Connectors Documentation

Release 4.2

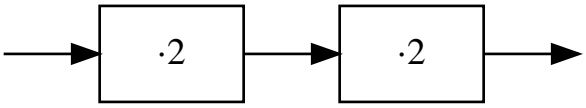
Jonas Schulte-Coerne

Jan 29, 2022

Contents

The *Connectors* package facilitates the writing of block-diagram-like processing networks. For this it provides decorators for the methods of processing classes, so they can be connected to each other. When a parameter in such a processing network is changed, the result values will also be updated automatically. This is similar to a pipes and filters architecture, the observer pattern or streams.

This short example demonstrates the core functionality of the *Connectors* package by implementing a processing network of two sequential blocks, which double their input value:




```
>>> import connectors
>>>
>>> class TimesTwo:
...     def __init__(self, value=0):
...         self.__value = value
...
...     @connectors.Input("get_double")
...     def set_value(self, value):
...         self.__value = value
...
...     @connectors.Output()
...     def get_double(self):
...          return 2 * self.__value
>>>
>>> d1 = TimesTwo()                                    # create an instance that␣
↪doubles its input value
>>> d2 = TimesTwo().set_value.connect(d1.get_double)   # create a second instance␣
↪and connect it to the first
>>> d2.get_double()
0
>>> d1.set_value(2)
>>> d2.get_double()                                    # causes the new input value␣
↪2 to be processed by d1 and d2
8
```

CHAPTER 1

Contents

1.1 Reference

This section contains the API reference for the *Connectors* package.

1.1.1 Decorators

The main feature of the *Connectors* package are the following decorators.

class connectors.**Output**(*caching=True*, *parallelization=<Parallelization.THREAD: 2>*, *executor=<connectors._common._executors.ThreadingExecutor object>*)

A decorator, that marks a method as an output connector. These connections can be used to automatically update a processing chain when a value has changed. The decorated method must not take any arguments.

Parameters

- **caching** – True, if caching shall be enabled, False otherwise. See the *OutputConnector*'s *set_caching()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *OutputConnector*'s *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *OutputConnector*'s *set_executor()* method for details

class connectors.**Input**(*observers=()*, *laziness=<Laziness.ON_REQUEST: 1>*, *parallelization=<Parallelization.SEQUENTIAL: 1>*, *executor=<connectors._common._executors.ThreadingExecutor object>*)

A decorator, that marks a method as an input for single connections. These connections can be used to automatically update a processing chain when a value has changed. The decorated method must take exactly one argument.

Parameters

- **observers** – the names of output methods that are affected by passing a value to this connector. For convenience it is also possible to pass a string here, if only one output connector depends on this input.
- **laziness** – a flag from the *connectors.Laziness* enum. See the *SingleInputConnector*'s *set_laziness()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *MultiInputConnector*'s *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *InputConnector*'s *set_executor()* method for details

announce_condition (*method*)

A decorator, that can be used as a method of the connector method, to define a condition for the propagation of announcements through the input connector.

The decorated method shall return `True`, if the announcement shall be propagated and `False` otherwise. For normal input connectors, it shall not require any arguments, while for multi-input connectors, it must accept the data ID of the connection, through which the announcement was received.

Before the values in a processing chain are updated, the value changes are announced to the downstream processors. Only if data is retrieved through an output connector, that has pending announcements, the actual value changes are requested from the upstream processors (lazy execution). If an input connector has defined a condition on the propagation of announcements and this condition evaluates to `False`, the announcements are not forwarded to the downstream processors. This also prevents those processors from requesting updated values from upstream.

The usage is described by the following example: if *A* is the name of the method, that is decorated with *connectors.Input* or *connectors.MultiInput*, the method for the condition has to be decorated with `@A.announce_condition`.

Param the method, that defines the condition

Returns the same method

notify_condition (*method*)

A decorator, that can be used as a method of the connector method, to define a condition for notifying the observing output connectors about a value, that has been changed by this connector.

The decorated method shall return `True`, if the notification shall be sent and `False` otherwise. For normal input connectors, it shall accept the new value as an argument, while for multi-input connectors, it must accept the data ID of the connection, through which the announcement was received and the new value.

This condition is checked after the input connector (the setter method) has been executed. If an input connector has defined a condition on the notification of its observing output connectors and this condition evaluates to `False`, the output connectors are sent a cancel notification, that informs them, that the state of the object, to which the connectors belong, has not changed in a relevant way. This prevents the update of values further down the processing chain.

The usage is described by the following example: if *A* is the name of the method, that is decorated with *connectors.Input* or *connectors.MultiInput*, the method for the condition has to be decorated with `@A.notify_condition`.

Param the method, that defines the condition

Returns the same method

class connectors.**MultiInput**(*observers=()*, *laziness=<Laziness.ON_REQUEST: 1>*, *parallelization=<Parallelization.SEQUENTIAL: 1>*, *executor=<connectors._common._executors.ThreadingExecutor object>*)

A decorator, that marks a method as an input for multiple connections. These connections can be used to automatically update a processing chain when a value has changed.

The decorated method must take exactly one argument and return a unique id. For every MultiInput-method a remove method has to be provided, which removes a value, that has previously been added.

A replace method can be provided optionally. This method is called, when the value of a connected output has changed, rather than removing the old value and adding the new.

See the *remove()* and *replace()* methods for documentation about how to define these methods for a multi-input connector.

Parameters

- **observers** – the names of output methods that are affected by passing a value to this connector. For convenience it is also possible to pass a string here, if only one output connector depends on this input.
- **laziness** – a flag from the *connectors.Laziness* enum. See the *MultiInputConnector*'s *set_laziness()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *MultiInputConnector*'s *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *MultiInputConnector*'s *set_executor()* method for details

announce_condition (*method*)

A decorator, that can be used as a method of the connector method, to define a condition for the propagation of announcements through the input connector.

The decorated method shall return `True`, if the announcement shall be propagated and `False` otherwise. For normal input connectors, it shall not require any arguments, while for multi-input connectors, it must accept the data ID of the connection, through which the announcement was received.

Before the values in a processing chain are updated, the value changes are announced to the downstream processors. Only if data is retrieved through an output connector, that has pending announcements, the actual value changes are requested from the upstream processors (lazy execution). If an input connector has defined a condition on the propagation of announcements and this condition evaluates to `False`, the announcements are not forwarded to the downstream processors. This also prevents those processors from requesting updated values from upstream.

The usage is described by the following example: if *A* is the name of the method, that is decorated with *connectors.Input* or *connectors.MultiInput*, the method for the condition has to be decorated with `@A.announce_condition`.

Param the method, that defines the condition

Returns the same method

notify_condition (*method*)

A decorator, that can be used as a method of the connector method, to define a condition for notifying the observing output connectors about a value, that has been changed by this connector.

The decorated method shall return `True`, if the notification shall be sent and `False` otherwise. For normal input connectors, it shall accept the new value as an argument, while for multi-input connectors, it must accept the data ID of the connection, through which the announcement was received and the new value.

This condition is checked after the input connector (the setter method) has been executed. If an input connector has defined a condition on the notification of its observing output connectors and this condition evaluates to `False`, the output connectors are sent a cancel notification, that informs them, that the state

of the object, to which the connectors belong, has not changed in a relevant way. This prevents the update of values further down the processing chain.

The usage is described by the following example: if *A* is the name of the method, that is decorated with *connectors.Input* or *connectors.MultiInput*, the method for the condition has to be decorated with `@A.notify_condition`.

Param the method, that defines the condition

Returns the same method

remove (*method*)

A method of the decorated method to decorate the remove method, with which data, that has been added through the decorated method can be removed.

A remove method has to take the ID, which has been returned by the multi-input method as a parameter, so it knows which value has to be removed.

The usage is described by the following example: if *A* is the name of the method, that is decorated with *MultiInput*, the remove method has to be decorated with `@A.remove`.

Parameters **method** – the decorated remove method

Returns a MultiInputAssociateDescriptor, that generates a MultiInputAssociateProxy, which enhances the decorated method with the functionality, that is required for the multi-input connector

replace (*method*)

A method of the decorated method to decorate the replace method, with which data, that has been added through the decorated method, can be replaced with new data without changing the ID under which it is stored.

A replace method has to take the ID, under which the old data is stored, as first parameter and the new data as second parameter. It is strongly recommended, that this method stores the new data under the same ID as the old data. And this method must return the ID, under which the new data is stored.

Also, if no data has been stored under the given ID, yet, the replace method shall simply store the data under the given ID instead of raising an error.

Specifying a replace method is optional. If no method has been decorated to be a replace method, the *MultiInput* connector falls back to removing the old data and adding the updated one, whenever updated data is propagated through its connections.

The usage is described by the following example: if *A* is the name of the method, that is decorated with *MultiInput*, the remove method has to be decorated with `@A.replace`.

Parameters **method** – the decorated replace method

Returns a MultiInputAssociateDescriptor, that generates a MultiInputAssociateProxy, which enhances the decorated method with the functionality, that is required for the multi-input connector

1.1.2 Connectors

This section documents the capabilities of the connector objects, with which the decorated methods are replaced. Instances of the following classes replace the decorated methods, so they are enhanced with the functionality of a connector. These classes are not instantiated by code outside the *Connectors* package.

Connectors for setter methods

class connectors.connectors.**SingleInputConnector**(*instance*, *method*, *observers*, *laziness*, *parallelization*, *executor*)

A connector-class that replaces setter methods, so they can be used to connect different objects in a processing chain.

Parameters

- **instance** – the instance of which the method is replaced by this connector
- **method** – the unbound method, that is replaced by this connector
- **observers** – the names of output methods that are affected by passing a value to this connector
- **laziness** – a flag from the *connectors.Laziness* enum. See the *set_laziness()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details

connect(*connector*)

Connects this *InputConnector* to an output.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *InputConnector* has replaced a method

disconnect(*connector*)

Disconnects this *InputConnector* from an output, to which is has been connected.

Parameters **connector** – a *Connector* instance from which this connector shall be disconnected

Returns the instance of which this *Connector* has replaced a method

set_executor(*executor*)

Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_laziness(*laziness*)

Configures the lazy execution of the connector. Normally the connectors are executed lazily, which means, that any computation is only started, when the result of a processing chain is requested. For certain use cases it is necessary to disable this lazy execution, though, so that the values are updated immediately as soon as new data is available. There are different behaviors for the (non) lazy execution, which are described in the *connectors.Laziness* enum.

Parameters **laziness** – a flag from the *connectors.Laziness* enum

set_parallelization(*parallelization*)

Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint,

which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

class connectors.connectors.**MultiInputConnector**(*instance*, *method*, *remove_method*, *replace_method*, *observers*, *laziness*, *parallelization*, *executor*)

A connector-class that replaces special setter methods, that allow to pass multiple values, so they can be used to connect different objects in a processing chain.

Parameters

- **instance** – the instance of which the method is replaced by this connector
- **method** – the unbound method, that is replaced by this connector
- **remove_method** – an unbound method, that is used to remove data, that has been added through this connector
- **replace_method** – an unbound method, that is used to replace data, that has been added through this connector
- **observers** – the names of output methods that are affected by passing a value to this connector
- **laziness** – a flag from the *connectors.Laziness* enum. See the *set_laziness()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details

__getitem__(*key*)

Allows to use a multi-input connector as multiple single-input connectors.

The key, under which a virtual single-input connector is accessed, shall also be returned the data ID, under which the result of the connected output is stored.

Parameters **key** – a key for accessing a particular virtual single-input connector

Returns a *MultiInputItem*, which enhances the decorated method with the functionality of the virtual single-input connector

connect(*connector*)

Connects this *InputConnector* to an output.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *InputConnector* has replaced a method

disconnect(*connector*)

Disconnects this *InputConnector* from an output, to which is has been connected.

Parameters **connector** – a *Connector* instance from which this connector shall be disconnected

Returns the instance of which this *Connector* has replaced a method

set_executor(*executor*)
Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_laziness(*laziness*)
Configures the lazy execution of the connector. Normally the connectors are executed lazily, which means, that any computation is only started, when the result of a processing chain is requested. For certain use cases it is necessary to disable this lazy execution, though, so that the values are updated immediately as soon as new data is available. There are different behaviors for the (non) lazy execution, which are described in the *connectors.Laziness* enum.

Parameters **laziness** – a flag from the *connectors.Laziness* enum

set_parallelization(*parallelization*)
Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

Connectors for getter methods

class connectors.connectors.**OutputConnector**(*instance*, *method*, *caching*, *parallelization*, *executor*)
A connector-class that replaces getter methods, so they can be used to connect different objects.

Parameters

- **instance** – the instance of which the method is replaced by this connector
- **method** – the unbound method that is replaced by this connector
- **caching** – True, if caching shall be enabled, False otherwise. See the *set_caching()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details

connect(*connector*)
A method for connecting this output connector to an input connector.

Parameters **connector** – the input connector to which this connector shall be connected

Returns the instance of which this *OutputConnector* has replaced a method

disconnect(*connector*)
A method for disconnecting this output connector from an input connector, to which it is currently connected.

Parameters **connector** – the input connector from which this connector shall be disconnected

Returns the instance of which this *OutputConnector* has replaced a method

set_caching(*caching*)
Specifies, if the result value of this output connector shall be cached. If caching is enabled and the result value is retrieved (e.g. through a connection or by calling the connector), the cached value is returned and the replaced getter method is not called unless the result value has to be re-computed, because an observed setter method has changed a parameter for the computation. In this case, the getter method is only called once, independent of the number of connections through which the result value has to be passed.

Parameters **caching** – True, if caching shall be enabled, False otherwise

set_executor(*executor*)
Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_parallelization(*parallelization*)
Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

class connectors.connectors.**MultiOutputConnector**(*instance*, *method*, *caching*, *parallelization*, *executor*, *keys*)
A connector-class that replaces getter methods, which accept one parameter, so they can be used as a multi-output connector, which can be connected to different objects.

Multi-output connectors can either be used to route a dynamic number of values to a multi-input connector. Or the argument for the `[]`-operator can be used to parameterize the getter method.

Parameters

- **instance** – the instance of which the method is replaced by this connector
- **method** – the unbound method that is replaced by this connector
- **caching** – True, if caching shall be enabled, False otherwise. See the *set_caching()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details
- **keys** – an unbound method, that returns the keys for which this multi-output connector shall compute values, when it is connected to a multi-input connector

__getitem__(*key*)
Allows to use a multi-output connector as multiple single-output connectors.

Parameters **key** – a key for accessing a particular virtual single-output connector

Returns a *connectors._common._multioutput_item.MultiOutputItem*, which enhances the decorated method with the functionality of the virtual single-output connector

connect(*connector*)
A method for connecting this output connector to an input connector. This is only allowed with multi-input connectors. In order to establish a connection to a single-input connector, use the `[]` operator to specify, which value shall be passed through the connection.

Parameters **connector** – the input connector to which this connector shall be connected

Returns the instance of which this *OutputConnector* has replaced a method

disconnect(*connector*)
A method for disconnecting this output connector from an input connector, to which it is currently connected.

Parameters **connector** – the input connector from which this connector shall be disconnected

Returns the instance of which this *OutputConnector* has replaced a method

set_caching(*caching*)
Specifies, if the result value of this output connector shall be cached. If caching is enabled and the result value is retrieved (e.g. through a connection or by calling the connector), the cached value is returned and the replaced getter method is not called unless the result value has to be re-computed, because an observed setter method has changed a parameter for the computation. In this case, the getter method is only called once, independent of the number of connections through which the result value has to be passed.

Parameters **caching** – True, if caching shall be enabled, False otherwise

set_executor(*executor*)
Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_parallelization(*parallelization*)
Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

1.1.3 Configuration options

This section describes

Automated parallelization

The following functionalities are for configuring the automated parallelization of the connector's computations.

class connectors.**Parallelization**
An enumeration type for the parallelization parameter of an executor's *run_method()* method:

- **SEQUENTIAL** the method can only be executed sequentially.
- **THREAD** the method should be executed in a separate thread, sequential execution is possible as a fallback.
- **PROCESS** the method should be executed in a separate process, threaded execution or sequential execution can be used as a fallback.

connectors.**executor**(*threads=None*, *processes=0*)
A factory function for creating *Executor* objects. Executors define how the computations of a processing chain are parallelized by executing them in separate threads or processes. This function creates an executor and configures it to use at maximum the given number of threads or processes.

Parameters

- **threads** – an integer number of threads or None to determine the number automatically. 0 disables the thread based parallelization.
- **processes** – an integer number of processes or None to determine the number automatically (in this case, the number of CPU cores will be taken). 0 disables the process based parallelization.

Configuring the laziness

Flags of the following enumeration can be passed to an input connectors *set_laziness()* method.

class connectors.**Laziness**
An enumeration type for defining the laziness of input connectors. The enumeration values are sorted by how lazy the behavior is, which they represent, so smaller/greater comparisons are possible. Do not rely on the exact integer value though, since they are not guaranteed to have the same value in any version of this package.

1. **ON_REQUEST** the setter is called, when its execution is requested by a method further down the processing chain.
2. **ON_NOTIFY** the setter is called, when the connected getter has computed the input value, even if the execution of this setter has not been requested.
3. **ON_ANNOUNCE** the setter is requests its input value immediately and is executed as soon as that is available. But connecting the setter, does not cause a request of the input value.
4. **ON_CONNECT** same as ON_ANNOUNCE, but the value is also requested, when a new connection is established, which influences the input value of this connector. This connection is does not necessarily have to be with this connector, but it can also be further upstream in the processing chain.

1.1.4 Helper functionalities

This is the API reference for helper functionalities, that facilitate the use of the *Connectors* package.

class connectors.**MultiInputData**(*datas=()*)
A container for data that is managed with a multi-input connector. This is basically an OrderedDict with an add method, that stores the added data under a unique key. This facilitates the implementation of a class with a *MultiInput* connector:

```
>>> import connectors
>>> class ReplacingMultiInput:
...     def __init__(self):
...         self.__data = connectors.MultiInputData()
...
...     @connectors.MultiInput()
...     def add_value(self, value):
...         return self.__data.add(value)
...
...     @add_value.remove
...     def remove_value(self, data_id):
...         del self.__data[data_id]
...
...     @add_value.replace
...     def replace_value(self, data_id, value):
...         self.__data[data_id] = value
...         return data_id
```

Parameters **datas** – an optional sequence of data objects, that shall be added to the container

add(*data*)
Adds a data set to the container.

Parameters **data** – the data set that shall be added

Returns the id under which the data is stored

1.1.5 Macro connectors for encapsulating processing networks in a class

Classes, which encapsulate networks of objects, that are connected with the functionalities of the *Connectors* package, are called *macros* or *macro classes* in the scope of this package. The connectors, that are used to export connectors from the internal network as connectors of the macro, are called macro connectors. This is the API reference for the functionalities, that are related to macro connectors.

Decorating methods

The following classes can be used to decorate methods, so they become macro connectors.

class connectors.**MacroOutput**
A decorator to replace a method with a macro output connector.

Macro connectors are useful, when a processing network shall be encapsulated in a class. In such a case, macro output connectors are used to export output connectors from the internal processing network to be accessible as connectors of the encapsulating class's instance.

The decorated method must not take any parameters and return the output connector from the internal processing network, that it exports.

The resulting connector will behave like an output connector, which is very different from the decorated method:

- the connector takes no argument.
- when called, the connector returns the result of the exported connector.

class connectors.**MacroInput**
A decorator to replace a method with a macro input connector.

Macro connectors are useful, when a processing network shall be encapsulated in a class. In such a case, macro input connectors are used to export input connectors from the internal processing network to be accessible as connectors of the encapsulating class's instance.

The decorated method must not take any parameters and yield all input connectors from the internal processing network, that it exports. Exporting multiple connectors through the same macro connector is possible and useful, when all of these connectors shall always receive the same input value.

The resulting connector will behave like an input connector, which is very different from the decorated method:

- the connector's arguments are passed to all the exported input connectors, when it is called.
- when called, the connector returns the instance to which it belongs, so that changing a parameter and retrieving a result in one line is possible
- when a behavior (e.g. laziness) of the connector is changed, the change is passed on to all the exported connectors.

Configuring macro connectors

Instances of the following classes replace the decorated methods, so they are enhanced with the functionality of a macro connector.

class connectors.connectors.**MacroOutputConnector**(*instance*, *method*)

A Connector-class that exports an output connector from an internal processing network to the API of the class, that encapsulates the network.

Parameters

- **instance** – the instance in which the method is replaced by this connector
- **method** – the unbound method, that is replaced by this connector

connect(*connector*)

Connects the exported output connector to the given input.

Parameters **connector** – the input connector to which the exported connector shall be connected

Returns the instance of which this connector has replaced a method

disconnect(*connector*)

Disconnects the exported output connector from the given input.

Parameters **connector** – the input connector from which the exported connector shall be disconnected

Returns the instance of which this connector has replaced a method

set_caching(*caching*)

Specifies, if the result value of this output connector shall be cached. If caching is enabled and the result value is retrieved (e.g. through a connection or by calling the connector), the cached value is returned and the replaced getter method is not called unless the result value has to be re-computed, because an observed setter method has changed a parameter for the computation. In this case, the getter method is only called once, independent of the number of connections through which the result value has to be passed.

Parameters **caching** – True, if caching shall be enabled, False otherwise

set_executor(*executor*)

Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors

in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_parallelization(*parallelization*)
Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

class connectors.connectors.**MacroInputConnector**(*instance*, *method*)
A Connector-class that exports input connectors from an internal processing network to the API of the class, that encapsulates the network.

Parameters

- **instance** – the instance in which the method is replaced by this connector
- **method** – the unbound method, that is replaced by this connector

connect(*connector*)
Connects all exported input connectors to the given output.

Parameters **connector** – the output connector to which the exported connectors shall be connected

Returns the instance of which this connector has replaced a method

disconnect(*connector*)
Disconnects all exported input connectors from the given output.

Parameters **connector** – the output connector from which the exported connectors shall be disconnected

Returns the instance of which this connector has replaced a method

set_executor(*executor*)
Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_laziness(*laziness*)
Configures the lazy execution of the connector. Normally the connectors are executed lazily, which means, that any computation is only started, when the result of a processing chain is requested. For certain use cases it is necessary to disable this lazy execution, though, so that the values are updated immediately as soon as new data is available. There are different behaviors for the (non) lazy execution, which are described in the *connectors.Laziness* enum.

Parameters **laziness** – a flag from the *connectors.Laziness* enum

set_parallelization(*parallelization*)
Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

1.1.6 Processing blocks for common tasks

This is the API reference for processing blocks, that help with common tasks, when constructing a processing network.

Routing data in a processing network

class connectors.blocks.**PassThrough**(*data=None*)
A trivial processing block, that simply passes its input value to its output. Instances of this can be useful to distribute a single parameter to multiple inputs, if this parameter is used in several places in a processing chain.

Parameters **data** – the input object

output()
Returns the object, that has been passed with the *input()* method.

Returns the given object

input(*data*)
Specifies the input object.

Parameters **data** – the input object

Returns the *PassThrough* instance

class connectors.blocks.**Multiplexer**(*selector=None*)
A class, that routes one of arbitrarily many inputs to its output.

Usage Example: (the assignments `_ = ...` are only to make `doctest` ignore the return values of the operations.)

```
>>> import connectors
>>> # Create some test objects
>>> test1 = connectors.blocks.PassThrough(data="One")
>>> test2 = connectors.blocks.PassThrough(data="Two")
>>> multiplexer = connectors.blocks.Multiplexer()
>>> # Connect the test objects to the multiplexer
>>> # Note, how the input of the input connector of the multiplexer is accessed␣
↪like a dictionary
>>> _ = multiplexer.input["1"].connect(test1.output)
>>> _ = test2.output.connect(multiplexer.input[2])     # the order, which is␣
↪connected to which is not important
>>> # Select the output with the same selector, that has been passed as key,␣
↪during connecting
>>> _ = multiplexer.select("1")
>>> multiplexer.output()
'One'
>>> _ = multiplexer.select(2)
```

(continues on next page)

(continued from previous page)

```
>>> multiplexer.output()
'Two'
```

Parameters **selector** – the selector of the input, that shall be routed to the output

output()
Returns the value from the selected input. If no data is found for the given selector, the first input data set, that was added, is returned.

Returns the selected input object

input(*data*)
Specifies an input object, that can be selected to be routed to the output. When connecting to this method, the selector, by which the given connection can be selected, can be specified with the *__getitem__()* overload:

```
multiplexer.input[selector].connect(generator.output())
```

Parameters **data** – the input object

Returns an ID, that can be used as a selector value t

remove(*data_id*)
Is used to remove an input object from the collection, when the respective connector is disconnected from the input.

Parameters **data_id** – the data_id under which the input object is stored

Returns the Multiplexer instance

replace(*data_id*, *data*)
Is used to replace an input object, when the respective connected connector has produced a new one.

Parameters

- **data_id** – the data_id under which the input object is stored
- **data** – the new input object

Returns data_id, because the ID does not change for the replaced data

Reducing the memory consumption

class connectors.blocks.**WeakrefProxyGenerator**(*data=None*)
A helper class to reduce memory usage in certain processing chains by discarding intermediate results. It takes an object as input and outputs a weak reference proxy to it. The deletion of the strong reference to the input object can triggered by connecting the output connector for the final result to this class's *delete_reference()* method. This will cause the input object to be garbage collected, if no further references to it exist. This class should be used in combination with the caching of the final result, since this prevents, that the deleted input data is required to re-compute the result, when it's retrieved repeatedly.

Parameters **data** – the input object

output()
Creates and returns a weak reference proxy to the input object. As long as the object has not been garbage collected, this proxy should behave exactly as the input object.

Returns a weak reference proxy to the input object, if it can be weakly referenced, otherwise the object itself

input(*data*)
Specifies the input object.

Parameters **data** – the input object

Returns the WeakrefProxyGenerator instance

delete_reference(**args*, ***kwargs*)
Causes the strong reference to the input object to be deleted, so that the input object can be garbage collected. This connector should be connected to the output of the final result, so it is notified, when the result has been computed and the input object is no longer required.

Parameters ***args,**kwargs** – these parameters just there fore compatibility with other input connectors and are not used in this method

Returns the WeakrefProxyGenerator instance

1.1.7 Internal features

This section contains the API reference for the internal features of the *Connectors* package. It contains implementation details, that are shown here for completeness and reference.

Common helper classes

This section contains helper classes, that are used internally in the *Connectors* package.

Container classes

class connectors._common._non_lazy_inputs.**NonLazyInputs**(*situation*)
A subclass of `set`, that is used internally to track the non-lazy input connectors, that request an immediate re-computation of the processing chain.

Parameters **situation** – a flag from the *Laziness* enumeration to which the laziness of the connectors is compared in order to decide, if it shall be added to this `set`.

add(*connector*, *laziness*)
Adds a connector to this container, if its laziness is low enough to cause immediate execution.

Parameters

- **connector** – the *InputConnector* instance, that shall be added
- **laziness** – the laziness setting of that connector as a flag from the *Laziness* enumeration

execute(*executor*)
Executes the necessary computations, that are requested by the non-lazy input connectors.

Parameters **executor** – the *connectors._common._executors.Executor* instance, that manages the executions

Supplementary classes

class connectors._common._multiinput_associate.**MultiInputAssociateDescriptor**(*method*, *observers*, *executor*)

A descriptor class for associating remove and replace methods with their multi-input. Instances of this class are created by the decorator methods *remove()* and *replace()*.

Parameters

- **method** – the unbound method, that is wrapped
- **observers** – the names of output methods that are affected by passing a value to the multi-input connector.
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *MultiInputConnector*'s *set_executor()* method for details

class connectors._common._multiinput_associate.**MultiInputAssociateProxy**(*instance*, *method*, *observers*, *executor*)

A proxy class for remove or replace methods of multi-input connectors. Connector proxies are returned by the connector decorators, while the methods are replaced by the actual connectors. Think of a connector proxy like of a bound method, which is also created freshly, whenever a method is accessed. The actual connector only has a weak reference to its instance, while this proxy has a hard reference, but mimics the connector in almost every other way. This makes constructions like `value = Class().connector()` possible. Without the proxy, the instance of the class would be deleted before the connector method is called, so that the weak reference of the connector would be expired during its call.

Parameters

- **instance** – the instance in which the method is replaced by the multi-input connector proxy
- **method** – the unbound method, that is replaced by this proxy (the remove or replace method)
- **observers** – the names of output methods that are affected by passing a value to the multi-input connector proxy
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details

Connector base classes

This section contains documentation of the base classes of the *connector* classes.

class connectors.connectors.**Connector**(*instance*, *method*, *parallelization*, *executor*)

Base class for connectors. Connectors are objects that replace methods of a class so that they can be connected to each other. This way changing data at one end of a connection chain will automatically cause the data at the other end of the chain to be updated, when that data is retrieved the next time.

Parameters

- **instance** – the instance of which the method is replaced by this connector
- **method** – the unbound method that is replaced by this connector
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details

connect(*connector*)

Abstract method that defines the interface of a *Connector* for connecting it with other connectors.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *Connector* has replaced a method

disconnect(*connector*)

Abstract method that defines the interface of a *Connector* for disconnecting it from a connector, to which it is currently connected.

Parameters **connector** – a *Connector* instance from which this connector shall be disconnected

Returns the instance of which this *Connector* has replaced a method

set_executor(*executor*)

Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_parallelization(*parallelization*)

Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

class connectors._connectors._baseclasses.**InputConnector**(*instance*, *method*, *laziness*, *parallelization*, *executor*)

Base class for input connectors, that replace setter methods.

Parameters

- **instance** – the instance of which the method is replaced by this connector
- **method** – the unbound method that is replaced by this connector
- **laziness** – a flag from the *connectors.Laziness* enum. See the *set_laziness()* method for details

- **parallelization** – a flag from the *Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details

connect(*connector*)
Connects this *InputConnector* to an output.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *InputConnector* has replaced a method

disconnect(*connector*)
Disconnects this *InputConnector* from an output, to which is has been connected.

Parameters **connector** – a *Connector* instance from which this connector shall be disconnected

Returns the instance of which this *Connector* has replaced a method

set_executor(*executor*)
Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_laziness(*laziness*)
Configures the lazy execution of the connector. Normally the connectors are executed lazily, which means, that any computation is only started, when the result of a processing chain is requested. For certain use cases it is necessary to disable this lazy execution, though, so that the values are updated immediately as soon as new data is available. There are different behaviors for the (non) lazy execution, which are described in the *connectors.Laziness* enum.

Parameters **laziness** – a flag from the *connectors.Laziness* enum

set_parallelization(*parallelization*)
Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

Executors

Executors control, if a connector is executed sequentially, in a parallel thread or even a separate process. Use the *connectors.executor()* function to instantiate an executor.

Base class

class connectors._common._executors.**Executor**
a base class for managing the event loop and the execution in threads or processes.

get_event_loop()
Returns the currently active event loop. This can be None, if no coroutine, task or future is currently being processed.

Returns the event loop or None

run_coroutine(*coro*)
Takes a coroutine and runs it in a newly created event loop.

Parameters **coro** – the coroutine

Returns the return value of the coroutine

run_coroutines(*coros*)
Takes multiple coroutines and runs them in a newly created event loop.

Parameters **coros** – a sequence of coroutines

run_method(*parallelization*, *method*, *instance*, **args*, ***kwargs*)
Abstract method, whose overrides shall execute the given method. The parallelization shall be implemented in this method.

Parameters

- **parallelization** – a flag of *connectors.Parallelization*, that specifies how the given method can be parallelized
- **method** – the unbound method, that shall be executed
- **instance** – the instance of which the method shall be executed
- ***args,**kwargs** – arguments for the method

Returns the return value of the method

run_until_complete(*future*)
Takes a future or a task and runs it in a newly created event loop. This is a wrapper for the event loop's run_until_complete() method.

Parameters **future** – the future or the task

Returns the return value of the execution

Executor classes

class connectors._common._executors.**SequentialExecutor**
An executor class, that executes everything sequentially.

get_event_loop()
Returns the currently active event loop. This can be None, if no coroutine, task or future is currently being processed.

Returns the event loop or None

run_coroutine(*coro*)
Takes a coroutine and runs it in a newly created event loop.

Parameters **coro** – the coroutine

Returns the return value of the coroutine

run_coroutines(*coros*)
Takes multiple coroutines and runs them in a newly created event loop.

Parameters **coros** – a sequence of coroutines

run_method(*parallelization*, *method*, *instance*, **args*, ***kwargs*)
Executes the given method sequentially.

Parameters

- **parallelization** – a flag of *connectors.Parallelization*, that specifies how the given method can be parallelized
- **method** – the unbound method, that shall be executed
- **instance** – the instance of which the method shall be executed
- ***args,**kwargs** – arguments for the method

Returns the return value of the method

run_until_complete(*future*)
Takes a future or a task and runs it in a newly created event loop. This is a wrapper for the event loop's `run_until_complete()` method.

Parameters **future** – the future or the task

Returns the return value of the execution

class connectors._common._executors.**ThreadingExecutor**(*number_of_threads*)
An executor class, that can parallelize computations with threads.

Parameters **number_of_threads** – the maximum number of threads, that shall be created, or None to determine this number automatically.

get_event_loop()
Returns the currently active event loop. This can be None, if no coroutine, task or future is currently being processed.

Returns the event loop or None

run_coroutine(*coro*)
Takes a coroutine and runs it in a newly created event loop.

Parameters **coro** – the coroutine

Returns the return value of the coroutine

run_coroutines(*coros*)
Takes multiple coroutines and runs them in a newly created event loop.

Parameters **coros** – a sequence of coroutines

run_method(*parallelization*, *method*, *instance*, **args*, ***kwargs*)
Executes the given method in a thread if possible and falls back to sequential execution if not.

Parameters

- **parallelization** – a flag of *connectors.Parallelization*, that specifies how the given method can be parallelized
- **method** – the unbound method, that shall be executed
- **instance** – the instance of which the method shall be executed

- ***args, **kwargs** – arguments for the method

Returns the return value of the method

run_until_complete(*future*)
Takes a future or a task and runs it in a newly created event loop. This is a wrapper for the event loop's run_until_complete() method.

Parameters **future** – the future or the task

Returns the return value of the execution

class connectors._common._executors.**MultiprocessingExecutor**(*number_of_processes*)
An executor class, that can parallelize computations with processes.

Parameters **number_of_processes** – the maximum number of processes, that shall be created, or None to determine this number automatically (in this case, the number of CPU cores will be taken).

get_event_loop()
Returns the currently active event loop. This can be None, if no coroutine, task or future is currently being processed.

Returns the event loop or None

run_coroutine(*coro*)
Takes a coroutine and runs it in a newly created event loop.

Parameters **coro** – the coroutine

Returns the return value of the coroutine

run_coroutines(*coros*)
Takes multiple coroutines and runs them in a newly created event loop.

Parameters **coros** – a sequence of coroutines

run_method(*parallelization*, *method*, *instance*, **args*, ***kwargs*)
Executes the given method in a process if possible and falls back to sequential execution if not.

Parameters

- **parallelization** – a flag of *connectors.Parallelization*, that specifies how the given method can be parallelized
- **method** – the unbound method, that shall be executed
- **instance** – the instance of which the method shall be executed
- ***args, **kwargs** – arguments for the method

Returns the return value of the method

run_until_complete(*future*)
Takes a future or a task and runs it in a newly created event loop. This is a wrapper for the event loop's run_until_complete() method.

Parameters **future** – the future or the task

Returns the return value of the execution

class connectors._common._executors.**ThreadingMultiprocessingExecutor**(*number_of_threads*, *number_of_processes*)
An executor class, that can parallelize computations with both threads and processes.

Parameters

- **number_of_threads** – the maximum number of threads, that shall be created, or None to determine this number automatically.
- **number_of_processes** – the maximum number of processes, that shall be created, or None to determine this number automatically (in this case, the number of CPU cores will be taken).

get_event_loop()
Returns the currently active event loop. This can be None, if no coroutine, task or future is currently being processed.

Returns the event loop or None

run_coroutine(*coro*)
Takes a coroutine and runs it in a newly created event loop.

Parameters **coro** – the coroutine

Returns the return value of the coroutine

run_coroutines(*coros*)
Takes multiple coroutines and runs them in a newly created event loop.

Parameters **coros** – a sequence of coroutines

run_method(*parallelization*, *method*, *instance*, **args*, ***kwargs*)
Executes the given method in a process if possible and falls back to threaded and then sequential execution if not.

Parameters

- **parallelization** – a flag of *connectors.Parallelization*, that specifies how the given method can be parallelized
- **method** – the unbound method, that shall be executed
- **instance** – the instance of which the method shall be executed
- ***args,**kwargs** – arguments for the method

Returns the return value of the method

run_until_complete(*future*)
Takes a future or a task and runs it in a newly created event loop. This is a wrapper for the event loop's run_until_complete() method.

Parameters **future** – the future or the task

Returns the return value of the execution

Proxy classes

This section contains the documentation of the connector proxy classes. The proxies emulate the behavior of the connectors and are used to avoid *circular references*. See the *decorators* on how to create connectors and the *connectors* themselves on how to use and configure them.

Base class

class connectors._proxies._baseclasses.**ConnectorProxy**(*instance*, *method*, *parallelization*, *executor*)
A base class for proxy objects of connectors. Connector proxies are returned by the connector decorators, while

the methods are replaced by the actual connectors. Think of a connector proxy like of a bound method, which is also created freshly, whenever a method is accessed. The actual connector only has a weak reference to its instance, while this proxy has a hard reference, but mimics the connector in almost every other way. This makes constructions like `value = Class().connector()` possible. Without the proxy, the instance of the class would be deleted before the connector method is called, so that the weak reference of the connector would be expired during its call.

Parameters

- **instance** – the instance in which the method is replaced by this connector proxy
- **method** – the unbound method that is replaced by this connector proxy
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *set_executor()* method for details

connect(*connector*)

Connects this connector with another one.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *Connector* has replaced a method

set_executor(*executor*)

Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_parallelization(*parallelization*)

Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

Proxy classes

class connectors._proxies.**OutputProxy**(*instance*, *method*, *caching*, *parallelization*, *executor*)

A proxy class for output connectors. Connector proxies are returned by the connector decorators, while the methods are replaced by the actual connectors. Think of a connector proxy like of a bound method, which is also created freshly, whenever a method is accessed. The actual connector only has a weak reference to its instance, while this proxy has a hard reference, but mimics the connector in almost every other way. This makes constructions like `value = Class().connector()` possible. Without the proxy, the instance of the class would be deleted before the connector method is called, so that the weak reference of the connector would be expired during its call.

Parameters

- **instance** – the instance in which the method is replaced by this connector proxy
- **method** – the unbound method that is replaced by this connector proxy
- **caching** – True, if caching shall be enabled, False otherwise. See the *set_caching()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *OutputConnector*'s *set_executor()* method for details

connect(*connector*)

Connects this connector with another one.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *Connector* has replaced a method

set_caching(*caching*)

Specifies, if the result value of this output connector shall be cached. If caching is enabled and the result value is retrieved (e.g. through a connection or by calling the connector), the cached value is returned and the replaced getter method is not called unless the result value has to be re-computed, because an observed setter method has changed a parameter for the computation. In this case, the getter method is only called once, independent of the number of connections through which the result value has to be passed.

Parameters **caching** – True, if caching shall be enabled, False otherwise

set_executor(*executor*)

Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_parallelization(*parallelization*)

Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

class connectors._proxies.**SingleInputProxy**(*instance*, *method*, *observers*, *announce_condition*, *notify_condition*, *laziness*, *parallelization*, *executor*)

A proxy class for input connectors. Connector proxies are returned by the connector decorators, while the methods are replaced by the actual connectors. Think of a connector proxy like of a bound method, which is also created freshly, whenever a method is accessed. The actual connector only has a weak reference to its instance, while this proxy has a hard reference, but mimics the connector in almost every other way. This makes constructions like `value = Class().connector()` possible. Without the proxy, the instance of the class

would be deleted before the connector method is called, so that the weak reference of the connector would be expired during its call.

Parameters

- **instance** – the instance in which the method is replaced by this connector proxy
- **method** – the unbound method that is replaced by this connector proxy
- **observers** – the names of output methods that are affected by passing a value to this connector proxy
- **announce_condition** – a method, that defines the condition for the announcements to the observing output connectors. This method must not require any arguments apart from `self`
- **notify_condition** – a method, that defines the condition for the notifications to the observing output connectors. This method must accept the new input value as an argument in addition to `self`
- **laziness** – a flag from the *connectors.Laziness* enum. See the *set_laziness()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *SingleInputConnector*'s *set_executor()* method for details

connect(*connector*)

Connects this connector with another one.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *Connector* has replaced a method

set_executor(*executor*)

Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_laziness(*laziness*)

Configures the lazy execution of the connector. Normally the connectors are executed lazily, which means, that any computation is only started, when the result of a processing chain is requested. For certain use cases it is necessary to disable this lazy execution, though, so that the values are updated immediately as soon as new data is available. There are different behaviors for the (non) lazy execution, which are described in the *connectors.Laziness* enum.

Parameters **laziness** – a flag from the *connectors.Laziness* enum

set_parallelization(*parallelization*)

Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g.

if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

class connectors._proxies.**MultiInputProxy**(*instance*, *method*, *remove_method*, *replace_method*, *observers*, *announce_condition*, *notify_condition*, *laziness*, *parallelization*, *executor*)

A proxy class for multi-input connectors. Connector proxies are returned by the connector decorators, while the methods are replaced by the actual connectors. Think of a connector proxy like of a bound method, which is also created freshly, whenever a method is accessed. The actual connector only has a weak reference to its instance, while this proxy has a hard reference, but mimics the connector in almost every other way. This makes constructions like `value = Class().connector()` possible. Without the proxy, the instance of the class would be deleted before the connector method is called, so that the weak reference of the connector would be expired during its call.

Parameters

- **instance** – the instance in which the method is replaced by this connector proxy
- **method** – the unbound method, that is replaced by this connector proxy
- **remove_method** – an unbound method, that is used to remove data, that has been added through this connector proxy
- **replace_method** – an unbound method, that is used to replace data, that has been added through this connector proxy
- **observers** – the names of output methods that are affected by passing a value to this connector proxy
- **announce_condition** – a method, that defines the condition for the announcements to the observing output connectors. This method must accept the data ID of the changed output connector as an argument in addition to `self`
- **notify_condition** – a method, that defines the condition for the notifications to the observing output connectors. This method must accept the data ID and the new input value as an argument in addition to `self`
- **laziness** – a flag from the *connectors.Laziness* enum. See the *set_laziness()* method for details
- **parallelization** – a flag from the *connectors.Parallelization* enum. See the *set_parallelization()* method for details
- **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function. See the *MultiInputConnector*'s *set_executor()* method for details

connect(*connector*)

Connects this connector with another one.

Parameters **connector** – the *Connector* instance to which this connector shall be connected

Returns the instance of which this *Connector* has replaced a method

set_executor(*executor*)

Sets the executor, which handles the computations, when the data is retrieved through this connector. An executor can be created with the *connectors.executor()* function. It manages the order and the parallelization of the computations, when updating the data in a processing chain. If multiple connectors

in a processing chain need to be computed, the executor of the connector, which started the computations, is used for all computations.

Parameters **executor** – an *Executor* instance, that can be created with the *connectors.executor()* function

set_laziness(*laziness*)
Configures the lazy execution of the connector. Normally the connectors are executed lazily, which means, that any computation is only started, when the result of a processing chain is requested. For certain use cases it is necessary to disable this lazy execution, though, so that the values are updated immediately as soon as new data is available. There are different behaviors for the (non) lazy execution, which are described in the *connectors.Laziness* enum.

Parameters **laziness** – a flag from the *connectors.Laziness* enum

set_parallelization(*parallelization*)
Specifies, if and how the execution of this connector can be parallelized. The choices are no parallelization, the execution in a separate thread and the execution in a separate process. This method specifies a hint, which level of parallelization is possible with the connector. If the executor of the connector, through which the computation is started, does not support the specified level, the next simpler one will be chosen. E.g. if a connector can be parallelized in a separate process, but the executor only allows threads or sequential execution, the connector will be executed in a separate thread.

Parameters **parallelization** – a flag from the *connectors.Parallelization* enum

Virtual single-connectors

This section contains helper classes, that are returned by the `[]` operator of *MultiInputConnector* and *MultiOutputConnector* and simulate single input- and output-connectors.

class connectors._common._multiinput_item.**MultiInputItem**(*connector*, *instance*, *replace_method*, *key*, *observers*, *executor*)
An object, that is returned by the *__getitem__()* overload.

It simulates the behavior of a single-input connector, so it is possible to use a multi-input connector as arbitrarily many single-inputs.

Parameters

- **connector** – the multi-input connector
- **instance** – the instance of which the method was replaced by the multi-input connector
- **replace_method** – an unbound method, that is used to replace data, that has been added through the multi-input connector
- **key** – the key with which the multi-input has been accessed.
- **observers** – a sequence of output connectors, that observe the multi-input connector's value changes.
- **executor** – an *Executor* instance, that is used, when calling the instance of this class.

connect(*connector*)
Connects this virtual single-input to an output.

Parameters **connector** – the connector, to which this connector shall be connected

Returns the instance of which the method was replaced by the multi-input connector

disconnect(*connector*)
Disconnects this virtual single-input from an output, to which is has been connected..

Parameters **connector** – the connector, from which this connector shall be disconnected

Returns the instance of which the method was replaced by the multi-input connector

class connectors._common._multioutput_item.**MultiOutputItem**(*connector*, *instance*, *key*)
An object, that is returned by the *__getitem__()* overload.

It simulates the behavior of a single-output connector, so it is possible to use a multi-output connector as arbitrarily many single-outputs.

Parameters

- **connector** – the multi-output connector
- **instance** – the instance of which the method was replaced by the multi-output connector
- **key** – the key with which the multi-output has been accessed.

connect(*connector*)
Connects this virtual single-output to an output.

Parameters **connector** – the connector, to which this connector shall be connected

Returns the instance of which the method was replaced by the multi-output connector

disconnect(*connector*)
Disconnects this virtual single-output from an input, to which is has been connected..

Parameters **connector** – the connector, from which this connector shall be disconnected

Returns the instance of which the method was replaced by the multi-output connector

key()
Returns the key, with which the multi-output connector has been accessed.

1.2 Organisation

This section contains organisational information and instructions for the installation of the *Connectors* package.

1.2.1 Installation

pip

The easiest way to install the *Connectors* package is probably through `pip`.

```
pip3 install connectors
```

If the package shall only be installed for the current user, rather than system wide, run the following command:

```
pip3 install --user connectors
```

Installation from source

To retrieve the sources, the `git`-repository of the *Connectors* package has to be cloned.

```
git clone https://github.com/JonasSC/Connectors.git Connectors
```

The `Connectors` at the end of this command specifies the directory, in which the local copy of the repository shall be created. After cloning, move to that directory.

```
cd Connectors
```

Now the *Connectors* package can be installed system wide with the following command:

```
python3 setup.py install
```

Alternatively, the package can be installed only for the current user.

```
python3 setup.py install --user
```

1.2.2 Dependencies

Python version

The *Connectors* package is currently developed and tested with Python 3.7. Python 3.6 is very likely to work as well. Earlier versions are not compatible with the *Connectors* package.

Other packages and tools

The *Connectors* package itself does not depend on other packages. Nevertheless, the setup, the tests and the documentation rely on third party packages and tools.

- the setup is done with `setuptools`.
- **the tests are run with `pytest`.**
  - the test coverage is assessed with `pytest-cov`.
  - the code is analyzed with *Pylint* and `flake8`.
  - spell-checking is done with *pyenchant*.
- **the documentation is built with `sphinx`.**
  - the documentation uses the `sphinx_rtd_theme`.
  - graphs are drawn with `sphinx.ext.graphviz`.
  - some examples rely on `numpy` and `matplotlib`.

1.2.3 Makefile targets

The Makefile in the source code repository of the *Connectors* package has the following targets:

- `make test` runs the unit tests.
- `make test_coverage` runs the unit tests and prints information about their test coverage.
- `make lint` checks the package and the unit tests with *Pylint*

- `make docs` builds the documentation

1.2.4 Licenses

LGPLv3+ for the source code

The source code of the *Connectors* package can be distributed and modified under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. A copy of this license can be found in the source code repository in the file `LICENSE.txt` or on the website of the GNU project.

CC0 for the documentation

The documentation for the *Connectors* package and the source code snippets in it can be distributed and modified under the terms of the CC0 license as published by the Creative Commons Corporation. This means, that the documentation and the source code snippets are practically in the public domain. The full text of the license can be found on the website of the Creative Commons Corporation.

It would be great, if the redistributed pieces of this work were marked with a reference to this project, but this is by no means mandatory.

1.3 Information

This section contains background information and implementation details about the *Connectors* package.

1.3.1 Lazy execution

By default, the connectors are executed lazily, which means, that input values are not propagated through a processing network, unless an output value, which depends on them, is requested. This is an important design decision, not only, because it saves potentially unnecessary computations, but also, because the correct input data might not be available at the time of setting up the processing network. In this case, the processing objects still have their default values, which might lead to incompatible results and subsequently errors, when executing computations in a processing network without correct input.

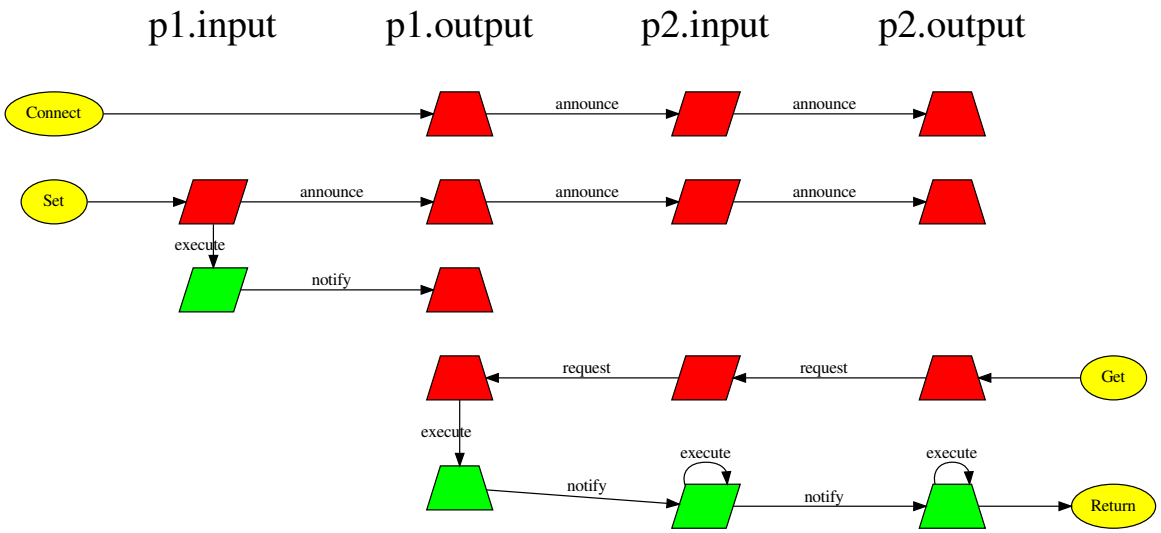
An example script

The following example shows, what happens, when a processing chain is set up, when a value is changed and when a value is retrieved. It consists of two trivial processors, that simply pass their input value to their output.

```
>>> import connectors
>>> p1 = connectors.blocks.PassThrough()
>>> p2 = connectors.blocks.PassThrough().input.connect(p1.output)  # (Connect)
>>> _ = p1.input("data")                                          # (Set)
>>> p2.output()                                                   # (Get)
'data'
```

The following graph illustrates the communication between the input and output connectors of `p1` and `p2`. The columns in this graph correspond to the connectors, while the rows are in the order, in which the depicted events occur. The yellow ellipses stand for the commands, from the example script above. The symbols, that represent the

connectors, are either green, after the respective method has been executed, or red, if there are pending value changes, that require an execution of the method, in order to compute the current output value.



This graph shows the behavior of the lazily executed connectors:

- When establishing a connection, the value change is announced to all connectors down the processing chain.
- When calling an input connector, the value change is announced, too. Additionally, the corresponding setter method is executed and the observing output connectors (which belong to the same object) are notified, that the setter has been executed. This means, that the output connectors do not have to request the setters execution, when they themselves receive a request to be executed.
- When an output connector is called, it requests the upstream connectors to be executed and waits for the corresponding notifications, before it executes its own getter method and returns the result. It happens only in this step, that the methods of the connectors are executed.

Disabling lazy execution

Input connectors can be configured to automatically request the data of connected outputs by passing a flag from the `connectors.Laziness` enum to its `set_laziness()` method.

An example for the necessity of eager execution would be a plot, which shall automatically update, whenever new data can be computed. In order to avoid the aforementioned problem of propagating default values, it is recommended to implement such classes with lazy execution enabled and disable the lazy execution as soon as the processing network has been intitialized with correct data.

1.3.2 Caching

By default, output connectors cache the return values of their wrapped getter methods. This shall avoid unnecessary recomputations, when the output connectors are called multiple times.

The caching can only work well, when all setters, that affect the return value of an output connector, are decorated as input connectors, which are observed by that output connector.

If this is not the case, the caching can be disabled by passing `False` to an output connector's *`set_caching()`* method. Alternatively, the default setting for caching the result of a particular method can be changed by passing `caching=False` to the *`Output`* decorator of the method.

1.3.3 Automated parallelization

If a processing network is branched, so that different operations can be executed in parallel, the *Connectors* package has the functionality to do so. This document provides some background information on this topic, since the distinction between a connector's *parallelization* parameter and its executor might be counter-intuitive.

The default settings

By default, the execution of output connectors are parallelized in different threads, while input connectors are executed sequentially. This choice has been made, because the complex computations are usually done in the getter methods, while the setters often only store parameters, which is such a short operation, that it would be slowed down by the overhead of a parallelization.

Preferring threads over processes in the default setting is justified by the lower overhead of threads and fewer constraints (e.g. pickle-ability of objects, that are passed between processes). Also, many libraries such as `numpy` release the GIL in many of their functions, so that they do run concurrently, when using threads.

The *parallelization* parameter

The *parallelization* parameter defines, which methods of parallelization are allowed for the given connector.

The *parallelization* parameter is meant to be configured in the decorator, when implementing processing classes, but connectors also provide a *`set_parallelization()`* method to configure the parallelization of individual instances. It must be set to a flag of the *`connectors.Parallelization`* enum.

- Enforcing a sequential execution is useful for reducing the overhead, when doing short operations. Sometimes it is even necessary to disable the parallelization, like for example in GUI applications, where the drawing operations must be executed in the same thread.
- As explained above, threaded parallelization is often a good compromise.
- Using processes for the parallelization requires both the processing class and the data, that is passed through its connectors, to be pickle-able. Also, the overhead of starting an operation in a separate process and retrieving its result is much higher than with using threads, so process-based parallelization is only worth the effort for long running computations. Nevertheless, it is often recommended to allow the use of processes with the *parallelization* parameter, when implementing a processing class, if the constraints on pickle-ability are met. The actual parallelization method can later be chosen after setting up a processing network for a specific application, by specifying the executor of the connector, that triggers the computations. At this stage, the lengths of the computations can often be estimated better, than during the implementation of the classes.

Executors

If an operation actually is parallelized as allowed by its *parallelization* parameter, is decided by the executor of the connector, that triggers the computations.

Executors are created with the factory function *`connectors.executor()`*, which takes the maximum number of threads and processes for the parallelization as parameters. Think of executors as wrappers around the `concurrent.futures.ThreadPoolExecutor` and `concurrent.futures.ProcessPoolExecutor` classes. They check, how a connector is allowed to be parallelized and then execute it accordingly:

- If a connector is allowed to be parallelized in a more complex way than the executor is capable of, the next simpler method for parallelization, that is available is used. The complexity hierarchy for this decision in descending order is process-based parallelization, separate threads and as last resort sequential execution.
- If a connector can be executed in a separate thread/process and the executor provides this functionality, the connector will never be executed in the main thread/process.

Only one executor is used for all computations, that are required for a requested result. This is usually the executor of the output connector, through which the result is retrieved. With non-lazy connectors, that request results immediately, when a parameter is set, the executor of the input connector for that parameter is used. So changing the executors of connectors in the middle of a processing chain usually has no effect.

1.3.4 Implementation details

asyncio

The *Connectors* package uses `asyncio` to model the dependencies between the connectors and schedule their execution. The event loop is started by the connector, which triggers the computations and ends, when that connector's computation has finished.

Connector proxies - avoiding circular references

Note: Circular references occur, when objects have references to each other, so that their reference count never reaches zero, even when there is no reference to the objects in the active code. In such a case, the objects are not automatically deleted by Python's reference counting mechanism.

Python uses *bound methods* and *unbound methods* to avoid circular references, in which an object has references to its methods, while the methods have a reference to the object. Unbound methods are basically functions, which require the object to be passed as the first parameter, which is commonly called *self*. An object has a reference to its class, which has references to its unbound methods, but the unbound methods have no reference to the object. When a method is accessed, Python automatically creates a bound method by predefining the first parameter with the object. Obviously, this causes bound methods to have a reference to the object, but since the object does not store references to bound methods, circular references do not occur.

Unbound methods are attributes of a class, which cannot store information about individual objects. Connectors on the other hand must be able to store such information, like established connections or configurations about their behavior. This requires the connectors to be persistent, unlike bound methods, which are created freshly, whenever they are required. Also, in order to execute the method, which they have replaced, connectors have a reference to the object, to which they belong, which leads to circular references.

In the *Connectors* package, this problem is addressed by using weak references for the connector's reference to their object. This alone is not sufficient though, since this would break the functionality to call a connector immediately after instantiating an object:

```
result = ExampleClass().connector()
```

In the above example, an object of `ExampleClass` is created and and its connector `connector()` is accessed. Since no reference of the object is stored, except for the weak reference of the connector, the object will be garbage collected after accessing the connector, but before executing it. So the connector cannot be executed.

To prevent objects from being garbage collected prematurely, the *Connectors* package uses a mechanism similar to the bound and unbound methods from Python. As long as no individual information has to be stored in the connector, the method is not replaced by a connector object. Instead a *`ConnectorProxy`* object is created each time, when the

connector is accessed. Similar to bound methods, the connector proxy has a reference to the object, but not vice versa. Only when establishing a connection or when changing the configuration, the method is replaced by a connector.

1.4 Tutorials

This section contains tutorials, which demonstrate and explain the functionalities of the *Connectors* package.

1.4.1 Measuring a transfer function *(demonstrates the core functionalities)*

This tutorial walks through a simple example of determining a transfer function with processing objects, that are connected with the functionalities of the *Connectors* package.

What is a transfer function

A transfer function describes, how a linear, time-invariant system amplifies and delays the frequency components of its input signal. Examples of such systems are equalizers of HiFi systems, which allow to tweak the system's sound by boosting or attenuating certain frequency regions. Or radio tuners, which supress all frequencies except for the one of the channel, that shall be received. The reflections and reverberations of a concert hall, which a listener experiences when attending an event there, can also be described by a transfer function.

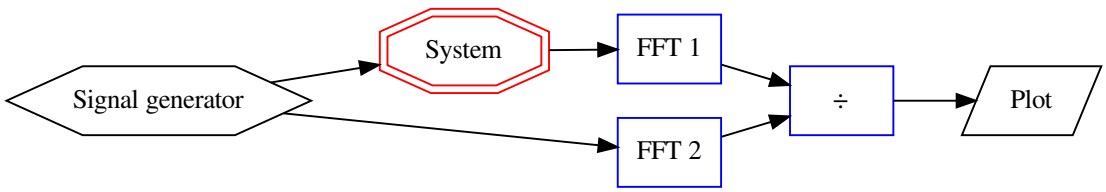
The transfer function of a system can be measured by sending a known input signal into the system and recording its response. After that, the spectrum of this response has to be divided by the spectrum of the input signal. Think of this as of compensating for a bias of the input signal, which might excite certain frequencies at a higher level than others. If this is the case, the recorded response will also have an exaggerated amount of these frequency components, which is not due them being boosted by the system, but due to a biased excitation. The divison normalizes the response by attenuating the frequency components, that had been exaggerated in the excitation signal.

Of course, the exctiation signal has to excite all frequencies, at which the system shall be modeled. Otherwise, the division will divide by zero and the resulting transfer function will be invalid at the non-excited frequencies.

For more background on transfer functions, you can read the following Wikipedia articles (sorted in increasing order of theoret

- Frequency response
- Transfer function
- Linear time-invariant theory

The following block diagram shows the computation steps for determining the transfer function of a system:



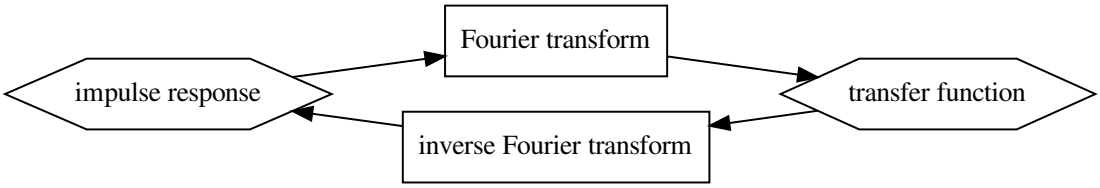
First, the excitation signal is created in *Signal generator*. In this tutorial, a linear sweep is used, which is a sine wave, which continuously increases its frequency over time, thus exciting all the frequencies at which the system shall be modeled. The sweep is used, because it is mathematically well defined, easy to implement and for demonstrating what

will happen, if the frequency range of the excitation signal is limited. For the purpose of measuring a transfer function, other signals such as noise or maximum length sequences are also suitable.

The generated excitation signal is fed into the *System*. The spectrum of the system's response is computed by transforming the response to the frequency domain with the help of the fast fourier transform in block *FFT 1*. Meanwhile, the spectrum of the excitation signal is computed by the block *FFT 2*. The resulting transfer function is computed by the division ÷ and displayed by the *Plot*.

Defining a system, of which the tranfer function shall be measured

For reducing the lines of code for this tutorial, the system, that shall be analyzed by measuring its transfer function, is modeled with its impulse response. The impulse response is mathematically connected to the transfer function by the (inverse) fourier transform.



The following code implements a class for defining a linear time-invariant system. It requires an impulse response as constructor parameter, in order to define the system's behavior. Furthermore it has a setter and a getter method for passing the excitation signal and retrieving the response.

```
class LinearSystem:
    def __init__(self, impulse_response):
        self.__impulse_response = impulse_response
        self.__input = None

    @connectors.Input("get_output")
    def set_input(self, signal):
        self.__input = signal

    @connectors.Output()
    def get_output(self):
        return numpy.convolve(self.__input, self.__impulse_response, mode="full
↪")[0:len(self.__input)]
```

The setter method `set_input()` is decorated to become an input connector, so that the output of the generator of the excitation signal can be connected to it. Note that the name of the getter method `get_output()` is passed as a parameter to the input decorator. This models the dependency of the getter's return value on whether the setter has been called. So whenever a new value is passed to the setter, the getter is notified, that it can produce a new result.

The getter method `get_output()` is decorated to become an output connector. Note that this is the method, that actually does the expensive computation, while the setter only stores the received parameter. This is a recommended practice when using the *Connectors* package, since the lazy execution and the caching capability of the output connectors can avoid, that these computations are performed unnecessarily.

Generation of a measurement signal

Using the *Connectors* package with the generator class for the linear sweep is straight forward. In production code, all parameters, that can be passed to the constructor, should have a setter method, that is decorated to become an input connector. This has been omitted in this tutorial to keep the code short.

```
class SweepGenerator:
    def __init__(self, start_frequency=20.0, stop_frequency=20000.0, length=2 ** 16):
        self.__start_frequency = start_frequency
        self.__stop_frequency = stop_frequency
        self.__length = length

    @connectors.Input("get_sweep")
    def set_start_frequency(self, frequency):
        self.__start_frequency = frequency

    @connectors.Output()
    def get_sweep(self):
        f0 = self.__start_frequency
        fT = self.__stop_frequency
        T = self.__length / sampling_rate                # the duration of the signal
        t = numpy.arange(0.0, T, 1.0 / sampling_rate)    # an array with the time␣
↪samples
        k = (fT - f0) / T                                # the "sweep rate"
        return numpy.sin(2.0 * math.pi * f0 * t + math.pi * k * (t ** 2))
```

Computation of the fourier transform

Decorating the methods of the class for the fourier transform works just like in the previous classes. But the deletion of the input signal in the getter method `get_spectrum()` is noteworthy.

```
class FourierTransform:
    def __init__(self, signal=None):
        self.__signal = signal

    @connectors.Input("get_spectrum")
    def set_signal(self, signal):
        self.__signal = signal

    @connectors.Output()
    def get_spectrum(self):
        spectrum = numpy.fft.rfft(self.__signal)
        self.__signal = None
        return spectrum
```

Since the input signal is the only parameter for the fourier transform, the reference to it can be deleted after computing the output spectrum, as long as the caching of the output spectrum is enabled. The only situation, in which the cached spectrum becomes invalid and the output spectrum has to be recomputed, is when a new input signal is provided. So the old input signal is no longer needed after the computation.

In this example, the input signals for the two fourier transform classes would not be garbage collected, because they are cached in the outputs of the signal generator and the system under test. The memory requirements for running the script of this tutorial are moderate, so that the code has not been optimized for minimal memory consumption by deactivating caching and other measures. In some practical situations, these optimizations can reduce the memory consumption significantly.

Computation of the transfer function

The class, that computes the transfer function by dividing the response spectrum by the excitation spectrum is again straight forward. The only difference, that has not been shown in previous classes is, that an output connector depends on the parameters of multiple input connectors. Each of these receives the name of the dependent output connector as a parameter for the *Input* decorator.

```
class TransferFunction:
    def __init__(self, excitation=None, response=None):
        self.__excitation = excitation
        self.__response = response

    @connectors.Input("get_transfer_function")
    def set_excitation(self, signal):
        self.__excitation = signal

    @connectors.Input("get_transfer_function")
    def set_response(self, signal):
        self.__response = signal

    @connectors.Output()
    def get_transfer_function(self):
        return numpy.divide(self.__response, self.__excitation)
```

Ploting the transfer function

For the sake of simplicity, the plotting class in this tutorial only plots the magnitude of the transfer function. Plotting the phase aswell, requires some additional functionalities of `matplotlib`, which is not in the scope of this tutorial.

The plotting class demonstrates the use of a multi-input connector for plotting multiple spectrums in one plot.

```
class MagnitudePlot:
    def __init__(self):
        self.__spectrums = connectors.MultiInputData()

    @connectors.MultiInput("show")
    def add_spectrum(self, spectrum):
        return self.__spectrums.add(spectrum)

    @add_spectrum.remove
    def remove_spectrum(self, data_id):
        del self.__spectrums[data_id]

    @add_spectrum.replace
    def replace_spectrum(self, data_id, spectrum):
        self.__spectrums[data_id] = spectrum

    @connectors.Output(parallelization=connectors.Parallelization.SEQUENTIAL)
    def show(self):
        for d in self.__spectrums:
            x_axis_data = numpy.linspace(0.0, sampling_rate / 2.0, len(self.__
↪spectrums[d]))
            magnitude = numpy.abs(self.__spectrums[d])
            pyplot.plot(x_axis_data, magnitude)
        pyplot.loglog()
        pyplot.xlim(20.0, sampling_rate / 2.0)
```

(continues on next page)

(continued from previous page)

```
        pyplot.grid(b=True, which="both")
        pyplot.show()
```

Decorating the `add_spectrum()` method to become a multi-input connector is similar to the regular input connectors. It also gets the name of the dependent output connectors passed as a parameter. It is improtant though, that the method of multi-input returns an ID, with which the added dataset can be identified, when it shall be deleted or replaced.

Specifying a remove-method for a multi-input connector is mandatory. This method is called whenever a dataset is removed, for example by disconnecting an output connector from the multi-input. Notice that the remove-method `remove_spectrum()` is decorated with a method of the multi-input connector instead of an object from the *Connectors* package.

The replace-method `replace_spectrum()` of the multi-input connector is called, whenever an added spectrum shall be replaced by an updated version. If none is specified, the replacement will be done by removing the old dataset and adding a new one, which does not preserve the order, in which the datasets have been added.

The spectrums, that are added through the `add_spectrum()` method are managed by a *`MultiInputData`* container. This is basically an `OrderedDict`, that has been extended with an *`add()`* method, which adds the given dataset to the dictionary and returns a unique ID, under which the dataset has been stored.

The `show()` method is decorated to become an output connector, despite the fact that it does not return any result value. Nevertheless, this allows to model, that showing the plot depends on the input data for the plot.

Note that the automated parallelization is disabled for this method by passing the flag *`Parallelization`*.`SEQUENTIAL` as the *parallelization* parameter for the output decorator. By default, the *Connectors* package parallelizes independent computations in separate threads. Process-based parallelization is also available, but this requires the data, that is passed through the connections, to be pickle-able and the pickling introduces additional overhead. GUI functionalities often require, that all updates of the GUI are done by the same thread, which is why this example script will raise errors if the parallelization of the `show()` method is not disabled.

Instantiating the processing network

Now that all the necessary processing classes are implemented, the network for measuring and computing the transfer function can be set up.

First the linear, time-invariant system is instantiated. The impulse response is chosen to have a rolloff at both high and low frequencies.

```
    impulse_response = numpy.zeros(2 ** 16)
    impulse_response[0:3] = (-1.0, 0.0, 1.0)
    system = LinearSystem(impulse_response)
```

After that, the sweep generator is created and connected to the system, that shall be measured. The connection is established by calling the *`connect()`* method of the output connector `get_sweep()` with the input connector `set_input()` from the system. The *`connect()`* method returns the instance, to which the connector belongs. This way, the instantiation of a processing class and the connection of one of its connectors can be done in one line, like this example shows.

```
    sweep = SweepGenerator().get_sweep.connect(system.set_input)
```

Instantiating the fourier transform classes is straight forward now. Note, that this time, the *`connect()`* method of the input connectors are called with an output connector as a parameter, while it is the other way around, during the

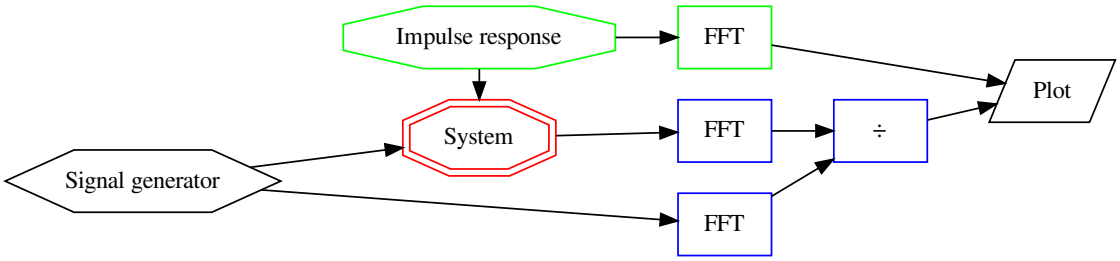
instantiation of the sweep generator. Both ways are possible.

```
excitation_fft = FourierTransform().set_signal.connect(sweep.get_sweep)
response_fft = FourierTransform().set_signal.connect(system.get_output)
```

The class for dividing the two spectrums is created without connecting any of its connectors in the same line. Since two of its connectors have to be connected, the connections are established in separate but similar lines, which improves the readability of the code.

```
transfer_function = TransferFunction()
transfer_function.set_excitation.connect(excitation_fft.get_spectrum)
transfer_function.set_response.connect(response_fft.get_spectrum)
```

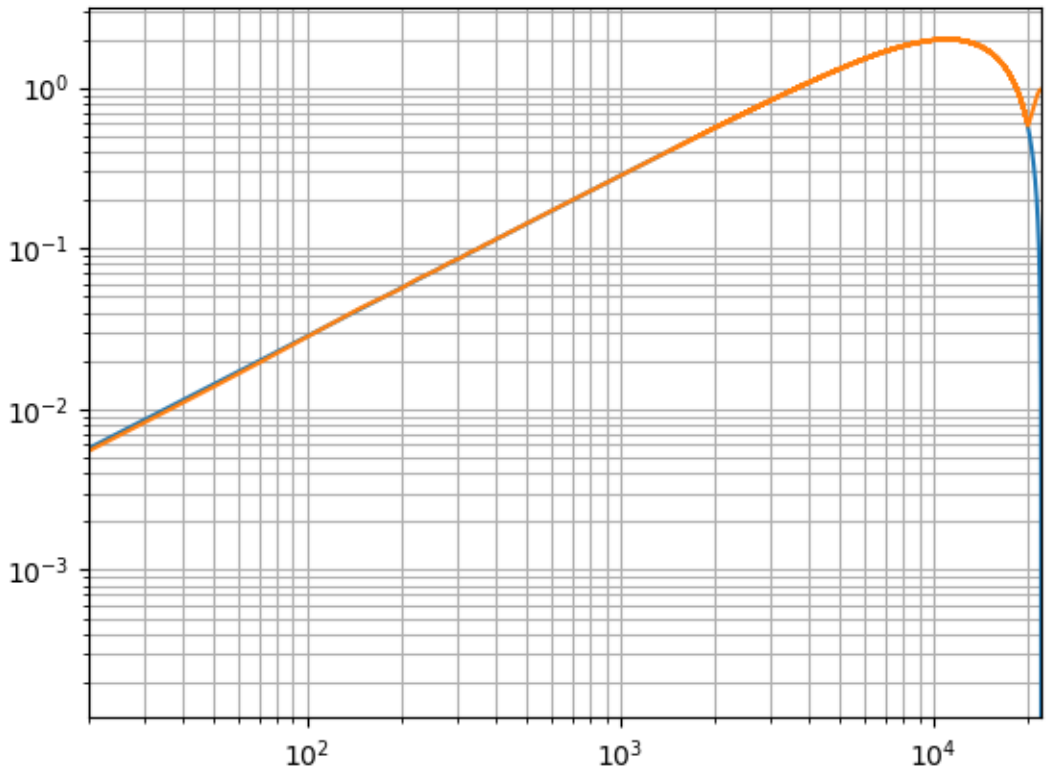
Finally, the plot is created and shown. In addition to the measured transfer function, the plot also shows the spectrum of the system's impulse response, so it can be seen how the measured transfer function deviates from the expected spectrum.



Adding the measured transfer function is done through connections, just like the other connections, that have been established before. It is noteworthy though, how the spectrum of the impulse response is added by simply calling the respective methods and without relying on the functionality of the *Connectors* package. This shows, that the decorated methods can still be used as normal methods, even when they are extended with the functionality of a connector.

```
magnitude_plot = MagnitudePlot()
magnitude_plot.add_spectrum(FourierTransform(impulse_response).get_spectrum())
transfer_function.get_transfer_function.connect(magnitude_plot.add_spectrum)
magnitude_plot.show()
```

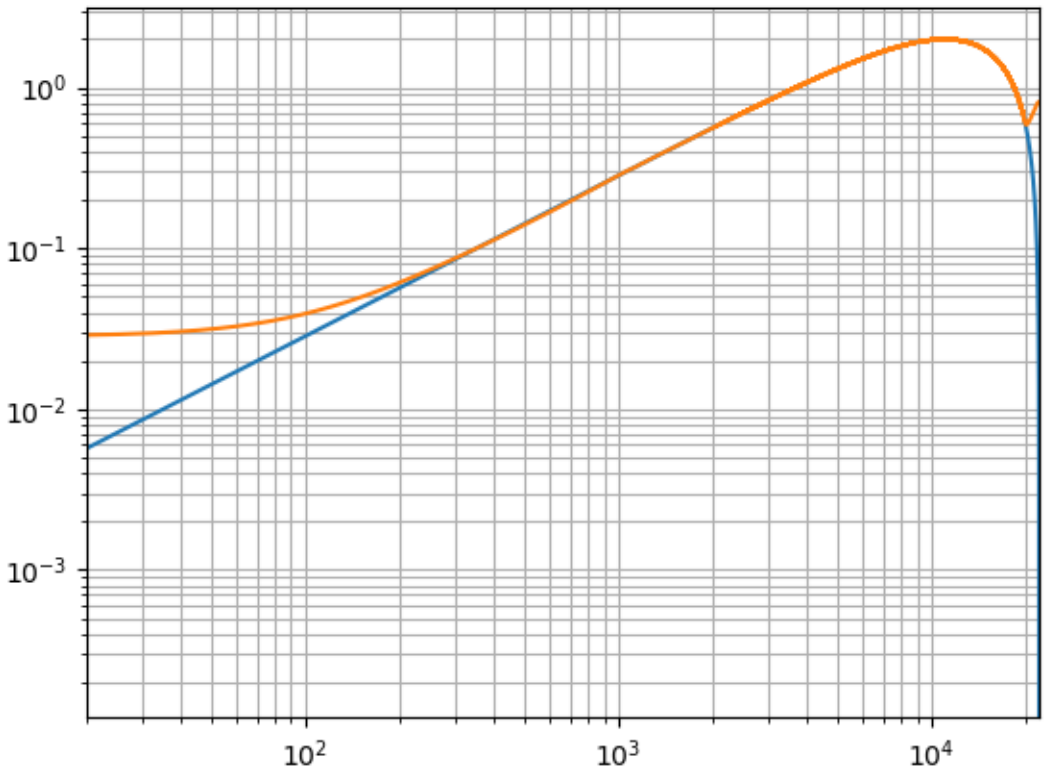
This results in the following plot. The measured transfer function matches well with the spectrum of the original impulse response, but especially at low frequencies, there are slight deviations. Above 20kHz, the two measured frequency response becomes highly inaccurate, which is because the sweep has not excited these frequencies, so the computation of the transfer function becomes a division by zero.



To demonstrate the automated updating of data in a processing network, the start frequency of the sweep is changed and the plot is shown again.

```
sweep.set_start_frequency(1000.0)
magnitude_plot.show()
```

The following plot shows the effect of raising the start frequency of the sweep to a value in the plotted frequency range. Since the low frequencies are no longer properly excited, the measurement of the tranfer function is invalid here aswell.



The complete script

```
import math
import numpy
import connectors
from matplotlib import pyplot

sampling_rate = 44100.0


class LinearSystem:
    def __init__(self, impulse_response):
        self.__impulse_response = impulse_response
        self.__input = None

    @connectors.Input("get_output")
    def set_input(self, signal):
        self.__input = signal

    @connectors.Output()
    def get_output(self):
        return numpy.convolve(self.__input, self.__impulse_response, mode="full
↪")[0:len(self.__input)]
```

(continues on next page)

(continued from previous page)

```
class SweepGenerator:
    def __init__(self, start_frequency=20.0, stop_frequency=20000.0, length=2 ** 16):
        self.__start_frequency = start_frequency
        self.__stop_frequency = stop_frequency
        self.__length = length

    @connectors.Input("get_sweep")
    def set_start_frequency(self, frequency):
        self.__start_frequency = frequency

    @connectors.Output()
    def get_sweep(self):
        f0 = self.__start_frequency
        fT = self.__stop_frequency
        T = self.__length / sampling_rate                # the duration of the signal
        t = numpy.arange(0.0, T, 1.0 / sampling_rate)    # an array with the time␣
↪samples
        k = (fT - f0) / T                                # the "sweep rate"
        return numpy.sin(2.0 * math.pi * f0 * t + math.pi * k * (t ** 2))


class FourierTransform:
    def __init__(self, signal=None):
        self.__signal = signal

    @connectors.Input("get_spectrum")
    def set_signal(self, signal):
        self.__signal = signal

    @connectors.Output()
    def get_spectrum(self):
        spectrum = numpy.fft.rfft(self.__signal)
        self.__signal = None
        return spectrum


class TransferFunction:
    def __init__(self, excitation=None, response=None):
        self.__excitation = excitation
        self.__response = response

    @connectors.Input("get_transfer_function")
    def set_excitation(self, signal):
        self.__excitation = signal

    @connectors.Input("get_transfer_function")
    def set_response(self, signal):
        self.__response = signal

    @connectors.Output()
    def get_transfer_function(self):
        return numpy.divide(self.__response, self.__excitation)


class MagnitudePlot:
    def __init__(self):
```

(continues on next page)

(continued from previous page)

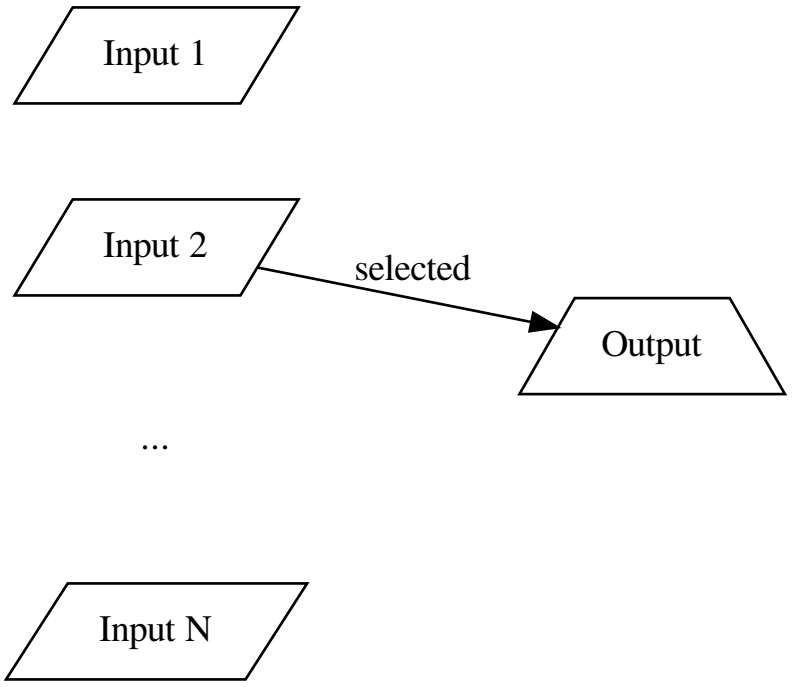
```
        self.__spectrums = connectors.MultiInputData()

    @connectors.MultiInput("show")
    def add_spectrum(self, spectrum):
        return self.__spectrums.add(spectrum)

    @add_spectrum.remove
    def remove_spectrum(self, data_id):
        del self.__spectrums[data_id]

    @add_spectrum.replace
    def replace_spectrum(self, data_id, spectrum):
        self.__spectrums[data_id] = spectrum

    @connectors.Output(parallelization=connectors.Parallelization.SEQUENTIAL)
    def show(self):
        for d in self.__spectrums:
            x_axis_data = numpy.linspace(0.0, sampling_rate / 2.0, len(self.__
↪spectrums[d]))
            magnitude = numpy.abs(self.__spectrums[d])
            pyplot.plot(x_axis_data, magnitude)
        pyplot.loglog()
        pyplot.xlim(20.0, sampling_rate / 2.0)
        pyplot.grid(b=True, which="both")
        pyplot.show()


if __name__ == "__main__":
    impulse_response = numpy.zeros(2 ** 16)
    impulse_response[0:3] = (-1.0, 0.0, 1.0)
    system = LinearSystem(impulse_response)

    sweep = SweepGenerator().get_sweep.connect(system.set_input)

    excitation_fft = FourierTransform().set_signal.connect(sweep.get_sweep)
    response_fft = FourierTransform().set_signal.connect(system.get_output)

    transfer_function = TransferFunction()
    transfer_function.set_excitation.connect(excitation_fft.get_spectrum)
    transfer_function.set_response.connect(response_fft.get_spectrum)

    magnitude_plot = MagnitudePlot()
    magnitude_plot.add_spectrum(FourierTransform(impulse_response).get_spectrum())
    transfer_function.get_transfer_function.connect(magnitude_plot.add_spectrum)
    magnitude_plot.show()

    sweep.set_start_frequency(1000.0)
    magnitude_plot.show()
```

1.4.2 Impementing a multiplexer *(demonstrates the use of a multi-input connector as arbitrarily many single-input connectors)*

This tutorial shows and explains a simple implementation of a multiplexer. With this example, the usage of a multi-input connector as arbitrarily many single-input connectors is demonstrated. A *follow-up* for this tutorial demonstrates the use of conditional input connectors, to avoid unnecessary computations.

What is a multiplexer

A multiplexer is a device with many inputs and one output, which allows to select, which one of the inputs shall be routed to the output.



A very common application for multiplexers is signal routing in electronic circuits, for which there is a huge variety of integrated circuits, such as the *74LS151* or the *CMOS 4097*. In some occasions, a multiplexer can also be helpful to implement a processing networs, which is why the *Connectors* package provides the `Multiplexer` class.

Arbitrarily many input connectors

To suit most applications, the number of inputs of the multiplexer should not be hard coded. Instead it should dynamically scale the number of input connectors. Also, the keys for selecting the input, that shall be routed to the output, should ideally be arbitrary, so the user can decide, if the keys are integers, strings or any other objects.

Theoretically, it is possible to implement such an array of an arbitrary number of input connectors by instantiating `SingleInputConnector`s dynamically. But such an implementation would require a lot of code and it would depend on implementation details of the *Connectors* package, that might change in the future.

For applications like this, a `MultiInputConnector` can be accessed with the `[]` operator, which returns a an object, that behaves like a single-input connector. This virtual input connector can be called directly or be connected to an output connector. The key, that is passed to the `[]` operator is the data ID, under which the `MultiInputConnector` stores the given input value. This allows the user to select the data ID manually, rather than having it generated by the connector, which in turn allows to use the data ID as selector for a multiplexer.

Implementation of the multiplexer

```
>>> import connectors
```

The following code shows the implementation of a multiplexer.

```
>>> class Multiplexer:
...     def __init__(self, selector=None):
...         self.__selector = selector
...         self.__data = connectors.MultiInputData()
...
...     @connectors.Output()
...     def output(self):
...         if self.__selector in self.__data:
...             return self.__data[self.__selector]
...         else:
...             return None
...
...     @connectors.Input("output")
...     def select(self, selector):
...         self.__selector = selector
...         return self
...
...     @connectors.MultiInput("output")
...     def input(self, data):
...         return self.__data.add(data)
...
...     @input.remove
...     def remove(self, data_id):
...         del self.__data[data_id]
...         return self
...
...     @input.replace
...     def replace(self, data_id, data):
...         self.__data[data_id] = data
...         return data_id
```

Note, that it is required, that the `replace()` method returns the ID, under which the new input value is stored. Apart from this, the implementation is straight forward.

- The `input()`, `remove()` and `replace()` methods implement a very common pattern for multi-input connectors, in which the input values are stored in a *`MultiInputData`* instance.
- The `select()` method is an input connector, through which the key for selecting the input, that is routed to the output.
- The `output()` method returns the value from the selected input or `None`, if the selector key is invalid.
- The `select()` and `remove()` methods return `self` to allow method chaining.

Usage of the multiplexer

Instantiating the multiplexer is done the usual way.

```
>>> multiplexer = Multiplexer()
```

When calling the input, it can be accessed with the `[]` operator to specify the selector key.

```
>>> multiplexer.input["key 1"]("value 1")
<__main__.Multiplexer object at 0x...>
>>> multiplexer.input["key 2"]("value 2")
<__main__.Multiplexer object at 0x...>
>>> multiplexer.select("key 2")
<__main__.Multiplexer object at 0x...>
>>> multiplexer.output()
'value 2'
```

Note, that the call of the virtual single-input method returns the multiplexer instance. This is an implementation choice of the *Connectors* package and cannot be influenced by how the decorated method is implemented. The idea behind this choice is, that it allows chaining the calls of the input method. Theoretically, all of the above can be written in one line:

```
>>> Multiplexer().input["key 1"]("value 1").input["key 2"]("value 2").select("key 2").
↪output()
'value 2'
```

Under the hood, the virtual single-inputs, that are created with the `[]` operator, call the `replace()` method. So the above script is equivalent to the following.

```
>>> multiplexer.replace("key 1", "value 1")
'key 1'
>>> multiplexer.replace("key 2", "value 2")
'key 2'
>>> _ = multiplexer.select("key 2")
>>> multiplexer.output()
'value 2'
```

The `input()` method can also be called like an ordinary multi-input connector. In this case, the returned data ID must be stored in a variable, so it can be used as selector key.

```
>>> key1 = multiplexer.input("value 1")
>>> key2 = multiplexer.input("value 2")
>>> _ = multiplexer.select(key2)
>>> multiplexer.output()
'value 2'
```

The latter two approaches do not work in the context of connecting an output connector to one of the inputs of the multiplexer. The `replace()` method does not accept connections, while when using the `input()` method the usual way, the data ID is unknown to the user, so it cannot be used as a selector key. Therefore, connections to the multiplexer have to use the virtual single-inputs from the `[]` operator.

```
>>> data_source = connectors.blocks.PassThrough("value 3 (value from the data source)
↪")
>>> _ = data_source.output.connect(multiplexer.input["key 3"])
>>> _ = multiplexer.select("key 3")
>>> multiplexer.output()
'value 3 (value from the data source)'
```

Restrictions and requirements for virtual single-input connectors

When the `[]` operator calls the `replace` method of the multi-input connector, it is possible, that the data ID, which is passed to the method, does not exist, yet. Therefore, the `replace` methods of multi-input connectors, that shall be used as virtual single-inputs, must be able to hanlde unknown data IDs in a reasonable manner. This is usually the case, when the input data is managed by dictionaries like a *`MultiInputData`* instance.

For ordinary multi-input connectors, it is optional to specify a `replace` method. If none is specified, replacing data is done with the `remove` method and the decorated input method. This will obviously not work with the `[]` operator, because calling the decorated input method will generate a new data ID, that is not known outside the class.

When managing the input data of a multi-input connector with dictionaries like a *`MultiInputData`* instance, the data IDs must be hashable. Therefore it is not possible to use mutable objects like `list` instances as selector keys for this `Multiplexer`.

1.4.3 Improving the multiplexer *(demonstrates avoiding unneccessary computations with conditional input connectors)*

The multiplexer from the *previous tutorial* can cause unnecessary computations in certain constellations. This tutorial shows how to avoid these computations by specifying conditions for the propagation of value changes of the input connector.

Situations, in which unnecessary computations occur

If any input of the simple multiplexer from the *previous tutorial* receives value update, this update will be propagated down the processing chain. In case, the updated input is not currently selected, the output of the multiplexer will produce the same value as before the value update, which causes an unnecessary recomputation of the processing chain.

In order to avoid these unnecessary computations, a means to interrupt the processing of the chain is required. The (multi-) input connectors have a feature to specify conditions on the propagation of value changes, which can be used for this purpose.

Conditions for the input connectors

Inputs and multi-inputs have two decorators for methods, which specify the conditional propagation of value changes. For a more detailed explanation of the *announcement* and *notification* phases of the propagation of value changes, it is recommended read the section about *lazy execution*.

- The method decorated with *`announce_condition`* is evaluated to check if the announcement of a value change shall be propagated. If the condition evaluates to `False`, processors further down the processing chain will not be informed about the pending value change, which means, that they will not request this value change to be performed. In case all end points, which request this value update, (such as manually called output connectors or non-lazy inputs) are behind the conditional input in the processing chain, this means, that also the connectors, which are before the conditional input, are not executed.
- The method decorated with *`notify_condition`* is evaluated after executing the input connector to check if the observing output connectors shall be notified about the changed value. If this evaluation yields `False`, the pending announcements are canceled, so that downstream connectors do not request an updated value.

Implementing a conditional multi-input connector for the multiplexer

Choosing condition for the multiplexer's input is trivial. It should simply check, if the changed input is the one, that is currently selected. The harder choice is to decide whether to use an *`announce_condition`* or a *`notify_condition`*.

At first glance, the *`announce_condition`* is tempting, because it also avoids the computations, that produce the input value, which is not selected by the multiplexer. Sadly, these computations cannot generally be avoided, because it is always possible, that the changed value is selected by the multiplexer at a later point in time. In this case, the

output connector of the multiplexer must have been informed about the pending value change, in order to request that value to be updated. And this announcement has not been sent, if the *announce_condition* evaluated to `False`.

Therefore, the multiplexer's input must specify a *notify_condition*.

An improved implementation of the multiplexer

```
>>> import connectors
```

The following implementation of the improved multiplexer is almost identical to the `Multiplexer` class from the *previous tutorial*. It is only enhanced by the `__input_condition()` method, which is decorated to become the `input()` method's *notify_condition*.

```
>>> class Multiplexer:
...     def __init__(self, selector=None):
...         self.__selector = selector
...         self.__data = connectors.MultiInputData()
...
...     @connectors.Output()
...     def output(self):
...         if self.__selector in self.__data:
...             return self.__data[self.__selector]
...         else:
...             return None
...
...     @connectors.Input("output")
...     def select(self, selector):
...         self.__selector = selector
...         return self
...
...     @connectors.MultiInput("output")
...     def input(self, data):
...         return self.__data.add(data)
...
...     @input.remove
...     def remove(self, data_id):
...         del self.__data[data_id]
...         return self
...
...     @input.replace
...     def replace(self, data_id, data):
...         self.__data[data_id] = data
...         return data_id
...
...     @input.notify_condition
...     def __input_condition(self, data_id, value):
...        return data_id == self.__selector
```

In order to test and demonstrate the avoidance of unnecessary computations, the following test class is implemented:

```
>>> class Tester:
...     @connectors.Input(laziness=connectors.Laziness.ON_ANNOUNCE)
...     def input(self, value):
...         print("Tester received value:", repr(value))
```

It has a non-lazy input, which requests the updated value as soon as an update is announced. And whenever it receives a new value, it prints a message.

In the following test set up, two *PassThrough* instances are connected to the inputs of a multiplexer, while a Tester instance is connected to its output. It is now expected, that the tester prints a message, whenever the selected input of the multiplexer changes its value, while it remains silent, when there is a value change in a not-selected input.

```
>>> source1 = connectors.blocks.PassThrough("value 1")
>>> source2 = connectors.blocks.PassThrough("value 2")
>>> multiplexer = Multiplexer()
>>> tester = Tester()
>>>
>>> _ = source1.output.connect(multiplexer.input[1])
>>> _ = source2.output.connect(multiplexer.input[2])
>>> _ = multiplexer.output.connect(tester.input)
```

Of course, selecting an input causes the output to be updated, so a message from the tester is expected.

```
>>> _ = multiplexer.select(1)
Tester received value: 'value 1'
```

When input *1* is selected, a change of that input's value shall also trigger a message from the tester.

```
>>> _ = source1.input("new value 1")
Tester received value: 'new value 1'
```

But since input *2* is not selected, the tester is not invoked when the value of that input is updated.

```
>>> _ = source2.input("new value 2")
```

1.4.4 Implementing a polynomial *(demonstrates the encapsulation of a processing network in a single class with macro connectors)*

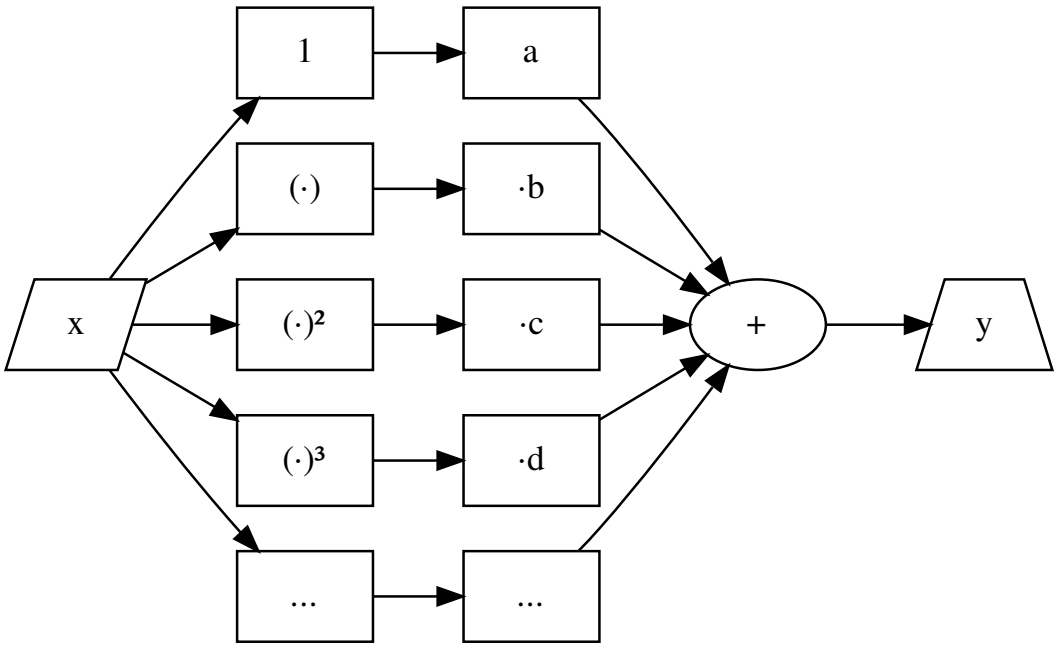
This tutorial demonstrates the use of the *MacroInput* and *MacroOutput* decorators to encapsulate the processing network for computing a polynomial in a single class.

The block diagram representation of a polynomial

A polynomial is a weighted sum of powers of its input variable:

$$y = a + bx + cx^2 + dx^3 + ...$$

This can be organized in a block diagram:



Implementing the basic building blocks: power, multiplication and summation

As seen in the block diagram, the basic building blocks can be implemented with three processing classes:

- one that computes a specified power of its input value
- one that multiplies its input value with a given factor
- one that sums up all its input values

For this task, it would be sufficient, if the exponent of the power and the weighting factor of the multiplication, were constants, that are specified through the constructor of the class. But assuming, that a project, in which polynomials are computed, would also benefit from processing classes, that compute arbitrary powers and products, the following, more general implementations are used in this tutorial.

```
>>> import numpy
>>> import connectors
```

```
>>> class Power:
...     def __init__(self, base=0, exponent=1):
...         self.__base = base
...         self.__exponent = exponent
...
...     @connectors.Output()
...     def get_result(self):
...         return numpy.power(self.__base, self.__exponent)
...
...     @connectors.Input("get_result")
```

(continues on next page)

(continued from previous page)

```
...     def set_base(self, base):
...         self.__base = base
...
...     @connectors.Input("get_result")
...     def set_exponent(self, exponent):
...         self.__exponent = exponent
```

```
>>> class Multiply:
...     def __init__(self, factor1=0, factor2=0):
...         self.__factor1 = factor1
...         self.__factor2 = factor2
...
...     @connectors.Output()
...     def get_result(self):
...         return numpy.multiply(self.__factor1, self.__factor2)
...
...     @connectors.Input("get_result")
...     def set_factor1(self, factor):
...         self.__factor1 = factor
...
...     @connectors.Input("get_result")
...     def set_factor2(self, factor):
...         self.__factor2 = factor
```

```
>>> class Sum:
...     def __init__(self):
...         self.__summands = connectors.MultiInputData()
...
...     @connectors.Output()
...     def get_result(self):
...         return sum(tuple(self.__summands.values()))
...
...     @connectors.MultiInput("get_result")
...     def add_summand(self, summand):
...         return self.__summands.add(summand)
...
...     @add_summand.remove
...     def remove_summand(self, data_id):
...         del self.__summands[data_id]
```

Implementing the polynomial

The following class implements the computation of a polynomial, by encapsulating the required processing chain and exporting the input and output connectors via macro connectors. It accepts a sequence of weighting factors (a, b, c, d, ... in the block diagram) and instantiates the required processing classes in the `for`-loop.

```
>>> class Polynomial:
...     def __init__(self, coefficients):
...         self.__powers = []
...         self.__sum = Sum()
...         for e, c in enumerate(coefficients):
...             power = Power(exponent=e)
...             weighting = Multiply(factor2=c).set_factor1.connect(power.get_result)
...             weighting.get_result.connect(self.__sum.add_summand)
```

(continues on next page)

(continued from previous page)

```
...             self.__powers.append(power)
...
...     @connectors.MacroInput()
...     def set_variable(self):
...         for p in self.__powers:
...             yield p.set_base
...
...     @connectors.MacroOutput()
...     def get_result(self):
...         return self.__sum.get_result
```

Each iteration of the `for`-loop in the constructor generates one of the parallel branches, that are shown in the block diagram. The input of each branch, which is a `set_base()` connector, is stored in the `__powers` list. These input connectors are exported to the interface of the `Polynomial` class through the `set_variable()` macro input method.

Storing the output connector of each branch is not necessary, since they are all connected to the summation block. The output of the summation is exported to the interface of the `Polynomial` class through the `get_result()` macro output method.

Note that the methods, that are decorated to become macro connectors, merely return the connectors of the internal processing chain. These methods will be replaced by macro connectors, that behave like setter or getter methods, so the behavior of macro connectors differs significantly from that of the methods, which they replace.

Using the implementation of the polynomial

The `Polynomial` class can now be instantiated and used for computations.

```
>>> polynomial = Polynomial(coefficients=(5.0, -3.0, 2.0))  # y = 2*x**2 - 3*x + 5
>>> polynomial.set_variable(4.0).get_result()               # compute the polynomial␣
↪for a scalar
25.0
>>> polynomial.set_variable([-2, -1, 0, 1, 2]).get_result() # compute the polynomial␣
↪for elements of an array
array([19., 10.,  5.,  4.,  7.])
```

Note how the `set_variable()` and `get_result()` methods work as actual setter and getter methods, rather than returning connectors and not accepting any parameters. The `get_result()` output connector of the polynomial basically mimics the `get_result()` connector of the summation. Since the macro input connector represents multiple connectors, all operations on it will be performed with each of these connectors:

- setting a value of a macro input connector, passes that value to all represented connectors.
- changing the behavior of a macro input connector applies the same changes to all represented connectors.
- connecting an output connector to a macro input connector connects that output to all represented connectors.

Also note, that macro input connectors return the instance of the processing class to which they belong, so that setting a parameter and retrieving the updated result can be programmed in one line.

1.4.5 Improving the polynomial implementation *(demonstrates memory saving techniques)*

This tutorial shows how to reduce the memory consumption of the polynomial implementation from the *previous tutorial* by disabling *caching* and using the *`WeakrefProxyGenerator`* class.

The problem of caching intermediate values

The `Power` and `Multiply` classes store references to their input parameters. Also, because these classes are meant to be useful outside the scope of the polynomial computation, the caching of their result value is not disabled. This leads to all intermediate results of the polynomial to remain in memory, even after the computation of the final result has finished. Due to the caching of the final result, the intermediate results are no longer needed, once the computation has finished.

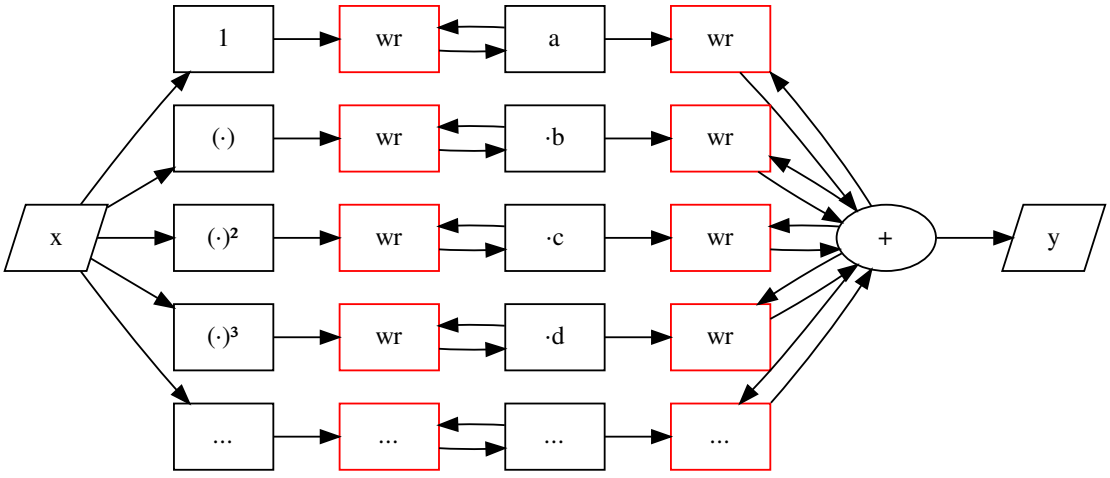
Avoiding the caching of intermediate results

The `FourierTransform` class in the *transfer function* tutorial has solved the issue of storing its input value, by deleting it in the getter method. For classes like this, it would be sufficient to disable the *caching* of the output value, to avoid that intermediate results are stored. But this solution is only applicable to classes with no more than one input and one output connector, which is not the case with the `Power` and `Multiply` classes.

To solve this problem in use cases like this, the *Connectors* package provides the *`WeakrefProxyGenerator`* class, that stores a strong reference to its input value, propagates a weak reference to it through its output connector. In order to delete the strong reference, once it is no longer needed, this class also provides an input connector, that deletes the strong reference, once the result of the following processing step has been computed. In combination with disabling the *caching* of the output connector, that produced the input value for the *`WeakrefProxyGenerator`* instance, this causes the input value to be garbage collected.

Block diagram of the improved polynomial implementation

The block diagram of a polynomial implementation, that uses *`WeakrefProxyGenerator`*s, is shown below. The *`WeakrefProxyGenerator`*s are highlighted in red. Note the backwards dependencies of the *`WeakrefProxyGenerator`*s on the output of processing classes, by which they are followed. This is a feedback loop to tell the *`WeakrefProxyGenerator`*s, that they can delete the strong reference to their input values.



Implementation of the improved polynomial

First, the building blocks of the polynomial have to be defined. They are identical to the ones from the *previous tutorial* (and they are only shown here, so the implementation of the improved polynomial can be tested with `doctest`).

```
>>> import numpy
>>> import connectors
>>>
>>> class Power:
...     def __init__(self, base=0, exponent=1):
...         self.__base = base
...         self.__exponent = exponent
...
...     @connectors.Output()
...     def get_result(self):
...         return numpy.power(self.__base, self.__exponent)
...
...     @connectors.Input("get_result")
...     def set_base(self, base):
...         self.__base = base
...
...     @connectors.Input("get_result")
...     def set_exponent(self, exponent):
...         self.__exponent = exponent
>>>
>>> class Multiply:
...     def __init__(self, factor1=0, factor2=0):
...         self.__factor1 = factor1
...         self.__factor2 = factor2
...
...     @connectors.Output()
...     def get_result(self):
...         return numpy.multiply(self.__factor1, self.__factor2)
...
...     @connectors.Input("get_result")
...     def set_factor1(self, factor):
...         self.__factor1 = factor
...
...     @connectors.Input("get_result")
...     def set_factor2(self, factor):
...         self.__factor2 = factor
>>>
>>> class Sum:
...     def __init__(self):
...         self.__summands = connectors.MultiInputData()
...
...     @connectors.Output()
...     def get_result(self):
...         return sum(tuple(self.__summands.values()))
...
...     @connectors.MultiInput("get_result")
...     def add_summand(self, summand):
...         return self.__summands.add(summand)
...
...     @add_summand.remove
...     def remove_summand(self, data_id):
...         del self.__summands[data_id]
```

The implementation of the `Polynomial` class is conceptually similar to that from the *previous tutorial*. But it contains extra lines of code for disabling the *caching* of the output connectors and for inserting the *`WeakrefProxyGenerator`* instances in the processing chain.

```
>>> class Polynomial:
...     def __init__(self, coefficients):
...         self.__powers = []
...         self.__sum = Sum()
...         for e, c in enumerate(coefficients):
...             power = Power(exponent=e)
...             self.__powers.append(power)
...             power.get_result.set_caching(False)
...             power_weakref = connectors.blocks.WeakrefProxyGenerator().input.
↪connect(power.get_result)
...             weighting = Multiply(factor2=c).set_factor1.connect(power_weakref.
↪output)
...             weighting.get_result.set_caching(False)
...             weighting.get_result.connect(power_weakref.delete_reference)
...             weighting_weakref = connectors.blocks.WeakrefProxyGenerator().input.
↪connect(weighting.get_result)
...             weighting_weakref.output.connect(self.__sum.add_summand)
...             self.__sum.get_result.connect(weighting_weakref.delete_reference)
...
...     @connectors.MacroInput()
...     def set_variable(self):
...         for p in self.__powers:
...             yield p.set_base
...
...     @connectors.MacroOutput()
...     def get_result(self):
...         return self.__sum.get_result
```

Using the implementation of the polynomial

The usage of the `Polynomial` is identical to that from the *previous tutorial*.

```
>>> polynomial = Polynomial(coefficients=(5.0, -3.0, 2.0))  # y = 2*x**2 - 3*x + 5
>>> polynomial.set_variable(4.0).get_result()               # compute the polynomial␣
↪for a scalar
25.0
>>> polynomial.set_variable([-2, -1, 0, 1, 2]).get_result() # compute the polynomial␣
↪for elements of an array
array([19., 10.,  5.,  4.,  7.])
```

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

C

Index

Symbols

A

C

D

S

T

W