
schedula Documentation

Release 1.0.0

Vincenzo Arcidiacono

Jan 05, 2020

Table of Contents

1	About schedula	3
2	Installation	5
3	Tutorial	7
4	Asynchronous and Parallel dispatching	15
5	Contributing to schedula	17
6	Donate	19
7	API Reference	21
8	Changelog	257
9	Indices and tables	269
	Python Module Index	271
	Index	273

2020-01-05 09:00:00

<https://github.com/vincilit2000/schedula>

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

<https://schedula.readthedocs.io/>

<https://github.com/vincilit2000/schedula/wiki/>

<https://github.com/vincilit2000/schedula/releases/>

flow-based programming, dataflow, parallel, async, scheduling, dispatch, functional programming, dataflow programming

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

EUPL 1.1+

CHAPTER 1

About `schedula`

`schedula` is a dynamic flow-based programming environment for python, that handles automatically the control flow of the program. The control flow generally is represented by a Directed Acyclic Graph (DAG), where nodes are the operations/functions to be executed and edges are the dependencies between them.

The algorithm of **`schedula`** dates back to 2014, when a colleague asked for a method to automatically populate the missing data of a database. The imputation method chosen to complete the database was a system of interdependent physical formulas - i.e., the inputs of a formula are the outputs of other formulas. The current library has been developed in 2015 to support the design of the CO₂MPAS [tool](#) - a CO₂ vehicle [simulator](#). During the developing phase, the physical formulas (more than 700) were known on the contrary of the software inputs and outputs.

1.1 Why `schedula`?

The design of flow-based programs begins with the definition of the control flow graph, and implicitly of its inputs and outputs. If the program accepts multiple combinations of inputs and outputs, you have to design and code all control flow graphs. With normal schedulers, it can be very demanding.

While with **`schedula`**, giving whatever set of inputs, it automatically calculates any of the desired computable outputs, choosing the most appropriate DAG from the dataflow execution model.

Note: The DAG is determined at runtime and it is extracted using the shortest path from the provided inputs. The path is calculated based on a weighted directed graph (dataflow execution model) with a modified Dijkstra algorithm.

`schedula` makes the code easy to debug, to optimize, and to present it to a non-IT audience through its interactive graphs and charts. It provides the option to run a model asynchronously or in parallel managing automatically the Global Interpreter Lock (GIL), and to convert a model into a web API service.

1.2 Dataflow Execution Model

The *Dispatcher* is the main model of **schedula** and it represents the dataflow execution model of your code. It is defined by a weighted directed graph. The nodes are the operations to be executed. The arcs between the nodes represent their dependencies. The weights are used to determine the control flow of your model (i.e. operations' invocation order).

Conceptually, when the model is executed, input-data flows as tokens along the arcs. When the execution/*dispatch()* begins, a special node (*START*) places the data onto key input arcs, triggering the computation of the control flow. The latter is represented by a Directed Acyclic Graph (DAG) and it is defined as the shortest path from the provided inputs. It is computed using the weighted directed graph and a modified Dijkstra algorithm. A node is executed when its inputs and domain are satisfied. After the node execution, new data are placed on some or all of its output arcs. In presence of cycles in the graph, to avoid undesired infinite loops, the nodes are computed only once. In case of an execution failure of a node, the algorithm searches automatically for an alternative path to compute the desired outputs. The nodes are differentiated according to their scope. **schedula** defines three node's types:

- **data node**: stores the data into the solution. By default, it is executable when it receives one input arch.
- **function node**: invokes the user defined function and place the results onto its output arcs. It is executable when all inputs are satisfied and it has at least one data output to be computed.
- **sub-dispatcher node**: packages particular dataflow execution model as sub component of the parent dispatcher. Practically, it creates a bridge between two dispatchers (parent and child) linking some data nodes. It allows to simplify your model, reusing some functionality defined in other models.

The key advantage is that, by this method, the scheduling is not affected by the operations' execution times. Therefore, it is deterministic and reproducible. Moreover, since it is based on flow-based programming, it inherits the ability to execute more than one operation at the same time, making the program executable in parallel. The following video shows an example of a runtime dispatch.

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](#).

Let's assume that we want develop a tool to automatically manage the symmetric cryptography. The base idea is to open a file, read its content, encrypt or decrypt the data and then write them out to a new file. This tutorial shows how to:

1. *define* and *execute* a dataflow execution model,
2. *extract* a sub-model, and
3. *deploy* a web API service.

Note: You can find more examples, on how to use the **schedula** library, into the folder [examples](#).

3.1 Model definition

First of all we start defining an empty *Dispatcher* named *symmetric_cryptography* that defines the dataflow execution model:

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

There are two main ways to get a key, we can either generate a new one or use one that has previously been generated. Hence, we can define three functions to simply generate, save, and load the key. To automatically populate the model inheriting the arguments names, we can use the decorator *add_function()* as follow:

```
>>> import os.path as osp
>>> from cryptography.fernet import Fernet
>>> @sh.add_function(dsp, outputs=['key'], weight=2)
... def generate_key():
...     return Fernet.generate_key().decode()
>>> @sh.add_function(dsp)
... def write_key(key_fpath, key):
```

(continues on next page)

(continued from previous page)

```
...     with open(key_fpath, 'w') as f:
...         f.write(key)
>>> @sh.add_function(dsp, outputs=['key'], input_domain=osp.isfile)
... def read_key(key_fpath):
...     with open(key_fpath) as f:
...         return f.read()
```

Note: Since Python does not come with anything that can encrypt/decrypt files, in this tutorial, we use a third party module named `cryptography`. To install it execute `pip install cryptography`.

To encrypt/decrypt a message, you will need a key as previously defined and your data *encrypted* or *decrypted*. Therefore, we can define two functions and add them, as before, to the model:

```
>>> @sh.add_function(dsp, outputs=['encrypted'])
... def encrypt_message(key, decrypted):
...     return Fernet(key.encode()).encrypt(decrypted.encode()).decode()
>>> @sh.add_function(dsp, outputs=['decrypted'])
... def decrypt_message(key, encrypted):
...     return Fernet(key.encode()).decrypt(encrypted.encode()).decode()
```

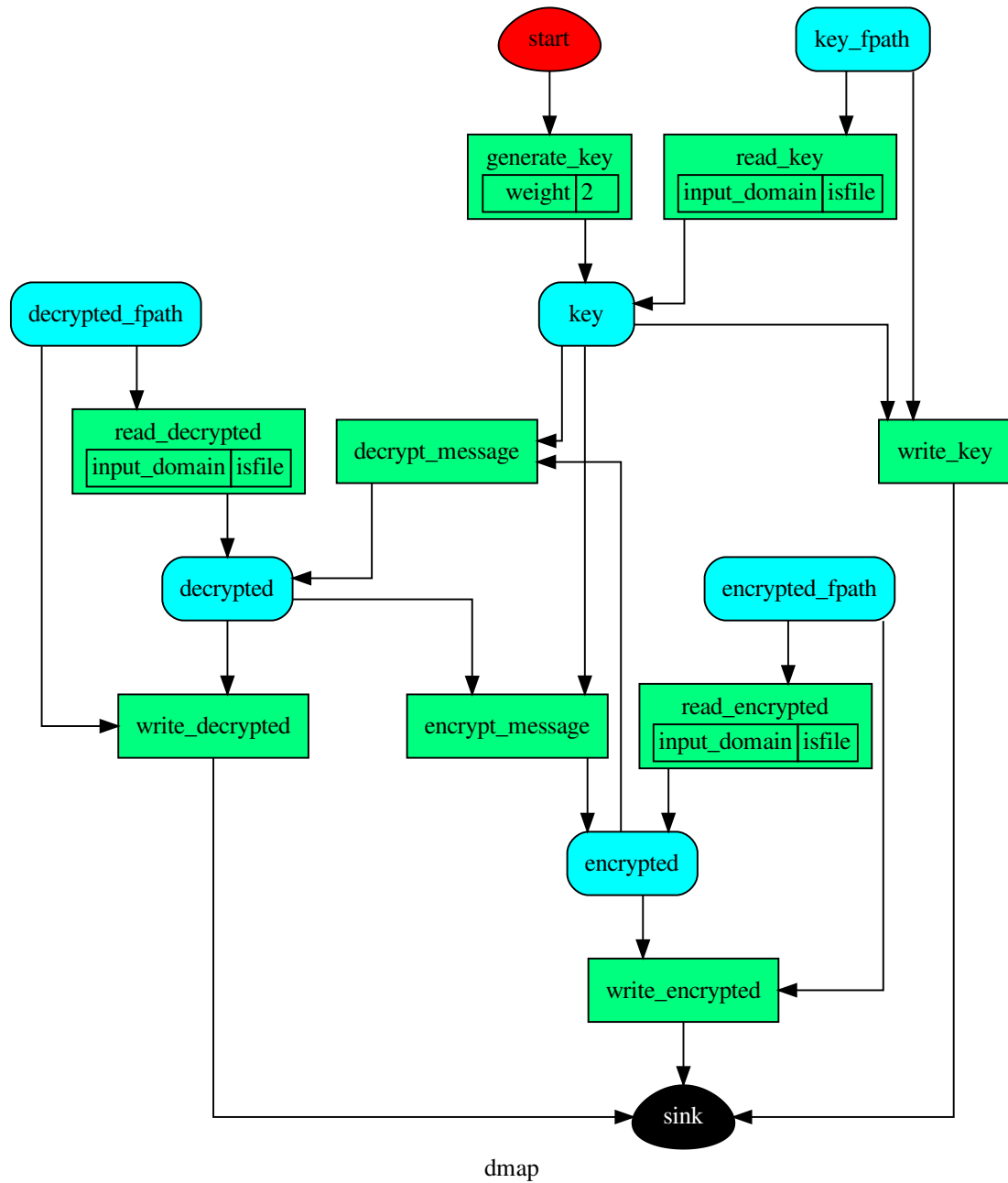
Finally, to read and write the encrypted or decrypted message, according to the functional programming philosophy, we can reuse the previously defined functions `read_key` and `write_key` changing the model mapping (i.e., *function_id*, *inputs*, and *outputs*). To add to the model, we can simply use the `add_function` method as follow:

```
>>> dsp.add_function(
...     function_id='read_decrypted',
...     function=read_key,
...     inputs=['decrypted_fpath'],
...     outputs=['decrypted']
... )
'read_decrypted'
>>> dsp.add_function(
...     'read_encrypted', read_key, ['encrypted_fpath'], ['encrypted'],
...     input_domain=osp.isfile
... )
'read_encrypted'
>>> dsp.add_function(
...     'write_decrypted', write_key, ['decrypted_fpath', 'decrypted'],
...     input_domain=osp.isfile
... )
'write_decrypted'
>>> dsp.add_function(
...     'write_encrypted', write_key, ['encrypted_fpath', 'encrypted']
... )
'write_encrypted'
```

Note: For more details on how to create a *Dispatcher* see: `add_data()`, `add_func()`, `add_function()`, `add_dispatcher()`, `SubDispatch`, `SubDispatchFunction`, `SubDispatchPipe`, and `DispatchPipe`.

To inspect and visualize the dataflow execution model, you can simply plot the graph as follow:

```
>>> dsp.plot()
```

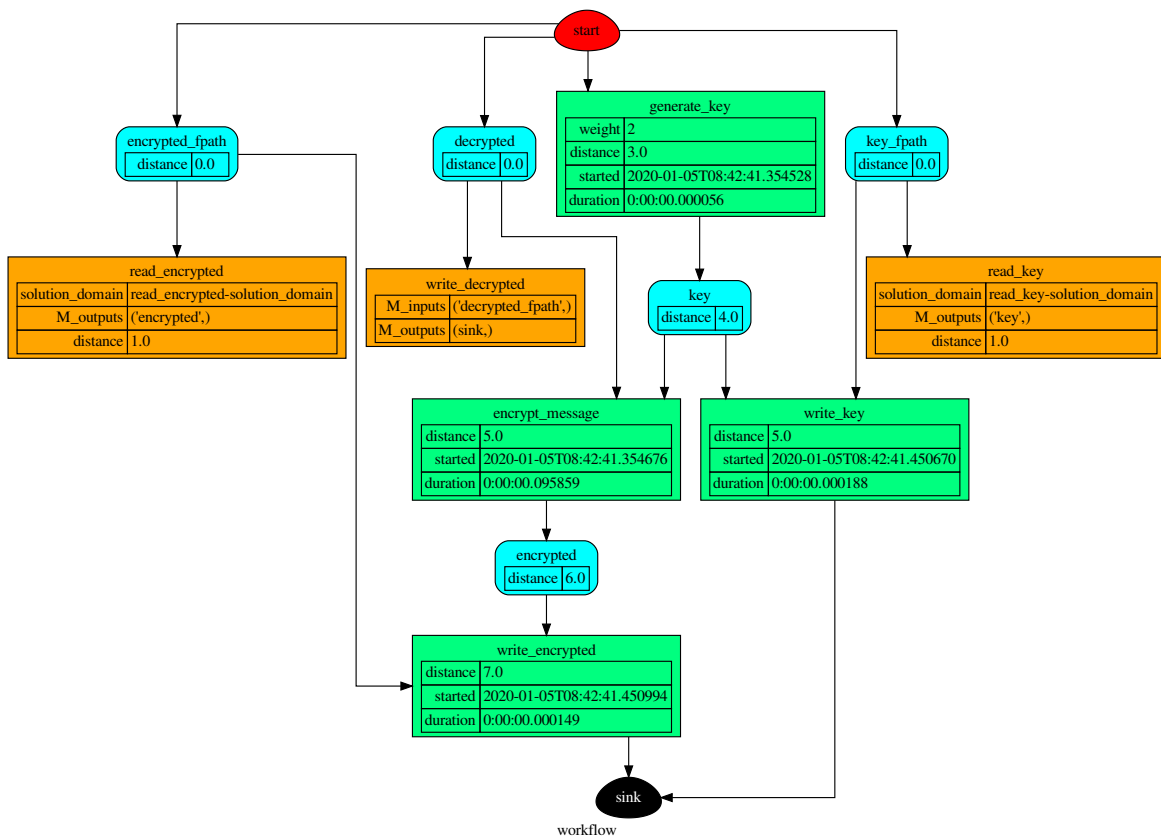


Tip: You can explore the diagram by clicking on it.

3.2 Dispatching

To see the dataflow execution model in action and its workflow to generate a key, to encrypt a message, and to write the encrypt data, you can simply invoke `dispatch()` or `__call__()` methods of the `dsp`:

```
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
>>> message = "secret message"
>>> sol = dsp(inputs=dict(
...     decrypted=message,
...     encrypted_fpath=osp.join(tempdir, 'data.secret'),
...     key_fpath=osp.join(tempdir, 'key.key')
... ))
>>> sol.plot(index=True) # doctest: +SKIP
```



Note: As you can see from the workflow graph (orange nodes), when some function's inputs does not respect its domain, the Dispatcher automatically finds an alternative path to estimate all computable outputs. The same logic applies when there is a function failure.

Now to decrypt the data and verify the message without saving the decrypted message, you just need to execute again the `dsp` changing the `inputs` and setting the desired `outputs`. In this way, the dispatcher automatically selects and executes only a sub-part of the dataflow execution model.

```
>>> dsp(
...     inputs=sh.selector(('encrypted_fpath', 'key_fpath'), sol),
...     outputs=['decrypted']
... )['decrypted'] == message
True
```

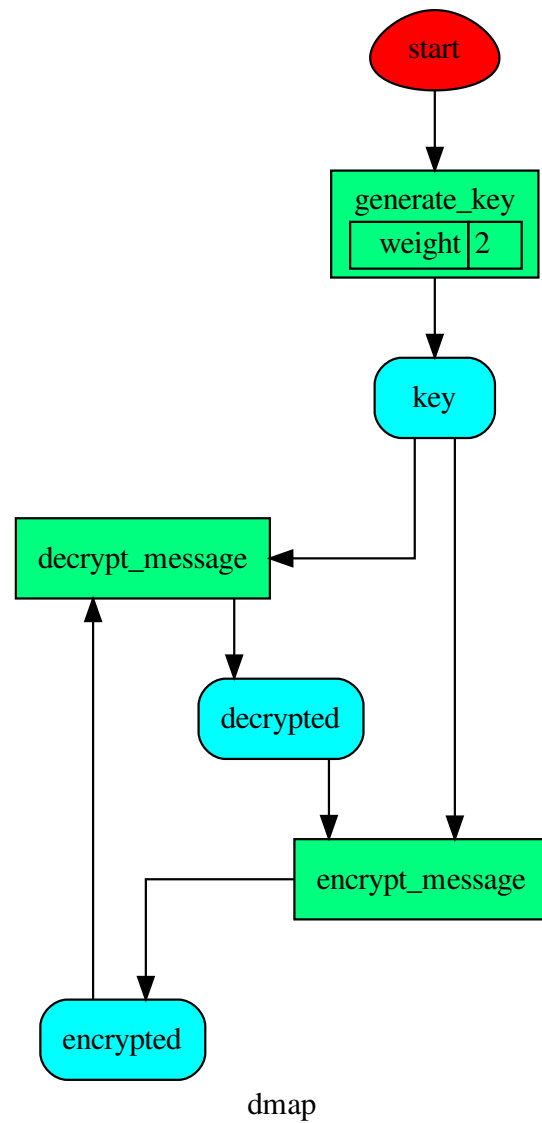
If you want to visualize the latest workflow of the dispatcher, you can use the `plot()` method with the keyword `workflow=True`:

```
>>> dsp.plot(workflow=True, index=True) # doctest: +SKIP
```

3.3 Sub-model extraction

A good security practice, when design a light web API service, is to avoid the unregulated access to the system's reading and writing features. Since our current dataflow execution model exposes these functionality, we need to extract sub-model without read/write of key and message functions:

```
>>> api = dsp.get_sub_dsp((
...     'decrypt_message', 'encrypt_message', 'key', 'encrypted',
...     'decrypted', 'generate_key', sh.START
... ))
```



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

3.4 API server

Now that the `api` model is secure, we can deploy our web API service. **schedula** allows to convert automatically a `Dispatcher` to a web API service using the `web()` method. By default, it exposes the `dispatch()` method of the `Dispatcher` and maps all its functions and sub-dispatchers. Each of these APIs are commonly called endpoints.

You can launch the server with the code below:

```
>>> server = api.web().site(host='127.0.0.1', port=5000).run()
>>> url = server.url; url
'http://127.0.0.1:5000'
```

Note: When server object is garbage collected, the server shutdowns automatically. To force the server shutdown, use its method `server.shutdown()`.

Once the server is running, you can try out the encryption functionality making a JSON POST request, specifying the *args* and *kwargs* of the `dispatch()` method, as follow:

```
>>> import requests
>>> res = requests.post(
...     'http://127.0.0.1:5000', json={'args': [{'decrypted': 'message'}]}
... ).json()
```

Note: By default, the server returns a JSON response containing the function results (i.e., 'return') or, in case of server code failure, it returns the 'error' message.

To validate the encrypted message, you can directly invoke the decryption function as follow:

```
>>> res = requests.post(
...     '%s/symmetric_cryptography/decrypt_message?data=input,return' % url,
...     json={'kwargs': sh.selector(('key', 'encrypted'), res['return'])}
... ).json(); sorted(res)
['input', 'return']
>>> res['return'] == 'message'
True
```

Note: The available endpoints are formatted like:

- `/ or /{dsp_name}`: calls the `dispatch()` method,
- `/ {dsp_name} / {function_id}`: invokes the relative function.

There is an optional query param `data=input, return`, to include the inputs into the server JSON response and exclude the possible error message.

Asynchronous and Parallel dispatching

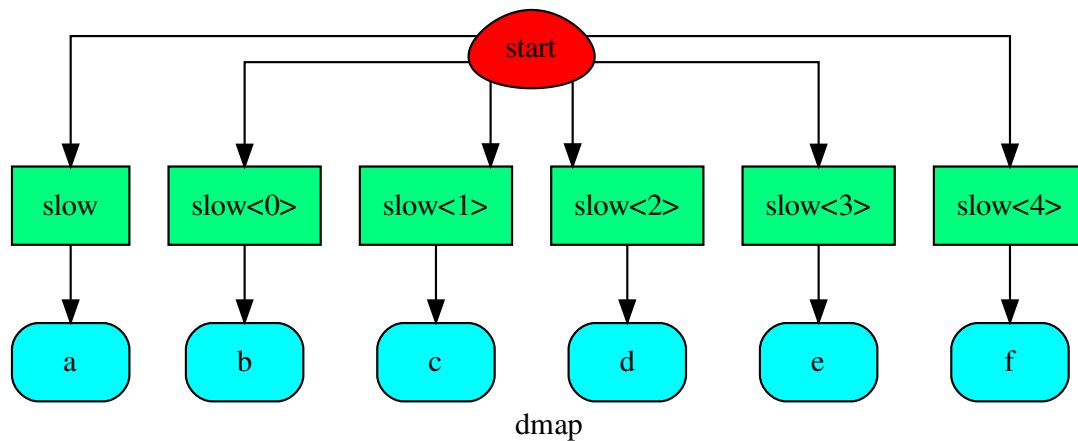
When there are heavy calculations which takes a significant amount of time, you want to run your model asynchronously or in parallel. Generally, this is difficult to achieve, because it requires an higher level of abstraction and a deeper knowledge of python programming and the Global Interpreter Lock (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])
...'
```



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 on 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 on different processes:

```

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

```

Contributing to schedula

If you want to contribute to **schedula** and make it better, your help is very welcome. The contribution should be sent by a *pull request*. Next sections will explain how to implement and submit a new functionality:

- clone the repository
- implement a new functionality
- open a pull request

5.1 Clone the repository

The first step to contribute to **schedula** is to clone the repository:

- Create a personal **fork** of the **schedula** repository on Github.
- **Clone** the fork on your local machine. Your remote repo on Github is called `origin`.
- **Add** the original repository as a remote called `upstream`, to maintain updated your fork.
- If you created your fork a while ago be sure to **pull** `upstream` changes into your local repository.
- Create a new branch to work on! Branch from `dev`.

5.2 How to implement a new functionality

Test cases are very important. This library uses a data-driven testing approach. To implement a new function I recommend the **test-driven development cycle**. Hence, when you think that the code is ready, add new test in `test` folder.

When all test cases are ok (`python setup.py test`), open a pull request.

Note: A pull request without new test case will not be taken into consideration.

5.3 How to open a pull request

Well done! Your contribution is ready to be submitted:

- Squash your commits into a single commit with git's `interactive rebase`. Create a new branch if necessary. Always write your commit messages in the present tense. Your commit message should describe what the commit, when applied, does to the code – not what you did to the code.
- `Push` your branch to your fork on Github (i.e., `git push origin dev`).
- From your fork `open` a *pull request* in the correct branch. Target the project's dev branch!
- Once the *pull request* is approved and merged you can pull the changes from `upstream` to your local repo and delete your extra branch(es).

CHAPTER 6

Donate

If you want to [support](#) the **schedula** development please donate.

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:

<i>dispatcher</i>	It provides Dispatcher class.
<i>utils</i>	It contains utility classes and functions.
<i>ext</i>	It provides sphinx extensions.

7.1 dispatcher

It provides Dispatcher class.

Classes

<i>Dispatcher</i>	It provides a data structure to process a complex system of functions.
-------------------	--

7.1.1 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:

$$b - a = c$$

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

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

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

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.
...     '''
```

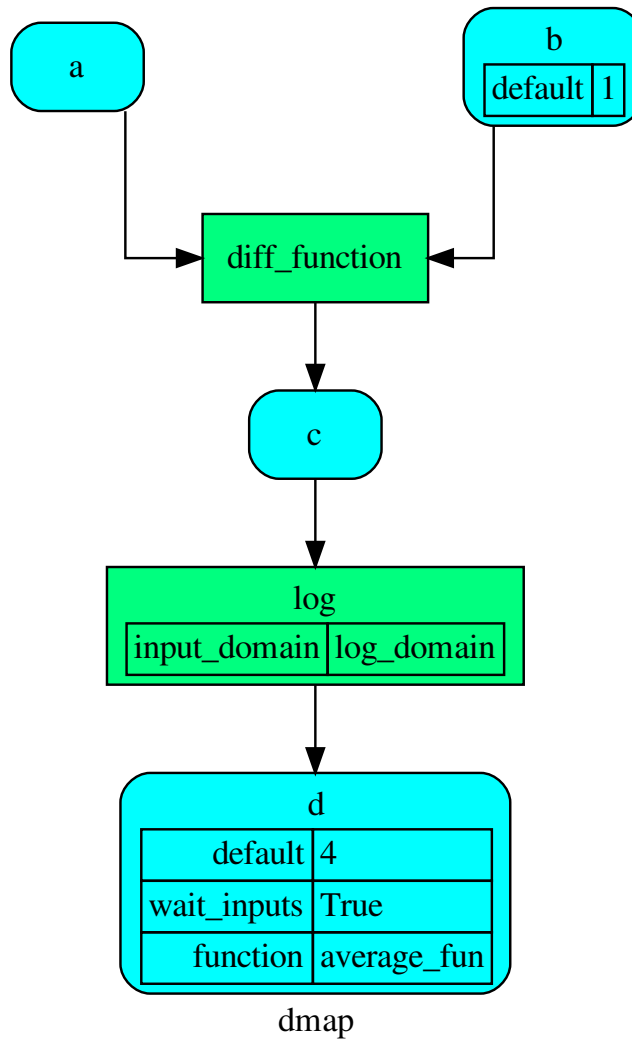
(continues on next page)

(continued from previous page)

```

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

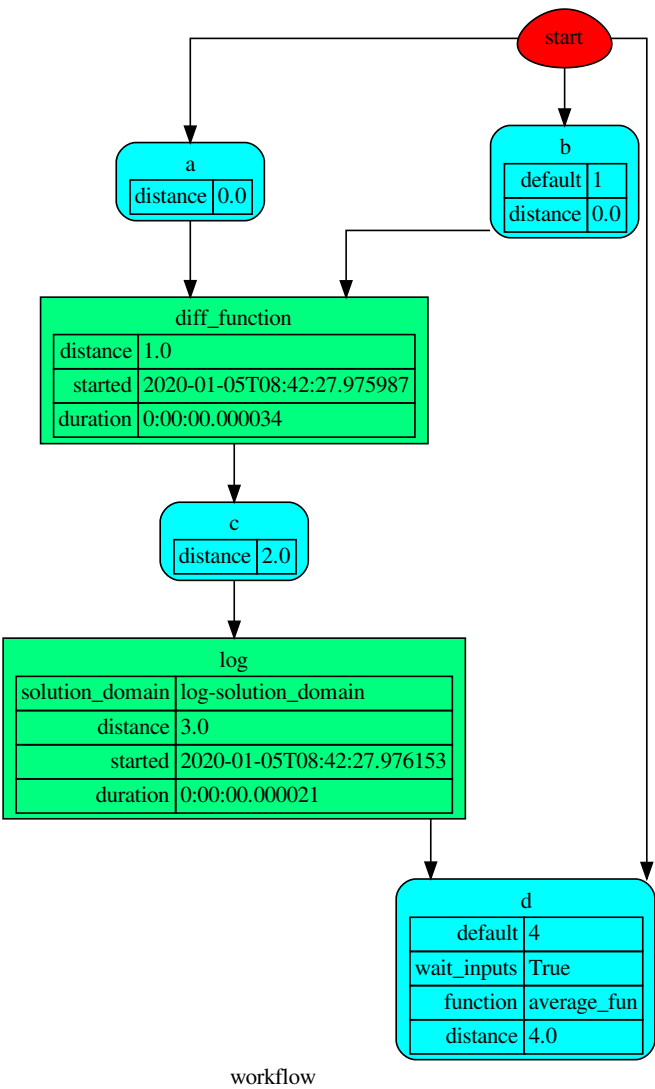
```



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

```

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



Methods

<code>__init__</code>	Initializes the dispatcher.
<code>add_data</code>	Add a single data node to the dispatcher.
<code>add_dispatcher</code>	Add a single sub-dispatcher node to dispatcher.
<code>add_from_lists</code>	Add multiple function and data nodes to dispatcher.
<code>add_func</code>	Add a single function node to dispatcher.
<code>add_function</code>	Add a single function node to dispatcher.
<code>blue</code>	Constructs a BlueDispatcher out of the current object.
<code>copy</code>	Returns a deepcopy of the Dispatcher.
<code>copy_structure</code>	Returns a copy of the Dispatcher structure.

Continued on next page

Table 3 – continued from previous page

<code>dispatch</code>	Evaluates the minimum workflow and data outputs of the dispatcher model from given inputs.
<code>extend</code>	Extends Dispatcher calling each deferred operation of given Blueprints.
<code>get_node</code>	Returns a sub node of a dispatcher.
<code>get_sub_dsp</code>	Returns the sub-dispatcher induced by given node and edge bunches.
<code>get_sub_dsp_from_workflow</code>	Returns the sub-dispatcher induced by the workflow from sources.
<code>plot</code>	Plots the Dispatcher with a graph in the DOT language with Graphviz.
<code>set_default_value</code>	Set the default value of a data node in the dispatcher.
<code>shrink_dsp</code>	Returns a reduced dispatcher.
<code>web</code>	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|str*, *optional*) – If `True` the dispatcher interrupt the dispatch when an error occur, otherwise if `raises != ""` 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',
```

(continues on next page)

(continued from previous page)

```
...         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.

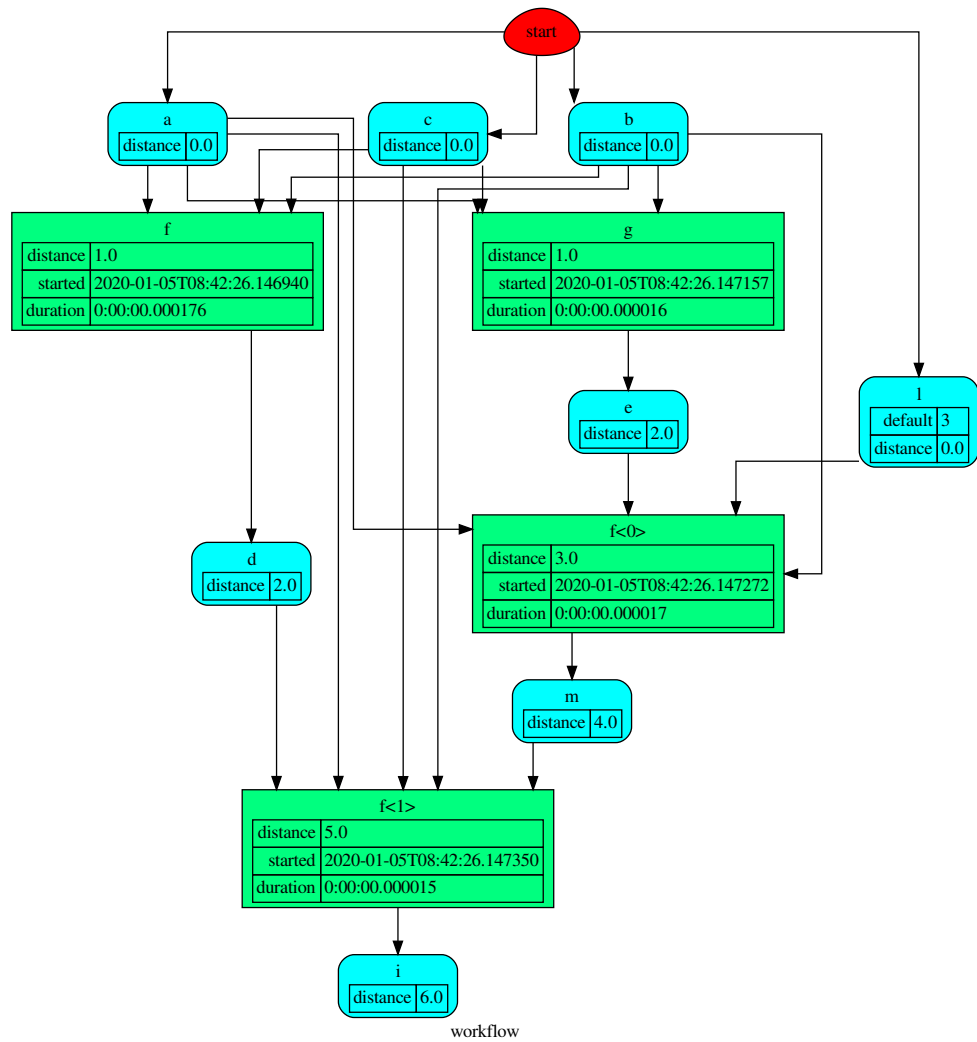
Return type `str`

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)
...
>>> 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 deepcopy 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*)

Returns a copy of the Dispatcher structure.

Parameters `kwargs` (*dict*) – Additional parameters to initialize the new class.

Returns A copy of the Dispatcher structure.

Return type *Dispatcher*

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()

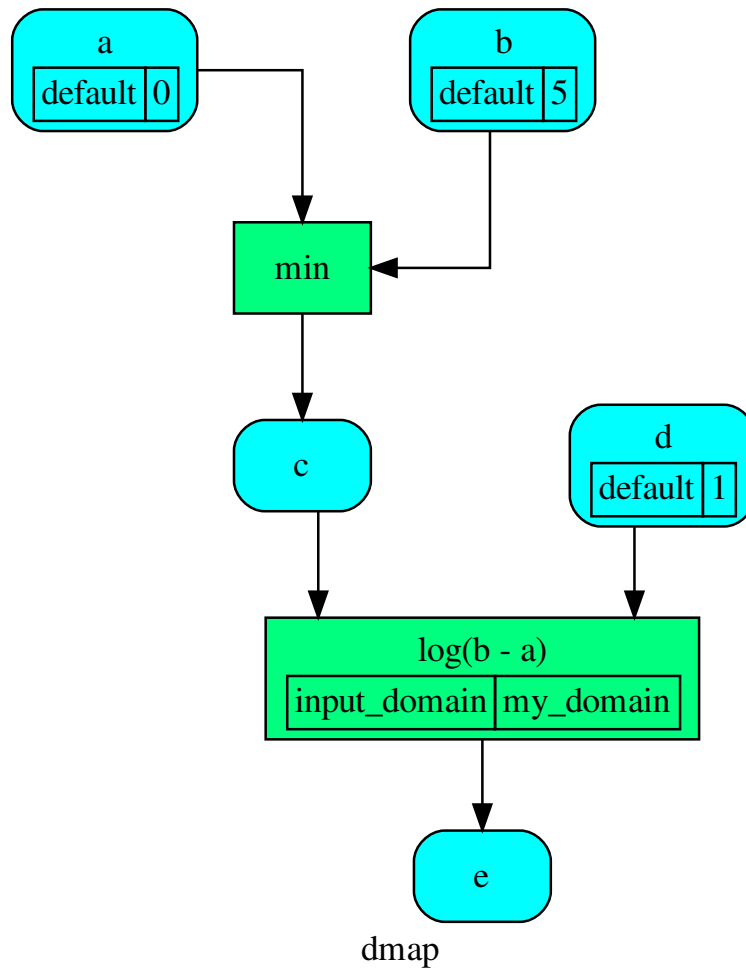
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **select_output_kw** (*dict, optional*) – Kwargs of selector function to select specific outputs.
- **_wait_in** (*dict, optional*) – Override wait inputs.
- **stopper** (*multiprocess.Event,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

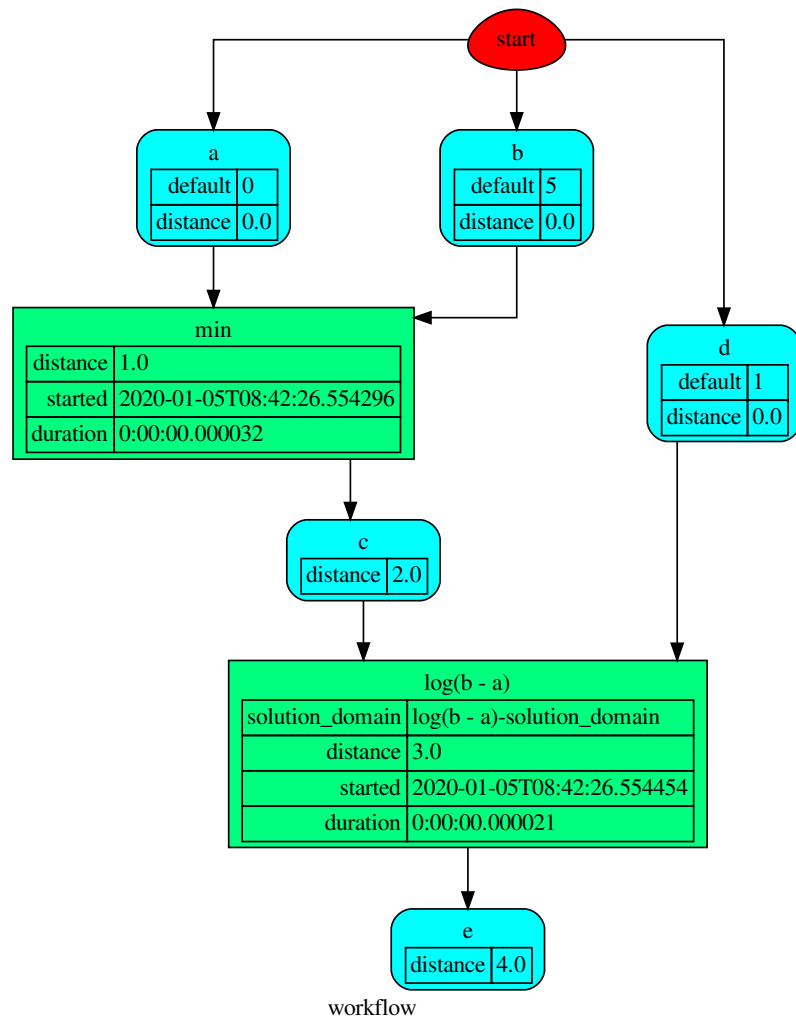
A dispatcher with a function $\log(b - a)$ and two data a and b with default values:



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)])
  
```

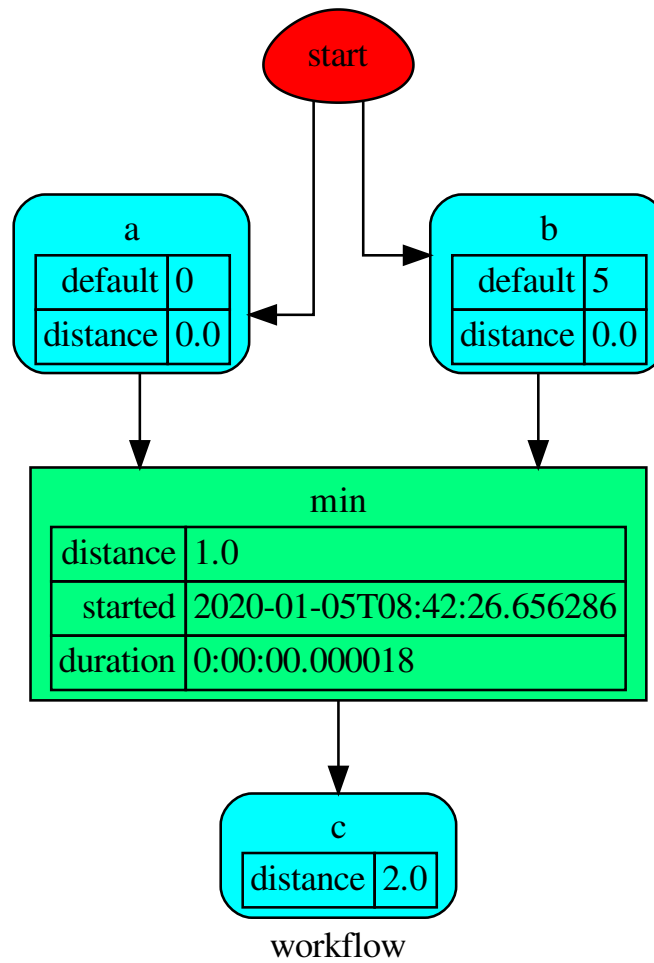


Dispatch until data node *c* is estimated:

```

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

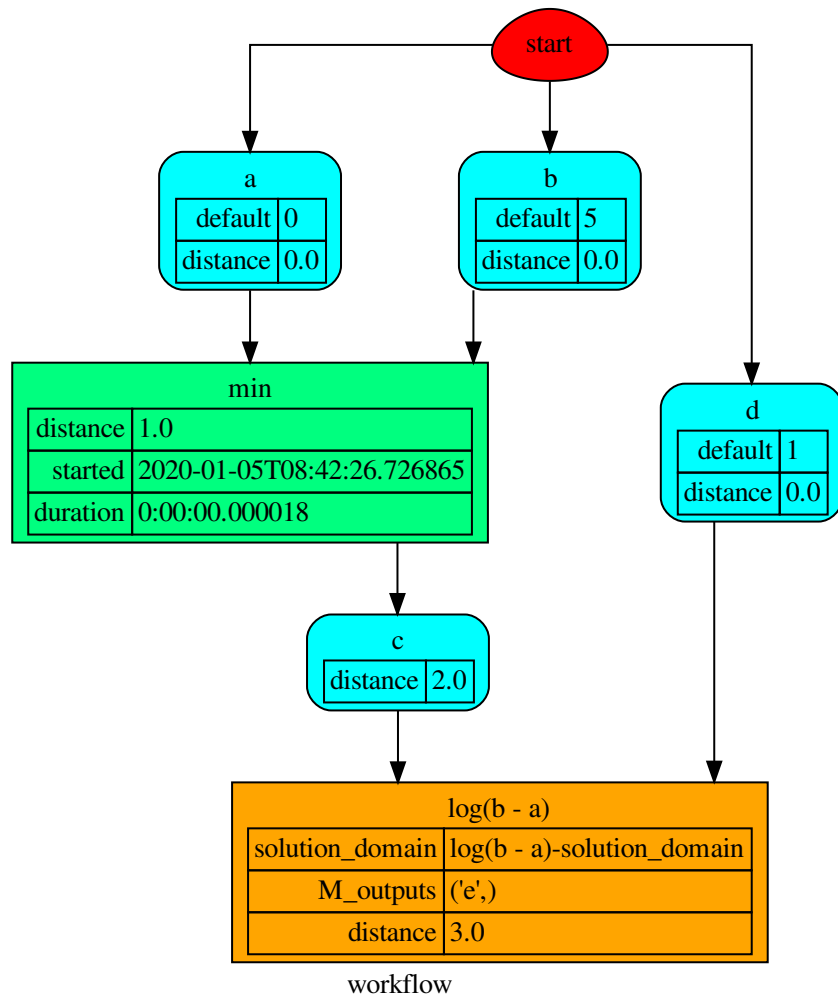
```



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)])
  
```



extend

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

Extends Dispatcher calling each deferred operation of given Blueprints.

Parameters

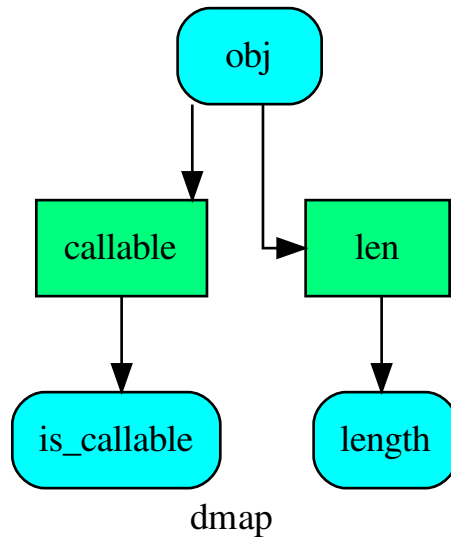
- **blues** (*Blueprint | schedula.dispatcher.Dispatcher*) – Blueprints or Dispatchers to extend deferred operations.
- **memo** (*dict[T, schedula.utils.blue.Blueprint | Dispatcher]*) – A dictionary to cache Blueprints and Dispatchers.

Returns Self.

Return type *Dispatcher*

Example:

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



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.

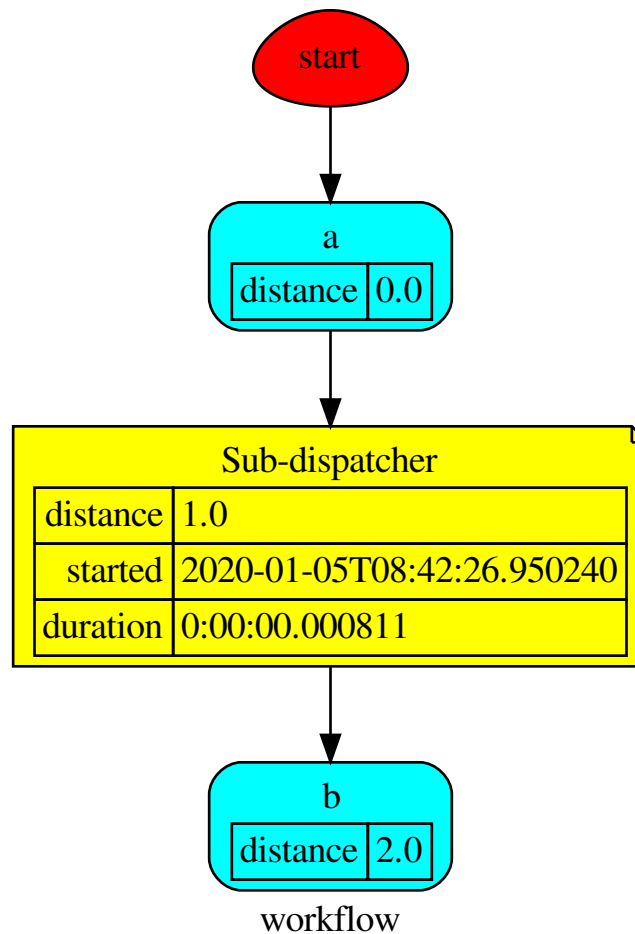
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

Example:



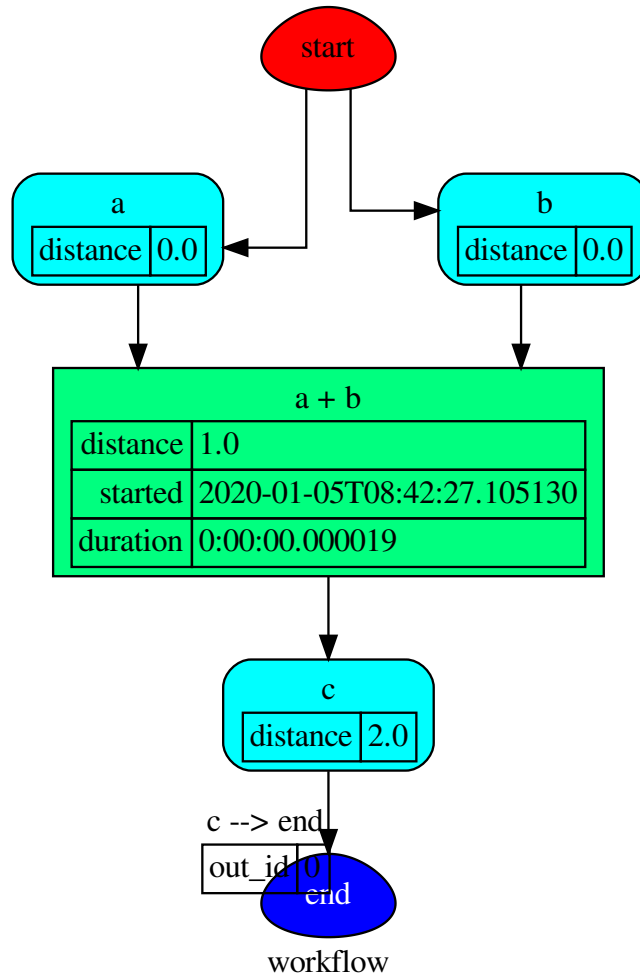
Get the sub node output:

```

>>> 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*

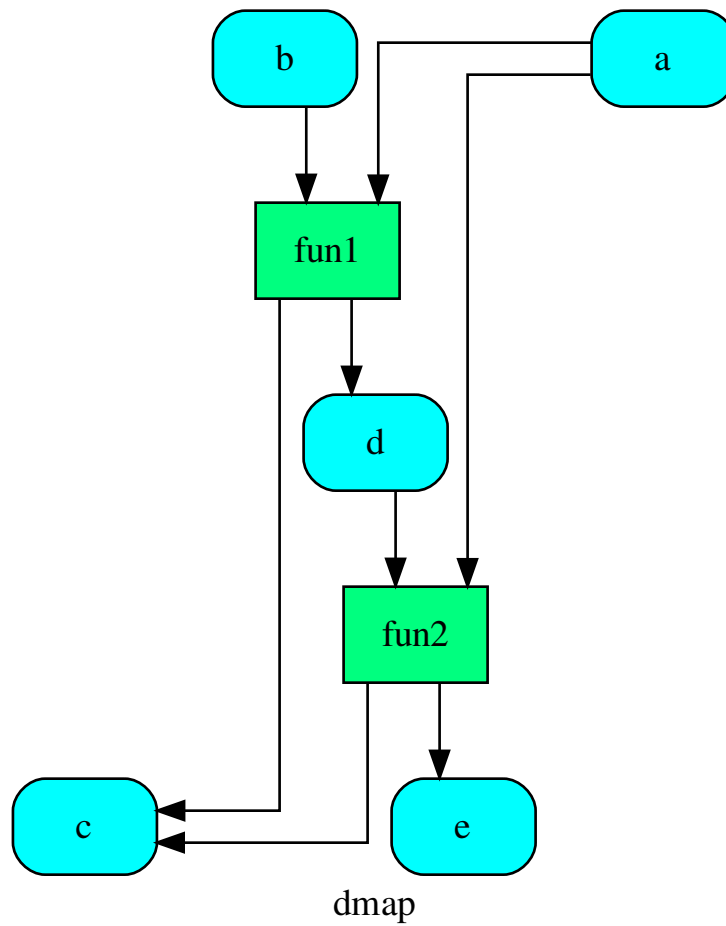
See also:

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.

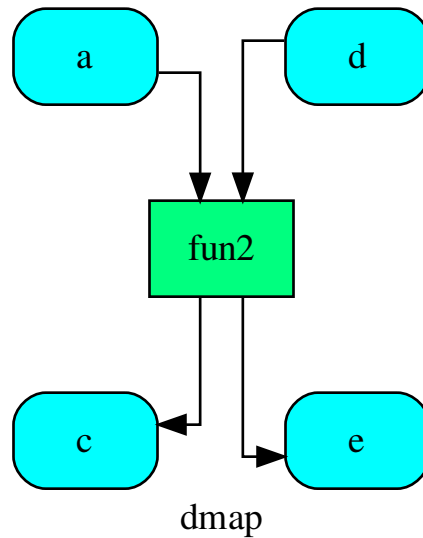
Example:

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*

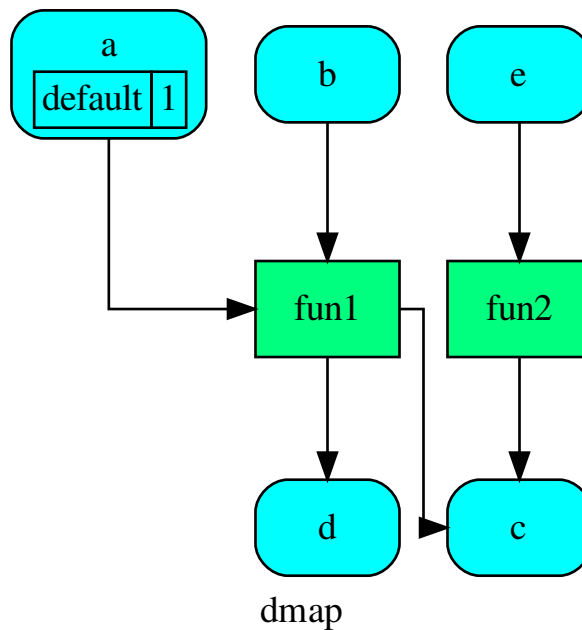
See also:

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:

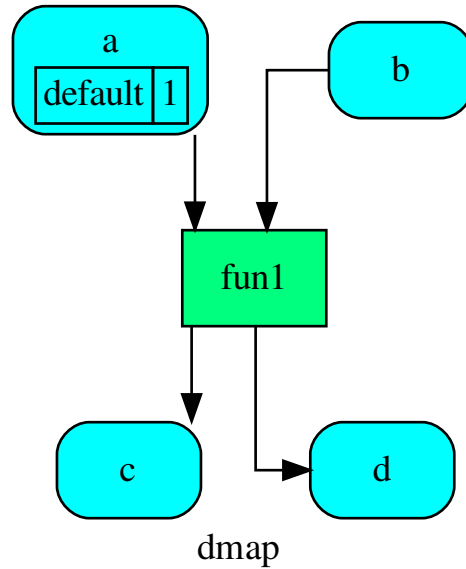


Dispatch with no calls in order to have a workflow:

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

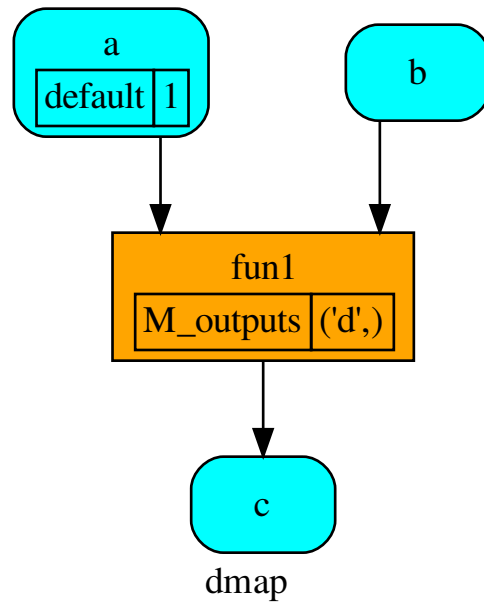
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> 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.

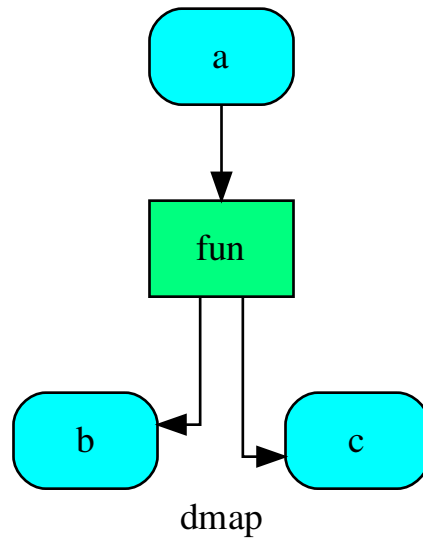
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str*, *optional*) – (Sub)directory for source saving and rendering.
- **format** (*str*, *optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str*, *optional*) – Layout command used ('dot', 'neato',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



set_default_value

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

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

Parameters

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

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

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

Example:

A dispatcher with a data node named *a*:

```

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

```

Add a default value to *a* node:

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

Remove the default value of *a* node:

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

shrink_dsp

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

Returns a reduced dispatcher.

Parameters

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

Returns A sub-dispatcher.

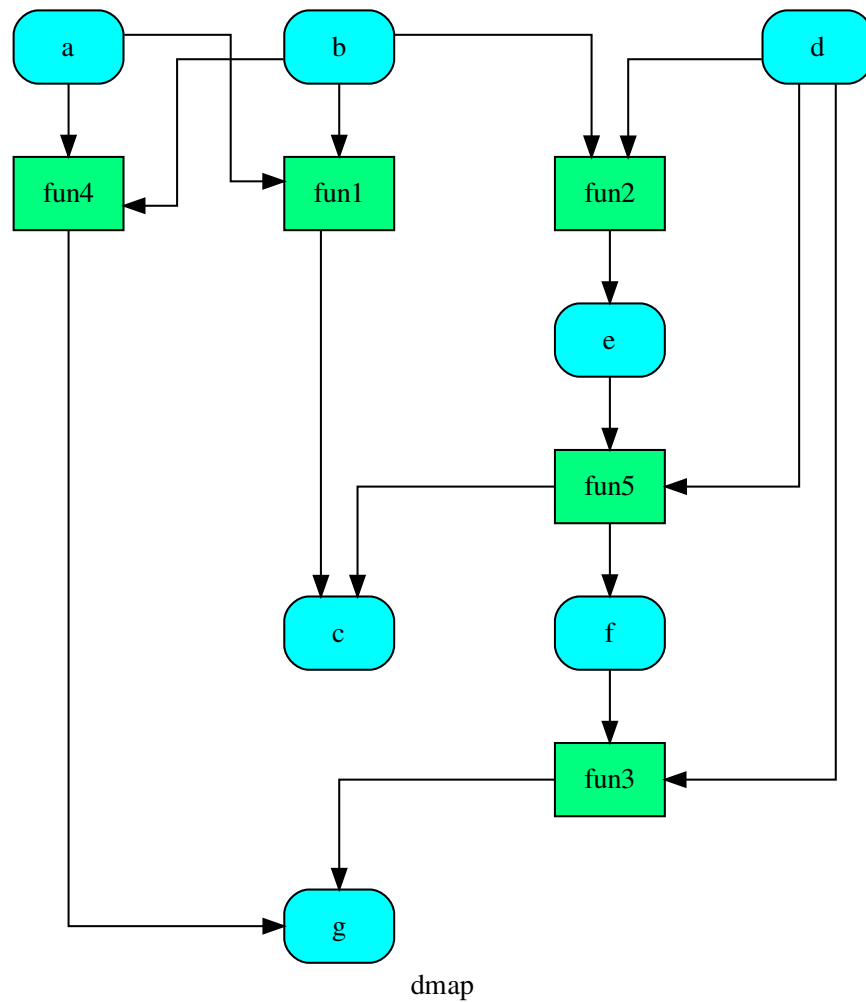
Return type *Dispatcher*

See also:

dispatch()

Example:

A dispatcher like this:

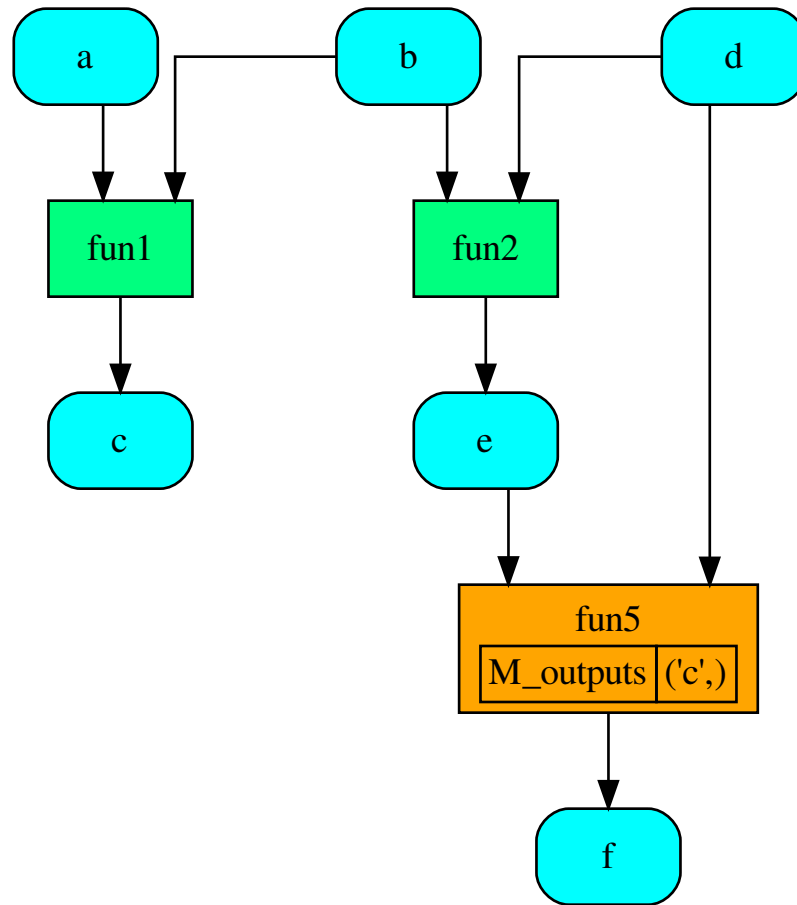


Get the sub-dispatcher induced by dispatching with no calls from inputs *a*, *b*, and *c* to outputs *c*, *e*, and *f*:

```

>>> 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 back-end server.
- **run** (*bool*, *optional*) – Run the backend server?

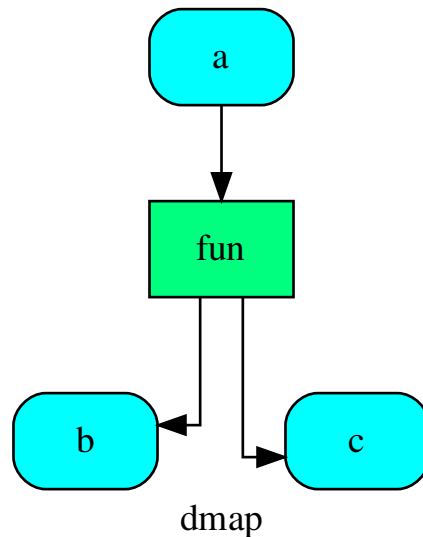
Returns A WebMap.

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/str*, *optional*) – If *True* the dispatcher interrupt the dispatch when an error occur, otherwise if *raises != ""* 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

<i>data_nodes</i>	Returns all data nodes of the dispatcher.
<i>function_nodes</i>	Returns all function nodes of the dispatcher.
<i>sub_dsp_nodes</i>	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.

copy_structure (***kwargs*)

Returns a copy of the Dispatcher structure.

Parameters **kwargs** (*dict*) – Additional parameters to initialize the new class.

Returns A copy of the Dispatcher structure.

Return type *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:
...         '''
```

(continues on next page)

(continued from previous page)

```

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

```

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

```

>>> dsp.add_data()
'unknown'

```

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

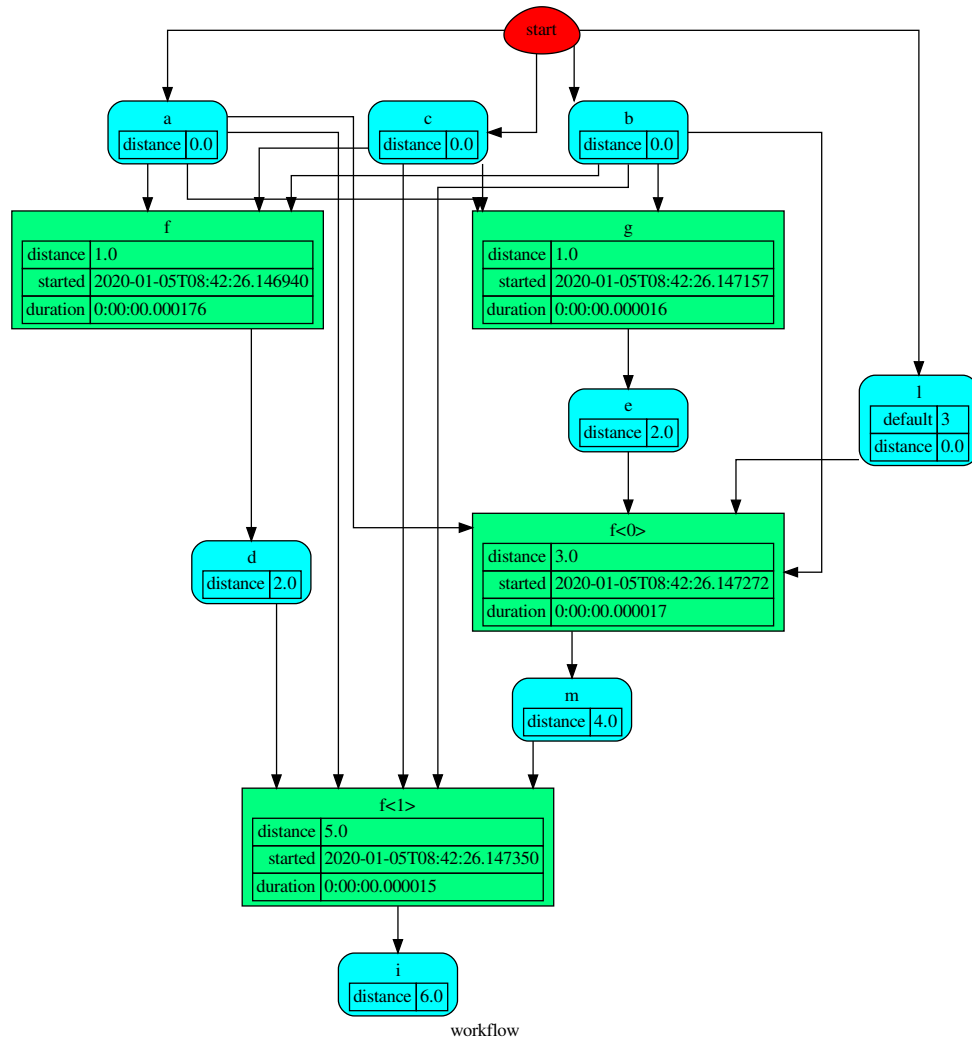
Example:

```
>>> 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'
```

(continues on next page)

(continued from previous page)

```
>>> 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_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
... 
```

(continues on next page)

(continued from previous page)

```
>>> 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*

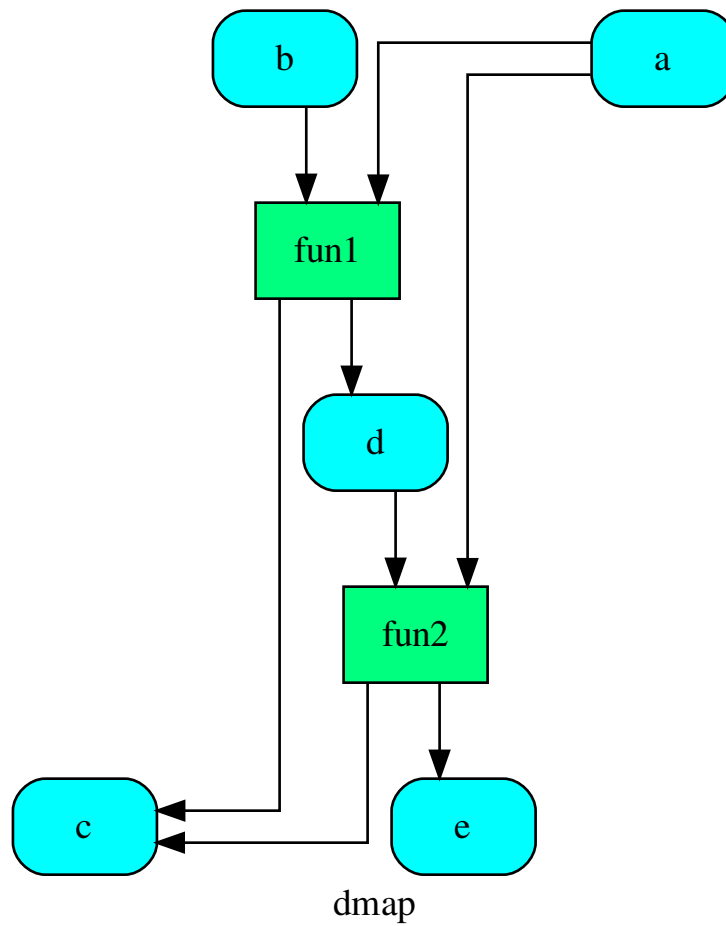
See also:

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.

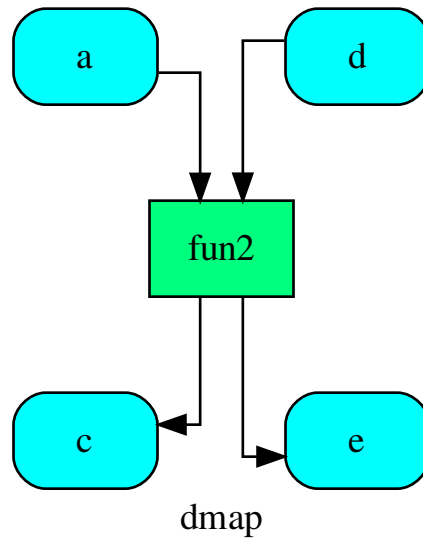
Example:

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*

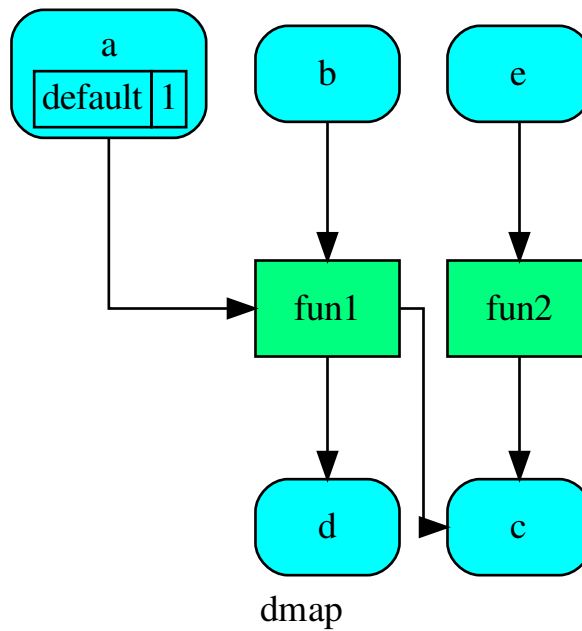
See also:

`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:

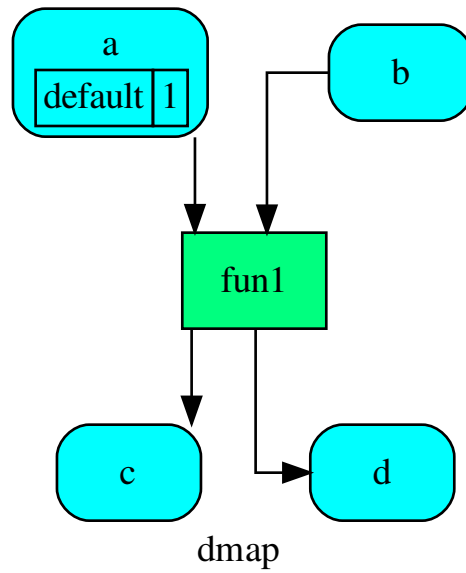


Dispatch with no calls in order to have a workflow:

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

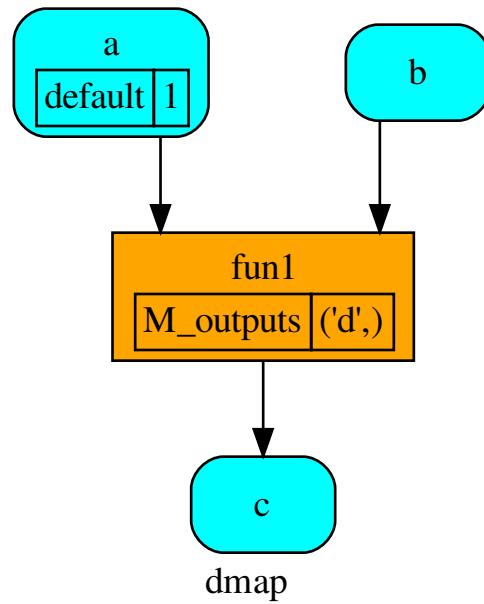
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> 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 deepcopy 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

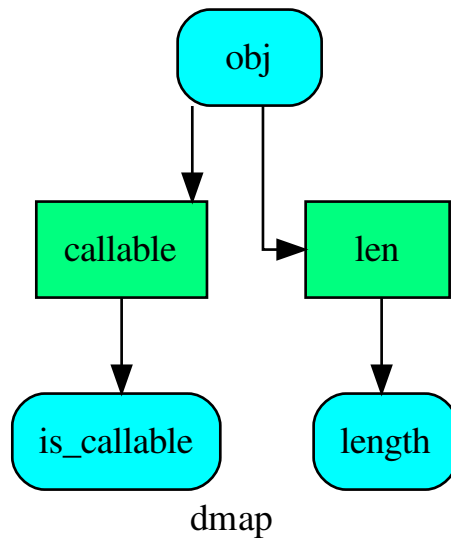
- **blues** (*Blueprint* | *schedula.dispatcher.Dispatcher*) – Blueprints or Dispatchers to extend deferred operations.
- **memo** (*dict* [*T*, *schedula.utils.blue.Blueprint* | *Dispatcher*]) – A dictionary to cache Blueprints and Dispatchers.

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()

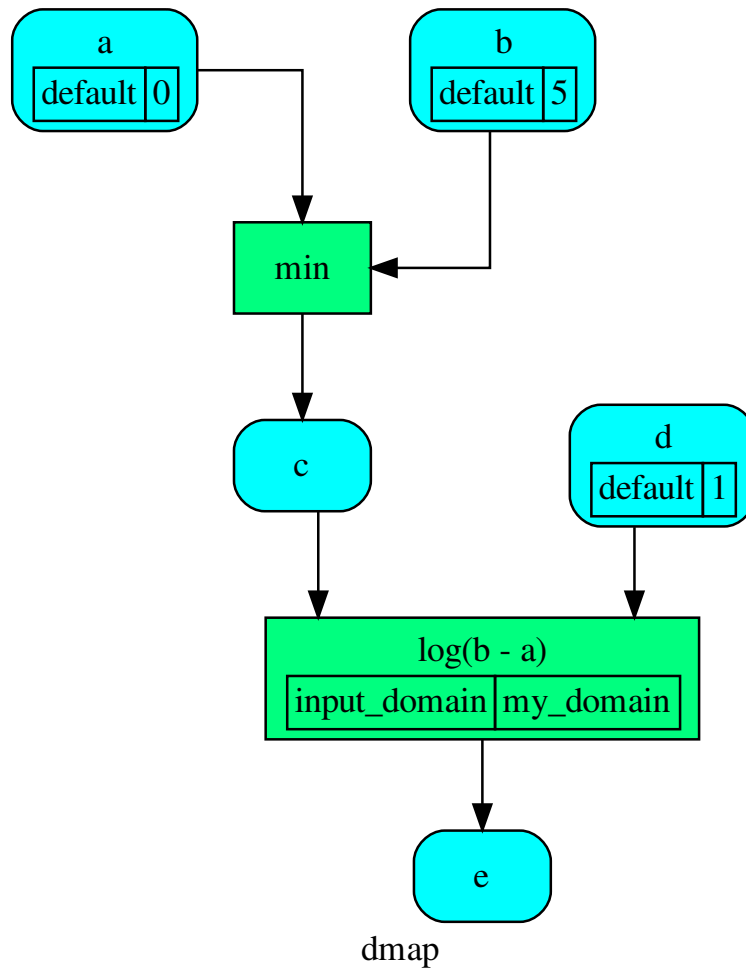
- **rm_unused_nds** (*bool, optional*) – If True unused function and sub-dispatcher nodes are removed from workflow.
- **select_output_kw** (*dict, optional*) – Kwargs of selector function to select specific outputs.
- **_wait_in** (*dict, optional*) – Override wait inputs.
- **stopper** (*multiprocess.Event,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

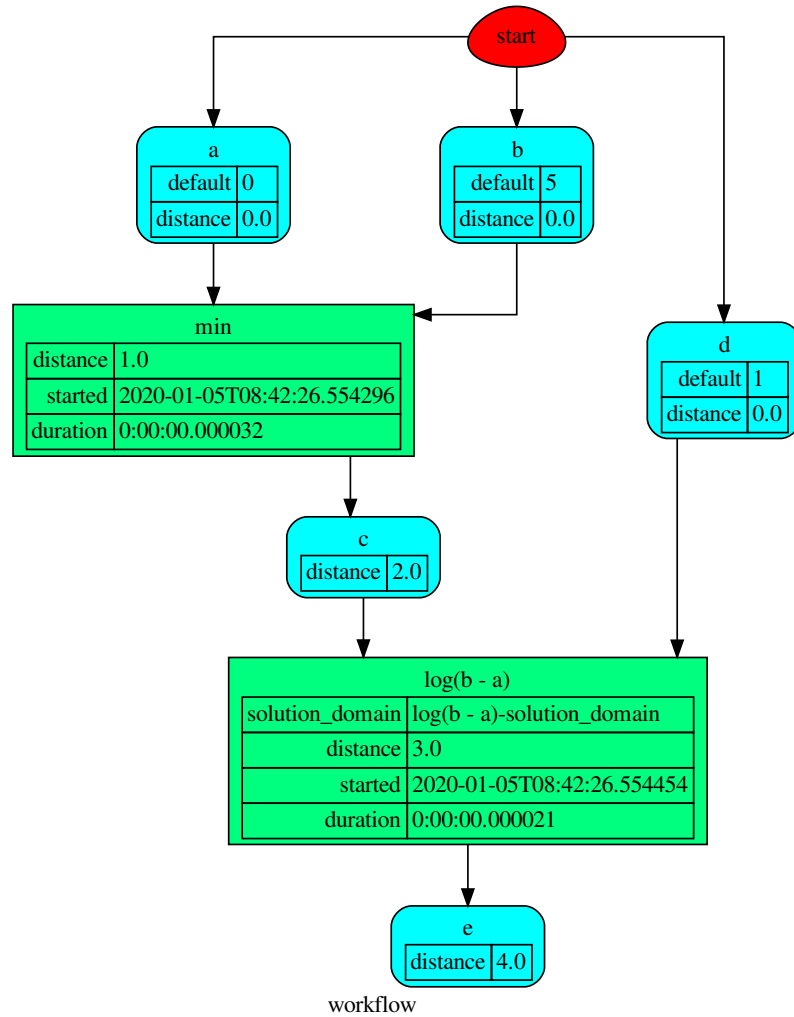
A dispatcher with a function $\log(b - a)$ and two data a and b with default values:



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)])
  
```

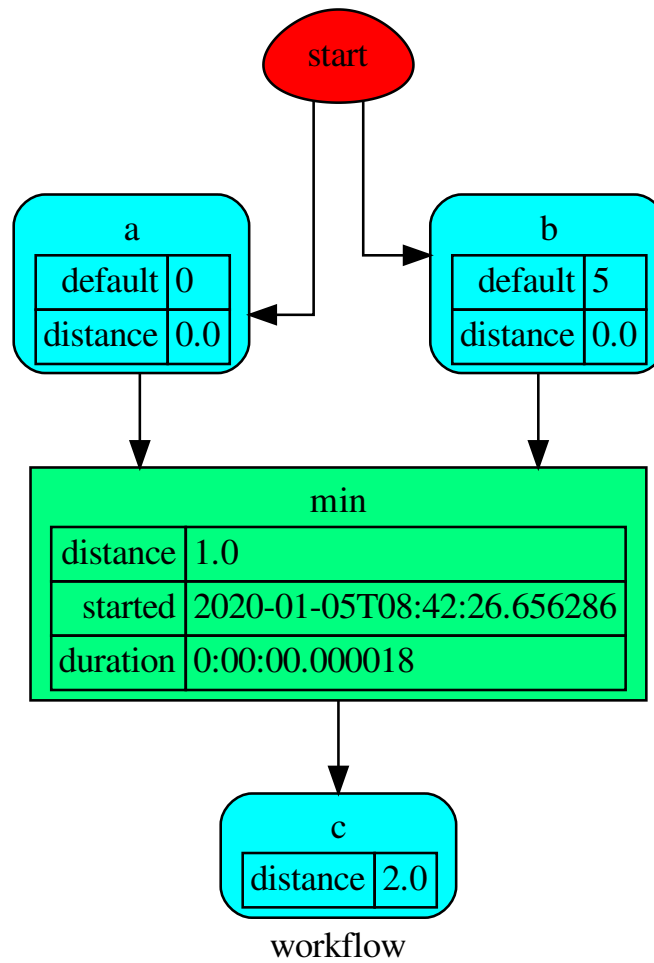


Dispatch until data node *c* is estimated:

```

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

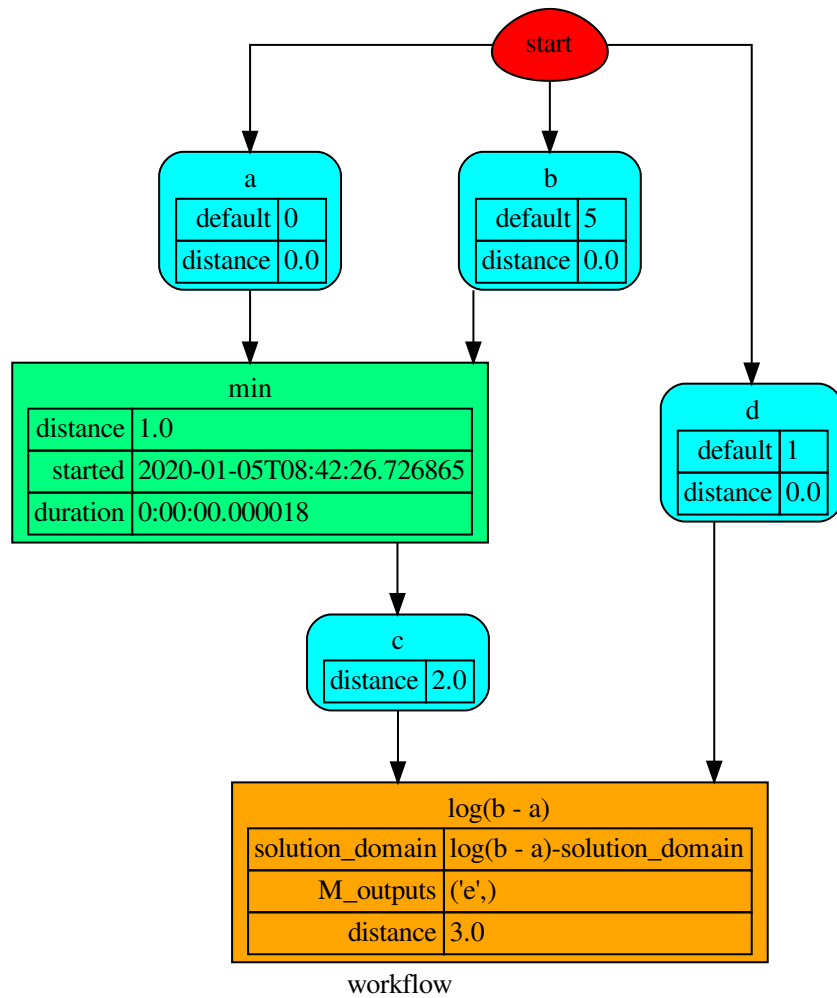
```



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.

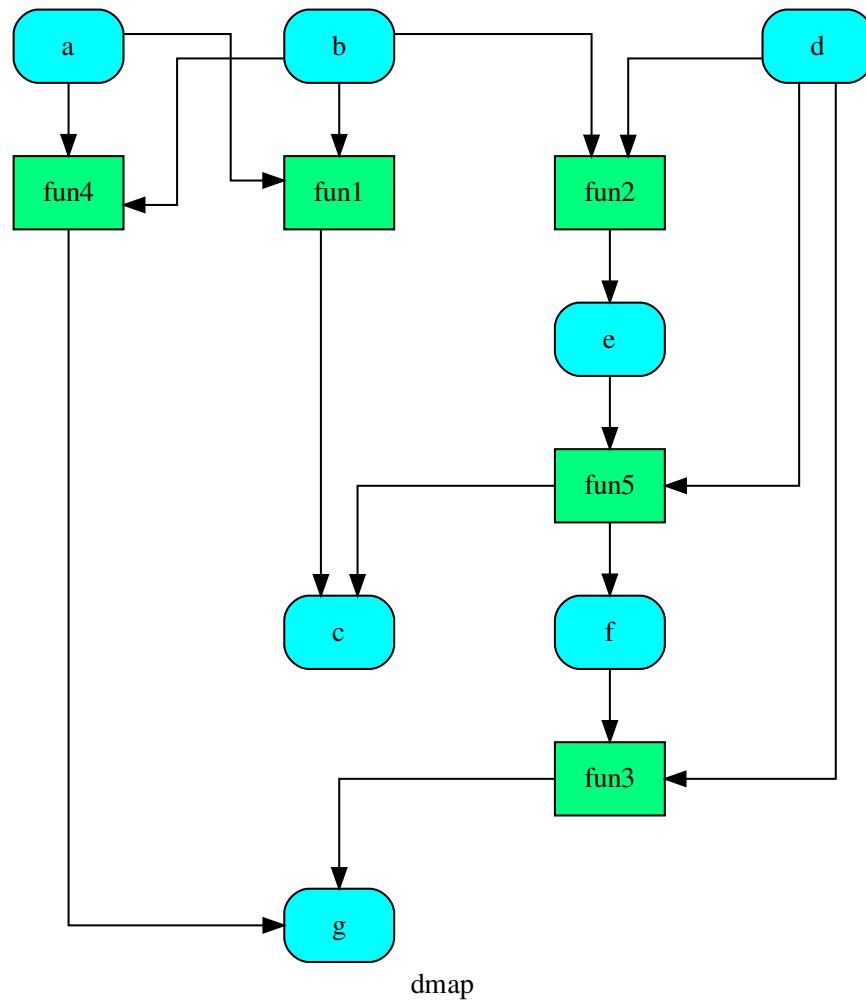
Return type *Dispatcher*

See also:

`dispatch()`

Example:

A dispatcher like this:

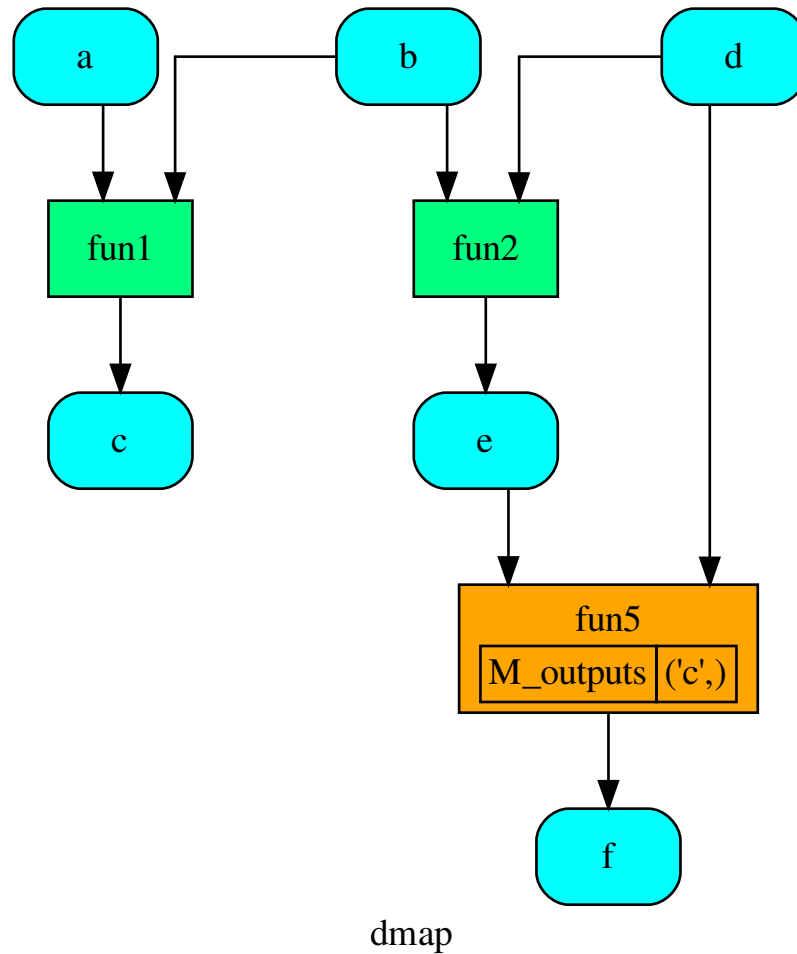


Get the sub-dispatcher induced by dispatching with no calls from inputs *a*, *b*, and *c* to outputs *c*, *e*, and *f*:

```

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

```



7.2 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:

<i>alg</i>	It contains basic algorithms, numerical tricks, and data processing tasks.
<i>asy</i>	It contains functions to dispatch asynchronously and in parallel.
<i>base</i>	It provides a base class for dispatcher objects.
<i>blue</i>	It provides a Blueprint class to construct a Dispatcher and SubDispatch objects.
<i>cst</i>	It provides constants data node ids and values.
<i>des</i>	It provides tools to find data, function, and sub-dispatcher node description.
<i>drw</i>	It provides functions to plot dispatcher map and workflow.
<i>dsp</i>	It provides tools to create models with the <i>Dispatcher</i> .
<i>exc</i>	Defines the dispatcher exception.
<i>gen</i>	It contains classes and functions of general utility.
<i>io</i>	It provides functions to read and save a dispatcher from/to files.
<i>sol</i>	It provides a solution class for dispatch result.
<i>web</i>	It provides functions to build a flask app from a dispatcher.

7.2.1 alg

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

Functions

<i>add_edge_fun</i>	Returns a function that adds an edge to the <i>graph</i> checking only the out node.
<i>add_func_edges</i>	Adds function node edges.
<i>get_full_pipe</i>	Returns the full pipe of a dispatch run.
<i>get_sub_node</i>	Returns a sub node of a dispatcher.
<i>get_unused_node_id</i>	Finds an unused node id in <i>graph</i> .
<i>remove_edge_fun</i>	Returns a function that removes an edge from the <i>graph</i> .

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.

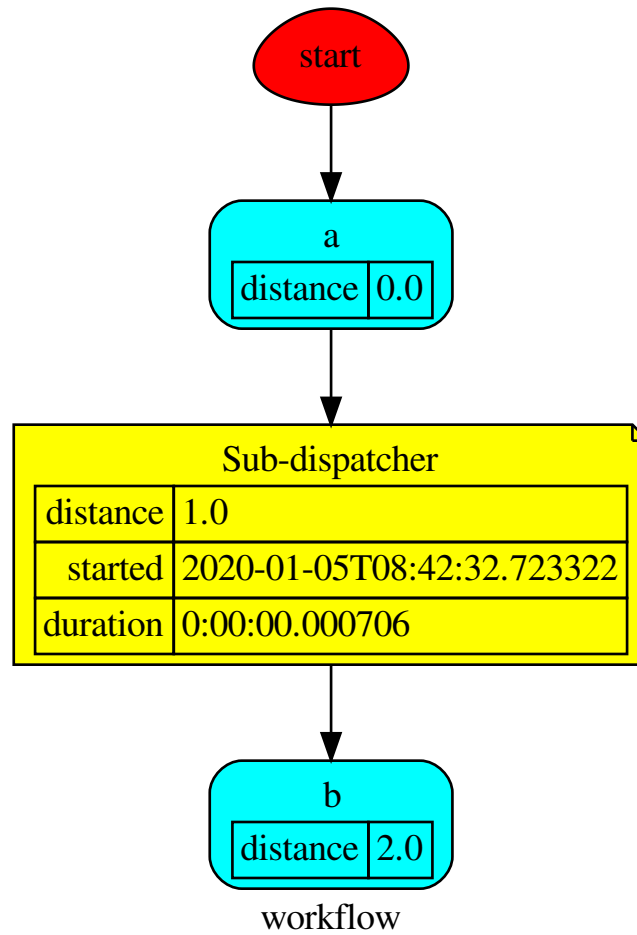
Returns A sub node of a dispatcher and its path.

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'))

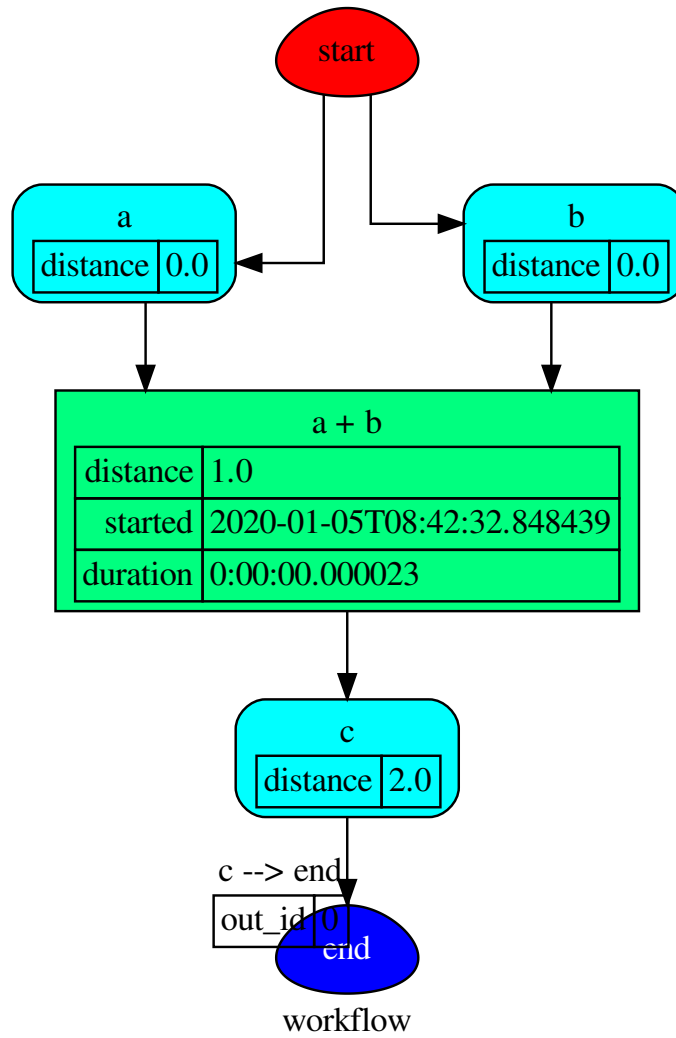
```

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.

7.2.2 asy

It contains functions to dispatch asynchronously and in parallel.

Functions

<code>async_process</code>	Execute <i>func(*args)</i> in an asynchronous parallel process.
<code>async_thread</code>	Execute <i>sol.evaluate_node</i> in an asynchronous thread.
<code>await_result</code>	Return the result of a <i>Future</i> object.
<code>register_executor</code>	Register a new executor type.
<code>shutdown_executor</code>	Clean-up the resources associated with the Executor.
<code>shutdown_executors</code>	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.

Return type `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

<i>AsyncList</i>	List of asynchronous results.
<i>Executor</i>	Base Executor
<i>PoolExecutor</i>	General PoolExecutor to dispatch asynchronously and in parallel.
<i>ProcessExecutor</i>	Multi Process Executor
<i>ProcessPoolExecutor</i>	Process Pool Executor
<i>ThreadExecutor</i>	Multi Thread Executor

AsyncList

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

List of asynchronous results.

Methods

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

`__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

<code>__init__</code>	Initialize self.
<code>shutdown</code>	
<code>submit</code>	

`__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

<code>__init__</code>	Initialize self.
<code>shutdown</code>	
<code>submit</code>	

`__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

<code>__init__</code>	Initialize self.
<code>shutdown</code>	
<code>submit</code>	

`__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

<code>__init__</code>	Initialize self.
<code>shutdown</code>	
<code>submit</code>	

`__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.

7.2.3 base

It provides a base class for dispatcher objects.

Classes

<i>Base</i>	Base class for dispatcher objects.
-------------	------------------------------------

Base

class Base

Base class for dispatcher objects.

Methods

<i>get_node</i>	Returns a sub node of a dispatcher.
<i>plot</i>	Plots the Dispatcher with a graph in the DOT language with Graphviz.
<i>web</i>	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.

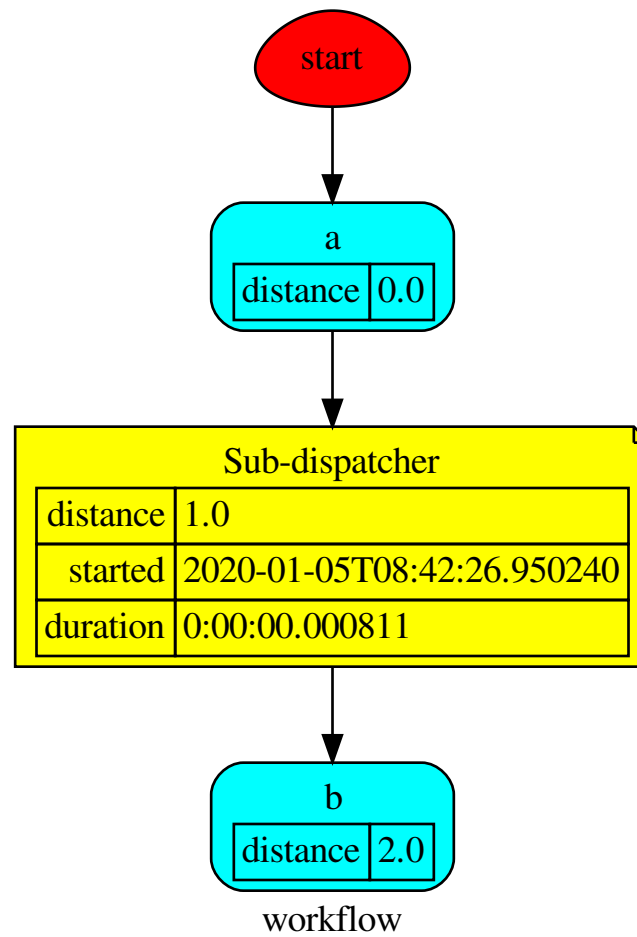
When ‘value_type’, returns the data node value’s type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

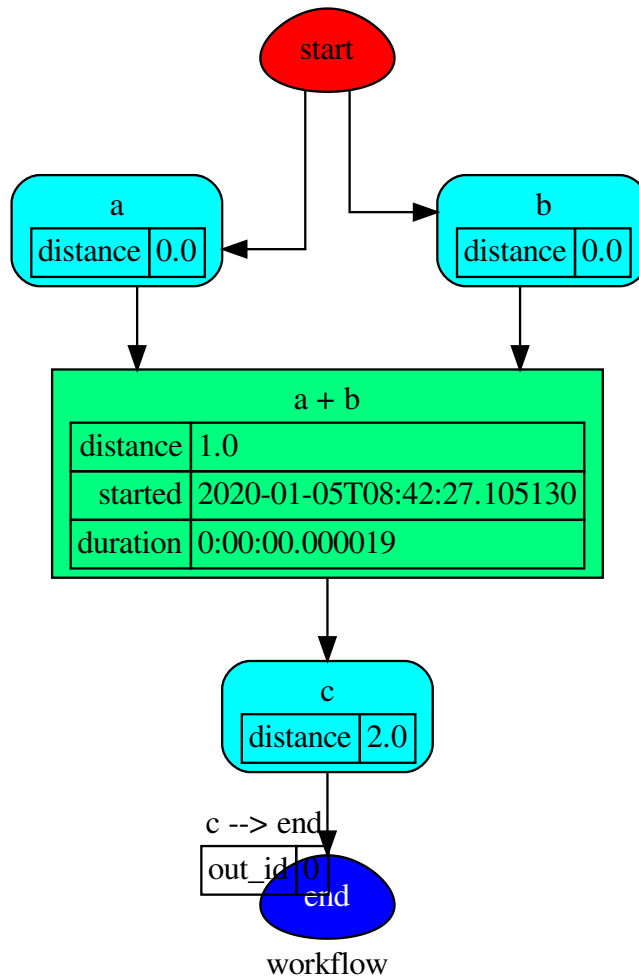
Example:



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.

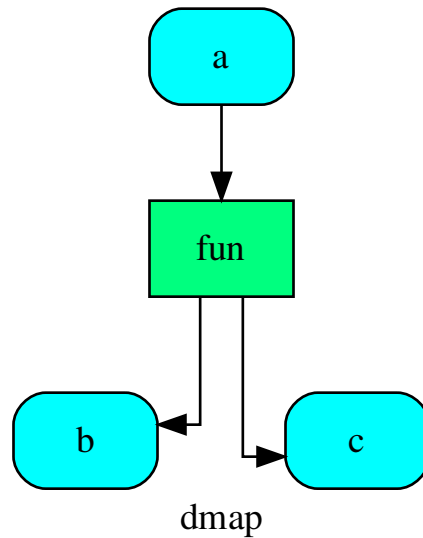
- **node_data** (*tuple[str], optional*) – Data node attributes to view.
- **node_function** (*tuple[str], optional*) – Function node attributes to view.
- **node_styles** (*dict[str|Token, dict[str, str]]*) – Default node styles according to graphviz node attributes.
- **depth** (*int, optional*) – 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 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())])
```



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 back-end server.
- **run** (*bool*, *optional*) – Run the backend server?

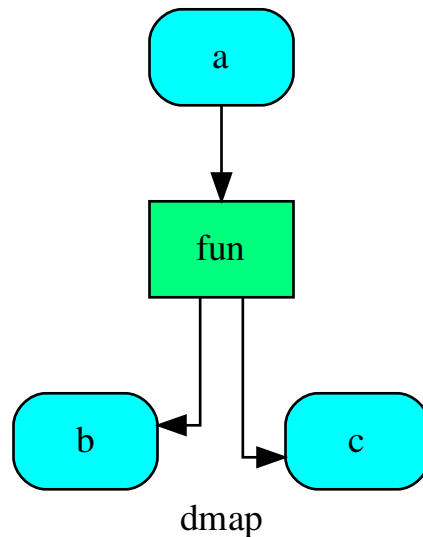
Returns A WebMap.

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 back-end server.
- **run** (*bool*, *optional*) – Run the backend server?

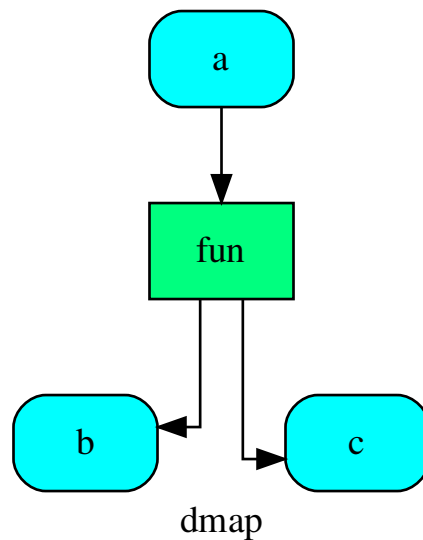
Returns A WebMap.

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.

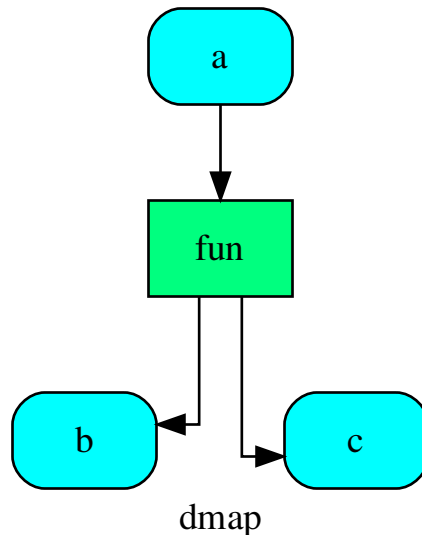
- **body** (*dict*, *optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the back-end server.
- **index** (*bool*, *optional*) – Add the site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

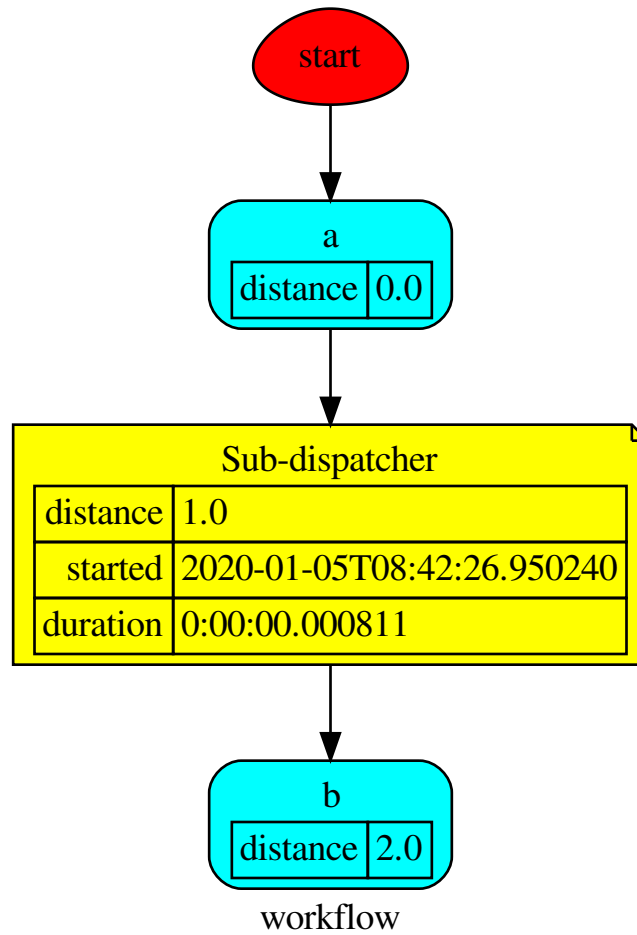
When ‘value_type’, returns the data node value’s type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

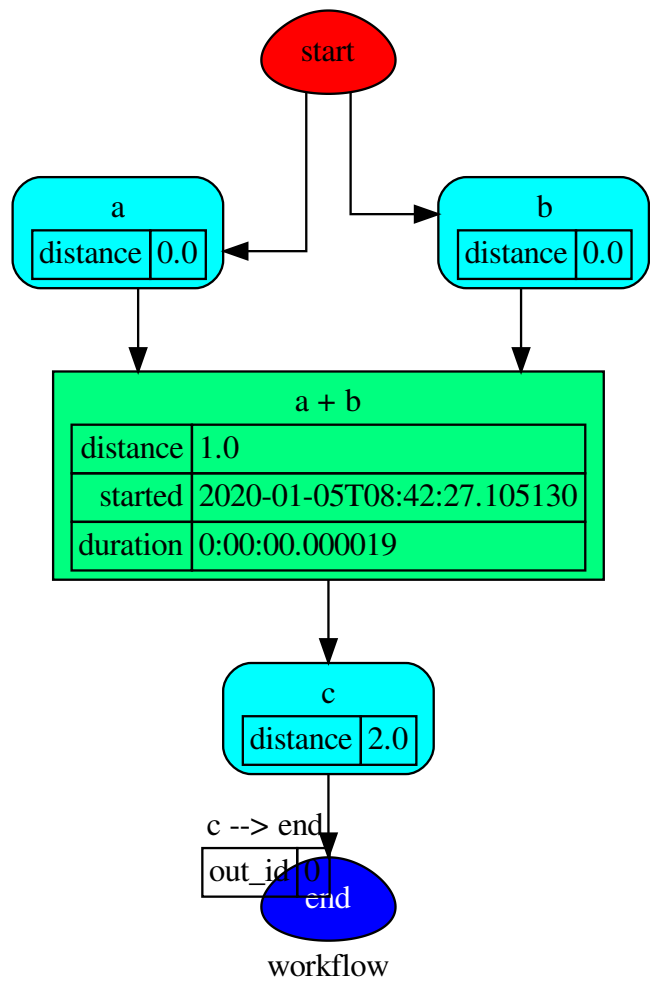
Example:



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')
```



7.2.4 blue

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

Classes

<i>BlueDispatcher</i>	Blueprint object is a blueprint of how to construct or extend a Dispatcher.
<i>Blueprint</i>	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:

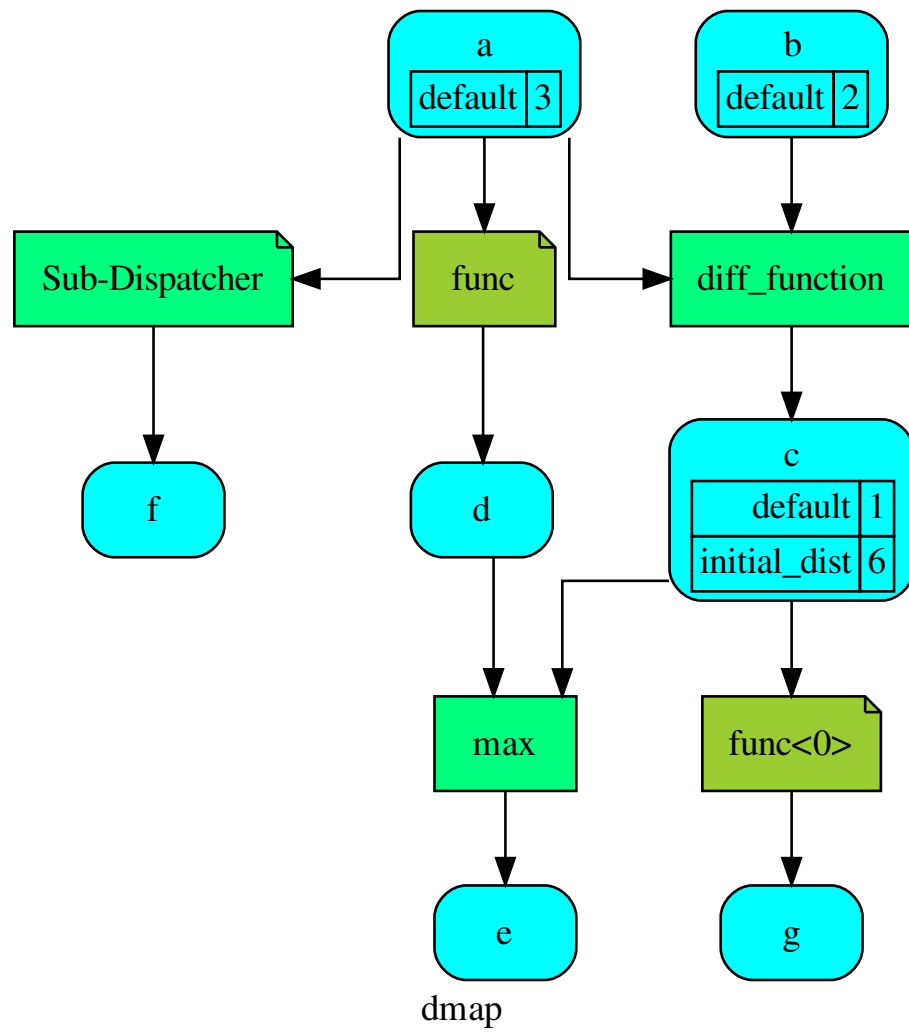
```
>>> blue.set_default_value(data_id='c', value=1, initial_dist=6)
<schedula.utils.blue.BlueDispatcher object at ...>
```

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*:

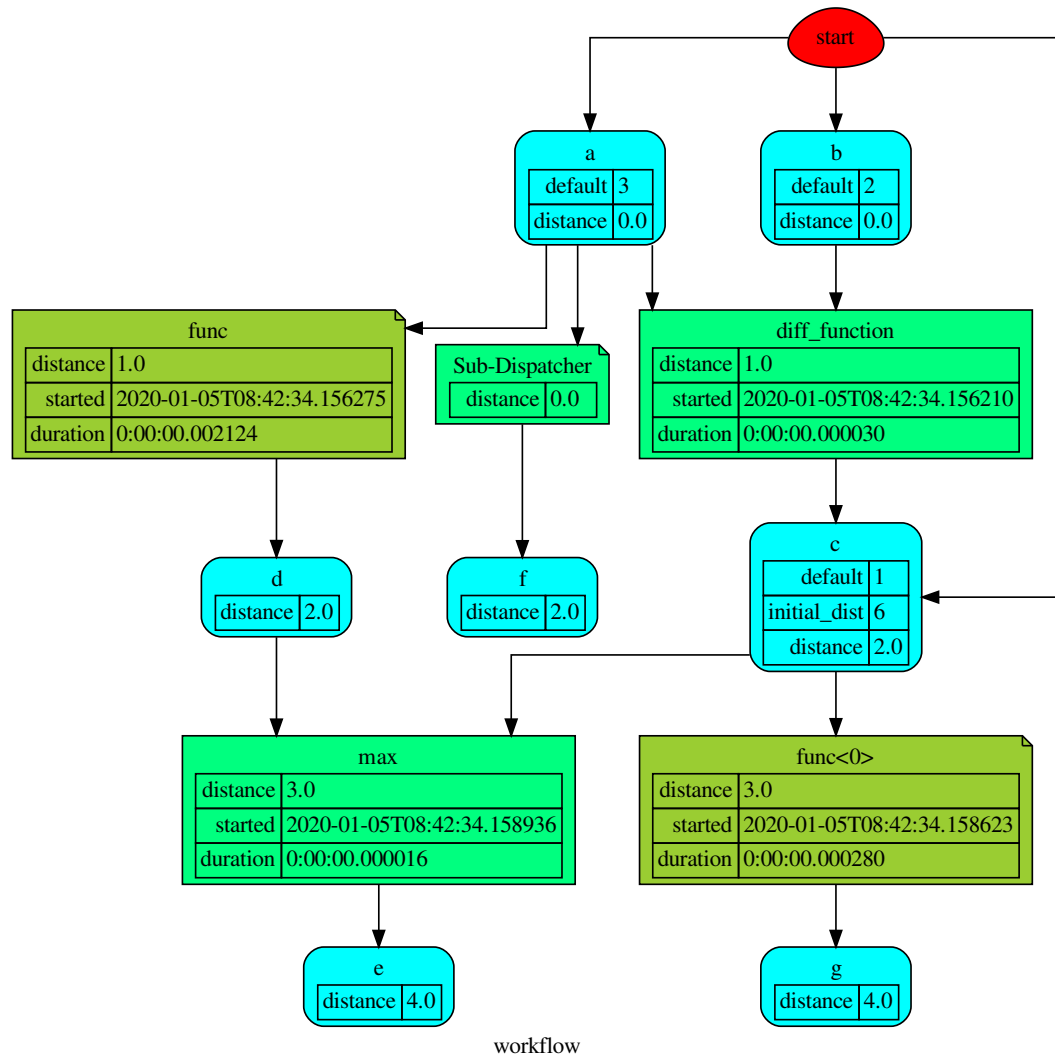
```
>>> dsp = blue.register(memo={}); dsp
<schedula.dispatcher.Dispatcher object at ...>
```



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)])
    
```

Methods

<code>__init__</code>	Initialize self.
<code>add_data</code>	Add a single data node to the dispatcher.
<code>add_dispatcher</code>	Add a single sub-dispatcher node to dispatcher.
<code>add_from_lists</code>	Add multiple function and data nodes to dispatcher.
<code>add_func</code>	Add a single function node to dispatcher.
<code>add_function</code>	Add a single function node to dispatcher.
<code>extend</code>	Extends deferred operations calling each operation of given Blueprints.
<code>register</code>	Creates a <code>Blueprint.cls</code> and calls each deferred operation.
<code>set_default_value</code>	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, ['length'])
>>> 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

<code>__init__</code>	Initialize self.
<code>extend</code>	Extends deferred operations calling each operation of given Blueprints.
<code>register</code>	Creates a <i>Blueprint.cls</i> 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, ['length'])
>>> 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, ['length'])
>>> 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 ...>
```

7.2.5 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()*

7.2.6 des

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

Functions

<i>get_attr_doc</i>
<i>get_link</i>
<i>get_summary</i>
<i>search_node_description</i>

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'*)

7.2.7 drw

It provides functions to plot dispatcher map and workflow.

Sub-Modules:

<i>nodes</i>	It provides docutils nodes to plot dispatcher map and workflow.
--------------	---

nodes

It provides docutils nodes to plot dispatcher map and workflow.

Functions

<i>autoplot_callback</i>
<i>autoplot_function</i>
<i>basic_app</i>
<i>before_request</i>
<i>cached_view</i>
<i>jinja2_format</i>
<i>render_output</i>
<i>run_server</i>
<i>site_view</i>
<i>uncpath</i>
<i>update_filenames</i>
<i>valid_filename</i>

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

<code>__init__</code>	Initialize self.
<code>dot</code>	
<code>href</code>	
<code>items</code>	
<code>parent_ref</code>	
<code>render_funcs</code>	
<code>render_size</code>	
<code>style</code>	
<code>yield_attr</code>	

`__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

<i>counter</i>
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 = ('?', '+wildcard', '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

<u>__init__</u>	Initialize self.
app	
get_port	
run	
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

<code>is_running</code>
<code>url</code>

is_running

`Site.is_running`

url

`Site.url`

SiteFolder

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

Methods

<code>__init__</code>	Initialize self.
-----------------------	------------------

Continued on next page

Table 30 – continued from previous page

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

<code>counter</code>
<code>digraph</code>
<code>ext</code>
<code>filename</code>
<code>inputs</code>
<code>label_name</code>
<code>name</code>
<code>outputs</code>
<code>title</code>
<code>view_id</code>

`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

<code>__init__</code>	Initialize self.
<code>legend</code>	
<code>render</code>	
<code>view</code>	

`__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

<code>__init__</code>	Initialize self.
<code>add_items</code>	
<code>app</code>	
<code>clear</code>	
<code>copy</code>	
<code>fromkeys</code>	Create a new ordered dictionary with keys from iterable and values set to value.
<code>get</code>	Return the value for key if key is in the dictionary, else default.
<code>get_dsp_from</code>	
<code>get_sol_from</code>	
<code>items</code>	
<code>keys</code>	
<code>move_to_end</code>	Move an existing element to the end (or beginning if last is false).
<code>pop</code>	value.
<code>popitem</code>	Remove and return a (key, value) pair from the dictionary.
<code>render</code>	
<code>rules</code>	
<code>setdefault</code>	Insert key with a value of default if key is not in the dictionary.
<code>site</code>	
<code>update</code>	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]
<code>values</code>	

`__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

<code>include_folders_as_filenames</code>
<code>nodes</code>
<code>options</code>

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

<code>__init__</code>	Initialize self.
<code>render</code>	
<code>view</code>	

`__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

<code>counter</code>
<code>ext</code>
<code>filename</code>
<code>name</code>
<code>title</code>
<code>view_id</code>

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>

7.2.8 dsp

It provides tools to create models with the *Dispatcher*.

Functions

<code>add_function</code>	Decorator to add a function to a dispatcher.
<code>are_in_nested_dicts</code>	Nested keys are inside of nested-dictionaries.
<code>bypass</code>	Returns the same arguments.
<code>combine_dicts</code>	Combines multiple dicts in one.
<code>combine_nested_dicts</code>	Merge nested-dictionaries.

Continued on next page

Table 38 – continued from previous page

<code>get_nested_dicts</code>	Get/Initialize the value of nested-dictionaries.
<code>kk_dict</code>	Merges and defines dictionaries with values identical to keys.
<code>map_dict</code>	Returns a dict with new key values.
<code>map_list</code>	Returns a new dict.
<code>parent_func</code>	Return the parent function of a wrapped function (wrapped with <code>functools.partial</code> and <code>add_args</code>).
<code>replicate_value</code>	Replicates <i>n</i> times the input value.
<code>selector</code>	Selects the chosen dictionary keys from the given dictionary.
<code>stack_nested_keys</code>	Stacks the keys of nested-dictionaries into tuples and yields a list of k-v pairs.
<code>stlp</code>	Converts a string in a tuple.
<code>summation</code>	Sums inputs values.

add_function

add_function (*dsp*, *inputs_kwargs=False*, *inputs_defaults=False*, ***kw*)

Decorator to add a function to a dispatcher.

Parameters

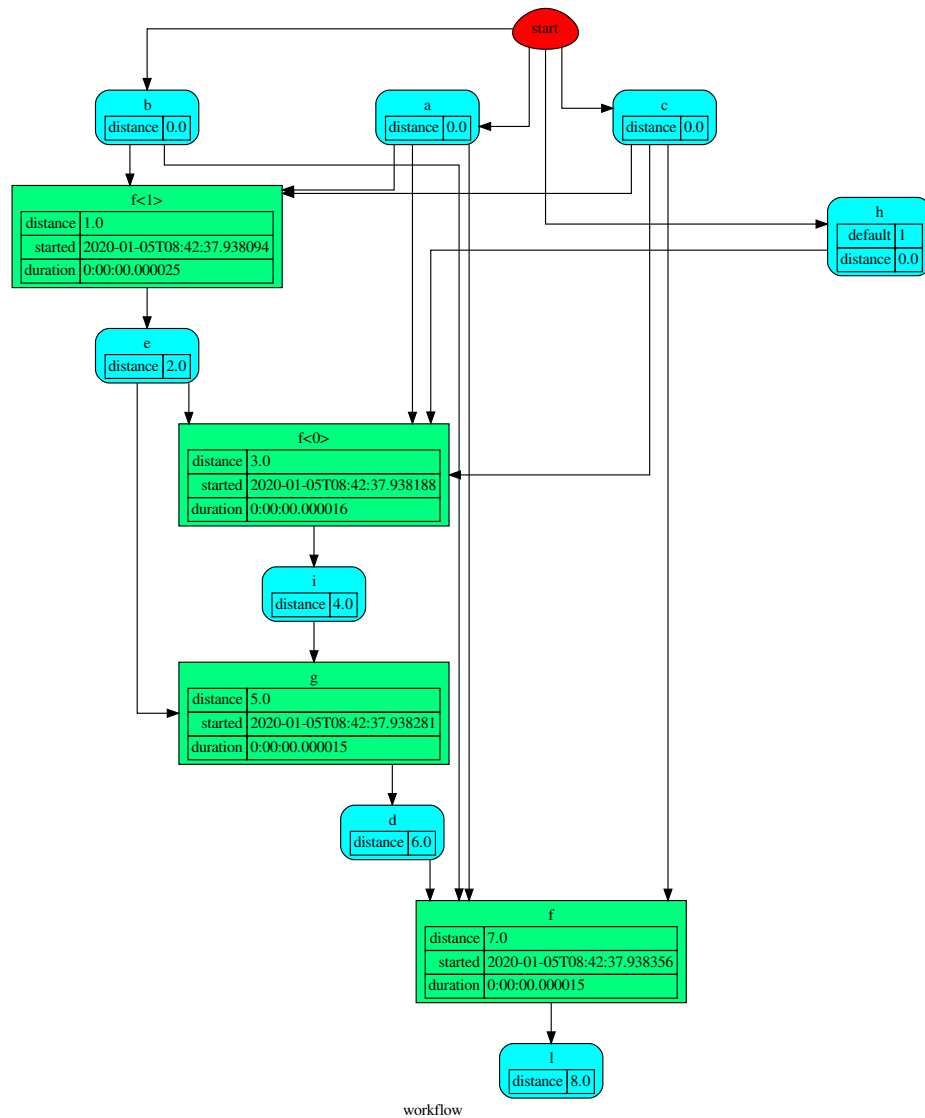
- **dsp** (*schedula.Dispatcher*) – A dispatcher.
- **inputs_kwargs** (*bool*) – Do you want to include kwargs as inputs?
- **inputs_defaults** (*bool*) – Do you want to set default values?
- **kw** (*dict*) – See :func:`~schedula.dispatcher.Dispatcher.add_function`.

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
>>> @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

<i>DispatchPipe</i>	It converts a <i>Dispatcher</i> into a function.
<i>NoSub</i>	Class for avoiding to add a sub solution to the workflow.
<i>SubDispatch</i>	It dispatches a given <i>Dispatcher</i> like a function.
<i>SubDispatchFunction</i>	It converts a <i>Dispatcher</i> into a function.
<i>SubDispatchPipe</i>	It converts a <i>Dispatcher</i> into a function.

Continued on next page

Table 39 – continued from previous page

<code>add_args</code>	Adds arguments to a function (left side).
<code>inf</code>	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

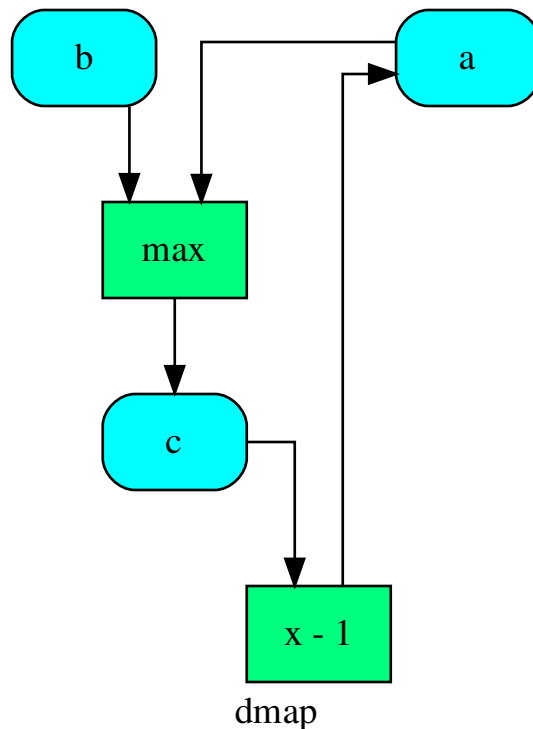
Note: This wrapper is not thread safe, because it overwrite the solution.

See also:

`dispatch()`, `shrink_dsp()`

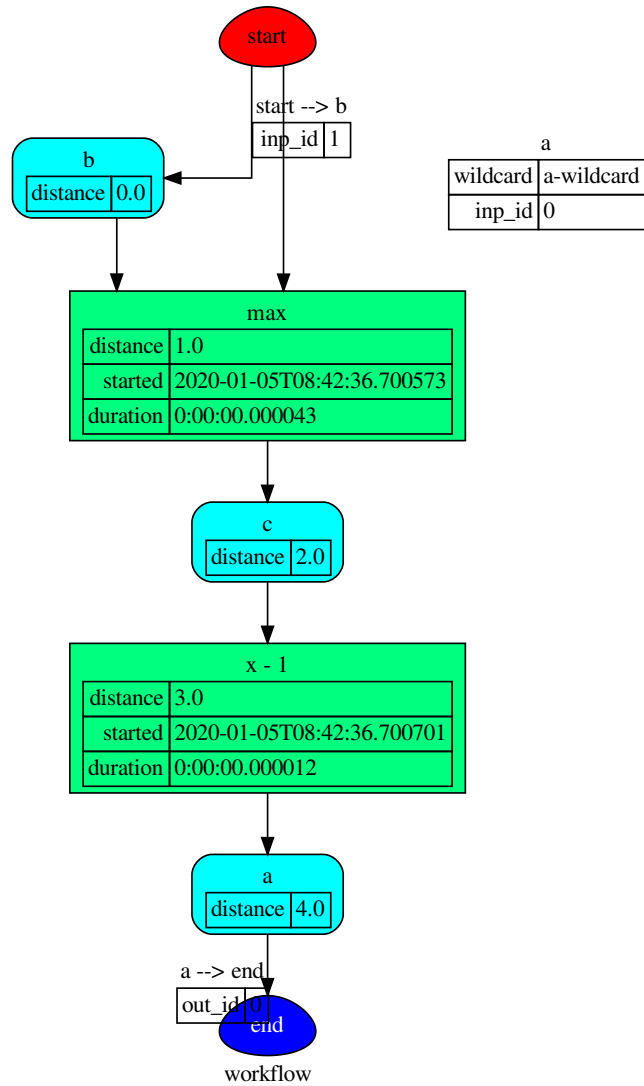
Example:

A dispatcher with two functions *max* and *min* and an unresolved cycle (i.e., $a \rightarrow \text{max} \rightarrow c \rightarrow \text{min} \rightarrow a$):



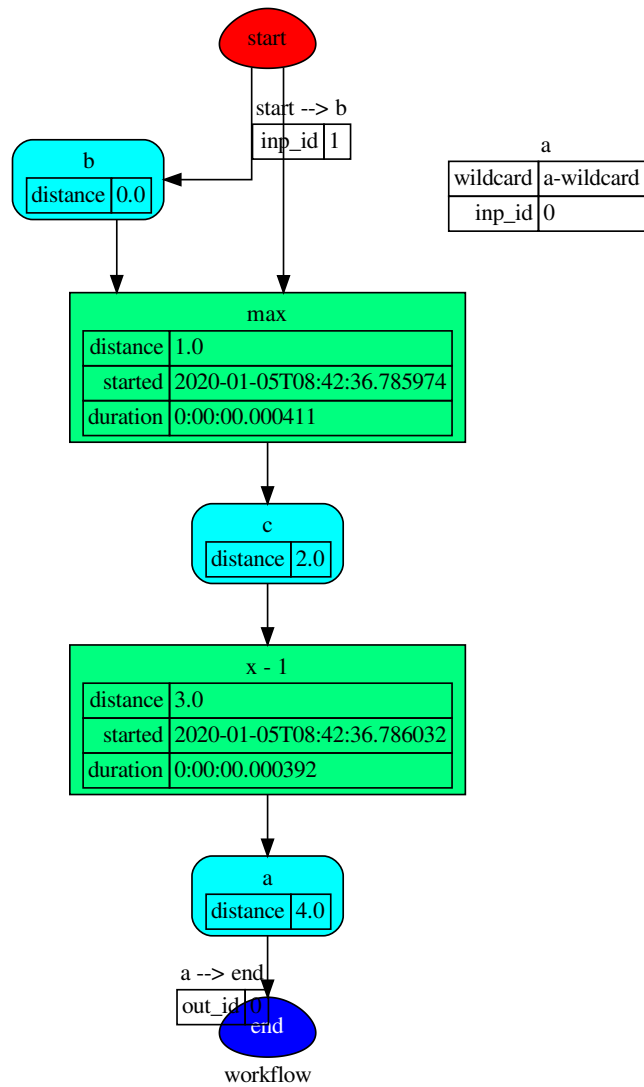
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```
>>> 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

<code>__init__</code>	Initializes the Sub-dispatch Function.
<code>blue</code>	Constructs a Blueprint out of the current object.
<code>copy</code>	
<code>get_node</code>	Returns a sub node of a dispatcher.
<code>plot</code>	Plots the Dispatcher with a graph in the DOT language with Graphviz.
<code>web</code>	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.

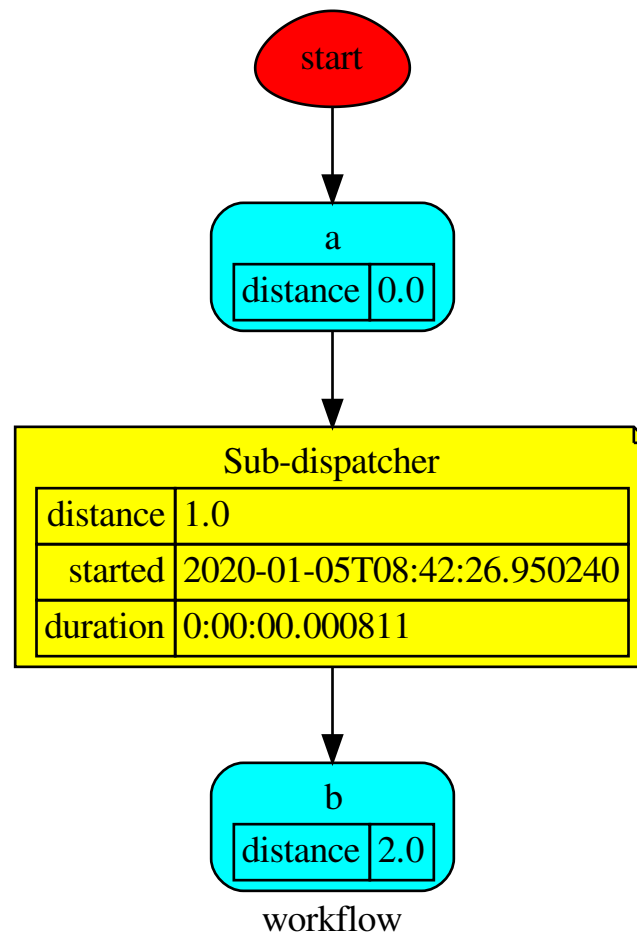
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

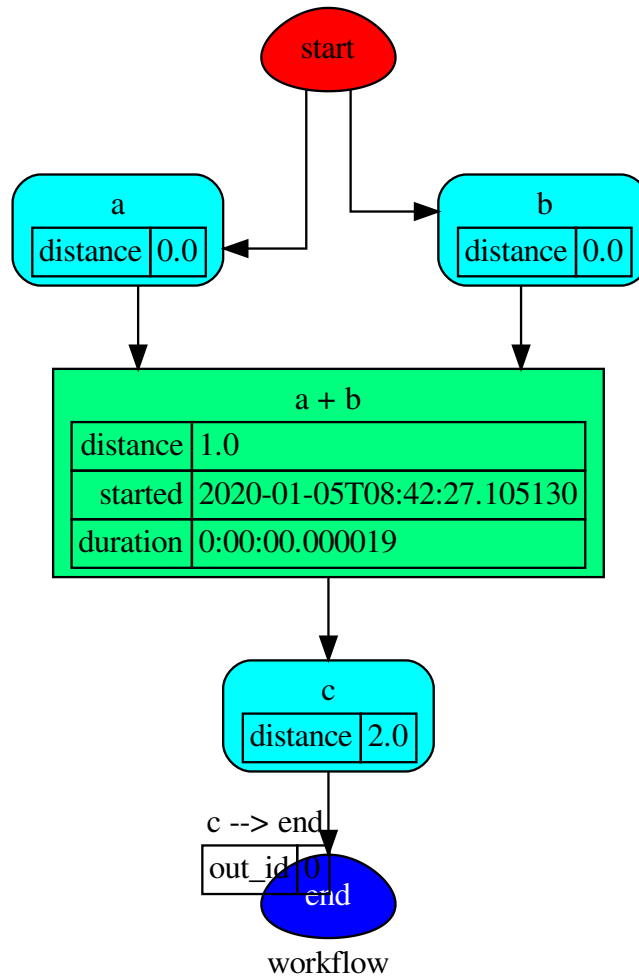
Example:



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, *args, **kwargs)`

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`

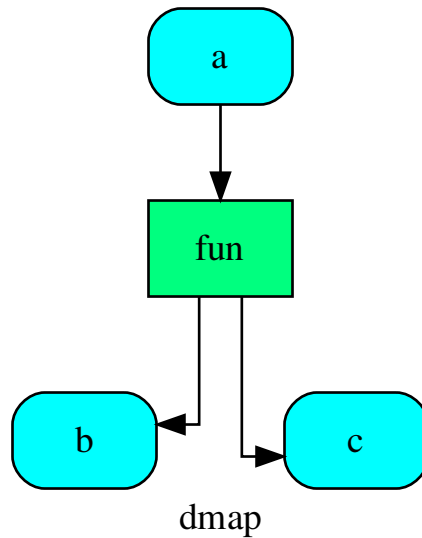
Example:

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

(continues on next page)

(continued from previous page)

```
>>> dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



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 back-end server.
- **run** (*bool*, *optional*) – Run the backend server?

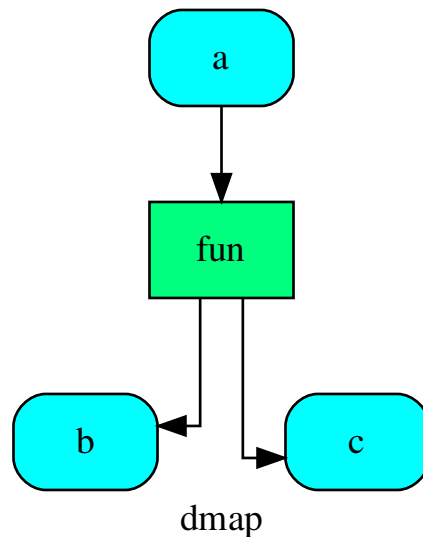
Returns A WebMap.

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`

plot (*workflow=None, *args, **kwargs*)

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.

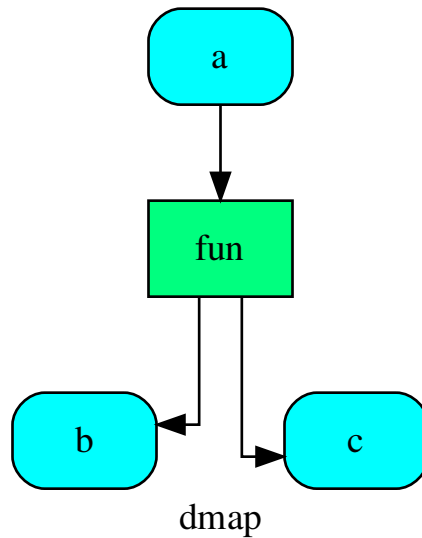
- **node_attr** (*dict*, *optional*) – Dict of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict*, *optional*) – Dict of (attribute, value) pairs set for all edges.
- **body** (*dict*, *optional*) – Dict of (attribute, value) pairs to add to the graph body.
- **directory** – Where is 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())])
```



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:

```

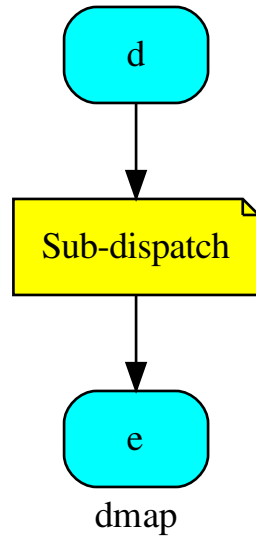
>>> from schedula import Dispatcher
>>> sub_dsp = Dispatcher(name='Sub-dispatcher')
...
>>> def fun(a):
...     return a + 1, a - 1
...

```

(continues on next page)

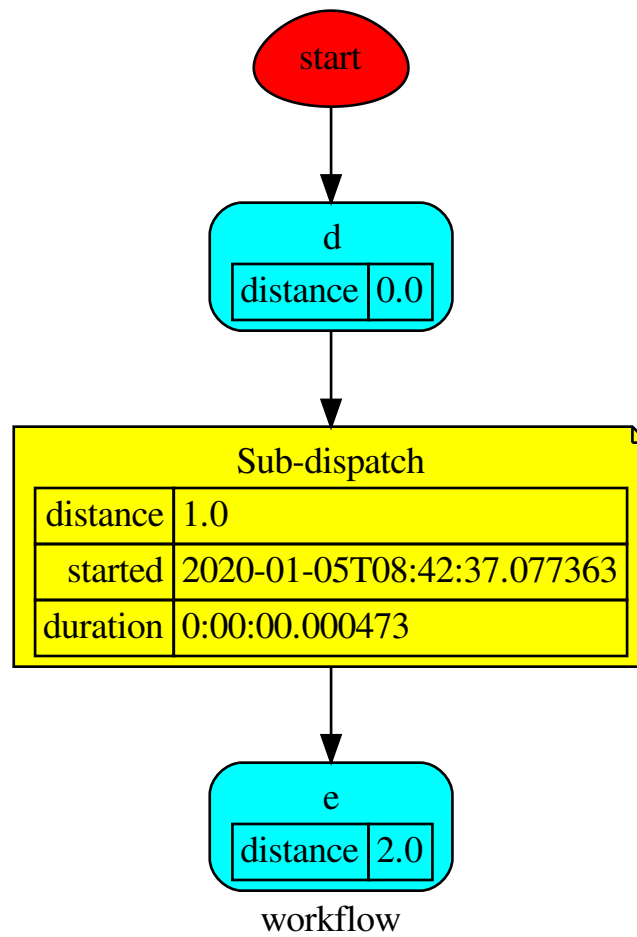
(continued from previous page)

```
>>> sub_dsp.add_function('fun', fun, ['a'], ['b', 'c'])
'fun'
>>> dispatch = SubDispatch(sub_dsp, ['a', 'b', 'c'], output_type='dict')
>>> dsp = Dispatcher(name='Dispatcher')
>>> dsp.add_function('Sub-dispatch', dispatch, ['d'], ['e'])
'Sub-dispatch'
```



The Dispatcher output is:

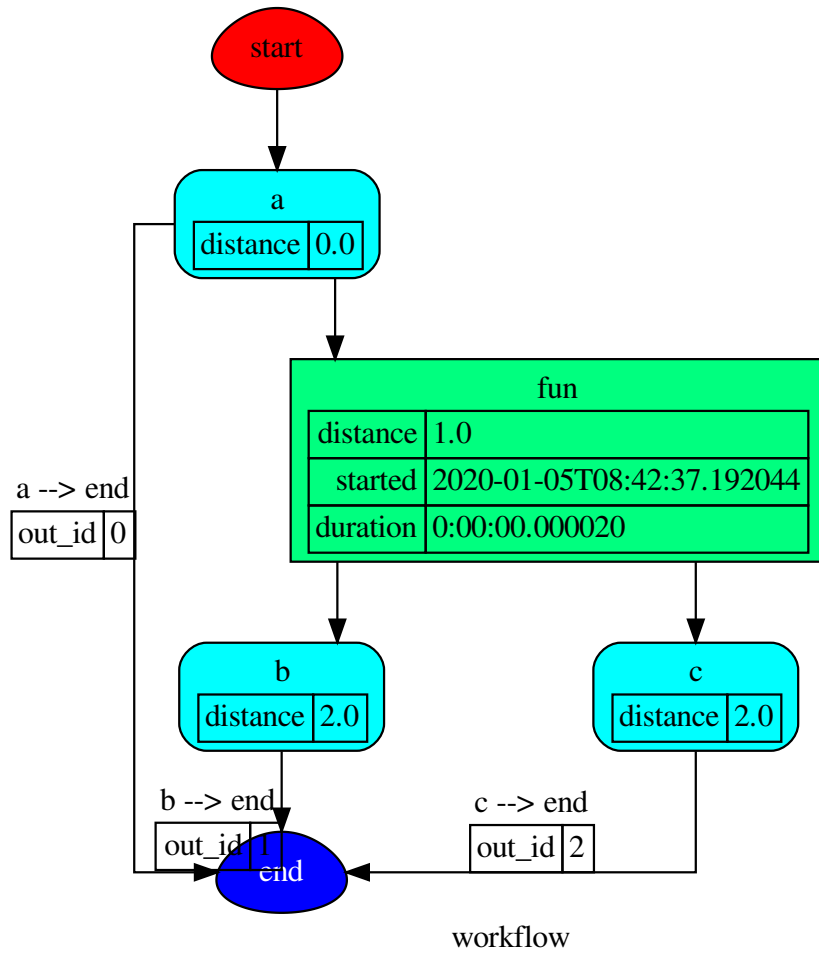
```
>>> o = dsp.dispatch(inputs={'d': {'a': 3}})
```

while, the Sub-dispatch is:

```

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



Methods

<code>__init__</code>	Initializes the Sub-dispatch.
<code>blue</code>	Constructs a Blueprint out of the current object.
<code>copy</code>	
<code>get_node</code>	Returns a sub node of a dispatcher.
<code>plot</code>	Plots the Dispatcher with a graph in the DOT language with Graphviz.
<code>web</code>	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.

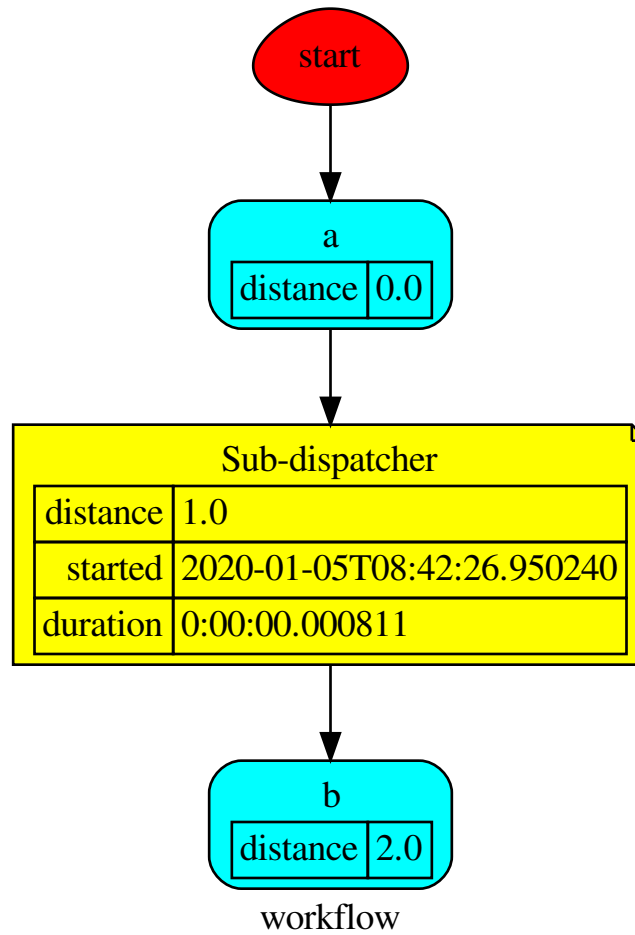
When ‘value_type’, returns the data node value’s type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

Example:



Get the sub node output:

```

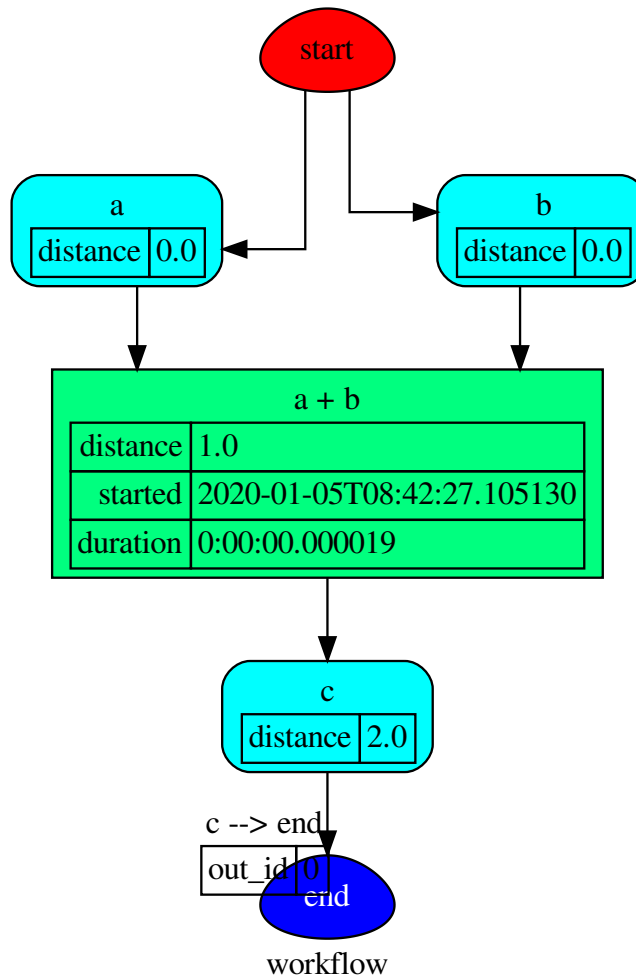
>>> dsp.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> dsp.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))

```

```

>>> 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 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*

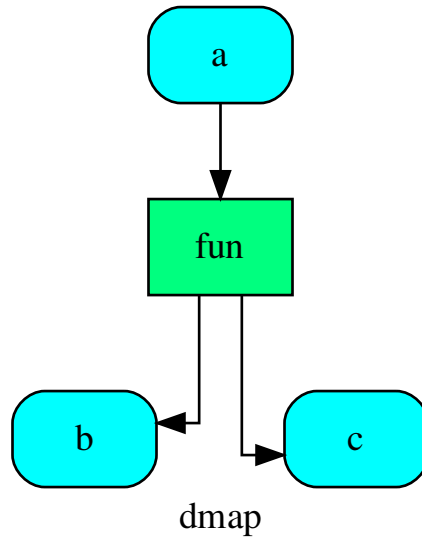
Example:

```
>>> 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'
```

(continues on next page)

(continued from previous page)

```
>>> dsp.plot(view=False, graph_attr={'ratio': '1'})
SiteMap([(Dispatcher, SiteMap())])
```



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 back-end server.
- **run** (*bool, optional*) – Run the backend server?

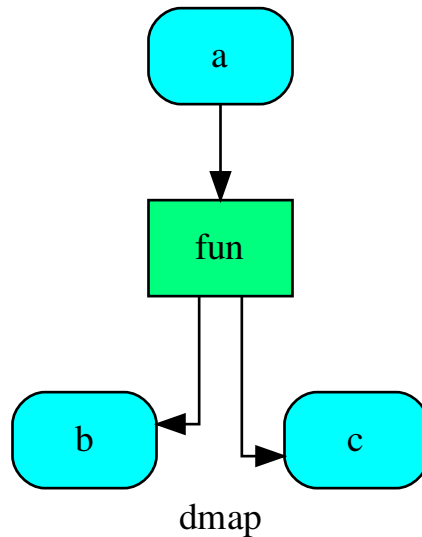
Returns A WebMap.

Return type *WebMap*

Example:

From a dispatcher like this:

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



You can create a web server with the following steps:

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

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

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

Parameters

- **dsp** (*schedula.Dispatcher*) – A dispatcher that identifies the model

adopted.

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

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*.

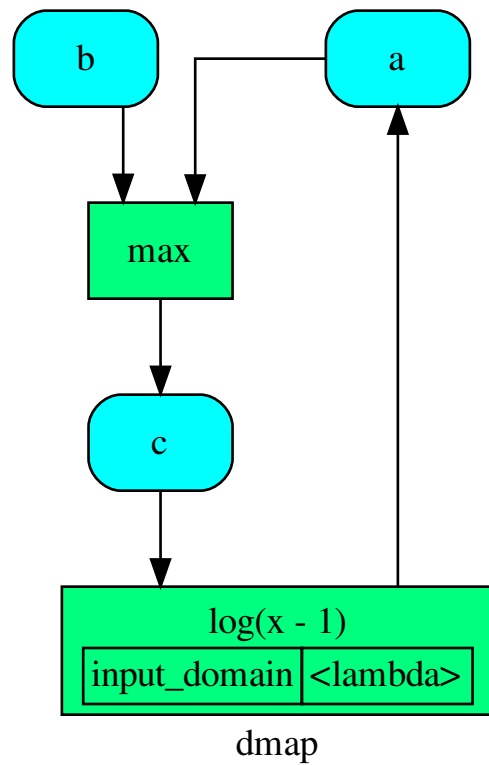
Return type callable

See also:

dispatch(), *shrink_dsp()*

Example:

A dispatcher with two functions *max* and *min* and an unresolved cycle (i.e., $a \rightarrow \text{max} \rightarrow c \rightarrow \text{min} \rightarrow a$):

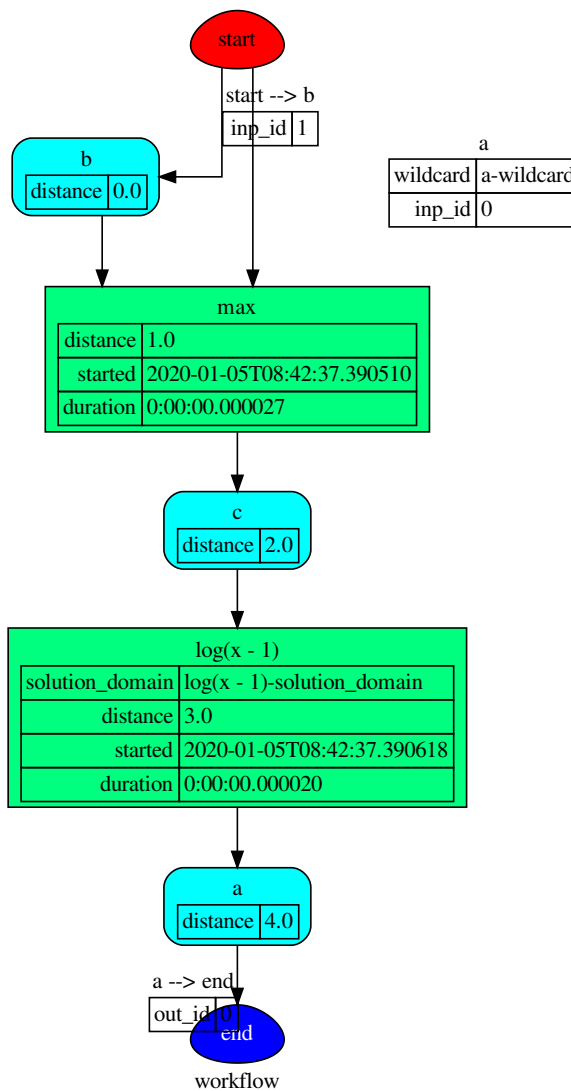


Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```

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

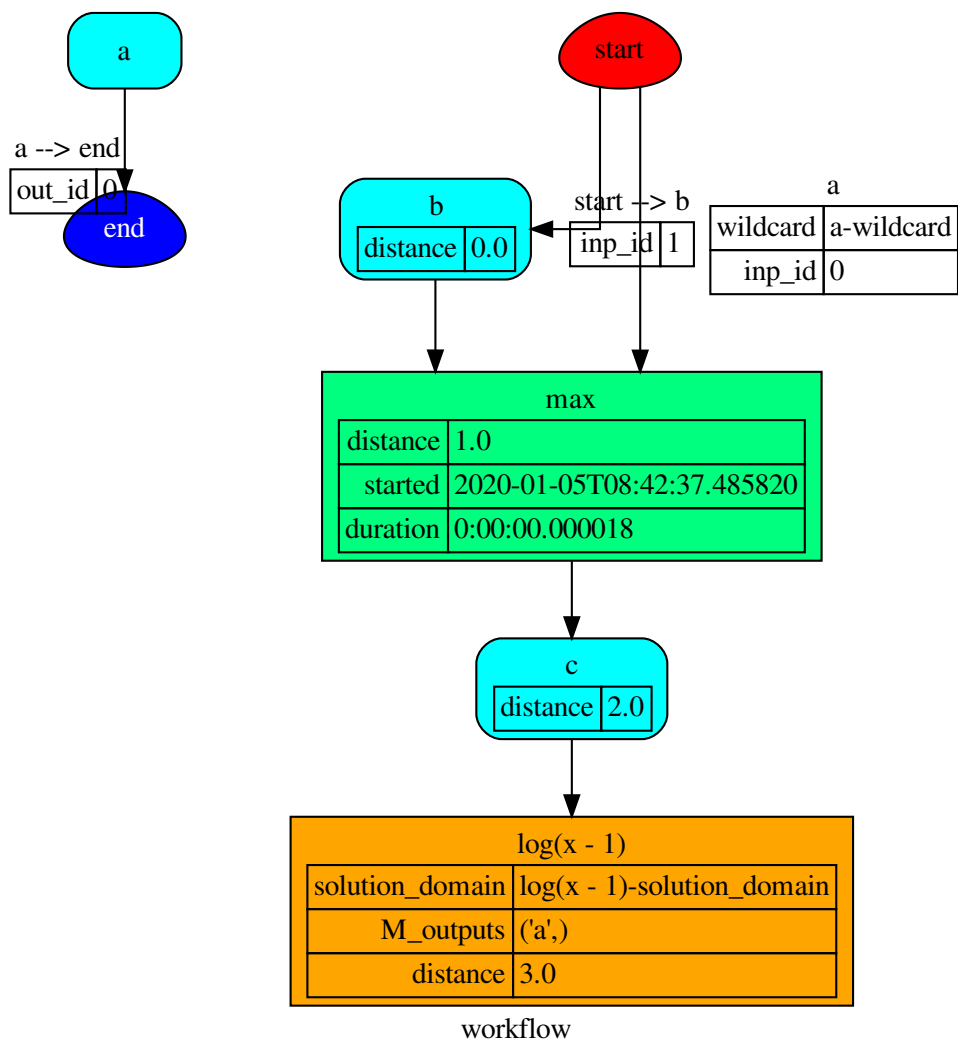
```



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: ...
  
```



Methods

<code>__init__</code>	Initializes the Sub-dispatch Function.
<code>blue</code>	Constructs a Blueprint out of the current object.
<code>copy</code>	
<code>get_node</code>	Returns a sub node of a dispatcher.
<code>plot</code>	Plots the Dispatcher with a graph in the DOT language with Graphviz.
<code>web</code>	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.

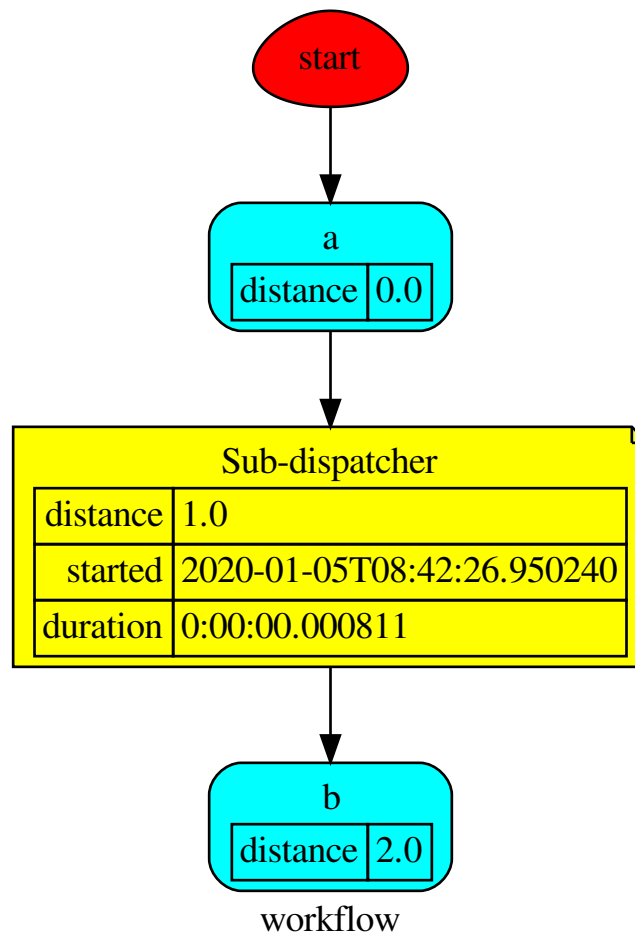
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

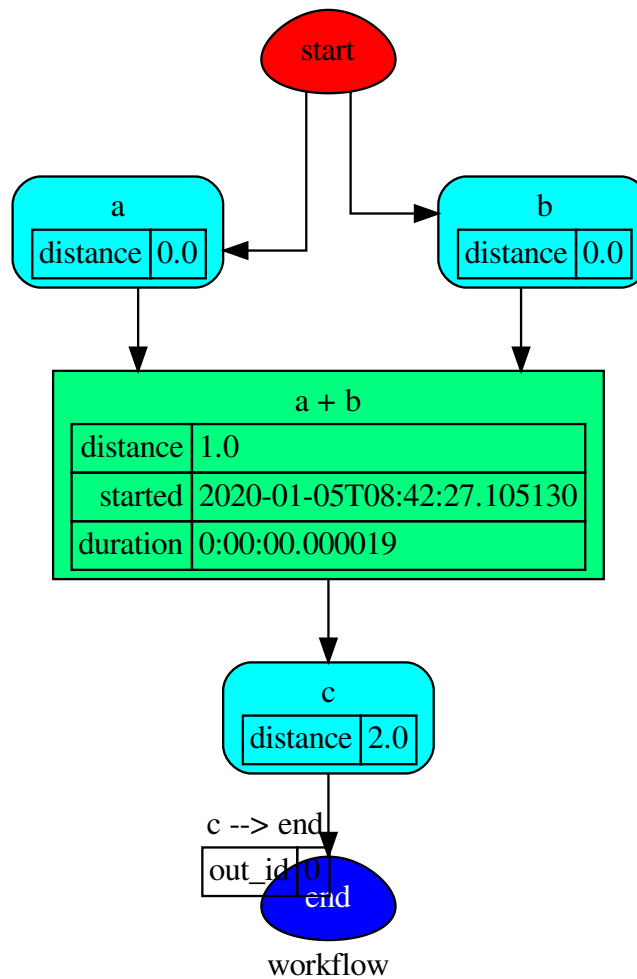
Example:



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.

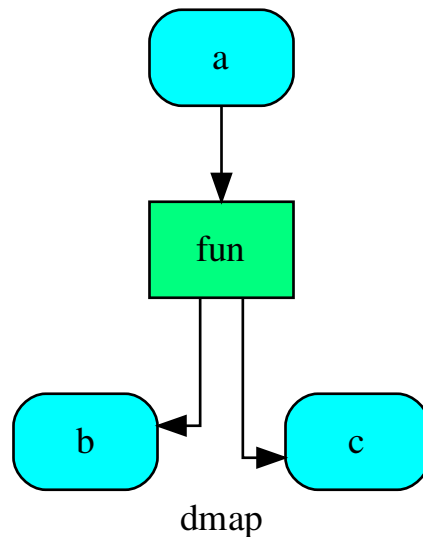
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

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())])
```



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 back-end server.
- **run** (*bool*, *optional*) – Run the backend server?

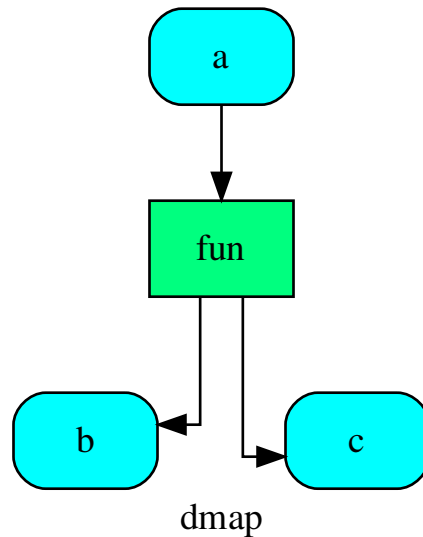
Returns A WebMap.

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__)
```

(continues on next page)

(continued from previous page)

```
>>> 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*.

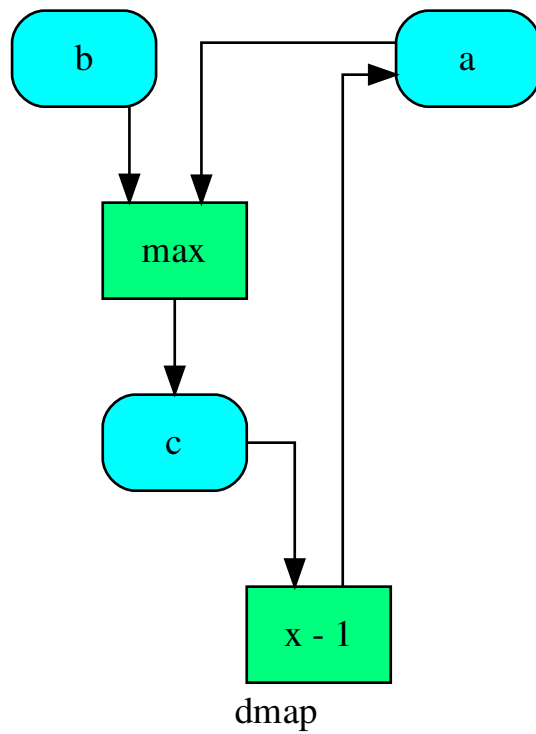
Return type callable

See also:

`dispatch()`, `shrink_dsp()`

Example:

A dispatcher with two functions *max* and *min* and an unresolved cycle (i.e., $a \rightarrow \text{max} \rightarrow c \rightarrow \text{min} \rightarrow a$):

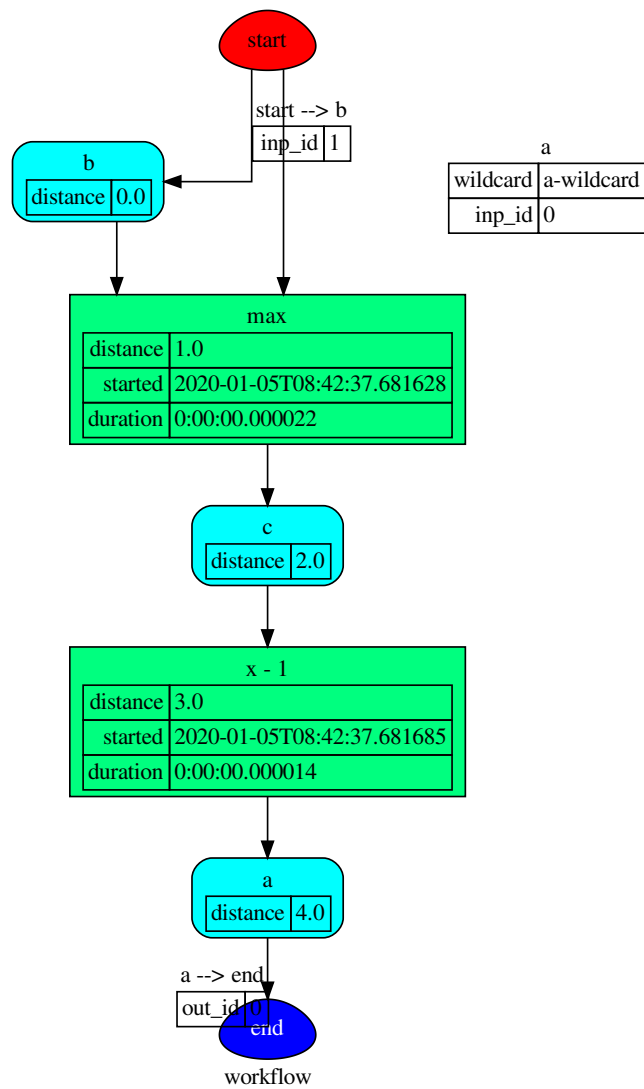


Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```

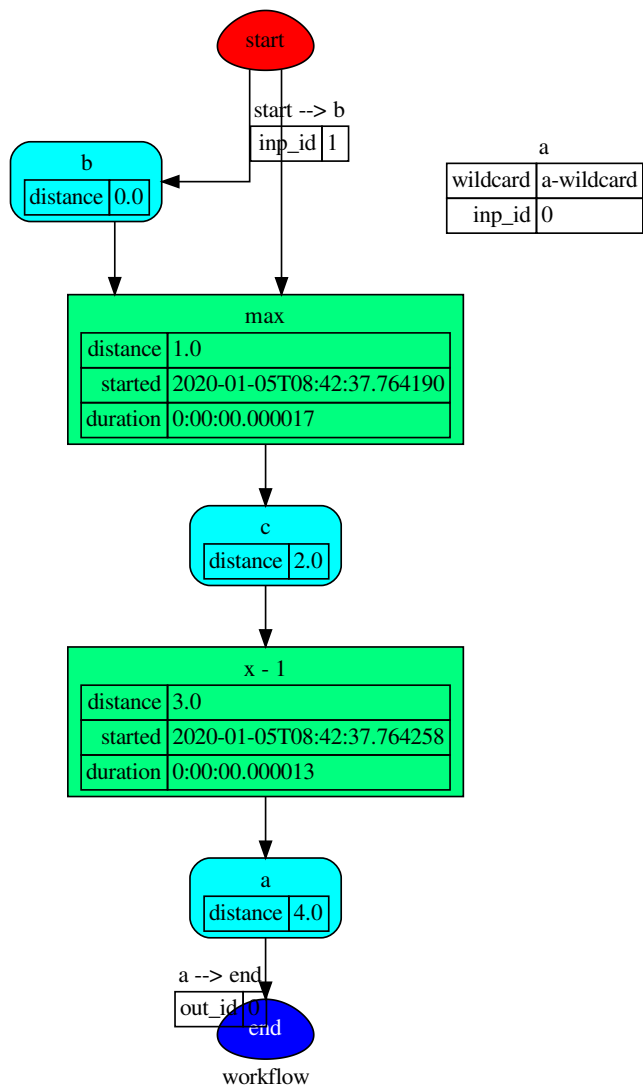
>>> fun = SubDispatchPipe(dsp, 'myF', ['a', 'b'], ['a'])
>>> fun.__name__
'myF'
>>> fun(2, 1)
1

```



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

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



Methods

<code>__init__</code>	Initializes the Sub-dispatch Function.
<code>blue</code>	Constructs a Blueprint out of the current object.
<code>copy</code>	
<code>get_node</code>	Returns a sub node of a dispatcher.
<code>plot</code>	Plots the Dispatcher with a graph in the DOT language with Graphviz.
<code>web</code>	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.

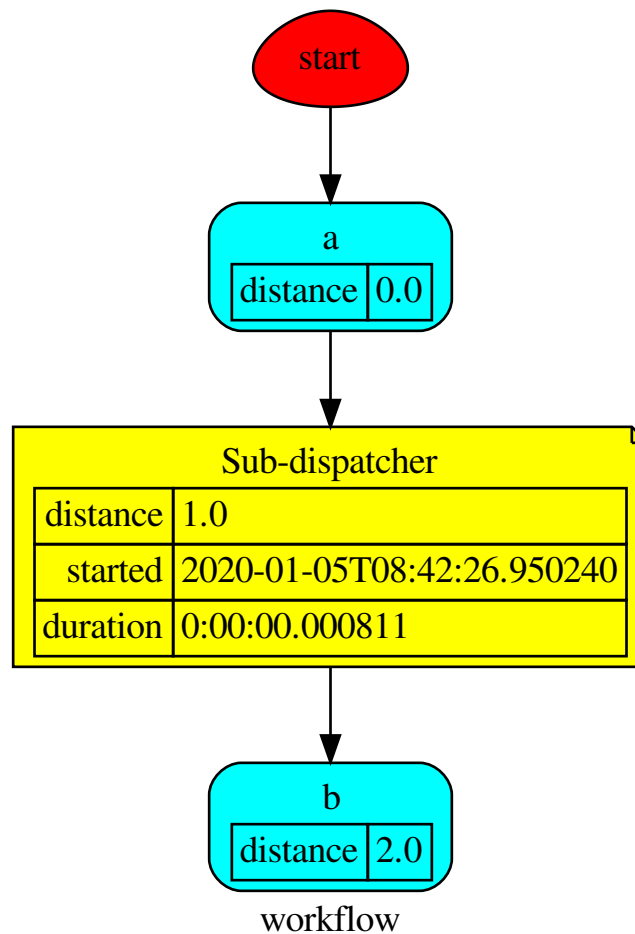
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

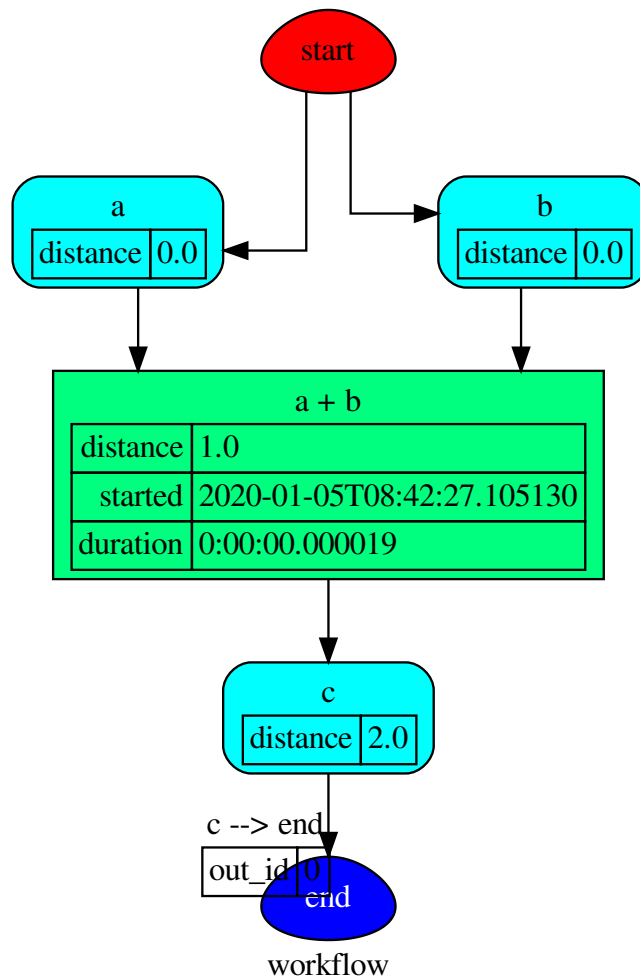
Example:



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 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.

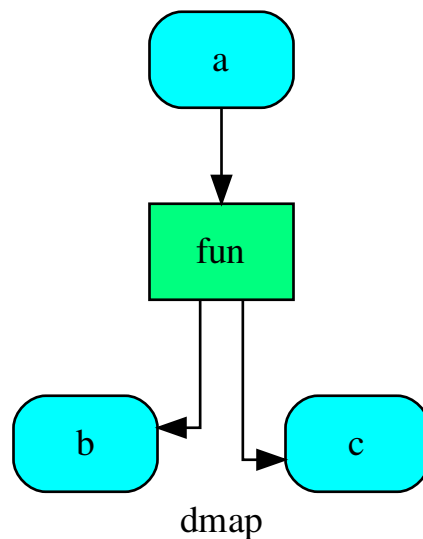
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?

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())])
```



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 back-end server.
- **run** (*bool, optional*) – Run the backend server?

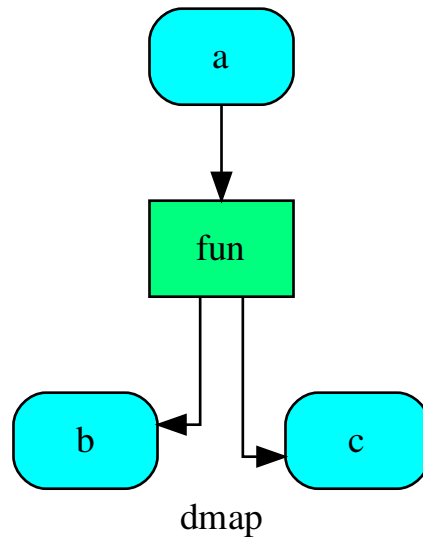
Returns A WebMap.

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__)
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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

<code>__init__</code>	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

<code>count</code>	Return number of occurrences of value.
<code>index</code>	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

<code>inf</code>	Alias for field number 0
<code>num</code>	Alias for field number 1

inf

`inf.inf`
Alias for field number 0

num

`inf.num`
Alias for field number 1

7.2.9 exc

Defines the dispatcher exception.

Exceptions

<code>DispatcherAbort</code>
<code>DispatcherError</code>
<code>ExecutorShutdown</code>
<code>SkipNode</code>

DispatcherAbort

exception `DispatcherAbort`

DispatcherError

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

ExecutorShutdown

exception `ExecutorShutdown`

SkipNode

exception `SkipNode` (**args*, *ex=None*, ***kwargs*)

7.2.10 gen

It contains classes and functions of general utility.

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

Functions

<code>counter</code>	Return a object whose <code>.__call__()</code> method returns consecutive values.
<code>pairwise</code>	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

<code>Token</code>	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

<code>__init__</code>	Initialize self.
<code>capitalize</code>	Return a capitalized version of the string.
<code>casefold</code>	Return a version of the string suitable for caseless comparisons.
<code>center</code>	Return a centered string of length width.
<code>count</code>	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
<code>encode</code>	Encode the string using the codec registered for encoding.
<code>endswith</code>	Return True if S ends with the specified suffix, False otherwise.
<code>expandtabs</code>	Return a copy where all tab characters are expanded using spaces.
<code>find</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>format</code>	Return a formatted version of S, using substitutions from args and kwargs.
<code>format_map</code>	Return a formatted version of S, using substitutions from mapping.
<code>index</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>isalnum</code>	Return True if the string is an alpha-numeric string, False otherwise.
<code>isalpha</code>	Return True if the string is an alphabetic string, False otherwise.
<code>isascii</code>	Return True if all characters in the string are ASCII, False otherwise.
<code>isdecimal</code>	Return True if the string is a decimal string, False otherwise.
<code>isdigit</code>	Return True if the string is a digit string, False otherwise.
<code>isidentifier</code>	Return True if the string is a valid Python identifier, False otherwise.
<code>islower</code>	Return True if the string is a lowercase string, False otherwise.
<code>isnumeric</code>	Return True if the string is a numeric string, False otherwise.
<code>isprintable</code>	Return True if the string is printable, False otherwise.
<code>isspace</code>	Return True if the string is a whitespace string, False otherwise.
<code>istitle</code>	Return True if the string is a title-cased string, False otherwise.
<code>isupper</code>	Return True if the string is an uppercase string, False otherwise.
<code>join</code>	Concatenate any number of strings.
<code>ljust</code>	Return a left-justified string of length width.
<code>lower</code>	Return a copy of the string converted to lowercase.
<code>lstrip</code>	Return a copy of the string with leading whitespace removed.

Continued on next page

Table 53 – continued from previous page

<code>maketrans</code>	Return a translation table usable for <code>str.translate()</code> .
<code>partition</code>	Partition the string into three parts using the given separator.
<code>replace</code>	Return a copy with all occurrences of substring <code>old</code> replaced by <code>new</code> .
<code>rfind</code>	Return the highest index in <code>S</code> where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>S[start:end]</code> .
<code>rindex</code>	Return the highest index in <code>S</code> where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>S[start:end]</code> .
<code>rjust</code>	Return a right-justified string of length <code>width</code> .
<code>rpartition</code>	Partition the string into three parts using the given separator.
<code>rsplit</code>	Return a list of the words in the string, using <code>sep</code> as the delimiter string.
<code>rstrip</code>	Return a copy of the string with trailing whitespace removed.
<code>split</code>	Return a list of the words in the string, using <code>sep</code> as the delimiter string.
<code>splitlines</code>	Return a list of the lines in the string, breaking at line boundaries.
<code>startswith</code>	Return <code>True</code> if <code>S</code> starts with the specified prefix, <code>False</code> otherwise.
<code>strip</code>	Return a copy of the string with leading and trailing whitespace remove.
<code>swapcase</code>	Convert uppercase characters to lowercase and lowercase characters to uppercase.
<code>title</code>	Return a version of the string where each word is titlecased.
<code>translate</code>	Replace each character in the string using the given translation table.
<code>upper</code>	Return a copy of the string converted to uppercase.
<code>zfill</code>	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.

7.2.11 io

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

Functions

<code>load_default_values</code>	Load Dispatcher default values in Python pickle format.
<code>load_dispatcher</code>	Load Dispatcher object in Python pickle format.
<code>load_map</code>	Load Dispatcher map in Python pickle format.
<code>save_default_values</code>	Write Dispatcher default values in Python pickle format.

Continued on next page

Table 54 – continued from previous page

<code>save_dispatcher</code>	Write Dispatcher object in Python pickle format.
<code>save_map</code>	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)
```

7.2.12 sol

It provides a solution class for dispatch result.

Classes

<i>Solution</i>	Solution class for dispatch result.
-----------------	-------------------------------------

Solution

```
class Solution (dsp=None, inputs=None, outputs=None, wildcard=False, cutoff=None, in-
                puts_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

<code>__init__</code>	Initialize self.
<code>clear</code>	
<code>copy</code>	
<code>fromkeys</code>	Create a new ordered dictionary with keys from iterable and values set to value.
<code>get</code>	Return the value for key if key is in the dictionary, else default.
<code>get_node</code>	Returns a sub node of a dispatcher.
<code>get_sub_dsp_from_workflow</code>	Returns the sub-dispatcher induced by the workflow from sources.
<code>items</code>	
<code>keys</code>	
<code>move_to_end</code>	Move an existing element to the end (or beginning if last is false).
<code>plot</code>	Plots the Dispatcher with a graph in the DOT language with Graphviz.
<code>pop</code>	value.
<code>popitem</code>	Remove and return a (key, value) pair from the dictionary.
<code>result</code>	Set all asynchronous results.
<code>setdefault</code>	Insert key with a value of default if key is not in the dictionary.
<code>update</code>	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]
<code>values</code>	
<code>web</code>	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.

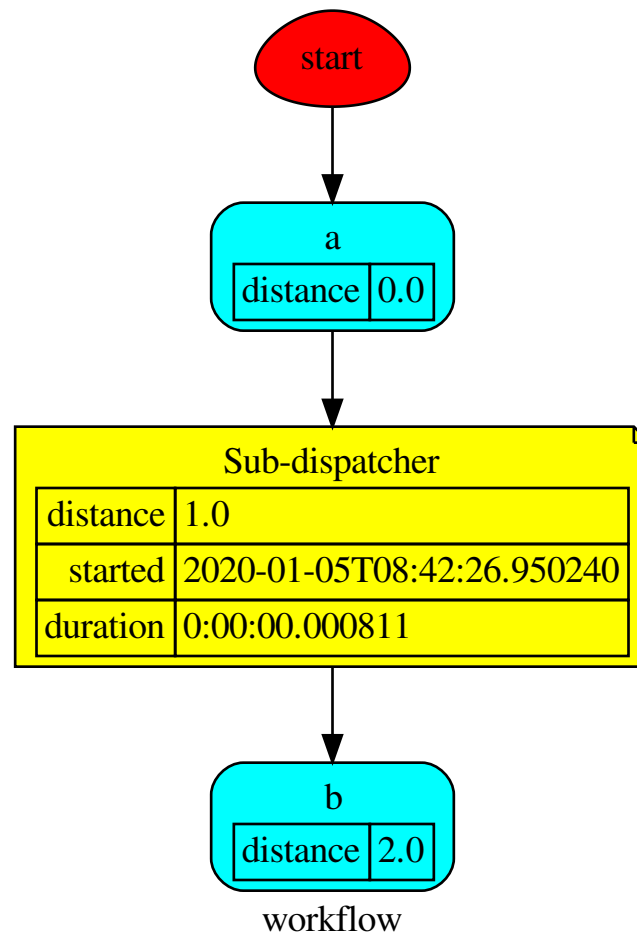
When 'value_type', returns the data node value's type.

When *None*, returns the node attributes.

Returns Node attributes and its real path.

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

Example:



Get the sub node output:

```

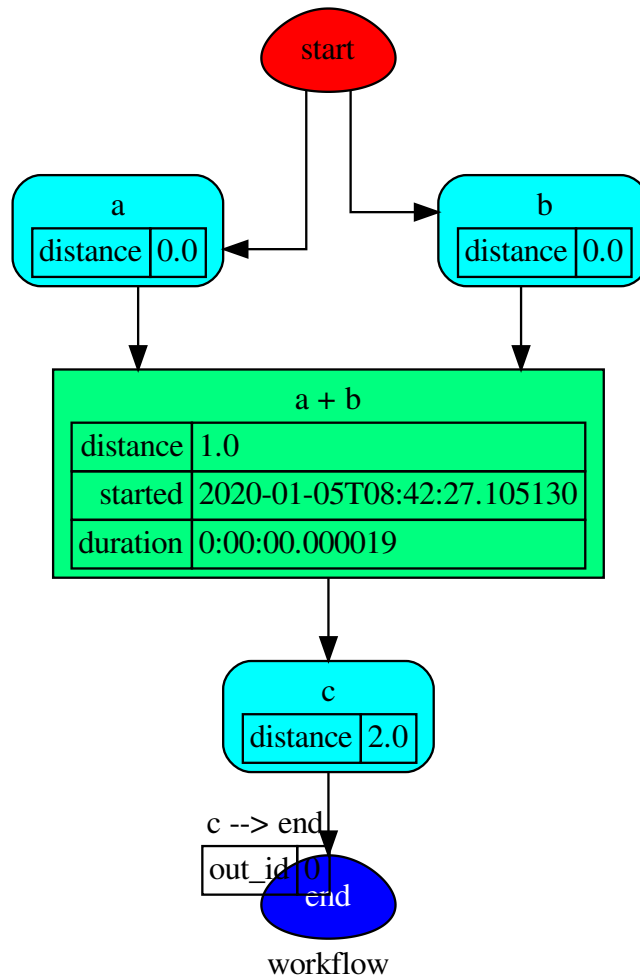
>>> dsp.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> dsp.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))

```

```

>>> 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.

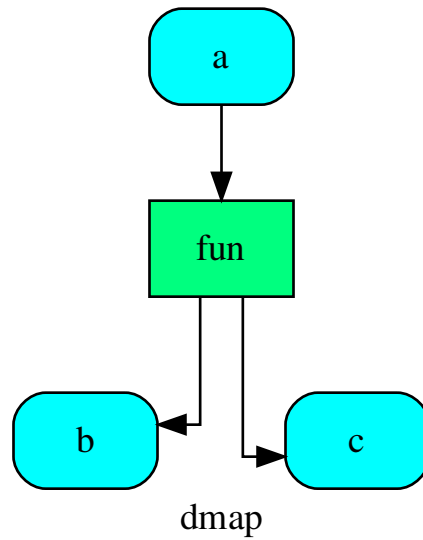
- **comment** (*str*) – Comment added to the first line of the source.
- **directory** (*str*, *optional*) – (Sub)directory for source saving and rendering.
- **format** (*str*, *optional*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str*, *optional*) – Layout command used ('dot', 'neato',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

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 back-end server.
- **run** (*bool*, *optional*) – Run the backend server?

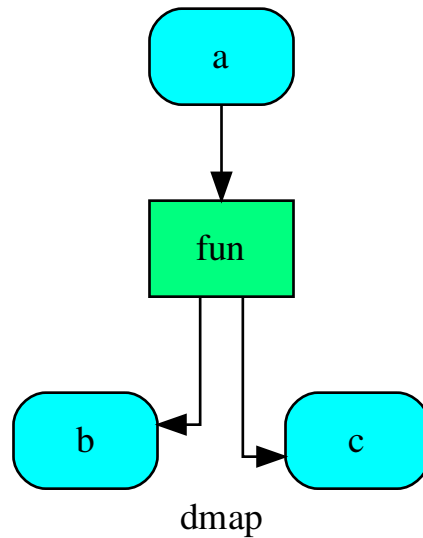
Returns A WebMap.

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

<i>pipe</i>	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.

7.2.13 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

<code>__init__</code>	Initialize self.
<code>dot</code>	
<code>href</code>	
<code>items</code>	
<code>parent_ref</code>	
<code>render_funcs</code>	
<code>render_size</code>	
<code>style</code>	
<code>yield_attr</code>	

`__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': 'empt`

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

<code>__init__</code>	Initialize self.
<code>dot</code>	
<code>view</code>	

`__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

<code>__init__</code>	Initialize self.
<code>add_items</code>	
<code>app</code>	
<code>clear</code>	
<code>copy</code>	
<code>fromkeys</code>	Create a new ordered dictionary with keys from iterable and values set to value.
<code>get</code>	Return the value for key if key is in the dictionary, else default.
<code>get_dsp_from</code>	
<code>get_sol_from</code>	
<code>items</code>	
<code>keys</code>	
<code>move_to_end</code>	Move an existing element to the end (or beginning if last is false).
<code>pop</code>	value.
<code>popitem</code>	Remove and return a (key, value) pair from the dictionary.
<code>render</code>	
<code>rules</code>	
<code>setdefault</code>	Insert key with a value of default if key is not in the dictionary.
<code>site</code>	
<code>site_index</code>	
<code>update</code>	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]
<code>values</code>	

`__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_label'}`

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`

7.3 ext

It provides sphinx extensions.

Extensions:

<i>autosummary</i>	It is a patch to sphinx.ext.autosummary.
<i>dispatcher</i>	It provides dispatcher sphinx documenter and directive.

7.3.1 autosummary

It is a patch to sphinx.ext.autosummary.

Functions

<i>generate_autosummary_docs</i>
<i>get_members</i>
<i>process_generate_options</i>
<i>setup</i>

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*)

7.3.2 dispatcher

It provides dispatcher sphinx documenter and directive.

Extensions:

<i>documenter</i>	Dispatcher documenter.
<i>graphviz</i>	Dispatcher directive.

documenter

Dispatcher documenter.

Functions

<code>add_autodocumenter</code>	
<code>contains_doctest</code>	
<code>get_grandfather_content</code>	
<code>get_grandfather_offset</code>	
<code>setup</code>	Setup <i>dispatcher</i> Sphinx extension module.

add_autodocumenter

add_autodocumenter (*app*, *cls*)

contains_doctest

contains_doctest (*text*)

get_grandfather_content

get_grandfather_content (*content*, *level*=2)

get_grandfather_offset

get_grandfather_offset (*content*)

setup

setup (*app*)
Setup *dispatcher* Sphinx extension module.

Classes

<code>DispatcherDirective</code>	
<code>DispatcherDocumenter</code>	Specialized Documenter subclass for dispatchers.

DispatcherDirective

class DispatcherDirective (*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, **args*, ***kwargs*)

Methods

<code>__init__</code>	Initialize self.
<code>add_name</code>	Append self.options['name'] to node['names'] if it exists.
<code>assert_has_content</code>	Throw an ERROR-level DirectiveError if the directive doesn't have contents.
<code>debug</code>	
<code>directive_error</code>	Return a DirectiveError suitable for being thrown as an exception.
<code>error</code>	
<code>info</code>	
<code>run</code>	
<code>severe</code>	
<code>warning</code>	

`__init__`

`DispatcherDirective.__init__(name, arguments, options, content, lineno, content_offset, *args, **kwargs)`
 Initialize self. See help(type(self)) for accurate signature.

`add_name`

`DispatcherDirective.add_name(node)`
 Append self.options['name'] to node['names'] if it exists.
 Also normalize the name string and register it as explicit target.

`assert_has_content`

`DispatcherDirective.assert_has_content()`
 Throw an ERROR-level DirectiveError if the directive doesn't have contents.

`debug`

`DispatcherDirective.debug(message)`

`directive_error`

`DispatcherDirective.directive_error(level, message)`
 Return a DirectiveError suitable for being thrown as an exception.
 Call “raise self.directive_error(level, message)” from within a directive implementation to return one single system message at level *level*, which automatically gets the directive block and the line number added.
 Preferably use the *debug*, *info*, *warning*, *error*, or *severe* wrapper methods, e.g. `self.error(message)` to generate an ERROR-level directive error.

error

`DispatcherDirective.error(message)`

info

`DispatcherDirective.info(message)`

run

`DispatcherDirective.run()`

severe

`DispatcherDirective.severe(message)`

warning

`DispatcherDirective.warning(message)`

`__init__(name, arguments, options, content, lineno, content_offset, *args, **kwargs)`
 Initialize self. See help(type(self)) for accurate signature.

Attributes

<code>config</code>	Reference to the <code>Config</code> object.
<code>env</code>	Reference to the <code>BuildEnvironment</code> object.
<code>final_argument_whitespace</code>	
<code>has_content</code>	
<code>optional_arguments</code>	
<code>required_arguments</code>	

config

`DispatcherDirective.config`
 Reference to the `Config` object.

env

`DispatcherDirective.env`
 Reference to the `BuildEnvironment` object.

final_argument_whitespace

`DispatcherDirective.final_argument_whitespace = True`

has_content

`DispatcherDirective.has_content = True`

optional_arguments

`DispatcherDirective.optional_arguments = 0`

required_arguments

`DispatcherDirective.required_arguments = 1`

DispatcherDocumenter

class DispatcherDocumenter (*directive, name, indent=""*)
Specialized Documenter subclass for dispatchers.

Methods

<code>__init__</code>	Initialize self.
<code>add_content</code>	Add content from docstrings, attribute documentation and user.
<code>add_directive_header</code>	Add the directive header and options to the generated content.
<code>add_line</code>	Append one line of generated reST to the output.
<code>can_document_member</code>	Called to see if a member can be documented by this documenter.
<code>check_module</code>	Check if <i>self.object</i> is really defined in the module given by <i>self.modname</i> .
<code>document_members</code>	Generate reST for member documentation.
<code>filter_members</code>	Filter the given member list.
<code>format_args</code>	Format the argument signature of <i>self.object</i> .
<code>format_name</code>	Format the name of <i>self.object</i> .
<code>format_signature</code>	Format the signature (arguments and return annotation) of the object.
<code>generate</code>	Generate reST for the object given by <i>self.name</i> , and possibly for its members.
<code>get_attr</code>	getattr() override for types such as Zope interfaces.
<code>get_doc</code>	Decode and return lines of the docstring(s) for the object.
<code>get_object_members</code>	Return (<i>members_check_module</i> , <i>members</i>) where <i>members</i> is a list of (<i>membername</i> , <i>member</i>) pairs of the members of <i>self.object</i> .
<code>get_real_modname</code>	Get the real module name of an object to document.
<code>get_sourcename</code>	
<code>import_object</code>	Import the object given by <i>self.modname</i> and <i>self.objpath</i> and set it as <i>self.object</i> .

Continued on next page

Table 74 – continued from previous page

<code>parse_name</code>	Determine what module to import and what attribute to document.
<code>process_doc</code>	Let the user process the docstrings before adding them.
<code>resolve_name</code>	Resolve the module and name of the object to document given by the arguments and the current module/class.

`__init__`

`DispatcherDocumenter.__init__(directive, name, indent=)`
 Initialize self. See `help(type(self))` for accurate signature.

`add_content`

`DispatcherDocumenter.add_content(more_content, no_docstring=False)`
 Add content from docstrings, attribute documentation and user.

`add_directive_header`

`DispatcherDocumenter.add_directive_header(sig)`
 Add the directive header and options to the generated content.

`add_line`

`DispatcherDocumenter.add_line(line, source, *lineno)`
 Append one line of generated reST to the output.

`can_document_member`

classmethod `DispatcherDocumenter.can_document_member(member, *args, **kwargs)`
 Called to see if a member can be documented by this documenter.

`check_module`

`DispatcherDocumenter.check_module()`
 Check if `self.object` is really defined in the module given by `self.modname`.

`document_members`

`DispatcherDocumenter.document_members(all_members=False)`
 Generate reST for member documentation.
 If `all_members` is True, do all members, else those given by `self.options.members`.

filter_members

`DispatcherDocumenter.filter_members (members, want_all)`

Filter the given member list.

Members are skipped if

- they are private (except if given explicitly or the private-members option is set)
- they are special methods (except if given explicitly or the special-members option is set)
- they are undocumented (except if the undoc-members option is set)

The user can override the skipping decision by connecting to the `autodoc-skip-member` event.

format_args

`DispatcherDocumenter.format_args ()`

Format the argument signature of *self.object*.

Should return `None` if the object does not have a signature.

format_name

`DispatcherDocumenter.format_name ()`

Format the name of *self.object*.

This normally should be something that can be parsed by the generated directive, but doesn't need to be (Sphinx will display it unparsed then).

format_signature

`DispatcherDocumenter.format_signature ()`

Format the signature (arguments and return annotation) of the object.

Let the user process it via the `autodoc-process-signature` event.

generate

`DispatcherDocumenter.generate (more_content=None, **kw)`

Generate reST for the object given by *self.name*, and possibly for its members.

If *more_content* is given, include that content. If *real_modname* is given, use that module name to find attribute docs. If *check_module* is `True`, only generate if the object is defined in the module name it is imported from. If *all_members* is `True`, document all members.

get_attr

`DispatcherDocumenter.get_attr (obj, name, *defargs)`

`getattr()` override for types such as Zope interfaces.

get_doc

`DispatcherDocumenter.get_doc(encoding=None, ignore=1)`

Decode and return lines of the docstring(s) for the object.

get_object_members

`DispatcherDocumenter.get_object_members(want_all)`

Return (*members_check_module*, *members*) where *members* is a list of (*membername*, *member*) pairs of the members of *self.object*.

If *want_all* is True, return all members. Else, only return those members given by *self.options.members* (which may also be none).

get_real_modname

`DispatcherDocumenter.get_real_modname()`

Get the real module name of an object to document.

It can differ from the name of the module through which the object was imported.

get_sourcename

`DispatcherDocumenter.get_sourcename()`

import_object

`DispatcherDocumenter.import_object()`

Import the object given by *self.modname* and *self.objpath* and set it as *self.object*.

Returns True if successful, False if an error occurred.

parse_name

`DispatcherDocumenter.parse_name()`

Determine what module to import and what attribute to document.

Returns True and sets *self.modname*, *self.objpath*, *self.fullname*, *self.args* and *self.retann* if parsing and resolving was successful.

process_doc

`DispatcherDocumenter.process_doc(docstrings)`

Let the user process the docstrings before adding them.

resolve_name

DispatcherDocumenter.**resolve_name** (*modname, parents, path, base*)

Resolve the module and name of the object to document given by the arguments and the current module/class.

Must return a pair of the module name and a chain of attributes; for example, it would return ('zipfile', ['ZipFile', 'open']) for the zipfile.ZipFile.open method.

__init__ (*directive, name, indent=*"

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

Attributes

blue_cache	
code	
content_indent	
default_opt	
directivetype	
documenters	Returns registered Documenter classes
is_doctest	
member_order	
objtype	
option_spec	
priority	
titles_allowed	

blue_cache

DispatcherDocumenter.**blue_cache** = {}

code

DispatcherDocumenter.**code** = None

content_indent

DispatcherDocumenter.**content_indent** = ''

default_opt

DispatcherDocumenter.**default_opt** = {'depth': 0, 'view': False}

directivetype

DispatcherDocumenter.**directivetype** = 'data'

documenters

`DispatcherDocumenter.documenters`
Returns registered Documenter classes

is_doctest

`DispatcherDocumenter.is_doctest = False`

member_order

`DispatcherDocumenter.member_order = 40`

objtype

`DispatcherDocumenter.objtype = 'dispatcher'`

option_spec

`DispatcherDocumenter.option_spec = {'annotation': <function annotation_option>, 'code`

priority

`DispatcherDocumenter.priority = -10`

titles_allowed

`DispatcherDocumenter.titles_allowed = False`

`get_real_modname()`

Get the real module name of an object to document.

It can differ from the name of the module through which the object was imported.

`classmethod can_document_member(member, *args, **kwargs)`

Called to see if a member can be documented by this documenter.

`add_directive_header(sig)`

Add the directive header and options to the generated content.

`parse_name()`

Determine what module to import and what attribute to document.

Returns True and sets `self.modname`, `self.objpath`, `self.fullname`, `self.args` and `self.retann` if parsing and resolving was successful.

`generate(more_content=None, **kw)`

Generate reST for the object given by `self.name`, and possibly for its members.

If `more_content` is given, include that content. If `real_modname` is given, use that module name to find attribute docs. If `check_module` is True, only generate if the object is defined in the module name it is imported from. If `all_members` is True, document all members.

import_object ()
 Import the object given by *self.modname* and *self.objpath* and set it as *self.object*.
 Returns True if successful, False if an error occurred.

format_signature ()
 Format the signature (arguments and return annotation) of the object.
 Let the user process it via the `autodoc-process-signature` event.

add_content (*more_content*, *no_docstring=False*)
 Add content from docstrings, attribute documentation and user.

graphviz

Dispatcher directive.

Functions

<i>html_visit_dispatcher</i>	
<i>setup</i>	Setup <i>dsp</i> Sphinx extension module.
<i>warn_for_deprecated_option</i>	

html_visit_dispatcher

html_visit_dispatcher (*self*, *node*)

setup

setup (*app*)
 Setup *dsp* Sphinx extension module.

warn_for_deprecated_option

warn_for_deprecated_option (**args*, ***kwargs*)

Classes

<i>Dispatcher</i>
<i>dsp</i>
<i>img</i>

Dispatcher

class Dispatcher (*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Methods

<code>__init__</code>	Initialize self.
<code>add_name</code>	Append self.options['name'] to node['names'] if it exists.
<code>assert_has_content</code>	Throw an ERROR-level DirectiveError if the directive doesn't have contents.
<code>debug</code>	
<code>directive_error</code>	Return a DirectiveError suitable for being thrown as an exception.
<code>error</code>	
<code>info</code>	
<code>run</code>	
<code>severe</code>	
<code>warning</code>	

`__init__`

`Dispatcher.__init__(name, arguments, options, content, lineno, content_offset, block_text, state, state_machine)`
 Initialize self. See help(type(self)) for accurate signature.

`add_name`

`Dispatcher.add_name(node)`
 Append self.options['name'] to node['names'] if it exists.
 Also normalize the name string and register it as explicit target.

`assert_has_content`

`Dispatcher.assert_has_content()`
 Throw an ERROR-level DirectiveError if the directive doesn't have contents.

`debug`

`Dispatcher.debug(message)`

`directive_error`

`Dispatcher.directive_error(level, message)`
 Return a DirectiveError suitable for being thrown as an exception.
 Call “raise self.directive_error(level, message)” from within a directive implementation to return one single system message at level *level*, which automatically gets the directive block and the line number added.
 Preferably use the *debug*, *info*, *warning*, *error*, or *severe* wrapper methods, e.g. `self.error(message)` to generate an ERROR-level directive error.

error

Dispatcher.**error** (*message*)

info

Dispatcher.**info** (*message*)

run

Dispatcher.**run** ()

severe

Dispatcher.**severe** (*message*)

warning

Dispatcher.**warning** (*message*)

__init__ (*name, arguments, options, content, lineno, content_offset, block_text, state, state_machine*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

<code>config</code>	Reference to the Config object.
<code>env</code>	Reference to the BuildEnvironment object.
<code>final_argument_whitespace</code>	
<code>has_content</code>	
<code>img_opt</code>	
<code>option_spec</code>	
<code>optional_arguments</code>	
<code>required_arguments</code>	

config

Dispatcher.**config**
Reference to the Config object.

env

Dispatcher.**env**
Reference to the BuildEnvironment object.

final_argument_whitespace

```
Dispatcher.final_argument_whitespace = False
```

has_content

```
Dispatcher.has_content = True
```

img_opt

```
Dispatcher.img_opt = {'height': <function length_or_unitless>, 'width': <function length_or_unitless>}
```

option_spec

```
Dispatcher.option_spec = {'align': <function align_spec>, 'alt': <function unchanged_spec>, 'type': <function type_spec>}
```

optional_arguments

```
Dispatcher.optional_arguments = 1
```

required_arguments

```
Dispatcher.required_arguments = 1
```

dsp

```
class dsp (rawsource="", *children, **attributes)
```

Methods

<code>__init__</code>	Initialize self.
<code>append</code>	
<code>append_attr_list</code>	For each element in values, if it does not exist in self[attr], append it.
<code>asdom</code>	Return a DOM fragment representation of this Node.
<code>astext</code>	
<code>attlist</code>	
<code>clear</code>	
<code>coerce_append_attr_list</code>	First, convert both self[attr] and value to a non-string sequence type; if either is not already a sequence, convert it to a list of one element.
<code>copy</code>	

Continued on next page

Table 80 – continued from previous page

<code>copy_attr_coerce</code>	If <code>attr</code> is an attribute of <code>self</code> and either <code>self[attr]</code> or <code>value</code> is a list, convert all non-sequence values to a sequence of 1 element and then concatenate the two sequence, setting the result to <code>self[attr]</code> .
<code>copy_attr_concatenate</code>	If <code>attr</code> is an attribute of <code>self</code> and both <code>self[attr]</code> and <code>value</code> are lists, concatenate the two sequences, setting the result to <code>self[attr]</code> .
<code>copy_attr_consistent</code>	If <code>replace</code> is <code>True</code> or <code>selfpattr</code> is <code>None</code> , replace <code>self[attr]</code> with <code>value</code> .
<code>copy_attr_convert</code>	If <code>attr</code> is an attribute of <code>self</code> , set <code>self[attr]</code> to <code>[self[attr], value]</code> , otherwise set <code>self[attr]</code> to <code>value</code> .
<code>deepcopy</code>	Return a deep copy of <code>self</code> (also copying children).
<code>delattr</code>	
<code>emptytag</code>	
<code>endtag</code>	
<code>extend</code>	
<code>first_child_matching_class</code>	Return the index of the first child whose class exactly matches.
<code>first_child_not_matching_class</code>	Return the index of the first child whose class does <i>not</i> match.
<code>get</code>	
<code>get_language_code</code>	Return node’s language tag.
<code>has_key</code>	
<code>hasattr</code>	
<code>index</code>	
<code>insert</code>	
<code>is_not_default</code>	
<code>is_not_known_attribute</code>	Returns <code>True</code> if and only if the given attribute is NOT recognized by this class.
<code>is_not_list_attribute</code>	Returns <code>True</code> if and only if the given attribute is NOT one of the basic list attributes defined for all Elements.
<code>next_node</code>	Return the first node in the iterable returned by <code>traverse()</code> , or <code>None</code> if the iterable is empty.
<code>non_default_attributes</code>	
<code>note_referenced_by</code>	Note that this Element has been referenced by its name <i>name</i> or id <i>id</i> .
<code>pformat</code>	Return an indented pseudo-XML representation, for test purposes.
<code>pop</code>	
<code>remove</code>	
<code>replace</code>	Replace one child <i>Node</i> with another child or children.
<code>replace_attr</code>	If <code>self[attr]</code> does not exist or <code>force</code> is <code>True</code> or omitted, set <code>self[attr]</code> to <code>value</code> , otherwise do nothing.
<code>replace_self</code>	Replace <i>self</i> node with <i>new</i> , where <i>new</i> is a node or a list of nodes.
<code>set_class</code>	Add a new class to the “classes” attribute.
<code>setdefault</code>	
<code>setup_child</code>	
<code>shortrepr</code>	

Continued on next page

Table 80 – continued from previous page

<code>starttag</code>	
<code>traverse</code>	Return an iterable containing
<code>update_all_atts</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_coercion</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_concatenating</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_consistently</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_convert</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_basic_atts</code>	Update basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') from node or dictionary <i>dict_</i> .
<code>walk</code>	Traverse a tree of <i>Node</i> objects, calling the <i>dispatch_visit()</i> method of <i>visitor</i> when entering each node.
<code>walkabout</code>	Perform a tree traversal similarly to <i>Node.walk()</i> (which see), except also call the <i>dispatch_departure()</i> method before exiting each node.

`__init__`

`dsp.__init__(rawsource="", *children, **attributes)`
Initialize self. See `help(type(self))` for accurate signature.

`append`

`dsp.append(item)`

`append_attr_list`

`dsp.append_attr_list(attr, values)`
For each element in *values*, if it does not exist in *self[attr]*, append it.

NOTE: Requires *self[attr]* and *values* to be sequence type and the former should specifically be a list.

`asdom`

`dsp.asdom(dom=None)`
Return a DOM **fragment** representation of this Node.

`astext`

`dsp.astext()`

`attlist`

`dsp.attlist()`

clear

`dsp.clear()`

coerce_append_attr_list

`dsp.coerce_append_attr_list(attr, value)`

First, convert both `self[attr]` and `value` to a non-string sequence type; if either is not already a sequence, convert it to a list of one element. Then call `append_attr_list`.

NOTE: `self[attr]` and `value` both must not be `None`.

copy

`dsp.copy()`

copy_attr_coerce

`dsp.copy_attr_coerce(attr, value, replace)`

If `attr` is an attribute of `self` and either `self[attr]` or `value` is a list, convert all non-sequence values to a sequence of 1 element and then concatenate the two sequence, setting the result to `self[attr]`. If both `self[attr]` and `value` are non-sequences and `replace` is `True` or `self[attr]` is `None`, replace `self[attr]` with `value`. Otherwise, do nothing.

copy_attr_concatenate

`dsp.copy_attr_concatenate(attr, value, replace)`

If `attr` is an attribute of `self` and both `self[attr]` and `value` are lists, concatenate the two sequences, setting the result to `self[attr]`. If either `self[attr]` or `value` are non-sequences and `replace` is `True` or `self[attr]` is `None`, replace `self[attr]` with `value`. Otherwise, do nothing.

copy_attr_consistent

`dsp.copy_attr_consistent(attr, value, replace)`

If `replace` is `True` or `self[attr]` is `None`, replace `self[attr]` with `value`. Otherwise, do nothing.

copy_attr_convert

`dsp.copy_attr_convert(attr, value, replace=True)`

If `attr` is an attribute of `self`, set `self[attr]` to `[self[attr], value]`, otherwise set `self[attr]` to `value`.

NOTE: `replace` is not used by this function and is kept only for compatibility with the other copy functions.

deepcopy

`dsp.deepcopy()`

Return a deep copy of `self` (also copying children).

delattr

`dsp.delattr(attr)`

emptytag

`dsp.emptytag()`

endtag

`dsp.endtag()`

extend

`dsp.extend(item)`

first_child_matching_class

`dsp.first_child_matching_class(childclass, start=0, end=9223372036854775807)`

Return the index of the first child whose class exactly matches.

Parameters:

- *childclass*: A *Node* subclass to search for, or a tuple of *Node* classes. If a tuple, any of the classes may match.
- *start*: Initial index to check.
- *end*: Initial index to *not* check.

first_child_not_matching_class

`dsp.first_child_not_matching_class(childclass, start=0, end=9223372036854775807)`

Return the index of the first child whose class does *not* match.

Parameters:

- *childclass*: A *Node* subclass to skip, or a tuple of *Node* classes. If a tuple, none of the classes may match.
- *start*: Initial index to check.
- *end*: Initial index to *not* check.

get

`dsp.get(key, failobj=None)`

get_language_code

`dsp.get_language_code (fallback=)`

Return node's language tag.

Look iteratively in self and parents for a class argument starting with `language-` and return the remainder of it (which should be a *BCP49* language tag) or the *fallback*.

has_key

`dsp.has_key (attr)`

hasattr

`dsp.hasattr (attr)`

index

`dsp.index (item)`

insert

`dsp.insert (index, item)`

is_not_default

`dsp.is_not_default (key)`

is_not_known_attribute

classmethod `dsp.is_not_known_attribute (attr)`

Returns True if and only if the given attribute is NOT recognized by this class.

is_not_list_attribute

classmethod `dsp.is_not_list_attribute (attr)`

Returns True if and only if the given attribute is NOT one of the basic list attributes defined for all Elements.

next_node

`dsp.next_node (condition=None, include_self=False, descend=True, siblings=False, ascend=False)`

Return the first node in the iterable returned by `traverse()`, or None if the iterable is empty.

Parameter list is the same as of `traverse`. Note that `include_self` defaults to 0, though.

non_default_attributes

`dsp.non_default_attributes()`

note_referenced_by

`dsp.note_referenced_by(name=None, id=None)`

Note that this Element has been referenced by its name *name* or id *id*.

pformat

`dsp.pformat(indent=' ', level=0)`

Return an indented pseudo-XML representation, for test purposes.

Override in subclasses.

pop

`dsp.pop(i=-1)`

remove

`dsp.remove(item)`

replace

`dsp.replace(old, new)`

Replace one child *Node* with another child or children.

replace_attr

`dsp.replace_attr(attr, value, force=True)`

If `self[attr]` does not exist or `force` is `True` or omitted, set `self[attr]` to `value`, otherwise do nothing.

replace_self

`dsp.replace_self(new)`

Replace *self* node with *new*, where *new* is a node or a list of nodes.

set_class

`dsp.set_class(name)`

Add a new class to the “classes” attribute.

setdefault

`dsp.setdefault (key, failobj=None)`

setup_child

`dsp.setup_child (child)`

shortrepr

`dsp.shortrepr ()`

starttag

`dsp.starttag (quoteattr=None)`

traverse

`dsp.traverse (condition=None, include_self=True, descend=True, siblings=False, ascend=False)`

Return an iterable containing

- self (if `include_self` is true)
- all descendants in tree traversal order (if `descend` is true)
- all siblings (if `siblings` is true) and their descendants (if also `descend` is true)
- the siblings of the parent (if `ascend` is true) and their descendants (if also `descend` is true), and so on

If *condition* is not `None`, the iterable contains only nodes for which `condition(node)` is true. If *condition* is a node class `cls`, it is equivalent to a function consisting of `return isinstance(node, cls)`.

If `ascend` is true, assume `siblings` to be true as well.

For example, given the following tree:

```
<paragraph>
  <emphasis>      <--- emphasis.traverse() and
    <strong>      <--- strong.traverse() are called.
      Foo
      Bar
  <reference name="Baz" refid="baz">
    Baz
```

Then `list(emphasis.traverse())` equals

```
[<emphasis>, <strong>, <#text: Foo>, <#text: Bar>]
```

and `list(strong.traverse(ascend=True))` equals

```
[<strong>, <#text: Foo>, <#text: Bar>, <reference>, <#text: Baz>]
```

update_all_atts

`dsp.update_all_atts(dict_, update_fun=<function Element.copy_attr_consistent>, replace=True, and_source=False)`

Updates all attributes from node or dictionary `dict_`.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in `dict_`, updates the same attribute in self. When attributes with the same identifier appear in both self and `dict_`, the two values are merged based on the value of `update_fun`. Generally, when `replace` is True, the values in self are replaced or merged with the values in `dict_`; otherwise, the values in self may be preserved or merged. When `and_source` is True, the 'source' attribute is included in the copy.

NOTE: When `replace` is False, and self contains a 'source' attribute, 'source' is not replaced even when `dict_` has a 'source' attribute, though it may still be merged into a list depending on the value of `update_fun`.

NOTE: It is easier to call the update-specific methods then to pass the `update_fun` method to this function.

update_all_atts_coercion

`dsp.update_all_atts_coercion(dict_, replace=True, and_source=False)`

Updates all attributes from node or dictionary `dict_`.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in `dict_`, updates the same attribute in self. When attributes with the same identifier appear in both self and `dict_` whose values are both not lists and `replace` is True, the values in self are replaced with the values in `dict_`; if either of the values from self and `dict_` for the given identifier are of list type, then first any non-lists are converted to 1-element lists and then the two lists are concatenated and the result stored in self; otherwise, the values in self are preserved. When `and_source` is True, the 'source' attribute is included in the copy.

NOTE: When `replace` is False, and self contains a 'source' attribute, 'source' is not replaced even when `dict_` has a 'source' attribute, though it may still be merged into a list depending on the value of `update_fun`.

update_all_atts_concatenating

`dsp.update_all_atts_concatenating(dict_, replace=True, and_source=False)`

Updates all attributes from node or dictionary `dict_`.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in `dict_`, updates the same attribute in self. When attributes with the same identifier appear in both self and `dict_` whose values aren't each lists and `replace` is True, the values in self are replaced with the values in `dict_`; if the values from self and `dict_` for the given identifier are both of list type, then the two lists are concatenated and the result stored in self; otherwise, the values in self are preserved. When `and_source` is True, the 'source' attribute is included in the copy.

NOTE: When `replace` is False, and self contains a 'source' attribute, 'source' is not replaced even when `dict_` has a 'source' attribute, though it may still be merged into a list depending on the value of `update_fun`.

update_all_atts_consistently

`dsp.update_all_atts_consistently(dict_, replace=True, and_source=False)`

Updates all attributes from node or dictionary *dict_*.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in **dict_**, updates the same attribute in self. When attributes with the same identifier appear in both self and **dict_** and replace is True, the values in self are replaced with the values in **dict_**; otherwise, the values in self are preserved. When *and_source* is True, the 'source' attribute is included in the copy.

NOTE: When replace is False, and self contains a 'source' attribute, 'source' is not replaced even when **dict_ has a 'source' attribute, though it may still be merged into a list depending on the value of *update_fun*.**

update_all_atts_convert

`dsp.update_all_atts_convert(dict_, and_source=False)`

Updates all attributes from node or dictionary *dict_*.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in **dict_**, updates the same attribute in self. When attributes with the same identifier appear in both self and **dict_** then first any non-lists are converted to 1-element lists and then the two lists are concatenated and the result stored in self; otherwise, the values in self are preserved. When *and_source* is True, the 'source' attribute is included in the copy.

NOTE: When replace is False, and self contains a 'source' attribute, 'source' is not replaced even when **dict_ has a 'source' attribute, though it may still be merged into a list depending on the value of *update_fun*.**

update_basic_atts

`dsp.update_basic_atts(dict_)`

Update basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') from node or dictionary *dict_*.

walk

`dsp.walk(visitor)`

Traverse a tree of *Node* objects, calling the *dispatch_visit()* method of *visitor* when entering each node. (The *walkabout()* method is similar, except it also calls the *dispatch_departure()* method before exiting each node.)

This tree traversal supports limited in-place tree modifications. Replacing one node with one or more nodes is OK, as is removing an element. However, if the node removed or replaced occurs after the current node, the old node will still be traversed, and any new nodes will not.

Within *visit* methods (and *depart* methods for *walkabout()*), *TreePruningException* subclasses may be raised (*SkipChildren*, *SkipSiblings*, *SkipNode*, *SkipDeparture*).

Parameter *visitor*: A *NodeVisitor* object, containing a *visit* implementation for each *Node* subclass encountered.

Return true if we should stop the traversal.

walkabout

`dsp.walkabout(visitor)`

Perform a tree traversal similarly to *Node.walk()* (which see), except also call the *dispatch_departure()* method before exiting each node.

Parameter *visitor*: A *NodeVisitor* object, containing a *visit* and *depart* implementation for each *Node* subclass encountered.

Return true if we should stop the traversal.

`__init__(rawsource="", *children, **attributes)`

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

Attributes

<code>basic_attributes</code>
<code>child_text_separator</code>
<code>document</code>
<code>known_attributes</code>
<code>line</code>
<code>list_attributes</code>
<code>local_attributes</code>
<code>parent</code>
<code>source</code>
<code>tagname</code>

basic_attributes

`dsp.basic_attributes = ('ids', 'classes', 'names', 'dupnames')`

child_text_separator

`dsp.child_text_separator = '\n\n'`

document

`dsp.document = None`

known_attributes

`dsp.known_attributes = ('ids', 'classes', 'names', 'dupnames', 'backrefs', 'source')`

line

`dsp.line = None`

list_attributes

```
dsp.list_attributes = ('ids', 'classes', 'names', 'dupnames', 'backrefs')
```

local_attributes

```
dsp.local_attributes = ('backrefs',)
```

parent

```
dsp.parent = None
```

source

```
dsp.source = None
```

tagname

```
dsp.tagname = None
```

img

```
class img (rawsource="", *children, **attributes)
```

Methods

<code>__init__</code>	Initialize self.
<code>append</code>	
<code>append_attr_list</code>	For each element in values, if it does not exist in self[attr], append it.
<code>asdom</code>	Return a DOM fragment representation of this Node.
<code>astext</code>	
<code>attlist</code>	
<code>clear</code>	
<code>coerce_append_attr_list</code>	First, convert both self[attr] and value to a non-string sequence type; if either is not already a sequence, convert it to a list of one element.
<code>copy</code>	
<code>copy_attr_coerce</code>	If attr is an attribute of self and either self[attr] or value is a list, convert all non-sequence values to a sequence of 1 element and then concatenate the two sequence, setting the result to self[attr].
<code>copy_attr_concatenate</code>	If attr is an attribute of self and both self[attr] and value are lists, concatenate the two sequences, setting the result to self[attr].

Continued on next page

Table 82 – continued from previous page

<code>copy_attr_consistent</code>	If <code>replace</code> is <code>True</code> or <code>self.pattr</code> is <code>None</code> , replace <code>self[attr]</code> with <code>value</code> .
<code>copy_attr_convert</code>	If <code>attr</code> is an attribute of <code>self</code> , set <code>self[attr]</code> to <code>[self[attr], value]</code> , otherwise set <code>self[attr]</code> to <code>value</code> .
<code>deepcopy</code>	Return a deep copy of <code>self</code> (also copying children).
<code>delattr</code>	
<code>emptytag</code>	
<code>endtag</code>	
<code>extend</code>	
<code>first_child_matching_class</code>	Return the index of the first child whose class exactly matches.
<code>first_child_not_matching_class</code>	Return the index of the first child whose class does <i>not</i> match.
<code>get</code>	
<code>get_language_code</code>	Return node’s language tag.
<code>has_key</code>	
<code>hasattr</code>	
<code>index</code>	
<code>insert</code>	
<code>is_not_default</code>	
<code>is_not_known_attribute</code>	Returns <code>True</code> if and only if the given attribute is NOT recognized by this class.
<code>is_not_list_attribute</code>	Returns <code>True</code> if and only if the given attribute is NOT one of the basic list attributes defined for all Elements.
<code>next_node</code>	Return the first node in the iterable returned by <code>traverse()</code> , or <code>None</code> if the iterable is empty.
<code>non_default_attributes</code>	
<code>note_referenced_by</code>	Note that this Element has been referenced by its name <i>name</i> or id <i>id</i> .
<code>pformat</code>	Return an indented pseudo-XML representation, for test purposes.
<code>pop</code>	
<code>remove</code>	
<code>replace</code>	Replace one child <i>Node</i> with another child or children.
<code>replace_attr</code>	If <code>self[attr]</code> does not exist or <code>force</code> is <code>True</code> or omitted, set <code>self[attr]</code> to <code>value</code> , otherwise do nothing.
<code>replace_self</code>	Replace <i>self</i> node with <i>new</i> , where <i>new</i> is a node or a list of nodes.
<code>set_class</code>	Add a new class to the “classes” attribute.
<code>setdefault</code>	
<code>setup_child</code>	
<code>shortrepr</code>	
<code>starttag</code>	
<code>traverse</code>	Return an iterable containing
<code>update_all_atts</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_coercion</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_concatenating</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_consistently</code>	Updates all attributes from node or dictionary <i>dict_</i> .
<code>update_all_atts_convert</code>	Updates all attributes from node or dictionary <i>dict_</i> .

Continued on next page

Table 82 – continued from previous page

update_basic_atts	Update basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') from node or dictionary <i>dict_</i> .
walk	Traverse a tree of <i>Node</i> objects, calling the <i>dispatch_visit()</i> method of <i>visitor</i> when entering each node.
walkabout	Perform a tree traversal similarly to <i>Node.walk()</i> (which see), except also call the <i>dispatch_departure()</i> method before exiting each node.

`__init__`

`img.__init__(rawsource="", *children, **attributes)`
Initialize self. See `help(type(self))` for accurate signature.

`append`

`img.append(item)`

`append_attr_list`

`img.append_attr_list(attr, values)`
For each element in *values*, if it does not exist in `self[attr]`, append it.
NOTE: Requires `self[attr]` and *values* to be sequence type and the former should specifically be a list.

`asdom`

`img.asdom(dom=None)`
Return a DOM **fragment** representation of this Node.

`astext`

`img.astext()`

`attlist`

`img.attlist()`

`clear`

`img.clear()`

coerce_append_attr_list

`img.coerce_append_attr_list (attr, value)`

First, convert both `self[attr]` and `value` to a non-string sequence type; if either is not already a sequence, convert it to a list of one element. Then call `append_attr_list`.

NOTE: `self[attr]` and `value` both must not be `None`.

copy

`img.copy ()`

copy_attr_coerce

`img.copy_attr_coerce (attr, value, replace)`

If `attr` is an attribute of `self` and either `self[attr]` or `value` is a list, convert all non-sequence values to a sequence of 1 element and then concatenate the two sequence, setting the result to `self[attr]`. If both `self[attr]` and `value` are non-sequences and `replace` is `True` or `self[attr]` is `None`, replace `self[attr]` with `value`. Otherwise, do nothing.

copy_attr_concatenate

`img.copy_attr_concatenate (attr, value, replace)`

If `attr` is an attribute of `self` and both `self[attr]` and `value` are lists, concatenate the two sequences, setting the result to `self[attr]`. If either `self[attr]` or `value` are non-sequences and `replace` is `True` or `self[attr]` is `None`, replace `self[attr]` with `value`. Otherwise, do nothing.

copy_attr_consistent

`img.copy_attr_consistent (attr, value, replace)`

If `replace` is `True` or `self[attr]` is `None`, replace `self[attr]` with `value`. Otherwise, do nothing.

copy_attr_convert

`img.copy_attr_convert (attr, value, replace=True)`

If `attr` is an attribute of `self`, set `self[attr]` to `[self[attr], value]`, otherwise set `self[attr]` to `value`.

NOTE: `replace` is not used by this function and is kept only for compatibility with the other copy functions.

deepcopy

`img.deepcopy ()`

Return a deep copy of `self` (also copying children).

delattr

`img.delattr (attr)`

emptytag

`img.emptytag()`

endtag

`img.endtag()`

extend

`img.extend(item)`

first_child_matching_class

`img.first_child_matching_class(childclass, start=0, end=9223372036854775807)`

Return the index of the first child whose class exactly matches.

Parameters:

- *childclass*: A *Node* subclass to search for, or a tuple of *Node* classes. If a tuple, any of the classes may match.
- *start*: Initial index to check.
- *end*: Initial index to *not* check.

first_child_not_matching_class

`img.first_child_not_matching_class(childclass, start=0, end=9223372036854775807)`

Return the index of the first child whose class does *not* match.

Parameters:

- *childclass*: A *Node* subclass to skip, or a tuple of *Node* classes. If a tuple, none of the classes may match.
- *start*: Initial index to check.
- *end*: Initial index to *not* check.

get

`img.get(key, failobj=None)`

get_language_code

`img.get_language_code(fallback="")`

Return node's language tag.

Look iteratively in self and parents for a class argument starting with `language-` and return the remainder of it (which should be a *BCP49* language tag) or the *fallback*.

has_key

`img.has_key (attr)`

hasattr

`img.hasattr (attr)`

index

`img.index (item)`

insert

`img.insert (index, item)`

is_not_default

`img.is_not_default (key)`

is_not_known_attribute

classmethod `img.is_not_known_attribute (attr)`

Returns True if and only if the given attribute is NOT recognized by this class.

is_not_list_attribute

classmethod `img.is_not_list_attribute (attr)`

Returns True if and only if the given attribute is NOT one of the basic list attributes defined for all Elements.

next_node

`img.next_node (condition=None, include_self=False, descend=True, siblings=False, ascend=False)`

Return the first node in the iterable returned by `traverse()`, or None if the iterable is empty.

Parameter list is the same as of `traverse`. Note that `include_self` defaults to 0, though.

non_default_attributes

`img.non_default_attributes ()`

note_referenced_by

`img.note_referenced_by (name=None, id=None)`

Note that this Element has been referenced by its name *name* or id *id*.

pformat

`img.pformat (indent=' ', level=0)`

Return an indented pseudo-XML representation, for test purposes.

Override in subclasses.

pop

`img.pop (i=-1)`

remove

`img.remove (item)`

replace

`img.replace (old, new)`

Replace one child *Node* with another child or children.

replace_attr

`img.replace_attr (attr, value, force=True)`

If `self[attr]` does not exist or `force` is `True` or omitted, set `self[attr]` to `value`, otherwise do nothing.

replace_self

`img.replace_self (new)`

Replace *self* node with *new*, where *new* is a node or a list of nodes.

set_class

`img.set_class (name)`

Add a new class to the “classes” attribute.

setdefault

`img.setdefault (key, failobj=None)`

setup_child

```
img.setup_child(child)
```

shortrepr

```
img.shortrepr()
```

starttag

```
img.starttag(quoteattr=None)
```

traverse

```
img.traverse(condition=None, include_self=True, descend=True, siblings=False, ascend=False)
```

Return an iterable containing

- self (if include_self is true)
- all descendants in tree traversal order (if descend is true)
- all siblings (if siblings is true) and their descendants (if also descend is true)
- the siblings of the parent (if ascend is true) and their descendants (if also descend is true), and so on

If *condition* is not None, the iterable contains only nodes for which `condition(node)` is true. If *condition* is a node class `cls`, it is equivalent to a function consisting of `return isinstance(node, cls)`.

If ascend is true, assume siblings to be true as well.

For example, given the following tree:

```
<paragraph>
  <emphasis>      <--- emphasis.traverse() and
    <strong>      <--- strong.traverse() are called.
      Foo
      Bar
  <reference name="Baz" refid="baz">
    Baz
```

Then `list(emphasis.traverse())` equals

```
[<emphasis>, <strong>, <#text: Foo>, <#text: Bar>]
```

and `list(strong.traverse(ascend=True))` equals

```
[<strong>, <#text: Foo>, <#text: Bar>, <reference>, <#text: Baz>]
```

update_all_atts

```
img.update_all_atts(dict_, update_fun=<function Element.copy_attr_consistent>, re-
                        place=True, and_source=False)
Updates all attributes from node or dictionary dict_.
```


Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in **dict_**, updates the same attribute in self. When attributes with the same identifier appear in both self and **dict_**, the two values are merged based on the value of `update_fun`. Generally, when `replace` is True, the values in self are replaced or merged with the values in **dict_**; otherwise, the values in self may be preserved or merged. When `and_source` is True, the 'source' attribute is included in the copy.

NOTE: When `replace` is False, and self contains a 'source' attribute, 'source' is not replaced even when **dict_** has a 'source' attribute, though it may still be merged into a list depending on the value of `update_fun`.

NOTE: It is easier to call the update-specific methods then to pass the `update_fun` method to this function.

update_all_atts_coercion

`img.update_all_atts_coercion(dict_, replace=True, and_source=False)`

Updates all attributes from node or dictionary **dict_**.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in **dict_**, updates the same attribute in self. When attributes with the same identifier appear in both self and **dict_** whose values are both not lists and `replace` is True, the values in self are replaced with the values in **dict_**; if either of the values from self and **dict_** for the given identifier are of list type, then first any non-lists are converted to 1-element lists and then the two lists are concatenated and the result stored in self; otherwise, the values in self are preserved. When `and_source` is True, the 'source' attribute is included in the copy.

NOTE: When `replace` is False, and self contains a 'source' attribute, 'source' is not replaced even when **dict_** has a 'source' attribute, though it may still be merged into a list depending on the value of `update_fun`.

update_all_atts_concatenating

`img.update_all_atts_concatenating(dict_, replace=True, and_source=False)`

Updates all attributes from node or dictionary **dict_**.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in **dict_**, updates the same attribute in self. When attributes with the same identifier appear in both self and **dict_** whose values aren't each lists and `replace` is True, the values in self are replaced with the values in **dict_**; if the values from self and **dict_** for the given identifier are both of list type, then the two lists are concatenated and the result stored in self; otherwise, the values in self are preserved. When `and_source` is True, the 'source' attribute is included in the copy.

NOTE: When `replace` is False, and self contains a 'source' attribute, 'source' is not replaced even when **dict_** has a 'source' attribute, though it may still be merged into a list depending on the value of `update_fun`.

update_all_atts_consistently

`img.update_all_atts_consistently(dict_, replace=True, and_source=False)`

Updates all attributes from node or dictionary **dict_**.

Appends the basic attributes ('ids', 'names', 'classes', 'dupnames', but not 'source') and then, for all other attributes in **dict_**, updates the same attribute in self. When attributes with the same identifier appear in both self and **dict_** and `replace` is True, the values in self are replaced with the values in **dict_**; otherwise, the values in self are preserved. When `and_source` is True, the 'source' attribute is included in the copy.

NOTE: When replace is False, and self contains a ‘source’ attribute, ‘source’ is not replaced even when `dict_` has a ‘source’ attribute, though it may still be merged into a list depending on the value of `update_fun`.

update_all_atts_convert

`img.update_all_atts_convert(dict_, and_source=False)`

Updates all attributes from node or dictionary `dict_`.

Appends the basic attributes (‘ids’, ‘names’, ‘classes’, ‘dupnames’, but not ‘source’) and then, for all other attributes in `dict_`, updates the same attribute in self. When attributes with the same identifier appear in both self and `dict_` then first any non-lists are converted to 1-element lists and then the two lists are concatenated and the result stored in self; otherwise, the values in self are preserved. When `and_source` is True, the ‘source’ attribute is included in the copy.

NOTE: When replace is False, and self contains a ‘source’ attribute, ‘source’ is not replaced even when `dict_` has a ‘source’ attribute, though it may still be merged into a list depending on the value of `update_fun`.

update_basic_atts

`img.update_basic_atts(dict_)`

Update basic attributes (‘ids’, ‘names’, ‘classes’, ‘dupnames’, but not ‘source’) from node or dictionary `dict_`.

walk

`img.walk(visitor)`

Traverse a tree of *Node* objects, calling the *dispatch_visit()* method of *visitor* when entering each node. (The *walkabout()* method is similar, except it also calls the *dispatch_departure()* method before exiting each node.)

This tree traversal supports limited in-place tree modifications. Replacing one node with one or more nodes is OK, as is removing an element. However, if the node removed or replaced occurs after the current node, the old node will still be traversed, and any new nodes will not.

Within *visit* methods (and *depart* methods for *walkabout()*), *TreePruningException* subclasses may be raised (*SkipChildren*, *SkipSiblings*, *SkipNode*, *SkipDeparture*).

Parameter *visitor*: A *NodeVisitor* object, containing a *visit* implementation for each *Node* subclass encountered.

Return true if we should stop the traversal.

walkabout

`img.walkabout(visitor)`

Perform a tree traversal similarly to *Node.walk()* (which see), except also call the *dispatch_departure()* method before exiting each node.

Parameter *visitor*: A *NodeVisitor* object, containing a *visit* and *depart* implementation for each *Node* subclass encountered.

Return true if we should stop the traversal.

`__init__(rawsource="", *children, **attributes)`
 Initialize self. See help(type(self)) for accurate signature.

Attributes

<code>basic_attributes</code>
<code>child_text_separator</code>
<code>document</code>
<code>known_attributes</code>
<code>line</code>
<code>list_attributes</code>
<code>local_attributes</code>
<code>parent</code>
<code>source</code>
<code>tagname</code>

`basic_attributes`

```
img.basic_attributes = ('ids', 'classes', 'names', 'dupnames')
```

`child_text_separator`

```
img.child_text_separator = '\n\n'
```

`document`

```
img.document = None
```

`known_attributes`

```
img.known_attributes = ('ids', 'classes', 'names', 'dupnames', 'backrefs', 'source')
```

`line`

```
img.line = None
```

`list_attributes`

```
img.list_attributes = ('ids', 'classes', 'names', 'dupnames', 'backrefs')
```

`local_attributes`

```
img.local_attributes = ('backrefs',)
```

parent

`img.parent = None`

source

`img.source = None`

tagname

`img.tagname = 'img'`

Functions

setup

Setup *dispatcher* Sphinx extension module.

setup

setup (*app*)

Setup *dispatcher* Sphinx extension module.

8.1 v1.0.0 (2020-01-02)

8.1.1 Feat

- (doc): Add code of conduct.
- (examples): Add new example + formatting.
- (sol): New *raises* option, if *raises=*” no warning logs.
- (web): Add query param *data* to include/exclude data into the server JSON response.
- (sphinx): Update dispatcher documenter and directive.
- (drw): Add wildcard rendering.

8.1.2 Fix

- (test): Update test cases.
- (dsp): Correct pipe extraction for wildcards.
- (setup): Add missing *drw* files.

8.2 v0.3.7 (2019-12-06)

8.2.1 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.

8.2.2 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.

8.3 v0.3.6 (2019-10-18)

8.3.1 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*.

8.4 v0.3.4 (2019-07-15)

8.4.1 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.

8.4.2 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.

8.5 v0.3.3 (2019-04-02)

8.5.1 Feat

- (dispatcher): Improve error message.

8.5.2 Fix

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

8.6 v0.3.2 (2019-02-23)

8.6.1 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.

8.6.2 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.

8.7 v0.3.1 (2018-12-10)

8.7.1 Fix

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

8.8 v0.3.0 (2018-12-08)

8.8.1 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.

8.8.2 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.

8.8.3 Other

- Refact: Update documentation.

8.9 v0.2.8 (2018-10-09)

8.9.1 Feat

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

8.10 v0.2.7 (2018-09-13)

8.10.1 Fix

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

8.11 v0.2.6 (2018-09-13)

8.11.1 Feat

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

8.12 v0.2.5 (2018-09-13)

8.12.1 Fix

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

8.13 v0.2.4 (2018-09-13)

8.13.1 Fix

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

8.14 v0.2.3 (2018-08-02)

8.14.1 Fix

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

8.15 v0.2.2 (2018-08-02)

8.15.1 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.

8.16 v0.2.1 (2018-07-24)

8.16.1 Feat

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

8.16.2 Fix

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

8.17 v0.2.0 (2018-07-19)

8.17.1 Feat

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

8.17.2 Fix

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

8.18 v0.1.19 (2018-06-05)

8.18.1 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.

8.19 v0.1.18 (2018-05-28)

8.19.1 Feat

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

8.20 v0.1.17 (2018-05-18)

8.20.1 Feat

- (travis): Run coveralls in python 3.6.

8.20.2 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*.

8.21 v0.1.16 (2017-09-26)

8.21.1 Fix

- (requirements): Update dill requirements.

8.22 v0.1.15 (2017-09-26)

8.22.1 Fix

- (networkx): Update according to networkx 2.0.

8.23 v0.1.14 (2017-07-11)

8.23.1 Fix

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

8.23.2 Other

- Merge pull request [#9](#) from ankostis/fixabortex.

8.24 v0.1.13 (2017-06-26)

8.24.1 Feat

- (appveyor): Add python 3.6.

8.24.2 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.

8.25 v0.1.12 (2017-05-04)

8.25.1 Fix

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

8.26 v0.1.11 (2017-05-04)

8.26.1 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.

8.26.2 Fix

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

8.27 v0.1.10 (2017-04-03)

8.27.1 Feat

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

8.27.2 Fix

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

8.28 v0.1.9 (2017-02-09)

8.28.1 Fix

- (appveyor): Setup of lmx1.
- (drw): Update plot index.

8.29 v0.1.8 (2017-02-09)

8.29.1 Feat

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

8.30 v0.1.7 (2017-02-08)

8.30.1 Fix

- (setup): Add missing package_data.

8.31 v0.1.6 (2017-02-08)

8.31.1 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.

8.32 v0.1.5 (2017-02-06)

8.32.1 Feat

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

8.32.2 Fix

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

8.33 v0.1.4 (2017-01-31)

8.33.1 Feat

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

8.33.2 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.

8.34 v0.1.3 (2017-01-29)

8.34.1 Fix

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

8.35 v0.1.2 (2017-01-28)

8.35.1 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.

8.35.2 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.

8.36 v0.1.1 (2017-01-21)

8.36.1 Fix

- (site): Fix ResourceWarning: unclosed socket.
- (setup): Not log sphinx warnings for long_description.
- (travis): Wait until 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.

8.36.2 Other

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

CHAPTER 9

Indices and tables

- `genindex`
- `modindex`
- `search`

S

- `schedula`, [21](#)
- `schedula.dispatcher`, [21](#)
- `schedula.ext`, [221](#)
- `schedula.ext.autosummary`, [221](#)
- `schedula.ext.dispatcher`, [221](#)
- `schedula.ext.dispatcher.documenter`, [222](#)
- `schedula.ext.dispatcher.graphviz`, [231](#)
- `schedula.utils`, [79](#)
- `schedula.utils.alg`, [80](#)
- `schedula.utils.asy`, [87](#)
- `schedula.utils.base`, [95](#)
- `schedula.utils.blue`, [106](#)
- `schedula.utils.cst`, [121](#)
- `schedula.utils.des`, [122](#)
- `schedula.utils.drw`, [123](#)
- `schedula.utils.drw.nodes`, [123](#)
- `schedula.utils.dsp`, [136](#)
- `schedula.utils.exc`, [188](#)
- `schedula.utils.gen`, [188](#)
- `schedula.utils.io`, [198](#)
- `schedula.utils.sol`, [201](#)
- `schedula.utils.web`, [211](#)

Symbols

`__init__()` (*AsyncList* method), 91
`__init__()` (*Base* method), 100
`__init__()` (*BlueDispatcher* method), 115
`__init__()` (*Blueprint* method), 120
`__init__()` (*DispatchPipe* method), 152
`__init__()` (*Dispatcher* method), 56, 233
`__init__()` (*DispatcherDirective* method), 224
`__init__()` (*DispatcherDocumenter* method), 229
`__init__()` (*DspPipe* method), 87
`__init__()` (*Executor* method), 92
`__init__()` (*FolderNode* method), 126
`__init__()` (*FolderNodeWeb* method), 213
`__init__()` (*NoSub* method), 155
`__init__()` (*PoolExecutor* method), 93
`__init__()` (*ProcessExecutor* method), 93
`__init__()` (*ProcessPoolExecutor* method), 94
`__init__()` (*Site* method), 128
`__init__()` (*SiteFolder* method), 129
`__init__()` (*SiteIndex* method), 131
`__init__()` (*SiteMap* method), 134
`__init__()` (*SiteNode* method), 135
`__init__()` (*Solution* method), 210
`__init__()` (*SubDispatch* method), 165
`__init__()` (*SubDispatchFunction* method), 176
`__init__()` (*SubDispatchPipe* method), 186
`__init__()` (*ThreadExecutor* method), 94
`__init__()` (*Token* method), 198
`__init__()` (*WebFolder* method), 214
`__init__()` (*WebMap* method), 219
`__init__()` (*WebNode* method), 220
`__init__()` (*add_args* method), 187
`__init__()` (*dsp* method), 244
`__init__()` (*img* method), 254
`__init__()` (*inf* method), 187

A

`add_args` (class in *schedula.utils.dsp*), 186

`add_autodocumenter()` (in module *schedula.ext.dispatcher.documenter*), 222
`add_content()` (*DispatcherDocumenter* method), 231
`add_data()` (*BlueDispatcher* method), 115
`add_data()` (*Dispatcher* method), 57
`add_directive_header()` (*DispatcherDocumenter* method), 230
`add_dispatcher()` (*BlueDispatcher* method), 118
`add_dispatcher()` (*Dispatcher* method), 62
`add_edge_fun()` (in module *schedula.utils.alg*), 80
`add_from_lists()` (*BlueDispatcher* method), 119
`add_from_lists()` (*Dispatcher* method), 64
`add_func()` (*BlueDispatcher* method), 117
`add_func()` (*Dispatcher* method), 60
`add_func_edges()` (in module *schedula.utils.alg*), 81
`add_function()` (*BlueDispatcher* method), 116
`add_function()` (*Dispatcher* method), 59
`add_function()` (in module *schedula.utils.dsp*), 137
`are_in_nested_dicts()` (in module *schedula.utils.dsp*), 138
`async_process()` (in module *schedula.utils.asy*), 87
`async_thread()` (in module *schedula.utils.asy*), 88
`AsyncList` (class in *schedula.utils.asy*), 89
`autoplot_callback()` (in module *schedula.utils.drw*), 123
`autoplot_function()` (in module *schedula.utils.drw*), 123
`await_result()` (in module *schedula.utils.asy*), 88

B

`Base` (class in *schedula.utils.base*), 95
`basic_app()` (in module *schedula.utils.drw*), 123
`before_request()` (in module *schedula.utils.drw*), 124
`blue()` (*Dispatcher* method), 71
`blue()` (*SubDispatch* method), 166
`BlueDispatcher` (class in *schedula.utils.blue*), 107
`Blueprint` (class in *schedula.utils.blue*), 119

bypass() (in module *schedula.utils.dsp*), 139

C

cached_view() (in module *schedula.utils.drw*), 124

can_document_member()

(*schedula.ext.dispatcher.documenter.DispatcherDocumenter* class method), 230

cls (*Blueprint* attribute), 120

combine_dicts() (in module *schedula.utils.dsp*), 139

combine_nested_dicts() (in module *schedula.utils.dsp*), 139

contains_doctest() (in module *schedula.ext.dispatcher.documenter*), 222

copy() (*Dispatcher* method), 71

copy_structure() (*Dispatcher* method), 57

counter (*Dispatcher* attribute), 57

counter (*FolderNode* attribute), 127

counter (*SiteFolder* attribute), 130

counter (*SiteNode* attribute), 136

counter() (in module *schedula.utils.gen*), 189

D

data_nodes (*Dispatcher* attribute), 71

default_values (*Dispatcher* attribute), 57

dispatch() (*Dispatcher* method), 72

Dispatcher (class in *schedula.dispatcher*), 21

Dispatcher (class *schedula.ext.dispatcher.graphviz*), 231

DispatcherDirective (class *schedula.ext.dispatcher.documenter*), 222

DispatcherDocumenter (class *schedula.ext.dispatcher.documenter*), 225

DispatchPipe (class in *schedula.utils.dsp*), 144

dmap (*Dispatcher* attribute), 57

dsp (class in *schedula.ext.dispatcher.graphviz*), 234

DspPipe (class in *schedula.utils.alg*), 85

E

EMPTY (in module *schedula.utils.cst*), 121

END (in module *schedula.utils.cst*), 122

Executor (class in *schedula.utils.asy*), 91

executor (*Dispatcher* attribute), 57

extend() (*Blueprint* method), 121

extend() (*Dispatcher* method), 72

F

FolderNode (class in *schedula.utils.drw*), 125

FolderNodeWeb (class in *schedula.utils.web*), 211

format_signature() (*DispatcherDocumenter* method), 231

function_nodes (*Dispatcher* attribute), 71

G

generate() (*DispatcherDocumenter* method), 230

generate_autosummary_docs() (in module *schedula.ext.autosummary*), 221

get_attr_doc() (in module *schedula.utils.des*), 122

get_attr_doc() (in module *schedula.utils.alg*), 81

get_grandfather_content() (in module *schedula.ext.dispatcher.documenter*), 222

get_grandfather_offset() (in module *schedula.ext.dispatcher.documenter*), 222

get_link() (in module *schedula.utils.des*), 122

get_members() (in module *schedula.ext.autosummary*), 221

get_nested_dicts() (in module *schedula.utils.dsp*), 140

get_node() (*Base* method), 103

get_real_modname() (*DispatcherDocumenter* method), 230

get_sub_dsp() (*Dispatcher* method), 65

get_sub_dsp_from_workflow() (*Dispatcher* method), 68

get_sub_dsp_from_workflow() (*Solution* method), 211

get_sub_node() (in module *schedula.utils.alg*), 81

get_summary() (in module *schedula.utils.des*), 123

get_unused_node_id() (in module *schedula.utils.alg*), 84

in

H

in html_visit_dispatcher() (in module *schedula.ext.dispatcher.graphviz*), 231

in

I

img (class in *schedula.ext.dispatcher.graphviz*), 245

import_object() (*DispatcherDocumenter* method), 231

inf (class in *schedula.utils.dsp*), 187

J

jinja2_format() (in module *schedula.utils.drw*), 124

K

kk_dict() (in module *schedula.utils.dsp*), 140

L

load_default_values() (in module *schedula.utils.io*), 199

load_dispatcher() (in module *schedula.utils.io*), 199

load_map() (in module *schedula.utils.io*), 200

M

map_dict() (in module *schedula.utils.dsp*), 140

`map_list()` (in module *schedula.utils.dsp*), 141

N

`name` (*Dispatcher* attribute), 57

`nodes` (*Dispatcher* attribute), 57

`NONE` (in module *schedula.utils.cst*), 122

`NoSub` (class in *schedula.utils.dsp*), 155

P

`pairwise()` (in module *schedula.utils.gen*), 189

`parent_func()` (in module *schedula.utils.dsp*), 142

`parse_name()` (*DispatcherDocumenter* method), 230

`pipe` (*Solution* attribute), 211

`PLOT` (in module *schedula.utils.cst*), 122

`plot()` (*Base* method), 102

`plot()` (*DispatchPipe* method), 153

`PoolExecutor` (class in *schedula.utils.asy*), 92

`process_generate_options()` (in module *schedula.ext.autosummary*), 221

`ProcessExecutor` (class in *schedula.utils.asy*), 93

`ProcessPoolExecutor` (class in *schedula.utils.asy*), 93

R

`raises` (*Dispatcher* attribute), 57

`register()` (*Blueprint* method), 121

`register_executor()` (in module *schedula.utils.asy*), 88

`remove_edge_fun()` (in module *schedula.utils.alg*), 85

`render_output()` (in module *schedula.utils.drw*), 124

`replicate_value()` (in module *schedula.utils.dsp*), 142

`result()` (*Solution* method), 211

`run_server()` (in module *schedula.utils.drw*), 124

S

`save_default_values()` (in module *schedula.utils.io*), 200

`save_dispatcher()` (in module *schedula.utils.io*), 200

`save_map()` (in module *schedula.utils.io*), 201

`schedula` (module), 21

`schedula.dispatcher` (module), 21

`schedula.ext` (module), 221

`schedula.ext.autosummary` (module), 221

`schedula.ext.dispatcher` (module), 221

`schedula.ext.dispatcher.documenter` (module), 222

`schedula.ext.dispatcher.graphviz` (module), 231

`schedula.utils` (module), 79

`schedula.utils.alg` (module), 80

`schedula.utils.asy` (module), 87

`schedula.utils.base` (module), 95

`schedula.utils.blue` (module), 106

`schedula.utils.cst` (module), 121

`schedula.utils.des` (module), 122

`schedula.utils.drw` (module), 123

`schedula.utils.drw.nodes` (module), 123

`schedula.utils.dsp` (module), 136

`schedula.utils.exc` (module), 188

`schedula.utils.gen` (module), 188

`schedula.utils.io` (module), 198

`schedula.utils.sol` (module), 201

`schedula.utils.web` (module), 211

`search_node_description()` (in module *schedula.utils.des*), 123

`selector()` (in module *schedula.utils.dsp*), 142

`SELF` (in module *schedula.utils.cst*), 122

`set_default_value()` (*BlueDispatcher* method), 119

`set_default_value()` (*Dispatcher* method), 65

`setup()` (in module *schedula.ext.autosummary*), 221

`setup()` (in module *schedula.ext.dispatcher*), 256

`setup()` (in module *schedula.ext.dispatcher.documenter*), 222

`setup()` (in module *schedula.ext.dispatcher.graphviz*), 231

`shrink_dsp()` (*Dispatcher* method), 77

`shutdown_executor()` (in module *schedula.utils.asy*), 88

`shutdown_executors()` (in module *schedula.utils.asy*), 89

`SINK` (in module *schedula.utils.cst*), 122

`Site` (class in *schedula.utils.drw*), 127

`site_view()` (in module *schedula.utils.drw*), 124

`SiteFolder` (class in *schedula.utils.drw*), 128

`SiteIndex` (class in *schedula.utils.drw*), 130

`SiteMap` (class in *schedula.utils.drw*), 132

`SiteNode` (class in *schedula.utils.drw*), 135

`Solution` (class in *schedula.utils.sol*), 201

`solution` (*Dispatcher* attribute), 57

`stack_nested_keys()` (in module *schedula.utils.dsp*), 143

`START` (in module *schedula.utils.cst*), 121

`stlp()` (in module *schedula.utils.dsp*), 143

`sub_dsp_nodes` (*Dispatcher* attribute), 71

`SubDispatch` (class in *schedula.utils.dsp*), 155

`SubDispatchFunction` (class in *schedula.utils.dsp*), 166

`SubDispatchPipe` (class in *schedula.utils.dsp*), 176

`summation()` (in module *schedula.utils.dsp*), 143

T

`ThreadExecutor` (class in *schedula.utils.asy*), 94

`Token` (class in *schedula.utils.gen*), 189

U

`uncpath()` (*in module `schedula.utils.drw`*), [124](#)

`update_filenames()` (*in module `schedula.utils.drw`*), [124](#)

V

`valid_filename()` (*in module `schedula.utils.drw`*), [124](#)

W

`warn_for_deprecated_option()` (*in module `schedula.ext.dispatcher.graphviz`*), [231](#)

`web()` (*Base method*), [100](#)

`WebFolder` (*class in `schedula.utils.web`*), [214](#)

`WebMap` (*class in `schedula.utils.web`*), [216](#)

`WebNode` (*class in `schedula.utils.web`*), [219](#)