
schedula Documentation

Release 1.4.9

Vincenzo Arcidiacono

Jan 23, 2023

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 | 309 |
| 9 | Indices and tables | 333 |
| | Python Module Index | 335 |
| | Index | 337 |

2022-01-23 11:30:00

<https://github.com/vinci1it2000/schedula>

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

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

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

<https://github.com/vinci1it2000/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+

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.

INSTALLATION

To install it use (with root privileges):

```
$ pip install schedula
```

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

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

2.1 Install extras

Some additional functionality is enabled installing the following extras:

- `io`: enables to read/write functions.
- `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](#).

TUTORIAL

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):
...     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()
```

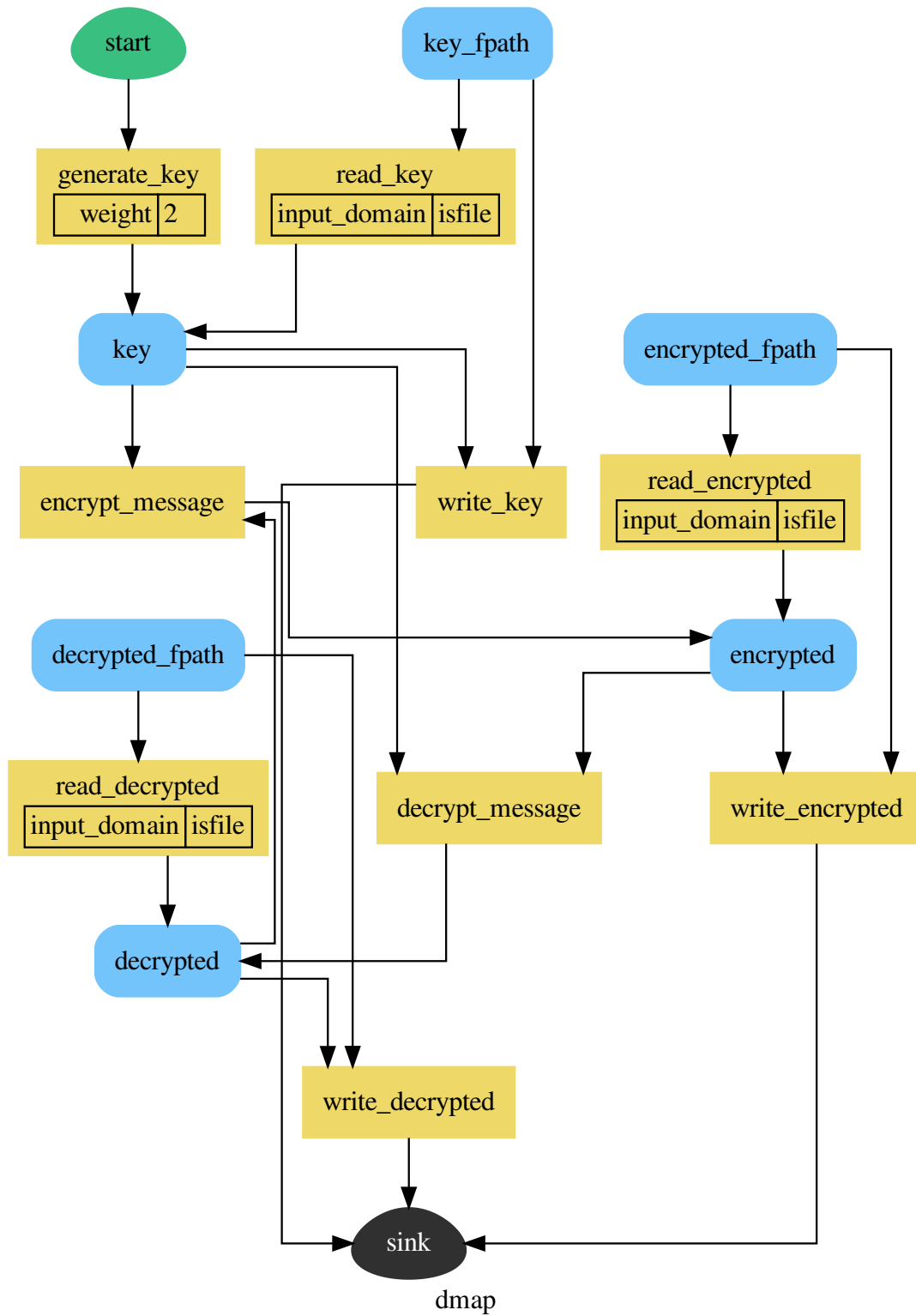
Finally, to read and write the encrypted or decrypted message, according to the functional programming philosophy, we can reuse the previously defined functions `read_key` and `write_key` changing the model mapping (i.e., *function_id*, *inputs*, and *outputs*). To add to the model, we can simply use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

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

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

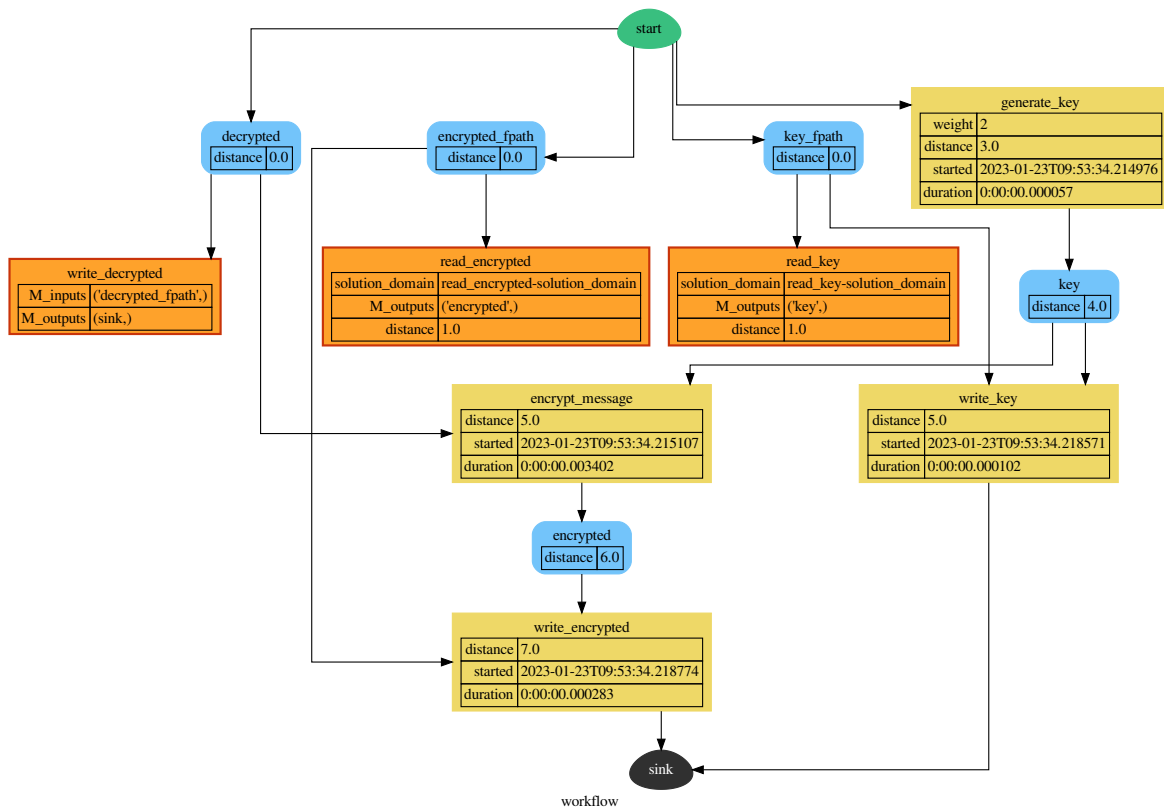


Tip: You can explore the diagram by clicking on it.

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)
```



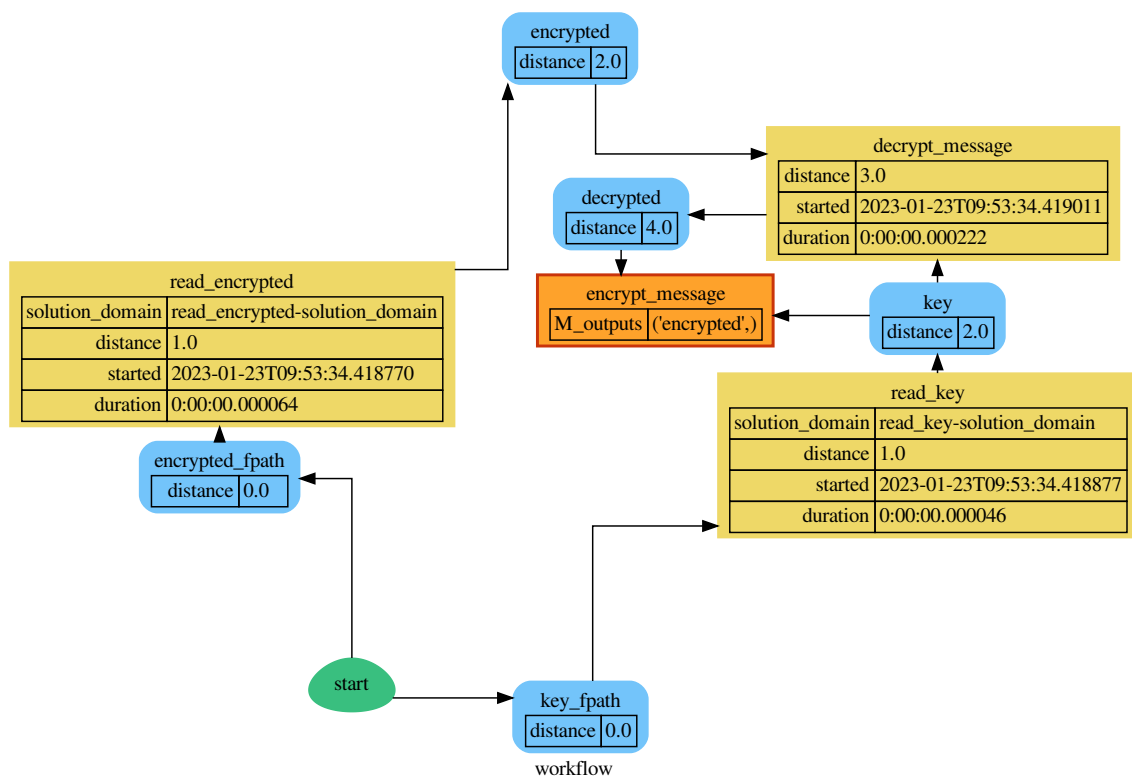
Note: As you can see from the workflow graph (orange nodes), when some function's inputs does not respect its domain, the Dispatcher automatically finds an alternative path to estimate all computable outputs. The same logic applies when there is a function failure.

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`:

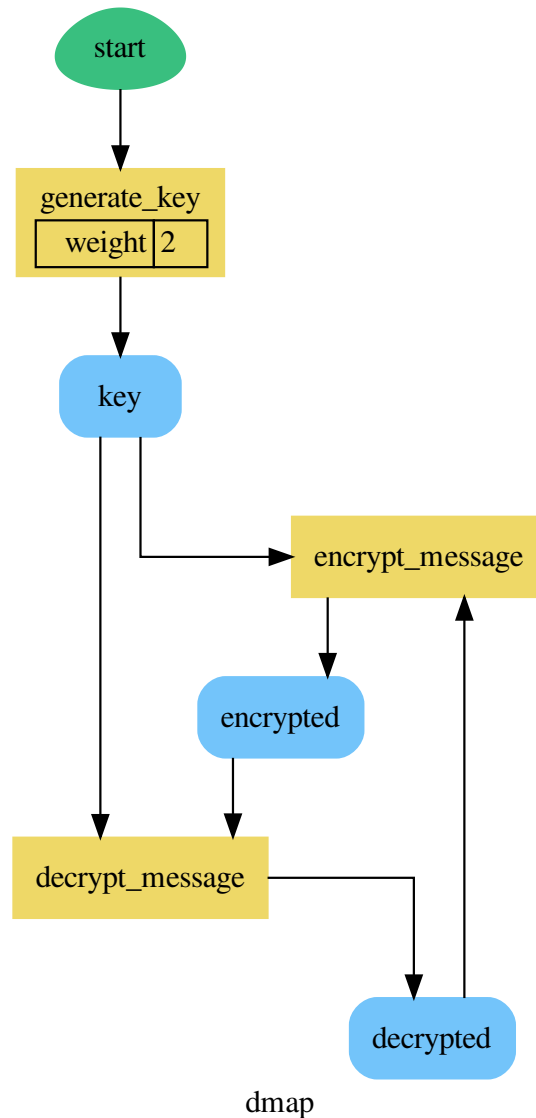
```
>>> dsp.plot(workflow=True, index=True)
```



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`, `MapDispatch`, `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(run=False).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

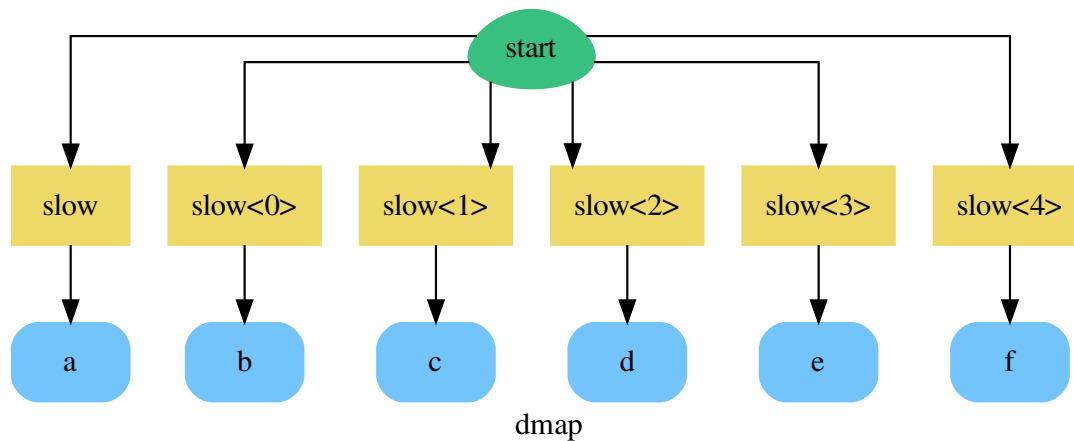
When there are heavy calculations which takes a significant amount of time, you want to run your model asynchronously or in parallel. Generally, this is difficult to achieve, because it requires an higher level of abstraction and a deeper knowledge of python programming and the Global Interpreter Lock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

DONATE

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

API REFERENCE

The core of the library is composed from the following modules:

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

Docstrings should provide sufficient understanding for any individual function.

Modules:

| | |
|-------------------|--|
| <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(*args, **kwargs)

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

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

A workflow is a sequence of function calls.

Example:

As an example, here is a system of equations:

$$b - a = c$$

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

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

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

Steps

Create an empty dispatcher:

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

Add data nodes to the dispatcher map:

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

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

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

Add a function node:

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

Add a function node with domain:

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

Add a data node with function estimation and callback function.

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

```
>>> def average_fun(kwarg):
...     """
...     Returns the average of node estimations.
...
...     :param kwarg:
...         Node estimations.
...     :type kwarg: dict
...
...     :return:
...         The average of node estimations.
...     :rtype: float
```

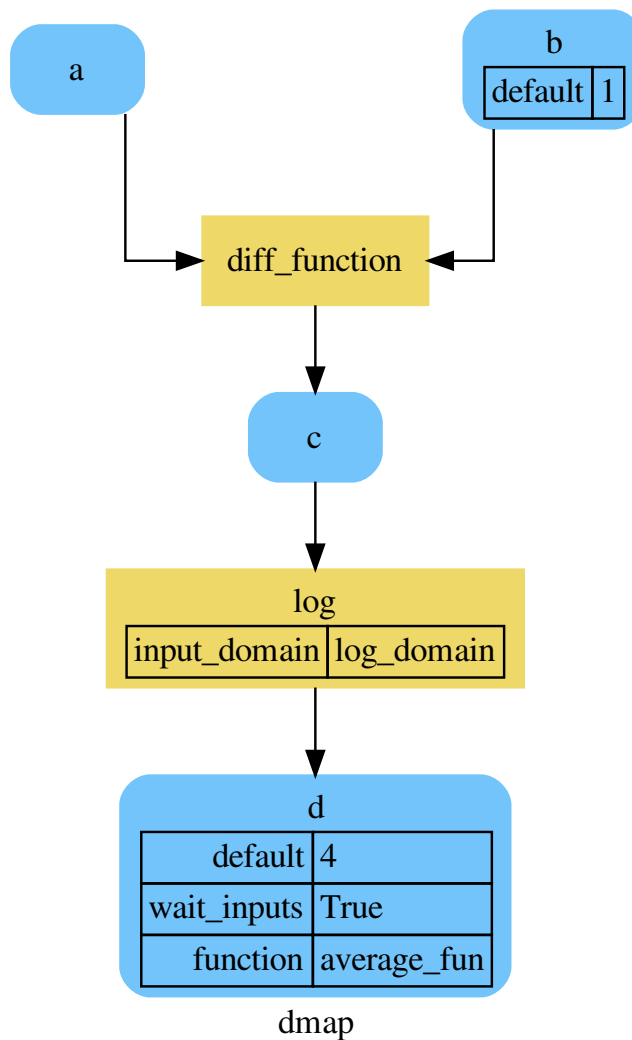
(continues on next page)

(continued from previous page)

```

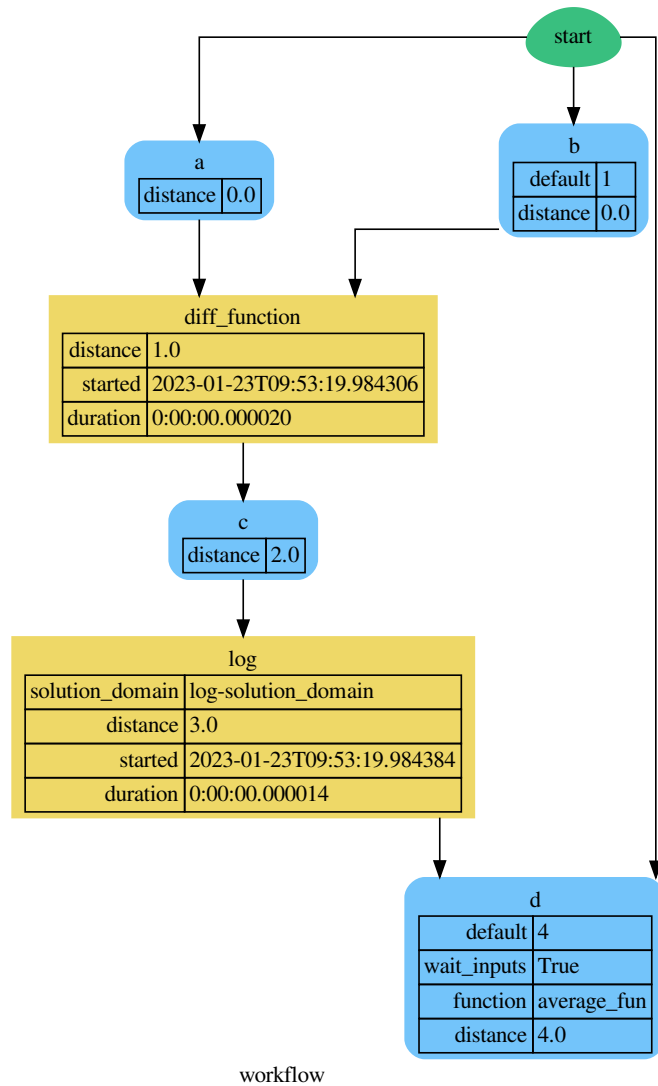
...     """
...     x = kwargs.values()
...     return sum(x) / len(x)
...
>>> def callback_fun(x):
...     print('(log(1) + 4) / 2 = %.1f' % x)
...
>>> dsp.add_data(data_id='d', default_value=4, wait_inputs=True,
...               function=average_fun, callback=callback_fun)
... 'd'

```



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

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



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. |
| <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>form</code> | Creates a dispatcher Form Flask app. |
| <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=False)`

Initializes the dispatcher.

Parameters

- **dmap** (`schedula.utils.graph.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=None, outputs=None, dsp_id=None, input_domain=None, weight=None, inp_weight=None, description=None, include_defaults=False, await_domain=None, inputs_prefix="", outputs_prefix="", **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. If `None` all child dispatcher nodes are used as inputs.
- **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. If `None` all child dispatcher nodes are used as outputs.
- **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.
- **inputs_prefix** (`str`) – Add a prefix to parent dispatcher inputs nodes.
- **outputs_prefix** (`str`) – Add a prefix to parent dispatcher outputs nodes.
- **kwargs** (`keyword arguments, optional`) – Set additional node attributes using `key=value`.

Returns

Sub-dispatcher node id.

Return type

`str`

See also:

`add_data()`, `add_func()`, `add_function()`, `add_from_lists()`

Example:

Create a sub-dispatcher:

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

Add the sub-dispatcher to the parent dispatcher:

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

Add a sub-dispatcher node with domain:

```
>>> def my_domain(kwargs):
...     return kwargs['C'] > 3
...
>>> dsp.add_dispatcher(dsp_id='Sub-Dispatcher with domain',
...                   dsp=sub_dsp, inputs={'C': 'a', 'D': 'b'},
...                   outputs={('c', 'b'): ('E', 'E1')},
...                   input_domain=my_domain)
'Sub-Dispatcher with domain'
```

add_from_lists

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

Add multiple function and data nodes to dispatcher.

Parameters

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

Returns

- Data node ids.
- Function node ids.

- Sub-dispatcher node ids.

Return type

(list[str], list[str], list[str])

See also:

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

Example:

Define a data list:

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

Define a functions list:

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

Define a sub-dispatchers list:

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

Add function and data nodes to dispatcher:

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

add_func

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

Add a single function node to dispatcher.

Parameters

- **inputs_kwargs** (*bool*) – Do you want to include kwargs as inputs?
- **inputs_defaults** (*bool*) – Do you want to set default values?
- **function_id** (*str*, *optional*) – Function node id. If None will be assigned as `<fun.__name__>`.
- **function** (*callable*, *optional*) – Data node estimation function.
- **inputs** (*list*, *optional*) – Ordered arguments (i.e., data node ids) needed by the function. If None it will take parameters names from function signature.
- **outputs** (*list*, *optional*) – Ordered results (i.e., data node ids) returned by the function.
- **input_domain** (*callable*, *optional*) – A function that checks if input values satisfy the function domain. This can be any function that takes the same inputs of the function and returns True if input values satisfy the domain, otherwise False. In this case the dispatch algorithm doesn't pass on the node.
- **weight** (*float*, *int*, *optional*) – Node weight. It is a weight coefficient that is used by the dispatch algorithm to estimate the minimum workflow.
- **inp_weight** (*dict[str, float | int]*, *optional*) – Edge weights from data nodes to the function node. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **out_weight** (*dict[str, float | int]*, *optional*) – Edge weights from the function node to data nodes. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **description** (*str*, *optional*) – Function node's description.
- **filters** (*list[function]*, *optional*) – A list of functions that are invoked after the invocation of the main function.
- **await_domain** (*bool | int | float*, *optional*) – If True the Dispatcher waits all input results before executing the `input_domain` function. If a number is defined this is used as `timeout` for `Future.result` method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool | int | float*, *optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as `timeout` for `Future.result` method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments*, *optional*) – Set additional node attributes using `key=value`.

Returns

Function node id.

Return type

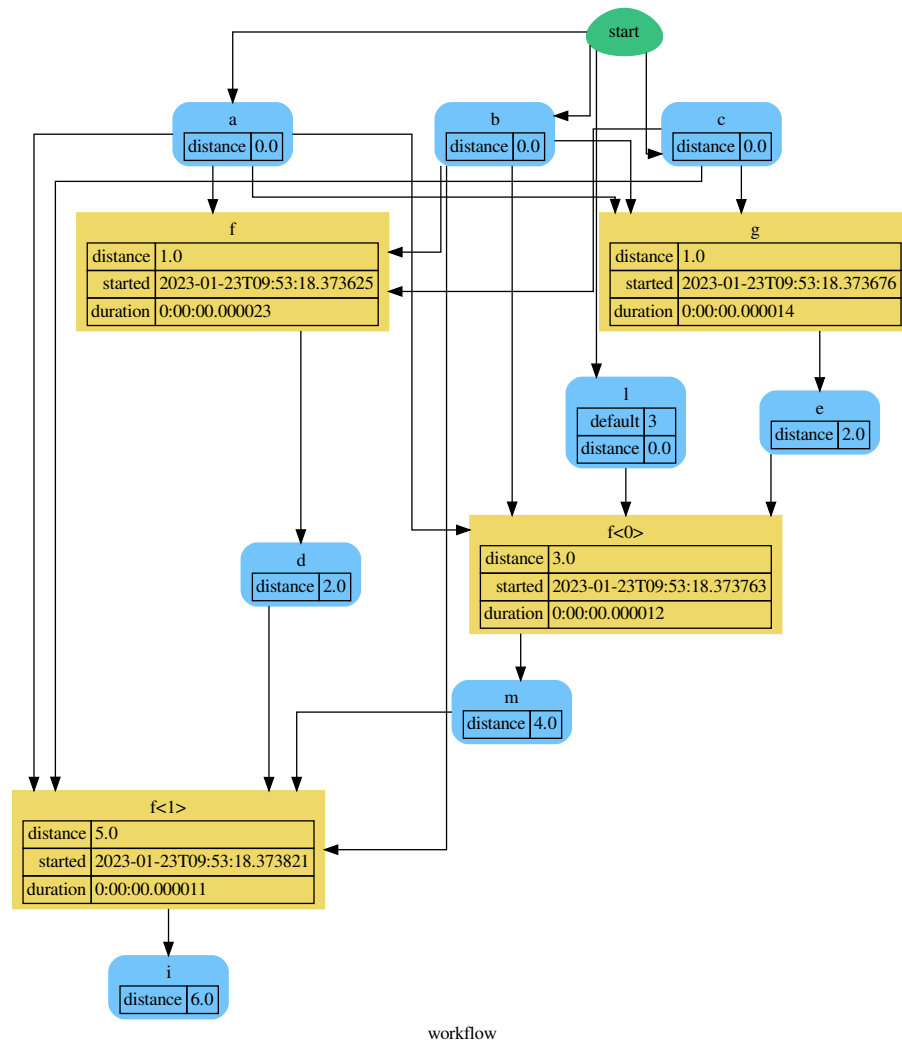
`str`

See also:

`add_func()`, `add_function()`, `add_dispatcher()`, `add_from_lists()`

Example:

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



add_function

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

Add a single function node to dispatcher.

Parameters

- **function_id** (*str*, *optional*) – Function node id. If None will be assigned as `<fun.__name__>`.
- **function** (*callable*, *optional*) – Data node estimation function.
- **inputs** (*list*, *optional*) – Ordered arguments (i.e., data node ids) needed by the function.

- **outputs** (*list*, *optional*) – Ordered results (i.e., data node ids) returned by the function.
- **input_domain** (*callable*, *optional*) – A function that checks if input values satisfy the function domain. This can be any function that takes the same inputs of the function and returns True if input values satisfy the domain, otherwise False. In this case the dispatch algorithm doesn't pass on the node.
- **weight** (*float*, *int*, *optional*) – Node weight. It is a weight coefficient that is used by the dispatch algorithm to estimate the minimum workflow.
- **inp_weight** (*dict[str, float | int]*, *optional*) – Edge weights from data nodes to the function node. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **out_weight** (*dict[str, float | int]*, *optional*) – Edge weights from the function node to data nodes. It is a dictionary (key=data node id) with the weight coefficients used by the dispatch algorithm to estimate the minimum workflow.
- **description** (*str*, *optional*) – Function node's description.
- **filters** (*list[function]*, *optional*) – A list of functions that are invoked after the invocation of the main function.
- **await_domain** (*bool | int | float*, *optional*) – If True the Dispatcher waits all input results before executing the *input_domain* function. If a number is defined this is used as *timeout* for *Future.result* method [default: True]. Note this is used when asynchronous or parallel execution is enable.
- **await_result** (*bool | int | float*, *optional*) – If True the Dispatcher waits output results before assigning them to the workflow. If a number is defined this is used as *timeout* for *Future.result* method [default: False]. Note this is used when asynchronous or parallel execution is enable.
- **kwargs** (*keyword arguments*, *optional*) – Set additional node attributes using key=value.

Returns

Function node id.

Return type

`str`

See also:

[`add_data\(\)`](#), [`add_func\(\)`](#), [`add_dispatcher\(\)`](#), [`add_from_lists\(\)`](#)

Example:

Add a function node:

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

Add a function node with domain:

```
>>> from math import log
>>> def my_log(a, b):
...     return log(b - a)
...
>>> 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, depth=-1)`

Constructs a BlueDispatcher out of the current object.

Parameters

- **memo** (*dict*[*T*, `schedula.utils.blue.Blueprint`]) – A dictionary to cache Blueprints.
- **depth** (*int*, *optional*) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

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, 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=(), verbose=False)`

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.
- **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.
- **verbose** (*str*, *optional*) – If True the dispatcher will log start and end of each function.

Returns

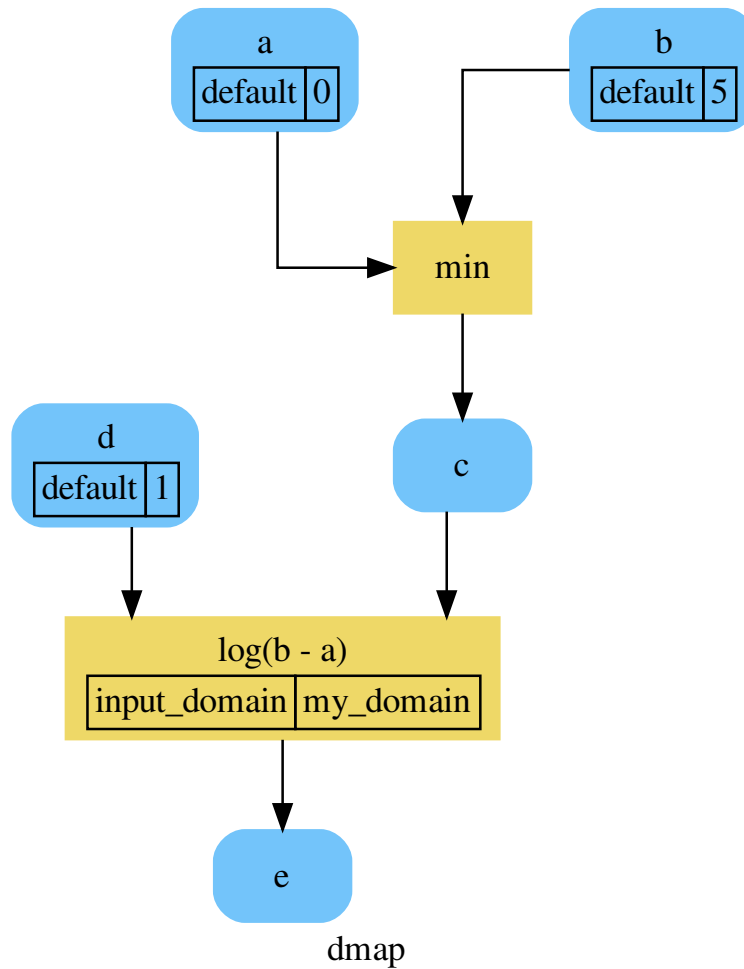
Dictionary of estimated data node outputs.

Return type

schedula.utils.sol.Solution

Example:

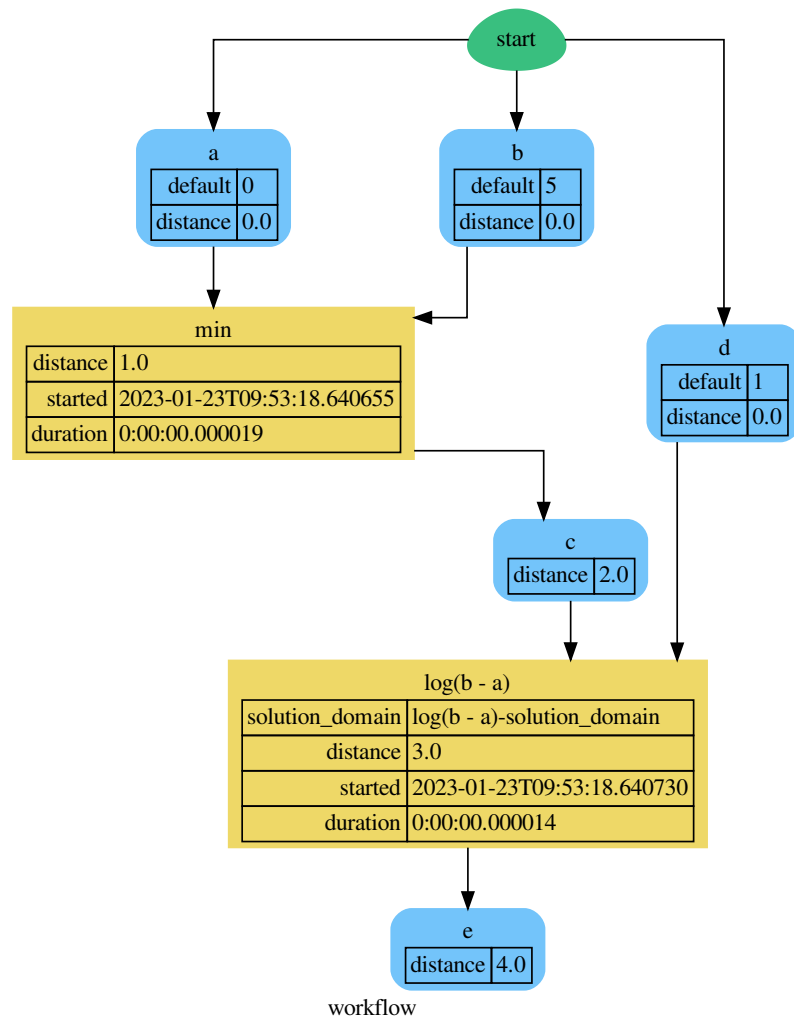
A dispatcher with a function $\log(b - a)$ and two data a and b with default values:



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))])
  
```

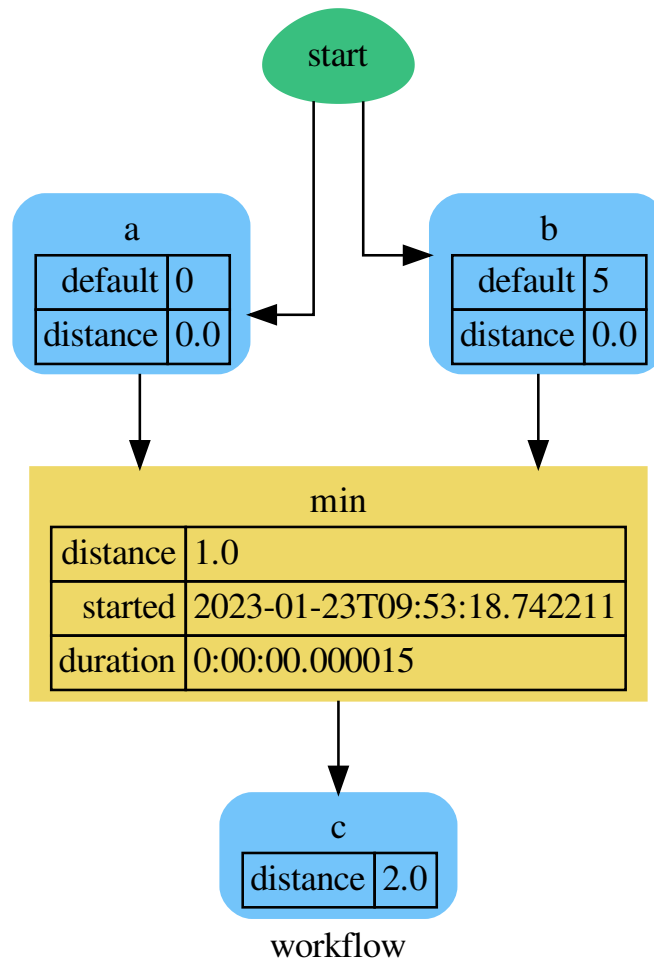


Dispatch until data node *c* is estimated:

```

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

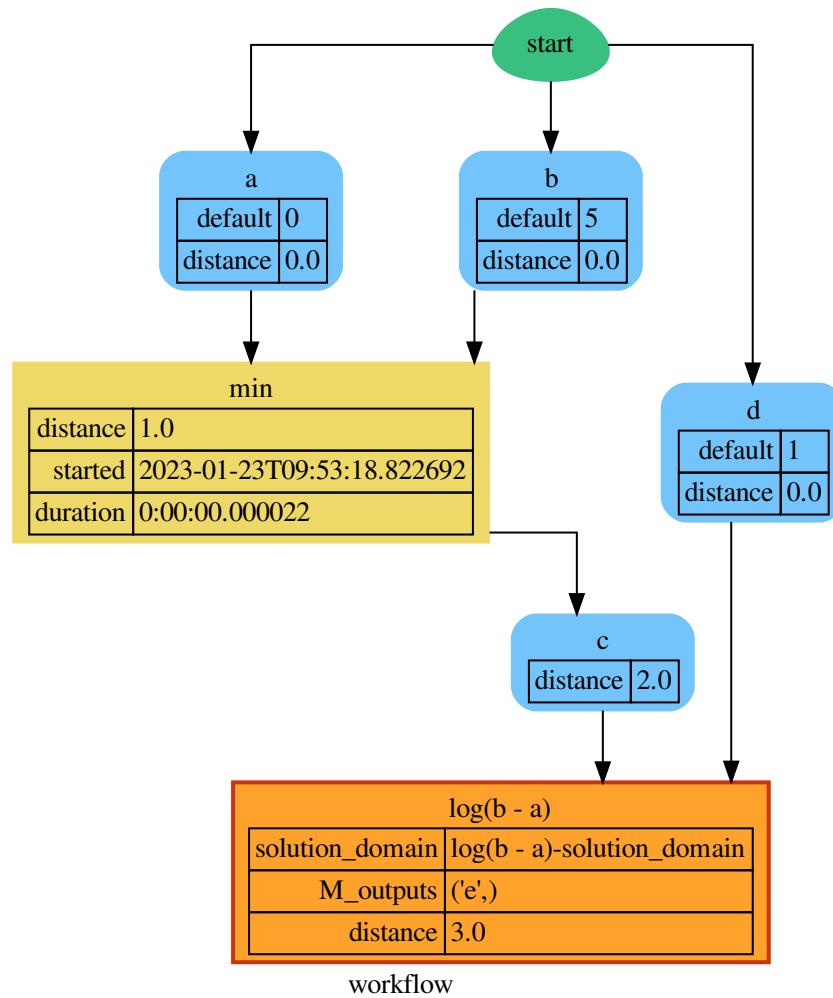
```



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

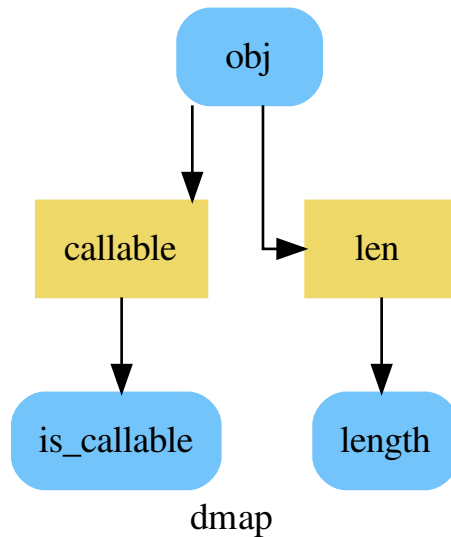
Self.

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)
```



form

`Dispatcher.form(depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600)`

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int*, *optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str]*, *optional*) – Data node attributes to produce API.
- **node_function** (*tuple[str]*, *optional*) – Function node attributes produce API.
- **directory** (*str*, *optional*) – Where is the generated Flask app root located?
- **sites** (*set[Site]*, *optional*) – A set of *Site* to maintain alive the backend server.

- **run**(*bool*, *optional*) – Run the backend server?
- **view**(*bool*, *optional*) – Open the url site with the sys default opener.
- **get_context**(*function* | *dict*, *optional*) – Function to pass extra data as form context.
- **get_data**(*function* | *dict*, *optional*) – Function to initialize the formdata.
- **edit_on_change**(*function* | *dict*, *optional*) – Function to initialize the formdata.
- **pre_submit**(*function* | *dict*, *optional*) – Function to initialize the formdata.
- **post_submit**(*function* | *dict*, *optional*) – Function to initialize the formdata.
- **subsite_idle_timeout**(*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run* or *view* is *True*.

Return type

[FormMap](#) | [Site](#)

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.

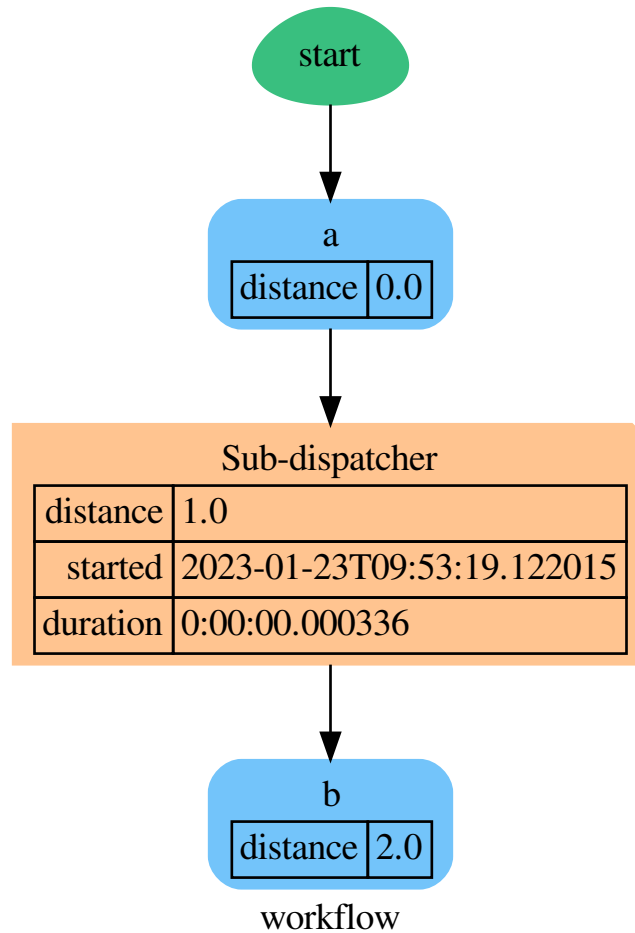
Returns

Node attributes and its real path.

Return type

(*T*, (*str*, ...))

Example:



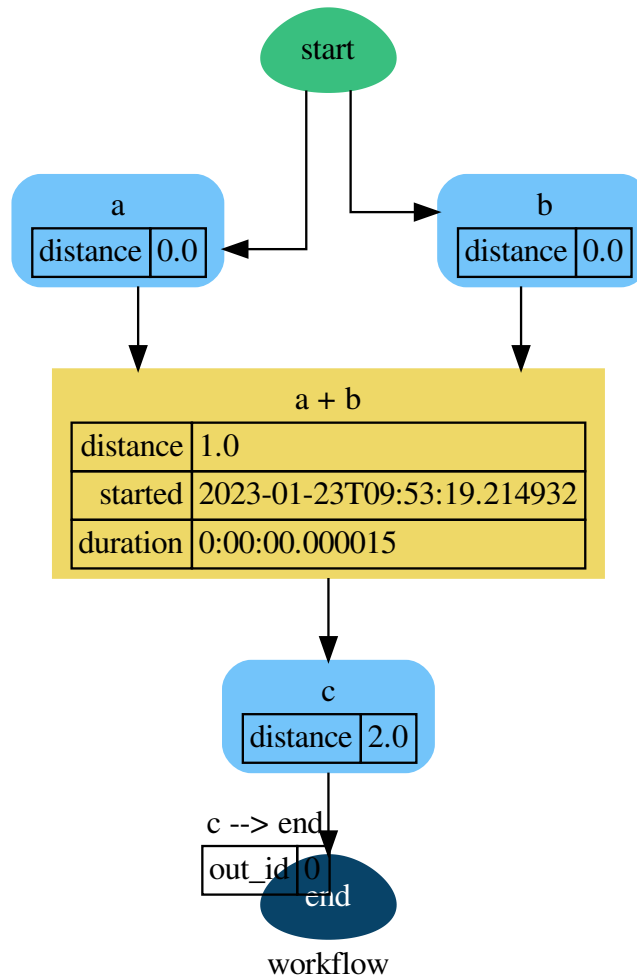
Get the sub node output:

```

>>> 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

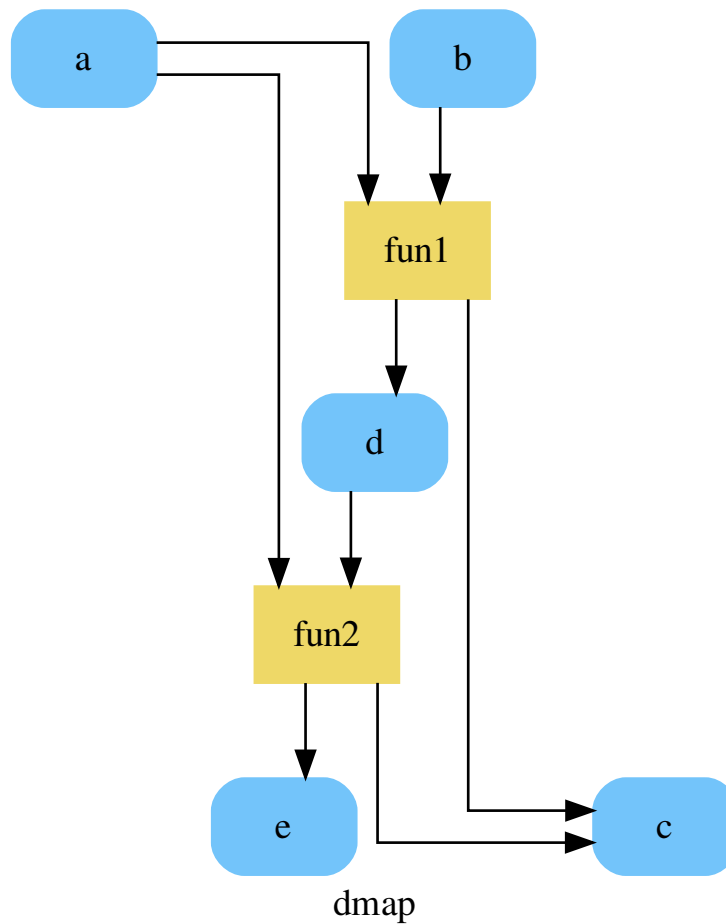
See also:

[*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.

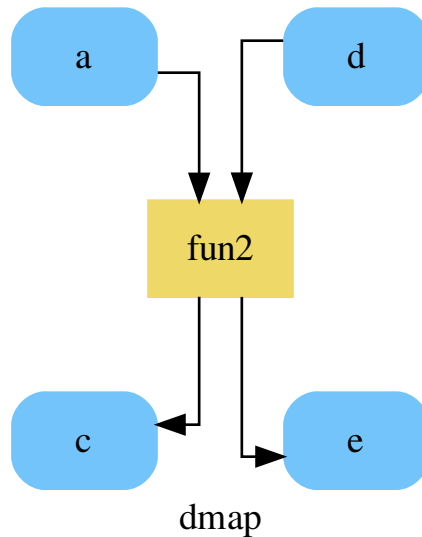
Example:

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** (*schedula.utils.graph.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

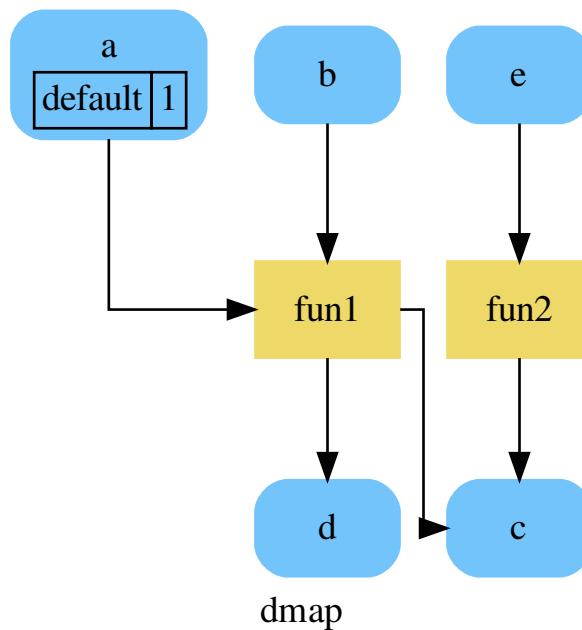
See also:

[*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:

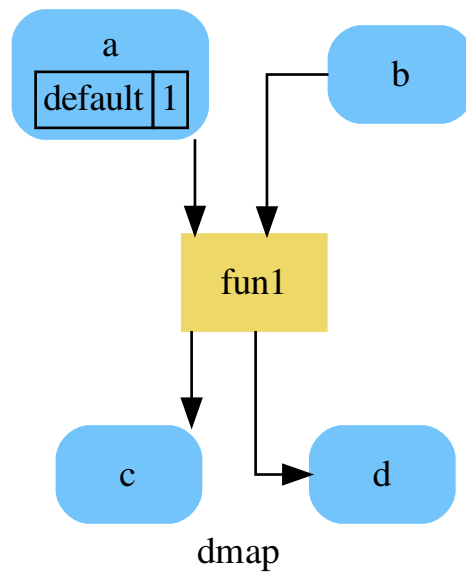


Dispatch with no calls in order to have a workflow:

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

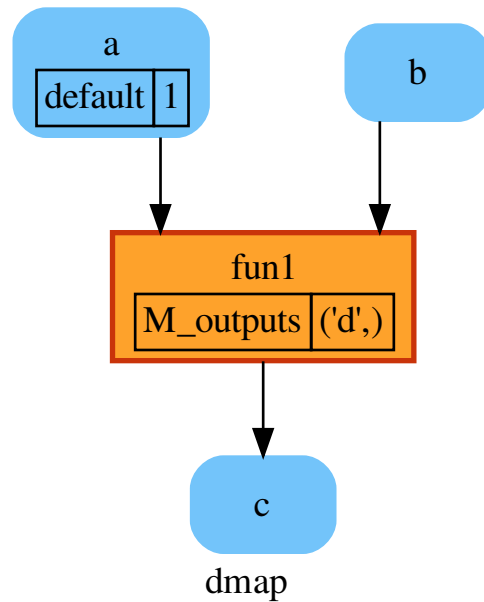
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> 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, raw_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=True, viz=False, short_name=None, executor='async', render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[Site]*, *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

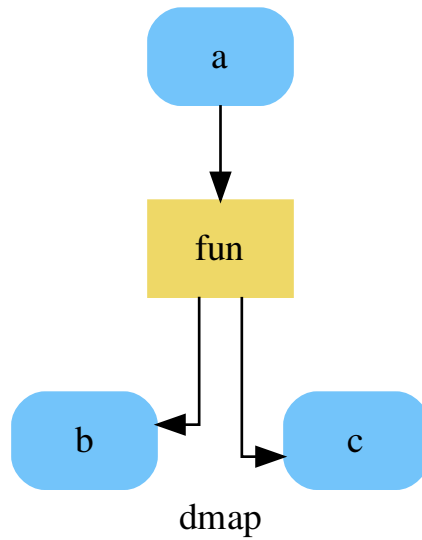
A SiteMap or a Site if .

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*:

```

>>> import schedula as sh
>>> dsp = sh.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=sh.EMPTY)
>>> dsp.default_values
{}
```

shrink_dsp

Dispatcher.**shrink_dsp**(*inputs=None, outputs=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.
- **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

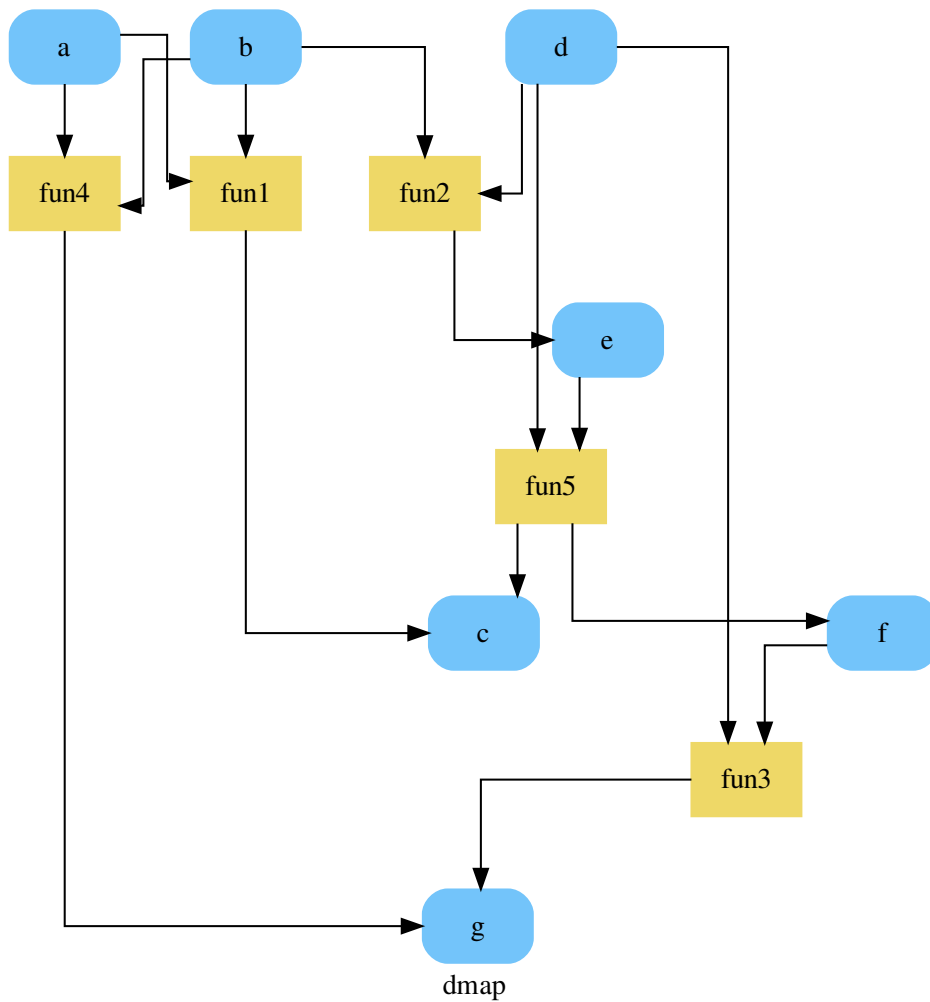
Dispatcher

See also:

[*dispatch\(\)*](#)

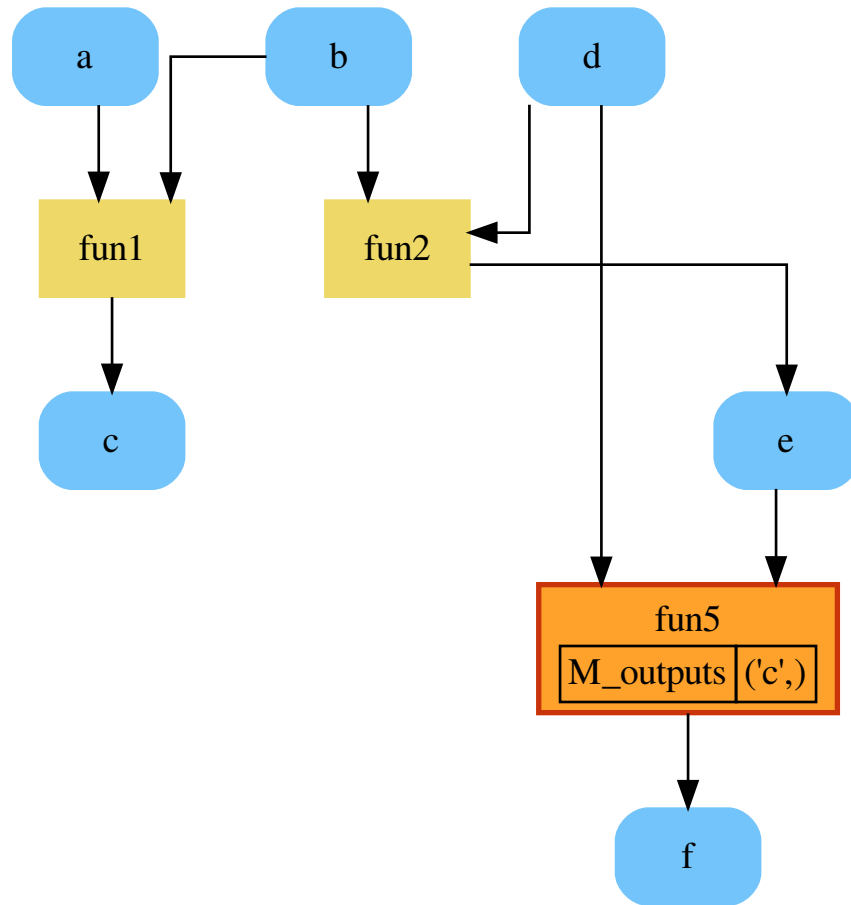
Example:

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, subsite_idle_timeout=600)`

Creates a dispatcher Flask app.

Parameters

- **depth** (*int*, *optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str]*, *optional*) – Data node attributes to produce API.
- **node_function** (*tuple[str]*, *optional*) – Function node attributes produce API.
- **directory** (*str*, *optional*) – Where is the generated Flask app root located?

- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool*, *optional*) – Run the backend server?
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

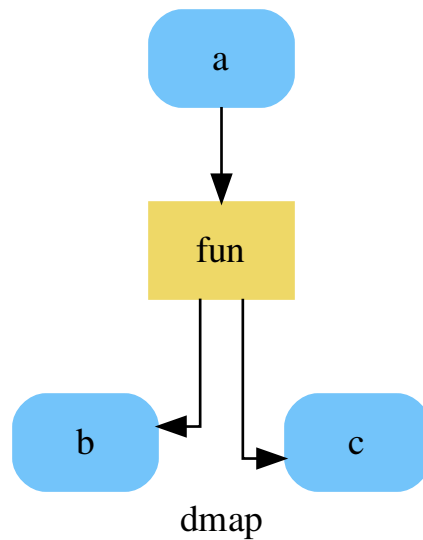
Return type

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:

```
>>> print("Starting...\n"); site = dsp.web(); 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__(*dmap=None, name="", default_values=None, raises=False, description="", executor=False*)

Initializes the dispatcher.

Parameters

- **dmap** (*schedula.utils.graph.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*.

dmap

The directed graph that stores data & functions parameters.

name

The dispatcher’s name.

nodes

The function and data nodes of the dispatcher.

default_values

Data node default values. These will be used as input if it is not specified as inputs in the ArciDispatch algorithm.

raises

If True the dispatcher interrupt the dispatch when an error occur.

executor

Pool executor to dispatch asynchronously.

solution

Last dispatch solution.

counter

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:
...         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

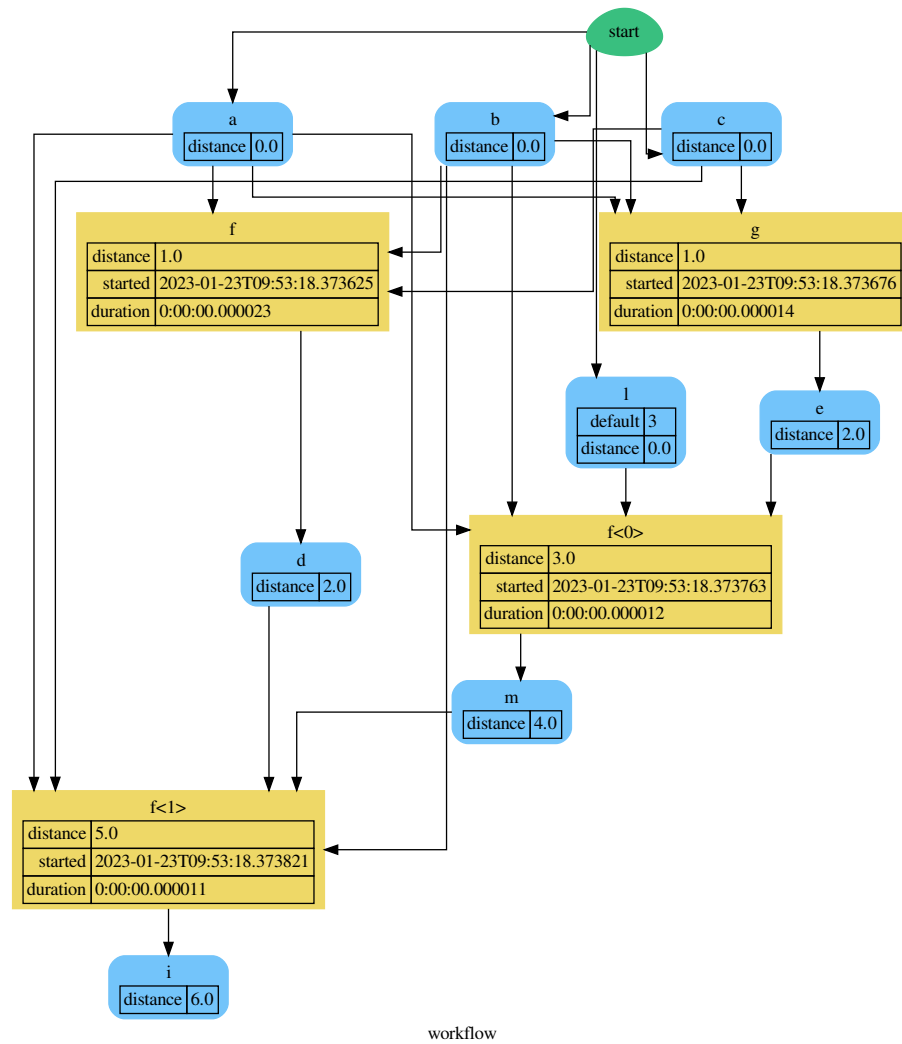
str

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'
>>> 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=None, outputs=None, dsp_id=None, input_domain=None, weight=None, inp_weight=None, description=None, include_defaults=False, await_domain=None, inputs_prefix="", outputs_prefix="", **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. If `None` all child dispatcher nodes are used as inputs.
- **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. If `None` all child dispatcher nodes are used as outputs.

- **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.
- **inputs_prefix**(*str*) – Add a prefix to parent dispatcher inputs nodes.
- **outputs_prefix**(*str*) – Add a prefix to parent dispatcher outputs nodes.
- **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'},
```

(continues on next page)

(continued from previous page)

```
...           outputs={'c': 'C'})
'Sub-Dispatcher'
```

Add a sub-dispatcher node with domain:

```
>>> def my_domain(kwarg):
...     return kwarg['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*:

```
>>> import schedula as sh
>>> dsp = sh.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=sh.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

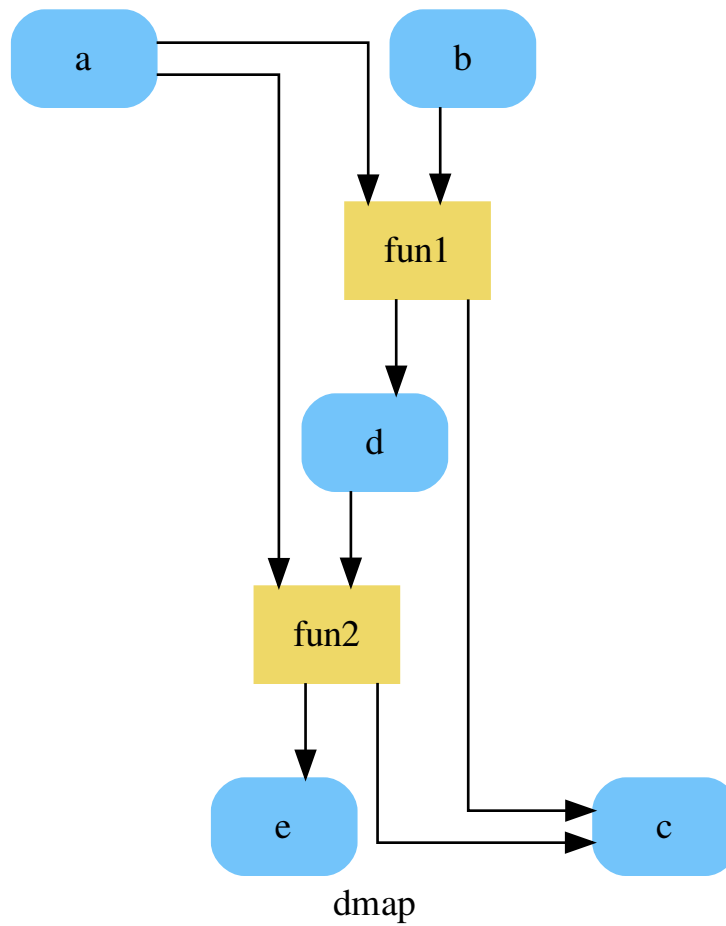
See also:

[*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.

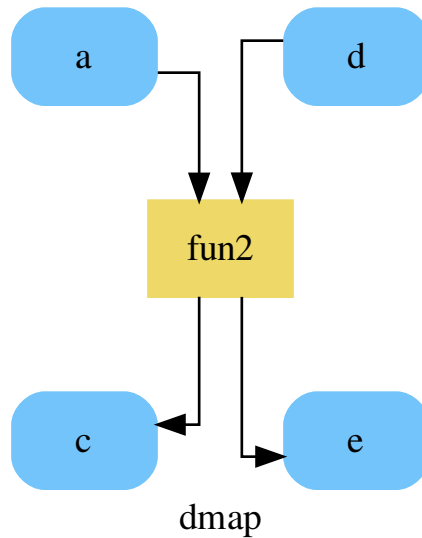
Example:

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** (*schedula.utils.graph.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

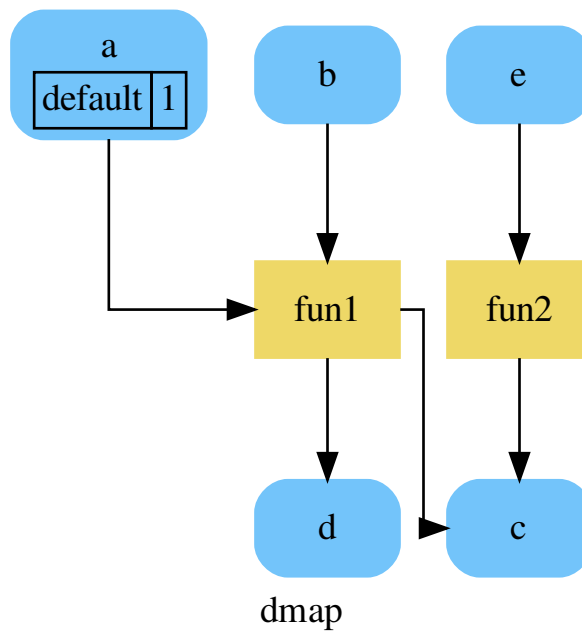
See also:

[*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:

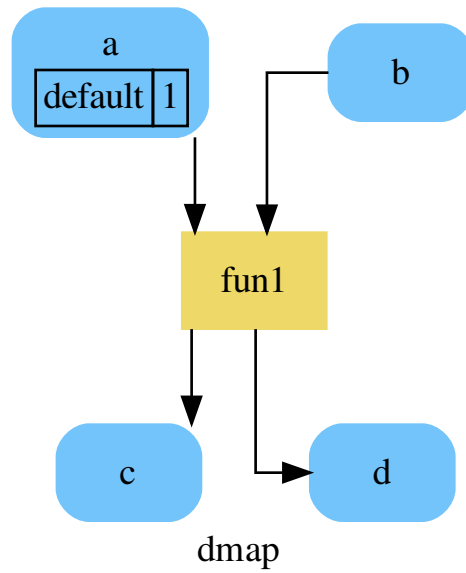


Dispatch with no calls in order to have a workflow:

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

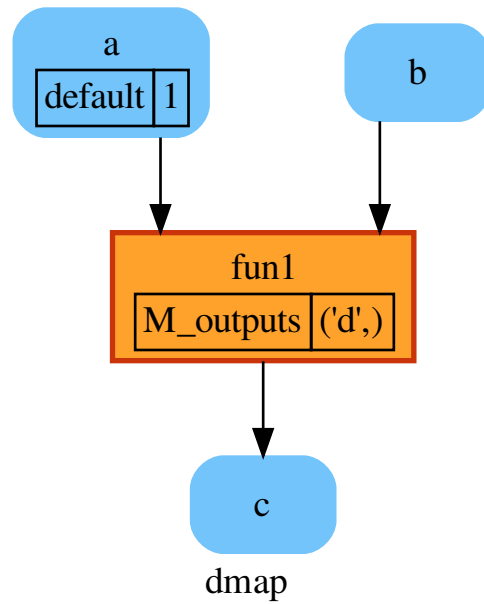
Get sub-dispatcher from workflow inputs *a* and *b*:

```
>>> 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)
```



property data_nodes

Returns all data nodes of the dispatcher.

Returns

All data nodes of the dispatcher.

Return type

`dict[str, dict]`

property function_nodes

Returns all function nodes of the dispatcher.

Returns

All data function of the dispatcher.

Return type

`dict[str, dict]`

property 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, depth=-1*)

Constructs a BlueDispatcher out of the current object.

Parameters

- **memo** (*dict*[*T*, *schedula.utils.blue.Blueprint*]) – A dictionary to cache Blueprints.
- **depth** (*int*, *optional*) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

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

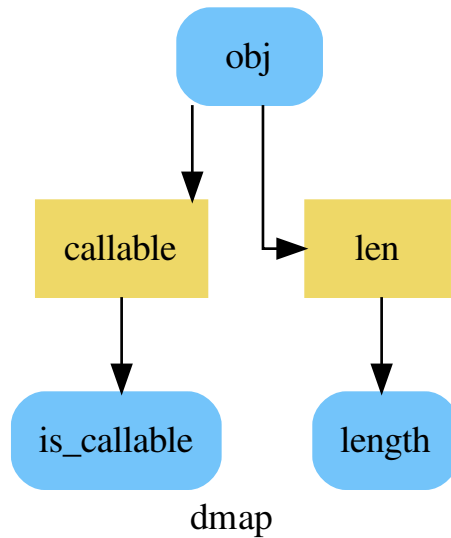
Self.

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, 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=(), verbose=False*)

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.
- **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.
- **verbose** (*str*, *optional*) – If True the dispatcher will log start and end of each function.

Returns

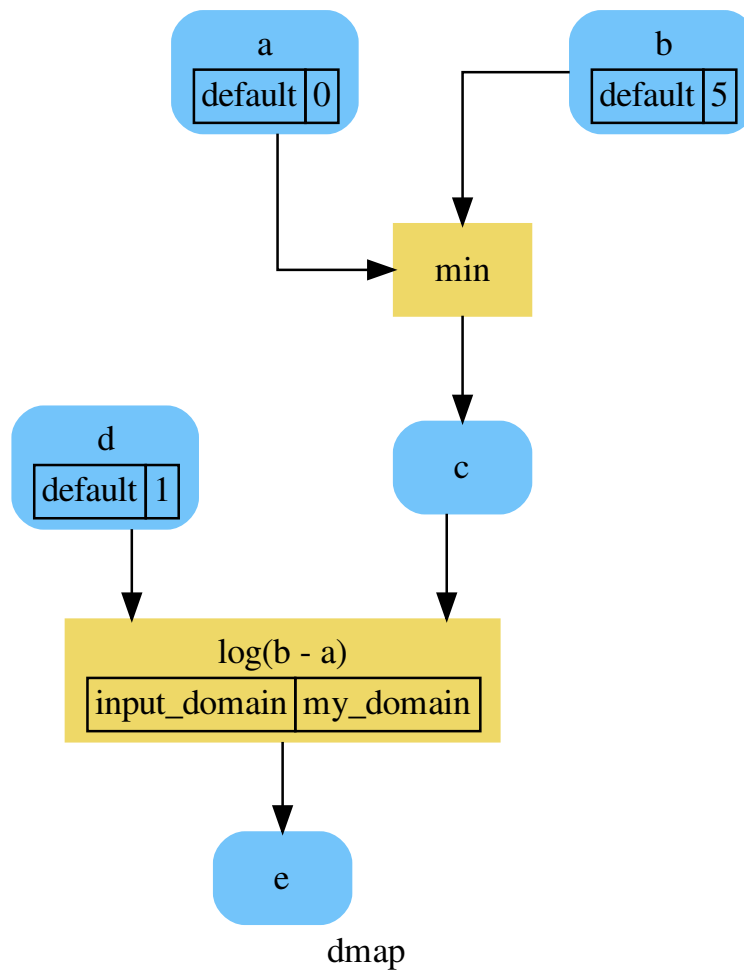
Dictionary of estimated data node outputs.

Return type

schedula.utils.sol.Solution

Example:

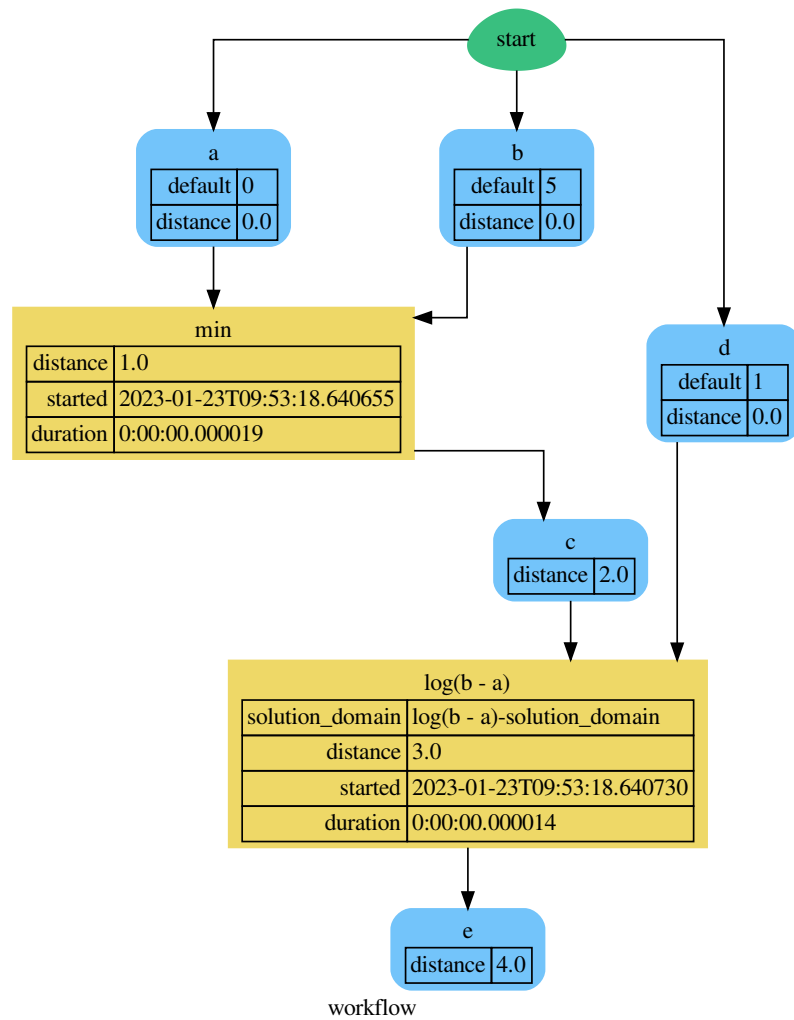
A dispatcher with a function $\log(b - a)$ and two data a and b with default values:



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)])
  
```

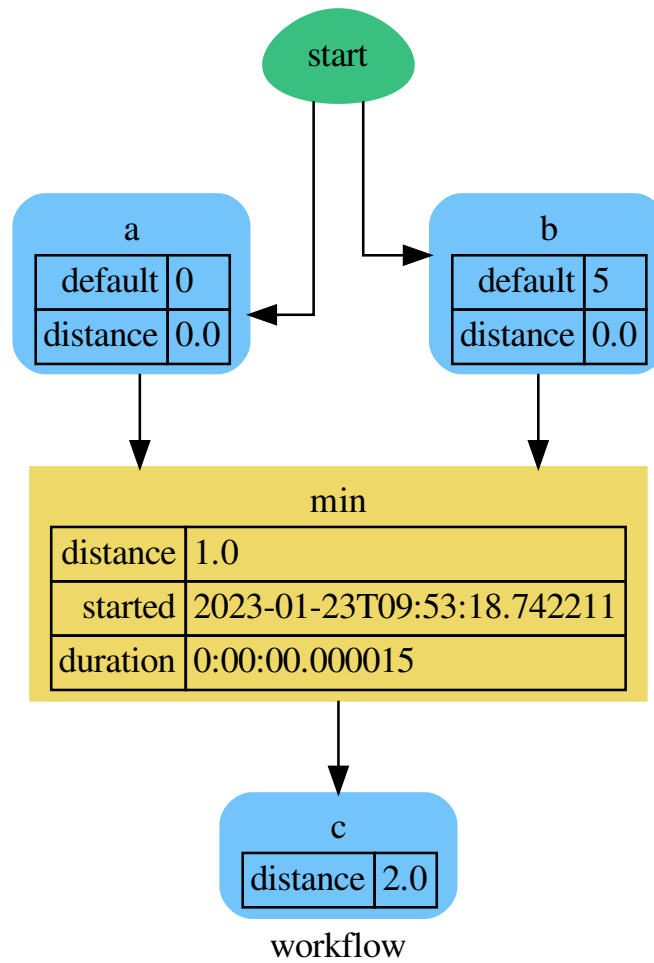


Dispatch until data node *c* is estimated:

```

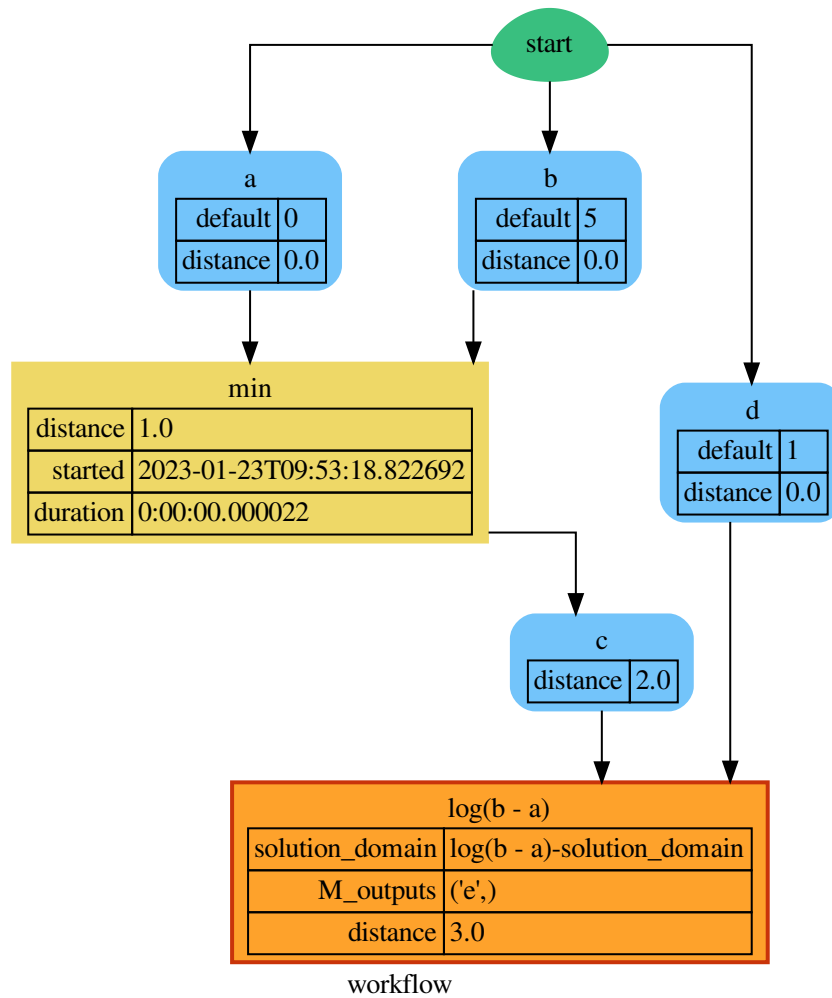
>>> outputs = dsp.dispatch(outputs=['c'])
>>> outputs
Solution([(('a', 0), ('b', 5), ('c', 0))])

```

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, 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.
- **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

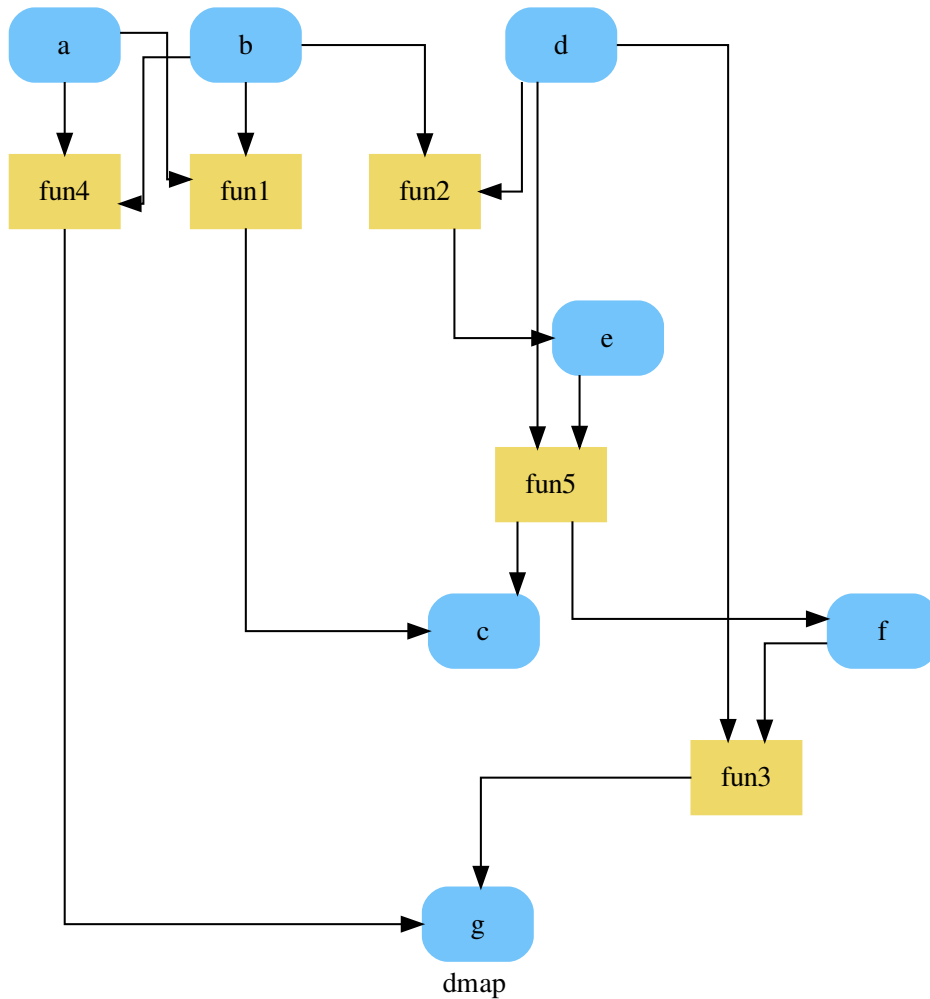
Dispatcher

See also:

`dispatch()`

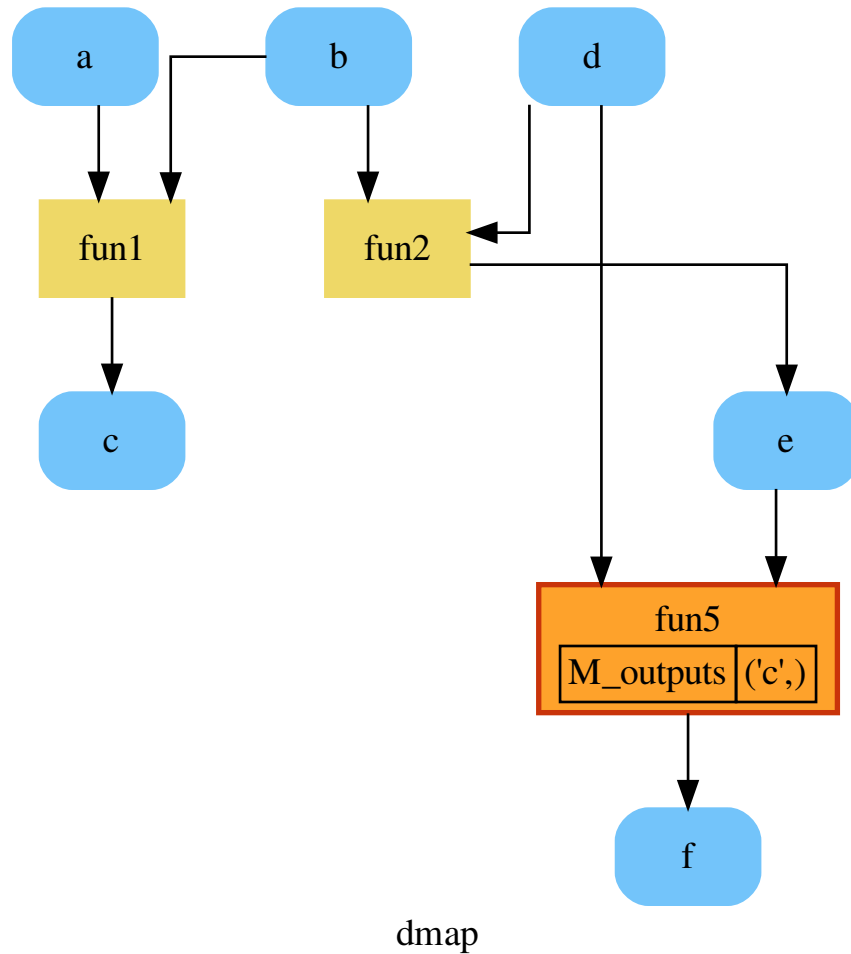
Example:

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>form</i> | It provides functions to build a flask app from a dispatcher. |
| <i>gen</i> | It contains classes and functions of general utility. |
| <i>graph</i> | It contains the <i>DiGraph</i> class. |
| <i>imp</i> | Fixes ImportError for MicroPython. |
| <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>utl</i> | |
| <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

| | |
|-----------------------------|--|
| <code>add_func_edges</code> | Adds function node edges. |
| <code>get_full_pipe</code> | Returns the full pipe of a dispatch run. |
| <code>get_sub_node</code> | Returns a sub node of a dispatcher. |

`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

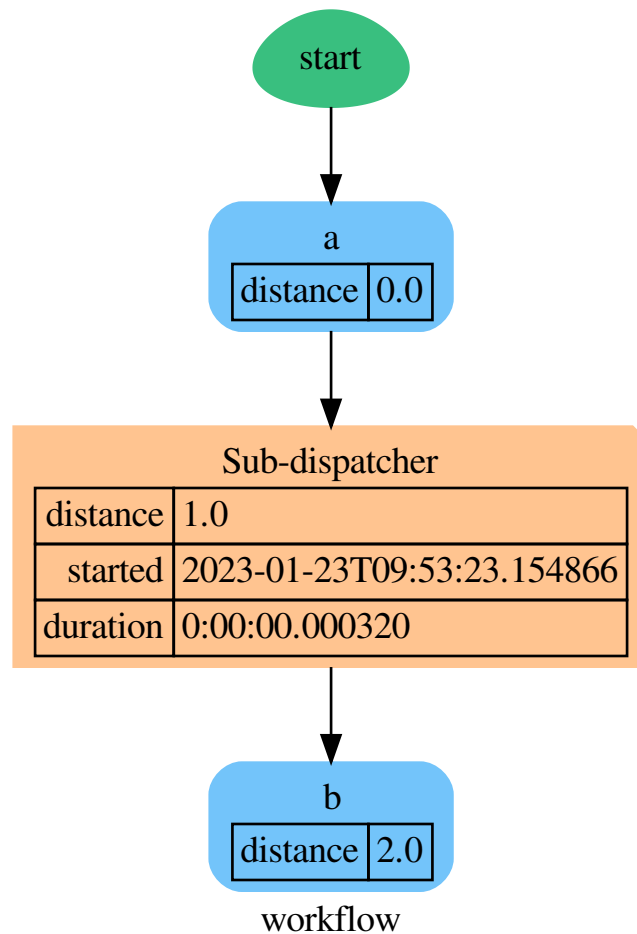
dict | *object*, *tuple*[*str*]

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}})
...

```

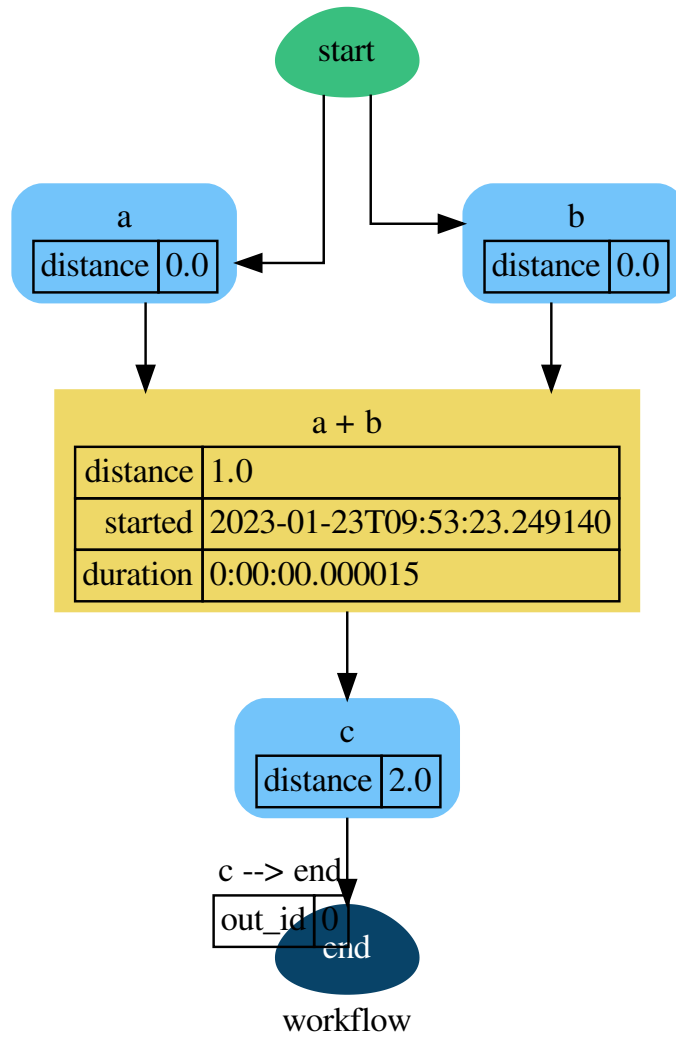


Get the sub node 'c' output or type:

```
>>> 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',))
```

Classes

DspPipe

DspPipe

class DspPipe

Methods

| | |
|--------------------------|--|
| <code>__init__</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>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>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> | |

`__init__`

DspPipe.`__init__`(*args, **kwargs)

`clear`

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

copy

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

fromkeys

`DspPipe.fromkeys(value=None)`

Create a new ordered dictionary with keys from iterable and values set to value.

get

`DspPipe.get(key, default=None, /)`

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(key, last=True)`

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(last=True)`

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(key, default=None)`

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__(*args, **kwargs)`

7.2.2 asy

It contains functions to dispatch asynchronously and in parallel.

Sub-Modules:

| | |
|------------------------|--|
| <code>executors</code> | It defines the executors classes. |
| <code>factory</code> | It defines the <i>ExecutorFactory</i> class. |

executors

It defines the executors classes.

Classes

| | |
|----------------------------------|--|
| <code>Executor</code> | Base Executor |
| <code>PoolExecutor</code> | General PoolExecutor to dispatch asynchronously and in parallel. |
| <code>ProcessExecutor</code> | Process Executor |
| <code>ProcessPoolExecutor</code> | Process Pool Executor |
| <code>ThreadExecutor</code> | Multi Thread Executor |

Executor

class Executor

Base Executor

Methods

`__init__`

`shutdown`

`submit`

`__init__`

Executor.`__init__`()

`shutdown`

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

`submit`

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

`__init__`()

PoolExecutor

class PoolExecutor(*thread_executor*, *process_executor=None*, *parallel=None*)

General PoolExecutor to dispatch asynchronously and in parallel.

Methods

| | |
|----------------------------|---|
| <code>__init__</code> | param <code>thread_executor</code> |
| <code>add_future</code> | |
| <code>get_futures</code> | |
| <code>process</code> | |
| <code>process_funcs</code> | |
| <code>shutdown</code> | |
| <code>thread</code> | |
| <code>wait</code> | |

`__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.

`add_future`

`PoolExecutor.add_future(sol_id, fut)`

`get_futures`

`PoolExecutor.get_futures(sol_id=empty)`

process

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

process_funcs

`PoolExecutor.process_funcs(exe_id, funcs, *args, **kw)`

shutdown

`PoolExecutor.shutdown(wait=True)`

thread

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

wait

`PoolExecutor.wait(timeout=None)`

`__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(*args, **state)`

Process Executor

Methods

`__init__`

`init`

`shutdown`

`submit`

`__init__`

`ProcessExecutor.__init__(*args, **state)`

`init`

`ProcessExecutor.init()`

`shutdown`

`ProcessExecutor.shutdown(wait=True)`

`submit`

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

`__init__(*args, **state)`

ProcessPoolExecutor

class `ProcessPoolExecutor(*args, **state)`

Process Pool Executor

Methods

`__init__`

`init`

`shutdown`

`submit`

`__init__`

`ProcessPoolExecutor.__init__(*args, **state)`

init

`ProcessPoolExecutor.init()`

shutdown

`ProcessPoolExecutor.shutdown(wait=True)`

submit

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

`__init__(*args, **state)`

ThreadExecutor

class ThreadExecutor

Multi Thread Executor

Methods

`__init__`

`shutdown`

`submit`

`__init__`

`ThreadExecutor.__init__()`

shutdown

`ThreadExecutor.shutdown(wait=True)`

submit

`ThreadPoolExecutor.submit(func, *args, **kwargs)`

`__init__()`

factory

It defines the *ExecutorFactory* class.

Classes

ExecutorFactory

ExecutorFactory

class `ExecutorFactory(*args, **kwargs)`

Methods

| | |
|--------------------------------|--|
| <code>__init__</code> | |
| <code>clear</code> | |
| <code>copy</code> | |
| <code>executor_id</code> | |
| <code>fromkeys</code> | Create a new 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_executor</code> | |
| <code>items</code> | |
| <code>keys</code> | |
| <code>pop</code> | If key is not found, d is returned if given, otherwise KeyError is raised |
| <code>pop_active</code> | |
| <code>popitem</code> | 2-tuple; but raise KeyError if D is empty. |
| <code>set_active</code> | |
| <code>set_executor</code> | |
| <code>setdefault</code> | Insert key with a value of default if key is not in the dictionary. |
| <code>shutdown_executor</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__`

`ExecutorFactory.__init__(*args, **kwargs)`

`clear`

`ExecutorFactory.clear()` → None. Remove all items from D.

`copy`

`ExecutorFactory.copy()` → a shallow copy of D

`executor_id`

static `ExecutorFactory.executor_id(name, sol)`

`fromkeys`

`ExecutorFactory.fromkeys(value=None, /)`

Create a new dictionary with keys from iterable and values set to value.

`get`

`ExecutorFactory.get(key, default=None, /)`

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

`get_executor`

`ExecutorFactory.get_executor(exe_id)`

`items`

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

`keys`

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

pop

`ExecutorFactory.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

pop_active

`ExecutorFactory.pop_active(sol_id)`

popitem

`ExecutorFactory.popitem()` → (*k*, *v*), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if *D* is empty.

set_active

`ExecutorFactory.set_active(sol_id, value=True)`

set_executor

`ExecutorFactory.set_executor(name, value)`

setdefault

`ExecutorFactory.setdefault(key, default=None, /)`

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.

shutdown_executor

`ExecutorFactory.shutdown_executor(name=empty, sol_id=empty, wait=True)`

update

`ExecutorFactory.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

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

__init__(*args, **kwargs)

Functions

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

atexit_register

atexit_register(*args, **kwargs)

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, executors=None)

Register a new executor type.

Parameters

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

shutdown_executor

shutdown_executor(*name=empty, sol_id=empty, wait=True, executors=None*)

Clean-up the resources associated with the Executor.

Parameters

- **name** (*str*) – Executor name.
- **sol_id** (*int*) – Solution id.
- **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.
- **executors** (*ExecutorFactory*) – Executor factory.

Returns

Shutdown pool executor.

Return type

`dict[concurrent.futures.Future, Thread|Process]`

shutdown_executors

shutdown_executors(*wait=True, executors=None*)

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.
- **executors** (*ExecutorFactory*) – Executor factory.

Returns

Shutdown pool executor.

Return type

`dict[str, dict]`

Classes

| | |
|------------------|-------------------------------|
| <i>AsyncList</i> | List of asynchronous results. |
|------------------|-------------------------------|

AsyncList

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

List of asynchronous results.

Methods

| | |
|-----------------------|--|
| <code>__init__</code> | |
| <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)`

`append`

`AsyncList.append(object, /)`

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(value, /)`

Return number of occurrences of value.

extend

`AsyncList.extend(iterable, /)`

Extend list by appending elements from the iterable.

index

`AsyncList.index(value, start=0, stop=9223372036854775807, /)`

Return first index of value.

Raises `ValueError` if the value is not present.

insert

`AsyncList.insert(index, object, /)`

Insert object before index.

pop

`AsyncList.pop(index=-1, /)`

Remove and return item at index (default last).

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

remove

`AsyncList.remove(value, /)`

Remove first occurrence of value.

Raises `ValueError` if the value is not present.

reverse

`AsyncList.reverse()`

Reverse *IN PLACE*.

sort

`AsyncList.sort(*, key=None, reverse=False)`

Stable sort *IN PLACE*.

`__init__(*, future=None, n=1)`

7.2.3 base

It provides a base class for dispatcher objects.

Classes

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

Base

class `Base(*args, **kwargs)`

Base class for dispatcher objects.

Methods

| | |
|-----------------------|--|
| <code>__init__</code> | |
| <code>form</code> | Creates a dispatcher Form Flask app. |
| <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__`

`Base.__init__()`

`form`

`Base.form(depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600)`

Creates a dispatcher Form Flask app.

Parameters

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

- **view** (*bool*, *optional*) – Open the url site with the sys default opener.
- **get_context** (*function* | *dict*, *optional*) – Function to pass extra data as form context.
- **get_data** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **edit_on_change** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **pre_submit** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **post_submit** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run* or *view* is *True*.

Return type

FormMap | Site

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.

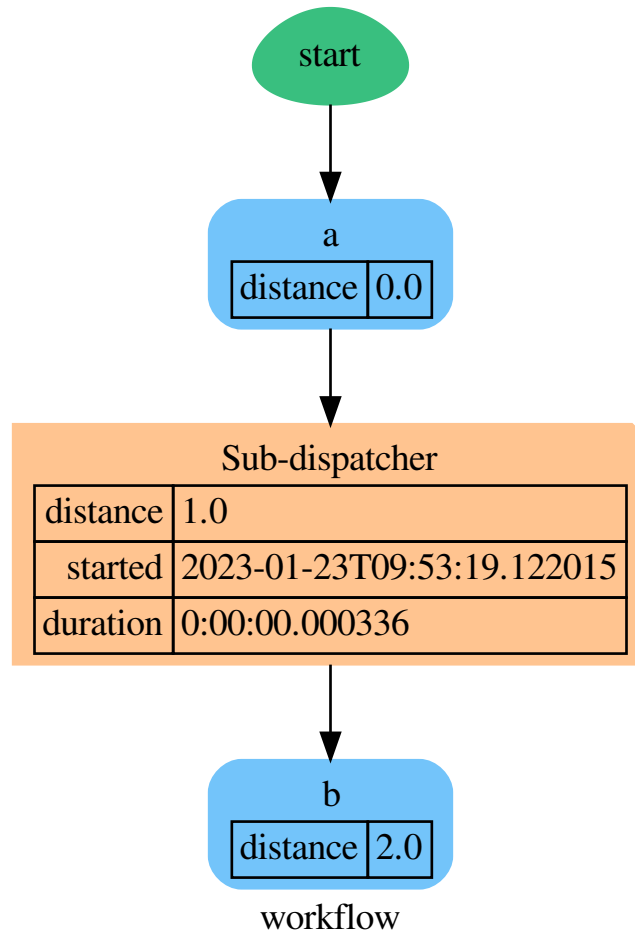
Returns

Node attributes and its real path.

Return type

(T, (str, ...))

Example:



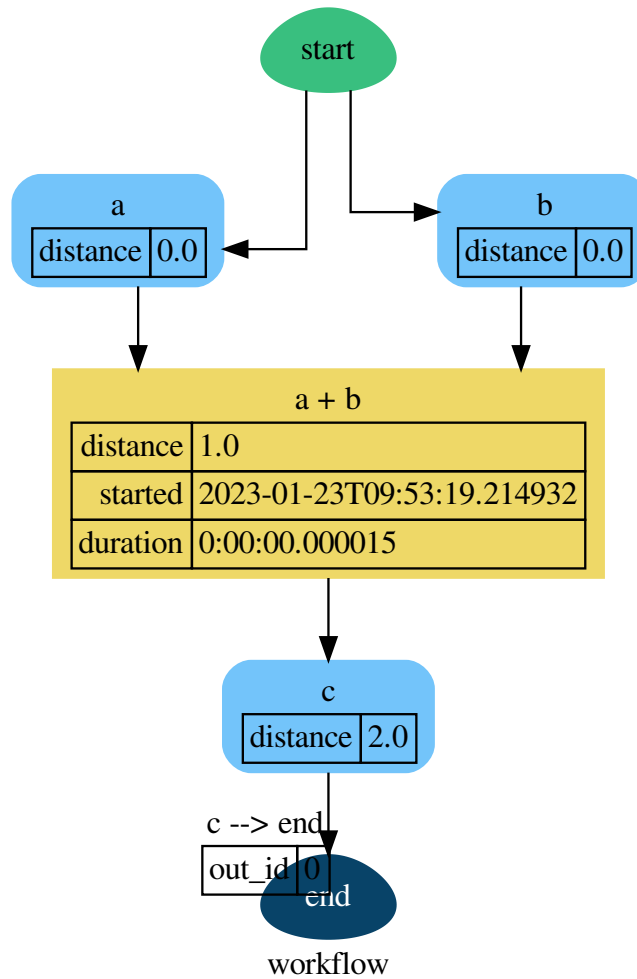
Get the sub node output:

```

>>> 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*, *raw_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=True*, *viz=False*, *short_name=None*, *executor='async'*, *render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

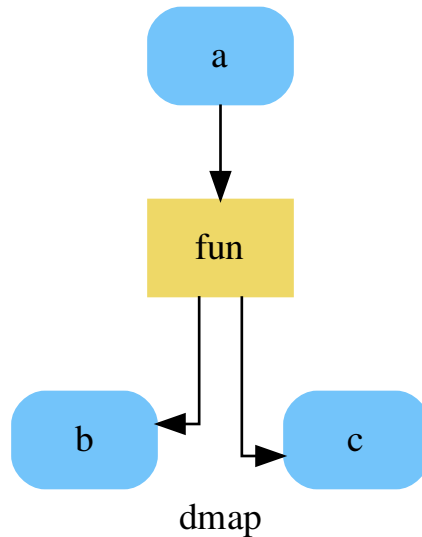
A SiteMap or a Site if .

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, subsite_idle_timeout=600*)

Creates a dispatcher Flask app.

Parameters

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

- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

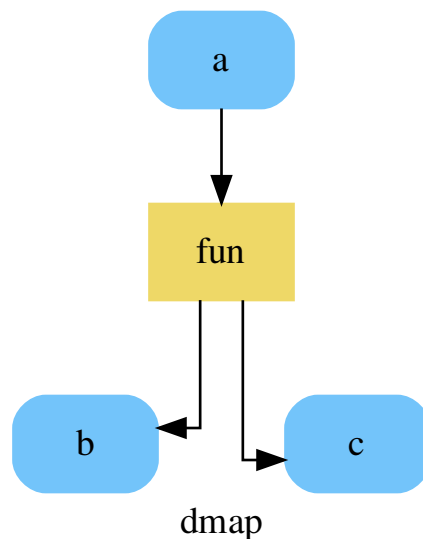
Return type

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:

```
>>> print("Starting...\n"); site = dsp.web(); 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__()`

web(*depth=-1, node_data=None, node_function=None, directory=None, sites=None, run=True, subsite_idle_timeout=600*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str], optional*) – Data node attributes to produce API.
- **node_function** (*tuple[str], optional*) – Function node attributes produce API.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[Site], optional*) – A set of [Site](#) to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?
- **subsite_idle_timeout** (*int, optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

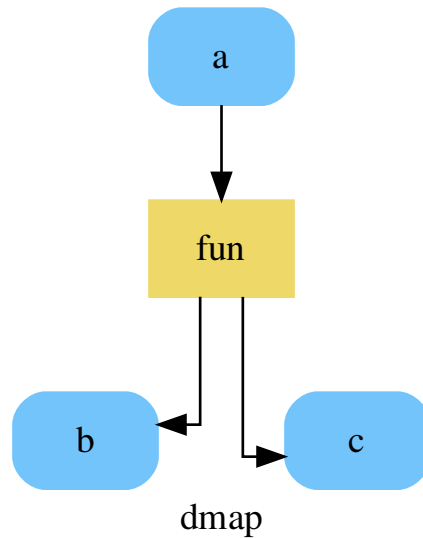
Return type

[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:

```

>>> print("Starting...\n"); site = dsp.web(); 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.

form(depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600)

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int*, optional) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple*[*str*], optional) – Data node attributes to produce API.
- **node_function** (*tuple*[*str*], optional) – Function node attributes produce API.
- **directory** (*str*, optional) – Where is the generated Flask app root located?

- **sites** (*set[Site]*, *optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool*, *optional*) – Run the backend server?
- **view** (*bool*, *optional*) – Open the url site with the sys default opener.
- **get_context** (*function* | *dict*, *optional*) – Function to pass extra data as form context.
- **get_data** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **edit_on_change** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **pre_submit** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **post_submit** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run* or *view* is *True*.

Return type

FormMap | Site

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*, *raw_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=True*, *viz=False*, *short_name=None*, *executor='async'*, *render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[Site]*, *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

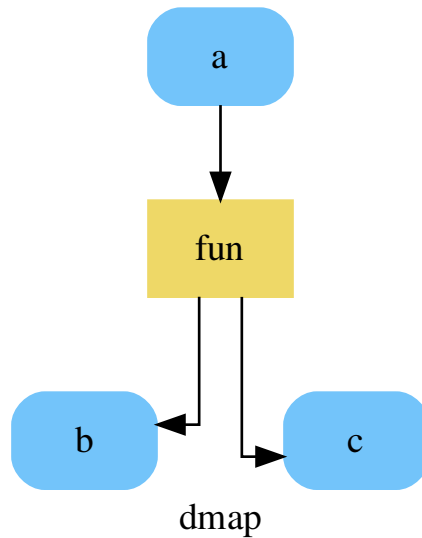
A SiteMap or a Site if .

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.

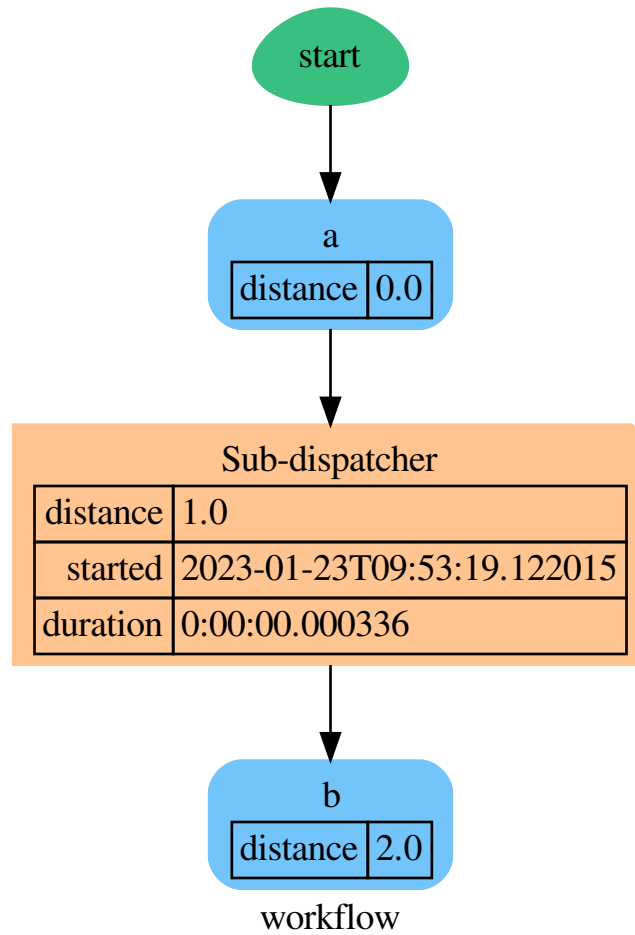
Returns

Node attributes and its real path.

Return type

(T, (*str*, ...))

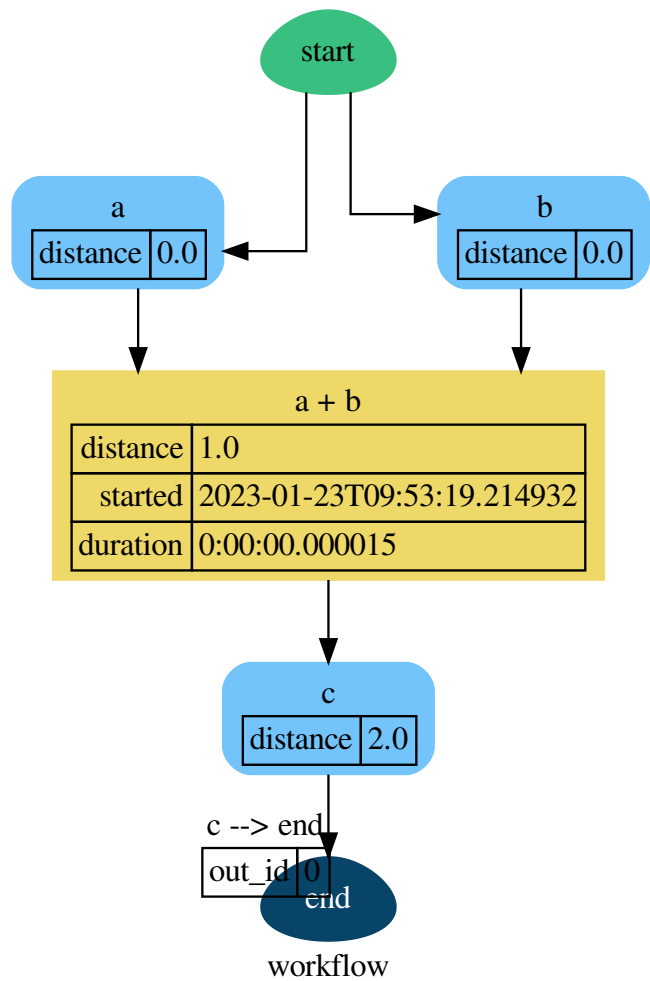
Example:



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=False*)

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:

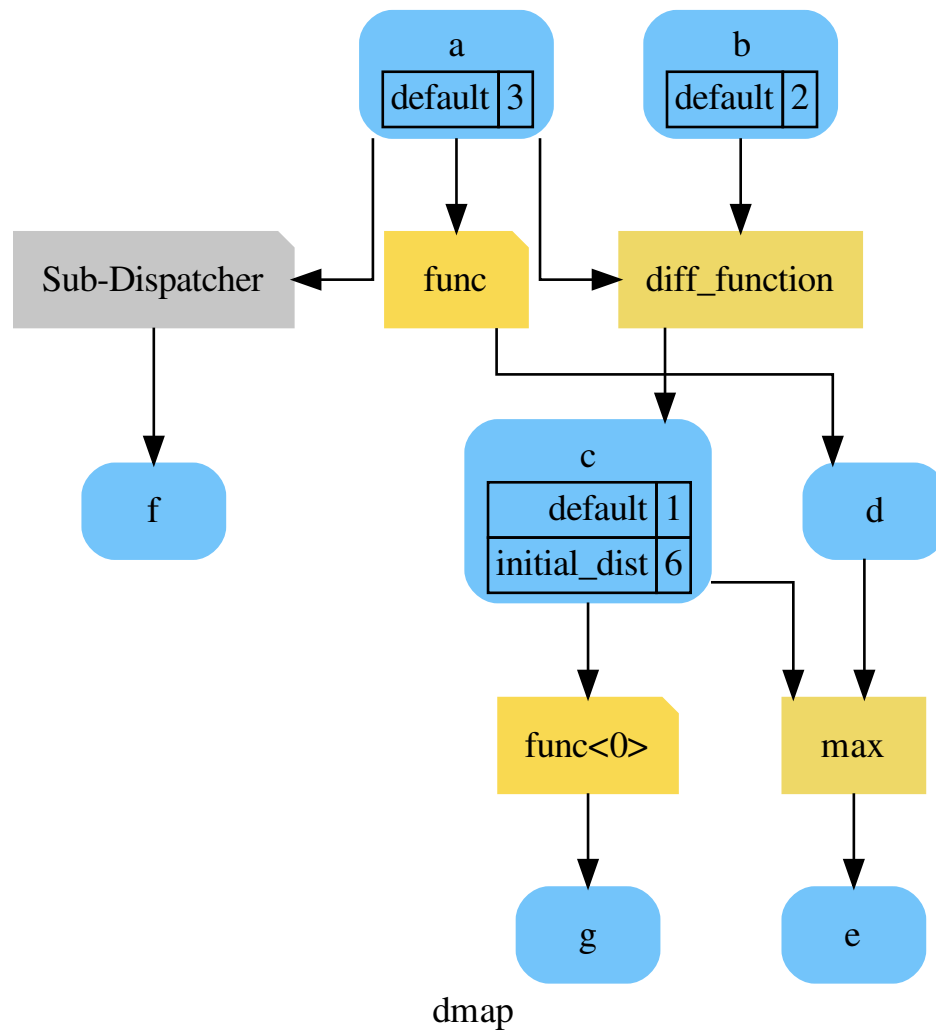
```
>>> blue.set_default_value(data_id='c', value=1, initial_dist=6)
<schedula.utils.blue.BlueDispatcher object at ...>
```

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 ...>
```

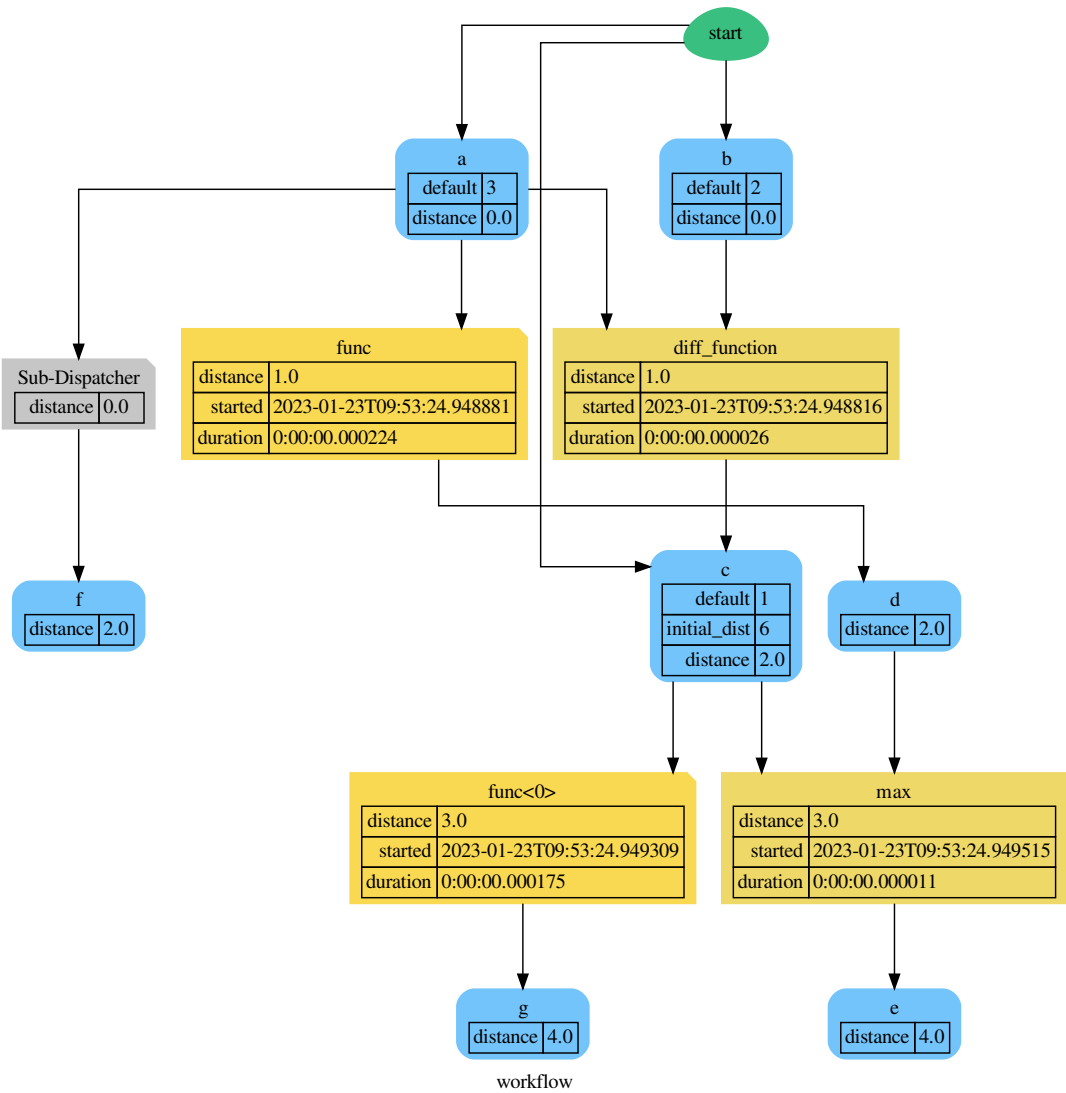
Finally you can create the dispatcher object using the method *new*:

```
>>> 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> | |
| <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 <i>Blueprint.cls</i> 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=False)`

`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=None, outputs=None, dsp_id=None, input_domain=None, weight=None, inp_weight=None, description=None, include_defaults=False, await_domain=None, inputs_prefix="", outputs_prefix="", **kwargs)`

Add a single sub-dispatcher node to dispatcher.

Parameters

- **dsp** (*BlueDispatcher | 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. If *None* all child dispatcher nodes are used as inputs.
- **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. If *None* all child dispatcher nodes are used as outputs.
- **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.
- **inputs_prefix** (*str*) – Add a prefix to parent dispatcher inputs nodes.
- **outputs_prefix** (*str*) – Add a prefix to parent dispatcher outputs nodes.
- **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=False)`

`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*=None, *outputs*=None, *dsp_id*=None, *input_domain*=None, *weight*=None, *inp_weight*=None, *description*=None, *include_defaults*=False, *await_domain*=None, *inputs_prefix*="", *outputs_prefix*="", ***kwargs*)

Add a single sub-dispatcher node to dispatcher.

Parameters

- **dsp** (*BlueDispatcher* | *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. If *None* all child dispatcher nodes are used as inputs.
- **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. If *None* all child dispatcher nodes are used as outputs.
- **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.
- **inputs_prefix** (*str*) – Add a prefix to parent dispatcher inputs nodes.
- **outputs_prefix** (*str*) – Add a prefix to parent dispatcher outputs nodes.
- **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> | |
| <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)`

`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)

cls

alias of *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

get_attr_doc

get_link

get_summary

search_node_description

get_attr_doc

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

get_link

get_link(**items*)

get_summary

get_summary(*doc*)

search_node_description

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

7.2.7 drw

It provides functions to plot dispatcher map and workflow.

Sub-Modules:

nodes

It provides docutils nodes to plot dispatcher map and workflow.

nodes

It provides docutils nodes to plot dispatcher map and workflow.

Functions

autoplot_callback

autoplot_function

before_request

cached_view

get_match_func

jinja2_format

parse_funcs

render_output

run_server

site_view

uncpath

update_filenames

valid_filename

autoplot_callback

autoplot_callback(*res*)

autoplot_function

autoplot_function(*kwargs*)

before_request

before_request(*mute*)

cached_view

cached_view(*node, directory, context, rendered, viz=False, executor='async', **render_ctx*)

get_match_func

get_match_func(*expr*)

jinja2_format

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

parse_funcs

parse_funcs(*expr, funcs*)

render_output

render_output(*out, pformat*)

run_server

run_server(*app, options*)

site_view

site_view(*context, rendered, rules, root, filepath=None, viz=False, executor='async'*)

uncpath

uncpath(*p*)

update_filenames

update_filenames(*node, filenames*)

valid_filename

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

Classes

FolderNode

IdleContainer

NoView

ReverseProxied

ServerThread

Site

SiteFolder

SiteIndex

SiteMap

SiteNode

SiteViz

FolderNode

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

Methods

`__init__`

`dot`

`href`

`items`

`parent_ref`

`render_funcs`

`render_size`

`style`

`yield_attr`

`__init__`

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

`dot`

`FolderNode.dot(context=None)`

`href`

`FolderNode.href(context, link_id)`

`items`

`FolderNode.items()`

`parent_ref`

`FolderNode.parent_ref(context, 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*)

Attributes

counter

edge_data

max_lines

max_width

node_data

node_function

node_map

node_styles

re_node

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_result', '+function|solution', 'weight', 'remote_links',
'+filters|solution_filters', 'distance', '!error', '*output')

node_function

FolderNode.node_function = ('-', '.tooltip', 'await_domain', 'await_result',
'+input_domain|solution_domain', 'weight', '+filters|solution_filters',
'missing_inputs_outputs', 'distance', 'started', 'duration', '!error',
'*function|solution')

node_map

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

node_styles


```

FolderNode.node_styles = {'error': {empty: {'fillcolor': '#FFFFFF', 'label':
'empty', 'shape': 'egg'}, end: {'color': '#084368', 'fillcolor': '#084368',
'fontcolor': '#FFFFFF', 'label': 'end', 'ordering': 'in', 'shape': 'egg'}, none:
{'data': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'penwidth': 2, 'shape':
'box', 'style': 'rounded,filled'}, 'dispatcher': {'color': '#5E1F00',
'fillcolor': '#FF3536', 'penwidth': 2, 'shape': 'note', 'style': 'filled'},
'dispatchpipe': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in',
'penwidth': 2, 'shape': 'note', 'style': 'filled'}, 'edge': {None: None},
'function': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in',
'penwidth': 2, 'shape': 'box'}, 'function-dispatcher': {'color': '#5E1F00',
'fillcolor': '#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape': 'note'},
'mapdispatch': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in',
'penwidth': 2, 'shape': 'note', 'style': 'filled'}, 'run_model': {'color':
'#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape':
'note'}, 'subdispatch': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'penwidth':
2, 'shape': 'note', 'style': 'filled'}, 'subdispatchfunction': {'color':
'#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape':
'note', 'style': 'filled'}, 'subdispatchpipe': {'color': '#5E1F00', 'fillcolor':
'#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape': 'note', 'style':
'filled'}}}, plot: {'color': '#fcf3dd', 'fillcolor': '#fcf3dd', 'label': 'plot',
'shape': 'egg'}, self: {'color': '#C1A4FE', 'fillcolor': '#C1A4FE', 'label':
'self', 'shape': 'egg'}, sink: {'color': '#303030', 'fillcolor': '#303030',
'fontcolor': '#FFFFFF', 'label': 'sink', 'shape': 'egg'}, start: {'color':
'#39bf7f', 'fillcolor': '#39bf7f', 'label': 'start', 'ordering': 'out', 'shape':
'egg'}}, 'info': {empty: {'fillcolor': '#FFFFFF', 'label': 'empty', 'shape':
'egg'}, end: {'color': '#084368', 'fillcolor': '#084368', 'fontcolor':
'#FFFFFF', 'label': 'end', 'ordering': 'in', 'shape': 'egg'}, none: {'data':
{'color': '#73c4fa', 'fillcolor': '#73c4fa', 'shape': 'box', 'style':
'rounded,filled'}, 'dispatcher': {'color': '#c6c6c6', 'fillcolor': '#c6c6c6',
'shape': 'note', 'style': 'filled'}, 'dispatchpipe': {'color': '#e8c268',
'fillcolor': '#e8c268', 'ordering': 'in', 'shape': 'note', 'style': 'filled'},
'edge': {None: None}, 'function': {'color': '#eed867', 'fillcolor': '#eed867',
'ordering': 'in', 'shape': 'box'}, 'function-dispatcher': {'color': '#eed867',
'fillcolor': '#eed867', 'ordering': 'in', 'shape': 'note'}, 'mapdispatch':
{'color': '#f4bd6a', 'fillcolor': '#f4bd6a', 'ordering': 'in', 'shape': 'note',
'style': 'filled'}, 'run_model': {'color': '#eed867', 'fillcolor': '#eed867',
'ordering': 'in', 'shape': 'note'}, 'subdispatch': {'color': '#ffc490',
'fillcolor': '#ffc490', 'shape': 'note', 'style': 'filled'},
'subdispatchfunction': {'color': '#f9d951', 'fillcolor': '#f9d951', 'ordering':
'in', 'shape': 'note', 'style': 'filled'}, 'subdispatchpipe': {'color':
'#f1cd5d', 'fillcolor': '#f1cd5d', 'ordering': 'in', 'shape': 'note', 'style':
'filled'}}}, plot: {'color': '#fcf3dd', 'fillcolor': '#fcf3dd', 'label': 'plot',
'shape': 'egg'}, self: {'color': '#C1A4FE', 'fillcolor': '#C1A4FE', 'label':
'self', 'shape': 'egg'}, sink: {'color': '#303030', 'fillcolor': '#303030',
'fontcolor': '#FFFFFF', 'label': 'sink', 'shape': 'egg'}, start: {'color':
'#39bf7f', 'fillcolor': '#39bf7f', 'label': 'start', 'ordering': 'out', 'shape':
'egg'}}, 'warning': {empty: {'fillcolor': '#FFFFFF', 'label': 'empty', 'shape':
'egg'}, end: {'color': '#084368', 'fillcolor': '#084368', 'fontcolor':
'#FFFFFF', 'label': 'end', 'ordering': 'in', 'shape': 'egg'}, none: {'data':
{'color': '#C9340A', 'fillcolor': '#fea22b', 'penwidth': 2, 'shape': 'box',
'style': 'rounded,filled'}, 'dispatcher': {'color': '#C9340A', 'fillcolor':
'#fea22b', 'penwidth': 2, 'shape': 'note', 'style': 'filled'}, 'dispatchpipe':
{'color': '#C9340A', 'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2,
'shape': 'note', 'style': 'filled'}, 'edge': {None: None}, 'function':
{'color': '#C9340A', 'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2,
'shape': 'box'}, 'function-dispatcher': {'color': '#C9340A', 'fillcolor':
'#fea22b', 'ordering': 'in', 'penwidth': 2, 'shape': 'note'}, 'mapdispatch':
{'color': '#C9340A', 'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2,
'shape': 'note', 'style': 'filled'}, 'run_model': {'color': '#C9340A',
'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2, 'shape': 'note'},

```

re_node

```
FolderNode.re_node = '^([.*+!]?)([\\w ]+)(?>\\|([\\w ]+))?$'
```

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

IdleContainer

```
class IdleContainer(interval=1)
```

Methods

| | |
|------------------------|--|
| <code>__init__</code> | This constructor should always be called with keyword arguments. |
| <code>add</code> | |
| <code>getName</code> | |
| <code>isAlive</code> | Return whether the thread is alive. |
| <code>isDaemon</code> | |
| <code>is_alive</code> | Return whether the thread is alive. |
| <code>join</code> | Wait until the thread terminates. |
| <code>run</code> | Method representing the thread's activity. |
| <code>setDaemon</code> | |
| <code>setName</code> | |
| <code>start</code> | Start the thread's activity. |

`__init__`

`IdleContainer.__init__(interval=1)`

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

add

`IdleContainer.add(site)`

getName

`IdleContainer.getName()`

isAlive

`IdleContainer.isAlive()`

Return whether the thread is alive.

This method is deprecated, use `is_alive()` instead.

isDaemon

`IdleContainer.isDaemon()`

is_alive

`IdleContainer.is_alive()`

Return whether the thread is alive.

This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

join

`IdleContainer.join(timeout=None)`

Wait until the thread terminates.

This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the timeout argument is not present or `None`, the operation will block until the thread terminates.

A thread can be `join()`ed many times.

`join()` raises a `RuntimeError` if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to `join()` a thread before it has been started and attempts to do so raises the same exception.

run

`IdleContainer.run()`

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

setDaemon

`IdleContainer.setDaemon(daemonic)`

setName

`IdleContainer.setName(name)`

start

`IdleContainer.start()`

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

`__init__(interval=1)`

This constructor should always be called with keyword arguments. Arguments are:

group should be `None`; reserved for future extension when a `ThreadGroup` class is implemented.

target is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to `()`.

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

NoView

class NoView

Methods

`__init__`

`__init__`

NoView.`__init__`()

`__init__`()

ReverseProxied

class ReverseProxied(*app*, *script_name*)

Methods

`__init__`

`__init__`

ReverseProxied.`__init__`(*app*, *script_name*)

`__init__`(*app*, *script_name*)

ServerThread

class ServerThread(*application*, *threaded=True*, ***kwargs*)

Methods

| | |
|------------------------|--|
| <code>__init__</code> | This constructor should always be called with keyword arguments. |
| <code>getName</code> | |
| <code>isAlive</code> | Return whether the thread is alive. |
| <code>isDaemon</code> | |
| <code>is_alive</code> | Return whether the thread is alive. |
| <code>join</code> | Wait until the thread terminates. |
| <code>run</code> | Method representing the thread's activity. |
| <code>setDaemon</code> | |
| <code>setName</code> | |
| <code>shutdown</code> | |
| <code>start</code> | Start the thread's activity. |

`__init__`

`ServerThread.__init__(application, threaded=True, **kwargs)`

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

`getName`

`ServerThread.getName()`

isAlive

ServerThread.isAlive()

Return whether the thread is alive.

This method is deprecated, use `is_alive()` instead.

isDaemon

ServerThread.isDaemon()

is_alive

ServerThread.is_alive()

Return whether the thread is alive.

This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

join

ServerThread.join(timeout=None)

Wait until the thread terminates.

This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the timeout argument is not present or `None`, the operation will block until the thread terminates.

A thread can be `join()`ed many times.

`join()` raises a `RuntimeError` if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to `join()` a thread before it has been started and attempts to do so raises the same exception.

run

ServerThread.run()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

setDaemon

ServerThread.**setDaemon**(*daemonic*)

setName

ServerThread.**setName**(*name*)

shutdown

ServerThread.**shutdown**()

start

ServerThread.**start**()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

__init__(*application*, *threaded=True*, ***kwargs*)

This constructor should always be called with keyword arguments. Arguments are:

group should be `None`; reserved for future extension when a `ThreadGroup` class is implemented.

target is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to `()`.

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

Site

```
class Site(sitemap, host='localhost', port=0, delay=0.1, until=30, run_options=None, idle_timeout=0,
           url_prefix=None, **kwargs)
```

Methods

`__init__`

`alive_view`

`app`

`format`

`get_port`

`run`

`shutdown_site`

`update_last_activity`

`wait_server`

`__init__`

```
Site.__init__(sitemap, host='localhost', port=0, delay=0.1, until=30, run_options=None, idle_timeout=0,
              url_prefix=None, **kwargs)
```

alive_view

```
static Site.alive_view()
```

app

```
Site.app()
```

format

Site.**format**(*string*)

get_port

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

run

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

shutdown_site

static Site.**shutdown_site**(*shutdown, subsites*)

update_last_activity

Site.**update_last_activity**(*response=None*)

wait_server

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

__init__(*sitemap, host='localhost', port=0, delay=0.1, until=30, run_options=None, idle_timeout=0, url_prefix=None, **kwargs*)

SiteFolder

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

Methods

[`__init__`](#)

[`dot`](#)

[`view`](#)

`__init__`

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

`dot`

`SiteFolder.dot(context=None)`

`view`

`SiteFolder.view(filepath, context=None, viz=False, **kwargs)`

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

Attributes

`counter`

`digraph`

`ext`

`counter`

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

`digraph`

`SiteFolder.digraph = {'body': {'splines': 'ortho', 'style': 'filled'}, 'edge_attr': {}, 'format': 'svg', 'graph_attr': {'bgcolor': 'transparent', 'nslimit': '1', 'nslimit1': '1'}, 'node_attr': {'style': 'filled'}}`

`ext`

`SiteFolder.ext = 'html'`

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

SiteIndex

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

Methods

`__init__`

`legend`

`render`

`view`

`__init__`

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

legend

static SiteIndex.**legend**(*viz*=False, *executor*='async', ***kwargs*)

render

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

view

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

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

Attributes

`counter`

`ext`

counter

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

ext

SiteIndex.ext = 'html'

SiteMap

class SiteMap

Methods

| | |
|----------------------------|--|
| <code>__init__</code> | |
| <code>add_items</code> | |
| <code>app</code> | |
| <code>basic_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_directory</code> | |
| <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__()`

`add_items`

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

`app`

`SiteMap.app(root_path=None, depth=-1, index=True, mute=True, viz_js=False, executor='async', blueprint_name=None, **kw)`

`basic_app`

`SiteMap.basic_app(root_path, mute=True, blueprint_name=None, **kwargs)`

`clear`

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

`copy`

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

`fromkeys`

`SiteMap.fromkeys(value=None)`

Create a new ordered dictionary with keys from iterable and values set to value.

`get`

`SiteMap.get(key, default=None, /)`

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

`get_directory`

`SiteMap.get_directory(directory)`

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(*key*, *last=True*)

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(*last=True*)

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*, *viz=False*, *viz_js=False*, *executor='async'*)

rules

`SiteMap.rules(depth=-1, index=True, viz_js=False)`

setdefault

`SiteMap.setdefault(key, default=None)`

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__()`

Attributes

`blueprint_name`

`directory`

`include_folders_as_filenames`

`options`

`short_name`

blueprint_name

```
SiteMap.blueprint_name = None
```

directory

```
SiteMap.directory = None
```

include_folders_as_filenames

```
SiteMap.include_folders_as_filenames = True
```

options

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

short_name

```
SiteMap.short_name = None
```

SiteNode

```
class SiteNode(folder, node_id, item, obj, dsp_node_id, short_name=None)
```

Methods

```
__init__
```

```
render
```

```
view
```

```
__init__
```

```
SiteNode.__init__(folder, node_id, item, obj, dsp_node_id, short_name=None)
```

render

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

view

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

`__init__(folder, node_id, item, obj, dsp_node_id, short_name=None)`

Attributes

`counter`

`ext`

counter

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

ext

`SiteNode.ext = 'html'`

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

SiteViz

`class SiteViz(sitemap, node_id='viz')`

Methods

`__init__`

`render`

`view`

`__init__`

`SiteViz.__init__(sitemap, node_id='viz')`

`render`

`SiteViz.render(context, *args, **kwargs)`

`view`

`SiteViz.view(filepath, *args, **kwargs)`

`__init__(sitemap, node_id='viz')`

Attributes

`counter`

`ext`

`counter`

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

`ext`

`SiteViz.ext = 'js'`

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

Returns

Decorator.

Return type

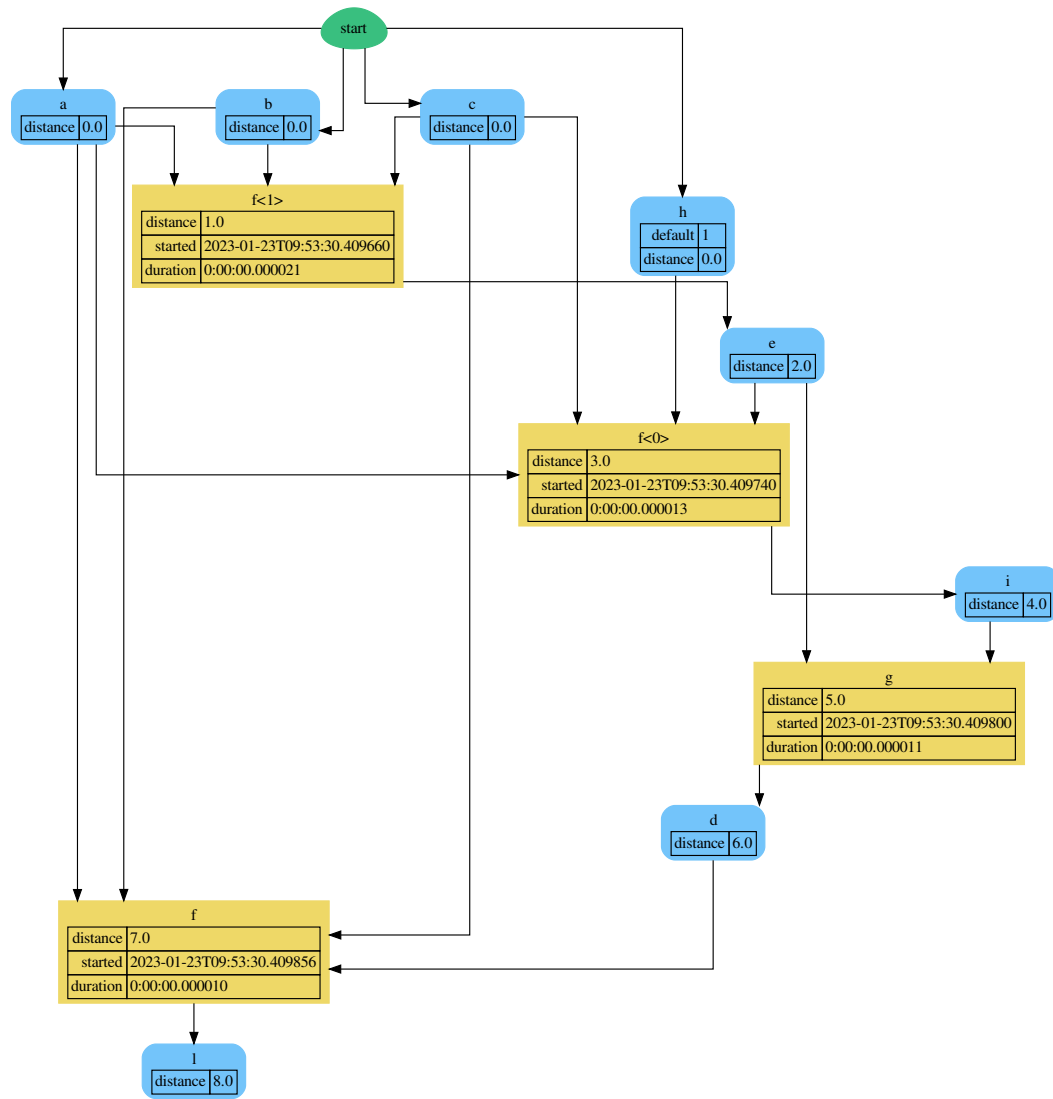
callable

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):
```

(continues on next page)

```
...     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)])
```



workflow

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:

```
>>> import schedula as sh
>>> fun = sh.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:

```
>>> import schedula as sh
>>> fun = sh.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>MapDispatch</i> | It dynamically builds a <i>Dispatcher</i> that is used to invoke recursively a <i>dispatching function</i> that is defined by a constructor function that takes a <i>dsp</i> base model as input. |
| <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. |
| <i>add_args</i> | |
| <i>inf</i> | Class to model infinite numbers for workflow distance. |
| <i>run_model</i> | It is an utility function to execute dynamically generated function/models and - if Dispatcher based - add their workflows to the parent solution. |

DispatchPipe

class `DispatchPipe(dsp=None, *args, **kwargs)`

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

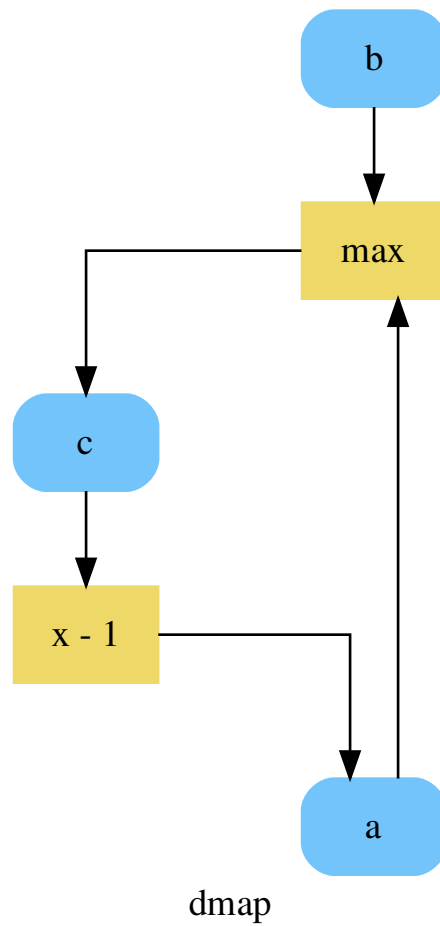
Note: This wrapper is not thread safe, because it overwrite the solution.

See also:

dispatch(), *shrink_dsp()*

Example:

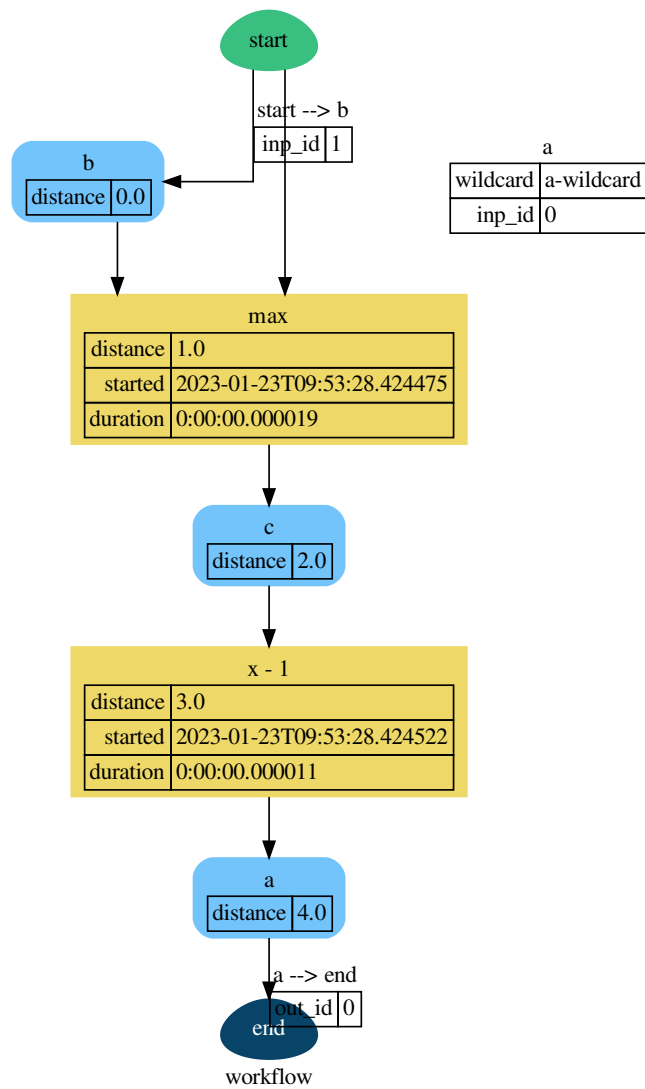
A dispatcher with two functions *max* and *min* and an unresolved cycle (i.e., $a \rightarrow \text{max} \rightarrow c \rightarrow \text{min} \rightarrow a$):



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

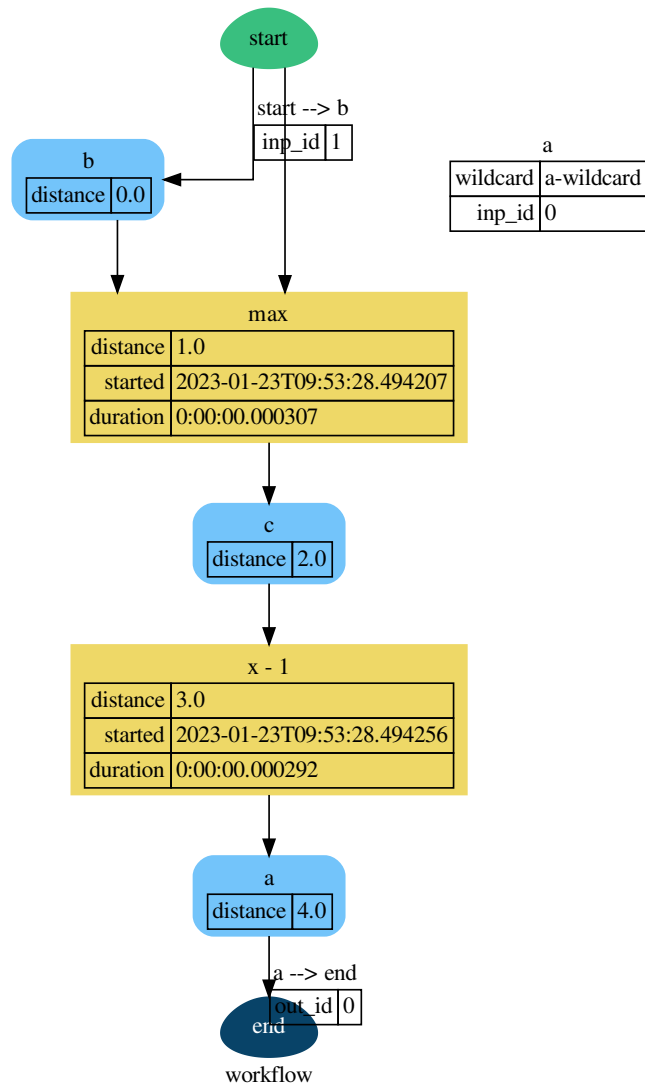
```

>>> 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>form</code> | Creates a dispatcher Form Flask app. |
| <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, inputs_dist=None, no_domain=True, wildcard=True, shrink=True, output_type=None, output_type_kw=None, first_arg_as_kw=False)`

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (`schedula.Dispatcher` | `schedula.utils.blue.BlueDispatcher`) – 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.
- **inputs_dist** (`dict[str, int | float]`, `optional`) – Initial distances of input data nodes.
- **no_domain** (`bool`, `optional`) – Skip the domain check.
- **shrink** (`bool`, `optional`) – If True the dispatcher is shrink before the dispatch.
- **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.
- **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`.
- **output_type_kw** (`bool`) – Extra kwargs to pass to the `selector` function.
- **first_arg_as_kw** – Converts first argument of the `__call__` method as `kwargs`.

`blue`

`DispatchPipe.blue(memo=None, depth=-1)`

Constructs a Blueprint out of the current object.

Parameters

- **memo** (`dict[T, schedula.utils.blue.Blueprint]`) – A dictionary to cache Blueprints.
- **depth** (`int`, `optional`) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

Returns

A Blueprint of the current object.

Return type

`schedula.utils.blue.Blueprint`

copy

`DispatchPipe.copy()`

form

`DispatchPipe.form(depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600)`

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int*, optional) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str]*, optional) – Data node attributes to produce API.
- **node_function** (*tuple[str]*, optional) – Function node attributes produce API.
- **directory** (*str*, optional) – Where is the generated Flask app root located?
- **sites** (*set[Site]*, optional) – A set of [Site](#) to maintain alive the backend server.
- **run** (*bool*, optional) – Run the backend server?
- **view** (*bool*, optional) – Open the url site with the sys default opener.
- **get_context** (*function | dict*, optional) – Function to pass extra data as form context.
- **get_data** (*function | dict*, optional) – Function to initialize the formdata.
- **edit_on_change** (*function | dict*, optional) – Function to initialize the formdata.
- **pre_submit** (*function | dict*, optional) – Function to initialize the formdata.
- **post_submit** (*function | dict*, optional) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int*, optional) – Idle timeout of a debug subsite in seconds.

Returns

A `FormMap` or a `Site` if *sites* is `None` and *run* or *view* is `True`.

Return type

[FormMap](#) | [Site](#)

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.

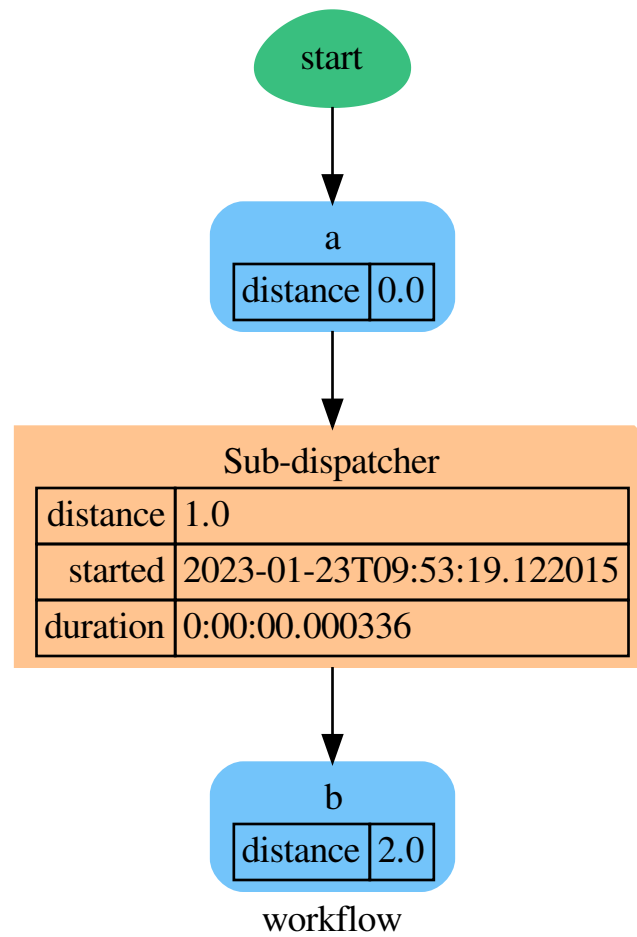
Returns

Node attributes and its real path.

Return type

(T, (*str*, ...))

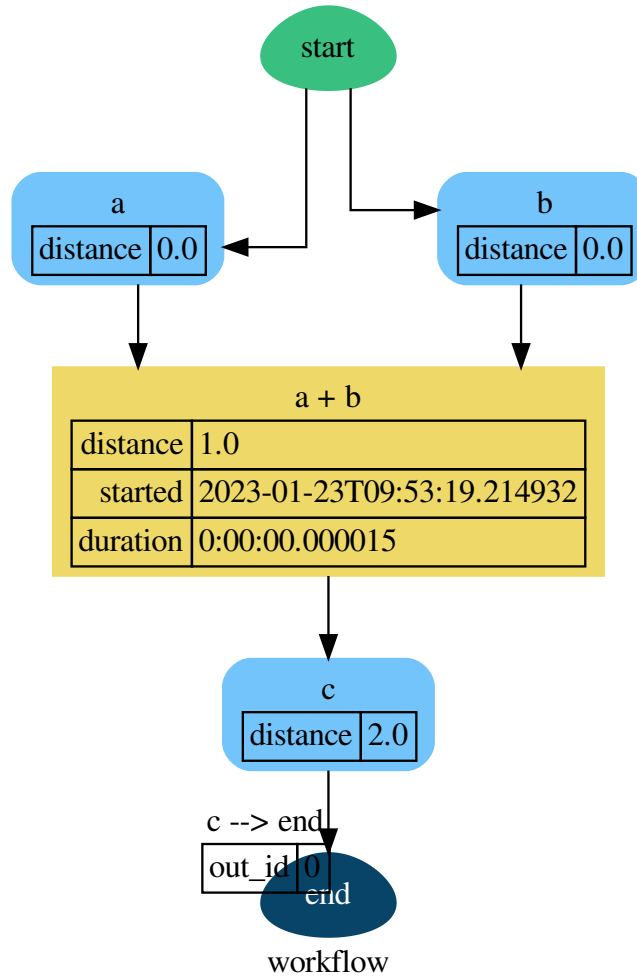
Example:



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.
- **raw_body**(*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites**(*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **index**(*bool*, *optional*) – Add the site index as first page?
- **max_lines**(*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width**(*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render**(*bool*, *optional*) – Render all pages statically?
- **viz**(*bool*, *optional*) – Use viz.js as back-end?
- **short_name**(*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor**(*str*, *optional*) – Pool executor to render object.

Returns

A SiteMap or a Site if .

Return type

schedula.utils.drw.SiteMap

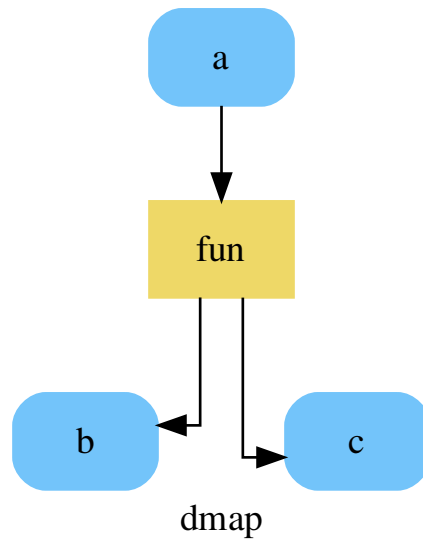
Example:

```
>>> 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

`DispatchPipe.web(depth=-1, node_data=None, node_function=None, directory=None, sites=None, run=True, subsite_idle_timeout=600)`

Creates a dispatcher Flask app.

Parameters

- **depth** (*int*, *optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str]*, *optional*) – Data node attributes to produce API.
- **node_function** (*tuple[str]*, *optional*) – Function node attributes produce API.
- **directory** (*str*, *optional*) – Where is the generated Flask app root located?
- **sites** (*set[Site]*, *optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool*, *optional*) – Run the backend server?
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

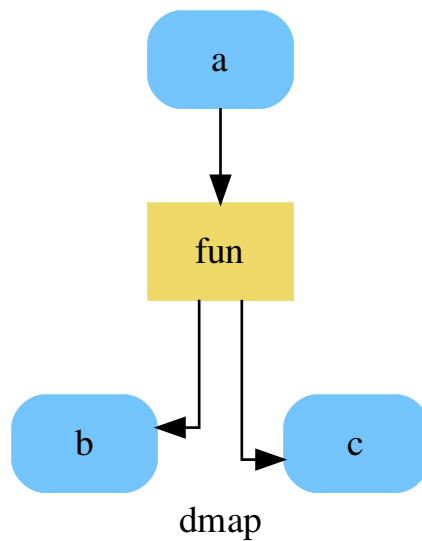
A WebMap.

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:

```
>>> print("Starting...\n"); site = dsp.web(); 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, inputs_dist=None, no_domain=True,
         wildcard=True, shrink=True, output_type=None, output_type_kw=None, first_arg_as_kw=False)
```


Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher* | *schedula.utils.blue.BlueDispatcher*) – 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.
- **inputs_dist** (*dict[str, int | float]*, *optional*) – Initial distances of input data nodes.
- **no_domain** (*bool*, *optional*) – Skip the domain check.
- **shrink** (*bool*, *optional*) – If True the dispatcher is shrink before the dispatch.
- **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.
- **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*.
- **output_type_kw** (*bool*) – Extra kwargs to pass to the *selector* function.
- **first_arg_as_kw** – Converts first argument of the `__call__` method as *kwargs*.

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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set* [*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

A SiteMap or a Site if .

Return type

schedula.utils.drw.SiteMap

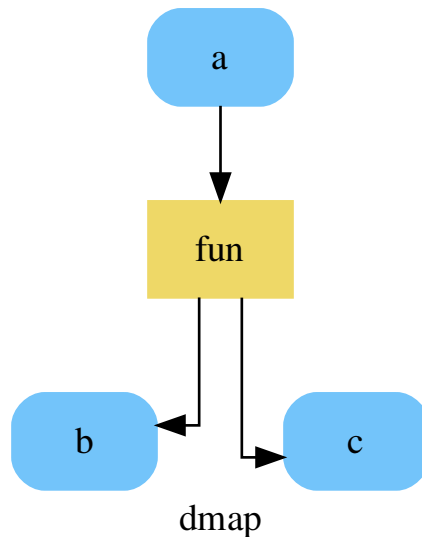
Example:

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



MapDispatch

class MapDispatch(*dsp=None, *args, **kwargs*)

It dynamically builds a *Dispatcher* that is used to invoke recursively a *dispatching function* that is defined by a constructor function that takes a *dsp* base model as input.

The created function takes a list of dictionaries as input that are used to invoke the mapping function and returns a list of outputs.

Returns

A function that executes the dispatch of the given *Dispatcher*.

Return type

callable

See also:

SubDispatch()

Example:

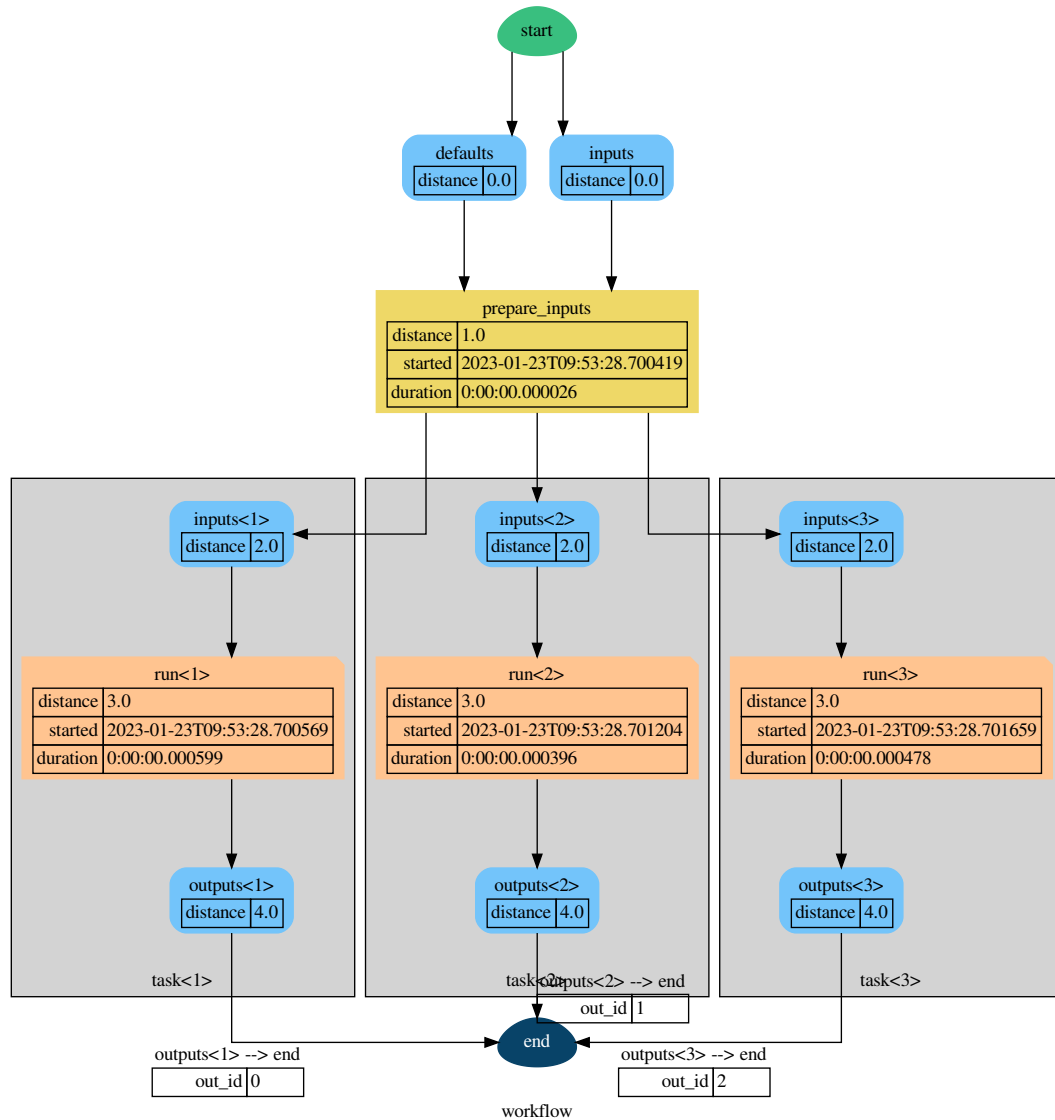
A simple example on how to use the *MapDispatch()*:

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

(continues on next page)

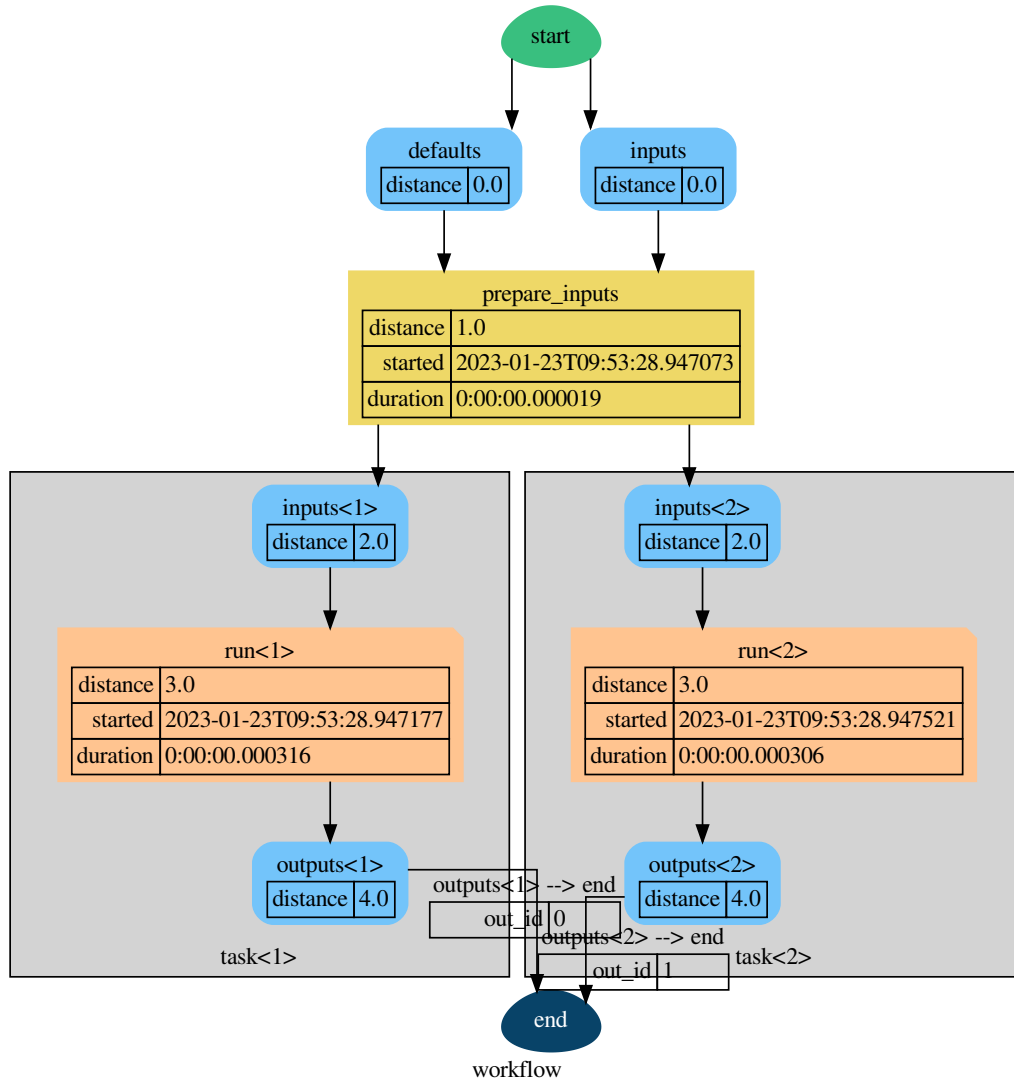
(continued from previous page)

```
...
>>> dsp.add_func(fun, ['c', 'd'], inputs_kwargs=True)
'fun'
>>> map_func = MapDispatch(dsp, constructor_kwargs={
...     'outputs': ['c', 'd'], 'output_type': 'list'
... })
>>> map_func([{'a': 1, 'b': 2}, {'a': 2, 'b': 2}, {'a': 3, 'b': 2}])
[[3, -1], [4, 0], [5, 1]]
```



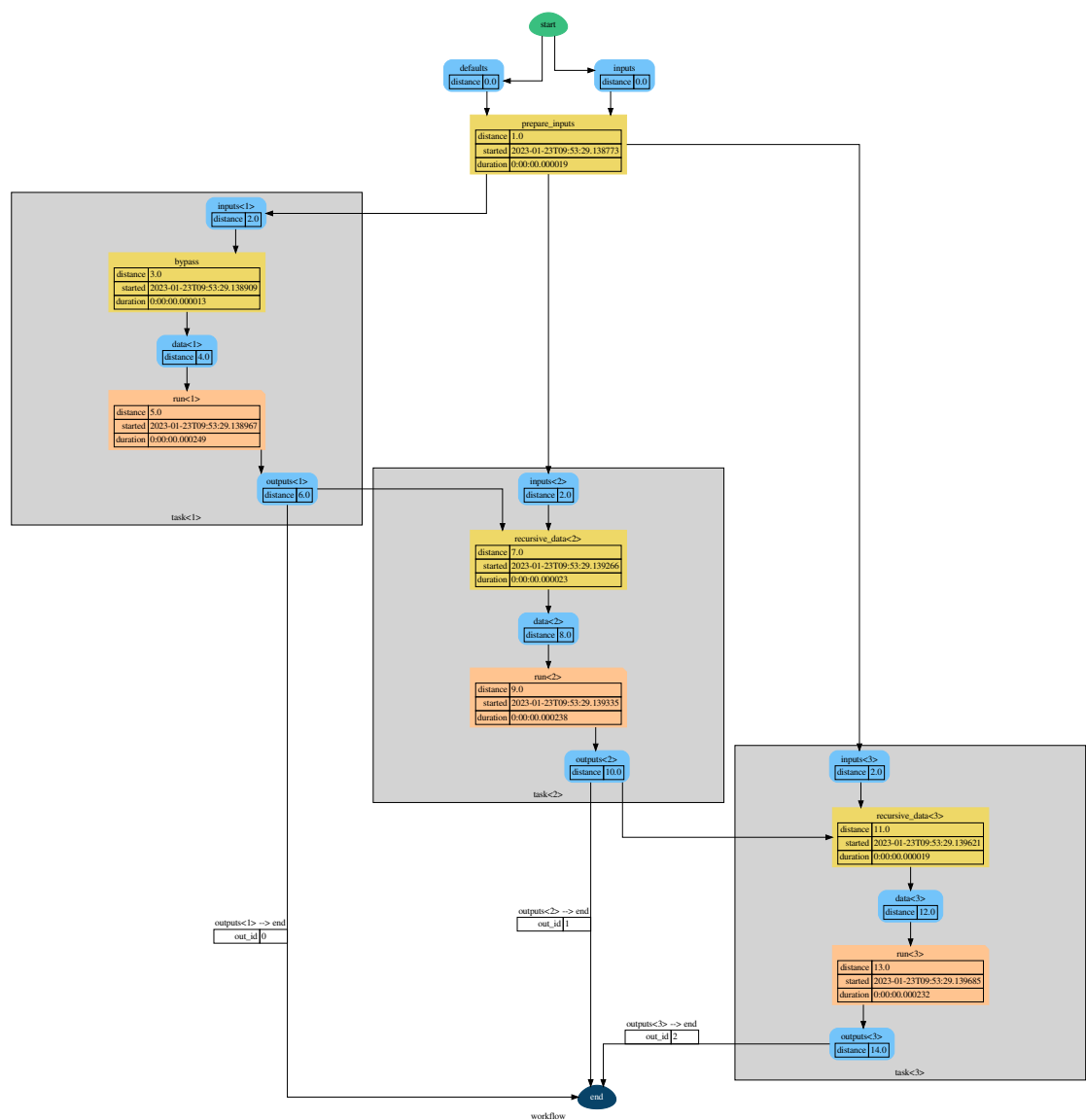
The execution model is created dynamically according to the length of the provided inputs. Moreover, the *MapDispatch()* has the possibility to define default values, that are recursively merged with the input provided to the *dispatching function* as follow:

```
>>> map_func([{'a': 1}, {'a': 3, 'b': 3}], defaults={'b': 2})
[[3, -1], [6, 0]]
```



The `MapDispatch()` can also be used as a partial reducing function, i.e., part of the output of the previous step are used as input for the successive execution of the *dispatching function*. For example:

```
>>> map_func = MapDispatch(dsp, recursive_inputs={'c': 'b'})
>>> map_func([{'a': 1, 'b': 1}, {'a': 2}, {'a': 3}])
[Solution([('a', 1), ('b', 1), ('c', 2), ('d', 0)]),
 Solution([('a', 2), ('b', 2), ('c', 4), ('d', 0)]),
 Solution([('a', 3), ('b', 4), ('c', 7), ('d', -1)])]
```



Methods

| | |
|------------------------------|--|
| <code>__init__</code> | Initializes the MapDispatch function. |
| <code>blue</code> | Constructs a Blueprint out of the current object. |
| <code>copy</code> | |
| <code>form</code> | Creates a dispatcher Form Flask app. |
| <code>format_clusters</code> | |
| <code>format_labels</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>prepare_inputs</code> | |
| <code>recursive_data</code> | |
| <code>web</code> | Creates a dispatcher Flask app. |

`__init__`

MapDispatch.`__init__`(*dsp*, *defaults=None*, *recursive_inputs=None*, *constructor=<class 'schedula.utils.dsp.SubDispatch'>*, *constructor_kwargs=None*, *function_id=None*, *func_kw=<function MapDispatch.<lambda>>*, *input_label='inputs<{}>'*, *output_label='outputs<{}>'*, *data_label='data<{}>'*, *cluster_label='task<{}>'*, ***kwargs*)

Initializes the MapDispatch function.

Parameters

- **dsp** (*schedula.Dispatcher* | *schedula.utils.blue.BlueDispatcher*) – A dispatcher that identifies the base model.
- **defaults** (*dict*) – Defaults values that are recursively merged with the input provided to the *dispatching function*.
- **recursive_inputs** (*list* | *dict*) – List of data node ids that are extracted from the outputs of the *dispatching function* and then merged with the inputs of the its successive evaluation. If a dictionary is given, this is used to rename the data node ids extracted.
- **constructor** (*function* | *class*) – It initializes the *dispatching function*.
- **constructor_kwargs** (*function* | *class*) – Extra keywords passed to the constructor function.
- **function_id** (*str*, *optional*) – Function name.
- **func_kw** (*function*, *optional*) – Extra keywords to add the *dispatching function* to execution model.
- **input_label** (*str*, *optional*) – Custom label formatter for recursive inputs.
- **output_label** (*str*, *optional*) – Custom label formatter for recursive outputs.

- **data_label** (*str*, *optional*) – Custom label formatter for recursive internal data.
- **kwargs** (*object*) – Keywords to initialize the execution model.

blue

MapDispatch.**blue**(*memo=None, depth=-1*)

Constructs a Blueprint out of the current object.

Parameters

- **memo** (*dict*[*T*, *schedula.utils.blue.Blueprint*]) – A dictionary to cache Blueprints.
- **depth** (*int*, *optional*) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

Returns

A Blueprint of the current object.

Return type

schedula.utils.blue.Blueprint

copy

MapDispatch.**copy**()

form

MapDispatch.**form**(*depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600*)

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int*, *optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple*[*str*], *optional*) – Data node attributes to produce API.
- **node_function** (*tuple*[*str*], *optional*) – Function node attributes produce API.
- **directory** (*str*, *optional*) – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **run** (*bool*, *optional*) – Run the backend server?
- **view** (*bool*, *optional*) – Open the url site with the sys default opener.
- **get_context** (*function* | *dict*, *optional*) – Function to pass extra data as form context.
- **get_data** (*function* | *dict*, *optional*) – Function to initialize the formdata.

- **edit_on_change** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **pre_submit** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **post_submit** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run or view* is *True*.

Return type

FormMap | Site

format_clusters

static MapDispatch.**format_clusters**(*it*, *label*)

format_labels

static MapDispatch.**format_labels**(*it*, *label*)

get_node

MapDispatch.**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.

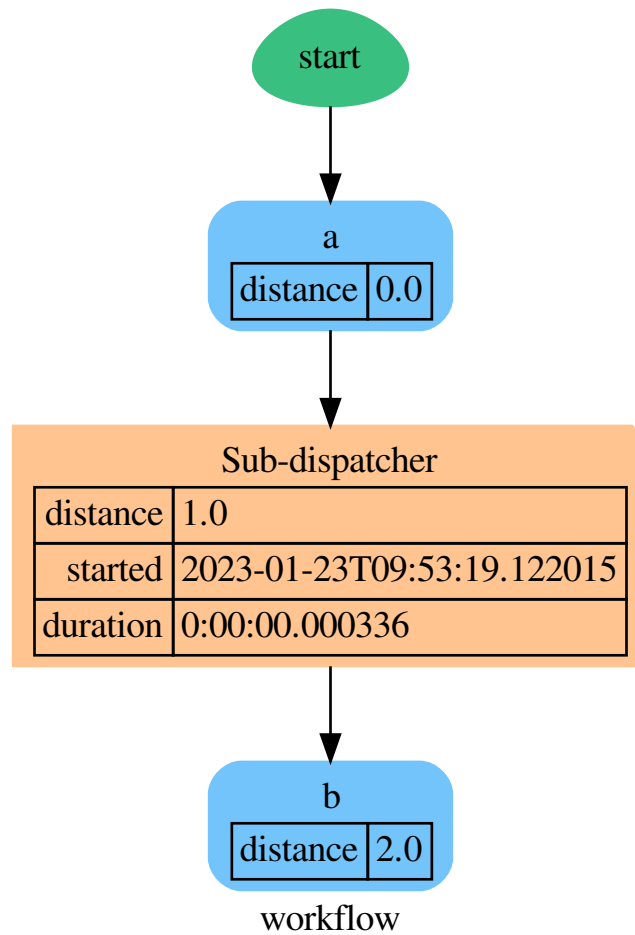
Returns

Node attributes and its real path.

Return type

(T, (str, ...))

Example:



Get the sub node output:

```

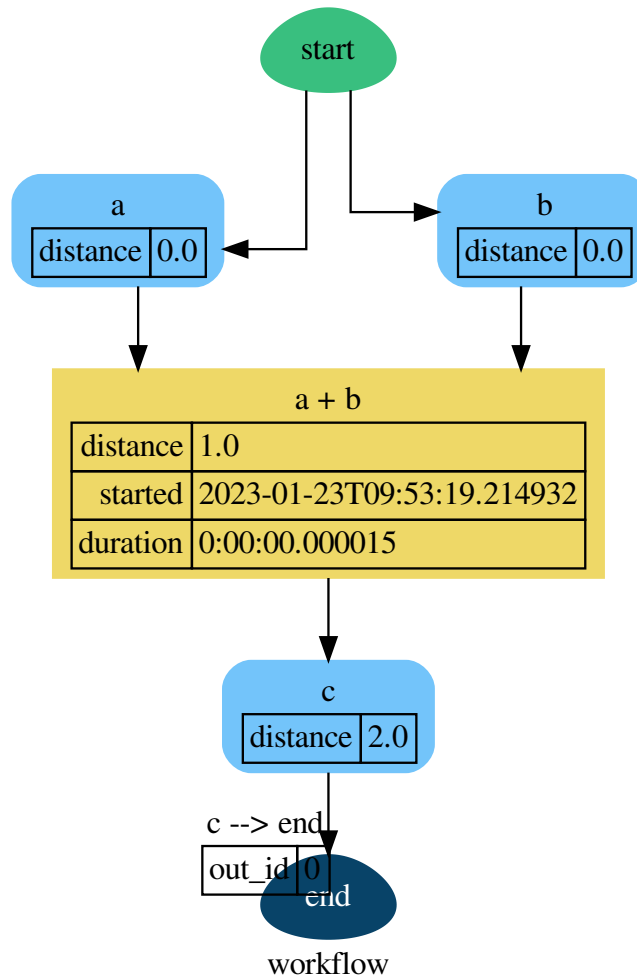
>>> dsp.get_node('Sub-dispatcher', 'c')
(4, ('Sub-dispatcher', 'c'))
>>> dsp.get_node('Sub-dispatcher', 'c', node_attr='type')
('data', ('Sub-dispatcher', 'c'))

```

```

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

```



plot

`MapDispatch.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, raw_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=True, viz=False, short_name=None, executor='async', render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

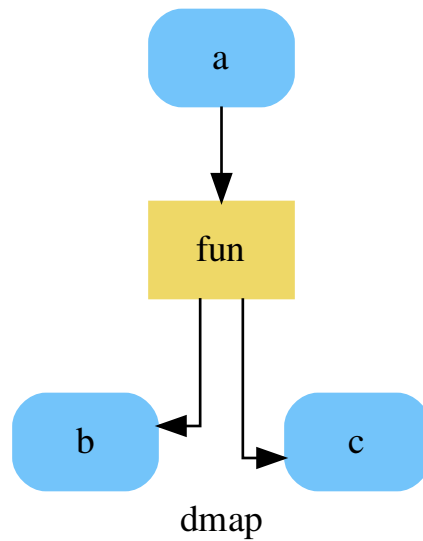
A SiteMap or a Site if .

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())])
```



prepare_inputs

static MapDispatch.**prepare_inputs**(*inputs*, *defaults*)

recursive_data

static MapDispatch.**recursive_data**(*recursive_inputs*, *input_data*, *outputs*)

web

MapDispatch.**web**(*depth=-1*, *node_data=None*, *node_function=None*, *directory=None*, *sites=None*, *run=True*, *subsite_idle_timeout=600*)

Creates a dispatcher Flask app.

Parameters

- **depth** (*int*, *optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str]*, *optional*) – Data node attributes to produce API.
- **node_function** (*tuple[str]*, *optional*) – Function node attributes produce API.
- **directory** (*str*, *optional*) – Where is the generated Flask app root located?
- **sites** (*set[Site]*, *optional*) – A set of [Site](#) to maintain alive the backend server.
- **run** (*bool*, *optional*) – Run the backend server?
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

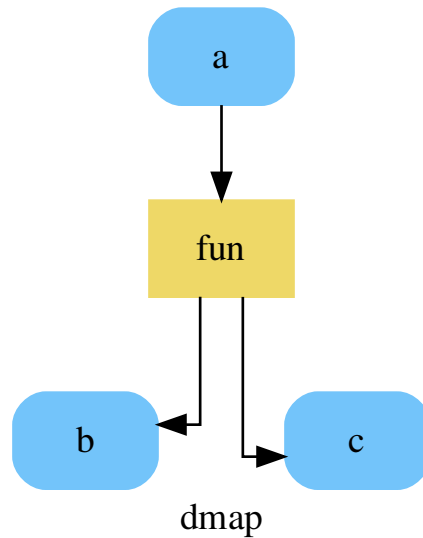
Return type

[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:

```

>>> print("Starting...\n"); site = dsp.web(); 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, defaults=None, recursive_inputs=None, constructor=<class
'schedula.utils.dsp.SubDispatch'>, constructor_kwargs=None, function_id=None,
func_kw=<function MapDispatch.<lambda>>, input_label='inputs<{}>',
output_label='outputs<{}>', data_label='data<{}>', cluster_label='task<{}>', **kwargs)

```

Initializes the MapDispatch function.

Parameters

- **dsp** (*schedula.Dispatcher* | *schedula.utils.blue.BlueDispatcher*) – A dispatcher that identifies the base model.
- **defaults** (*dict*) – Defaults values that are recursively merged with the input provided to the *dispatching function*.
- **recursive_inputs** (*list* | *dict*) – List of data node ids that are extracted from the outputs of the *dispatching function* and then merged with the inputs of the its

successive evaluation. If a dictionary is given, this is used to rename the data node ids extracted.

- **constructor** (*function* | *class*) – It initializes the *dispatching function*.
- **constructor_kwargs** (*function* | *class*) – Extra keywords passed to the constructor function.
- **function_id** (*str*, *optional*) – Function name.
- **func_kw** (*function*, *optional*) – Extra keywords to add the *dispatching function* to execution model.
- **input_label** (*str*, *optional*) – Custom label formatter for recursive inputs.
- **output_label** (*str*, *optional*) – Custom label formatter for recursive outputs.
- **data_label** (*str*, *optional*) – Custom label formatter for recursive internal data.
- **kwargs** (*object*) – Keywords to initialize the execution model.

NoSub

class NoSub

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

Methods

`__init__`

`__init__`

`NoSub.__init__()`

`__init__()`

SubDispatch

class SubDispatch(*dsp=None*, **args*, ***kwargs*)

It dispatches a given *Dispatcher* like a function.

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

Returns

A function that executes the dispatch of the given *Dispatcher*.

Return type

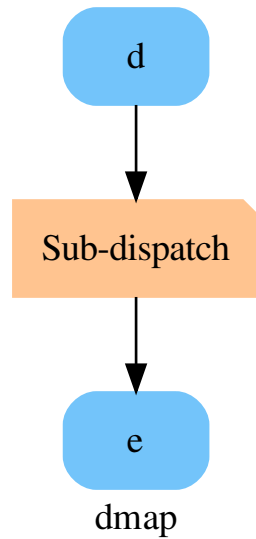
callable

See also:

`dispatch()`, `combine_dicts()`

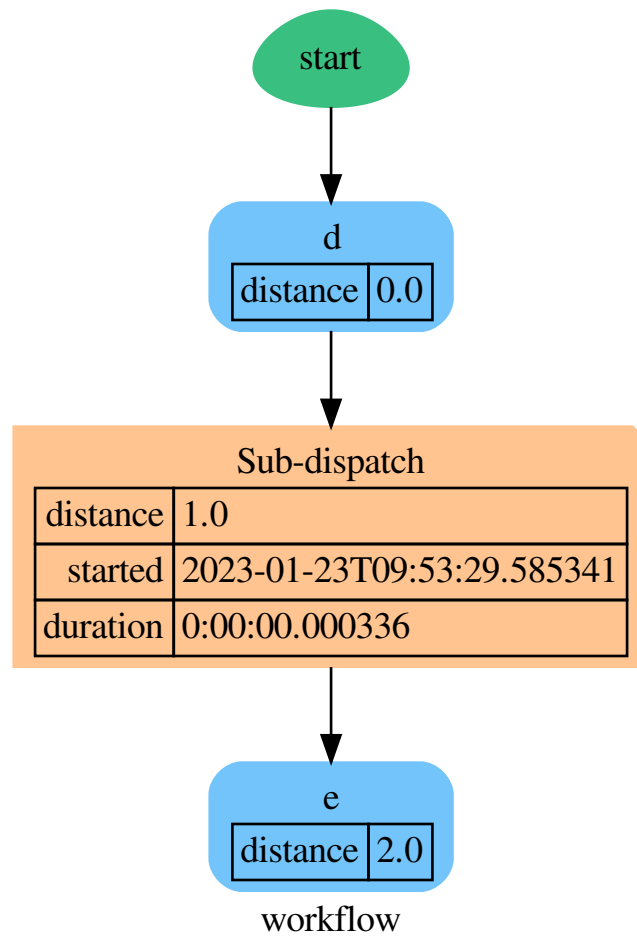
Example:


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



The Dispatcher output is:

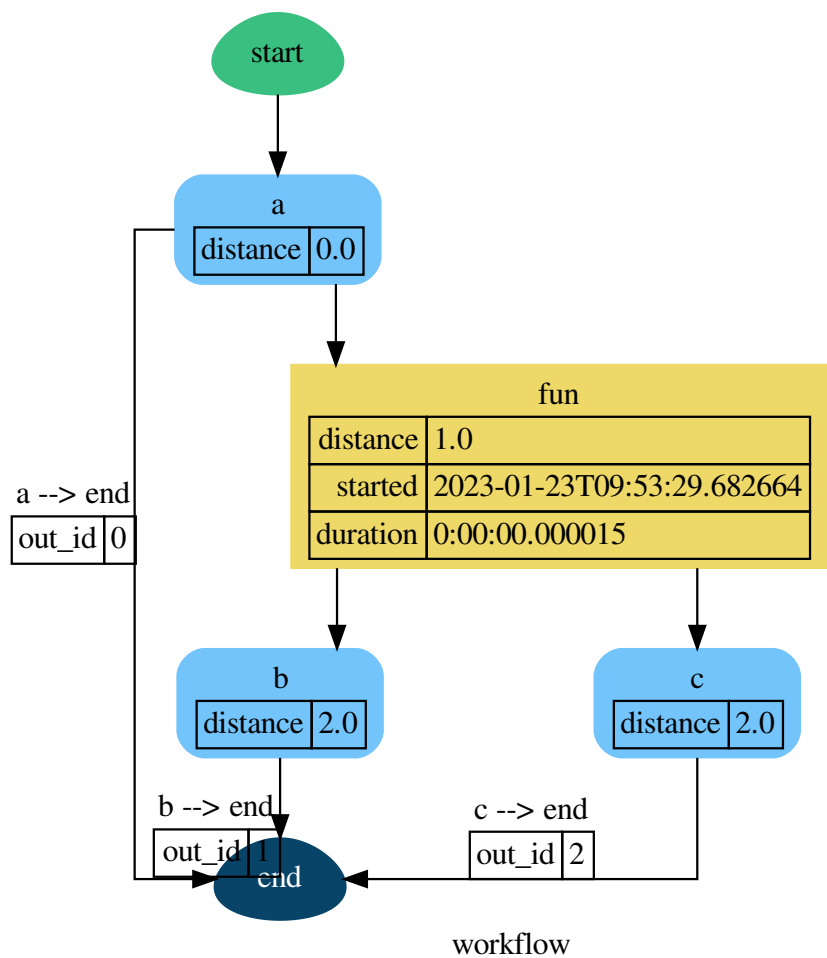
```
>>> o = dsp.dispatch(inputs={'d': {'a': 3}})
```



while, the Sub-dispatch is:

```

>>> sol = o.workflow.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>form</code> | Creates a dispatcher Form Flask app. |
| <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, inputs_dist=None, wildcard=False, no_call=False, shrink=False, rm_unused_nds=False, output_type='all', function_id=None, output_type_kw=None)`

Initializes the Sub-dispatch.

Parameters

- **dsp** (`schedula.Dispatcher` | `schedula.utils.blue.BlueDispatcher`) – A dispatcher that identifies the model adopted.
- **outputs** (`list[str]`, `iterable`) – Ending data nodes.
- **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`.
- **output_type_kw** (`dict`, `optional`) – Extra kwargs to pass to the `selector` function.
- **function_id** (`str`, `optional`) – Function name.

`blue`

`SubDispatch.blue(memo=None, depth=-1)`

Constructs a Blueprint out of the current object.

Parameters

- **memo** (`dict[T, schedula.utils.blue.Blueprint]`) – A dictionary to cache Blueprints.
- **depth** (`int`, `optional`) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

Returns

A Blueprint of the current object.

Return type

`schedula.utils.blue.Blueprint`

copy

SubDispatch.**copy**()

form

SubDispatch.**form**(depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600)

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int*, optional) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple*[*str*], optional) – Data node attributes to produce API.
- **node_function** (*tuple*[*str*], optional) – Function node attributes produce API.
- **directory** (*str*, optional) – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], optional) – A set of *Site* to maintain alive the backend server.
- **run** (*bool*, optional) – Run the backend server?
- **view** (*bool*, optional) – Open the url site with the sys default opener.
- **get_context** (*function* | *dict*, optional) – Function to pass extra data as form context.
- **get_data** (*function* | *dict*, optional) – Function to initialize the formdata.
- **edit_on_change** (*function* | *dict*, optional) – Function to initialize the formdata.
- **pre_submit** (*function* | *dict*, optional) – Function to initialize the formdata.
- **post_submit** (*function* | *dict*, optional) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int*, optional) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run* or *view* is *True*.

Return type

FormMap | Site

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.

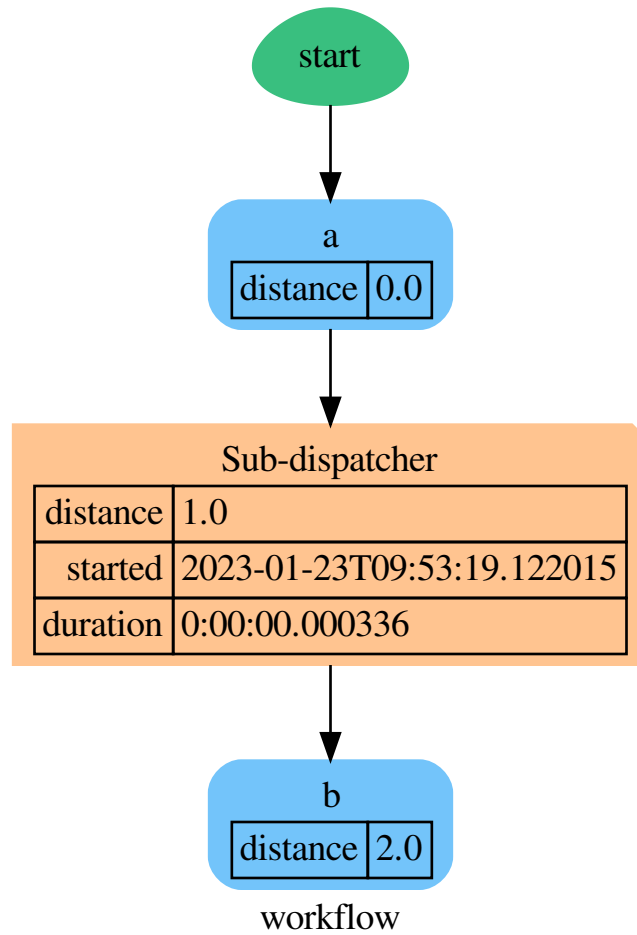
Returns

Node attributes and its real path.

Return type

(T, (*str*, ...))

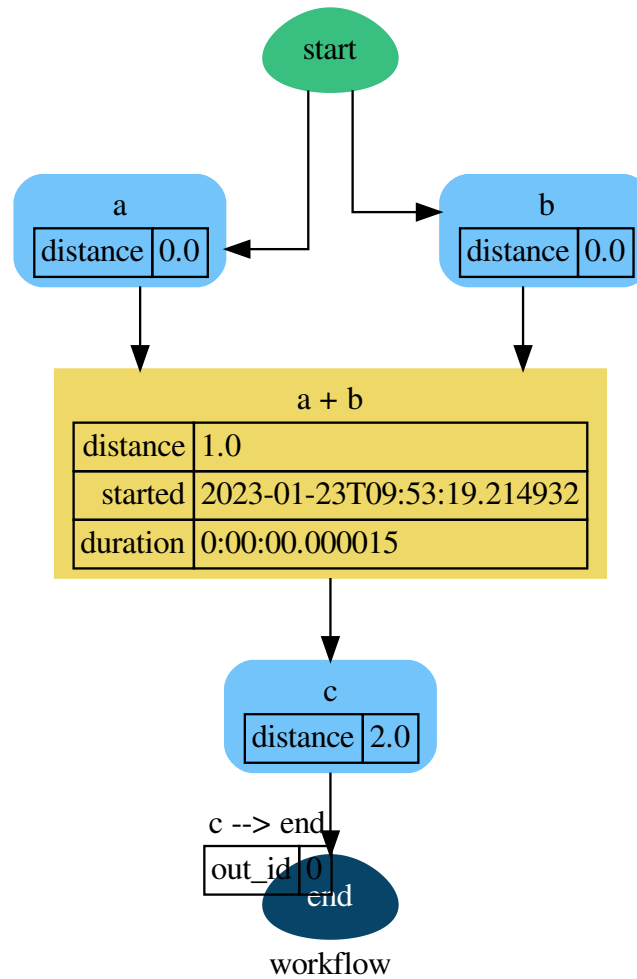
Example:



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, raw_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=True, viz=False, short_name=None, executor='async', render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

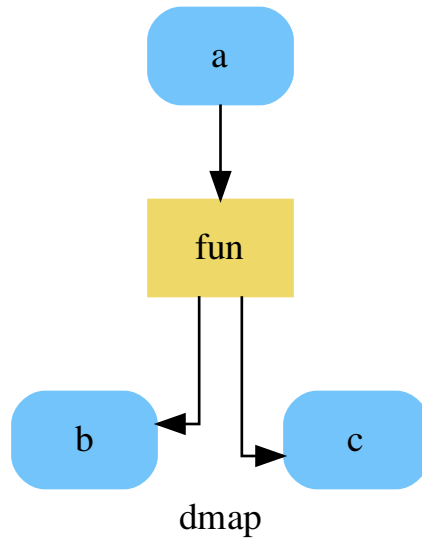
A SiteMap or a Site if .

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

`SubDispatch.web(depth=-1, node_data=None, node_function=None, directory=None, sites=None, run=True, subsite_idle_timeout=600)`

Creates a dispatcher Flask app.

Parameters

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

- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

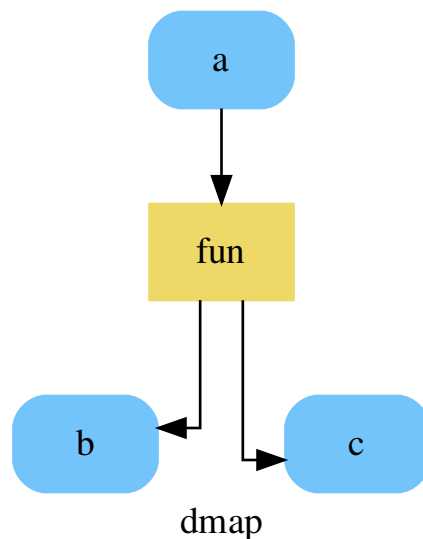
Return type

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:

```
>>> print("Starting...\n"); site = dsp.web(); 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*, *inputs_dist=None*, *wildcard=False*, *no_call=False*, *shrink=False*, *rm_unused_nds=False*, *output_type='all'*, *function_id=None*, *output_type_kw=None*)

Initializes the Sub-dispatch.

Parameters

- **dsp** (*schedula.Dispatcher* | *schedula.utils.blue.BlueDispatcher*) – A dispatcher that identifies the model adopted.
- **outputs** (*list[str]*, *iterable*) – Ending data nodes.
- **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*.
- **output_type_kw** (*dict*, *optional*) – Extra kwargs to pass to the *selector* function.
- **function_id** (*str*, *optional*) – Function name.

blue(*memo=None*, *depth=-1*)

Constructs a Blueprint out of the current object.

Parameters

- **memo** (*dict[T, schedula.utils.blue.Blueprint]*) – A dictionary to cache Blueprints.
- **depth** (*int*, *optional*) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

Returns

A Blueprint of the current object.

Return type

schedula.utils.blue.Blueprint

SubDispatchFunction

class SubDispatchFunction(*dsp=None*, **args*, ***kwargs*)

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

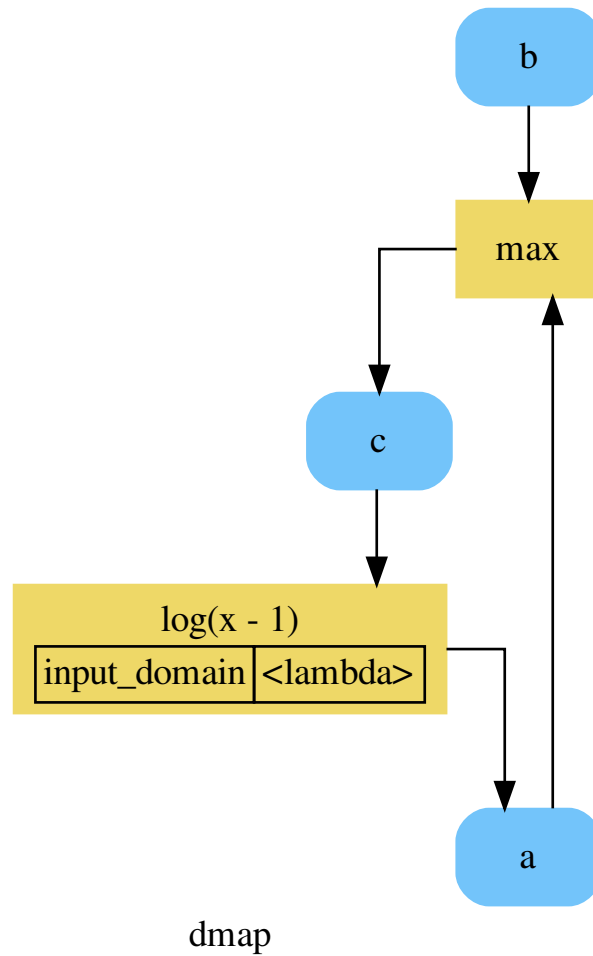
callable

See also:

dispatch(), *shrink_dsp()*

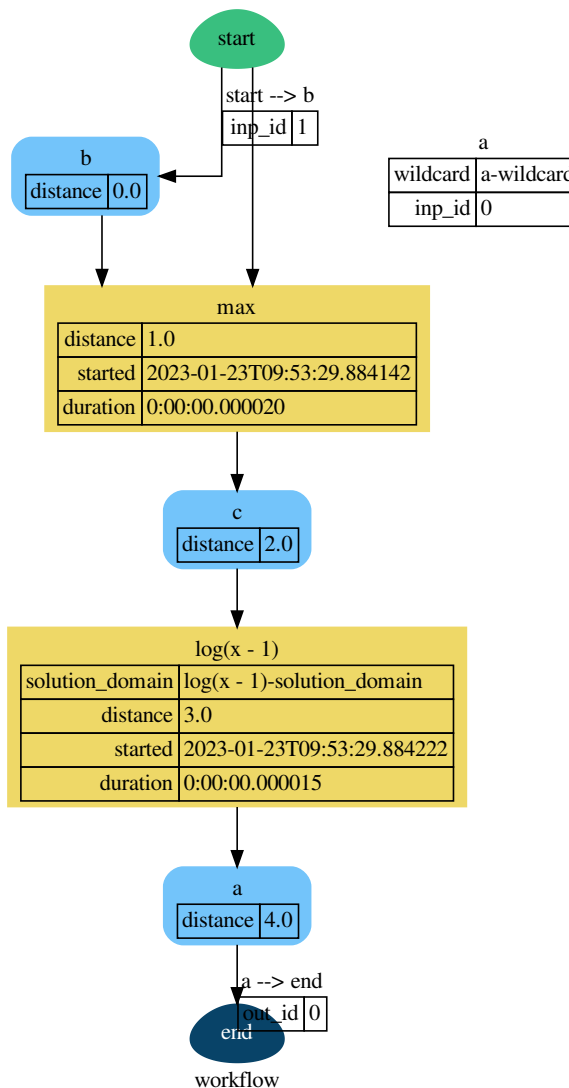
Example:

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
```



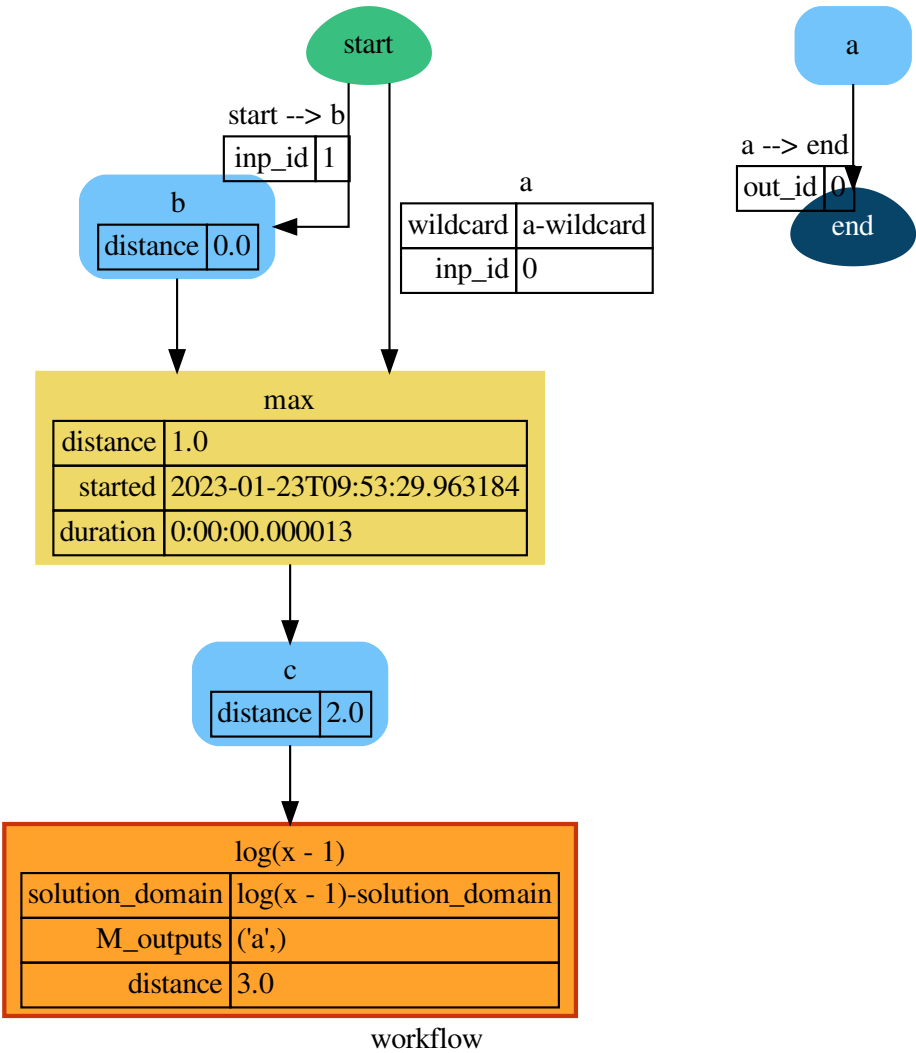
The created function raises a `ValueError` if un-valid inputs are provided:

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

(continues on next page)

(continued from previous page)

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>form</code> | Creates a dispatcher Form Flask app. |
| <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, inputs_dist=None, shrink=True, wildcard=True, output_type=None, output_type_kw=None, first_arg_as_kw=False)`

Initializes the Sub-dispatch Function.

Parameters

- **`dsp`** (`schedula.Dispatcher` | `schedula.utils.blue.BlueDispatcher`) – 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.
- **`inputs_dist`** (`dict[str, int | float]`, *optional*) – Initial distances of input data nodes.
- **`shrink`** (`bool`, *optional*) – If True the dispatcher is shrink before the dispatch.
- **`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.
- **`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*.
- **`output_type_kw`** (`bool`) – Extra kwargs to pass to the *selector* function.
- **`first_arg_as_kw`** – Uses the first argument of the `__call__` method as *kwargs*.

blue

SubDispatchFunction.**blue**(memo=None, depth=-1)

Constructs a Blueprint out of the current object.

Parameters

- **memo** (*dict*[*T*, [schedula.utils.blue.Blueprint](#)]) – A dictionary to cache Blueprints.
- **depth** (*int*, *optional*) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

Returns

A Blueprint of the current object.

Return type

[schedula.utils.blue.Blueprint](#)

copy

SubDispatchFunction.**copy**()

form

SubDispatchFunction.**form**(depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600)

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int*, *optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple*[*str*], *optional*) – Data node attributes to produce API.
- **node_function** (*tuple*[*str*], *optional*) – Function node attributes produce API.
- **directory** (*str*, *optional*) – Where is the generated Flask app root located?
- **sites** (*set*[[Site](#)], *optional*) – A set of [Site](#) to maintain alive the backend server.
- **run** (*bool*, *optional*) – Run the backend server?
- **view** (*bool*, *optional*) – Open the url site with the sys default opener.
- **get_context** (*function* | *dict*, *optional*) – Function to pass extra data as form context.
- **get_data** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **edit_on_change** (*function* | *dict*, *optional*) – Function to initialize the formdata.

- **pre_submit** (*function* | *dict*, *optional*) – Function to initialize the form-data.
- **post_submit** (*function* | *dict*, *optional*) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run* or *view* is *True*.

Return type

FormMap | Site

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.

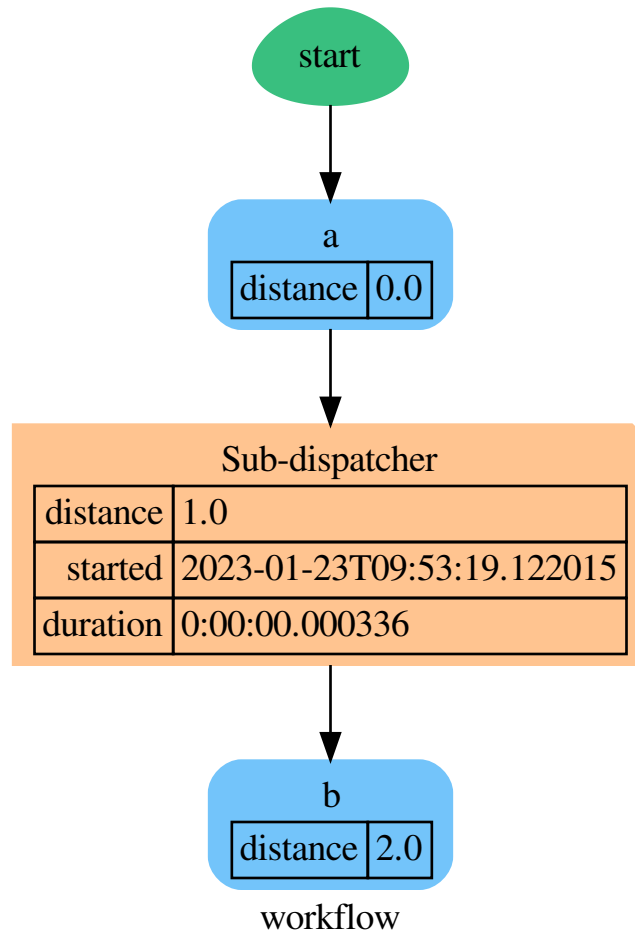
Returns

Node attributes and its real path.

Return type

(T, (str, ...))

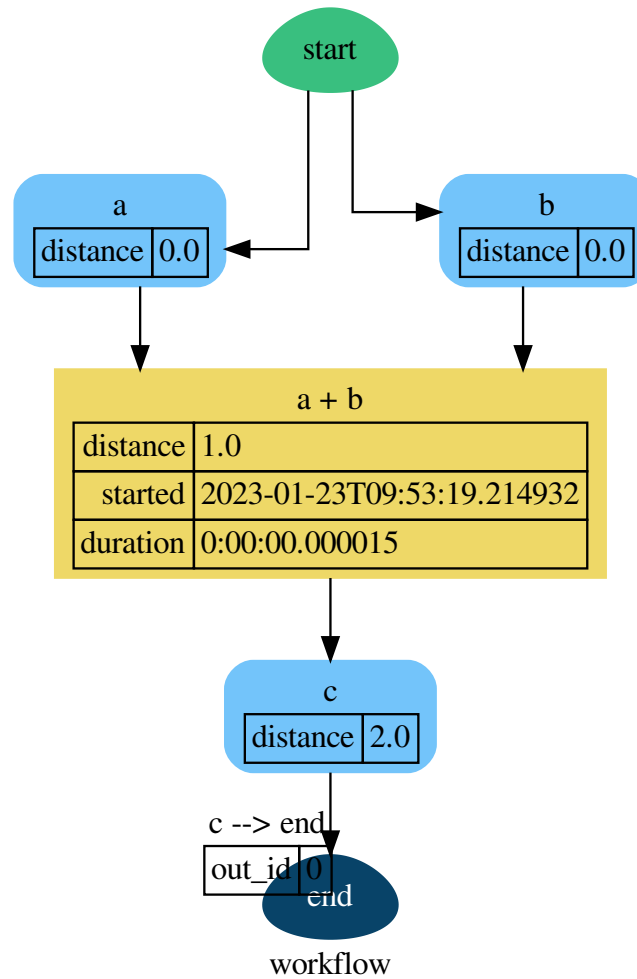
Example:



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, raw_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=True, viz=False, short_name=None, executor='async', render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

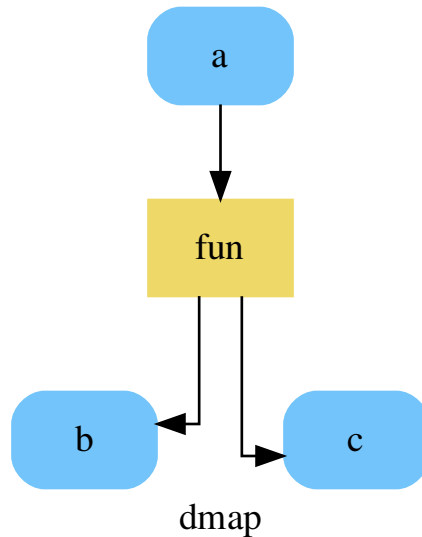
A SiteMap or a Site if .

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, subsite_idle_timeout=600)

Creates a dispatcher Flask app.

Parameters

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

- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

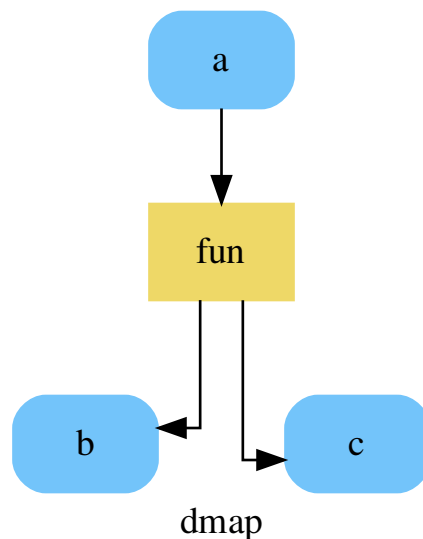
Return type

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:

```
>>> print("Starting...\n"); site = dsp.web(); 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, *inputs_dist*=None, *shrink*=True, *wildcard*=True, *output_type*=None, *output_type_kw*=None, *first_arg_as_kw*=False)

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher* | *schedula.utils.blue.BlueDispatcher*) – 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.
- **inputs_dist** (*dict[str, int | float]*, *optional*) – Initial distances of input data nodes.
- **shrink** (*bool*, *optional*) – If True the dispatcher is shrink before the dispatch.
- **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.
- **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*.
- **output_type_kw** (*bool*) – Extra kwargs to pass to the *selector* function.
- **first_arg_as_kw** – Uses the first argument of the *__call__* method as *kwargs*.

Attributes

var_keyword

var_keyword

SubDispatchFunction.**var_keyword** = 'kw'

SubDispatchPipe

class SubDispatchPipe(*dsp=None*, *args, **kwargs)

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

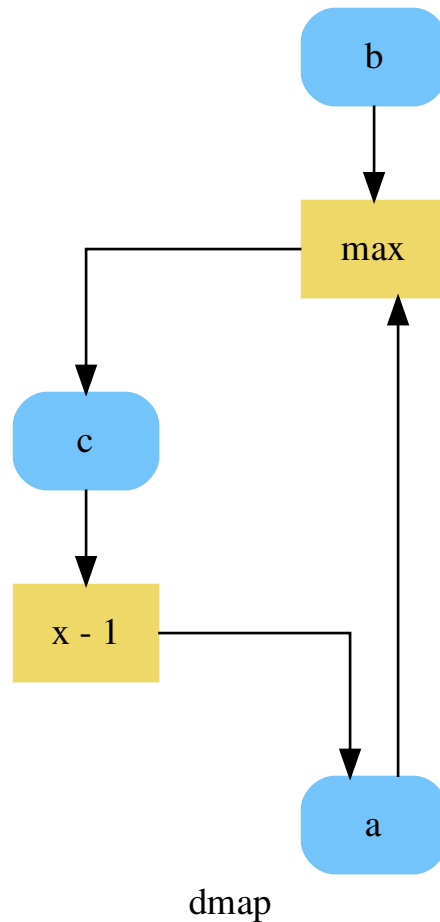
callable

See also:

dispatch(), *shrink_dsp()*

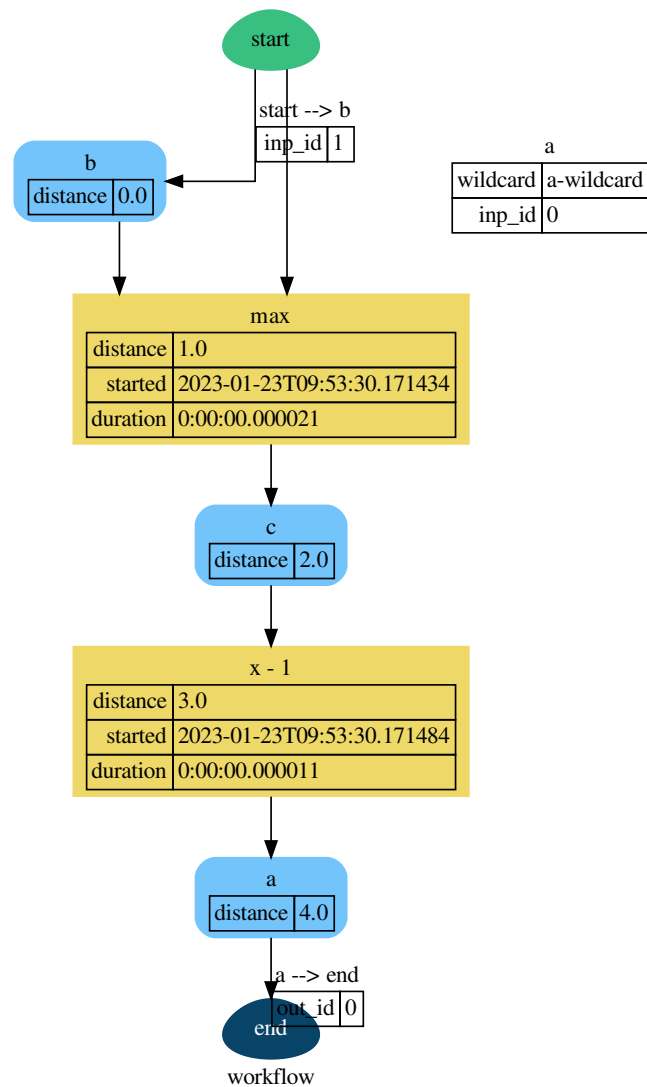
Example:

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



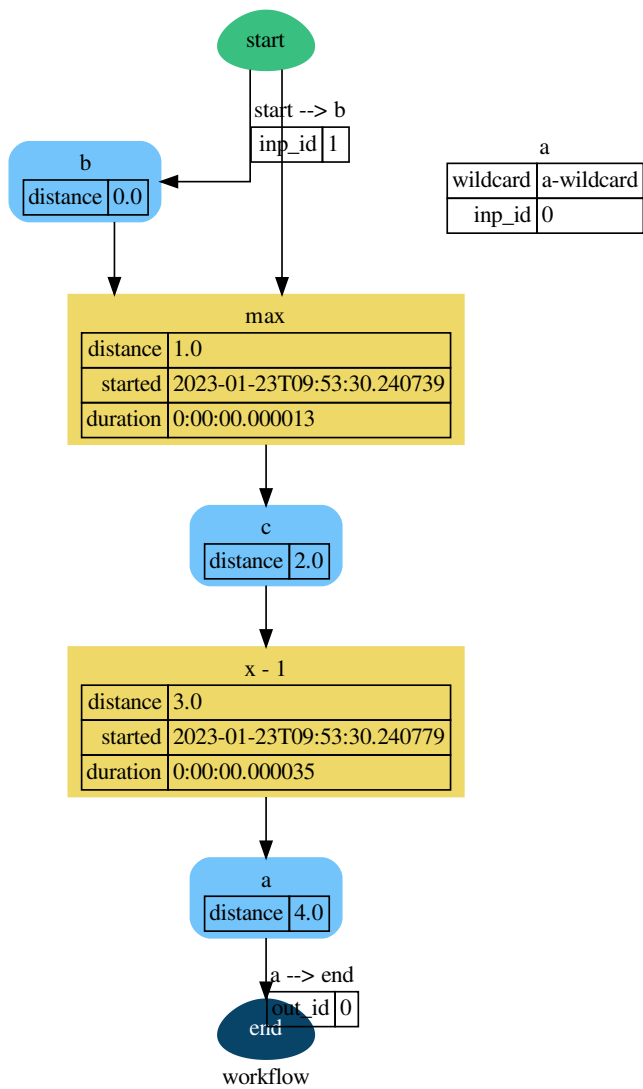
Extract a static function node, i.e. the inputs *a* and *b* and the output *a* are fixed:

```
>>> 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>form</code> | Creates a dispatcher Form Flask app. |
| <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, inputs_dist=None, no_domain=True, wildcard=True, shrink=True, output_type=None, output_type_kw=None, first_arg_as_kw=False)`

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (`schedula.Dispatcher` | `schedula.utils.blue.BlueDispatcher`) – 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.
- **inputs_dist** (`dict[str, int | float]`, `optional`) – Initial distances of input data nodes.
- **no_domain** (`bool`, `optional`) – Skip the domain check.
- **shrink** (`bool`, `optional`) – If True the dispatcher is shrink before the dispatch.
- **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.
- **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`.
- **output_type_kw** (`bool`) – Extra kwargs to pass to the `selector` function.
- **first_arg_as_kw** – Converts first argument of the `__call__` method as `kwargs`.

`blue`

`SubDispatchPipe.blue(memo=None, depth=-1)`

Constructs a Blueprint out of the current object.

Parameters

- **memo** (`dict[T, schedula.utils.blue.Blueprint]`) – A dictionary to cache Blueprints.
- **depth** (`int`, `optional`) – Depth of sub-dispatch blue. If negative all levels are bluprinted.

Returns

A Blueprint of the current object.

Return type

`schedula.utils.blue.Blueprint`

copy

SubDispatchPipe.**copy**()

form

SubDispatchPipe.**form**(*depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600*)

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int, optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str], optional*) – Data node attributes to produce API.
- **node_function** (*tuple[str], optional*) – Function node attributes produce API.
- **directory** (*str, optional*) – Where is the generated Flask app root located?
- **sites** (*set[Site], optional*) – A set of [Site](#) to maintain alive the backend server.
- **run** (*bool, optional*) – Run the backend server?
- **view** (*bool, optional*) – Open the url site with the sys default opener.
- **get_context** (*function | dict, optional*) – Function to pass extra data as form context.
- **get_data** (*function | dict, optional*) – Function to initialize the formdata.
- **edit_on_change** (*function | dict, optional*) – Function to initialize the formdata.
- **pre_submit** (*function | dict, optional*) – Function to initialize the formdata.
- **post_submit** (*function | dict, optional*) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int, optional*) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run* or *view* is *True*.

Return type

[FormMap](#) | [Site](#)

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.

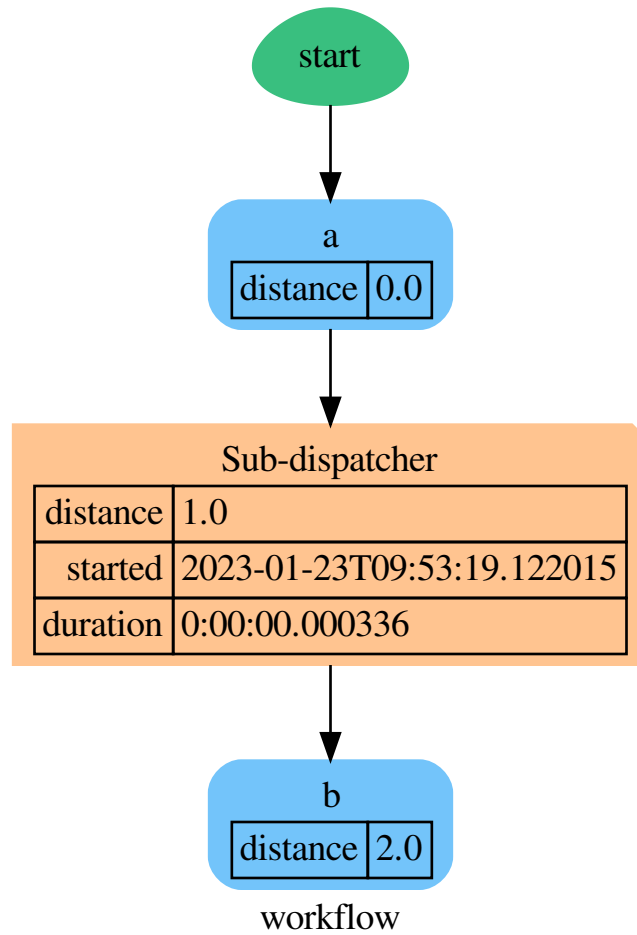
Returns

Node attributes and its real path.

Return type

(T, (*str*, ...))

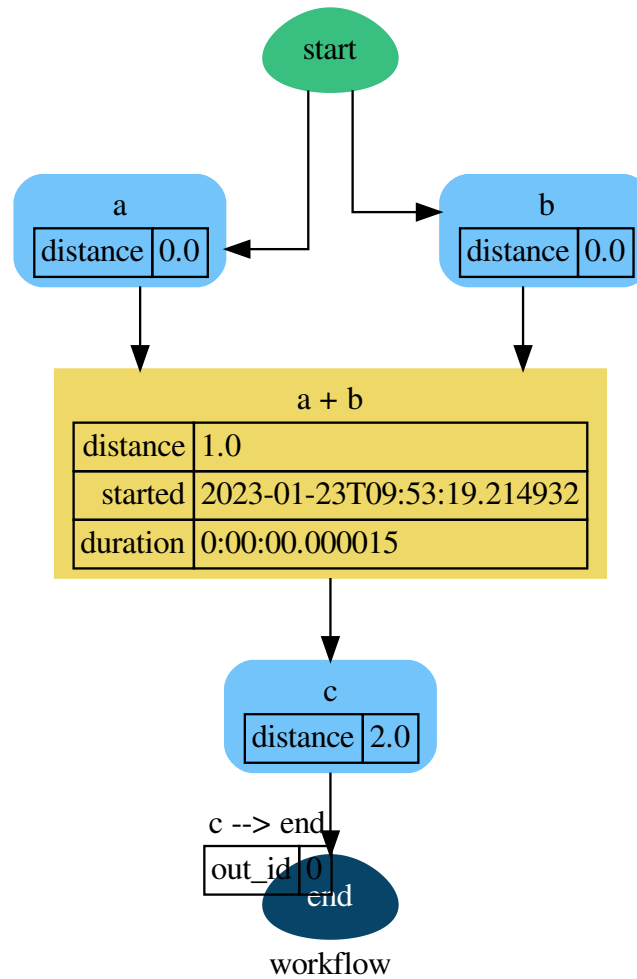
Example:



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, raw_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=True, viz=False, short_name=None, executor='async', render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set*[*Site*], *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

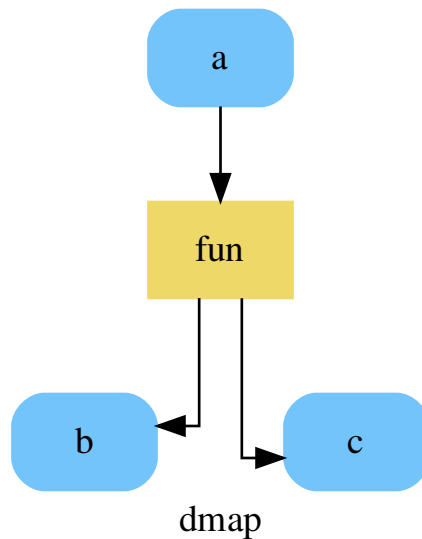
A SiteMap or a Site if .

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, subsite_idle_timeout=600)`

Creates a dispatcher Flask app.

Parameters

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

- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

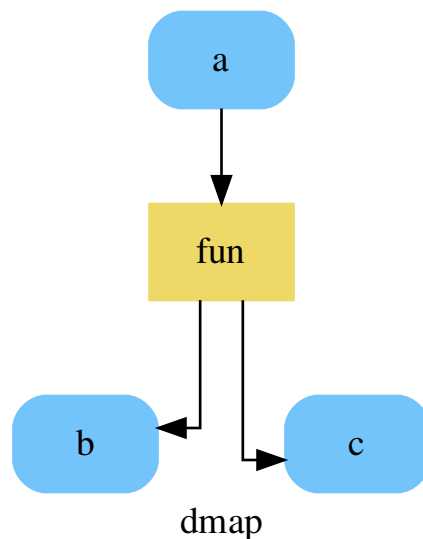
Return type

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:

```
>>> print("Starting...\n"); site = dsp.web(); 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, *inputs_dist*=None, *no_domain*=True, *wildcard*=True, *shrink*=True, *output_type*=None, *output_type_kw*=None, *first_arg_as_kw*=False)

Initializes the Sub-dispatch Function.

Parameters

- **dsp** (*schedula.Dispatcher* | *schedula.utils.blue.BlueDispatcher*) – 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.
- **inputs_dist** (*dict[str, int | float]*, *optional*) – Initial distances of input data nodes.
- **no_domain** (*bool*, *optional*) – Skip the domain check.
- **shrink** (*bool*, *optional*) – If True the dispatcher is shrink before the dispatch.
- **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.
- **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*.
- **output_type_kw** (*bool*) – Extra kwargs to pass to the *selector* function.
- **first_arg_as_kw** – Converts first argument of the *__call__* method as *kwargs*.

Attributes

var_keyword

var_keyword

SubDispatchPipe.**var_keyword** = None

var_keyword = None

add_args

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

Methods

`__init__`

`__init__`

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

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

inf

class `inf`(*inf: float = 0*, *num: float = 0*)

Class to model infinite numbers for workflow distance.

Methods

`__init__`

`format`

format

static `inf.format(val)`

`__init__(inf: float = 0, num: float = 0) → None`

Attributes

`inf`

`num`

inf

`inf.inf: float = 0`

num

`inf.num: float = 0`

run_model

class `run_model(func, *args, _init=None, **kwargs)`

It is an utility function to execute dynamically generated function/models and - if Dispatcher based - add their workflows to the parent solution.

Returns

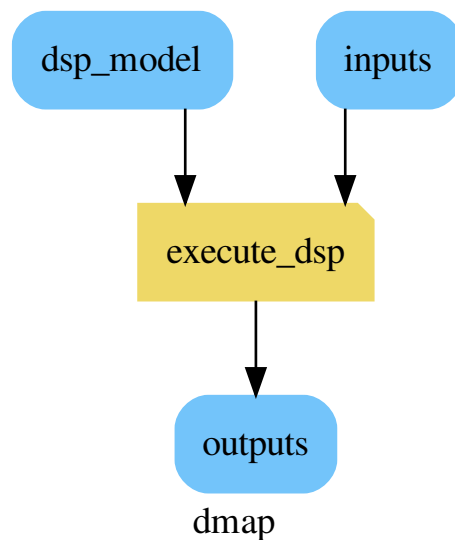
A function that executes the dispatch of the given *dsp*.

Return type

callable

Example:

Follows a simple example on how to use the `run_model()`:



Moreover, it can be used also with all `SubDispatcher()` like objects:

```

>>> sub_dsp = SubDispatch(dsp_model, outputs=['c'], output_type='list')
>>> sol = dsp({'dsp_model': sub_dsp, 'inputs': {'b': 1, 'a': 2}})
>>> sol['outputs']
[2]
  
```

(continues on next page)

(continued from previous page)

```
>>> sol.workflow.nodes['execute_dsp']['solution']
Solution([('a', 2), ('b', 1), ('c', 2)])
```

Methods

`__init__`

`__init__`

`run_model.__init__(func, *args, _init=None, **kwargs)`

`__init__(func, *args, _init=None, **kwargs)`

7.2.9 exc

Defines the dispatcher exception.

Exceptions

DispatcherAbort

DispatcherError

ExecutorShutdown

SkipNode

WebResponse

DispatcherAbort

exception DispatcherAbort

DispatcherError

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

ExecutorShutdown

exception `ExecutorShutdown`

SkipNode

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

WebResponse

exception `WebResponse(response)`

7.2.10 form

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

Classes

FormMap

FormMap

class `FormMap`

Methods

| | |
|---------------------------------|--|
| <code>__init__</code> | |
| <code>add_items</code> | |
| <code>app</code> | |
| <code>basic_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>generate_csrf</code> | |
| <code>get</code> | Return the value for key if key is in the dictionary, else default. |
| <code>get_directory</code> | |
| <code>get_dsp_from</code> | |
| <code>get_sol_from</code> | |
| <code>init_debug_subsite</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>render_form</code> | |
| <code>rules</code> | |
| <code>send_static_file</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 E: D[k] = E[k] |
| <code>validate_csrf</code> | |
| <code>values</code> | |

`__init__`

`FormMap.__init__()`

`add_items`

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

`app`

`FormMap.app(root_path=None, depth=1, mute=False, blueprint_name=None, **kwargs)`

`basic_app`

`FormMap.basic_app(root_path, mute=True, blueprint_name=None, **kwargs)`

`clear`

`FormMap.clear()` → None. Remove all items from od.

`copy`

`FormMap.copy()` → a shallow copy of od

`fromkeys`

`FormMap.fromkeys(value=None)`

Create a new ordered dictionary with keys from iterable and values set to value.

`generate_csrf`

`FormMap.generate_csrf()`

`get`

`FormMap.get(key, default=None, /)`

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

get_directory

`FormMap.get_directory(directory)`

get_dsp_from

`static FormMap.get_dsp_from(item)`

get_sol_from

`static FormMap.get_sol_from(item)`

init_debug_subsite

`FormMap.init_debug_subsite(func)`

items

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

keys

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

move_to_end

`FormMap.move_to_end(key, last=True)`

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

Raise `KeyError` if the element does not exist.

pop

`FormMap.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

`FormMap.popitem(last=True)`

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

`FormMap.render(*args, **kwargs)`

render_form

`FormMap.render_form(form='index')`

rules

`FormMap.rules(depth=-1, index=True, viz_js=False)`

send_static_file

`static FormMap.send_static_file(filename)`

setdefault

`FormMap.setdefault(key, default=None)`

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

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

site_index

`FormMap.site_index(**kwargs)`

update

`FormMap.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]

validate_csrf

`FormMap.validate_csrf()`

values

`FormMap.values()` → an object providing a view on D's values

`__init__()`

Attributes

`blueprint_name`

`csrf_defaults`

`csrf_required`

`directory`

`idle_timeout`

`include_folders_as_filenames`

`methods`

`options`

`short_name`

`subsite_methods`

blueprint_name

```
FormMap.blueprint_name = None
```

csrf_defaults

```
FormMap.csrf_defaults = {'CSRF_ENABLED': True, 'CSRF_FIELD_NAME': 'csrf_token',  
'CSRF_HEADERS': {'X-CSRF-Token', 'X-CSRFToken'}, 'CSRF_METHODS': {'DELETE', 'PATCH',  
'POST', 'PUT'}, 'CSRF_SECRET_KEY': <function FormMap.<lambda>>, 'CSRF_SSL_STRICT':  
True, 'CSRF_TIME_LIMIT': 3600}
```

csrf_required

```
FormMap.csrf_required = {'CSRF_FIELD_NAME': 'A field name is required to use CSRF.',  
'CSRF_HEADERS': 'A valid headers is required to use CSRF.', 'CSRF_METHODS': 'A valid  
request methods is required to use CSRF.', 'CSRF_SECRET_KEY': 'A secret key is  
required to use CSRF.'}
```

directory

```
FormMap.directory = None
```

idle_timeout

```
FormMap.idle_timeout = 600
```

include_folders_as_filenames

```
FormMap.include_folders_as_filenames = False
```

methods

```
FormMap.methods = ['POST']
```

options

```
FormMap.options = {'digraph', 'edge_data', 'max_lines', 'max_width', 'node_data',  
'node_function', 'node_styles'}
```

short_name

`FormMap.short_name = None`

subsite_methods

`FormMap.subsite_methods = ['GET', 'POST']`

7.2.11 gen

It contains classes and functions of general utility.

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

Functions

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

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.

Classes

| | |
|--------------|---|
| <i>Token</i> | 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
```

(continues on next page)

(continued from previous page)

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

Methods

| | |
|---------------------------|--|
| <code>__init__</code> | |
| <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. |

continues on next page

Table 1 – continued from previous page

| | |
|-------------------------|---|
| <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. |
| <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 old replaced by new. |
| <code>rfind</code> | Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. |
| <code>rindex</code> | Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. |
| <code>rjust</code> | Return a right-justified string of length width. |
| <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 sep 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 sep as the delimiter string. |
| <code>splitlines</code> | Return a list of the lines in the string, breaking at line boundaries. |
| <code>startswith</code> | Return True if S starts with the specified prefix, False otherwise. |
| <code>strip</code> | Return a copy of the string with leading and trailing whitespace removed. |
| <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)`

`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(width, fillchar=' ', /)`

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(encoding='utf-8', errors='strict')`

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**(*tabsize*=8)

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(*iterable*, /)

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(*width*, *fillchar*=' ', /)

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(*chars*=None, /)

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(*x*, *y*=None, *z*=None, /)

Return a translation table usable for `str.translate()`.

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

partition

Token.partition(*sep*, /)

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(old, new, count=-1, /)`

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(width, fillchar=' ', /)`

Return a right-justified string of length `width`.

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

rpartition

`Token.rpartition(sep, /)`

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(sep=None, maxsplit=-1)`

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(chars=None, /)`

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(sep=None, maxsplit=-1)`

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(keepends=False)`

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(chars=None, /)`

Return a copy of the string with leading and trailing whitespace removed.

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(table, /)`

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(width, /)`

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

The string is never truncated.

`__init__(*args)`

7.2.12 graph

It contains the *DiGraph* class.

Classes

DiGraph

DiGraph

class **DiGraph**(*nodes=None, adj=None*)

Methods

`__init__`

`add_edge`

`add_edge_fw`

`add_edges_from`

`add_node`

`add_nodes_from`

`copy`

`has_edge`

`remove_edge`

`remove_edges_from`

`remove_node`

`remove_nodes_from`

`subgraph`

`__init__`

`DiGraph.__init__(nodes=None, adj=None)`

`add_edge`

`DiGraph.add_edge(u, v, **attr)`

`add_edge_fw`

`DiGraph.add_edge_fw(u, v, **attr)`

`add_edges_from`

`DiGraph.add_edges_from(ebunch_to_add)`

`add_node`

`DiGraph.add_node(n, **attr)`

`add_nodes_from`

`DiGraph.add_nodes_from(nodes_for_adding)`

`copy`

`DiGraph.copy()`

`has_edge`

`DiGraph.has_edge(u, v)`

`remove_edge`

`DiGraph.remove_edge(u, v)`

remove_edges_from

`DiGraph.remove_edges_from(ebunch)`

remove_node

`DiGraph.remove_node(n)`

remove_nodes_from

`DiGraph.remove_nodes_from(nodes)`

subgraph

`DiGraph.subgraph(nodes)`

`__init__(nodes=None, adj=None)`

Attributes

`nodes`

`pred`

`succ`

nodes

`DiGraph.nodes`

pred

`DiGraph.pred`

succ

`DiGraph.succ`

7.2.13 imp

Fixes ImportError for MicroPython.

7.2.14 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. |
| <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)
>>> 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)
```

7.2.15 sol

It provides a solution class for dispatch result.

Classes

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

Solution

class Solution(*args, **kwargs)

Solution class for dispatch result.

Methods

| | |
|--|--|
| <code>__init__</code> | |
| <code>check_targets</code> | Terminates ArciDispatch algorithm when all targets have been visited. |
| <code>check_wait_in</code> | Stops the search of the investigated node of the ArciDispatch algorithm, until all inputs are satisfied. |
| <code>clear</code> | |
| <code>copy</code> | |
| <code>form</code> | Creates a dispatcher Form Flask app. |
| <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, inputs_dist=None, no_call=False, rm_unused_nds=False, wait_in=None, no_domain=False, _empty=False, index=(-1,), full_name=(), verbose=False, excluded_defaults=())`

`check_targets`

`Solution.check_targets(node_id)`

Terminates ArciDispatch algorithm when all targets have been visited.

Parameters

node_id (*str*) – Data or function node id.

Returns

True if all targets have been visited, otherwise False.

Return type

bool

`check_wait_in`

`Solution.check_wait_in(wait_in, n_id)`

Stops the search of the investigated node of the ArciDispatch algorithm, until all inputs are satisfied.

Parameters

- **wait_in** (*bool*) – If True the node is waiting input estimations.
- **n_id** (*str*) – Data or function node id.

Returns

True if all node inputs are satisfied, otherwise False.

Return type

bool

`clear`

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

`copy`

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

form

`Solution.form(depth=1, node_data=None, node_function=None, directory=None, sites=None, run=True, view=True, get_context=None, get_data=None, edit_on_change=None, pre_submit=None, post_submit=None, subsite_idle_timeout=600)`

Creates a dispatcher Form Flask app.

Parameters

- **depth** (*int*, optional) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str]*, optional) – Data node attributes to produce API.
- **node_function** (*tuple[str]*, optional) – Function node attributes produce API.
- **directory** (*str*, optional) – Where is the generated Flask app root located?
- **sites** (*set[Site]*, optional) – A set of [Site](#) to maintain alive the backend server.
- **run** (*bool*, optional) – Run the backend server?
- **view** (*bool*, optional) – Open the url site with the sys default opener.
- **get_context** (*function* | *dict*, optional) – Function to pass extra data as form context.
- **get_data** (*function* | *dict*, optional) – Function to initialize the formdata.
- **edit_on_change** (*function* | *dict*, optional) – Function to initialize the formdata.
- **pre_submit** (*function* | *dict*, optional) – Function to initialize the formdata.
- **post_submit** (*function* | *dict*, optional) – Function to initialize the formdata.
- **subsite_idle_timeout** (*int*, optional) – Idle timeout of a debug subsite in seconds.

Returns

A FormMap or a Site if *sites* is *None* and *run* or *view* is *True*.

Return type

[FormMap](#) | [Site](#)

fromkeys

`Solution.fromkeys(value=None)`

Create a new ordered dictionary with keys from iterable and values set to value.

get

`Solution.get(key, default=None, /)`

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.

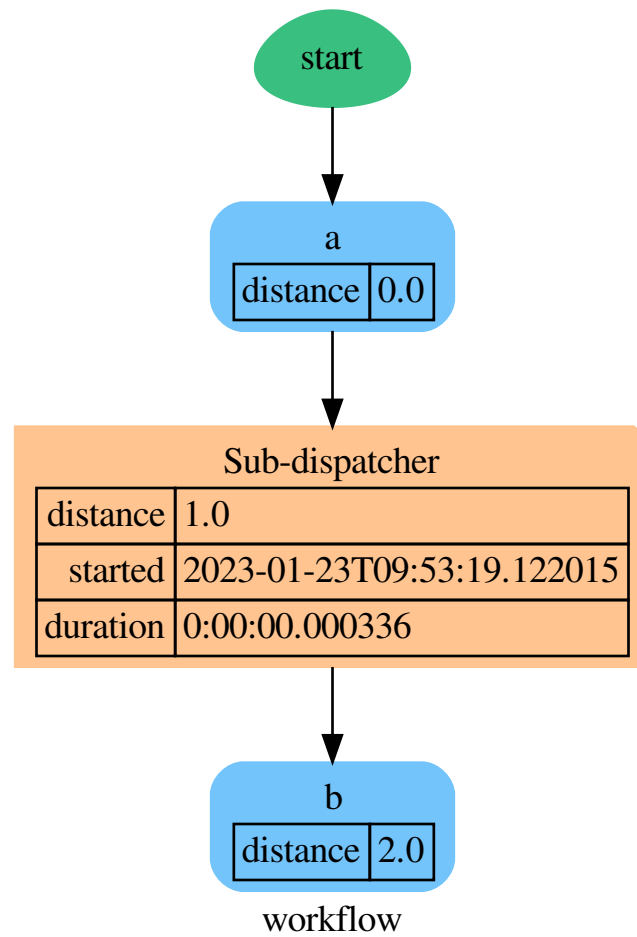
Returns

Node attributes and its real path.

Return type

(T, (*str*, ...))

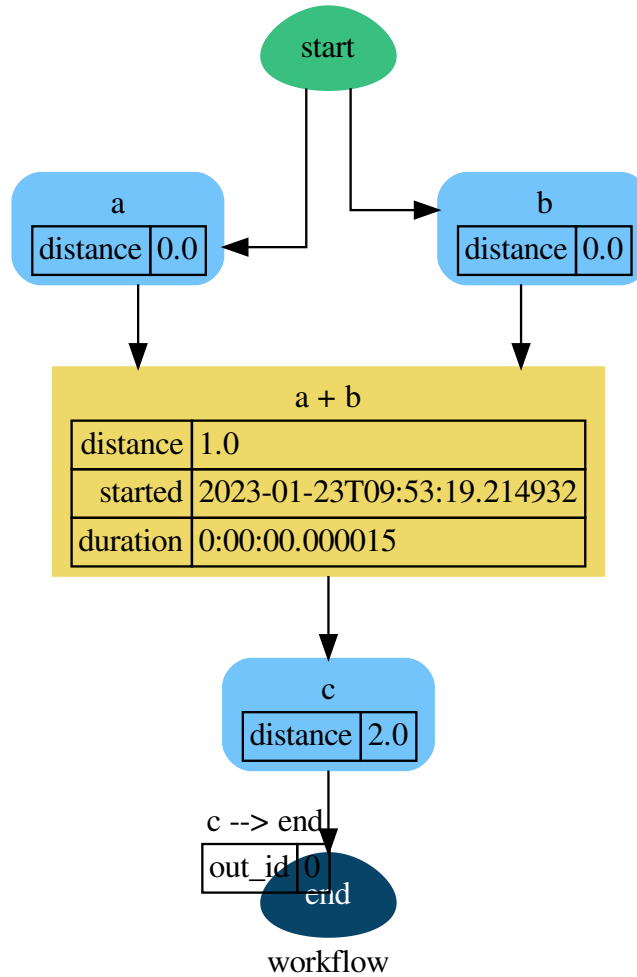
Example:



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(key, last=True)`

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, raw_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=True, viz=False, short_name=None, executor='async', render=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.
- **raw_body** (*list*, *optional*) – List of command to add to the graph body.
- **directory** – Where is the generated Flask app root located?
- **sites** (*set[Site]*, *optional*) – A set of *Site* to maintain alive the backend server.
- **index** (*bool*, *optional*) – Add the site index as first page?
- **max_lines** (*int*, *optional*) – Maximum number of lines for rendering node attributes.
- **max_width** (*int*, *optional*) – Maximum number of characters in a line to render node attributes.
- **view** – Open the main page of the site?
- **render** (*bool*, *optional*) – Render all pages statically?
- **viz** (*bool*, *optional*) – Use viz.js as back-end?
- **short_name** (*int*, *optional*) – Maximum length of the filename, if set name is hashed and reduced.
- **executor** (*str*, *optional*) – Pool executor to render object.

Returns

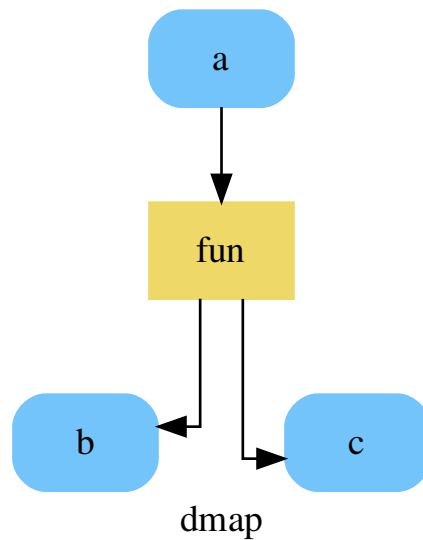
A SiteMap or a Site if .

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(last=True)`

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(key, default=None)`

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, subsite_idle_timeout=600)`

Creates a dispatcher Flask app.

Parameters

- **depth** (*int*, *optional*) – Depth of sub-dispatch API. If negative all levels are configured.
- **node_data** (*tuple[str]*, *optional*) – Data node attributes to produce API.
- **node_function** (*tuple[str]*, *optional*) – Function node attributes produce API.
- **directory** (*str*, *optional*) – Where is the generated Flask app root located?
- **sites** (*set[Site]*, *optional*) – A set of [Site](#) to maintain alive the backend server.
- **run** (*bool*, *optional*) – Run the backend server?
- **subsite_idle_timeout** (*int*, *optional*) – Idle timeout of a debug subsite in seconds.

Returns

A WebMap.

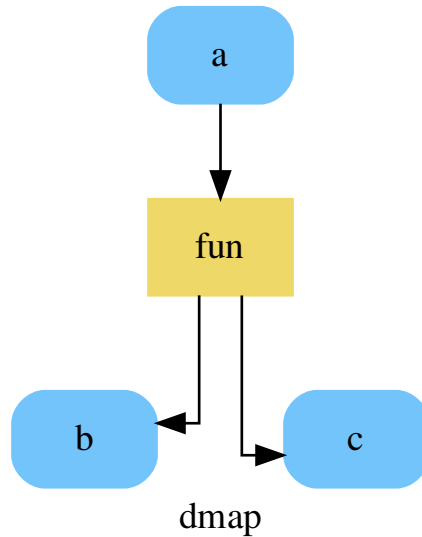
Return type

[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:

```
>>> print("Starting...\n"); site = dsp.web(); 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, inputs_dist=None, no_call=False,
         rm_unused_nds=False, wait_in=None, no_domain=False, _empty=False, index=(-1,),
         full_name=(), verbose=False, excluded_defaults=())
```

```
check_wait_in(wait_in, n_id)
```

Stops the search of the investigated node of the ArciDispatch algorithm, until all inputs are satisfied.

Parameters

- **wait_in** (*bool*) – If True the node is waiting input estimations.
- **n_id** (*str*) – Data or function node id.

Returns

True if all node inputs are satisfied, otherwise False.

Return type

bool

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

property pipe

Returns the full pipe of a dispatch run.

check_targets(*node_id*)

Terminates ArciDispatch algorithm when all targets have been visited.

Parameters

node_id (*str*) – Data or function node id.

Returns

True if all targets have been visited, otherwise False.

Return type

bool

7.2.16 utl

Functions

| | |
|---------------------------|---|
| <i>dict_diff</i> | |
| <i>get_unused_node_id</i> | Finds an unused node id in <i>graph</i> . |
| <i>select_diff</i> | |

dict_diff

dict_diff(*adict*: *dict*, *excluded*: *set*) → *dict*

get_unused_node_id

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

Finds an unused node id in *graph*.

Parameters

- **graph** (*schedula.utils.graph.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

select_diff

select_diff(*adict*: *dict*, *excluded*: *set*, *key*: *str*) → *dict*

7.2.17 web

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

Classes

| |
|----------------------|
| <i>FolderNodeWeb</i> |
| <i>WebFolder</i> |
| <i>WebMap</i> |
| <i>WebNode</i> |

FolderNodeWeb

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

Methods

`__init__`

`dot`

`href`

`items`

`parent_ref`

`render_funcs`

`render_size`

`style`

`yield_attr`

`__init__`

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

dot

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

href

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

items

FolderNodeWeb.**items**()

parent_ref

FolderNodeWeb.**parent_ref**(*context*, *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*)

Attributes

counter

edge_data

max_lines

max_width

node_data

node_function

node_map

node_styles

re_node

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',), '+': ('dot', 'table'), '-': (), '.': ('dot',), '?': ()}

node_styles

```

FolderNodeWeb.node_styles = {'error': {empty: {'fillcolor': '#FFFFFF', 'label':
'empty', 'shape': 'egg'}, end: {'color': '#084368', 'fillcolor': '#084368',
'fontcolor': '#FFFFFF', 'label': 'end', 'ordering': 'in', 'shape': 'egg'}, none:
{'data': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'penwidth': 2, 'shape':
'box', 'style': 'rounded, filled'}, 'dispatcher': {'color': '#5E1F00',
'fillcolor': '#FF3536', 'penwidth': 2, 'shape': 'note', 'style': 'filled'},
'dispatchpipe': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in',
'penwidth': 2, 'shape': 'note', 'style': 'filled'}, 'edge': {None: None},
'function': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in',
'penwidth': 2, 'shape': 'box'}, 'function-dispatcher': {'color': '#5E1F00',
'fillcolor': '#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape': 'note'},
'mapdispatch': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in',
'penwidth': 2, 'shape': 'note', 'style': 'filled'}, 'run_model': {'color':
'#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape':
'note'}, 'subdispatch': {'color': '#5E1F00', 'fillcolor': '#FF3536', 'penwidth':
2, 'shape': 'note', 'style': 'filled'}, 'subdispatchfunction': {'color':
'#5E1F00', 'fillcolor': '#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape':
'note', 'style': 'filled'}, 'subdispatchpipe': {'color': '#5E1F00', 'fillcolor':
'#FF3536', 'ordering': 'in', 'penwidth': 2, 'shape': 'note', 'style':
'filled'}}}, plot: {'color': '#fcf3dd', 'fillcolor': '#fcf3dd', 'label': 'plot',
'shape': 'egg'}, self: {'color': '#C1A4FE', 'fillcolor': '#C1A4FE', 'label':
'self', 'shape': 'egg'}, sink: {'color': '#303030', 'fillcolor': '#303030',
'fontcolor': '#FFFFFF', 'label': 'sink', 'shape': 'egg'}, start: {'color':
'#39bf7f', 'fillcolor': '#39bf7f', 'label': 'start', 'ordering': 'out', 'shape':
'egg'}}, 'info': {empty: {'fillcolor': '#FFFFFF', 'label': 'empty', 'shape':
'egg'}, end: {'color': '#084368', 'fillcolor': '#084368', 'fontcolor':
'#FFFFFF', 'label': 'end', 'ordering': 'in', 'shape': 'egg'}, none: {'data':
{'color': '#73c4fa', 'fillcolor': '#73c4fa', 'shape': 'box', 'style':
'rounded, filled'}, 'dispatcher': {'color': '#c6c6c6', 'fillcolor': '#c6c6c6',
'shape': 'note', 'style': 'filled'}, 'dispatchpipe': {'color': '#e8c268',
'fillcolor': '#e8c268', 'ordering': 'in', 'shape': 'note', 'style': 'filled'},
'edge': {None: None}, 'function': {'color': '#eed867', 'fillcolor': '#eed867',
'ordering': 'in', 'shape': 'box'}, 'function-dispatcher': {'color': '#eed867',
'fillcolor': '#eed867', 'ordering': 'in', 'shape': 'note'}, 'mapdispatch':
{'color': '#f4bd6a', 'fillcolor': '#f4bd6a', 'ordering': 'in', 'shape': 'note',
'style': 'filled'}, 'run_model': {'color': '#eed867', 'fillcolor': '#eed867',
'ordering': 'in', 'shape': 'note'}, 'subdispatch': {'color': '#ffc490',
'fillcolor': '#ffc490', 'shape': 'note', 'style': 'filled'},
'subdispatchfunction': {'color': '#f9d951', 'fillcolor': '#f9d951', 'ordering':
'in', 'shape': 'note', 'style': 'filled'}, 'subdispatchpipe': {'color':
'#f1cd5d', 'fillcolor': '#f1cd5d', 'ordering': 'in', 'shape': 'note', 'style':
'filled'}}}, plot: {'color': '#fcf3dd', 'fillcolor': '#fcf3dd', 'label': 'plot',
'shape': 'egg'}, self: {'color': '#C1A4FE', 'fillcolor': '#C1A4FE', 'label':
'self', 'shape': 'egg'}, sink: {'color': '#303030', 'fillcolor': '#303030',
'fontcolor': '#FFFFFF', 'label': 'sink', 'shape': 'egg'}, start: {'color':
'#39bf7f', 'fillcolor': '#39bf7f', 'label': 'start', 'ordering': 'out', 'shape':
'egg'}}, 'warning': {empty: {'fillcolor': '#FFFFFF', 'label': 'empty', 'shape':
'egg'}, end: {'color': '#084368', 'fillcolor': '#084368', 'fontcolor':
'#FFFFFF', 'label': 'end', 'ordering': 'in', 'shape': 'egg'}, none: {'data':
{'color': '#C9340A', 'fillcolor': '#fea22b', 'penwidth': 2, 'shape': 'box',
'style': 'rounded, filled'}, 'dispatcher': {'color': '#C9340A', 'fillcolor':
'#fea22b', 'penwidth': 2, 'shape': 'note', 'style': 'filled'}, 'dispatchpipe':
{'color': '#C9340A', 'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2,
'shape': 'note', 'style': 'filled'}, 'edge': {None: None}, 'function':
{'color': '#C9340A', 'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2,
'shape': 'box'}, 'function-dispatcher': {'color': '#C9340A', 'fillcolor':
'#fea22b', 'ordering': 'in', 'penwidth': 2, 'shape': 'note', 'style': 'filled'},
'run_model': {'color': '#C9340A',
'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2, 'shape': 'note'},
'dispatchpipe': {'color': '#C9340A', 'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2,
'shape': 'note', 'style': 'filled'}, 'run_model': {'color': '#C9340A',
'fillcolor': '#fea22b', 'ordering': 'in', 'penwidth': 2, 'shape': 'note'},

```

re_node

FolderNodeWeb.re_node = '^([.*+!]?)([\\w]+)(?>\\|([\\w]+))?\$'

WebFolder

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

Methods

`__init__`

`dot`

`view`

`__init__`

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

dot

WebFolder.dot(context=None)

view

WebFolder.view(filepath, context=None, viz=False, **kwargs)

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

Attributes

counter

digraph

ext

counter

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

digraph

```
WebFolder.digraph = {'body': {'splines': 'ortho', 'style': 'filled'},
'edge_attr': {}, 'format': 'svg', 'graph_attr': {'bgcolor': 'transparent',
'nslimit': '1', 'nslimit1': '1'}, 'node_attr': {'style': 'filled'}}
```

ext

`WebFolder.ext = ''`

WebMap

`class WebMap`

Methods

| | |
|---------------------------------|--|
| <code>__init__</code> | |
| <code>add_items</code> | |
| <code>app</code> | |
| <code>basic_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_directory</code> | |
| <code>get_dsp_from</code> | |
| <code>get_sol_from</code> | |
| <code>init_debug_subsite</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__()`

`add_items`

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

`app`

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

`basic_app`

`WebMap.basic_app(root_path, mute=True, blueprint_name=None, **kwargs)`

`clear`

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

`copy`

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

`fromkeys`

`WebMap.fromkeys(value=None)`

Create a new ordered dictionary with keys from iterable and values set to value.

`get`

`WebMap.get(key, default=None, /)`

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

`get_directory`

`WebMap.get_directory(directory)`

get_dsp_from

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

get_sol_from

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

init_debug_subsite

WebMap.**init_debug_subsite**(*func*)

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**(*key*, *last=True*)

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**(*last=True*)

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, viz_js=False)`

setdefault

`WebMap.setdefault(key, default=None)`

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__()`

Attributes

| |
|---|
| <code>blueprint_name</code> |
| <code>directory</code> |
| <code>idle_timeout</code> |
| <code>include_folders_as_filenames</code> |
| <code>methods</code> |
| <code>options</code> |
| <code>short_name</code> |
| <code>subsite_methods</code> |

`blueprint_name`

`WebMap.blueprint_name = None`

`directory`

`WebMap.directory = None`

`idle_timeout`

`WebMap.idle_timeout = 600`

`include_folders_as_filenames`

`WebMap.include_folders_as_filenames = False`

`methods`

`WebMap.methods = ['POST']`

options

```
WebMap.options = {'digraph', 'edge_data', 'max_lines', 'max_width', 'node_data',
                  'node_function', 'node_styles'}
```

short_name

```
WebMap.short_name = None
```

subsite_methods

```
WebMap.subsite_methods = ['GET', 'POST']
```

WebNode

```
class WebNode(folder, node_id, item, obj, dsp_node_id, short_name=None)
```

Methods

```
__init__
```

```
render
```

```
view
```

```
__init__
```

```
WebNode.__init__(folder, node_id, item, obj, dsp_node_id, short_name=None)
```

render

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

view

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

```
__init__(folder, node_id, item, obj, dsp_node_id, short_name=None)
```

Attributes

counter

ext

counter

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

ext

WebNode.ext = ''

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

generate_autosummary_docs

get_members

process_generate_options

setup

generate_autosummary_docs

generate_autosummary_docs(*sources*, *output_dir*=None, *suffix*='.rst', *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

| | |
|--------------------------------|--|
| <i>add_autodocumenter</i> | |
| <i>contains_doctest</i> | |
| <i>get_grandfather_content</i> | |
| <i>setup</i> | 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, *offset*=None)

setup

setup(*app*)

Setup *dispatcher* Sphinx extension module.

Classes

DispatcherDirective

DispatcherDocumenter

Specialized Documenter subclass for dispatchers.

DispatcherDirective

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

Methods

| | |
|---------------------------------|---|
| <code>__init__</code> | |
| <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>get_location</code> | Get current location info for logging. |
| <code>get_source_info</code> | Get source and line number. |
| <code>info</code> | |
| <code>run</code> | |
| <code>set_source_info</code> | Set source and line number to the node. |
| <code>severe</code> | |
| <code>warning</code> | |

`__init__`

`DispatcherDirective.__init__(name, arguments, options, content, lineno, content_offset, *args, **kwargs)`

`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*)

get_location

DispatcherDirective.**get_location**() → str

Get current location info for logging.

get_source_info

DispatcherDirective.**get_source_info**() → Tuple[str, int]

Get source and line number.

info

DispatcherDirective.**info**(*message*)

run

DispatcherDirective.**run**() → List[Node]

set_source_info

DispatcherDirective.**set_source_info**(*node*: Node) → None

Set source and line number to the node.

severe

DispatcherDirective.**severe**(*message*)

warning

DispatcherDirective.**warning**(*message*)

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

Attributes

| | |
|--|--|
| <code>final_argument_whitespace</code> | May the final argument contain whitespace? |
| <code>has_content</code> | May the directive have content? |
| <code>optional_arguments</code> | Number of optional arguments after the required arguments. |
| <code>required_arguments</code> | Number of required directive arguments. |

final_argument_whitespace

DispatcherDirective.**final_argument_whitespace** = True

May the final argument contain whitespace?

has_content

DispatcherDirective.**has_content** = True

May the directive have content?

optional_arguments

DispatcherDirective.**optional_arguments** = 0

Number of optional arguments after the required arguments.

required_arguments

DispatcherDirective.**required_arguments** = 1

Number of required directive arguments.

DispatcherDocumenter

class DispatcherDocumenter(*directive: DocumenterBridge, name: str, indent: str = ""*)

Specialized Documenter subclass for dispatchers.

Methods

| | |
|---|--|
| <code>__init__</code> | |
| <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> | <code>getattr()</code> override for types such as Zope interfaces. |
| <code>get_doc</code> | Decode and return lines of the docstring(s) for the object. |
| <code>get_module_comment</code> | |
| <code>get_object_members</code> | Return <i>(members_check_module, members)</i> where <i>members</i> is a list of <i>(membername, 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> . |
| <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. |
| <code>should_suppress_directive_header</code> | Check directive header should be suppressed. |
| <code>should_suppress_value_header</code> | Check <code>:value:</code> header should be suppressed. |
| <code>sort_members</code> | Sort the given member list. |
| <code>update_annotations</code> | Update <code>__annotations__</code> to support <code>type_comment</code> and so on. |
| <code>update_content</code> | Update docstring for the NewType object. |

`__init__`

`DispatcherDocumenter.__init__(directive: DocumenterBridge, name: str, indent: str = "") → None`

`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: str, source: str, *lineno: int) → None`

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() → bool`

Check if `self.object` is really defined in the module given by `self.modname`.

`document_members`

`DispatcherDocumenter.document_members(all_members: bool = False) → None`

Generate reST for member documentation.

If `all_members` is True, document all members, else those given by `self.options.members`.

`filter_members`

`DispatcherDocumenter.filter_members(members: Union[List[ObjectMember], List[Tuple[str, Any]]], want_all: bool) → List[Tuple[str, Any, bool]]`

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(**kwargs: Any) → str`

Format the argument signature of `self.object`.

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

`format_name`

`DispatcherDocumenter.format_name() → str`

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: Any, name: str, *defargs: Any) → Any`

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

`get_doc`

`DispatcherDocumenter.get_doc() → Optional[List[List[str]]]`

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

When it returns `None`, `autodoc-process-docstring` will not be called for this object.

get_module_comment

DispatcherDocumenter.get_module_comment(attrname: str) → Optional[List[str]]

get_object_members

DispatcherDocumenter.get_object_members(want_all: bool) → Tuple[bool,
Union[List[ObjectMember], List[Tuple[str, Any]]]]

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() → str

import_object

DispatcherDocumenter.import_object(*args, **kwargs)

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: List[List[str]]) → Iterator[str]`

Let the user process the docstrings before adding them.

resolve_name

`DispatcherDocumenter.resolve_name(modname: str, parents: Any, path: str, base: Any) → Tuple[str, List[str]]`

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.

should_suppress_directive_header

`DispatcherDocumenter.should_suppress_directive_header() → bool`

Check directive header should be suppressed.

should_suppress_value_header

`DispatcherDocumenter.should_suppress_value_header() → bool`

Check :value: header should be suppressed.

sort_members

`DispatcherDocumenter.sort_members(documenters: List[Tuple[Documenter, bool]], order: str) → List[Tuple[Documenter, bool]]`

Sort the given member list.

update_annotations

`DispatcherDocumenter.update_annotations(parent: Any) → None`

Update `__annotations__` to support `type_comment` and so on.

update_content

`DispatcherDocumenter.update_content(more_content: StringList) → None`

Update docstring for the `NewType` object.

`__init__(directive: DocumenterBridge, name: str, indent: str = "") → None`

Attributes

| | |
|-----------------------|--|
| blue_cache | |
| code | |
| config | |
| <i>content_indent</i> | indentation by which to indent the directive content |
| default_opt | |
| directivetype | |
| env | |
| is_doctest | |
| member_order | order if autodoc_member_order is set to 'groupwise' |
| modname | |
| object | |
| objpath | |
| <i>objtype</i> | name by which the directive is called (auto...) and the default generated directive name |
| option_spec | |
| parent | |
| priority | priority if multiple documenters return True from can_document_member |
| titles_allowed | true if the generated content may contain titles |

blue_cache

DispatcherDocumenter.**blue_cache** = {}

code

`DispatcherDocumenter.code = None`

config

`DispatcherDocumenter.config: Config = None`

content_indent

`DispatcherDocumenter.content_indent = ''`
 indentation by which to indent the directive content

default_opt

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

directivetype

`DispatcherDocumenter.directivetype = 'data'`

env

`DispatcherDocumenter.env: BuildEnvironment = None`

is_doctest

`DispatcherDocumenter.is_doctest = False`

member_order

`DispatcherDocumenter.member_order = 40`
 order if autodoc_member_order is set to ‘groupwise’

modname

`DispatcherDocumenter.modname: str = None`

object

`DispatcherDocumenter.object: Any = None`

objpath

`DispatcherDocumenter.objpath: List[str] = None`

objtype

`DispatcherDocumenter.objtype = 'dispatcher'`
 name by which the directive is called (auto...) and the default generated directive name

option_spec

`DispatcherDocumenter.option_spec: Dict[str, Callable[[str], Any]] = {'annotation': <function annotation_option>, 'code': <function bool_option>, 'data': <function bool_option>, 'description': <function bool_option>, 'dsp': <function bool_option>, 'func': <function bool_option>, 'height': <function length_or_unitless>, 'no-value': <function bool_option>, 'noindex': <function bool_option>, 'opt': <function _dsp2dot_option>, 'width': <function length_or_percentage_or_unitless>}`

parent

`DispatcherDocumenter.parent: Any = None`

priority

`DispatcherDocumenter.priority = -10`
 priority if multiple documenters return True from `can_document_member`

titles_allowed

`DispatcherDocumenter.titles_allowed = False`
 true if the generated content may contain titles

`content_indent = ''`
 indentation by which to indent the directive content

`objtype = 'dispatcher'`
 name by which the directive is called (auto...) and the default generated directive name

`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(*args, **kwargs)

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

html_visit_dispatcher

setup

Setup *dsp* Sphinx extension module.

html_visit_dispatcher

html_visit_dispatcher(self, node)

setup

setup(*app*)

Setup *dsp* Sphinx extension module.

Classes

DispatcherSphinxDirective

DispatcherSphinxDirective

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

Methods

| | |
|---------------------------------|---|
| <code>__init__</code> | |
| <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>get_location</code> | Get current location info for logging. |
| <code>get_source_info</code> | Get source and line number. |
| <code>info</code> | |
| <code>run</code> | |
| <code>set_source_info</code> | Set source and line number to the node. |
| <code>severe</code> | |
| <code>warning</code> | |

`__init__`

`DispatcherSphinxDirective.__init__(name, arguments, options, content, lineno, content_offset, block_text, state, state_machine)`

`add_name`

`DispatcherSphinxDirective.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`

`DispatcherSphinxDirective.assert_has_content()`

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

`debug`

`DispatcherSphinxDirective.debug(message)`

`directive_error`

`DispatcherSphinxDirective.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`

`DispatcherSphinxDirective.error(message)`

`get_location`

`DispatcherSphinxDirective.get_location() → str`

Get current location info for logging.

get_source_info

DispatcherSphinxDirective.**get_source_info**() → `Tuple[str, int]`

Get source and line number.

info

DispatcherSphinxDirective.**info**(*message*)

run

DispatcherSphinxDirective.**run**()

set_source_info

DispatcherSphinxDirective.**set_source_info**(*node: Node*) → `None`

Set source and line number to the node.

severe

DispatcherSphinxDirective.**severe**(*message*)

warning

DispatcherSphinxDirective.**warning**(*message*)

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

Attributes

| | |
|--|--|
| <code>final_argument_whitespace</code> | May the final argument contain whitespace? |
| <code>has_content</code> | May the directive have content? |
| <code>img_opt</code> | |
| <code>option_spec</code> | Mapping of option names to validator functions. |
| <code>optional_arguments</code> | Number of optional arguments after the required arguments. |
| <code>required_arguments</code> | Number of required directive arguments. |

final_argument_whitespace

`DispatcherSphinxDirective.final_argument_whitespace = False`

May the final argument contain whitespace?

has_content

`DispatcherSphinxDirective.has_content = True`

May the directive have content?

img_opt

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

option_spec

`DispatcherSphinxDirective.option_spec: Dict[str, Callable[[str], Any]] = {'align': <function align_spec>, 'alt': <function unchanged>, 'caption': <function unchanged>, 'class': <function class_option>, 'graphviz_dot': <function unchanged>, 'height': <function length_or_unitless>, 'index': <function bool_option>, 'layout': <function unchanged>, 'name': <function unchanged>, 'viz': <function bool_option>, 'width': <function length_or_percentage_or_unitless>}`

Mapping of option names to validator functions.

optional_arguments

`DispatcherSphinxDirective.optional_arguments = 1`

Number of optional arguments after the required arguments.

required_arguments

`DispatcherSphinxDirective.required_arguments = 1`

Number of required directive arguments.

`required_arguments = 1`

Number of required directive arguments.

`option_spec: Dict[str, Callable[[str], Any]] = {'align': <function align_spec>, 'alt': <function unchanged>, 'caption': <function unchanged>, 'class': <function class_option>, 'graphviz_dot': <function unchanged>, 'height': <function length_or_unitless>, 'index': <function bool_option>, 'layout': <function unchanged>, 'name': <function unchanged>, 'viz': <function bool_option>, 'width': <function length_or_percentage_or_unitless>}`

Mapping of option names to validator functions.

Functions

setup

Setup *dispatcher* Sphinx extension module.

setup

setup(*app*)

Setup *dispatcher* Sphinx extension module.

CHANGELOG

8.1 v1.4.9 (2023-01-23)

8.1.1 Feat

- (form): Update bundle.
- (dsp): Use *dataclass* for inf instance.

8.1.2 Fix

- (ext): Correct parent content getter.
- (form): Correct fullscreen behaviour.
- (form): Clean wrong error states.

8.2 v1.4.8 (2023-01-06)

8.2.1 Feat

- (form): Update bundle.
- (form): Make modal unmount.

8.2.2 Fix

- (form): Correct *useEffect* loop.
- (form): Add missing invocation of *editOnChange*.

8.3 v1.4.7 (2023-01-05)

8.3.1 Feat

- (form): Update bundle.
- (form): Request gzip schemas.
- (form): Enforce correct defaults.
- (form): Resolve schema.

8.3.2 Fix

- (test): Test only one python version for windows.
- (form): Invoke form validation after submit.
- (form): Use *retrieveSchema* function to retrieve field schema.
- (web): Correct debug url.

8.4 v1.4.6 (2023-01-04)

8.4.1 Feat

- (site): Drop gevent dependence.
- (form): Update bundle.
- (form): Add error handling on file widget.
- (form): Move *ReactModal* in a custom component.
- (form): Add *savingData* option to nav component.
- (form): Add download buttons to file widget.
- (form): Group all states to a single state + debounce live validation.
- (site): Enable async routes.
- (form): Reduce bundle size.
- (form): Add new method *path* for *ui:layout*.
- (form): Use gzip to POST requests.
- (form): Add download buttons to file widget.

8.4.2 Fix

- (form): Correct modal css.
- (form): Ensure datagrid string or bool format.

8.5 v1.4.5 (2022-12-27)

8.5.1 Feat

- (form): Add FileWidget + Improve Autosaving and enforce code splitting.

8.5.2 Fix

- (site): Correct *gevent* error when watcher is *None*.

8.6 v1.4.4 (2022-12-22)

8.6.1 Feat

- (test): Add more form test cases.
- (test): Disable logging for test cases.
- (site): Add option *url_prefix*.

8.6.2 Fix

- (form): Use modal instead popup to show the debug view.
- (web): Remove custom methods *PING* and *DEBUG* for standards *GET* and *POST*.

8.7 v1.4.3 (2022-12-21)

8.7.1 Feat

- (web): Add *DEBUG* method as *API* service.

8.7.2 Fix

- (test): Correct test cases to generate autodispatcher.
- (form): Correct bug when plot is empty.

8.8 v1.4.2 (2022-12-15)

8.8.1 Feat

- (form): Add options to edit/pre- post-process within the form dynamically.

8.9 v1.4.1 (2022-12-12)

8.9.1 Feat

- (base): Update default behaviour when invoking *plot*, *web* and *form*.
- (sol): Remove unused code.
- (core): Create a new module *util*.

8.9.2 Fix

- (form): Correct form *url* API.
- (doc): Remove *requires.io*.

8.10 v1.4.0 (2022-12-12)

8.10.1 Feat

- (form): Add extension for forms with test cases.
- (drw): Add option to add raw body to dot graphviz file.
- (dsp): Improve readability of *MapDispatch* results.
- (core): Drop cutoff functionality.
- (dsp): Add options to use *SubDispatchFunction* like *SubDispatch*.
- (setup) #19: Add option to publish schedula-core.
- (form): Add delete all button on datagrid.
- (parallel): Make sync the default executor.
- (setup) #19: Add feature to install only core functionalities.

8.10.2 Fix

- (binder): Correct installation of binder.
- (form): Correct *CSRF* error handling.
- (jinja)Disable HTML AutoEscape.
- (asy): Avoid adding solution when *NoSub*.

8.11 v1.3.6 (2022-11-21)

8.11.1 Feat

- (form): Add data saver and restore options + fix fullscreen + improve *ScrollTop*.

8.11.2 Fix

- (form): Fix layout *isEmpty*.

8.12 v1.3.5 (2022-11-08)

8.12.1 Fix

- (form): Correct data import in nav.

8.13 v1.3.4 (2022-11-07)

8.13.1 Feat

- (form): Add fullscreen support.
- (form): Add nunjucks support.
- (form): Add react-reflex component.
- (web): Add option to rise a *WebResponse* from a dispatch.
- (form): Add *CSRF* protection.

8.14 v1.3.3 (2022-11-03)

8.14.1 Feat

- (form): Add markdown.
- (form): Avoid rendering elements with empty children.
- (form): Add more option to accordion and stepper.

- (form): Change position of error messages.

8.14.2 Fix

- (rtd): Correct doc rendering.
- (form): Correct plotting behaviour.

8.15 v1.3.2 (2022-10-24)

8.15.1 Feat

- (drw, web, form): Add option to return a blueprint.
- (form): Update bundle.

8.15.2 Fix

- (form): Add extra missing package data.

8.16 v1.3.1 (2022-10-20)

8.16.1 Fix

- (form): Add missing package data.
- (ext): Correct documenter doctest import.

8.17 v1.3.0 (2022-10-19)

8.17.1 Feat

- (form): Add new method form to create jsonschema react forms automatically.
- (blue): Add option to limit the depth of sub-dispatch blue.

8.17.2 Fix

- (sol): Correct default initialization for sub-dispatchers.
- (setup): Ensure correct size of distribution pkg.

8.18 v1.2.19 (2022-07-06)

8.18.1 Feat

- (dsp): Add new utility function *run_model*.
- (dsp): Add *output_type_kw* option to *SubDispatch* utility.
- (core): Add workflow when function is a dsp.

8.18.2 Fix

- (blue): Add memo when call register by default.

8.19 v1.2.18 (2022-07-02)

8.19.1 Feat

- (micropython): Update build for *micropython==v1.19.1*.
- (sol): Improve speed performance.
- (dsp): Make *shrink* optional for *SubDispatchPipe*.
- (core): Improve performance dropping *set* instances.

8.20 v1.2.17 (2022-06-29)

8.20.1 Feat

- (sol): Improve speed performances.

8.20.2 Fix

- (sol): Correct missing reference due to sphinx update.
- (dsp): Correct wrong workflow.pred reference.

8.21 v1.2.16 (2022-05-10)

8.21.1 Fix

- (drw): Correct recursive plots.
- (doc): Correct *requirements.io* link.

8.22 v1.2.15 (2022-04-12)

8.22.1 Feat

- (sol): Improve performances of *_see_remote_link_node*.
- (drw): Improve performances of site rendering.

8.23 v1.2.14 (2022-01-21)

8.23.1 Fix

- (drw): Correct plot of *DispatchPipe*.

8.24 v1.2.13 (2022-01-13)

8.24.1 Feat

- (doc): Update copyright.
- (actions): Add *fail-fast: false*.
- (setup): Add missing dev requirement.

8.24.2 Fix

- (drw): Skip permission error in server cleanup.
- (core): Correct import dependencies.
- (doc): Correct link target.

8.25 v1.2.12 (2021-12-03)

8.25.1 Feat

- (test): Add test cases improving coverage.

8.25.2 Fix

- (drw): Correct graphviz *_view* attribute call.
- (drw): Correct cleanup function.

8.26 v1.2.11 (2021-12-02)

8.26.1 Feat

- (actions): Add test cases.
- (test): Update test cases.
- (drw): Make plot rendering parallel.
- (asy): Add *sync* executor.
- (dispatcher): Add auto inputs and outputs + prefix tags for *add_dispatcher* method.
- (setup): Pin sphinx version.

8.26.2 Fix

- (test): Remove windows long path test.
- (test): Correct test cases for parallel.
- (drw): Correct optional imports.
- (doc): Remove sphinx warning.
- (drw): Correct body format.
- (asy): Correct *atexit_register* function.
- (bin): Correct script.

8.27 v1.2.10 (2021-11-11)

8.27.1 Feat

- (drw): Add custom style per node.
- (drw): Make clean-up site optional.
- (drw): Add *force_plot* option to data node to plot Solution results.
- (drw): Update graphs colors.

8.27.2 Fix

- (setup): Pin graphviz version <0.18.
- (alg): Ensure *str* type of *node_id*.
- (drw): Remove empty node if some node is available.
- (drw): Add missing node type on js script.
- (drw): Extend short name to sub-graphs.

8.28 v1.2.9 (2021-10-05)

8.28.1 Feat

- (drw): Add option to reduce length of file names.

8.28.2 Fix

- (setup): Correct supported python versions.
- (doc): Correct typos.

8.29 v1.2.8 (2021-05-31)

8.29.1 Fix

- (doc): Skip KeyError when searching descriptions.

8.30 v1.2.7 (2021-05-19)

8.30.1 Feat

- (travis): Remove python 3.6 and add python 3.9 from text matrix.

8.30.2 Fix

- (sphinx): Add missing attribute.
- (sphinx): Update option parser.
- (doc): Update some documentation.
- (test): Correct test case missing library.

8.31 v1.2.6 (2021-02-09)

8.31.1 Feat

- (sol): Improve performances.

8.31.2 Fix

- (des): Correct description error due to *MapDispatch*.
- (drw): Correct *index* plotting.

8.32 v1.2.5 (2021-01-17)

8.32.1 Fix

- (core): Update copyright.
- (drw): Correct viz rendering.

8.33 v1.2.4 (2020-12-12)

8.33.1 Fix

- (drw): Correct plot auto-opening.

8.34 v1.2.3 (2020-12-11)

8.34.1 Feat

- (drw): Add plot option to use viz.js as back-end.

8.34.2 Fix

- (setup): Add missing requirement *requests*.

8.35 v1.2.2 (2020-11-30)

8.35.1 Feat

- (dsp): Add custom formatters for *MapDispatch* class.

8.36 v1.2.1 (2020-11-04)

8.36.1 Feat

- (dsp): Add *MapDispatch* class.
- (core): Add execution function log.

8.36.2 Fix

- (rtd): Correct documentation rendering in *rtd*.
- (autosummary): Correct bug for *AutosummaryEntry*.

8.37 v1.2.0 (2020-04-08)

8.37.1 Feat

- (dispatcher): Avoid failure when functions does not have the name.
- (ubuild): Add compiled and not compiled code.
- (sol): Improve speed importing functions directly for *heappop* and *heappush*.
- (dispatcher): Avoid failure when functions does not have the name.
- (dsp): Simplify repr of inf numbers.
- (micropython): Pin specific MicroPython version *v1.12*.
- (micropython): Add test using *.mpy* files.
- (setup): Add *MicroPython* support.
- (setup): Drop *dill* dependency and add *io* extra.
- (github): Add pull request templates.

8.37.2 Fix

- (test): Skip micropython tests.
- (ext): Update code for sphinx 3.0.0.
- (sphinx): Remove documentation warnings.
- (utils): Drop unused *pairwise* function.
- (dsp): Avoid fringe increment in *SubDispatchPipe*.

8.38 v1.1.1 (2020-03-12)

8.38.1 Feat

- (github): Add issue templates.
- (exc): Add base exception to *DispatcherError*.
- (build): Update build script.

8.39 v1.1.0 (2020-03-05)

8.39.1 Feat

- (core): Drop *networkx* dependency.
- (core): Add *ProcessPoolExecutor*.
- (asy): Add *ExecutorFactory* class.
- (asy): Split *asy* module.
- (core): Add support for python 3.8 and drop python 3.5.
- (asy): Check if *stopper* is set when getting executor.
- (asy): Add *mp_context* option in *ProcessExecutor* and *ProcessPoolExecutor*.

8.39.2 Fix

- (alg): Correct pipe generation when *NoSub* found.
- (asy): Remove un-useful and dangerous states before serialization.
- (asy): Ensure wait of all executor futures.
- (asy): Correct bug when future is set.
- (asy): Correct init and shutdown of executors.
- (sol): Correct raise exception order in *sol.result*.
- (travis): Correct tests collector.
- (test): Correct test for multiple async.

8.40 v1.0.0 (2020-01-02)

8.40.1 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.

8.40.2 Fix

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

8.41 v0.3.7 (2019-12-06)

8.41.1 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.

8.41.2 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.

8.42 v0.3.6 (2019-10-18)

8.42.1 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*.

8.43 v0.3.4 (2019-07-15)

8.43.1 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.

8.43.2 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.

8.44 v0.3.3 (2019-04-02)

8.44.1 Feat

- (dispatcher): Improve error message.

8.44.2 Fix

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

8.45 v0.3.2 (2019-02-23)

8.45.1 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.

8.45.2 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.

8.46 v0.3.1 (2018-12-10)

8.46.1 Fix

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

8.47 v0.3.0 (2018-12-08)

8.47.1 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.

8.47.2 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.

8.47.3 Other

- Refact: Update documentation.

8.48 v0.2.8 (2018-10-09)

8.48.1 Feat

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

8.49 v0.2.7 (2018-09-13)

8.49.1 Fix

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

8.50 v0.2.6 (2018-09-13)

8.50.1 Feat

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

8.51 v0.2.5 (2018-09-13)

8.51.1 Fix

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

8.52 v0.2.4 (2018-09-13)

8.52.1 Fix

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

8.53 v0.2.3 (2018-08-02)

8.53.1 Fix

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

8.54 v0.2.2 (2018-08-02)

8.54.1 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.

8.55 v0.2.1 (2018-07-24)

8.55.1 Feat

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

8.55.2 Fix

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

8.56 v0.2.0 (2018-07-19)

8.56.1 Feat

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

8.56.2 Fix

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

8.57 v0.1.19 (2018-06-05)

8.57.1 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.

8.58 v0.1.18 (2018-05-28)

8.58.1 Feat

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

8.59 v0.1.17 (2018-05-18)

8.59.1 Feat

- (travis): Run coveralls in python 3.6.

8.59.2 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* $\geq 0.2.7.1 \rightarrow dill \neq 0.2.7$.

8.60 v0.1.16 (2017-09-26)

8.60.1 Fix

- (requirements): Update dill requirements.

8.61 v0.1.15 (2017-09-26)

8.61.1 Fix

- (networkx): Update according to networkx 2.0.

8.62 v0.1.14 (2017-07-11)

8.62.1 Fix

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

8.62.2 Other

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

8.63 v0.1.13 (2017-06-26)

8.63.1 Feat

- (appveyor): Add python 3.6.

8.63.2 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.

8.64 v0.1.12 (2017-05-04)

8.64.1 Fix

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

8.65 v0.1.11 (2017-05-04)

8.65.1 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.

8.65.2 Fix

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

8.66 v0.1.10 (2017-04-03)

8.66.1 Feat

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

8.66.2 Fix

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

8.67 v0.1.9 (2017-02-09)

8.67.1 Fix

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

8.68 v0.1.8 (2017-02-09)

8.68.1 Feat

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

8.69 v0.1.7 (2017-02-08)

8.69.1 Fix

- (setup): Add missing `package_data`.

8.70 v0.1.6 (2017-02-08)

8.70.1 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.

8.71 v0.1.5 (2017-02-06)

8.71.1 Feat

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

8.71.2 Fix

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

8.72 v0.1.4 (2017-01-31)

8.72.1 Feat

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

8.72.2 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.

8.73 v0.1.3 (2017-01-29)

8.73.1 Fix

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

8.74 v0.1.2 (2017-01-28)

8.74.1 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.

8.74.2 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.

8.75 v0.1.1 (2017-01-21)

8.75.1 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.

8.75.2 Other

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

INDICES AND TABLES

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

PYTHON MODULE INDEX

S

- [schedula](#), 21
- [schedula.dispatcher](#), 21
- [schedula.ext](#), 287
 - [schedula.ext.autosummary](#), 287
 - [schedula.ext.dispatcher](#), 288
 - [schedula.ext.dispatcher.documenter](#), 288
 - [schedula.ext.dispatcher.graphviz](#), 302
- [schedula.utils](#), 80
 - [schedula.utils.alg](#), 81
 - [schedula.utils.asy](#), 88
 - [schedula.utils.asy.executors](#), 88
 - [schedula.utils.asy.factory](#), 94
 - [schedula.utils.base](#), 103
 - [schedula.utils.blue](#), 116
 - [schedula.utils.cst](#), 133
 - [schedula.utils.des](#), 134
 - [schedula.utils.drw](#), 134
 - [schedula.utils.drw.nodes](#), 135
 - [schedula.utils.dsp](#), 160
 - [schedula.utils.exc](#), 235
 - [schedula.utils.form](#), 236
 - [schedula.utils.gen](#), 244
 - [schedula.utils.graph](#), 255
 - [schedula.utils.imp](#), 258
 - [schedula.utils.io](#), 258
 - [schedula.utils.sol](#), 261
 - [schedula.utils.utl](#), 274
 - [schedula.utils.web](#), 274

Symbols

__init__() (*AsyncList method*), 102
 __init__() (*Base method*), 110
 __init__() (*BlueDispatcher method*), 126
 __init__() (*Blueprint method*), 132
 __init__() (*DiGraph method*), 257
 __init__() (*DispatchPipe method*), 180
 __init__() (*Dispatcher method*), 56
 __init__() (*DispatcherDirective method*), 292
 __init__() (*DispatcherDocumenter method*), 298
 __init__() (*DispatcherSphinxDirective method*), 305
 __init__() (*DspPipe method*), 88
 __init__() (*Executor method*), 89
 __init__() (*ExecutorFactory method*), 98
 __init__() (*FolderNode method*), 139
 __init__() (*FolderNodeWeb method*), 276
 __init__() (*FormMap method*), 242
 __init__() (*IdleContainer method*), 144
 __init__() (*MapDispatch method*), 195
 __init__() (*NoSub method*), 196
 __init__() (*NoView method*), 145
 __init__() (*PoolExecutor method*), 91
 __init__() (*ProcessExecutor method*), 92
 __init__() (*ProcessPoolExecutor method*), 93
 __init__() (*ReverseProxied method*), 145
 __init__() (*ServerThread method*), 148
 __init__() (*Site method*), 150
 __init__() (*SiteFolder method*), 151
 __init__() (*SiteIndex method*), 152
 __init__() (*SiteMap method*), 157
 __init__() (*SiteNode method*), 159
 __init__() (*SiteViz method*), 160
 __init__() (*Solution method*), 272
 __init__() (*SubDispatch method*), 208
 __init__() (*SubDispatchFunction method*), 220
 __init__() (*SubDispatchPipe method*), 232
 __init__() (*ThreadExecutor method*), 94
 __init__() (*Token method*), 254
 __init__() (*WebFolder method*), 279
 __init__() (*WebMap method*), 284
 __init__() (*WebNode method*), 286
 __init__() (*add_args method*), 233

__init__() (*inf method*), 233
 __init__() (*run_model method*), 235

A

add_args (*class in schedula.utils.dsp*), 233
 add_autodocumenter() (*in module schedula.ext.dispatcher.documenter*), 289
 add_content() (*DispatcherDocumenter method*), 302
 add_data() (*BlueDispatcher method*), 126
 add_data() (*Dispatcher method*), 57
 add_directive_header() (*DispatcherDocumenter method*), 302
 add_dispatcher() (*BlueDispatcher method*), 129
 add_dispatcher() (*Dispatcher method*), 62
 add_from_lists() (*BlueDispatcher method*), 130
 add_from_lists() (*Dispatcher method*), 64
 add_func() (*BlueDispatcher method*), 128
 add_func() (*Dispatcher method*), 60
 add_func_edges() (*in module schedula.utils.alg*), 82
 add_function() (*BlueDispatcher method*), 127
 add_function() (*Dispatcher method*), 58
 add_function() (*in module schedula.utils.dsp*), 161
 are_in_nested_dicts() (*in module schedula.utils.dsp*), 163
 async_process() (*in module schedula.utils.asy*), 98
 async_thread() (*in module schedula.utils.asy*), 98
 AsyncList (*class in schedula.utils.asy*), 100
 atexit_register() (*in module schedula.utils.asy*), 99
 autoplot_callback() (*in module schedula.utils.drw*), 135
 autoplot_function() (*in module schedula.utils.drw*), 135
 await_result() (*in module schedula.utils.asy*), 99

B

Base (*class in schedula.utils.base*), 103
 before_request() (*in module schedula.utils.drw*), 136
 blue() (*Dispatcher method*), 72
 blue() (*SubDispatch method*), 208
 BlueDispatcher (*class in schedula.utils.blue*), 117
 Blueprint (*class in schedula.utils.blue*), 130
 bypass() (*in module schedula.utils.dsp*), 163

C

`cached_view()` (in module `schedula.utils.drw`), 136
`can_document_member()` (*DispatcherDocumenter* class method), 301
`check_targets()` (*Solution* method), 273
`check_wait_in()` (*Solution* method), 272
`cls` (*Blueprint* attribute), 132
`combine_dicts()` (in module `schedula.utils.dsp`), 163
`combine_nested_dicts()` (in module `schedula.utils.dsp`), 164
`contains_doctest()` (in module `schedula.ext.dispatcher.documenter`), 289
`content_indent` (*DispatcherDocumenter* attribute), 301
`copy()` (*Dispatcher* method), 71
`copy_structure()` (*Dispatcher* method), 57
`counter` (*Dispatcher* attribute), 57
`counter` (*FolderNode* attribute), 142
`counter` (*SiteFolder* attribute), 151
`counter` (*SiteNode* attribute), 159
`counter()` (in module `schedula.utils.gen`), 244

D

`data_nodes` (*Dispatcher* property), 71
`default_values` (*Dispatcher* attribute), 56
`dict_diff()` (in module `schedula.utils.utl`), 274
`DiGraph` (class in `schedula.utils.graph`), 255
`dispatch()` (*Dispatcher* method), 73
`Dispatcher` (class in `schedula.dispatcher`), 21
`DispatcherDirective` (class in `schedula.ext.dispatcher.documenter`), 289
`DispatcherDocumenter` (class in `schedula.ext.dispatcher.documenter`), 293
`DispatcherSphinxDirective` (class in `schedula.ext.dispatcher.graphviz`), 303
`DispatchPipe` (class in `schedula.utils.dsp`), 169
`dmap` (*Dispatcher* attribute), 56
`DspPipe` (class in `schedula.utils.alg`), 86

E

`EMPTY` (in module `schedula.utils.cst`), 133
`END` (in module `schedula.utils.cst`), 133
`Executor` (class in `schedula.utils.asy.executors`), 89
`executor` (*Dispatcher* attribute), 56
`ExecutorFactory` (class in `schedula.utils.asy.factory`), 94
`extend()` (*Blueprint* method), 132
`extend()` (*Dispatcher* method), 72

F

`FolderNode` (class in `schedula.utils.drw`), 137
`FolderNodeWeb` (class in `schedula.utils.web`), 275
`form()` (*Base* method), 111

`format_signature()` (*DispatcherDocumenter* method), 302
`FormMap` (class in `schedula.utils.form`), 236
`function_nodes` (*Dispatcher* property), 71

G

`generate()` (*DispatcherDocumenter* method), 302
`generate_autosummary_docs()` (in module `schedula.ext.autosummary`), 288
`get_attr_doc()` (in module `schedula.utils.des`), 134
`get_full_pipe()` (in module `schedula.utils.alg`), 82
`get_grandfather_content()` (in module `schedula.ext.dispatcher.documenter`), 289
`get_link()` (in module `schedula.utils.des`), 134
`get_match_func()` (in module `schedula.utils.drw`), 136
`get_members()` (in module `schedula.ext.autosummary`), 288
`get_nested_dicts()` (in module `schedula.utils.dsp`), 164
`get_node()` (*Base* method), 114
`get_real_modname()` (*DispatcherDocumenter* method), 301
`get_sub_dsp()` (*Dispatcher* method), 66
`get_sub_dsp_from_workflow()` (*Dispatcher* method), 68
`get_sub_dsp_from_workflow()` (*Solution* method), 273
`get_sub_node()` (in module `schedula.utils.alg`), 83
`get_summary()` (in module `schedula.utils.des`), 134
`get_unused_node_id()` (in module `schedula.utils.utl`), 274

in

in

in

H

`html_visit_dispatcher()` (in module `schedula.ext.dispatcher.graphviz`), 302

I

`IdleContainer` (class in `schedula.utils.drw`), 142
`import_object()` (*DispatcherDocumenter* method), 302
`inf` (class in `schedula.utils.dsp`), 233

J

`jinja2_format()` (in module `schedula.utils.drw`), 136

K

`kk_dict()` (in module `schedula.utils.dsp`), 164

L

`load_default_values()` (in module `schedula.utils.io`), 258
`load_dispatcher()` (in module `schedula.utils.io`), 259
`load_map()` (in module `schedula.utils.io`), 259

M

map_dict() (in module *schedula.utils.dsp*), 165
 map_list() (in module *schedula.utils.dsp*), 165
 MapDispatch (class in *schedula.utils.dsp*), 183
 module
 schedula, 21
 schedula.dispatcher, 21
 schedula.ext, 287
 schedula.ext.autosummary, 287
 schedula.ext.dispatcher, 288
 schedula.ext.dispatcher.documenter, 288
 schedula.ext.dispatcher.graphviz, 302
 schedula.utils, 80
 schedula.utils.alg, 81
 schedula.utils.asy, 88
 schedula.utils.asy.executors, 88
 schedula.utils.asy.factory, 94
 schedula.utils.base, 103
 schedula.utils.blue, 116
 schedula.utils.cst, 133
 schedula.utils.des, 134
 schedula.utils.drw, 134
 schedula.utils.drw.nodes, 135
 schedula.utils.dsp, 160
 schedula.utils.exc, 235
 schedula.utils.form, 236
 schedula.utils.gen, 244
 schedula.utils.graph, 255
 schedula.utils.imp, 258
 schedula.utils.io, 258
 schedula.utils.sol, 261
 schedula.utils.utl, 274
 schedula.utils.web, 274

N

name (*Dispatcher* attribute), 56
 nodes (*Dispatcher* attribute), 56
 NONE (in module *schedula.utils.cst*), 133
 NoSub (class in *schedula.utils.dsp*), 196
 NoView (class in *schedula.utils.drw*), 145

O

objtype (*DispatcherDocumenter* attribute), 301
 option_spec (*DispatcherSphinxDirective* attribute), 306

P

parent_func() (in module *schedula.utils.dsp*), 166
 parse_funcs() (in module *schedula.utils.drw*), 136
 parse_name() (*DispatcherDocumenter* method), 302
 pipe (*Solution* property), 273
 PLOT (in module *schedula.utils.cst*), 133
 plot() (*Base* method), 112
 plot() (*DispatchPipe* method), 181

PoolExecutor (class in *schedula.utils.asy.executors*), 89
 process_generate_options() (in module *schedula.ext.autosummary*), 288
 ProcessExecutor (class in *schedula.utils.asy.executors*), 91
 ProcessPoolExecutor (class in *schedula.utils.asy.executors*), 92

R

raises (*Dispatcher* attribute), 56
 register() (*Blueprint* method), 132
 register_executor() (in module *schedula.utils.asy*), 99
 render_output() (in module *schedula.utils.drw*), 136
 replicate_value() (in module *schedula.utils.dsp*), 167
 required_arguments (*DispatcherSphinxDirective* attribute), 306
 result() (*Solution* method), 273
 ReverseProxied (class in *schedula.utils.drw*), 145
 run() (*IdleContainer* method), 144
 run() (*ServerThread* method), 148
 run_model (class in *schedula.utils.dsp*), 234
 run_server() (in module *schedula.utils.drw*), 136

S

save_default_values() (in module *schedula.utils.io*), 260
 save_dispatcher() (in module *schedula.utils.io*), 260
 save_map() (in module *schedula.utils.io*), 261
schedula
 module, 21
 schedula.dispatcher
 module, 21
 schedula.ext
 module, 287
 schedula.ext.autosummary
 module, 287
 schedula.ext.dispatcher
 module, 288
 schedula.ext.dispatcher.documenter
 module, 288
 schedula.ext.dispatcher.graphviz
 module, 302
 schedula.utils
 module, 80
 schedula.utils.alg
 module, 81
 schedula.utils.asy
 module, 88
 schedula.utils.asy.executors
 module, 88
 schedula.utils.asy.factory
 module, 94
 schedula.utils.base

- module, 103
- schedula.utils.blue
 - module, 116
- schedula.utils.cst
 - module, 133
- schedula.utils.des
 - module, 134
- schedula.utils.drw
 - module, 134
- schedula.utils.drw.nodes
 - module, 135
- schedula.utils.dsp
 - module, 160
- schedula.utils.exc
 - module, 235
- schedula.utils.form
 - module, 236
- schedula.utils.gen
 - module, 244
- schedula.utils.graph
 - module, 255
- schedula.utils.imp
 - module, 258
- schedula.utils.io
 - module, 258
- schedula.utils.sol
 - module, 261
- schedula.utils.utl
 - module, 274
- schedula.utils.web
 - module, 274
- search_node_description() (in module schedula.utils.des), 134
- select_diff() (in module schedula.utils.utl), 274
- selector() (in module schedula.utils.dsp), 167
- SELF (in module schedula.utils.cst), 133
- ServerThread (class in schedula.utils.drw), 145
- set_default_value() (BlueDispatcher method), 130
- set_default_value() (Dispatcher method), 65
- setup() (in module schedula.ext.autosummary), 288
- setup() (in module schedula.ext.dispatcher), 307
- setup() (in module schedula.ext.dispatcher.documenter), 289
- setup() (in module schedula.ext.dispatcher.graphviz), 303
- shrink_dsp() (Dispatcher method), 78
- shutdown_executor() (in module schedula.utils.asy), 100
- shutdown_executors() (in module schedula.utils.asy), 100
- SINK (in module schedula.utils.cst), 133
- Site (class in schedula.utils.drw), 149
- site_view() (in module schedula.utils.drw), 136
- SiteFolder (class in schedula.utils.drw), 150
- SiteIndex (class in schedula.utils.drw), 152
- SiteMap (class in schedula.utils.drw), 153
- SiteNode (class in schedula.utils.drw), 158
- SiteViz (class in schedula.utils.drw), 159
- Solution (class in schedula.utils.sol), 261
- solution (Dispatcher attribute), 57
- stack_nested_keys() (in module schedula.utils.dsp), 168
- START (in module schedula.utils.cst), 133
- stlp() (in module schedula.utils.dsp), 168
- sub_dsp_nodes (Dispatcher property), 71
- SubDispatch (class in schedula.utils.dsp), 196
- SubDispatchFunction (class in schedula.utils.dsp), 209
- SubDispatchPipe (class in schedula.utils.dsp), 221
- summation() (in module schedula.utils.dsp), 168

T

- ThreadExecutor (class in schedula.utils.asy.executors), 93
- Token (class in schedula.utils.gen), 244

U

- uncpath() (in module schedula.utils.drw), 136
- update_filenames() (in module schedula.utils.drw), 136

V

- valid_filename() (in module schedula.utils.drw), 137
- var_keyword (SubDispatchPipe attribute), 232

W

- web() (Base method), 110
- WebFolder (class in schedula.utils.web), 279
- WebMap (class in schedula.utils.web), 280
- WebNode (class in schedula.utils.web), 286