chainlet Documentation

Release 1.3.1

Max Fischer

Jun 12, 2018

Documentation Topics Overview:

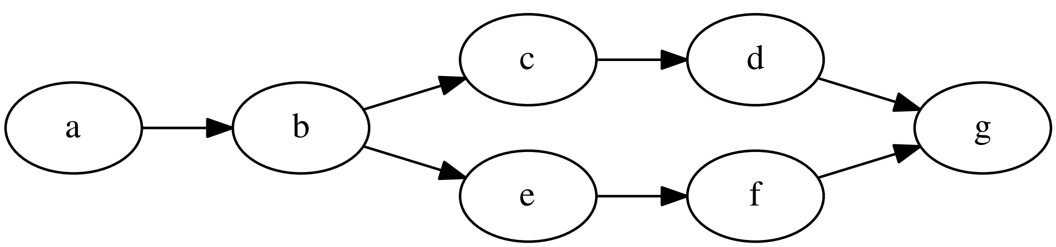
CHAPTER 1

Chainlet Mini Language

Linking chainlets can be done using a simple grammar based on >> and << operators[1]. These are used to create a directed connection between nodes. You can even include forks and joins easily.

```
a >> b >> (c >> d, e >> f) >> g
```

This example links elements to form a directed graph:



1.1 Basic Links

Linking is based on a few, fundamental primitives. Combining them allows for complex data flows from simple building blocks.

[1] These are the `__rshift__` and `__lshift__` operators. Overwriting these operators on objects changes their linking behaviour.

1.1.1 Single Link - Pairs

The most fundamental operation is the directed link between parent and child. The direction of the link is defined by the direction of the operator.

```
parent >> child
child << parent
```

This creates and returns a *chain* linking parent and child.

1.1.2 Chained Link - Flat Chains

A pair can be linked again to extend the *chain*. Adding a parent to a *chain* links it to the initial parent, while a new child is linked to the initial child. Note that *chains* preserve only *logical*, but not *syntactic* orientation: a >>-linked chain can be extended via << and vice versa.

```
chain_a = parent >> child
chain_b = chain_a << parent2
chain_c = chain_b >> child2
```

Links can be chained directly; there is no need to store intermediate subchains if you do not use them.

```
chain_c = parent2 >> parent >> child >> child2
```

The above examples create the same underlying links between objects.

Chains represent only the link they have been created with. Subsequent changes and links are not propagated. Each of the objects `chain_a`, `chain_b` and `chain_c` represent another part of the chain.

```
chain_d = parent2 >> parent >> child >> child2
#                    \-- chain_a --/
#          \------------- chain_b --/
#          \------------- chain_c ------------/
```

> **note** Linking automatically flattens *chains* to create the longest possible *chain*. This preserves equality but not identity of sub-chains. This is similar to using the + operator on a `list`.

Links follow standard operation order, i.e. they are evaluated from left to right. This can be confusing when mixing >> and << in a single chain. The following chain is equivalent to `chain_c`.

```
chain_d = child << parent >> child2 << parent2
```

> **danger** Mixing << and >> is generally a bad idea. The use of >> is suggested, as it conforms to public and private interface implementations.

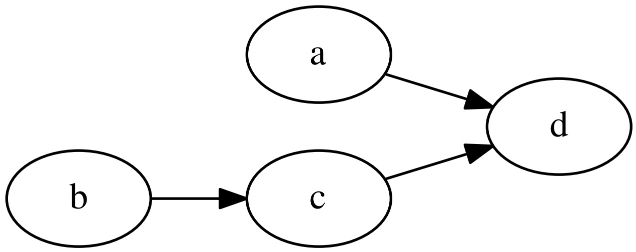
1.1.3 Forking and Joining Links - Bundles

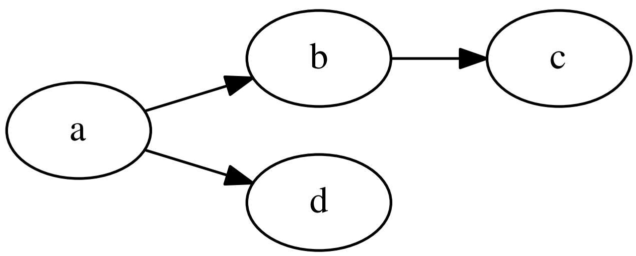
Any *chainlink* can have an arbitrary number of parents and children. This allows *forking* and *joining* the *data stream*. Simply use a `tuple()`, `list()` or `set()` as child or parent[2].

```
fork_chain = a >> (b >> c, d)
join_chain = (a, b >> c) >> d
```

[2] There may be additional implications to using different types in the future.

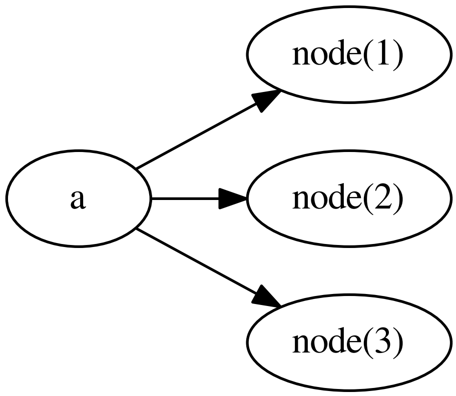
The resulting chains are actually fully featured, directed graphs.





Links are agnostic with regard to *how* a group of elements is created. This allows you to use comprehensions and calls to generate forks and joins dynamically.

```
a >> {node(idx) for idx in range(3)}
```



note A `tuple()`, `list()` or `set()` is not by itself a *chainlink*. It must be linked to an existing *chainlink* to trigger a conversion.

1.2 Advanced Linking Rules

Linking only guarantees element identity and a specific *data flow* graph. This reflects that some dataflows which can be realised in multiple ways. Several advanced rules allow `chainlet` to superseed the default link process.

1.2.1 Link Operator Reflection

The >> and << operators are subject to the regular operator reflection of Python[3]. In addition, there is an underlying linker which allows for similar behaviour beyond class hierarchies.

[3] If the right operand's type is a subclass of the left operand's type and that subclass provides the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

CHAPTER 2

Chainlet Data Flow

Chains created via *chainlet* have two operation modes: pulling at the end of the chain, and pushing to the top of the chain. As both modes return the result, the only difference is whether the chain is given an input.

```
chain = chainlet1 >> chainlet2 >> chainlet3
print('pull', next(chain))
print('push', chain.send('input'))
```

Data cascades through chains: output of each parent is passed to its children, which again provide output for their children. At each step, an element may inspect, transform or replace the data it receives.

The data flow is thus dictated by several primitive steps: Each individual *chainlink* processes data. Compound *chains* pass data from element to element. At *forks* and *joins*, data is split or merged to further elements.

2.1 Single Element Processing

Each element, be it a primitive *chainlet* or *compound link*, implements the generator protocol[1]. Most importantly, it allows to pull and push data from and to it:

- New data is *pulled from* an element using `next(element)`. The element may produce a new data chunk and return it.
- Existing data is *pushed to* the element using `element.send(data)`. The element may transform the data and return the result.

In accordance with the generator protocol, `next(element)` is equivalent to `element.send(None)`. Consequently, both operations are handled completely equivalently by *any* *chainlink*, even complex ones. Whether pulling, pushing or both is *sensible* depends on the use case - for example, it cannot be inferred from the interface whether a *chainlink* can operate without input.

Elements that work in pull mode can also be used in iteration. For every iteration step, the equivalent of `next(element)` is called to produce a value.

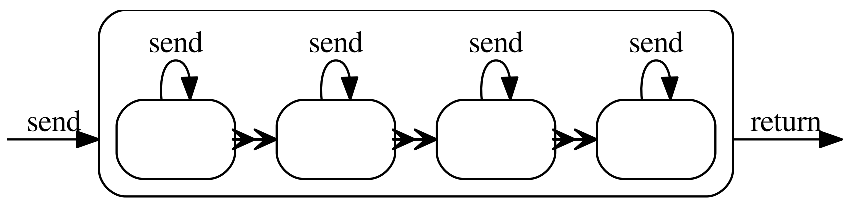
[1] See the Generator-Iterator Methods.

```
for value in chain:
    print(value)
```

Both `next(element)` and `element.send(None)` form the *public* interface of an element. They take care of unwinding chain complexities, such as multiple paths and skipping of values. Custom *chainlinks* should implement `chainlet_send()` to change how data is processed.

2.2 Linear Flow – Flat Chains

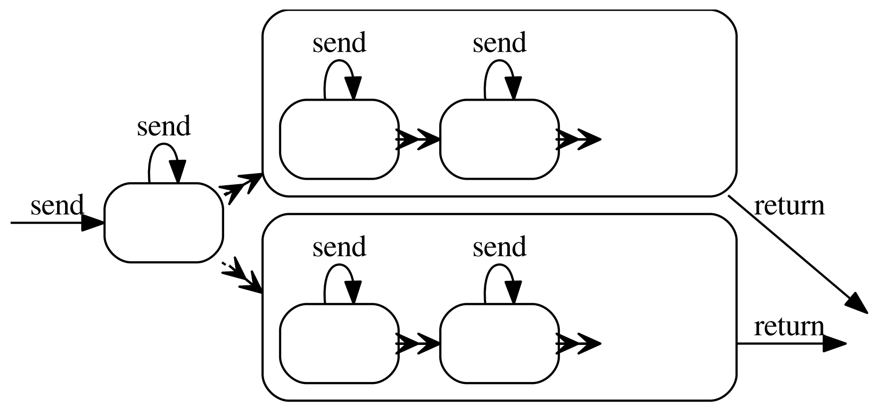
The simplest compound object is a *flat chain*, which is a sequence of *chainlinks*. Data sent to the chain is transformed incrementally: Input is passed to the first element, and its result to the second, and so on. Once all elements have been traversed, the result is returned.



Linear chains are special in that they always take a single input *chunk* and return a single output *chunk*. Even when *linking* flat chains, the result is flat linear chain with the same features. This makes them a suitable replacement for generators in any way.

2.3 Concurrent Flow – Chain Bundles

Processing of data can be split to multiple sub-chains in a *bundle*, a group of concurrent *chainlinks*. When a chain *branches* to multiple sub-chains, data flows along each sub-chain independently. In specific, the return value of the element *before* the *branch* is passed to *each* sub-chain individually.



In contrast to a *flat chain*, a *bundle* always returns multiple *chunks* at once: its return value is an iterable over *all chunks* returned by sub-chains. This holds true even if just one subchain returns anything.

Note: To avoid unnecessary overhead, parallel chains **never** copy data for each pipeline. If an element changes a mutable data structure, it should explicitly create a copy. Otherwise, peers may see the changes as well.

2.4 Compound Flow - Generic Chains

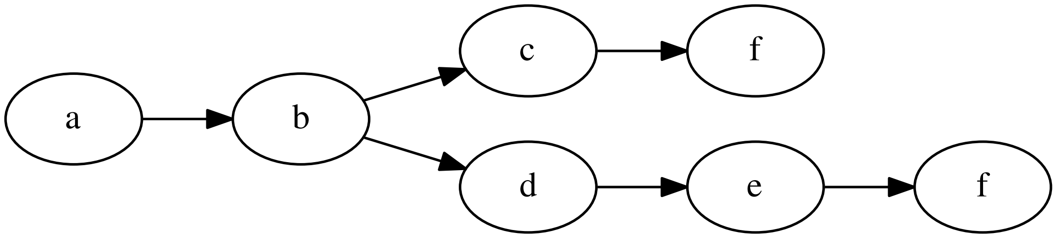
Combinations of *flat chains* and *bundles* automatically create a generic *chain*. This *compound link* is aware of *joining* and *forking* of the data flow for processing. *Flat chains* and *bundles* implement a specific combination of these feature; custom elements can freely provide other combinations.

Both *flat chains* and *bundles* do not *join* - they process each *data chunk* individually. A *flat chain* always produces one output *chunk* for every input *chunk*. In contrast, a *bundle* produces multiple output *chunks* for each input *chunk*.

A statement such as the following:

```
name('a') >> name('b') >> (name('c'), name('d') >> name('e')) >> name('f')
```

Creates a *chain* that *branches* from `f` to both `c` and `d >> e`. For the data flow, `f` is visited *separately* for the results from `c` and `e`.



Note: Stay aware of object identity when linking, especially if objects carry state. There is a difference in connecting nodes to the same objects, and connecting nodes to equivalent but separate objects.

2.4.1 Generic Join and Fork

The traversal through a *chain* is agnostic towards the type of elements: Each element explicitly specifies whether it joins the data flow or forks it. This is signaled via the attributes `element.chain_join` and `element.chain_fork`, respectively.

A *joining* element *receives* an iterable providing all data chunks produced by its preceding element. A *forking* element *produces* an iterable providing all valid data chunks. These features can be combined to have an element *join* the incoming data flow and *fork* it to another number of outgoing *chunks*.

Fork/Join	False	True
False	1->1	n->1
True	1->m	n->m

A *flat chain* is an example for a 1 -> 1 data flow, while a *bundle* implements a 1 -> m data flow. A generic *chain* is adjusted depending on its elements.

CHAPTER 3

Traversal Synchronicity

By default, *chainlet* operates in synchronous mode: there is a fixed ordering by which elements are traversed. Both *chains* and *bundles* are traversed one element at a time.

However, *chainlet* also allows for asynchronous mode: any elements which do not explicitly depend on each other can be traversed in parallel.

3.1 Synchronous Traversal

Synchronous mode follows the order of elements in *chains* and *bundles*[1]. Consider the following setup:

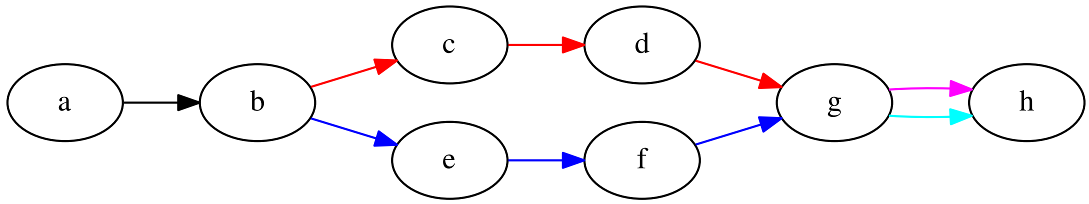
```
a >> b >> [c >> d, e >> f] >> g >> h
```

This is broken down into four *chains*, two of which are part of a *bundle*. Every *chain* is simply traversed according to its ordering - `a` before `b`, `c` before `d` and so on.

The *bundle* implicitly *forks* the data stream to *both* `c` and `e`. This *fork* is traversed in definition order, in this case `c >> d` before `e >> f`.

Synchronous traversal only guarantees consistency in each stream - but not about the ordering of *chainlinks* *across* the forked data stream. That is, the final sequence `g >> h` is always traversed after its respective source *chain* `c >> d` or `e >> f`. However, the *first* traversal of `g >> h` may or may not occur before `e >> f`, the *second* element of the *bundle*.

[1] In some cases, such as bundles from a `set`, traversal order may be arbitrary. However, it is still fixed and stable.



In other words, the traversal always picks black over red, red over blue, red over magenta and blue over cyan. This implies that magenta is traversed before cyan. However, it does *not* imply an ordering between blue and magenta.

Finally, synchronous traversal always respects the ordering of complete traversals. For every input, the *entire chain*

CHAPTER 4

Glossary

link

linking The combination of multiple *chainlinks* to form a *compound link*.

chunk

data chunk The smallest piece of data passed along individually. There is no restriction on the size or type of chunks: A *chunk* may be a primitive, such as an `int`, a container, such as a `dict`, or an arbitrary `object`.

stream

data stream An iterable of *data chunks*. It is implicitly passed along a *chain*, as *chainlinks* operate on its individual *chunks*.

The *stream* is an abstract object and never implicitly materialized by `chainlet`. For example, it can be an actual sequence, an (in)finite generator, or created piecewise via `send()`.

stream slice A portion of the *data stream*, containing multiple adjacent *data chunks*. Slices are the underlying unit of *chunks* passing through a *chainlink*: a slice may shrink or expand as elements remove or add items, retaining the order of chunks.

chainlet An atomic *chainlink*. The most primitive elements capable of forming chains and bundles.

chainlink Primitive and compound elements from which chains can be formed.

compound link A group of *chainlinks*, which can be used as a whole as elements in chains and bundles.

The *chain* and *bundle* are the most obvious forms, created implicitly by the >> operator.

chain A *chainlink* consisting of a sequence of elements to be processed one after another. The output of a *chain* is one *data chunk* for every successful traversal.

bundle A *chainlink* forming a group of elements which process each *data chunk* concurrently. The output of a *bundle* are zero or many *data chunks* for every successful traversal.

flat chain A *chain* consisting only of primitive elements.

fork

forking Splitting of the data flow by a *chainlink*. A *chainlink* which forks may *produce* multiple *data chunks*, each of which are passed on individually.

join

joining Merging of the data flow by a *chainlink*. A *chainlink* which joins may *receive* multiple *data chunks*, all of which are passed to it at once.

branch A processing sequence that is traversed concurrently with others.

branching Splitting of the processing sequence into multiple *branches*. Usually implies a *fork*.

merging Combining of multiple *branches* into one. Usually implies a *join*.

CHAPTER 5

chainlet package

class chainlet.**ChainLink**

Bases: `object`

BaseClass for elements in a chain

A chain is created by binding *ChainLink*s together. This is a directional process: a binding is always made between parent and child. Each child can be the parent to another child, and vice versa.

The direction dictates how data is passed along the chain:

- A parent may *send()* a data chunk to a child.
- A child may pull the *next()* data chunk from the parent.

Chaining is done with >> and << operators as `parent >> child` and `child << parent`. Forking and joining of chains requires a sequence of multiple elements as parent or child.

parent >> child
child << parent

Bind `child` and `parent`. Both directions of the statement are equivalent: if `a` is made a child of `b`, then *b‘* is made a parent of `a`, and vice versa.

parent >> (child_a, child_b, ...)
parent >> [child_a, child_b, ...]
parent >> {child_a, child_b, ...}

Bind `child_a`, `child_b`, etc. as children of `parent`.

(parent_a, parent_b, ...) >> child
[parent_a, parent_b, ...] >> child
{parent_a, parent_b, ...} >> child

Bind `parent_a`, `parent_b`, etc. as parents of `child`.

Aside from binding, every *ChainLink* implements the Generator-Iterator Methods interface:

iter(*link*)

Create an iterator over all data chunks that can be created. Empty results are ignored.

link.**__next__**()

`link.`**`send`**`(`*None*`)`
`next``(`*link*`)`
Create a new chunk of data. Raise `StopIteration` if there are no more chunks. Implicitly used by `next(link)`.

`link.`**`send`**`(`*chunk*`)`
Process a data `chunk`, and return the result.

Note: The `next` variants contrast with `iter` by also returning empty chunks. Use variations of `next(iter(link))` for an explicit iteration.

`link.`**`chainlet_send`**`(`*chunk*`)`
Process a data `chunk` locally, and return the result.

This method implements data processing in an element; subclasses must overwrite it to define how they handle data.

This method should only be called to explicitly traverse elements in a chain. Client code should use `next(link)` and `link.send(chunk)` instead.

`link.`**`throw`**`(`*type*[, *value*[, *traceback*]]`)`
Raises an exception of `type` inside the link. The link may either return a final result (including `None`), raise `StopIteration` if there are no more results, or propagate any other, unhandled exception.

`link.`**`close`**`()`
Close the link, cleaning up any resources.. A closed link may raise `RuntimeError` if data is requested via `next` or processed via `send`.

When used in a chain, each *`ChainLink`* is distinguished by its handling of input and output. There are two attributes to signal the behaviour when chained. These specify whether the element performs a *1 -> 1*, *n -> 1*, *1 -> m* or *n -> m* processing of data.

`chain_join`
A `bool` indicating that the element expects the values of all preceding elements at once. That is, the *chunk* passed in via *`send()`* is an *iterable* providing the return values of the previous elements.

`chain_fork`
A `bool` indicating that the element produces several values at once. That is, the return value is an *iterable* of data chunks, each of which should be passed on independently.

To prematurely stop the traversal of a chain, *1 -> n* and *n -> m* elements should return an empty container. Any *1 -> 1* and *n -> 1* element must raise *`StopTraversal`*.

`chain_fork = False`
whether this element produces several data chunks at once

`chain_join = False`
whether this element processes several data chunks at once

`chain_types = <chainlet.chainlink.LinkPrimitives object>`

`chainlet_send``(`*value=None*`)`
Send a value to this element for processing

`close``()`
Close this element, freeing resources and blocking further interactions

`dispatch``(`*values*`)`
Dispatch multiple values to this element for processing

`next``()`

send (*value=None*)
Send a single value to this element for processing

static throw (*type*, *value=None*, *traceback=None*)
Throw an exception in this element

exception chainlet.**StopTraversal**
Bases: `exceptions.Exception`

Stop the traversal of a chain

Any chain element raising *StopTraversal* signals that subsequent elements of the chain should not be visited with the current value.

Raising *StopTraversal* does *not* mean the element is exhausted. It may still produce values regularly on future traversal. If an element will *never* produce values again, it should raise `ChainExit`.

Note This signal explicitly affects the current chain only. It does not affect other, parallel chains of a graph.

Changed in version 1.3: The `return_value` parameter was removed.

chainlet.**funclet** (*function*)
Convert a function to a *ChainLink*

```
@funclet
def square(value):
    "Convert every data chunk to its numerical square"
    return value ** 2
```

The *data chunk* `value` is passed anonymously as the first positional parameter. In other words, the wrapped function should have the signature:

.slave (*value*, **args*, ***kwargs*)

chainlet.**genlet** (*generator_function=None*, *prime=True*)
Decorator to convert a generator function to a *ChainLink*

Parameters

- **generator_function** (*generator*) – the generator function to convert
- **prime** (*bool*) – advance the generator to the next/first yield

When used as a decorator, this function can also be called with and without keywords.

```
@genlet
def pingpong():
    "Chainlet that passes on its value"
    last = yield
    while True:
        last = yield last

@genlet(prime=True)
def produce():
    "Chainlet that produces a value"
    while True:
        yield time.time()

@genlet(True)
def read(iterable):
    "Chainlet that reads from an iterable"
```

(continues on next page)

(continued from previous page)

```
    for item in iterable:
        yield item
```

chainlet.**joinlet**(*chainlet*)

Decorator to mark a chainlet as joining

Parameters **chainlet** (chainlink.ChainLink) – a chainlet to mark as joining

Returns the chainlet modified inplace

Return type *chainlink.ChainLink*

Applying this decorator is equivalent to setting *chain_join* on chainlet: every *data chunk* is an iterable containing all data returned by the parents. It is primarily intended for use with decorators that implicitly create a new *ChainLink*.

```
@joinlet
@funclet
def average(value: Iterable[Union[int, float]]):
    "Reduce all data of the last step to its average"
    values = list(value)  # value is an iterable of values due to joining
    if not values:
        return 0
    return sum(values) / len(values)
```

chainlet.**forklet**(*chainlet*)

Decorator to mark a chainlet as forking

Parameters **chainlet** (chainlink.ChainLink) – a chainlet to mark as forking

Returns the chainlet modified inplace

Return type *chainlink.ChainLink*

See the note on *joinlet()* for general features. This decorator sets *chain_fork*, and implementations *must* provide an iterable.

```
@forklet
@funclet
def friends(value):
    "Split operations for every friend of a person"
    return (person for person in persons if person.is_friend(value))
```

5.1 Subpackages

5.1.1 chainlet.compat package

Compatibility layer for different python implementations

chainlet.compat.**COMPAT_VERSION = sys.version_info(major=2, minor=7, micro=12, releaselevel=**

Python version for which compatibility has been established

chainlet.compat.**throw_method**

staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C: def f(arg1, arg2, . . .): . . . f = staticmethod(f)

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

Submodules

chainlet.compat.python2 module

`chainlet.compat.python2.`**`throw_method`**

staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C: def f(arg1, arg2, . . .): . . . f = staticmethod(f)

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

chainlet.compat.python3 module

`chainlet.compat.python3.`**`throw_method`**

staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C: def f(arg1, arg2, . . .): . . . f = staticmethod(f)

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

5.1.2 chainlet.concurrency package

Primitives and tools to construct concurrent chains

`chainlet.concurrency.`**`threads`**`(`*element*`)`

Convert a regular *chainlink* to a thread based version

Parameters **`element`** – the chainlink to convert

Returns a threaded version of `element` if possible, or the element itself

Submodules

chainlet.concurrency.base module

class `chainlet.concurrency.base.`**`ConcurrentBundle`**(*elements*)

Bases: `chainlet.chainlink.Bundle`

A group of chainlets that concurrently process each *data chunk*

Processing of chainlets is performed using only the requesting threads. This allows thread-safe usage, but requires explicit concurrent usage for blocking actions, such as file I/O or `time.sleep()`, to be run in parallel.

Concurrent bundles implement element concurrency: the same data is processed concurrently by multiple elements.

`chainlet_send`(*value=None*)

Send a value to this element for processing

`executor = <chainlet.concurrency.base.LocalExecutor object>`

class `chainlet.concurrency.base.`**`ConcurrentChain`**(*elements*)

Bases: `chainlet.chainlink.Chain`

A group of chainlets that concurrently process each *data chunk*

Processing of chainlets is performed using only the requesting threads. This allows thread-safe usage, but requires explicit concurrent usage for blocking actions, such as file I/O or `time.sleep()`, to be run in parallel.

Concurrent chains implement data concurrency: multiple data is processed concurrently by the same elements.

Note A `ConcurrentChain` will *always join* and *fork* to handle all data.

`chainlet_send`(*value=None*)

Send a value to this element for processing

`executor = <chainlet.concurrency.base.LocalExecutor object>`

class `chainlet.concurrency.base.`**`FutureChainResults`**(*futures*)

Bases: `object`

Chain result computation stored for future and concurrent execution

Acts as an iterable for the actual results. Each future can be executed prematurely by a concurrent executor, with a synchronous fallback as required. Iteration can lazily advance through all available results before blocking.

If any future raises an exception, iteration re-raises the exception at the appropriate position.

Parameters **`futures`** (`list[StoredFuture]`) – the stored futures for each result chunk

class `chainlet.concurrency.base.`**`LocalExecutor`**(*max_workers, identifier=''*)

Bases: `object`

Executor for futures using local execution stacks without concurrency

Parameters

- **`max_workers`** (`int or float`) – maximum number of threads in pool
- **`identifier`** (`str`) – base identifier for all workers

static submit(*call, *args, **kwargs*)

Submit a call for future execution

Returns future for the call execution

Return type *StoredFuture*

class chainlet.concurrency.base.**SafeTee**(*iterable*, *n=2*)

Bases: object

Thread-safe version of itertools.tee()

Parameters

- **iterable** – source iterable to split
- **n** (*int*) – number of safe iterators to produce for *iterable*

class chainlet.concurrency.base.**StoredFuture**(*call*, **args*, ***kwargs*)

Bases: object

Call stored for future execution

Parameters

- **call** – callable to execute
- **args** – positional arguments to call
- **kwargs** – keyword arguments to call

await_result()

Wait for the future to be realised

realise()

Realise the future if possible

If the future has not been realised yet, do so in the current thread. This will block execution until the future is realised. Otherwise, do not block but return whether the result is already available.

This will not return the result nor propagate any exceptions of the future itself.

Returns whether the future has been realised

Return type bool

result

The result from realising the future

If the result is not available, block until done.

Returns result of the future

Raises any exception encountered during realising the future

chainlet.concurrency.base.**multi_iter**(*iterable*, *count=2*)

Return *count* independent, thread-safe iterators for *iterable*

chainlet.concurrency.thread module

Thread based concurrency domain

Primitives of this module implement concurrency based on threads. This allows blocking actions, such as I/O and certain extension modules, to be run in parallel. Note that regular Python code is not parallelised by threads due to the Global Interpreter Lock. See the threading module for details.

warning The primitives in this module should not be used manually, and may change without deprecation warning. Use *convert()* instead.

class chainlet.concurrency.thread.**ThreadBundle**(*elements*)

Bases: *chainlet.concurrency.base.ConcurrentBundle*

chain_types = <chainlet.concurrency.thread.ThreadLinkPrimitives object>

executor = <chainlet.concurrency.thread.ThreadPoolExecutor object>

class chainlet.concurrency.thread.**ThreadChain**(*elements*)
Bases: *chainlet.concurrency.base.ConcurrentChain*

chain_types = <chainlet.concurrency.thread.ThreadLinkPrimitives object>

executor = <chainlet.concurrency.thread.ThreadPoolExecutor object>

class chainlet.concurrency.thread.**ThreadLinkPrimitives**
Bases: chainlet.chainlink.LinkPrimitives

base_bundle_type
alias of *ThreadBundle*

base_chain_type
alias of *ThreadChain*

flat_chain_type
alias of *ThreadChain*

class chainlet.concurrency.thread.**ThreadPoolExecutor**(*max_workers*, *identifier=''*)
Bases: *chainlet.concurrency.base.LocalExecutor*

Executor for futures using a pool of threads

Parameters

- **max_workers** (*int or float*) – maximum number of threads in pool
- **identifier** (*str*) – base identifier for all workers

submit(*call*, **args*, ***kwargs*)
Submit a call for future execution

Returns future for the call execution

Return type *StoredFuture*

chainlet.concurrency.thread.**convert**(*element*)
Convert a regular *chainlink* to a thread based version

Parameters **element** – the chainlink to convert

Returns a threaded version of element if possible, or the element itself

5.2 Submodules

5.2.1 chainlet.chainlink module

class chainlet.chainlink.**ChainLink**
Bases: object

BaseClass for elements in a chain

A chain is created by binding *ChainLink*s together. This is a directional process: a binding is always made between parent and child. Each child can be the parent to another child, and vice versa.

The direction dictates how data is passed along the chain:

- A parent may *send()* a data chunk to a child.

- A child may pull the `next()` data chunk from the parent.

Chaining is done with >> and << operators as `parent >> child` and `child << parent`. Forking and joining of chains requires a sequence of multiple elements as parent or child.

parent >> child
child << parent
Bind `child` and `parent`. Both directions of the statement are equivalent: if `a` is made a child of `b`, then *b'* is made a parent of `a`, and vice versa.

parent >> (child_a, child_b, ...)
parent >> [child_a, child_b, ...]
parent >> {child_a, child_b, ...}
Bind `child_a`, `child_b`, etc. as children of `parent`.

(parent_a, parent_b, ...) >> child
[parent_a, parent_b, ...] >> child
{parent_a, parent_b, ...} >> child
Bind `parent_a`, `parent_b`, etc. as parents of `child`.

Aside from binding, every `ChainLink` implements the Generator-Iterator Methods interface:

iter(*link*)
Create an iterator over all data chunks that can be created. Empty results are ignored.

`link.`**__next__**()
`link.`**send**(*None*)
next(*link*)
Create a new chunk of data. Raise `StopIteration` if there are no more chunks. Implicitly used by `next(link)`.

`link.`**send**(*chunk*)
Process a data `chunk`, and return the result.

Note: The `next` variants contrast with `iter` by also returning empty chunks. Use variations of `next(iter(link))` for an explicit iteration.

`link.`**chainlet_send**(*chunk*)
Process a data `chunk` locally, and return the result.

This method implements data processing in an element; subclasses must overwrite it to define how they handle data.

This method should only be called to explicitly traverse elements in a chain. Client code should use `next(link)` and `link.send(chunk)` instead.

`link.`**throw**(*type*[, *value*[, *traceback*]])
Raises an exception of `type` inside the link. The link may either return a final result (including `None`), raise `StopIteration` if there are no more results, or propagate any other, unhandled exception.

`link.`**close**()
Close the link, cleaning up any resources.. A closed link may raise `RuntimeError` if data is requested via `next` or processed via `send`.

When used in a chain, each `ChainLink` is distinguished by its handling of input and output. There are two attributes to signal the behaviour when chained. These specify whether the element performs a *1 -> 1*, *n -> 1*, *1 -> m* or *n -> m* processing of data.

chain_join
A `bool` indicating that the element expects the values of all preceding elements at once. That is, the *chunk* passed in via `send()` is an *iterable* providing the return values of the previous elements.

chain_fork
A `bool` indicating that the element produces several values at once. That is, the return value is an *iterable* of data chunks, each of which should be passed on independently.

To prematurely stop the traversal of a chain, *1 -> n* and *n -> m* elements should return an empty container. Any *1 -> 1* and *n -> 1* element must raise `StopTraversal`.

chain_fork = False
whether this element produces several data chunks at once

chain_join = False
whether this element processes several data chunks at once

chain_types = <chainlet.chainlink.LinkPrimitives object>

chainlet_send(*value=None*)
Send a value to this element for processing

close()
Close this element, freeing resources and blocking further interactions

dispatch(*values*)
Dispatch multiple values to this element for processing

next()

send(*value=None*)
Send a single value to this element for processing

static throw(*type*, *value=None*, *traceback=None*)
Throw an exception in this element

5.2.2 chainlet.chainsend module

`chainlet.chainsend.`**lazy_send**(*chainlet*, *chunks*)
Canonical version of *chainlet_send* that always takes and returns an iterable

Parameters

- **chainlet** (`chainlink.ChainLink`) – the chainlet to receive and return data
- **chunks** (*iterable*) – the stream slice of data to pass to `chainlet`

Returns the resulting stream slice of data returned by `chainlet`

Return type iterable

`chainlet.chainsend.`**eager_send**(*chainlet*, *chunks*)
Eager version of *lazy_send* evaluating the return value immediately

Note The return value by an `n` to `m` link is considered fully evaluated.

Parameters

- **chainlet** (`chainlink.ChainLink`) – the chainlet to receive and return data
- **chunks** (*iterable*) – the stream slice of data to pass to `chainlet`

Returns the resulting stream slice of data returned by `chainlet`

Return type iterable

5.2.3 chainlet.dataflow module

Helpers to modify the flow of data through a *chain*

class chainlet.dataflow.**NoOp**

Bases: chainlet.chainlink.NeutralLink

A noop element that returns any input unchanged

This element is useful when an element is syntactically required, but no action is desired. For example, it can be used to split a pipeline into a modified and unmodifed version:

```
translate = parse_english >> (NoOp(), to_french, to_german)
```

Note Unlike the NeutralLink, this element is not optimized away by linking.

chainlet.dataflow.**joinlet**(*chainlet*)

Decorator to mark a chainlet as joining

Parameters **chainlet** (chainlink.ChainLink) – a chainlet to mark as joining

Returns the chainlet modified inplace

Return type *chainlink.ChainLink*

Applying this decorator is equivalent to setting *chain_join* on chainlet: every *data chunk* is an iterable containing all data returned by the parents. It is primarily intended for use with decorators that implicitly create a new *ChainLink*.

```
@joinlet
@funclet
def average(value: Iterable[Union[int, float]]):
    "Reduce all data of the last step to its average"
    values = list(value)  # value is an iterable of values due to joining
    if not values:
        return 0
    return sum(values) / len(values)
```

chainlet.dataflow.**forklet**(*chainlet*)

Decorator to mark a chainlet as forking

Parameters **chainlet** (chainlink.ChainLink) – a chainlet to mark as forking

Returns the chainlet modified inplace

Return type *chainlink.ChainLink*

See the note on *joinlet()* for general features. This decorator sets *chain_fork*, and implementations *must* provide an iterable.

```
@forklet
@funclet
def friends(value):
    "Split operations for every friend of a person"
    return (person for person in persons if person.is_friend(value))
```

class chainlet.dataflow.**MergeLink**(**mergers*)

Bases: *chainlet.chainlink.ChainLink*

Element that joins the data flow by merging individual data chunks

Parameters **mergers** (*tuple[type, callable]*) – pairs of `type, merger` to merge subclasses of `type` with `merger`

Merging works on the assumption that all *data chunks* from the previous step are of the same type. The type is deduced by peeking at the first *chunk*, based on which a `merger` is selected to perform the actual merging. The choice of a `merger` is re-evaluated at every step; a single *MergeLink* can handle a different type on each step.

Selection of a `merger` is based on testing `issubclass(type(first), merger_type)`. This check is evaluated in order, iterating through `mergers` before using *default_merger*. For example, `Counter` precedes `dict` to use a summation based merge strategy.

Each `merger` must implement the call signature

merger (*base_value: T, iter_values: Iterable[T]*) → T

where `base_value` is the value used for selecting the `merger`.

chain_fork = False

chain_join = True

chainlet_send (*value=None*)
Send a value to this element for processing

default_merger = [(<class 'numbers.Number'>, <function merge_numerical>), (<class 'col
type specific merge function mapping of the form `(type, merger)`

`chainlet.dataflow.`**either**
alias of `chainlet.dataflow.Either`

5.2.4 chainlet.driver module

class `chainlet.driver.`**ChainDriver**
Bases: `object`

Actively drives chains by pulling them

This driver pulls all mounted chains via a single thread. This drives chains synchronously, but blocks all chains if any individual chain blocks.

mount (**chains*)
Add chains to this driver

run ()
Start driving the chain, block until done

running
Whether the driver is running, either via *run()* or *start()*

start (*daemon=True*)
Start driving the chain asynchronously, return immediately

Parameters **daemon** (*bool*) – ungracefully kill the driver when the program terminates

class `chainlet.driver.`**ConcurrentChainDriver** (*daemon=True*)
Bases: *chainlet.driver.ChainDriver*

Actively drives chains by pulling them

This driver pulls all mounted chains via independent stacks. This drives chains concurrently, without blocking for any specific chain. Chains sharing elements may need to be synchronized explicitly.

Parameters **daemon** (*bool*) – run chains as `daemon`, i.e. do not wait for them to exit when terminating

create_runner(*mount*)

run()
Start driving the chain, block until done

class `chainlet.driver.`**MultiprocessChainDriver**(*daemon=True*)
Bases: *chainlet.driver.ConcurrentChainDriver*

Actively drives chains by pulling them

This driver pulls all mounted chains via independent processes. This drives chains concurrently, without blocking for any specific chain. Chains sharing elements cannot exchange state between them.

Parameters **daemon** (*bool*) – run processes as `daemon`, i.e. do not wait for them to finish

create_runner(*mount*)

class `chainlet.driver.`**ThreadedChainDriver**(*daemon=True*)
Bases: *chainlet.driver.ConcurrentChainDriver*

Actively drives chains by pulling them

This driver pulls all mounted chains via independent threads. This drives chains concurrently, without blocking for any specific chain. Chains sharing elements may need to be synchronized explicitly.

Parameters **daemon** (*bool*) – run threads as `daemon`, i.e. do not wait for them to finish

create_runner(*mount*)

5.2.5 chainlet.funclink module

Helpers for creating ChainLinks from functions

Tools of this module allow writing simpler code by expressing functionality via functions. The interface to other *chainlet* objects is automatically built around the functions. Using functions in chains allows for simple, stateless blocks.

A regular function can be directly used by wrapping *FunctionLink* around it:

```
from mylib import producer, consumer

def stepper(value, resolution=10):
    return (value // resolution) * resolution

producer >> FunctionLink(stepper, 20) >> consumer
```

If a function is used only as a chainlet, one may permanently convert it by applying a decorator:

```
from collections import deque
from mylib import producer, consumer

@GeneratorLink.linklet
def stepper(value, resolution=10):
    # ...

producer >> stepper(20) >> consumer
```

class chainlet.funclink.**FunctionLink**(*slave*, **args*, ***kwargs*)
Bases: *chainlet.wrapper.WrapperMixin*, *chainlet.chainlink.ChainLink*

Wrapper making a function act like a ChainLink

Parameters

- **slave** – the function to wrap
- **args** – positional arguments for the slave
- **kwargs** – keyword arguments for the slave

Note Use the *funclet()* function if you wish to decorate a function to produce FunctionLinks.

This class wraps a function (or other callable), calling it to perform work when receiving a value and passing on the result. The `slave` can be any object that is callable, and should take at least a named parameter `value`.

When receiving a **:tern:'data chunk'** `value` as part of a chain, `send()` acts like `slave(value, *args, **kwargs)`. Any calls to `throw()` and `close()` are ignored.

chainlet_send(*value=None*)
Send a value to this element

class chainlet.funclink.**PartialSlave**
Bases: `object`

args

func

keywords

chainlet.funclink.**funclet**(*function*)
Convert a function to a `ChainLink`

```
@funclet
def square(value):
    "Convert every data chunk to its numerical square"
    return value ** 2
```

The *data chunk* `value` is passed anonymously as the first positional parameter. In other words, the wrapped function should have the signature:

.**slave**(*value*, **args*, ***kwargs*)

5.2.6 chainlet.genlink module

Helpers for creating ChainLinks from generators

Tools of this module allow writing simpler code by expressing functionality via generators. The interface to other *chainlet* objects is automatically built around the generator. Using generators in chains allows to carry state between steps.

A regular generator can be directly used by wrapping *GeneratorLink* around it:

```
from collections import deque
from mylib import producer, consumer

def windowed_average(size=8):
    buffer = collections.deque([(yield)], maxlen=size)
    while True:
```

(continues on next page)

(continued from previous page)

```
        new_value = yield(sum(buffer)/len(buffer))
        buffer.append(new_value)

producer >> GeneratorLink(windowed_average(16)) >> consumer
```

If a generator is used only as a chainlet, one may permanently convert it by applying a decorator:

```
from collections import deque
from mylib import producer, consumer

@genlet
def windowed_average(size=8):
    # ...

producer >> windowed_average(16) >> consumer
```

class chainlet.genlink.**GeneratorLink**(*slave*, *prime=True*)

Bases: *chainlet.wrapper.WrapperMixin*, *chainlet.chainlink.ChainLink*

Wrapper making a generator act like a ChainLink

Parameters

- **slave** – the generator instance to wrap
- **prime** (*bool*) – advance the generator to the next/first yield

Note Use the *genlet()* function if you wish to decorate a generator *function* to produce GeneratorLinks.

This class wraps a generator, using it to perform work when receiving a value and passing on the result. The `slave` can be any object that implements the generator protocol - the methods `send`, `throw` and `close` are directly called on the `slave`.

chainlet_send(*value=None*)

Send a value to this element for processing

close()

Close this element, freeing resources and blocking further interactions

throw(*type*, *value=None*, *traceback=None*)

Raise an exception in this element

class chainlet.genlink.**StashedGenerator**(*generator_function*, **args*, ***kwargs*)

Bases: `object`

A generator iterator which can be copied/pickled before any other operations

Parameters

- **generator_function** (*function*) – the source generator function
- **args** – positional arguments to pass to `generator_function`
- **kwargs** – keyword arguments to pass to `generator_function`

This class can be used instead of instantiating a generator function. The following two calls will behave the same for all generator operations:

```
my_generator(1, 2, 3, foo='bar')
StashedGenerator(my_generator, 1, 2, 3, foo='bar')
```

However, a `StashedGenerator` can be pickled and unpickled before any generator operations are used on it. It explicitly disallows pickling after `next()`, `send()`, `throw()` or `close()`.

```
def parrot(what='Polly says %s'):
    value = yield
    while True:
        value = yield (what % value)

simon = StashedGenerator(parrot, 'Simon says %s')
simon2 = pickle.loads(pickle.dumps(simon))
next(simon2)
print(simon2.send('Hello'))   # Simon says Hello
simon3 = pickle.loads(pickle.dumps(simon2))   # raise TypeError
```

close() → raise GeneratorExit inside generator.

next() → the next value, or raise StopIteration

send(*arg*) → send 'arg' into generator,
return next yielded value or raise StopIteration.

throw(*typ*[, *val*[, *tb*]]) → raise exception in generator,
return next yielded value or raise StopIteration.

chainlet.genlink.**genlet**(*generator_function=None*, *prime=True*)
Decorator to convert a generator function to a `ChainLink`

Parameters

- **generator_function** (*generator*) – the generator function to convert
- **prime** (*bool*) – advance the generator to the next/first yield

When used as a decorator, this function can also be called with and without keywords.

```
@genlet
def pingpong():
    "Chainlet that passes on its value"
    last = yield
    while True:
        last = yield last

@genlet(prime=True)
def produce():
    "Chainlet that produces a value"
    while True:
        yield time.time()

@genlet(True)
def read(iterable):
    "Chainlet that reads from an iterable"
    for item in iterable:
        yield item
```

5.2.7 chainlet.protolink module

Helpers for creating ChainLinks from standard protocols of objects

Tools of this module allow writing simpler code by reusing functionality of existing protocol interfaces and builtins. The interface to other *chainlet* objects is automatically built around the objects. Using protocol interfaces in chains allows to easily create chainlets from existing code.

Every protolink represents a specific Python protocol or builtin. For example, the `iterlet()` protolink maps to `iter(iterable)`. This allows pulling chunks from iterables to a chain:

```
from examples import windowed_average

fixed_iterable = [1, 2, 4, 3]
chain = iterlet(fixed_iterable) >> windowed_average(size=2)
for value in chain:
    print(value)  # prints 1.0, 1.5, 3.0, 3.5
```

The protolinks exist mostly for convenience - they are thin wrappers using `chainlet` primitives. As such, they are most useful to adjust existing code and objects for pipelines.

Any protolink that works on iterables supports two modes of operation:

pull: iterable provided at instantiation Pull data chunks directly from an iterable, work on them, and send them along a chain. These are usually equivalent to a corresponding builtin, but support chaining.

push: no iterable provided at instantiation Wait for data chunks to be pushed in, work on them, and send them along a chain. These are usually equivalent to wrapping a chain in the corresponding builtin, but preserve chain features.

class chainlet.protolink.**callet**(**slave_args*, ***slave_kwargs*)

Bases: `chainlet.genlink.GeneratorLink`

Pull chunks from an object using individual calls

Parameters **callee** (*callable*) – object supporting `callee()`

```
import random

chain = callet(random.random) >> windowed_average(size=200)
for _ in range(50):
    print(next(chain))  # prints series converging to 0.5
```

callet.**_slave_factory**(*callee*)

Pull chunks from an object using individual calls

Parameters **callee** (*callable*) – object supporting `callee()`

```
import random

chain = callet(random.random) >> windowed_average(size=200)
for _ in range(50):
    print(next(chain))  # prints series converging to 0.5
```

chainlet.protolink.**enumeratelet**(*iterable=None*, *start=0*)

Enumerate chunks of data from an iterable or a chain

Parameters

- **iterable** (*iterable, None or int*) – object supporting iteration, or an index
- **start** (*int*) – an index to start counting from

Raises **TypeError** – if both parameters are set and `iterable` does not support iteration

In pull mode, `enumeratelet()` works similar to the builtin `enumerate()` but is chainable:

```
chain = enumeratelet(['Paul', 'Thomas', 'Brian']) >> printlet(sep=':\t')
for value in chain:
    pass  # prints `0:  Paul`, `1:      Thomas`, `2:    Brian`
```

By default, `enumeratelet()` enumerates chunks passed in from a pipeline. To use a different starting index, *either* set the `start` keyword parameter *or* set the first positional parameter.

```
chain = iteratelet(['Paul', 'Thomas', 'Brian']) >> enumeratelet() >> printlet(sep=
↪':\t')
for value in chain:
    pass  # prints `0:  Paul`, `1:      Thomas`, `2:    Brian`
```

`chainlet.protolink.`**`filterlet`**(*function=<type 'bool'>, iterable=None*)
Filter chunks of data from an iterable or a chain

Parameters

- **`function`** (*`callable`*) – callable selecting valid elements
- **`iterable`** (*`iterable or None`*) – object providing chunks via iteration

For any chunk in `iterable` or the chain, it is passed on only if `function(chunk)` returns true.

```
chain = iterlet(range(10)) >> filterlet(lambda chunk: chunk % 2 == 0)
for value in chain:
    print(value)  # prints 0, 2, 4, 6, 8
```

class `chainlet.protolink.`**`iterlet`**(**slave_args, **slave_kwargs*)
Bases: `chainlet.genlink.GeneratorLink`

Pull chunks from an object using iteration

Parameters **`iterable`** (*`iterable`*) – object supporting iteration

```
chain = iterlet([1, 2, 3, 4, 5, 5, 6, 6]) >> filterlet(lambda chunk: chunk % 2 ==␣
↪0)
for element in chain:
    print(element)  # prints 2, 4, 6, 6
```

`iterlet.`**`_slave_factory`**(*iterable*)
Pull chunks from an object using iteration

Parameters **`iterable`** (*`iterable`*) – object supporting iteration

```
chain = iterlet([1, 2, 3, 4, 5, 5, 6, 6]) >> filterlet(lambda chunk: chunk % 2 ==␣
↪0)
for element in chain:
    print(element)  # prints 2, 4, 6, 6
```

class `chainlet.protolink.`**`printlet`**(**slave_args, **slave_kwargs*)
Bases: `chainlet.genlink.GeneratorLink`

Print chunks of data from a chain

Parameters

- **`flatten`** – whether to flatten data chunks
- **`kwargs`** – keyword arguments as for `print()`

If `flatten` is `True`, every chunk received is unpacked. This is useful when passing around connected data, e.g. from *`enumeratelet()`*.

Keyword arguments via `kwargs` are equivalent to those of `print()`. For example, passing `file=sys.stderr` is a simple way of creating a debugging element in a chain:

```
debug_chain = chain[:i] >> printlet(file=sys.stderr) >> chain[i:]
```

`printlet.`**`_slave_factory`**(*flatten=False*, ***kwargs*)

Print chunks of data from a chain

Parameters

- **`flatten`** – whether to flatten data chunks
- **`kwargs`** – keyword arguments as for `print()`

If `flatten` is `True`, every chunk received is unpacked. This is useful when passing around connected data, e.g. from *`enumeratelet()`*.

Keyword arguments via `kwargs` are equivalent to those of `print()`. For example, passing `file=sys.stderr` is a simple way of creating a debugging element in a chain:

```
debug_chain = chain[:i] >> printlet(file=sys.stderr) >> chain[i:]
```

`chainlet.protolink.`**`reverselet`**(*iterable*)

Pull chunks from an object using reverse iteration

Parameters **`iterable`** (*`iterable`*) – object supporting reverse iteration

See `iterlet()` for an example.

5.2.8 chainlet.signals module

exception `chainlet.signals.`**`ChainExit`**

Bases: `exceptions.Exception`

Terminate the traversal of a chain

exception `chainlet.signals.`**`StopTraversal`**

Bases: `exceptions.Exception`

Stop the traversal of a chain

Any chain element raising *`StopTraversal`* signals that subsequent elements of the chain should not be visited with the current value.

Raising *`StopTraversal`* does *not* mean the element is exhausted. It may still produce values regularly on future traversal. If an element will *never* produce values again, it should raise *`ChainExit`*.

Note This signal explicitly affects the current chain only. It does not affect other, parallel chains of a graph.

Changed in version 1.3: The `return_value` parameter was removed.

5.2.9 chainlet.utility module

class `chainlet.utility.`**`Sentinel`**(*name=None*)

Bases: `object`

Unique placeholders for signals

5.2.10 chainlet.wrapper module

class chainlet.wrapper.**WrapperMixin**(*slave*)

Bases: object

Mixin for ChainLinks that wrap other objects

Apply as a mixin via multiple inheritance:

```
class SimpleWrapper(WrapperMixin, ChainLink):
    /"/"/"Chainlink that calls ``slave`` for each chunk/"/"/"
    def __init__(self, slave):
        super().__init__(slave=slave)

    def chainlet_send(self, value):
        value = self.__wrapped__.send(value)
```

Wrappers bind their slave to `__wrapped__`, as is the Python standard, and also expose them via the `slave` property for convenience.

Additionally, subclasses provide the *wraplet()* to create factories of wrappers. This requires `__init_slave__()` to be defined.

slave

classmethod wraplet(**cls_args*, ***cls_kwargs*)

Create a factory to produce a Wrapper from a slave factory

Parameters

- **cls_args** – positional arguments to provide to the Wrapper class
- **cls_kwargs** – keyword arguments to provide to the Wrapper class

Returns

```
cls_wrapper_factory = cls.wraplet(*cls_args, **cls_kwargs)
link_factory = cls_wrapper_factory(slave_factory)
slave_link = link_factory(*slave_args, **slave_kwargs)
```

chainlet.wrapper.**getname**(*obj*)

Return the most qualified name of an object

Parameters **obj** – object to fetch name

Returns name of `obj`

CHAPTER 6

chainlet Changelog

6.1 v1.3.0

New Features

- The >> and << operators use experimental reflection precedence based on domains.
- Added a future based `concurrency` module.
- Added a `threading` based chain domain offering concurrent bundles.
- Added a `multiprocessing` based `Driver`.

Major Changes

- Due to inconsistent semantics, stopping a chain with `StopTraversal` no longer allows for a return value. Aligned `chainlet.send` to `generator.send`, returning `None` or an empty iterable instead of blocking indefinitely. See issue #8 for details.
- Added `chainlet.dispatch(iterable)` to `send` an entire stream slice at once. This allows for internal lazy and concurrent evaluation.
- Deprecated the use of external linkers in favour of operator+constructor.
- Linking to chains ignores elements which are `False` in a boolean sense, e.g. an empty `CompoundLink`.

Minor Changes

- `CompoundLink` objects are now considered boolean `False` based on elements.
- Added a neutral element for internal use.

Bug Fixes

- A `Bundle` will now properly `join` the stream if any of its elements does so.
- Correctly unwrapping return value for any `Chain` which does not `fork`.
- `FunctionLink` and `funclet` support positional arguments

6.2 v1.2.0

New Features

- Decorator/Wrapper versions of `FunctionLink` and `GeneratorLink` are proper subclasses of their class. This allows setting attributes and inspection. Previously, they were factory functions.
- Instances of `FunctionLink` can be copied and pickled.
- Instances of `GeneratorLink` can be copied and pickled.
- Subchains can be extracted from a `Chain` via slicing.

Major Changes

- Renamed compound chains and simplified inheritance to better reflect their structure:
  - `Chain` has been renamed to `CompoundLink`
  - `ConcurrentChain` has been removed
  - `MetaChain` has been renamed to `Chain`
  - `LinearChain` has been renamed to `FlatChain`
  - `ParallelChain` has been renamed to `Bundle`
- A `Chain` that never forks or definitely joins yields raw data chunks, instead of nesting each in a `list`
- A `Chain` whose first element does a `fork` inherits this.

Minor Changes

- The top-level namespace `chainlet` has been cleared from some specialised aliases.

Fixes

- Chains containing any `chainlet_fork` elements but no `Bundle` are properly built

6.3 v1.1.0 2017-06-08

New Features

- Protolinks: chainlet versions of builtins and protocols

Minor Changes

- Removed outdated sections from documentation

6.4 v1.0.0 2017-06-03

Notes

- Initial release

New Features

- Finalized definition of chainlet element interface on `chainlet.ChainLink`
- Wrappers for generators, coroutines and functions as `chainlet.genlet` and `chainlet.funclet`

- Finalized dataflow definition for chains, fork and join
- Drivers for sequential and threaded driving of chains

CHAPTER 7

chainlet

The *chainlet* library lets you quickly build iterative processing sequences. At its heart, it is built for chaining generators/coroutines, but supports arbitrary objects. It offers an easy, readable way to link elements using a concise mini language:

```
data_chain = read('data.txt') >> filterlet(preserve=bool) >> convert(apply=ast.
↪literal_eval)
for element in chain:
    print(element)
```

The same interface can be used to create chains that push data from the start downwards, or to pull from the end upwards.

```
push_chain = uppercase >> encode_r13 >> mark_of_insanity >> printer
push_chain.send('uryyb jbeyq')  # outputs 'Hello World!!!'

pull_chain = word_maker >> cleanup >> encode_r13 >> lowercase
print(next(pull_chain))  # outputs 'uryyb jbeyq'
```

Creating new elements is intuitive and simple, as *chainlet* handles all the gluing and binding for you. Most functionality can be created from regular functions, generators and coroutines:

```
@chainlet.genlet
def moving_average(window_size=8):
    buffer = collections.deque([(yield)], maxlen=window_size)
    while True:
        new_value = yield(sum(buffer)/len(buffer))
        buffer.append(new_value)
```

CHAPTER 8

Quick Overview

To just plug together existing chainlets, have a look at the *Chainlet Mini Language*. To port existing imperative code, the *chainlet.protolink module* provides simple helpers and equivalents of builtins.

Writing new chainlets is easily done writing generators, coroutines and functions, decorated with `chainlet.genlet()` or `chainlet.funclet()`. A `chainlet.genlet()` is best when state must be preserved between calls. A `chainlet.funclet()` allows resuming even after exceptions.

Advanced chainlets are best implemented as a subclass of `chainlet.ChainLink`. Overwrite instantiation and `chainlet_send()` to change their behaviour[1]. In order to change binding semantics, overwrite the `__rshift__` and `__lshift__` operators.

[1] Both `chainlet.genlet()` and `chainlet.funclet()` implement instantiation and `chainlet_send()` for the most common use case. They simply bind their callables on instantitation, then call them on `chainlet_send()`.

CHAPTER 9

Contributing and Feedback

The project is hosted on github. If you have issues or suggestion, check the issue tracker: For direct contributions, feel free to fork the development branch and open a pull request.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Documentation built from chainlet 1.3.1 at Jun 12, 2018.

Python Module Index

C

Index

Symbols

A

B

C

R

S

T

W