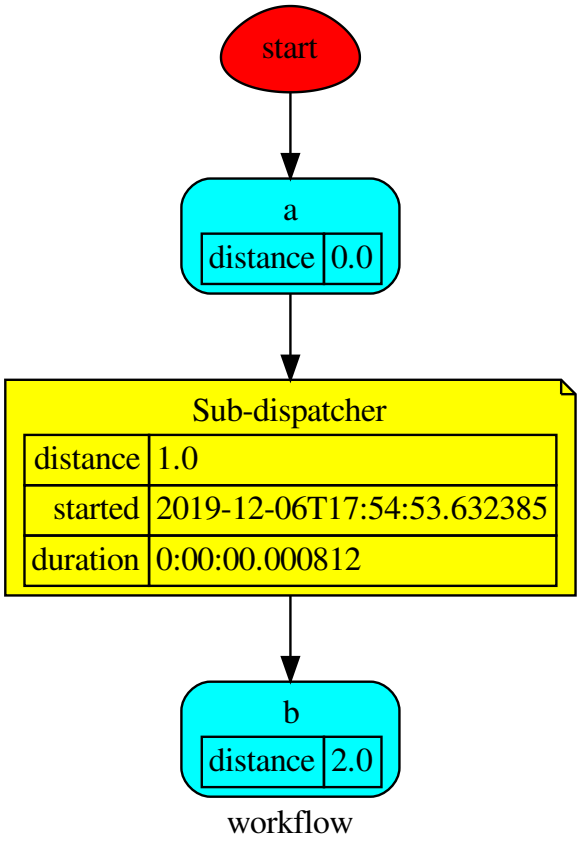
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

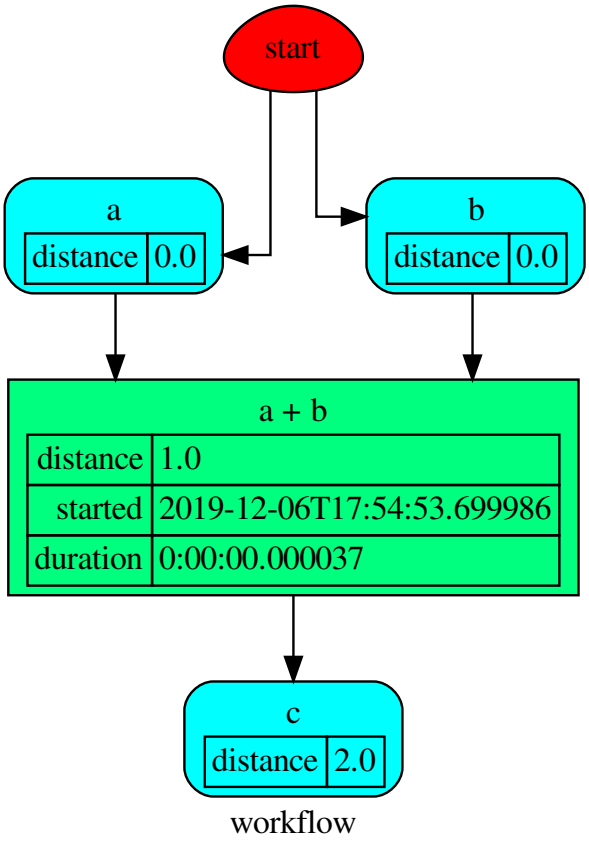
Example:



Get the sub node output:

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

```
>>> sub_dsp, sub_dsp_id = dsp.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.

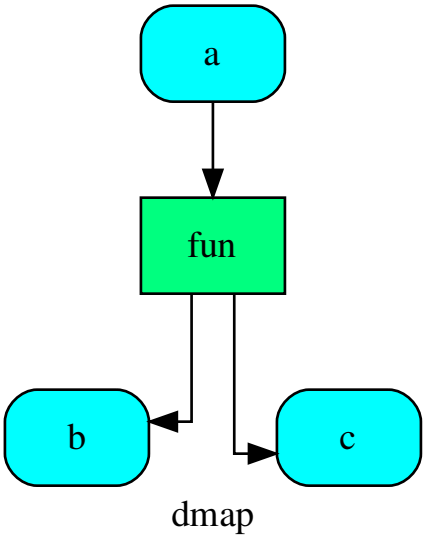
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment 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[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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

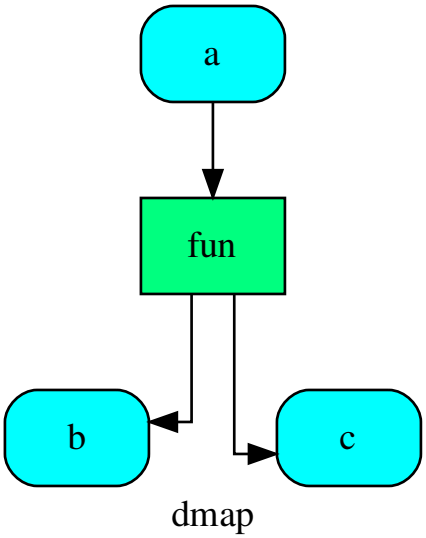
Returns A WebMap.

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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

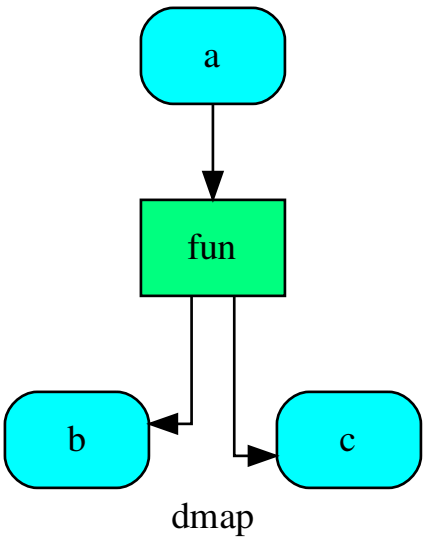
Returns A WebMap.

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.

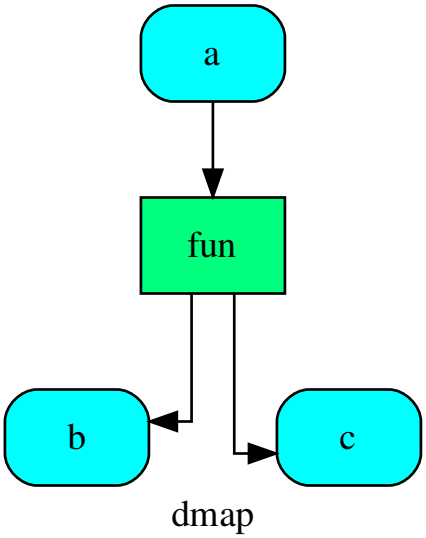
- **body** (*dict, optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[Site], optional*) – A set of *Site* to maintain alive the back-end server.
- **index** (*bool, optional*) – Add the site index 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.

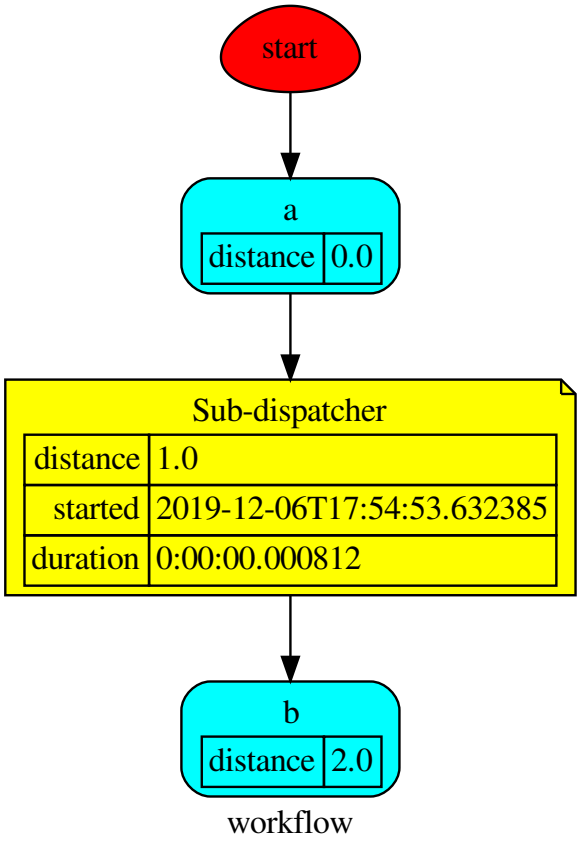
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

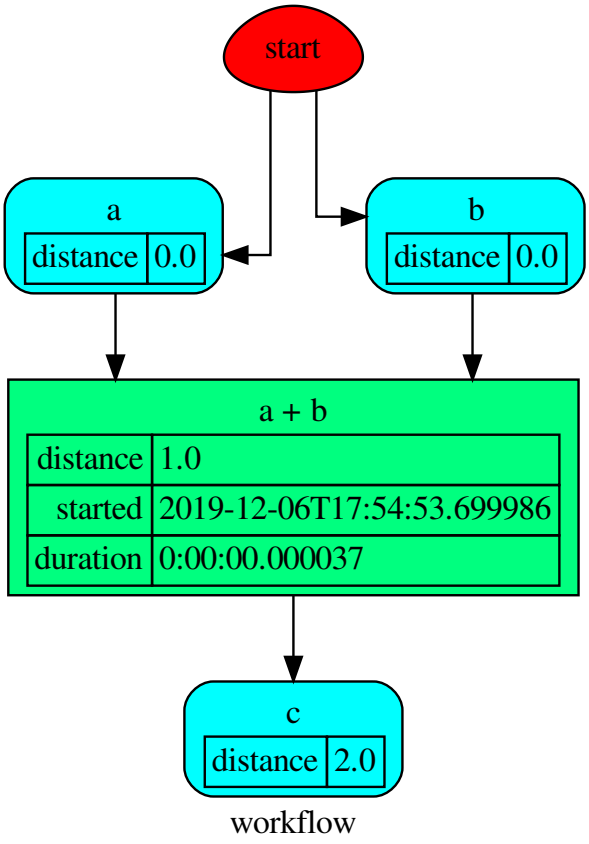
Example:



Get the sub node output:

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

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



blue

It provides a Blueprint class to construct a Dispatcher and SubDispatch objects.

Classes

BlueDispatcher	Blueprint object is a blueprint of how to construct or extend a Dispatcher.
Blueprint	Base Blueprint class.

BlueDispatcher

class BlueDispatcher(*dmap=None*, *name=''*, *default_values=None*, *raises=False*, *description=''*, *executor=None*)

Blueprint object is a blueprint of how to construct or extend a Dispatcher.

Example:

Create a BlueDispatcher:

```
>>> import schedula as sh
>>> blue = sh.BlueDispatcher(name='Dispatcher')
```

Add data/function/dispatcher nodes to the dispatcher map as usual:

```
>>> blue.add_data(data_id='a', default_value=3)
<schedula.utils.blue.BlueDispatcher object at ...>
>>> @sh.add_function(blue, True, True, outputs=['c'])
... def diff_function(a, b=2):
...     return b - a
...
>>> blue.add_function(function=max, inputs=['c', 'd'], outputs=['e'])
<schedula.utils.blue.BlueDispatcher object at ...>
>>> from math import log
>>> sub_blue = sh.BlueDispatcher(name='Sub-Dispatcher')
>>> sub_blue.add_data(data_id='a', default_value=2).add_function(
...    function=log, inputs=['a'], outputs=['b']
... )
<schedula.utils.blue.BlueDispatcher object at ...>
>>> blue.add_dispatcher(sub_blue, ('a',), {'b': 'f'})
<schedula.utils.blue.BlueDispatcher object at ...>
```

You can set the default values as usual:

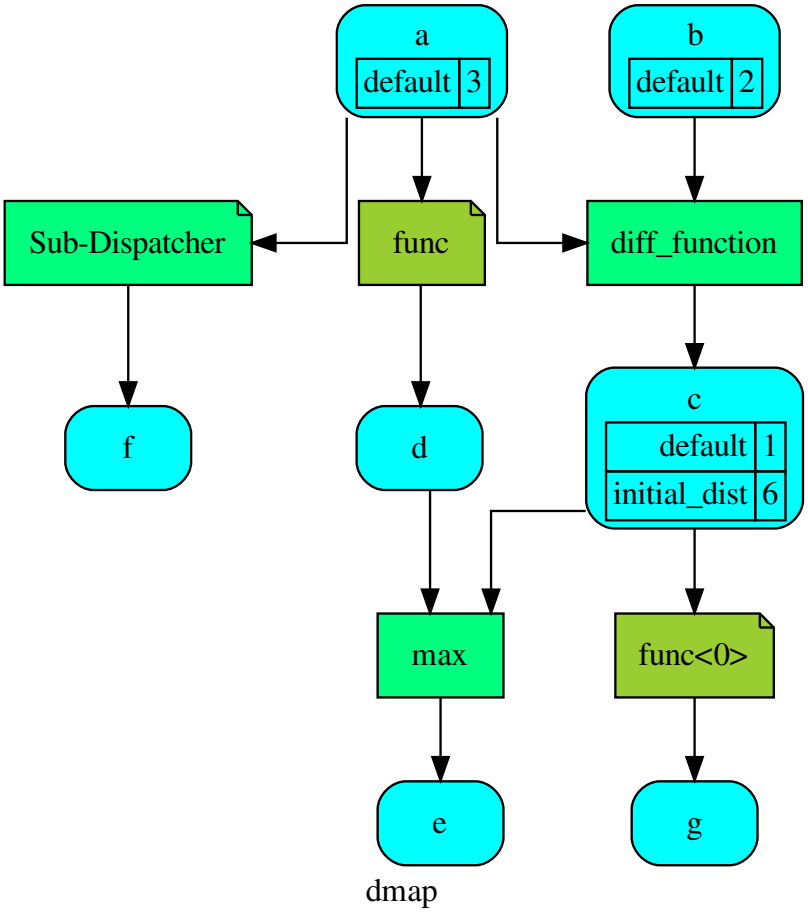
```
>>> blue.set_default_value(data_id='c', value=1, initial_dist=6)
<schedula.utils.blue.BlueDispatcher object at ...>
```

You can also create a *Blueprint* out of *SubDispatchFunction* and add it to the *Dispatcher* as follow:

```
>>> func = sh.SubDispatchFunction(sub_blue, 'func', ['a'], ['b'])
>>> blue.add_from_lists(fun_list=[
...    dict(function=func, inputs=['a'], outputs=['d']),
...    dict(function=func, inputs=['c'], outputs=['g']),
... ])
<schedula.utils.blue.BlueDispatcher object at ...>
```

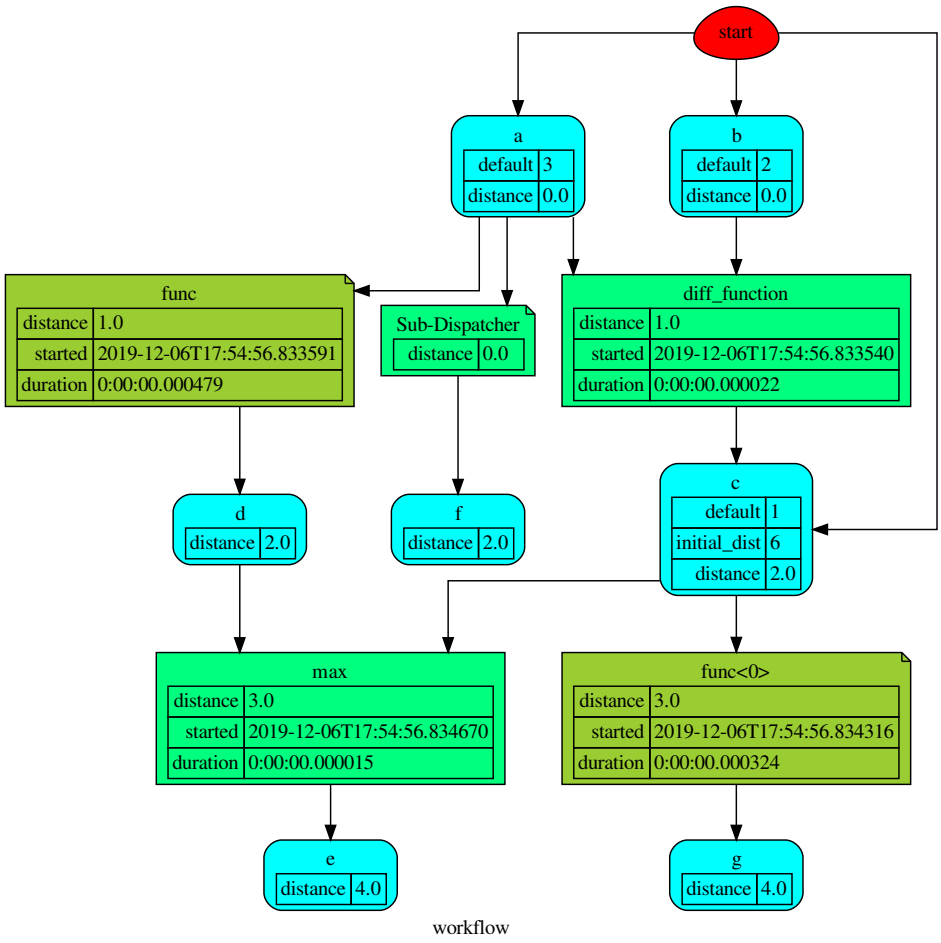
Finally you can create the dispatcher object using the method *new*:

```
>>> dsp = blue.register(memo={}); dsp
<schedula.dispatcher.Dispatcher object at ...>
```



Or dispatch, calling the Blueprint object:

```
>>> sol = blue({'a': 1}); sol
Solution([('a', 1), ('b', 2), ('c', 1), ('d', 0.0),
          ('f', 0.0), ('e', 1), ('g', 0.0)])
```



workflow

Methods

__init__	Initialize self.
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_func	Add a single function node to dispatcher.
add_function	Add a single function node to dispatcher.
extend	Extends deferred operations calling each operation of given Blueprints.
register	Creates a *Blueprint.cls* and calls each deferred operation.
set_default_value	Set the default value of a data node in the dispatcher.

__init__

BlueDispatcher.**__init__**(*dmap=None*, *name=''*, *default_values=None*, *raises=False*, *description=''*, *executor=None*)

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

add_data

BlueDispatcher.**add_data**(*data_id=None*, *default_value=empty*, *initial_dist=0.0*, *wait_inputs=False*, *wildcard=None*, *function=None*, *callback=None*, *description=None*, *filters=None*, *await_result=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.
- **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.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits data results before assigning them to the solution. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Self.

Return type *BlueDispatcher*

add_dispatcher

BlueDispatcher.**add_dispatcher**(*dsp*, *inputs*, *outputs*, *dsp_id=None*, *input_domain=None*, *weight=None*, *inp_weight=None*, *description=None*, *include_defaults=False*, *await_domain=None*, ***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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Self.

Return type *BlueDispatcher*

add_from_lists

BlueDispatcher.**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 Self.

Return type *BlueDispatcher*

add_func

BlueDispatcher.**add_func**(*function*, *outputs=None*, *weight=None*, *inputs_kwargs=False*, *inputs_defaults=False*, *filters=None*, *input_domain=None*, *await_domain=None*, *await_result=None*, *inp_weight=None*, *out_weight=None*, *description=None*, *inputs=None*, *function_id=None*, ***kwargs*)

Add a single function node to dispatcher.

Parameters

- **inputs_kwargs** (*bool*) – Do you want to include kwargs as inputs?
- **inputs_defaults** (*bool*) – Do you want to set default values?
- **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. If None it will take parameters names from function signature.
- **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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Self.

Return type *BlueDispatcher*

add_function

BlueDispatcher.**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*, *await_domain=None*, *await_result=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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

extend

BlueDispatcher.**extend**(**blues*, *memo=None*)

Extends deferred operations calling each operation of given Blueprints.

Parameters

- **blues** (*Blueprint | schedula.dispatcher.Dispatcher*) – Blueprints or Dispatchers to extend deferred operations.
- **memo** (*dict[T,Blueprint]*) – A dictionary to cache Blueprints.

Returns Self.

Return type *Blueprint*

Example:

```
>>> import schedula as sh
>>> blue = sh.BlueDispatcher()
>>> blue.extend(
...     BlueDispatcher().add_func(len, ['length']),
...     BlueDispatcher().add_func(callable, ['is_callable'])
... )
<schedula.utils.blue.BlueDispatcher object at ...>
```

register

BlueDispatcher.**register**(*obj=None*, *memo=None*)

Creates a *Blueprint.cls* and calls each deferred operation.

Parameters

- **obj** (*object*) – The initialized object with which to call all deferred operations.

- **memo** (*dict[Blueprint, T]*) – A dictionary to cache registered Blueprints.

Returns The initialized object.

Return type Blueprint.cls | Blueprint

Example:

```
>>> import schedula as sh
>>> blue = sh.BlueDispatcher().add_func(len, ['lenght'])
>>> blue.register()
<schedula.dispatcher.Dispatcher object at ...>
```

set_default_value

BlueDispatcher.**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.

Returns Self.

Return type *BlueDispatcher*

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

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

add_data(*data_id=None*, *default_value=empty*, *initial_dist=0.0*, *wait_inputs=False*, *wildcard=None*, *function=None*, *callback=None*, *description=None*, *filters=None*, *await_result=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.
- **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.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits data results before assigning them to the solution. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Self.

Return type *BlueDispatcher*

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*, *await_domain=None*, *await_result=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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

add_func(*function*, *outputs=None*, *weight=None*, *inputs_kwargs=False*, *inputs_defaults=False*, *filters=None*, *input_domain=None*, *await_domain=None*, *await_result=None*, *inp_weight=None*, *out_weight=None*, *description=None*, *inputs=None*, *function_id=None*, ***kwargs*)

Add a single function node to dispatcher.

Parameters

- **inputs_kwargs** (*bool*) – Do you want to include kwargs as inputs?
- **inputs_defaults** (*bool*) – Do you want to set default values?
- **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. If None it will take parameters names from function signature.
- **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.

- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool|int|float, optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments, optional*) – Set additional node attributes using key=value.

Returns Self.

Return type *BlueDispatcher*

add_dispatcher(*dsp*, *inputs*, *outputs*, *dsp_id=None*, *input_domain=None*, *weight=None*, *inp_weight=None*, *description=None*, *include_defaults=False*, *await_domain=None*, ***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.
- **await_domain** (*bool|int|float, optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is

defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.

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

Returns Self.

Return type *BlueDispatcher*

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 Self.

Return type *BlueDispatcher*

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.

Returns Self.

Return type *BlueDispatcher*

Blueprint

class Blueprint(**args*, ***kwargs*)
Base Blueprint class.

Methods

__init__	Initialize self.
extend	Extends deferred operations calling each operation of given Blueprints.

Continued on next page

Table 21 – continued from previous page

register	Creates a *Blueprint.cls* and calls each deferred operation.

__init__

Blueprint.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

extend

Blueprint.**extend**(**blues*, *memo=None*)
Extends deferred operations calling each operation of given Blueprints.

Parameters

- **blues** (*Blueprint | schedula.dispatcher.Dispatcher*) – Blueprints or Dispatchers to extend deferred operations.
- **memo** (*dict[T,Blueprint]*) – A dictionary to cache Blueprints.

Returns Self.

Return type *Blueprint*

Example:

```
>>> import schedula as sh
>>> blue = sh.BlueDispatcher()
>>> blue.extend(
...     BlueDispatcher().add_func(len, ['length']),
...     BlueDispatcher().add_func(callable, ['is_callable'])
... )
<schedula.utils.blue.BlueDispatcher object at ...>
```

register

Blueprint.**register**(*obj=None*, *memo=None*)
Creates a *Blueprint.cls* and calls each deferred operation.

Parameters

- **obj** (*object*) – The initialized object with which to call all deferred operations.
- **memo** (*dict[Blueprint,T]*) – A dictionary to cache registered Blueprints.

Returns The initialized object.

Return type Blueprint.cls | Blueprint

Example:

```
>>> import schedula as sh
>>> blue = sh.BlueDispatcher().add_func(len, ['lenght'])
>>> blue.register()
<schedula.dispatcher.Dispatcher object at ...>
```

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

cls
alias of *schedula.dispatcher.Dispatcher*

register(*obj=None*, *memo=None*)
Creates a *Blueprint.cls* and calls each deferred operation.

Parameters

- **obj** (*object*) – The initialized object with which to call all deferred operations.
- **memo** (*dict[Blueprint, T]*) – A dictionary to cache registered Blueprints.

Returns The initialized object.

Return type Blueprint.cls | Blueprint

Example:

```
>>> import schedula as sh
>>> blue = sh.BlueDispatcher().add_func(len, ['lenght'])
>>> blue.register()
<schedula.dispatcher.Dispatcher object at ...>
```

extend(**blues*, *memo=None*)
Extends deferred operations calling each operation of given Blueprints.

Parameters

- **blues** (*Blueprint | schedula.dispatcher.Dispatcher*) – Blueprints or Dispatchers to extend deferred operations.
- **memo** (*dict[T, Blueprint]*) – A dictionary to cache Blueprints.

Returns Self.

Return type *Blueprint*

Example:

```
>>> import schedula as sh
>>> blue = sh.BlueDispatcher()
>>> blue.extend(
...     BlueDispatcher().add_func(len, ['length']),
...     BlueDispatcher().add_func(callable, ['is_callable'])
... )
<schedula.utils.blue.BlueDispatcher object at ...>
```

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_func()*, *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_func()*, *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.

Sub-Modules:

nodes	It provides docutils nodes to plot dispatcher map and workflow.

nodes

It provides docutils nodes to plot dispatcher map and workflow.

Functions

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

autoplot_callback

autoplot_callback(*res*)

autoplot_function

autoplot_function(*kwargs*)

basic_app

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

before_request

before_request(*mute*)

cached_view

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

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*, *extra=None*)

uncpath

uncpath(*p*)

update_filenames

update_filenames(*node*, *filenames*)

valid_filename

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

Classes

FolderNode
Site
SiteFolder
SiteIndex
SiteMap
SiteNode

FolderNode

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

Methods

__init__	Initialize self.
dot	
href	
items	
parent_ref	
render_funcs	
render_size	
style	
yield_attr	

__init__

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

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

dot

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

href

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

items

FolderNode.**items**()

parent_ref

FolderNode.**parent_ref**(*context*, *node_id*, *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*)
Initialize self. See help(type(self)) for accurate signature.

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', 'await_resu**

node_function

FolderNode.**node_function = ('-', '.tooltip', 'await_domain', 'await_result', '+input_d**

node_map

FolderNode.**node_map = {'': ('dot', 'table'), '!': ('dot', 'table'), '*': ('link',),**

node_styles

FolderNode.**node_styles = {'error': {empty: {'fillcolor': 'gray', 'label': 'empty',**

re_node

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

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*, *run_options=None*, ***kwargs*)

Methods

__init__	Initialize self.
app	
get_port	
run	

Continued on next page

Table 28 – continued from previous page

shutdown_site
wait_server

__init__

Site.**__init__**(*sitemap*, *host='localhost'*, *port=0*, *delay=0.1*, *until=30*, *run_options=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

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*, *run_options=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

is_running
url

is_running

Site.**is_running**

url

Site.**url**

SiteFolder

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

Methods

__init__	Initialize self.
dot	
view	

__init__

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

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

dot

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

view

SiteFolder.**view**(*filepath*, *context=None*)

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

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

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 = {'body': {'splines': 'ortho', 'style': 'filled'}, 'edge_attr':
```

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__	Initialize self.
legend	
render	
view	

__init__

SiteIndex.**__init__**(*sitemap*, *node_id='index'*)
Initialize self. See help(type(self)) for accurate signature.

legend

static SiteIndex.**legend**()

render

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

view

SiteIndex.**view**(*filepath*, **args*, ***kwargs*)

__init__(*sitemap*, *node_id='index'*)
Initialize self. See help(type(self)) for accurate signature.

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__	Initialize self.
add_items	
app	
clear	
copy	
fromkeys	Create a new ordered dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
get_dsp_from	
get_sol_from	
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last is false).
pop	value.
popitem	Remove and return a (key, value) pair from the dictionary.
render	
rules	
setdefault	Insert key with a value of default if key is not in the dictionary.
site	
update	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values	

__init__

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

add_items

SiteMap.**add_items**(*item*, *workflow=False*, *depth=-1*, *folder=None*, ***options*)

app

SiteMap.**app**(*root_path=None*, *depth=-1*, *index=True*, *mute=True*, ***kw*)

clear

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

copy

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

fromkeys

SiteMap.**fromkeys**()
Create a new ordered dictionary with keys from iterable and values set to value.

get

SiteMap.**get**()
Return the value for key if key is in the dictionary, else default.

get_dsp_from

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

get_sol_from

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

items

SiteMap.**items**() → a set-like object providing a view on D's items

keys

SiteMap.**keys**() → a set-like object providing a view on D's keys

move_to_end

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

Raise KeyError if the element does not exist.

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**()
Remove and return a (key, value) pair from the dictionary.

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*)

rules

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

setdefault

SiteMap.**setdefault**()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

site

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

update

SiteMap.**update**(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

SiteMap.**values**() → an object providing a view on D's values

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

Attributes

include_folders_as_filenames
nodes
options

include_folders_as_filenames

SiteMap.**include_folders_as_filenames = True**

nodes

SiteMap.**nodes**

options

SiteMap.**options = {'digraph', 'edge_data', 'max_lines', 'max_width', 'node_data', 'nod**

SiteNode

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

Methods

__init__	Initialize self.
render	
view	

__init__

SiteNode.**__init__**(*folder*, *node_id*, *item*, *obj*, *dsp_node_id*)
Initialize self. See help(type(self)) for accurate signature.

render

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

view

SiteNode.**view**(*filepath*, **args*, ***kwargs*)

__init__(*folder*, *node_id*, *item*, *obj*, *dsp_node_id*)
Initialize self. See help(type(self)) for accurate signature.

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	Return the parent function of a wrapped function (wrapped with `functools.partial` and *add_args*).
replicate_value	Replicates *n* times the input value.
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.Dispatcher.add_function‘.

Returns Decorator.

Return type callable

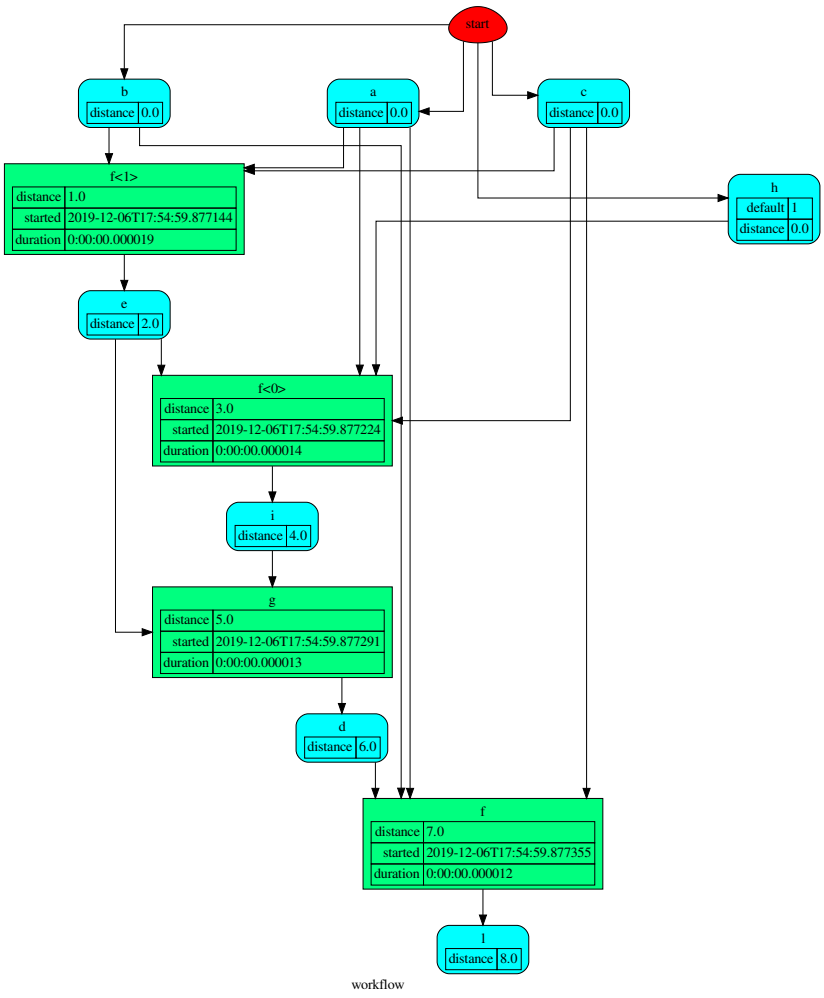
Example:

```
>>> import schedula as sh
>>> dsp = sh.Dispatcher(name='Dispatcher')
>>> @sh.add_function(dsp, outputs=['e'])
... @sh.add_function(dsp, False, True, outputs=['i'], inputs='ecah')
... @sh.add_function(dsp, True, outputs=['l'])
... def f(a, b, c, d=1):
...     return (a + b) - c + d
```

(continues on next page)

(continued from previous page)

```
>>> @sh.add_function(dsp, True, outputs=['d'])
... def g(e, i, *args, d=0):
...     return e + i + d
>>> sol = dsp({'a': 1, 'b': 2, 'c': 3}); sol
Solution([('a', 1), ('b', 2), ('c', 3), ('h', 1), ('e', 1), ('i', 4),
          ('d', 5), ('l', 5)])
```



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 (b)
>>> sorted(kk_dict({'a': 0, 'b': 1}, **{'b': 2, 'a': 3}).items())
Traceback (most recent call last):
 ...
ValueError: keyword argument repeated (a, b)
```

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*)

Return the parent function of a wrapped function (wrapped with `functools.partial` and *add_args*).

Parameters

- **func** (*callable*) – Wrapped function.
- **input_id** (*int*) – Index of the first input of the wrapped function.

Returns Parent function.

Return type callable

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

DispatchPipe	It converts a *Dispatcher* into a function.
NoSub	Class for avoiding to add a sub solution to the workflow.

Continued on next page

Table 39 – continued from previous page

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).
inf	Class to model infinite numbers for workflow distance.

DispatchPipe

class DispatchPipe(*dsp*, *function_id=None*, *inputs=None*, *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

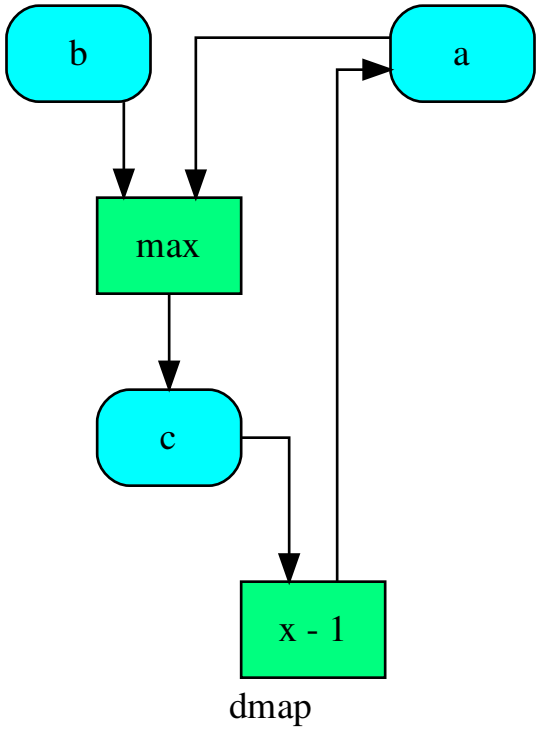
Note: This wrapper is not thread safe, because it overwrite the solution.

See also:

dispatch(), *shrink_dsp()*

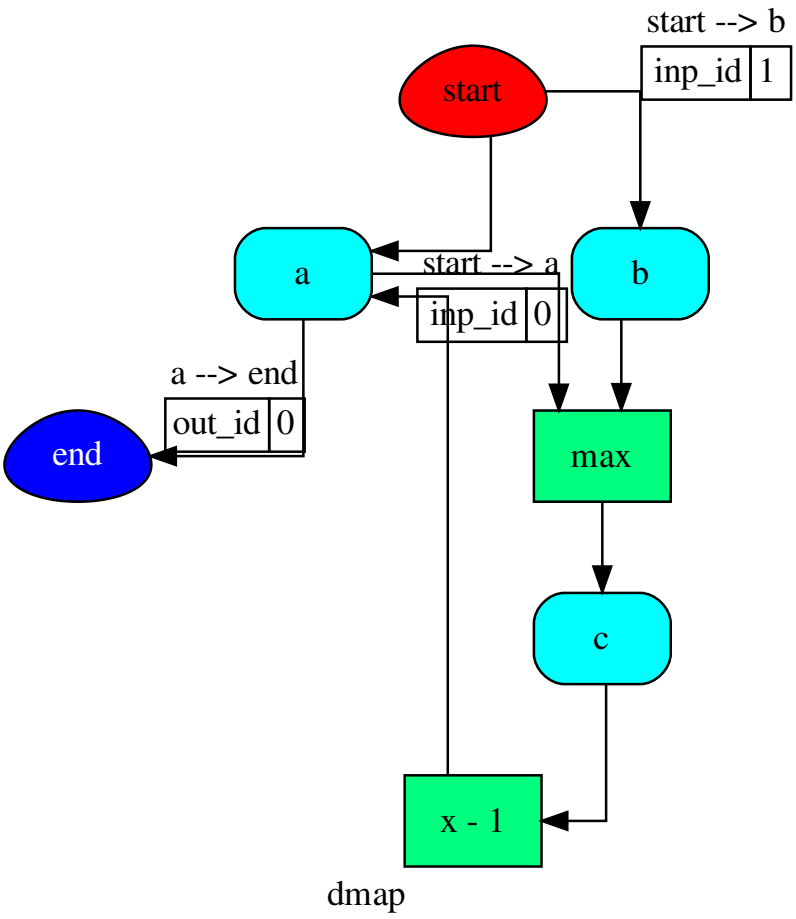
Example:

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



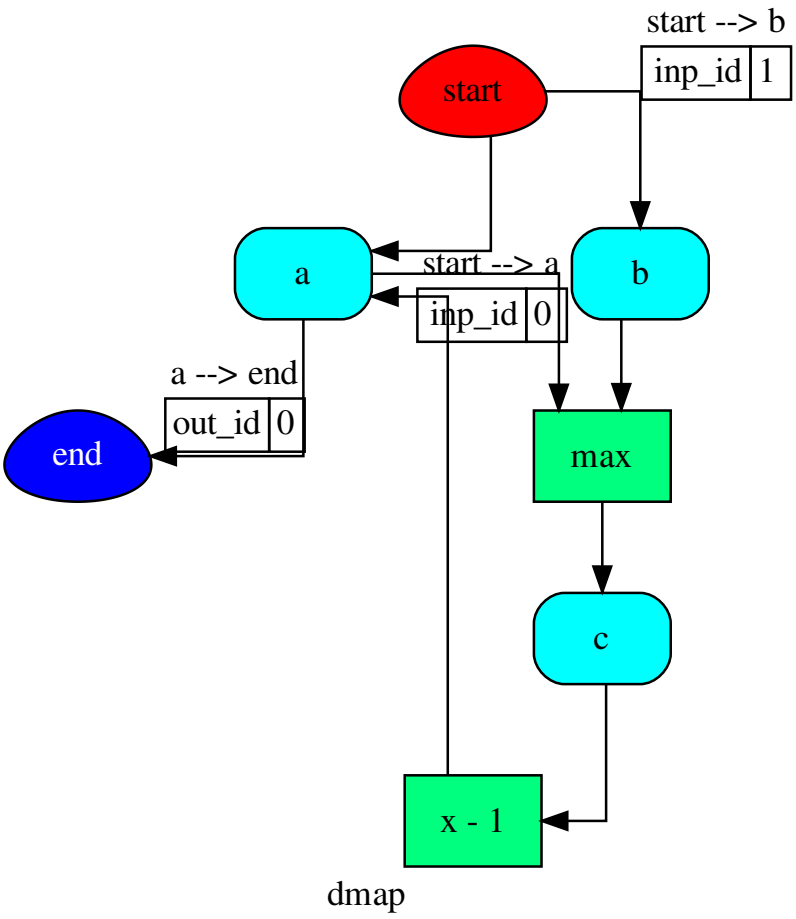
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```
>>> 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.
`blue`	Constructs a Blueprint out of the current object.
`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__

DispatchPipe.**__init__**(*dsp*, *function_id=None*, *inputs=None*, *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.

blue

DispatchPipe.**blue**(*memo=None*)

Constructs a Blueprint out of the current object.

Parameters **memo** (*dict[T,schedula.utils.blue.Blueprint]*) – A dictionary to cache Blueprints.

Returns A Blueprint of the current object.

Return type *schedula.utils.blue.Blueprint*

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.

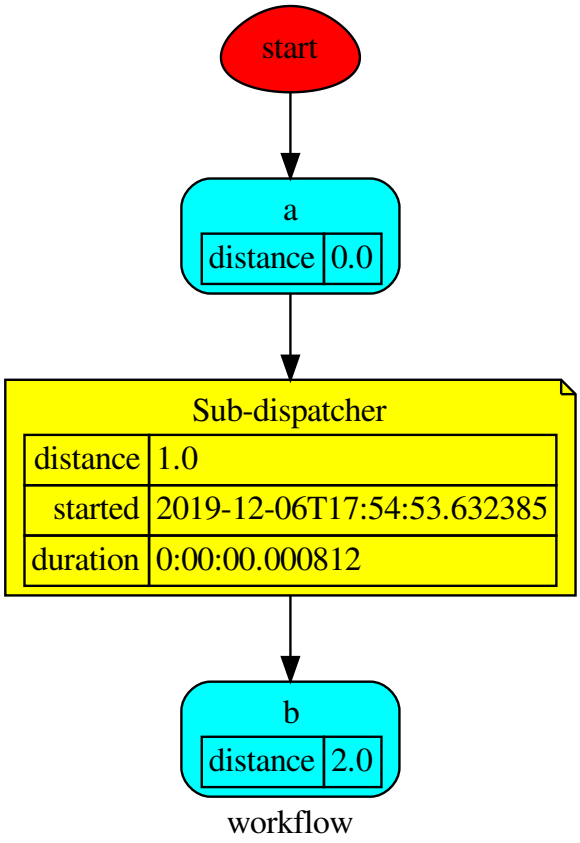
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

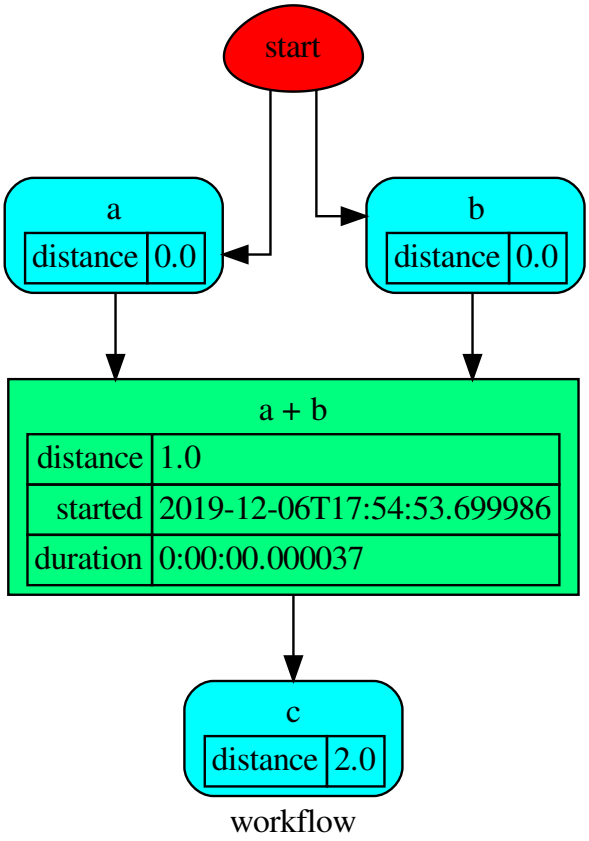
Example:



Get the sub node output:

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

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



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.

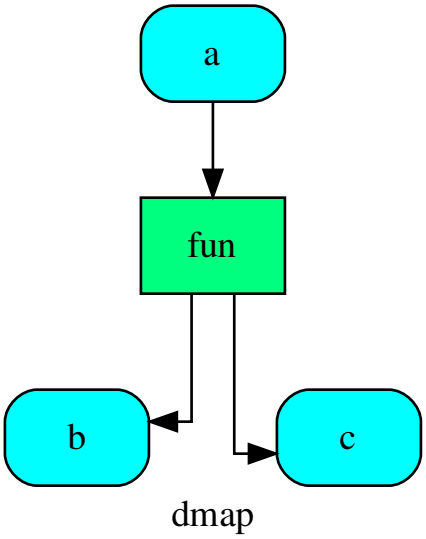
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*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[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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

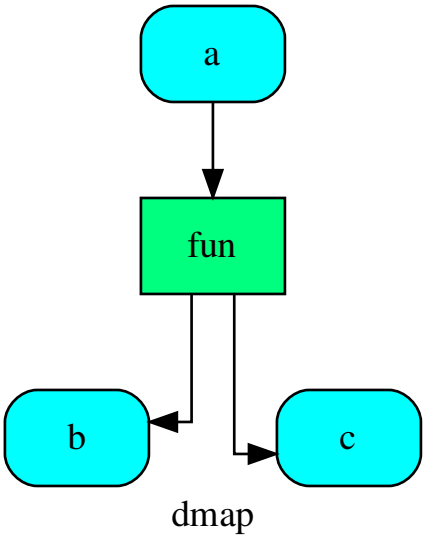
Returns A WebMap.

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=None*, *inputs=None*, *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.

Attributes

var_keyword

var_keyword

DispatchPipe.**var_keyword = None**

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()*

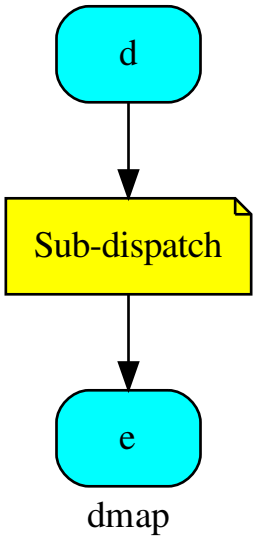
Example:

```
>>> 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')
```

(continues on next page)

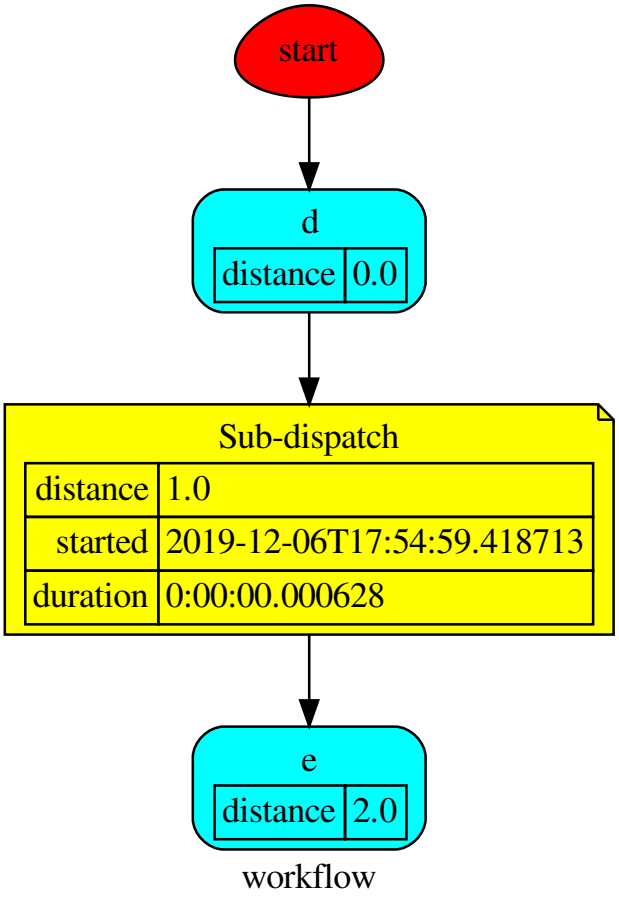
(continued from previous page)

```
>>> dsp = Dispatcher(name='Dispatcher')
>>> dsp.add_function('Sub-dispatch', dispatch, ['d'], ['e'])
'Sub-dispatch'
```



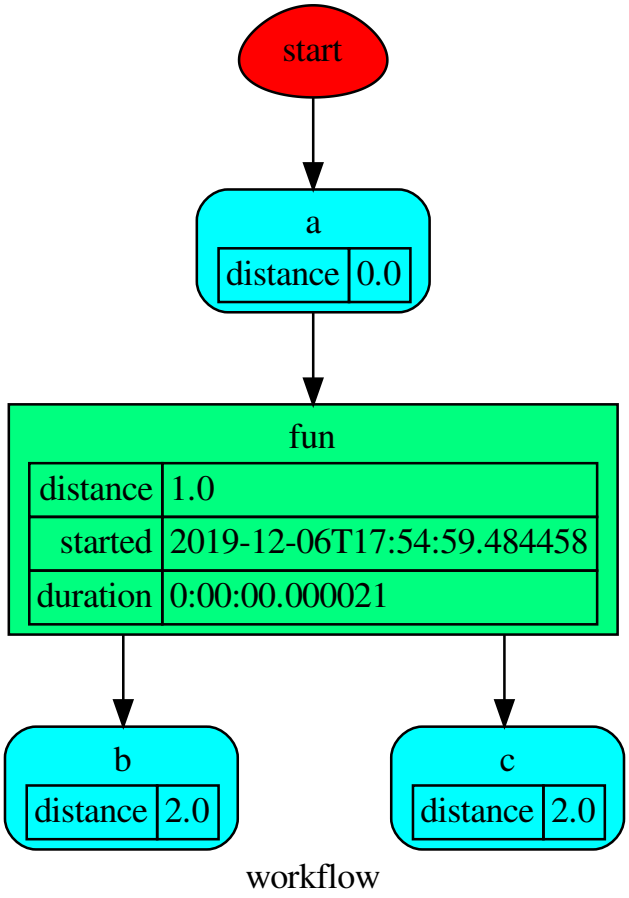
The Dispatcher output is:

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



while, the Sub-dispatch is:

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



Methods

__init__	Initializes the Sub-dispatch.
blue	Constructs a Blueprint out of the current object.
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*.

blue

SubDispatch.**blue**(*memo=None*)

Constructs a Blueprint out of the current object.

Parameters **memo** (*dict[T,schedula.utils.blue.Blueprint]*) – A dictionary to cache Blueprints.

Returns A Blueprint of the current object.

Return type *schedula.utils.blue.Blueprint*

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.

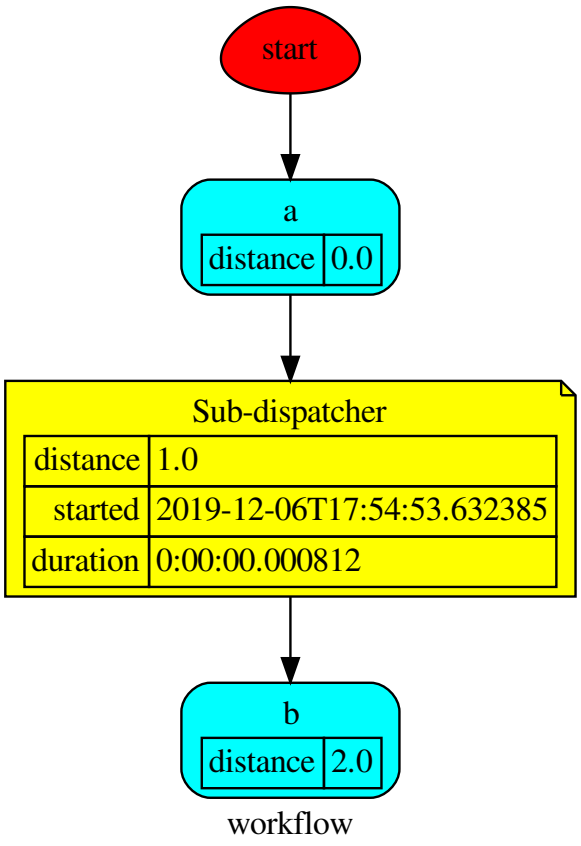
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

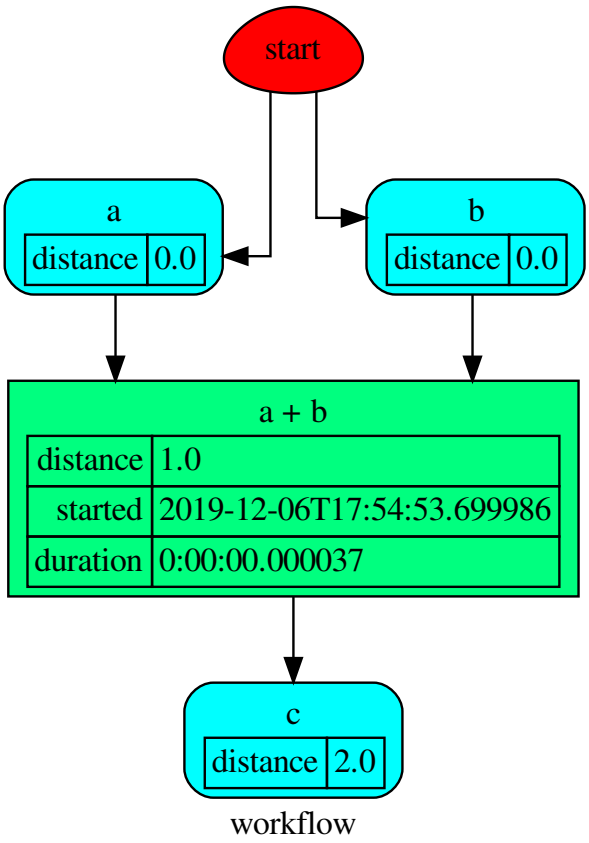
Example:



Get the sub node output:

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

```
>>> sub_dsp, sub_dsp_id = dsp.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

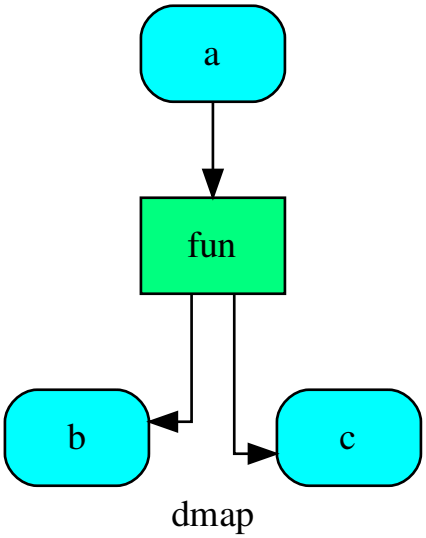
- **workflow** (*bool, optional*) – If True the latest solution will be plotted, otherwise the dmap.
- **view** (*bool, optional*) – Open the rendered directed graph in the DOT language with the sys default opener.
- **edge_data** (*tuple[str], optional*) – Edge attributes to view.
- **node_data** (*tuple[str], 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[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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

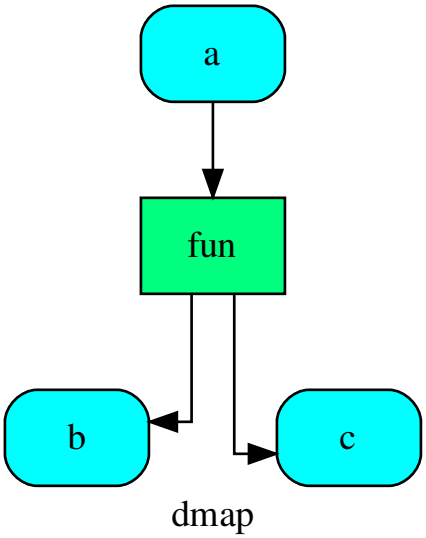
Returns A WebMap.

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]
```

(continues on next page)

(continued from previous page)

```
>>> 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*.

blue(*memo=None*)

Constructs a Blueprint out of the current object.

Parameters **memo** (*dict[T,schedula.utils.blue.Blueprint]*) – A dictionary to cache Blueprints.

Returns A Blueprint of the current object.

Return type *schedula.utils.blue.Blueprint*

SubDispatchFunction

class SubDispatchFunction(*dsp*, *function_id=None*, *inputs=None*, *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*.

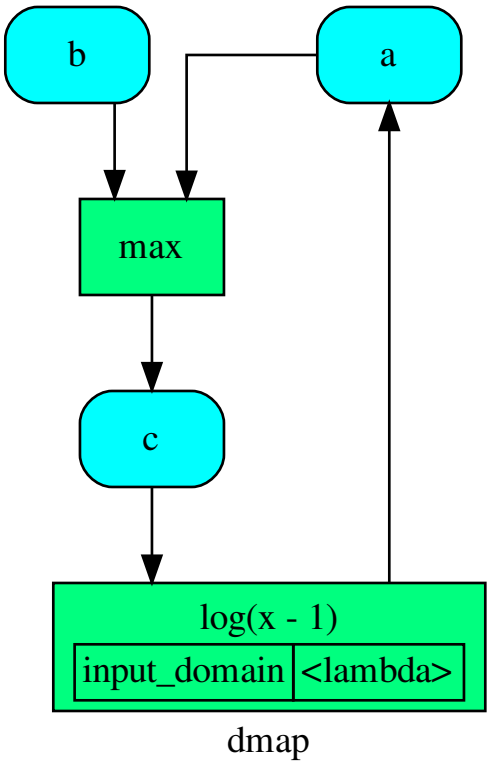
Return type callable

See also:

dispatch(), *shrink_dsp()*

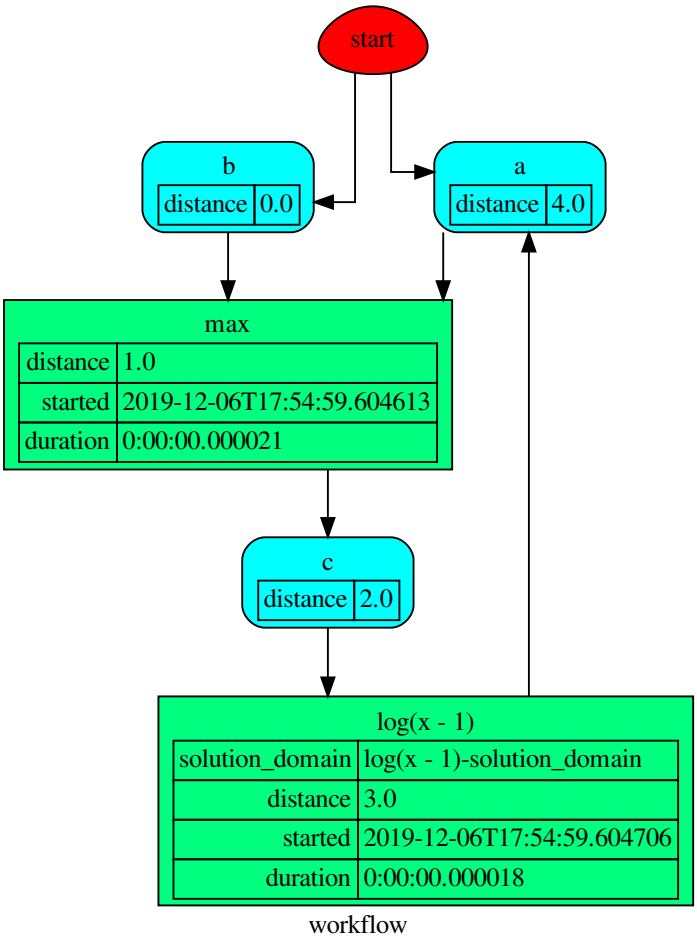
Example:

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



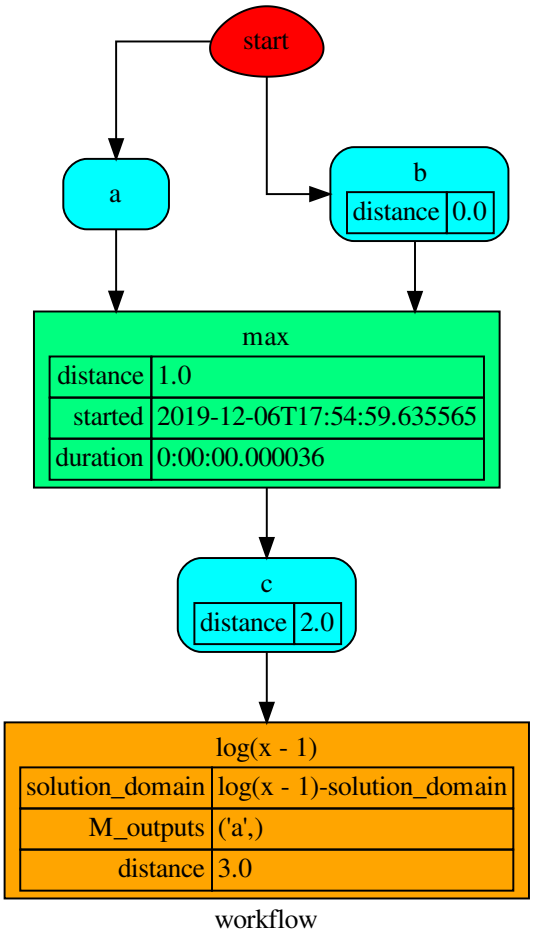
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

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



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

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



workflow

Methods

__init__	Initializes the Sub-dispatch Function.
blue	Constructs a Blueprint out of the current object.
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__

SubDispatchFunction.**__init__**(*dsp*, *function_id=None*, *inputs=None*, *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, optional*) – Function name.
- **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.

blue

SubDispatchFunction.**blue**(*memo=None*)

Constructs a Blueprint out of the current object.

Parameters **memo** (*dict[T,schedula.utils.blue.Blueprint]*) – A dictionary to cache Blueprints.

Returns A Blueprint of the current object.

Return type *schedula.utils.blue.Blueprint*

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.

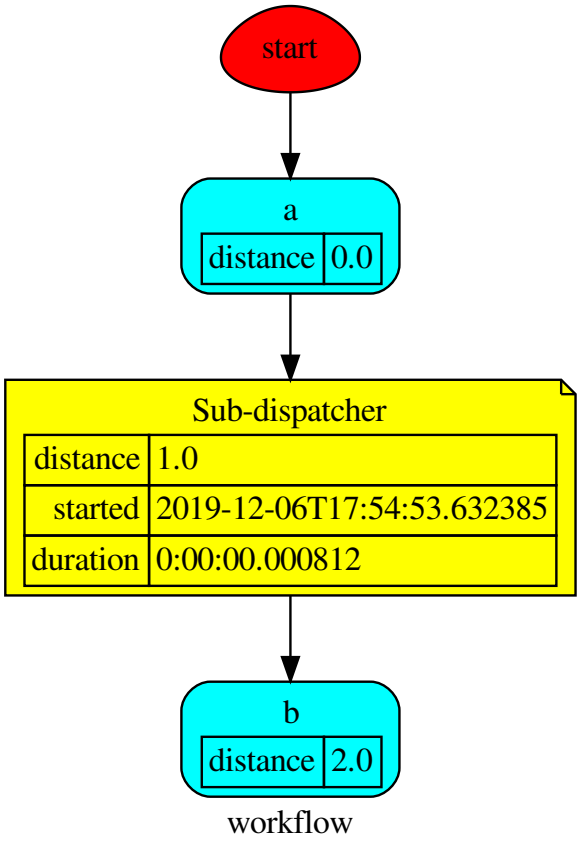
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

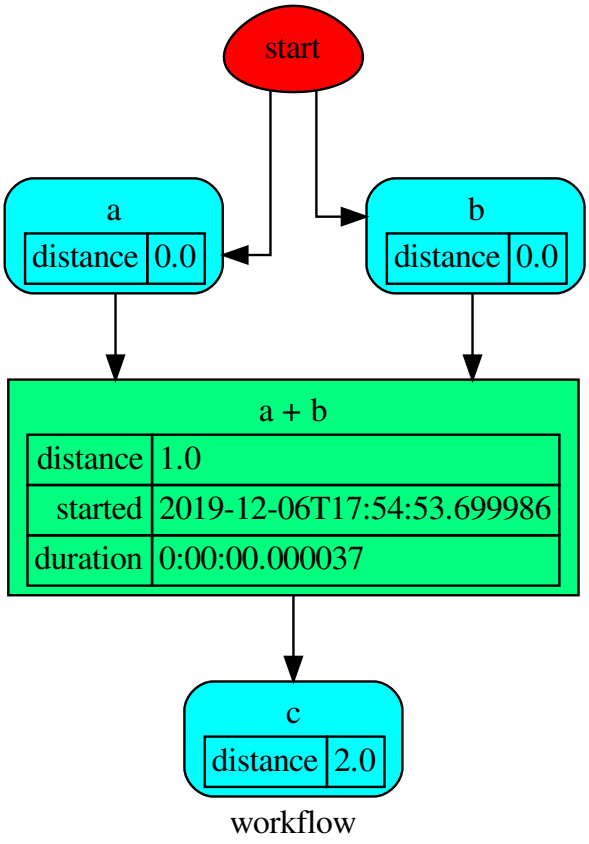
Example:



Get the sub node output:

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

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



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.

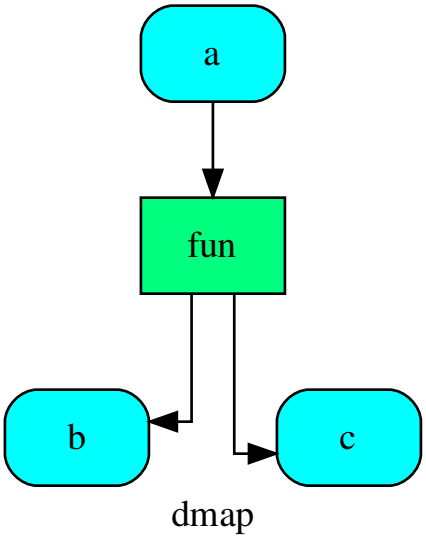
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*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[Site], optional*) – A set of *Site* to maintain alive the back-end 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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

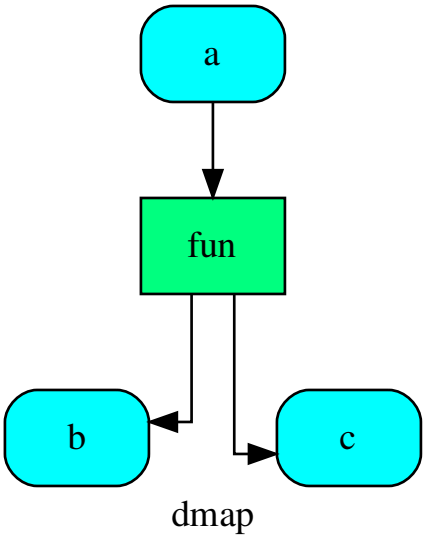
Returns A WebMap.

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=None*, *inputs=None*, *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, optional*) – Function name.
- **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.

Attributes

var_keyword

var_keyword

SubDispatchFunction.**var_keyword = 'kw'**

SubDispatchPipe

class SubDispatchPipe(*dsp*, *function_id=None*, *inputs=None*, *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*.

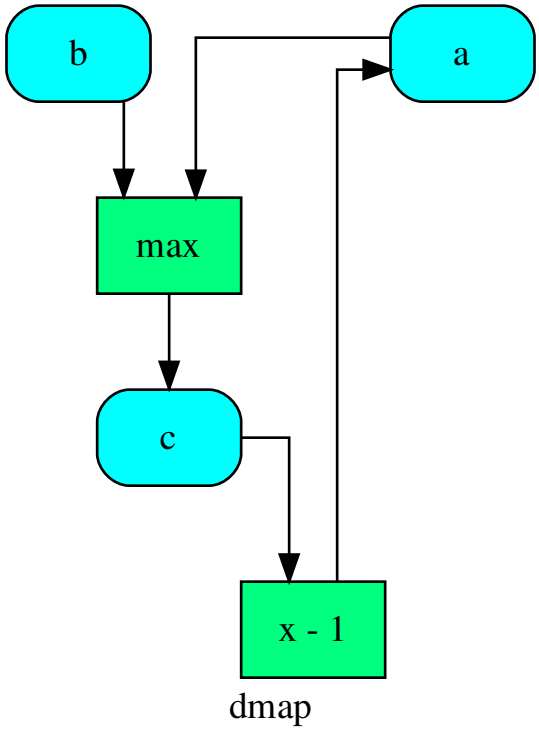
Return type callable

See also:

dispatch(), *shrink_dsp()*

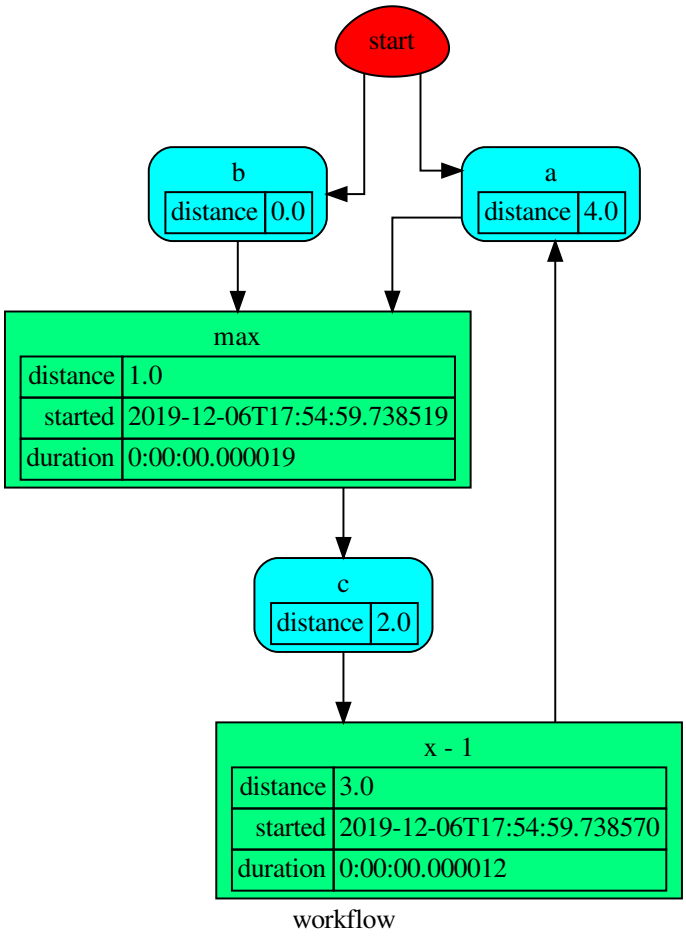
Example:

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



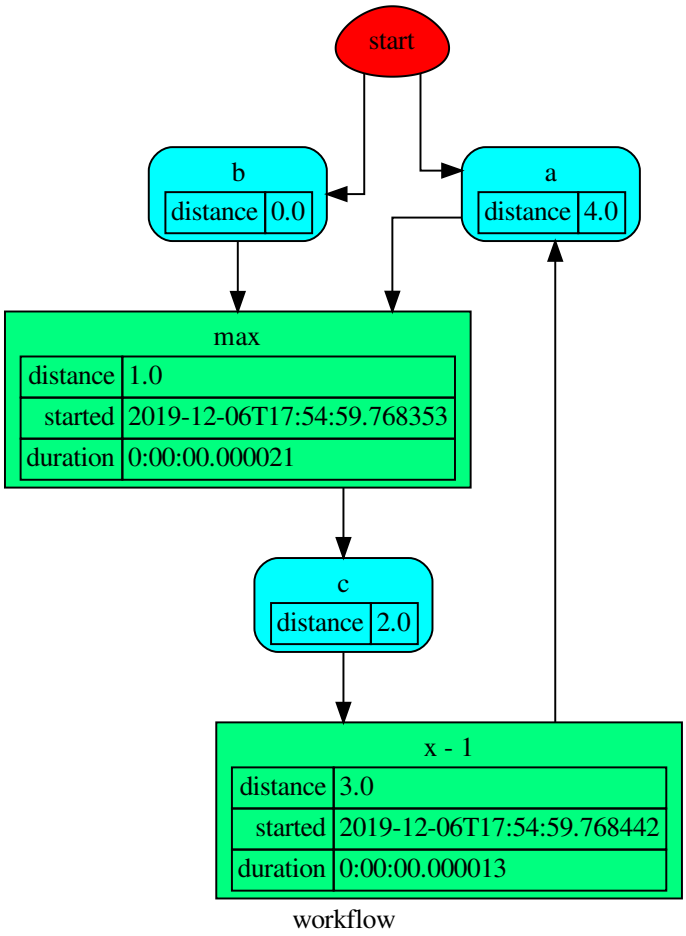
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```
>>> 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.
`blue`	Constructs a Blueprint out of the current object.
`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__

`SubDispatchPipe.__init__`(*dsp*, *function_id=None*, *inputs=None*, *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.

blue

SubDispatchPipe.**blue**(*memo=None*)

Constructs a Blueprint out of the current object.

Parameters **memo** (*dict[T,*schedula.utils.blue.Blueprint*]*) – A dictionary to cache Blueprints.

Returns A Blueprint of the current object.

Return type *schedula.utils.blue.Blueprint*

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.

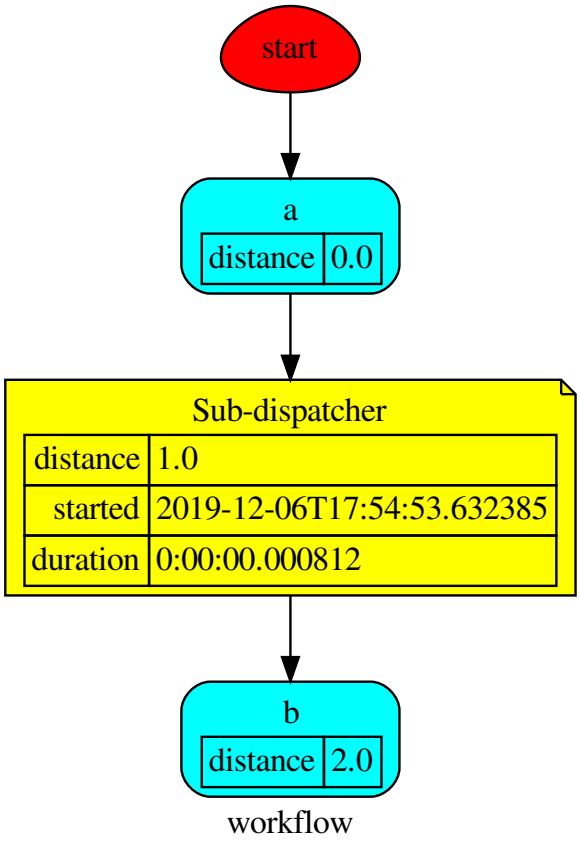
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

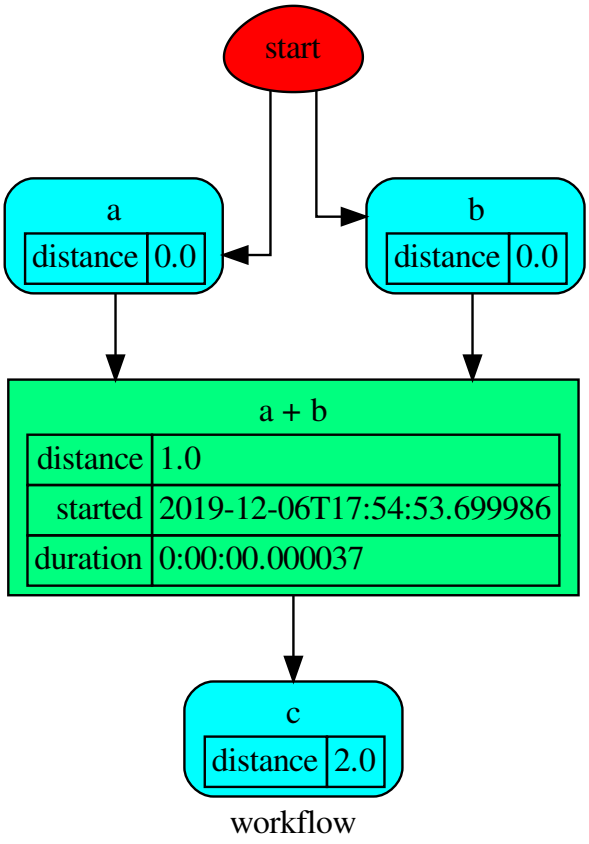
Example:



Get the sub node output:

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

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



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.

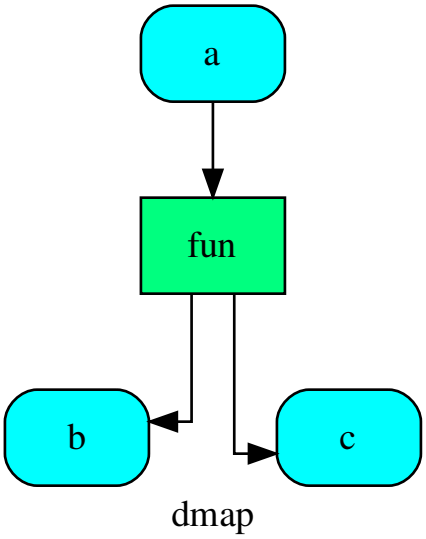
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – Depth of sub-dispatch plots. If negative all levels are plotted.
- **name** (*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[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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

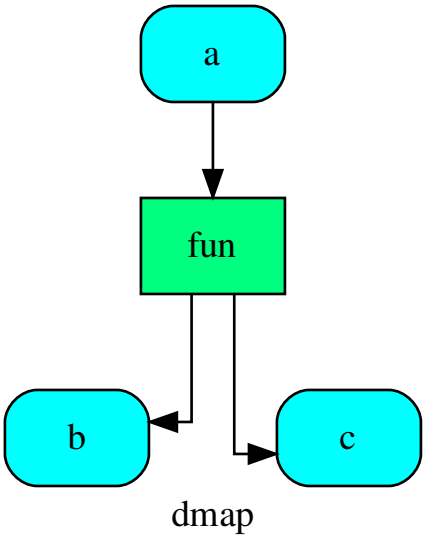
Returns A WebMap.

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=None*, *inputs=None*, *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.

Attributes

var_keyword

var_keyword

SubDispatchPipe.**var_keyword = None**

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:

```
>>> import inspect
>>> def original_func(a, b, *args, c=0):
...     '''Doc'''
...     return a + b + c
>>> func = add_args(original_func, n=2)
>>> func.__name__, func.__doc__
('original_func', 'Doc')
>>> func(1, 2, 3, 4, c=5)
12
>>> str(inspect.signature(func))
'(none, none, a, b, *args, c=0)'
```

Methods

__init__	Initialize self.

__init__

add_args.**__init__** (*func*, *n=1*, *callback=None*)
Initialize self. See help(type(self)) for accurate signature.

__init__ (*func*, *n=1*, *callback=None*)
Initialize self. See help(type(self)) for accurate signature.

inf

class inf
Class to model infinite numbers for workflow distance.

Methods

count	Return number of occurrences of value.
index	Return first index of value.

count

inf.**count**()
Return number of occurrences of value.

index

inf.**index**()
Return first index of value.

Raises ValueError if the value is not present.

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

Attributes

inf	Alias for field number 0
k	
num	Alias for field number 1

inf

inf.**inf**
Alias for field number 0

k

inf.**k = 'ne'**

num

inf.**num**
Alias for field number 1

exc

Defines the dispatcher exception.

Exceptions

DispatcherAbort
DispatcherError
ExecutorShutdown
SkipNode

DispatcherAbort

exception DispatcherAbort

DispatcherError

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

ExecutorShutdown

exception ExecutorShutdown

SkipNode

exception SkipNode(**args*, *ex=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(**args*)

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

__init__	Initialize self.
capitalize	Return a capitalized version of the string.

Continued on next page

Table 53 – continued from previous page

casefold	Return a version of the string suitable for caseless comparisons.
center	Return a centered string of length width.
count	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
encode	Encode the string using the codec registered for encoding.
endswith	Return True if S ends with the specified suffix, False otherwise.
expandtabs	Return a copy 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	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
isalnum	Return True if the string is an alpha-numeric string, False otherwise.
isalpha	Return True if the string is an alphabetic string, False otherwise.
isascii	Return True if all characters in the string are ASCII, False otherwise.
isdecimal	Return True if the string is a decimal string, False otherwise.
isdigit	Return True if the string is a digit string, False otherwise.
isidentifier	Return True if the string is a valid Python identifier, False otherwise.
islower	Return True if the string is a lowercase string, False otherwise.
isnumeric	Return True if the string is a numeric string, False otherwise.
isprintable	Return True if the string is printable, False otherwise.
isspace	Return True if the string is a whitespace string, False otherwise.
istitle	Return True if the string is a title-cased string, False otherwise.
isupper	Return True if the string is an uppercase string, False otherwise.
join	Concatenate any number of strings.
ljust	Return a left-justified string of length width.
lower	Return a copy of the string converted to lowercase.
lstrip	Return a copy of the string with leading whitespace removed.
maketrans	Return a translation table usable for str.translate().
partition	Partition the string into three parts using the given separator.

Continued on next page

Table 53 – continued from previous page

replace	Return a copy 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	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
rjust	Return a right-justified string of length width.
rpartition	Partition the string into three parts using the given separator.
rsplit	Return a list of the words in the string, using sep as the delimiter string.
rstrip	Return a copy of the string with trailing whitespace removed.
split	Return a list of the words in the string, using sep as the delimiter string.
splitlines	Return a list of the lines in the string, breaking at line boundaries.
startswith	Return True if S starts with the specified prefix, False otherwise.
strip	Return a copy of the string with leading and trailing whitespace remove.
swapcase	Convert uppercase characters to lowercase and lowercase characters to uppercase.
title	Return a version of the string where each word is titlecased.
translate	Replace each character in the string using the given translation table.
upper	Return a copy of the string converted to uppercase.
zfill	Pad a numeric string with zeros on the left, to fill a field of the given width.

__init__

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

capitalize

Token.**capitalize**()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold

Token.**casefold**()
Return a version of the string suitable for caseless comparisons.

center

Token.**center**()
Return a centered 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**()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The 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**()
Return a copy 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
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.

Raises ValueError when the substring is not found.

isalnum

Token.**isalnum**()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha

Token.**isalpha**()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii

Token.**isascii**()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal

Token.**isdecimal**()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit

Token.**isdigit**()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier

Token.**isidentifier**()
Return True if the string is a valid Python identifier, False otherwise.

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

islower

Token.**islower**()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric

Token.**isnumeric**()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable

Token.**isprintable**()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace

Token.**isspace**()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle

Token.**istitle**()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper

Token.**isupper**()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join

Token.**join**()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust

Token.**ljust**()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower

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

lstrip

Token.**lstrip**()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

maketrans

static 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**()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

replace

Token.**replace**()
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

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
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.

Raises ValueError when the substring is not found.

rjust

Token.**rjust**()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition

Token.**rpartition**()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit

Token.**rsplit**()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip

Token.**rstrip**()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split

Token.**split**()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines

Token.**splitlines**()
Return a list of the lines in the string, 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**()
Return a copy of the string with leading and trailing whitespace remove.

If chars is given and not None, remove characters in chars instead.

swapcase

Token.**swapcase**()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title

Token.**title**()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate

Token.**translate**()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper

Token.**upper**()
Return a copy of the string converted to uppercase.

zfill

Token.**zfill**()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

__init__(**args*)
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.

Continued on next page

Table 54 – continued from previous page

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)
```

(continues on next page)

(continued from previous page)

```
>>> 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 provides a solution class for dispatch result.

Classes

Solution	Solution class for dispatch result.

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,)*, *full_name=()*)

Solution class for dispatch result.

Methods

__init__	Initialize self.
clear	
copy	
fromkeys	Create a new ordered dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
get_node	Returns a sub node of a dispatcher.
get_sub_dsp_from_workflow	Returns the sub-dispatcher induced by the workflow from sources.
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last is false).
plot	Plots the Dispatcher with a graph in the DOT language with Graphviz.
pop	value.
popitem	Remove and return a (key, value) pair from the dictionary.
result	Set all asynchronous results.
search_node_description	
setdefault	Insert key with a value of default if key is not in the dictionary.
update	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
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,)*, *full_name=()*)

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

clear

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

copy

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

fromkeys

Solution.**fromkeys**()
Create a new ordered dictionary with keys from iterable and values set to value.

get

Solution.**get**()
Return the value for key if key is in the dictionary, else default.

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.

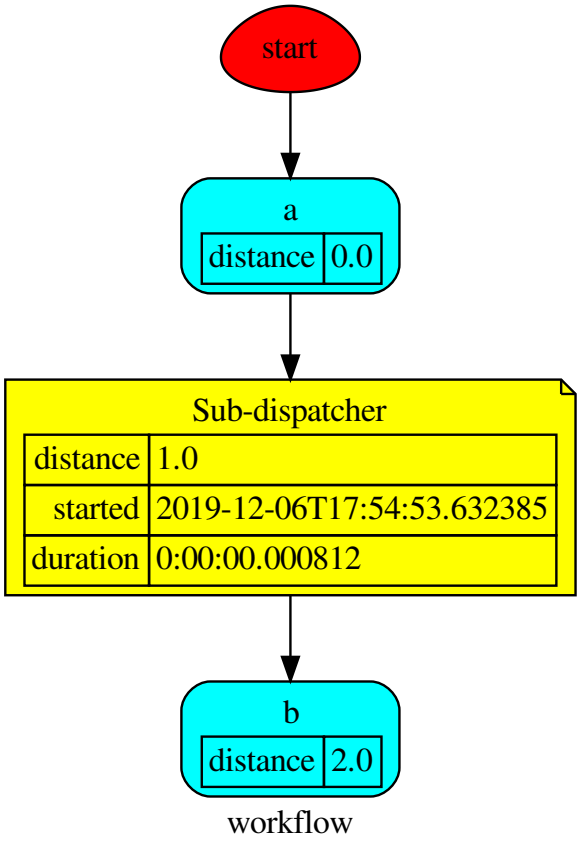
  When 'value_type', returns the data node value's type.

  When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

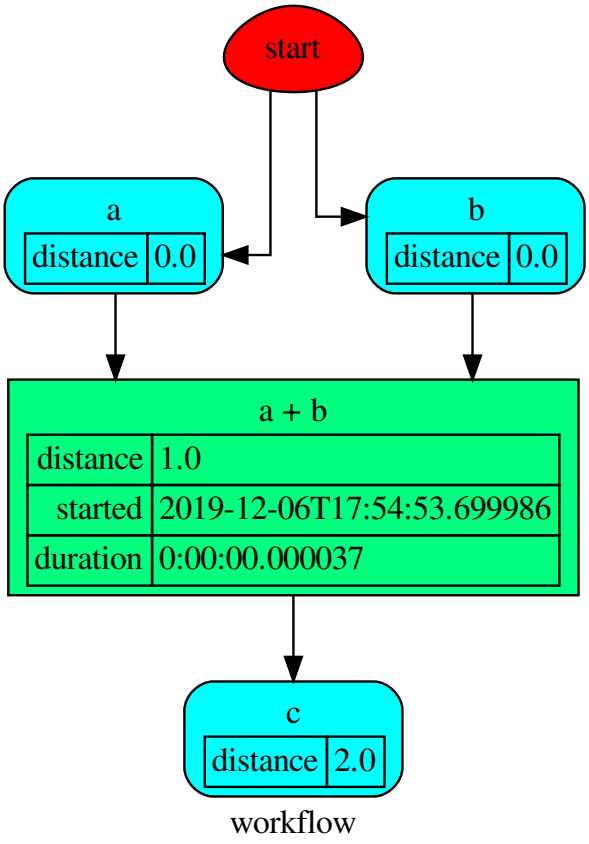
Example:



Get the sub node output:

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

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



get_sub_dsp_from_workflow

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

Returns A sub-dispatcher.

Return type *schedula.dispatcher.Dispatcher*

items

`Solution.`**`items`**`()` → a set-like object providing a view on D's items

keys

`Solution.`**`keys`**`()` → a set-like object providing a view on D's keys

move_to_end

`Solution.`**`move_to_end`**`()`

Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

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', ...).

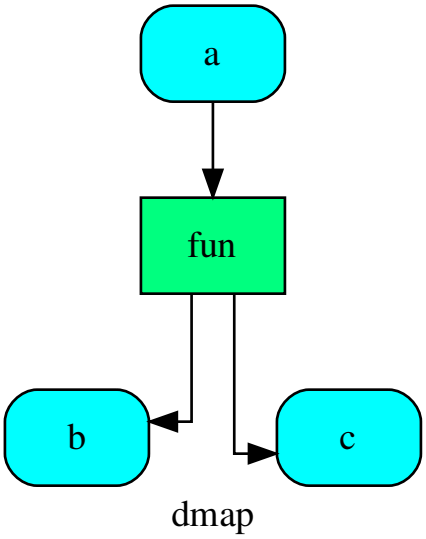
- **engine** (*str, optional*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str, optional*) – Encoding for saving the source.
- **graph_attr** (*dict, optional*) – Dict of (attribute, value) pairs for the graph.
- **node_attr** (*dict, optional*) – Dict of (attribute, value) pairs set 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[Site], optional*) – A set of *Site* to maintain alive the back-end 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**()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

result

Solution.**result**(*timeout=None*)
Set all asynchronous results.

Parameters **timeout** (*float*) – The number of seconds to wait for the result if the futures aren't done. If None, then there is no limit on the wait time.

Returns Update Solution.

Return type *Solution*

search_node_description

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

setdefault

Solution.**setdefault**()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update

Solution.**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

Solution.**values**() → an object providing a view on D's 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[Site], optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?

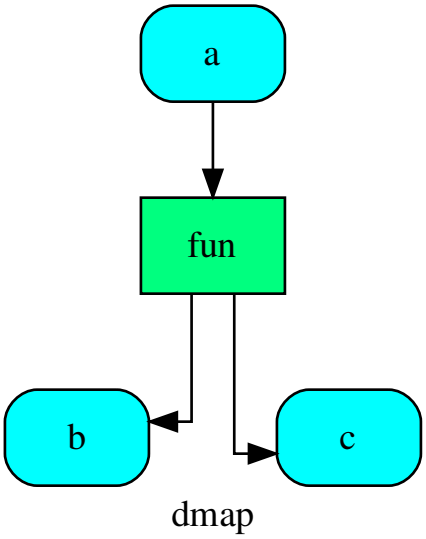
Returns A WebMap.

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,)*, *full_name=()*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

pipe	Returns the full pipe of a dispatch run.

pipe

Solution.**pipe**
Returns the full pipe of a dispatch run.

result(*timeout=None*)
Set all asynchronous results.

Parameters **timeout** (*float*) – The number of seconds to wait for the result if the futures aren't done. If None, then there is no limit on the wait time.

Returns Update Solution.

Return type *Solution*

get_sub_dsp_from_workflow(*sources*, *reverse=False*, *add_missing=False*, *check_inputs=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.
- **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.

Returns A sub-dispatcher.

Return type *schedula.dispatcher.Dispatcher*

pipe
Returns the full pipe of a dispatch run.

web

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

Classes

FolderNodeWeb
WebFolder
WebMap
WebNode

FolderNodeWeb

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

Methods

__init__	Initialize self.
dot	
href	
items	
parent_ref	
render_funcs	
render_size	
style	
yield_attr	

__init__

FolderNodeWeb.**__init__**(*folder*, *node_id*, *attr*, ***options*)
Initialize self. See help(type(self)) for accurate signature.

dot

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

href

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

items

FolderNodeWeb.**items**()

parent_ref

FolderNodeWeb.**parent_ref**(*context*, *node_id*, *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*)
Initialize self. See help(type(self)) for accurate signature.

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'), '!': ('dot', 'table'), '*': ('link'**

node_styles

FolderNodeWeb.**node_styles = {'error': {empty: {'fillcolor': 'gray', 'label': 'empty**

re_node

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

title

FolderNodeWeb.**title**

type

FolderNodeWeb.**type**

WebFolder

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

Methods

__init__	Initialize self.
dot	
view	

__init__

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

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

dot

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

view

WebFolder.**view**(*filepath*, *context=None*)

__init__ (*item*, *dsp*, *graph*, *obj*, *name=''*, *workflow=False*, *digraph=None*, *parent=None*, ***options*)
Initialize self. See help(type(self)) for accurate signature.

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 = {'body': {'splines': 'ortho', 'style': 'filled'}, 'edge_attr':**

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__	Initialize self.
add_items	
app	
clear	
copy	
fromkeys	Create a new ordered dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
get_dsp_from	
get_sol_from	
items	
keys	
move_to_end	Move an existing element to the end (or beginning if last is false).
pop	value.
popitem	Remove and return a (key, value) pair from the dictionary.
render	
rules	
setdefault	Insert key with a value of default if key is not in the dictionary.
site	
site_index	
update	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values	

__init__

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

add_items

WebMap.**add_items**(*item*, *workflow=False*, *depth=-1*, *folder=None*, ***options*)

app

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

clear

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

copy

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

fromkeys

WebMap.**fromkeys**()
Create a new ordered dictionary with keys from iterable and values set to value.

get

WebMap.**get**()
Return the value for key if key is in the dictionary, else default.

get_dsp_from

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

get_sol_from

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

items

WebMap.**items**() → a set-like object providing a view on D's items

keys

WebMap.**keys**() → a set-like object providing a view on D's keys

move_to_end

WebMap.**move_to_end**()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

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**()
Remove and return a (key, value) pair from the dictionary.

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**()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

site

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

site_index

WebMap.**site_index**(***kwargs*)

update

WebMap.**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

WebMap.**values**() → an object providing a view on D's values

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

Attributes

include_folders_as_filenames
nodes
options

include_folders_as_filenames

WebMap.**include_folders_as_filenames = False**

nodes

WebMap.**nodes**

options

WebMap.**options = {'digraph', 'edge_data', 'max_lines', 'max_width', 'node_data', 'node**

WebNode

class WebNode(*folder*, *node_id*, *item*, *obj*, *dsp_node_id*)

Methods

__init__	Initialize self.
render	
view	

__init__

WebNode.**__init__**(*folder*, *node_id*, *item*, *obj*, *dsp_node_id*)
Initialize self. See help(type(self)) for accurate signature.

render

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

view

WebNode.**view**(*filepath*, **args*, ***kwargs*)

__init__(*folder*, *node_id*, *item*, *obj*, *dsp_node_id*)
Initialize self. See help(type(self)) for accurate signature.

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**

2.1.8.3 ext

It provides sphinx extensions.

Extensions:

autosummary	It is a patch to sphinx.ext.autosummary.
dispatcher	

autosummary

It is a patch to sphinx.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*, *app=None*)

get_members

get_members(*app*, *obj*, *typ*, *include_public=()*, *imported=False*)

process_generate_options

process_generate_options(*app*)

setup

setup(*app*)

dispatcher

Functions

setup

setup

setup(*app*)

2.1.9 Changelog

2.1.9.1 v0.3.7 (2019-12-06)

Feat

- (drw): Update the *index* GUI of the plot.
- (appveyor): Drop *appveyor* in favor of *travis*.
- (travis): Update travis configuration file.
- (plot): Add node link and id in graph plot.

Fix

- (drw): Render dot in temp folder.
- (plot): Add *quiet* arg to *_view* method.
- (doc): Correct missing gh links.
- (core) #17: Correct deprecated Graph attribute.

2.1.9.2 v0.3.6 (2019-10-18)

Fix

- (setup) #17: Update version networkx.
- (setup) #13: Build universal wheel.
- (alg) #15: Escape % in node id.
- (setup) #14: Update tests requirements.
- (setup): Add env *ENABLE_SETUP_LONG_DESCRIPTION*.

2.1.9.3 v0.3.4 (2019-07-15)

Feat

- (binder): Add *@jupyterlab/plotly-extension*.
- (binder): Customize *Site._repr_html_* with env *SCHEDULA_SITE_REPR_HTML*.
- (binder): Add *jupyter-server-proxy*.
- (doc): Add binder examples.
- (gen): Create super-class of *Token*.
- (dsp): Improve error message.

Fix

- (binder): Simplify *processing_chain* example.
- (setup): Exclude *binder* and *examples* folders as packages.
- (doc): Correct binder data.
- (doc): Update examples for binder.
- (doc): Add missing requirements binder.
- (test): Add *state* to fake directive.
- (import): Remove stub file to enable autocomplete.
- Update to canonical pypi name of beautifulsoup4.

2.1.9.4 v0.3.3 (2019-04-02)

Feat

- (dispatcher): Improve error message.

Fix

- (doc): Correct bug for sphinx AutoDirective.
- (dsp): Add dsp as kwargs for a new Blueprint.
- (doc): Update PEP and copyright.

2.1.9.5 v0.3.2 (2019-02-23)

Feat

- (core): Add stub file.
- (sphinx): Add Blueprint in Dispatcher documenter.
- (sphinx): Add BlueDispatcher in documenter.
- (doc): Add examples.
- (blue): Customizable memo registration of blueprints.

Fix

- (sphinx): Correct bug when " is in csv-table directive.
- (core): Set module attribute when *__getattr__* is invoked.
- (doc): Correct utils description.
- (setup): Improve keywords.
- (drw): Correct tooltip string format.
- (version): Correct import.

2.1.9.6 v0.3.1 (2018-12-10)

Fix

- (setup): Correct long description for pypi.
- (dsp): Correct bug *DispatchPipe* when dill.

2.1.9.7 v0.3.0 (2018-12-08)

Feat

- (blue, dispatcher): Add method *extend* to extend Dispatcher or Blueprint with Dispatchers or Blueprints.
- (blue, dsp): Add *BlueDispatcher* class + remove *DFun* util.
- (core): Remove *weight* attribute from *Dispatcher* struc.
- (dispatcher): Add method *add_func* to *Dispatcher*.
- (core): Remove *remote_links* attribute from dispatcher data nodes.
- (core): Implement callable raise option in *Dispatcher*.
- (core): Add feature to dispatch asynchronously and in parallel.
- (setup): Add python 3.7.
- (dsp): Use the same *dsp.solution* class in *SubDispatch* functions.

Fix

- (dsp): Do not copy solution when call *DispatchPipe*, but reset solution when copying the obj.
- (alg): Correct and clean *get_sub_dsp_from_workflow* algorithm.
- (sol): Ensure *bool* output from *input_domain* call.
- (dsp): Parse arg and kw using *SubDispatchFunction.__signature__*.
- (core): Do not support python 3.4.
- (asy): Do not dill the Dispatcher solution.
- (dispatcher): Correct bug in removing remote links.
- (core): Simplify and correct Exception handling.
- (dsp): Postpone *__signature__* evaluation in *add_args*.
- (gen): Make Token constant when pickled.
- (sol): Move callback invocation in *_evaluate_node*.
- (core) #11: Lazy import of modules.
- (sphinx): Remove warnings.
- (dsp): Add missing *code* option in *add_function* decorator.

Other

- Refact: Update documentation.

2.1.9.8 v0.2.8 (2018-10-09)

Feat

- (dsp): Add inf class to model infinite numbers.

2.1.9.9 v0.2.7 (2018-09-13)

Fix

- (setup): Correct bug when *long_description* fails.

2.1.9.10 v0.2.6 (2018-09-13)

Feat

- (setup): Patch to use *sphinxcontrib.restbuilder* in setup *long_description*.

2.1.9.11 v0.2.5 (2018-09-13)

Fix

- (doc): Correct link docs_status.
- (setup): Use text instead rst to compile *long_description* + add logging.

2.1.9.12 v0.2.4 (2018-09-13)

Fix

- (sphinx): Correct bug sphinx==1.8.0.
- (sphinx): Remove all sphinx warnings.

2.1.9.13 v0.2.3 (2018-08-02)

Fix

- (des): Correct bug when SubDispatchFunction have no *outputs*.

2.1.9.14 v0.2.2 (2018-08-02)

Fix

- (des): Correct bug of get_id when tuple ids nodes are given as input or outputs of a sub_dsp.
- (des): Correct bug when tuple ids are given as *inputs* or *outputs* of *add_dispatcher* method.

2.1.9.15 v0.2.1 (2018-07-24)

Feat

- (setup): Update *Development Status* to *5 - Production/Stable*.
- (setup): Add additional project_urls.
- (doc): Add changelog to rtd.

Fix

- (doc): Correct link docs_status.
- (des): Correct bugs get_des.

2.1.9.16 v0.2.0 (2018-07-19)

Feat

- (doc): Add changelog.
- (travis): Test extras.
- (des): Avoid using sphinx for *getargspec*.
- (setup): Add extras_require to setup file.

Fix

- (setup): Correct bug in *get_long_description*.

2.1.9.17 v0.1.19 (2018-06-05)

Fix

- (dsp): Add missing content block in note directive.
- (drw): Make sure to plot same sol as function and as node.
- (drw): Correct format of started attribute.

2.1.9.18 v0.1.18 (2018-05-28)

Feat

- (dsp): Add *DispatchPipe* class (faster pipe execution, it overwrite the existing solution).
- (core): Improve performances replacing *datetime.today()* with *time.time()*.

2.1.9.19 v0.1.17 (2018-05-18)

Feat

- (travis): Run coveralls in python 3.6.

Fix

- (web): Skip Flask logging for the doctest.
- (ext.dispatcher): Update to the latest Sphinx 1.7.4.
- (des): Use the proper dependency (i.e., *sphinx.util.inspect*) for *getargspec*.
- (drw): Set socket option to reuse the address (host:port).
- (setup): Correct dill requirements *dill>=0.2.7.1 –> dill!=0.2.7*.

2.1.9.20 v0.1.16 (2017-09-26)

Fix

- (requirements): Update dill requirements.

2.1.9.21 v0.1.15 (2017-09-26)

Fix

- (networkx): Update according to networkx 2.0.

2.1.9.22 v0.1.14 (2017-07-11)

Fix

- (io): pin dill version <=0.2.6.
- (abort): abort was setting Exception.args instead of *sol* attribute.

Other

- Merge pull request #9 from ankostis/fixabortex.

2.1.9.23 v0.1.13 (2017-06-26)

Feat

- (appveyor): Add python 3.6.

Fix

- (install): Force update setuptools>=36.0.1.
- (exc): Do not catch KeyboardInterrupt exception.
- (doc) #7: Catch exception for sphinx 1.6.2 (listeners are moved in EventManager).
- (test): Skip empty error message.

2.1.9.24 v0.1.12 (2017-05-04)

Fix

- (drw): Catch dot error and log it.

2.1.9.25 v0.1.11 (2017-05-04)

Feat

- (dsp): Add *add_function* decorator to add a function to a dsp.
- (dispatcher) #4: Use *kk_dict* function to parse inputs and outputs of *add_dispatcher* method.
- (dsp) #4: Add *kk_dict* function.

Fix

- (doc): Replace type function with callable.
- (drw): Folder name without ext.
- (test): Avoid Documentation of DspPlot.
- (doc): fix docstrings types.

2.1.9.26 v0.1.10 (2017-04-03)

Feat

- (sol): Close sub-dispatcher solution when all outputs are satisfied.

Fix

- (drw): Log error when dot is not able to render a graph.

2.1.9.27 v0.1.9 (2017-02-09)

Fix

- (appveyor): Setup of lmxl.
- (drw): Update plot index.

2.1.9.28 v0.1.8 (2017-02-09)

Feat

- (drw): Update plot index + function code highlight + correct plot outputs.

2.1.9.29 v0.1.7 (2017-02-08)

Fix

- (setup): Add missing package_data.

2.1.9.30 v0.1.6 (2017-02-08)

Fix

- (setup): Avoid setup failure due to get_long_description.
- (drw): Avoid to plot unneeded weight edges.
- (dispatcher): get_sub_dsp_from_workflow set correctly the remote links.

2.1.9.31 v0.1.5 (2017-02-06)

Feat

- (exl): Drop exl module because of formulas.
- (sol): Add input value of filters in solution.

Fix

- (drw): Plot just one time the filer attribute in workflow *+filers|solution_filters* .

2.1.9.32 v0.1.4 (2017-01-31)

Feat

- (drw): Save autoplot output.
- (sol): Add filters and function solutions to the workflow nodes.
- (drw): Add filters to the plot node.

Fix

- (dispatcher): Add missing function data inputs edge representation.
- (sol): Correct value when apply filters on setting the node output.
- (core): get_sub_dsp_from_workflow blockers can be applied to the sources.

2.1.9.33 v0.1.3 (2017-01-29)

Fix

- (dsp): Raise a DispatcherError when the pipe workflow is not respected instead KeyError.
- (dsp): Unresolved references.

2.1.9.34 v0.1.2 (2017-01-28)

Feat

- (dsp): add_args _set_doc.
- (dsp): Remove parse_args class.
- (readme): Appveyor badge status == master.
- (dsp): Add _format option to *get_unused_node_id*.
- (dsp): Add wildcard option to *SubDispatchFunction* and *SubDispatchPipe*.
- (drw): Create sub-package drw.

Fix

- (dsp): combine nested dicts with different length.
- (dsp): are_in_nested_dicts return false if nested_dict is not a dict.
- (sol): Remove defaults when setting wildcards.
- (drw): Misspelling *outpus* –> *outputs*.
- (directive): Add exception on graphviz patch for sphinx 1.3.5.

2.1.9.35 v0.1.1 (2017-01-21)

Fix

- (site): Fix ResourceWarning: unclosed socket.
- (setup): Not log sphinx warnings for long_description.
- (travis): Wait util the server is up.
- (rtd): Missing requirement dill.
- (travis): Install first - pip install -r dev-requirements.txt.

- (directive): Tagname from _img to img.
- (directive): Update minimum sphinx version.
- (readme): Badge svg links.

Other

- Add project descriptions.
- (directive): Rename schedula.ext.dsp_directive –> schedula.ext.dispatcher.
- Update minimum sphinx version and requests.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

P

R

S

T

U

V

W