
ptera

Release 1.0

Olivier Breuleux

Jan 07, 2023

CONTENTS:

1	What is Ptera?	1
2	Getting started	3
2.1	Install	3
2.2	Usage	3
3	Guide	5
3.1	Probing	6
3.2	Operations	11
3.3	Selected operators	15
3.4	Miscellaneous	17
4	Use cases	21
4.1	Instrumenting external code	21
4.2	Advanced logging	22
4.3	Advanced debugging	22
4.4	Testing	22
5	Testing with Ptera	23
5.1	Test properties	23
5.2	Test information flow	24
5.3	Test for infinite loops	25
5.4	Test trends	26
5.5	Test caching	26
6	Reference	29
6.1	Main API	29
6.2	List of operators	30
6.3	ptera.interpret	92
6.4	ptera.opparse	95
6.5	ptera.overlay	96
6.6	ptera.probe	98
6.7	ptera.selector	109
6.8	ptera.tags	110
6.9	ptera.transform	110
6.10	ptera.utils	113
7	Indices and tables	115
	Python Module Index	117

WHAT IS PTERA?

Ptera is a way to instrument code from the outside. More precisely, it allows you to specify a set of variables to watch in an arbitrary Python call graph and manipulate a stream of their values.

For example, the following code will print `a = 12` and `b = 34`, at the moment that the variable `a` is set.

```
from ptera import probing

def f():
    a = 12

def g():
    b = 34
    f()

# "g(b) > f > a" is a *selector* that selects the variable b in the function g
# and the variable a in the function f, with a focus on variable a
with probing("g(b) > f > a") as prb:
    # The following line declares a processing pipeline. It must be declared
    # before the main functionality is called.
    prb.print("a = {a} and b = {b}")

    # When the watched variables are set, the values will go through the pipeline
    # declared previously
    g()
```

See *Probing* for more information on the probing syntax.

You can use Ptera to:

- *Instrument code that you do not control.*
- *Collect data across function scopes.*
- Perform *complex filters* and *reductions* on the stream of values.
- *Test* complex conditions on a program's internal state.

GETTING STARTED

2.1 Install

```
pip install ptera
```

2.2 Usage

The main API for Ptera is `probing()`, which is used as a context manager.

Here's an example involving a `fun` function. Even though the function returns nothing, we can use Ptera to extract all sorts of interesting things:

```
from ptera import probing

def collatz(n):
    while n != 1:
        n = (3 * n + 1) if n % 2 else (n // 2)

# `collatz > n` means: probe variable `n` in function `collatz`
# Every time `n` is set (including when it is given as a parameter)
# an event is sent through `prb`
with probing("collatz > n") as prb:
    # Declare one or more pipelines on the data.
    prb["n"].print("n = {}".format(n))
    prb["n"].max().print("max(n) = {}".format(n))
    prb["n"].count().print("number of steps: {}".format(n))

    # We can also ask for all values to be accumulated into a list
    values = prb["n"].accum()

    # Call the function once all the pipelines are set up.
    collatz(2021)

    # Print the values
    print("values =", values)

# Output:
# n = 2021
# ...
```

(continues on next page)

(continued from previous page)

```
# n = 1
# values = [2021, ..., 1]
# max(n) = 6064
# number of steps: 63
```

Note that in the example above the max/count are printed after the with block ends (they are triggered when there is no more data, and the stream is ended when the with block ends), which is why `print(values)` is not the last thing that's printed.

Contents

- *Probing*
 - *Probe a variable*
 - *Probe the return value*
 - *Probe multiple variables*
 - *Probe across scopes*
 - *Probe sibling calls*
 - *Total probes*
 - *Global probes*
 - *Wrapper probe*
- *Operations*
 - *Printing*
 - *Subscribe*
 - *Map, filter, reduce*
 - *Overriding values*
 - *Asserts*
 - *Conditional breakpoints*
- *Selected operators*
 - *Filtering*
 - *Mapping*
 - *Reduction*
 - *Arithmetic reductions*
 - *Wrapping*
 - *Timing*
 - *Debugging*
- *Miscellaneous*

- *Meta-variables*
- *Generic variables*
- *Selecting based on tags*
- *Probe methods*
- *Absolute references*

3.1 Probing

3.1.1 Probe a variable

To get a stream of the value of the variable named `a` in the function `f`, pass the selector `"f > a"` to `probing()`:

```
def f(x, y):
    a = x * x
    b = y * y
    return a + b

with probing("f > a").values() as values:
    f(12, 5)

assert values == [{"a": 144}]
```

The function `f` should be visible in the scope of the call to `probing` (alternatively, you can provide an explicit environment as the `env` argument).

3.1.2 Probe the return value

To probe the return value of `f`, use the selector `f()` as `result` (you can name the result however you like):

```
def f(x, y):
    return x + y

with probing("f() as result").values() as values:
    f(2, 5)

assert values == [{"result": 7}]
```

3.1.3 Probe multiple variables

Ptera is not limited to probing a single variable in a function: it can probe several at the same time (this is different from passing more than one selector to `probing`).

When probing multiple variables at the same time, it is important to understand the concept of **focus variable**. The **focus variable**, if present, is the variable that triggers the events in the pipeline when it is assigned to (note that parameters are considered to be “assigned to” at the beginning of the function):

1. `probing("f(x) > y")`: The focus is `y`, this triggers when `y` is set. (Probe type: *Immediate*)
2. `probing("f(y) > x")`: The focus is `x`, this triggers when `x` is set. (Probe type: *Immediate*)

3. `probing("f(x, y)")`: There is no focus, this triggers when `f` returns. (Probe type: *Total* – these may be a bit less intuitive, see the section on *Total probes* but don't feel like you have to use them)

To wit:

```
def f():
    x = 1
    y = 2
    x = 3
    y = 4
    x = 5
    return x

# Case 1: focus on y
with probing("f(x) > y").values() as values:
    f()

assert values == [
    {"x": 1, "y": 2},
    {"x": 3, "y": 4},
]

# Case 2: focus on x
with probing("f(y) > x").values() as values:
    f()

assert values == [
    {"x": 1}, # y is not set yet, so it is not in this entry
    {"x": 3, "y": 2},
    {"x": 5, "y": 4},
]

# Case 3: no focus
# See the section on total probes
with probing("f(x, y)", raw=True).values() as values:
    f()

assert values[0]["x"].values == [1, 3, 5]
assert values[0]["y"].values == [2, 4]
```

Note: The selector syntax does not necessarily mirror the syntax of actual function calls. For example, `f(x)` does not necessarily refer to a *parameter* of `f` called `x`. As shown above, you can put any local variable between the parentheses. You can also probe global/closure variables that are used in the body of `f`.

Note: The selector `f(x, !y)` is an alternative syntax for `f(x) > y`. The exclamation mark denotes the focus variable. There can only be one in a selector.

3.1.4 Probe across scopes

Sometimes you would like to get some context about whatever you are probing, and the context might not be in the same scope: it might be, for example, in the caller. Thankfully, Ptera has you covered.

```
def outer(n):
    x = 0
    for i in range(n):
        x += inner(i)
    return x

def inner(x):
    a = x * x
    return a + 1

with probing("outer(n) > inner > a").values() as values:
    outer(3)

assert values == [
    {"n": 3, "a": 0},
    {"n": 3, "a": 1},
    {"n": 3, "a": 4},
]
```

As you can see, this probe gives us the context of what the value of `n` is in the outer scope, and that context is attached to every entry.

Note: The selector `outer > inner > a` does not require `inner` to be called *directly* within `outer`. The call can be indirect, for example if `outer` calls `middle`, and `middle` calls `inner`, the selector will still match. This makes it even more practical, since you can easily capture context quite removed from the focus variable.

3.1.5 Probe sibling calls

Now we're getting into power features that are a bit more niche, but Ptera goes even beyond probing across caller/callee scopes: it can also attach results from sibling calls!

```
def main(x):
    return negmul(side(3), side(6))

def side(x):
    return x + 1

def negmul(x, y):
    a = x * y
    return -a

with probing("main(x, side(x as x2), negmul(!a))", raw=True).values() as values:
    main(12)

assert values == [
    {"x": 12, "x2": 6, "a": 28}
]
```

Here we use the `!` notation to indicate the focus variable, but it is not fundamentally different from doing `... > negmul`
`> a`. The probe above gives us, all at once:

- The value of `x` in the main function.
- The latest value of `x` in `side` (under a different name, to avoid clashing)
- The value of the local variable `a` in `negmul`

3.1.6 Total probes

A probe that does not have a focus variable is a “total” probe. Total probes function differently:

- Instead of triggering when a specific focus variable is set, they trigger when the outermost function in the selector ends.
- Instead of providing the latest values of all the variables, they collect *all* the values the variables have taken (hence the name “total”).
- Since the default interface of `probing` assumes there is only one value for each variable in each entry, total probes will fail if multiple values are captured for the same variable in the same entry, unless you pass `raw=True` to `probing`. This will cause *Capture* instances to be provided instead.

For example, if we remove the focus from the previous example (and add `raw=True`):

```
def main(x):
    return negmul(side(3), side(6))

def side(x):
    return x + 1

def negmul(x, y):
    a = x * y
    return -a

with probing("main(x, side(x as x2), negmul(a))", raw=True).values() as values:
    main(12)

assert values[0]["x"].values == [12]
assert values[0]["x2"].values == [3, 6]
assert values[0]["a"].values == [28]
```

In this example, each call to `main` will produce exactly one event, because `main` is the outermost call in the selector. You can observe that `x2` is associated to two values, because `side` was called twice.

Note: You can in fact create a total probe that has a focus with `probing(selector, probe_type="total")`. In this case, it will essentially duplicate the data for the outer scopes for each value of the focus variable.

3.1.7 Global probes

The `global_probe()` function can be used to set up a probe that remains active for the rest of the program. Unlike probing it is not a context manager.

```
def f(x):
    a = x * x
    return a

gprb = global_probe("f > a")
gprb.print()

f(4)  # prints 16
f(5)  # prints 25

gprb.deactivate()

f(6)  # prints nothing
```

Note: Probes can only be activated once, so after calling `deactivate` you will need to make a new probe if you want to reactivate it.

Note: Reduction operators such as `min()` or `sum()` are finalized when the probe exits. With probing, that happens at the end of the `with` block. With `global_probe`, that happens either when `deactivate` is called or when the program exits.

3.1.8 Wrapper probe

Warning: This is a less mature feature, use at your own risk.

A wrapper probe is a probe that has *two* focuses. On the first focus, it generates an opening event, and on the second focus, it generates a closing event. These events can be fed into a context manager or generator using `wrap()`, `kwrap()` (subscribers), or `wmap()` (operator).

The first focus works as normal and can be specified with `!`. The second focus is specified with `!!`. In the example below we compute the elapsed time between `a = 1` and `b = 2`:

```
def main(x):
    for i in range(1, x + 1):
        a = 1
        time.sleep(i)
        b = 2

def _timeit():
    t0 = time.time()
    yield
    t1 = time.time()
    return t1 - t0
```

(continues on next page)

(continued from previous page)

```

with probing("main(!a, !!b)") as prb:
    times = prb.wmap(_timeit).accum()
    main(3)

print(times)  # Approximately [0.1, 0.2, 0.3]

```

The `wmap` method takes a generator that yields exactly once. It is called when the first focus is triggered (captured values may be passed as keyword arguments). Then it must yield and will be resumed when the second focus is triggered (yield returns the captured data). The return value becomes the next value of the resulting stream.

The `wrap` and `kwrap` functions are similar, but they do not return streams. They work like `subscribe` and `ksubscribe`, but you can pass either a generator that yields once or an arbitrary context manager.

You can use meta-variables if needed:

- `main(!#enter, !!#exit)` can be used to wrap the entire function.
- `main(!#loop_i, !!#endloop_i)` can be used to wrap each iteration of the for loop that uses an iteration variable named `i`.

Note: If `prb` is a stream that contains multiple wrapper probes and you only want to wrap one of them, you can pass the name of the focus variable of its selector as the first argument to `wmap`.

Important: Wrapper probes work a little like `with` statements, but not really: if an error occurs between the two focuses, the wrapper probe will not be informed. The second focus will simply not happen and the generator will not be called back (it will just hang somewhere forever, wasting memory).

There is one safe special case: if you use a selector like `f(!#enter, #error, !!#exit)`, it should always complete because the special meta-variable `#exit` is always emitted when a function ends, even if there is an error. The error, if there is one, will be offered as `#error`. You can get that from the dictionary returned by `yield` in the handler you pass to `wmap`.

3.2 Operations

In all of the previous examples, I have used the `.values()` method to gather all the results into a list. This is a perfectly fine way to use Ptera and it has the upside of being simple and easy to understand. There are however many other ways to interact with the streams produced by probing.

3.2.1 Printing

Use `.print(<format>)` or `.display()` to print each element of the stream on its own line.

```

def f(x):
    y = 0
    for i in range(1, x + 1):
        y = y + x
    return y

```

(continues on next page)

(continued from previous page)

```
with probing("f > y").print("y = {y}"):
    f(3)

# Prints:
# y = 0
# y = 1
# y = 3
# y = 6
```

If `print` is given no arguments it will use plain `str()` to convert the elements to strings. `display()` displays dictionaries a bit more nicely.

3.2.2 Subscribe

You can, of course, subscribe arbitrary functions to a probe's stream. You can do so with:

1. The `>>` operator
2. The `subscribe` method (passes the dictionary as a positional argument)
3. The `ksubscribe` method (passes the dictionary as keyword arguments)

For example:

```
def f(x):
    y = 0
    for i in range(1, x + 1):
        y = y + x
    return y

with probing("f > y") as prb:
    # 1. The >> operator
    prb >> print

    # 2. The subscribe method
    @prb.subscribe
    def _(data):
        print("subscribe", data)

    # 3. The ksubscribe method
    @prb.ksubscribe
    def _(y):
        print("ksubscribe", y)

    f(3)

# Prints:
# {"y": 0}
# subscribe {"y": 0}
# ksubscribe 0
# ...
```


3.2.3 Map, filter, reduce

Let's say you have a sequence and you want to print out the maximum absolute value. You can do it like this:

```
def f():
    y = 1
    y = -7
    y = 3
    y = 6
    y = -2

with probing("f > y") as prb:
    maximum = prb["y"].map(abs).max()
    maximum.print("The maximum is {}")

    f()

# Prints: The maximum is 7
```

- The [...] notation indexes each element in the stream (you can use it multiple times to get deep into the structure, if you're probing lists or dictionaries. There is also a `.getattr()` operator if you want to get deep into arbitrary objects)
- `map` maps a function to each element, here the absolute value
- `min` reduces the stream using the minimum function

Note: `map` is different from `subscribe`. The pipelines are lazy, so `map` might not execute if there is no subscriber down the pipeline.

If the stream interface is getting in your way and you would rather get the maximum value as an integer that you can manipulate normally, you have two (pretty much equivalent) options:

```
# With values()
with probing("f > y")["y"].map(abs).max().values() as values:
    f()

assert values == [7]

# With accum()
with probing("f > y") as prb:
    maximum = prb["y"].map(abs).max()
    values = maximum.accum()

    f()

assert values == [7]
```

That same advice goes for pretty much all the other operators.

3.2.4 Overriding values

Using `overridable=True`, Ptera's probes are able to override the values of the variables being probed (unless the probe is total; nonlocal variables are also not overridable). For example:

```
def f(x):
    hidden = 1
    return x + hidden

assert f(10) == 11

with probing("f > hidden", overridable=True) as prb:
    prb.override(2)

    assert f(10) == 12
```

The argument to `override()` can also be a function that takes the current value of the stream. Also see `koverride()`.

Warning: `override()` only overrides the **focus variable**. Recall that the focus variable is the one to the right of `>`, or the one prefixed with `!`.

This is because a Ptera selector is triggered when the focus variable is set, so realistically it is the only one that it makes sense to override.

Be careful, because it is easy to write misleading code:

```
# THIS WILL SET y = x + 1, NOT x
OverridableProbe("f(x) > y")["x"].override(lambda x: x + 1)
```

Note: `override` will only work at the end of a synchronous pipe (map/filter are OK, but not e.g. `sample`)

If the focus variable is the return value of a function (as explained in *Probe the return value*), `override` will indeed override that return value.

Note: Operations subscribed to `probing(selector, overridable=True)` happen before those that are subscribed to `probing(selector)`. If you want a probe to see the values after the override, that probe needs to be the non-overridable type, otherwise it will see the values before the override. You can use both probe types at the same time:

```
def f():
    return 1

with probing("f() as ret", overridable=True) as oprb:
    with probing("f() as ret") as prb:
        oprb.override(2)

        oprb.print() # will print {"ret": 1} (because concurrent with override)
        prb.print()  # will print {"ret": 2} (because after override)

    print(f())      # will print 2
```

3.2.5 Asserts

The `fail()` method can be used to raise an exception. If you put it after a filter, you can effectively fail when certain conditions occur. This can be a way to beef up a test suite.

```
def median(xs):
    # Don't copy this because it's incorrect if the length is even
    return xs[len(xs) // 2]

with probing("median > xs") as prb:
    prb.kfilter(lambda xs: len(xs) == 0).fail("List is empty!")
    prb.kfilter(lambda xs: list(sorted(xs)) != xs).fail("List is not sorted!")

median([])           # Fails immediately
median([1, 2, 5, 3, 4]) # Also fails
```

Note the use of the `kfilter()` operator, which receives the data as keyword arguments. Whenever it returns False, the corresponding datum is omitted from the stream. An alternative to using `kfilter` here would be to simply write `prb["xs"].filter(...)`.

3.2.6 Conditional breakpoints

Interestingly, you can use probes to set conditional breakpoints. Modifying the previous example:

```
def median(xs):
    return xs[len(xs) // 2]

with probing("median > xs") as prb:
    prb.kfilter(lambda xs: list(sorted(xs)) != xs).breakpoint()

median([1, 2, 5, 3, 4]) # Enters breakpoint
median([1, 2, 3, 4])    # Does not enter breakpoint
```

Using this code, you can set a breakpoint in `median` that is triggered only if the input list is not sorted. The breakpoint will occur wherever in the function the focus variable is set, in this case the beginning of the function since the focus variable is a parameter.

3.3 Selected operators

Here is a classification of available operators.

3.3.1 Filtering

- `filter()`: filter with a function
- `kfilter()`: filter with a function (keyword arguments)
- `where()`: filter based on keys and simple conditions
- `where_any()`: filter based on keys
- `keep()`: filter based on keys (+drop the rest)

- *distinct()*: only emit distinct elements
- *norepeat()*: only emit distinct consecutive elements
- *first()*: only emit the first element
- *last()*: only emit the last element
- *take()*: only emit the first n elements
- *take_last()*: only emit the last n elements
- *skip()*: suppress the first n elements
- *skip_last()*: suppress the last n elements

3.3.2 Mapping

- *map()*: map with a function
- *kmap()*: map with a function (keyword arguments)
- *augment()*: add extra keys using a mapping function
- *getitem()*: extract value for a specific key
- *sole()*: extract value from dict of length 1
- *as_()*: wrap as a dict

3.3.3 Reduction

- *reduce()*: reduce with a function
- *scan()*: emit a result at each reduction step
- *roll()*: reduce using overlapping windows
- *kmerge()*: merge all dictionaries in the stream
- *kscan()*: incremental version of *kmerge*

3.3.4 Arithmetic reductions

Most of these reductions can be called with the `scan` argument set to `True` to use `scan` instead of `reduce`. `scan` can also be set to an integer, in which case `roll` is used.

- *average()*
- *average_and_variance()*
- *count()*
- *max()*
- *min()*
- *sum()*
- *variance()*

3.3.5 Wrapping

- `give.wrap()`: give a special key at the beginning and end of a block
- `give.wrap_inherit()`: give a special key at the beginning and end of a block
- `give.inherit()`: add default key/values for every `give()` in the block
- `given.wrap()`: plug a context manager at the location of a `give.wrap`
- `given.kwrap()`: same as `wrap`, but pass kwargs

3.3.6 Timing

- `debounce()`: suppress events that are too close in time
- `sample()`: sample an element every `n` seconds
- `throttle()`: emit at most once every `n` seconds

3.3.7 Debugging

- `breakpoint()`: set a breakpoint whenever data comes in. Use this with filters.
- `tag()`: assigns a special word to every entry. Use with `breakword`.
- `breakword()`: set a breakpoint on a specific word set by `tag`, using the `BREAKWORD` environment variable.
- `print()`: print out the stream.
- `display()`: print out the stream (pretty).
- `accum()`: accumulate into a list.
- `values()`: accumulate into a list (context manager).
- `subscribe()`: run a task on every element.
- `ksubscribe()`: run a task on every element (keyword arguments).

3.4 Miscellaneous

3.4.1 Meta-variables

There are a few meta-variables recognized by Ptera that start with a hash sign:

- `#enter` is triggered immediately when entering a function. For example, if you want to set a breakpoint at the start of a function with no arguments you can use `probing("f > #enter").breakpoint()`.
- `#value` stands in for the return value of a function. `f()` as `x` is sugar for `f > #value as x`.
- `#error` stands for the exception raised by the function, if there is one.
- `#exit` is triggered when exiting a function, both on a normal return and when there is an error.
- `#yield` is triggered whenever a generator yields.
- `#receive` stands for the output of `yield`.

- `#loop_X` and `#endloop_X` are triggered respectively at the beginning and end of *each* iteration of a `for X in ...: loop` (the meta-variables are named after the iteration variable). If there are multiple iteration variables, you can use any of them. There is no way to differentiate loops that have the same iteration variables.

The `#enter` and `#receive` meta-variables both bear the `@enter` tag (meaning that they are points at which execution might enter the function). You can therefore refer to both using the selector `$x:@enter`. Conversely, `#exit` and `#yield` bear the `@exit` tag. You can leverage this feature to compute e.g. how much time is spent inside a function or generator.

3.4.2 Generic variables

It is possible to indiscriminately capture all variables from a function, or all variables that have a certain “tag”. Simply prefix a variable with `$` to indicate it is generic. When doing so, you will need to set `raw=True` if you want to be able to access the variable names. For example:

```
def f(a):
    b = a + 1
    c = b + 1
    d = c + 1
    return d

with probing("f > $x", raw=True) as prb:
    prb.print("{x.name} is {x.value}").

    f(10)

# Prints:
# a is 10
# b is 11
# c is 12
# d is 13
```

Note: `$x` will also pick up global and nonlocal variables, so if for example you use the `sum` builtin in the function, you will get an entry for `sum` in the stream. It will not pick up meta-variables such as `#value`, however.

3.4.3 Selecting based on tags

This feature admittedly clashes with type annotations, but Ptera recognizes a specific kind of annotation on variables:

```
def f(a):
    b = a + sum([1])
    c: "@Cool" = b + 1
    d: "@Cool & @Hot" = c + 1
    return d

with probing("f > $x:@Cool", raw=True) as prb:
    prb.print("{x.name} is {x.value}")

    f(10)
```

(continues on next page)

(continued from previous page)

```
# Prints:
# c is 12
# d is 13
```

In the above code, only variables tagged as `@Cool` will be instrumented. Multiple tags can be combined using the `&` operator.

3.4.4 Probe methods

Probing methods works as one would expect. When using a selector such as `self.f > x`, it will be interpreted as `cls.f(self = <self>) > x` so that it only triggers when it is called on this particular `self`.

3.4.5 Absolute references

Ptera inspects the locals and globals of the frame in which `probing` is called in order to figure out what to instrument. In addition to this system, there is a second system whereas each function corresponds to a unique reference. These references always start with `/`:

```
global_probe("/xyz.submodule/Klass/method > x")

# is essentially equivalent to:

from xyz.submodule import Klass
global_probe("Klass.method > x")
```

The slashes represent a physical nesting rather than object attributes. For example, `/module.submodule/x/y` means:

- Go in the file that defines `module.submodule`
- Enter `def x` or `class x` (it will *not* work if `x` is imported from elsewhere)
- Within that definition, enter `def y` or `class y`

The helper function `refstring()` can be used to get the absolute reference for a function.

Note:

- Unlike the normal notation, the absolute notation bypasses decorators. `/module/function` will probe the function inside the `def function(): ... in module.py`, so it will work even if the function was wrapped by a decorator (unless the decorator does not actually call the function).
 - Use `/module.submodule/func`, *not* `/module/submodule/func`. The former roughly corresponds to `from module.submodule import func` and the latter to `from module import submodule; func = submodule.func`, which can be different in Python. It's a bit odd, but it works that way to properly address Python quirks.
-

USE CASES

4.1 Instrumenting external code

There can be situations where you are interested in something an external library or program is computing, but is not easily available from its interface.

For example, if you are using someone else's code to train a neural network and are interested in how the training loss evolves, but that information is tucked inside a while loop, that can be a bit annoying to work with.

Instead of modifying the code to log the information you need, you can use Ptera to extract it.

For example, here is an example script to train a model on MNIST with Pytorch that you can download from Pytorch's main repository:

```
wget https://raw.githubusercontent.com/pytorch/examples/main/mnist/main.py
```

If you look at the code you can see that a loss variable is set in the train function. Let's do something with it.

Try running the following script instead of main.py (put that script in the same directory as main.py):

```
from main import main, train
from ptera import probing

if __name__ == "__main__":
    with probing("train > loss") as prb:
        (
            prb["loss"]          # Extract the loss variable
            .average(scan=100)   # Running average of last 100
            .throttle(1)         # Produce at most once per second
            .print("Loss = {}")
        )

    # Run the original script within context of the probe
    main()
```

In addition to the original script's output, you will now get new output that corresponds to the running average of the last 100 training losses, reported at most once per second.

Tip: If you like the idea of using this for logging data in your own scripts because of how powerful the probe interface is, you certainly can! But you can have the same interface in a more explicit way with the [giving](#) library, using `give/given` instead of `probing`.

4.2 Advanced logging

Since probes are defined outside of the code they instrument, they can be used to log certain metrics without littering the main program. These logs can be easily augmented with information from outer scopes, limited using throttling, reduced in order to compute an average/min/max, and so on.

4.3 Advanced debugging

Probes have a `breakpoint()` method. Coupled with operators such as `filter()`, it is easy to define reusable conditional breakpoints. For example:

```
from ptera import probing

def f(x):
    y = x * x
    return y

with probing("f > x") as prb:
    prb["x"].filter(lambda x: x == 2).breakpoint()

    f(1)
    f(2)  # <- will set a breakpoint at the start
    f(3)
```

Such breakpoints should work regardless of the IDE you use, and they should be robust to most code changes, short of changing variable and function names.

Using operators like `pairwise()`, you can also set breakpoints that activate if a variable increases or decreases in value.

4.4 Testing

Ptera's ability to extract arbitrary variables from multiple scopes can be very useful for writing tests that verify conditions about a program or library's internal state.

See *Testing with Ptera* for detailed examples.

TESTING WITH PTERA

Ptera is a general instrumentation framework for the inner state of Python programs and can be used to test that certain conditions obtain deep within the code.

For example: perhaps a function only works properly on sorted lists and you want to test that every time it is called, the input is sorted (or some other invariant). Ptera allows you to do this simply, composably, and in a way that is generally easy to debug.

In a nutshell, you can test:

- *Properties*: test that variable X in function F is sorted, or any other invariant that the code is supposed to keep
- *Information flow*: test that variable X in function F matches variable Y in function G.
- *Infinite loops*: limit how many times a function can be called within a test
- *Trends*: test that variable X monotonically decreases/increases/etc. within function F
- *Caching*: test that call results that are supposed to be cached are not recomputed

Many of these tests could be done with clever monkey patching, but they are a lot simpler using Ptera, and composable.

Note: If you want to test a particular property in many different situations, for instance through a battery of integration tests, you can abstract it into a fixture and easily apply it to many tests, or even to all tests.

5.1 Test properties

Some libraries have to do bookkeeping on data structures, ensuring certain invariants (element types, proper sorting, lack of duplicates, etc.) Ptera can be used to verify these invariances during testing, anywhere that's relevant. For example, here's how you could test that a bisect function receives a sorted array:

```
from ptera import probing

def bisect(arr, key):
    lo = -1
    hi = len(arr)
    while lo < hi - 1:
        mid = lo + (hi - lo) // 2
        if (elem := arr[mid]) > key:
            hi = mid
        else:
            lo = mid
```

(continues on next page)

(continued from previous page)

```

    return lo + 1

def test_bisect_argument_sorted():
    with probing("bisect > arr") as prb:
        # First: set up the pipeline
        (
            prb
            .kfilter(lambda arr: list(sorted(arr)) != arr)
            .fail("Input arr is not sorted")
        )

        # Second: call the functionality to test
        something_that_calls_bisect()

```

The probe: `bisect > arr` is triggered when the `arr` variable in the `bisect` function is set. Since `arr` is a parameter, that corresponds to the entry of the function.

The pipeline:

- `kfilter()` runs a function on every entry, with the arguments passed as keyword arguments, so it is important to name the argument `arr` in this case. It only keeps elements where the return value is truthy. Here it will only keep the arrays that are *not* sorted.
- `fail()` raises an exception whenever it receives anything. Because of the `kfilter`, `fail` will only get data if we see an array `arr` that is not properly sorted.

The tested functionality in `something_that_calls_bisect` must be executed *after* the pipeline is set up, but it can be arbitrarily complex. When a failure occurs, the traceback will be situated at the beginning of the offending `bisect` call.

5.2 Test information flow

There are many situations where you provide an argument to a top level function and you expect its value to bubble down to subroutines. This can be a source of subtle bugs, especially if these subroutines have default parameters that you forget to pass them along (silent bugs). Oftentimes this could be checked by verifying the program's expected output, but that can be tricky for very complex programs and it makes the test sensitive to many other bugs.

Ptera can help you verify that the information flow is as you expect it:

```

def g(x, opt=0):
    return x * opt

def f(x, opt=0):
    return g(x + 1) # BUG: should be g(x + 1, opt=opt)

def test_flow():
    with probing("f(opt as fopt) > g(!opt as gopt)") as prb:
        prb.fail_if_empty()
        prb.kfilter(lambda fopt, gopt: fopt != gopt).fail()

        f(10, opt=11) # Fails!

```

The probe: `f(opt as fopt) > g(!opt as gopt)` is triggered when `g` is called within `f`, and the `opt` variable or parameter in `g` is set.

- The `!` denotes the *focus variable*. When that variable is set, the pipeline is activated.
- Two variables are collected: `opt` in `f` which we rename `fopt`, and `opt` in `g` which we rename `gopt`.

The pipeline:

- `fail_if_empty()` ensures that the selector is triggered at least once. This is a recommended sanity check to make sure that the test is doing something!
- The `kfilter()` method will be fed both of our variables as keyword arguments. This means that the parameter names of the lambda must be the same as the variable names.
- `kfilter` will only produce the elements where `fopt` and `gopt` are not the same (where the lambda returns `True`).
- `fail()` will raise an exception whenever it receives anything. Because of the `kfilter`, `fail` will only get data if `fopt` and `gopt` differ (which is the precise error we want the test to catch).

5.3 Test for infinite loops

The following test will check that the function `f` is called no more than a thousand times during the test:

```
def loopy(i):
    while i != 0:
        f()
        i = i - 1

def test_loopy():
    with probing("f > #enter") as prb:
        prb.skip(1000).fail()

    loopy(-1) # Fails
```

The probe: `f > #enter` uses the special variable `#enter` that is triggered immediately at the start of `f`. Every time `f` is called, this pipeline is triggered.

Note: In this example, you could also set a probe on `loopy > i`. It is up to you to choose what makes the most sense.

The pipeline:

- `skip()` will throw away the first thousand entries in the pipeline, corresponding to the first 1000 calls to `f`.
- `fail()` will fail whenever it sees anything. If `f` is called less than 1000 times, all calls are skipped and there will be no failure. Otherwise, the 1001st call will trigger a failure.

Of course, this test can be adapted to check that a function is called once or more (use `fail_if_empty()`), or a specific number of times (`count().filter(lambda x: x != expected_count).fail()`).

5.4 Test trends

Another great use for Ptera is to check for trends in the values of certain variables in the program as it progresses. Perhaps they must be monotonically increasing or decreasing, perhaps they should be convergent, and so on.

For example, let's say you want to verify that a variable in a loop always goes down:

```
def loopy(i, step):
    while i != 0:
        f()
        i = i - step

def test_loopy():
    with probing("loopy > i") as prb:
        (
            prb["i"]
            .pairwise()
            .starmap(lambda i1, i2: i2 - i1)
            .filter(lambda x: x >= 0)
            .fail()
        )

    loopy(10, 0) # Fails
```

The probe: `loopy > i` is triggered when `i` is set in `loopy`. Being passed as an argument counts as being set.

The pipeline:

- `prb["i"]` extracts the field named `"i"`.
- `pairwise()` pairs consecutive elements. It will transform the sequence `(x, y, z, ...)` into `((x, y), (y, z), ...)`. Therefore, after this operator, we have pairs of successive values taken by `i`.
- `starmap()` applies a function on each tuple as a list of arguments, so the pairs we just created are passed as two separate argument. We compute the difference between them.
- `filter()` applies on the differences we just created. Unlike `kfilter` it does not take the arguments as keyword arguments, just the raw values we have so far.
- `fail()` will fail as soon as we detect a non-negative difference.

5.5 Test caching

In this example, we test that a function is never called twice with the same argument. For example, maybe it computes something expensive, so we want to cache the results, and we want to make sure the cache is properly used.

```
cache = {}

def _expensive(x):
    return x * x # oof! so expensive

def expensive(x):
    if x in cache:
        return cache[x]
    else:
```

(continues on next page)

(continued from previous page)

```
# We forget to populate the cache
return _expensive(x)

def test_expensive():
    with probing("_expensive > x") as prb:
        xs = prb["x"].accum()

        expensive(12)
        expensive(12) # again

    assert len(set(xs)) == len(xs) > 0 # fails
```

The probe: `_expensive > x` instruments the argument `x` of `_expensive`. It is important to probe the function that unconditionally does the computation in this case.

The pipeline:

- `prb["x"]` extracts the field named "x".
- `accum()` creates a (currently empty) list and returns it. Every time the probe is activated, the current value is appended to the list.
- After calling `expensive` twice, we can look at what's in the list. Here we could simply check that it only contains one element, but more generally we can check that its distinct elements (`set(xs)`) are exactly as numerous as the complete list, from which we can conclude that there are no duplicates.
- The `> 0` is added for good measure, to make sure we are not testing a dud that never calls `_expensive` at all.

You can of course do whatever you want with the list returned by `accum`, which is what makes it very polyvalent. You only have to make sure not to use it until *after* the `probing` block concludes, especially if you accumulate the result of a reduction operator like `min` or `average`.

REFERENCE

6.1 Main API

This page collates the main API functions. Other reference files contain further details that may or may not be relevant to typical users.

- *Probing API*
- *Overlay API*
- *Low level API*

6.1.1 Probing API

The preferred API to Ptera's functionality. It is the most powerful.

<i>probing()</i>	Context manager for probing
<i>global_probe()</i>	Create a global probe
<i>Probe</i>	Probe class returned by probing and global_probe

6.1.2 Overlay API

The Overlay API is more low level than the probing API (the latter uses the former under the hood).

<i>tooled()</i>	Transform a function to report variable changes
<i>is_tooled()</i>	Return whether a function is tooled or not
<i>autotool()</i>	Automatically tool inplace all functions a selector refers to
<i>BaseOverlay</i>	Simple context manager to apply handlers corresponding to selectors
<i>Overlay</i>	A BaseOverlay with extra functionality
<i>Immediate</i>	A handler that triggers when the focus variable is set
<i>Total</i>	A handler that triggers when the outer scope for a selector ends

6.1.3 Low level API

<code>select()</code>	Parse a string into a <i>Selector</i> object
<code>transform()</code>	Transform a function to filter its behavior

6.2 List of operators

The operators listed here are all available as methods on the *Probe* objects yielded by `probing()`.

Important: Ptera's operators come from the `giving` package, which itself derives most of its operators from the `rx` package.

Since most of these operators are defined in different packages, the documentation might not fully match how they are used with Ptera. So, keep this in mind:

- with `given()` as `gv` works the same as with `probing(...)` as `prb`. They are different streams, but they have the same interface and the same operators are defined on them.
- Anything that is done on a variable named `gv` also works on a probe.
- The operators defined in the `rx` package are not originally methods, because they have decided to use a different API. However, Ptera does offer them as methods with the same name. So if you see code such as `op = contains(42)`, that means you can call `probe.contains(42)` (which, to be clear, is equivalent to `probe.pipe(contains(42))`).

`giving.operators.affix(**streams)`

Affix streams as extra keys on an existing stream of dicts.

The affixed streams should have the same length as the main one, so when affixing a reduction, one should set

`scan=True`, or `scan=n`.

marble\sphinxhyphen-{}f4cfbdab21b5ae227f5ec6aad9f42d44e04dc39.png

Example

```
obs.where("x", "y").affix(
    minx=obs["x"].min(scan=True),
    xpy=obs["x", "y"].starmap(lambda x, y: x + y),
)
```

Or:

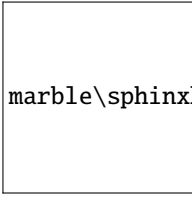
```
obs.where("x", "y").affix(
    # o is obs.where("x", "y")
    minx=lambda o: o["x"].min(scan=True),
    xpy=lambda o: o["x", "y"].starmap(lambda x, y: x + y),
)
```

Parameters **streams** – A mapping from extra keys to add to the dicts to Observables that generate the values, or to functions of one argument that will be called with the main Observable.

`giving.operators.all(predicate)`

Determines whether all elements of an observable sequence satisfy a condition.

9ae3e04b8e979d8accefa93939e26a38d0d435bb.png



Example

```
>>> op = all(lambda value: value.length > 3)
```

Parameters **predicate** (Callable[[~_T], bool]) – A function to test each element for a condition.

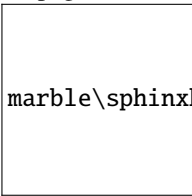
Return type Callable[[Observable[~_T]], Observable[bool]]

Returns An operator function that takes an observable source and returns an observable sequence containing a single element determining whether all elements in the source sequence pass the test in the specified predicate.

`giving.operators.amb(right_source)`

Propagates the observable sequence that reacts first. f87bbff3e9ad64d121cc9d2ab3d1aacd4d9015c8.png

marble\sphinxhyphen {}f87bbff3e9ad64d121cc9d2ab3d1aacd4d9015c8.png



Example

```
>>> op = amb(ys)
```

Return type Callable[[Observable[~_T]], Observable[~_T]]

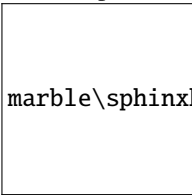
Returns An operator function that takes an observable source and returns an observable sequence that surfaces any of the given sequences, whichever reacted first.

`giving.operators.as_(key)`

Make a stream of dictionaries using the given key.

For example, `[1, 2].as_("x") => [{"x": 1}, {"x": 2}]` ff061583c7cbf3a16a4487bcf4369a37d7305e8f.png

marble\sphinxhyphen {}ff061583c7cbf3a16a4487bcf4369a37d7305e8f.png



Parameters **key** – Key under which to generate each element of the stream.

`giving.operators.as_observable()`

Hides the identity of an observable sequence.

Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns An operator function that takes an observable source and returns an observable sequence that hides the identity of the source sequence.

`giving.operators.augment(**fns)`

Augment a stream of dicts with new keys.

Each key in `fns` should be associated to a function that will be called with the rest of the data as keyword arguments, so the argument names matter. The results overwrite the old data, if any keys are in common.

Note: The functions passed in `fns` will be wrapped with `lax_function()` if possible.

This means that these functions are considered to have an implicit `**kwargs` argument, so that any data they do not need is ignored.

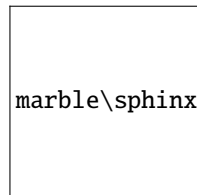
```
# [{"x": 1, "y": 2}, ...] => [{"x": 1, "y": 2, "z": 3}, ...]
gv.augment(z=lambda x, y: x + y)

# [{"lo": 2, "hi": 3}, ...] => [{"lo": 2, "hi": 3, "higher": 9}, ...]
gv.augment(higher=lambda hi: hi * hi)
```

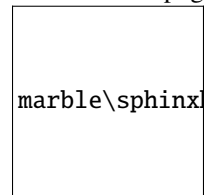
Parameters **fns** – A map from new key names to the functions to compute them.

`giving.operators.average(*, scan=False)`

Produce the average of a stream of values. 9a9098a73855ea49525d17a3eae8437882192b92.png

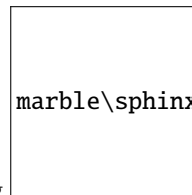


marble\sphinxhyphen {}9a9098a73855ea49525d17a3eae8437882192b92.png



marble\sphinxhyphen {}e911a

e911a425a8492e32ea772ac02d0773656ddb4f18.png



marble\sphinxhyphen {}649f84d747fb71121f2e58a2dd3a2c00bf37

649f84d747fb71121f2e58a2dd3a2c00bf37e3ed.png

Parameters

- **scan** – If True, generate the current average on every element. If a number *n*, generate the average on the last *n* elements.
- **seed** – First element of the reduction.

`giving.operators.average_and_variance(*, scan=False)`

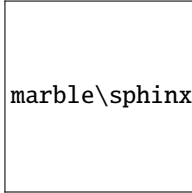
Produce the average and variance of a stream of values.

Note: The variance for the first element is always None.

Parameters **scan** – If True, generate the current average+variance on every element. If a number *n*, generate the average+variance on the last *n* elements.

giving.operators.**bottom**(*n=10, key=None, reverse=False*)

Return the bottom *n* values, sorted in ascending order. 274eb497c61eca9b0c5d5b1e5a0e07222b2add05.png



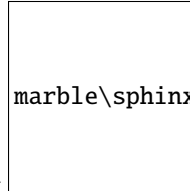
bottom may emit less than *n* elements, if there are less than *n* elements in the original sequence.

Parameters

- **n** – The number of bottom entries to return.
- **key** – The comparison key function to use or a string.

giving.operators.**buffer**(*boundaries*)

Projects each element of an observable sequence into zero or more buffers.



81dbcd0cd7a21fb6c1f4939387f5ff16b1556101.png

Examples

```
>>> res = buffer(reactivex.interval(1.0))
```

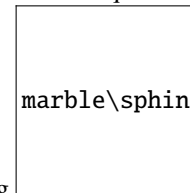
Parameters **boundaries** (Observable[Any]) – Observable sequence whose elements denote the creation and completion of buffers.

Return type Callable[[Observable[~_T]], Observable[List[~_T]]]

Returns A function that takes an observable source and returns an observable sequence of buffers.

giving.operators.**buffer_toggle**(*openings, closing_mapper*)

Projects each element of an observable sequence into zero or more buffers.



9c9ad906399c982a7ca5eb85fd1ce3d2c4ab39a5.png

```
>>> res = buffer_toggle(reactivex.interval(0.5), lambda i: reactivex.timer(i))
```

Parameters

- **openings** (Observable[Any]) – Observable sequence whose elements denote the creation of buffers.

- **closing_mapper** (Callable[[Any], Observable[Any]]) – A function invoked to define the closing of each produced buffer. Value from openings Observable that initiated the associated buffer is provided as argument to the function. The buffer is closed when one item is emitted or when the observable completes.

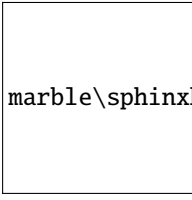
Return type Callable[[Observable[~_T]], Observable[List[~_T]]]

Returns A function that takes an observable source and returns an observable sequence of windows.

giving.operators.**buffer_when**(closing_mapper)

Projects each element of an observable sequence into zero or more buffers.

1a971c0609b22ba1f5d5d63abfc62be378139340.png



Examples

```
>>> res = buffer_when(lambda: reactivex.timer(0.5))
```

Parameters **closing_mapper** (Callable[[], Observable[Any]]) – A function invoked to define the closing of each produced buffer. A buffer is started when the previous one is closed, resulting in non-overlapping buffers. The buffer is closed when one item is emitted or when the observable completes.

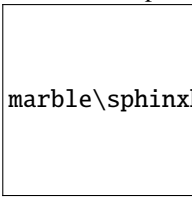
Return type Callable[[Observable[~_T]], Observable[List[~_T]]]

Returns A function that takes an observable source and returns an observable sequence of windows.

giving.operators.**buffer_with_count**(count, skip=None)

Projects each element of an observable sequence into zero or more buffers which are produced based on element

count information. 58fe81f8f4a247bd862f50ae3cc219a98197ff80.png



Examples

```
>>> res = buffer_with_count(10)(xs)
>>> res = buffer_with_count(10, 1)(xs)
```

Parameters

- **count** (int) – Length of each buffer.
- **skip** (Optional[int]) – [Optional] Number of elements to skip between creation of consecutive buffers. If not provided, defaults to the count.

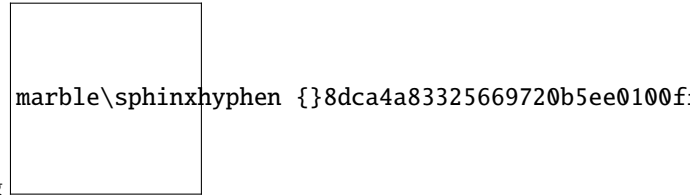
Return type Callable[[Observable[~_T]], Observable[List[~_T]]]

Returns A function that takes an observable source and returns an observable sequence of buffers.

giving.operators.buffer_with_time(timespan, timeshift=None, scheduler=None)

Projects each element of an observable sequence into zero or more buffers which are produced based on timing

information. 8dca4a83325669720b5ee0100ff548f37f73038f.png



Examples

```
>>> # non-overlapping segments of 1 second
>>> res = buffer_with_time(1.0)
>>> # segments of 1 second with time shift 0.5 seconds
>>> res = buffer_with_time(1.0, 0.5)
```

Parameters

- **timespan** (Union[timedelta, float]) – Length of each buffer (specified as a float denoting seconds or an instance of timedelta).
- **timeshift** (Union[timedelta, float, None]) – [Optional] Interval between creation of consecutive buffers (specified as a float denoting seconds or an instance of timedelta). If not specified, the timeshift will be the same as the timespan argument, resulting in non-overlapping adjacent buffers.
- **scheduler** (Optional[SchedulerBase]) – [Optional] Scheduler to run the timer on. If not specified, the timeout scheduler is used

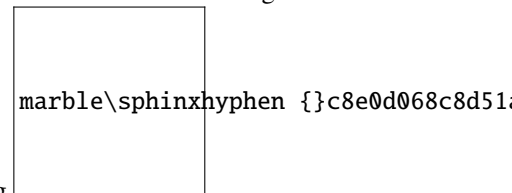
Return type Callable[[Observable[~_T]], Observable[List[~_T]]]

Returns An operator function that takes an observable source and returns an observable sequence of buffers.

giving.operators.buffer_with_time_or_count(timespan, count, scheduler=None)

Projects each element of an observable sequence into a buffer that is completed when either it's full or a given

amount of time has elapsed. c8e0d068c8d51a7ab5499296538b2c74025dfe40.png



Examples

```
>>> # 5s or 50 items in an array
>>> res = source._buffer_with_time_or_count(5.0, 50)
>>> # 5s or 50 items in an array
>>> res = source._buffer_with_time_or_count(5.0, 50, Scheduler.timeout)
```

Parameters

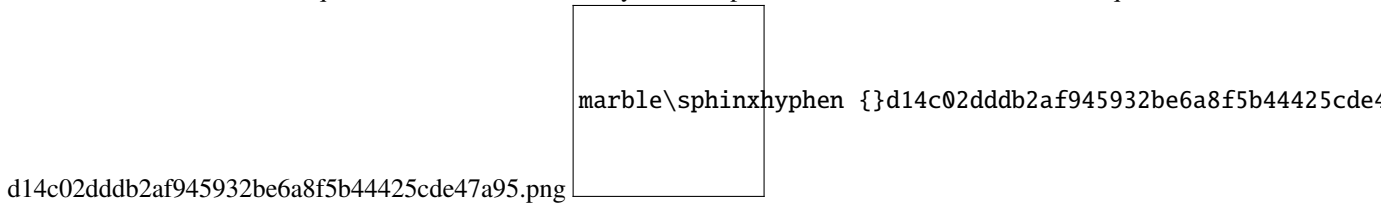
- **timespan** (Union[timedelta, float]) – Maximum time length of a buffer.
- **count** (int) – Maximum element count of a buffer.
- **scheduler** (Optional[SchedulerBase]) – [Optional] Scheduler to run buffering timers on. If not specified, the timeout scheduler is used.

Return type Callable[[Observable[~T]], Observable[List[~T]]]

Returns An operator function that takes an observable source and returns an observable sequence of buffers.

giving.operators.**catch**(*handler*)

Continues an observable sequence that is terminated by an exception with the next observable sequence.



Examples

```
>>> op = catch(ys)
>>> op = catch(lambda ex, src: ys(ex))
```

Parameters handler (Union[Observable[~T], Callable[[Exception, Observable[~T]], Observable[~T]]) – Second observable sequence used to produce results when an error occurred in the first sequence, or an exception handler function that returns an observable sequence given the error and source observable that occurred in the first sequence.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns A function taking an observable source and returns an observable sequence containing the first sequence's elements, followed by the elements of the handler sequence in case an exception occurred.

giving.operators.**collect_between**(*start*, *end*, *common=None*)

Collect all data between the start and end keys.

Example

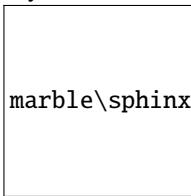
```
with given() as gv:
    gv.collect_between("A", "Z") >> (results := [])
    give(A=1)
    give(B=2)
    give(C=3, D=4, A=5)
    give(Z=6)
    assert results == [{"A": 5, "B": 2, "C": 3, "D": 4, "Z": 6}]
```

Parameters

- **start** – The key that marks the beginning of the accumulation.
- **end** – The key that marks the end of the accumulation.
- **common** – A key that must be present in all data and must have the same value in the whole group.

giving.operators.**combine_latest**(*others)

Merges the specified observable sequences into one observable sequence by creating a tuple whenever any of the observable sequences produces an element. 86e55845149aaa02237e62b0ba0b70d2fbaf984f.png



marble\sphinxhyphen {}86e55845149aaa02237e62b0ba0b70d2fbaf984f.png

Examples

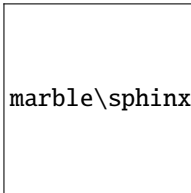
```
>>> obs = combine_latest(other)
>>> obs = combine_latest(obs1, obs2, obs3)
```

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable sources and returns an observable sequence containing the result of combining elements of the sources into a tuple.

giving.operators.**concat**(*sources)

Concatenates all the observable sequences. 4544dd242d02df82ac059a82701a17b5b31135d4.png



marble\sphinxhyphen {}4544dd242d02df82ac059a82701a17b5b31135d4.png

Examples

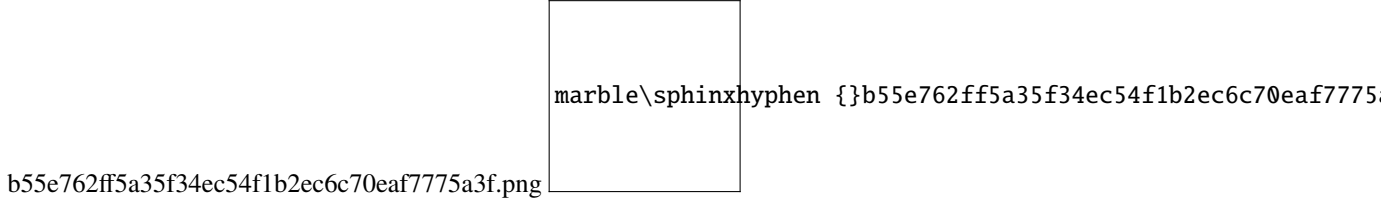
```
>>> op = concat(xs, ys, zs)
```

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes one or more observable sources and returns an observable sequence that contains the elements of each given sequence, in sequential order.

giving.operators.contains(value, comparer=None)

Determines whether an observable sequence contains a specified element with an optional equality comparer.



Examples

```
>>> op = contains(42)
>>> op = contains({ "value": 42 }, lambda x, y: x["value"] == y["value"])
```

Parameters

- **value** (~T) – The value to locate in the source sequence.
- **comparer** (Optional[Callable[[~T, ~T], bool]]) – [Optional] An equality comparer to compare elements.

Return type Callable[[Observable[~T]], Observable[bool]]

Returns A function that takes a source observable that returns an observable sequence containing a single element determining whether the source sequence contains an element that has the specified value.

giving.operators.count(*, predicate=None, scan=False)

Count operator.

Returns an observable sequence containing a value that represents how many elements in the specified observable sequence satisfy a condition if provided, else the count of items.

Parameters

- **predicate** – A function to test each element for a condition.
- **scan** – If True, generate a running count, if a number *n*, count the number of elements/matches in the last *n* elements.

giving.operators.debounce(duetime, scheduler=None)

Ignores values from an observable sequence which are followed by another value before duetime.

ddf862178779eaa8593df342589be502be49ddb.png

marble\sphinxhyphen-{}ddf862178779eaa8593df342589be502be49ddb.png

Example

```
>>> res = debounce(5.0) # 5 seconds
```

Parameters

- **duetime** (Union[timedelta, float]) – Duration of the throttle period for each value (specified as a float denoting seconds or an instance of timedelta).
- **scheduler** (Optional[SchedulerBase]) – Scheduler to debounce values on.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes the source observable and returns the debounced observable sequence.

giving.operators.default_if_empty(*default_value: reactivex.operators._T*) →
 Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
 reactivex.observable.observable.Observable[reactivex.operators._T]]

giving.operators.default_if_empty() →
 Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
 reactivex.observable.observable.Observable[Optional[reactivex.operators._T]]]

Returns the elements of the specified sequence or the specified value in a singleton sequence if the sequence is

empty. bb4eb7511f251165be29bce867fc77f1f2e56724.png

marble\sphinxhyphen-{}bb4eb7511f251165be29bce867fc77f1f2e56724.png

Examples

```
>>> res = obs = default_if_empty()
>>> obs = default_if_empty(False)
```

Parameters **default_value** (Optional[Any]) – The value to return if the sequence is empty. If not provided, this defaults to None.

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the specified default value if the source is empty otherwise, the elements of the source.

giving.operators.**delay**(duetime, scheduler=None)

marble\sphinxhyphen-{}1c57db95ece5fc3af60515bb5b99603381d3f000.png

The delay operator. 1c57db95ece5fc3af60515bb5b99603381d3f000.png

Time shifts the observable sequence by duetime. The relative time intervals between the values are preserved.

Examples

```
>>> res = delay(timedelta(seconds=10))
>>> res = delay(5.0)
```

Parameters

- **duetime** (Union[timedelta, float]) – Relative time, specified as a float denoting seconds or an instance of timedelta, by which to shift the observable sequence.
- **scheduler** (Optional[SchedulerBase]) – [Optional] Scheduler to run the delay timers on. If not specified, the timeout scheduler is used.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns A partially applied operator function that takes the source observable and returns a time-shifted sequence.

giving.operators.**delay_subscription**(duetime, scheduler=None)

Time shifts the observable sequence by delaying the subscription. 1c57db95ece5fc3af60515bb5b99603381d3f000.png

marble\sphinxhyphen-{}1c57db95ece5fc3af60515bb5b99603381d3f000.png

Example

```
>>> res = delay_subscription(5.0) # 5s
```

Parameters

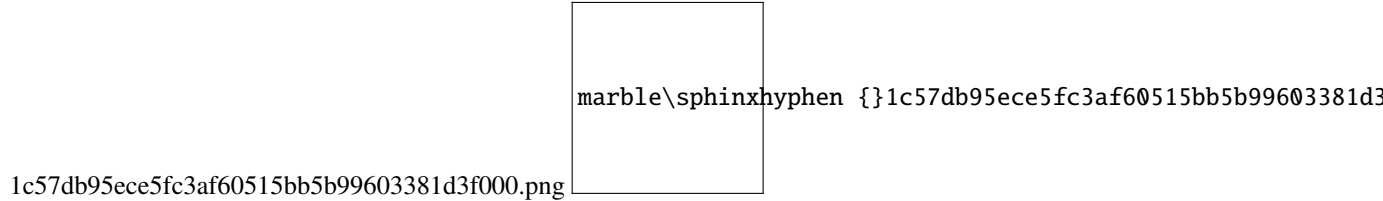
- **duetime** (Union[datetime, timedelta, float]) – Absolute or relative time to perform the subscription
- **at.** –
- **scheduler** (Optional[SchedulerBase]) – Scheduler to delay subscription on.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns A function that take a source observable and returns a time-shifted observable sequence.

`giving.operators.delay_with_mapper(subscription_delay=None, delay_duration_mapper=None)`

Time shifts the observable sequence based on a subscription delay and a delay mapper function for each element.



Examples

```
>>> # with mapper only
>>> res = source.delay_with_mapper(lambda x: Scheduler.timer(5.0))
>>> # with delay and mapper
>>> res = source.delay_with_mapper(
    reactivex.timer(2.0), lambda x: reactivex.timer(x)
)
```

Parameters

- **subscription_delay** (Union[Observable[Any], Callable[[Any], Observable[Any]], None]) – [Optional] Sequence indicating the delay for the subscription to the source.
- **delay_duration_mapper** (Optional[Callable[[~T], Observable[Any]]]) – [Optional] Selector function to retrieve a sequence indicating the delay for each given element.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns A function that takes an observable source and returns a time-shifted observable sequence.

`giving.operators.dematerialize()`

Dematerialize operator.

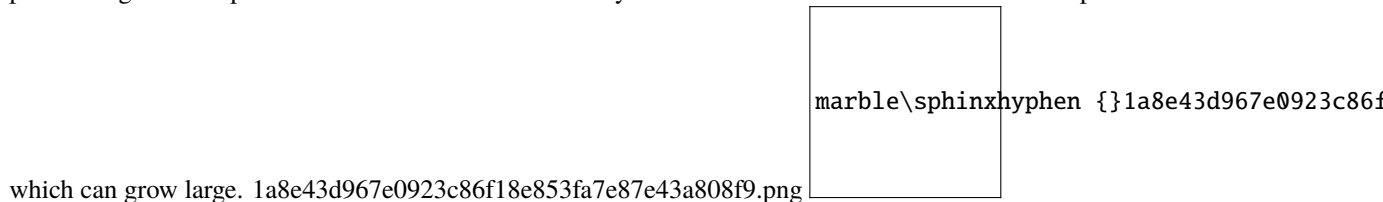
Dematerializes the explicit notification values of an observable sequence as implicit notifications.

Return type Callable[[Observable[Notification[~T]], Observable[~T]]

Returns An observable sequence exhibiting the behavior corresponding to the source sequence's notification values.

`giving.operators.distinct(key_mapper=None, comparer=None)`

Returns an observable sequence that contains only distinct elements according to the key_mapper and the comparer. Usage of this operator should be considered carefully due to the maintenance of an internal lookup structure



Examples

```
>>> res = obs = xs.distinct()
>>> obs = xs.distinct(lambda x: x.id)
>>> obs = xs.distinct(lambda x: x.id, lambda a,b: a == b)
```

Parameters

- **key_mapper** (Optional[Callable[[~_T], ~_TKey]]) – [Optional] A function to compute the comparison key for each element.
- **comparer** (Optional[Callable[[~_TKey, ~_TKey], bool]]) – [Optional] Used to compare items in the collection.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence only containing the distinct elements, based on a computed key value, from the source sequence.

giving.operators.**distinct_until_changed**(key_mapper=None, comparer=None)

Returns an observable sequence that contains only distinct contiguous elements according to the key_mapper

marble\sphinxhyphen-{}3783ceec8ed2d9cdf16e74

and the comparer. 3783ceec8ed2d9cdf16e748d7f8c2f8271ddbb2f.png

Examples

```
>>> op = distinct_until_changed();
>>> op = distinct_until_changed(lambda x: x.id)
>>> op = distinct_until_changed(lambda x: x.id, lambda x, y: x == y)
```

Parameters

- **key_mapper** (Optional[Callable[[~_T], ~_TKey]]) – [Optional] A function to compute the comparison key for each element. If not provided, it projects the value.
- **comparer** (Optional[Callable[[~_TKey, ~_TKey], bool]]) – [Optional] Equality comparer for computed key values. If not provided, defaults to an equality comparer function.

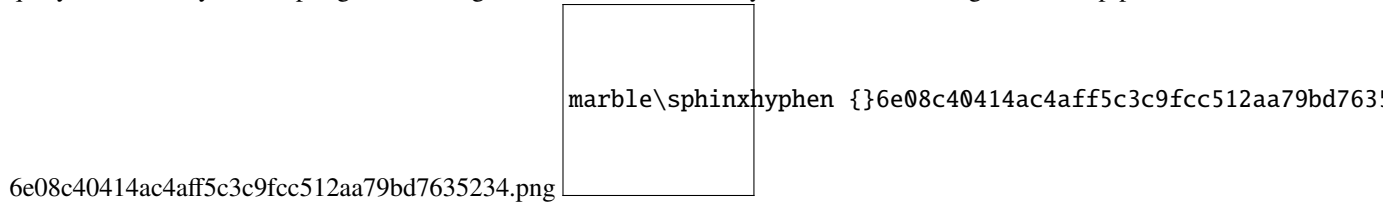
Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence only containing the distinct contiguous elements, based on a computed key value, from the source sequence.

giving.operators.**do**(observer)

Invokes an action for each element in the observable sequence and invokes an action on graceful or exceptional termination of the observable sequence. This method can be used for debugging, logging, etc. of

query behavior by intercepting the message stream to run arbitrary actions for messages on the pipeline.



```
>>> do(observer)
```

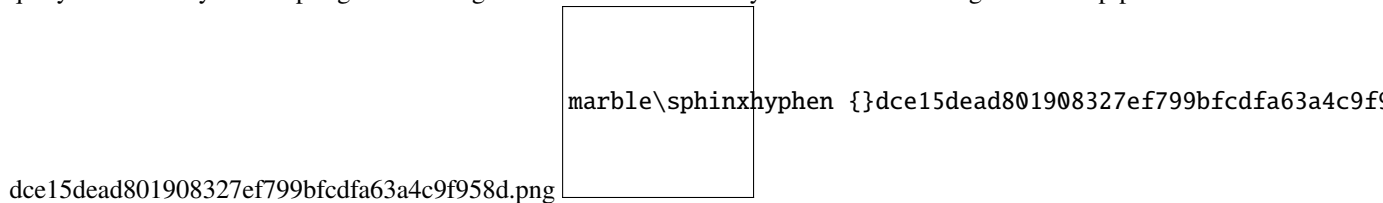
Parameters **observer** (ObserverBase[~_T]) – Observer

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes the source observable and returns the source sequence with the side-effecting behavior applied.

giving.operators.do_action(*on_next=None, on_error=None, on_completed=None*)

Invokes an action for each element in the observable sequence and invokes an action on graceful or exceptional termination of the observable sequence. This method can be used for debugging, logging, etc. of query behavior by intercepting the message stream to run arbitrary actions for messages on the pipeline.



Examples

```
>>> do_action(send)
>>> do_action(on_next, on_error)
>>> do_action(on_next, on_error, on_completed)
```

Parameters

- **on_next** (Optional[Callable[[~_T], None]]) – [Optional] Action to invoke for each element in the observable sequence.
- **on_error** (Optional[Callable[[Exception], None]]) – [Optional] Action to invoke on exceptional termination of the observable sequence.
- **on_completed** (Optional[Callable[[], None]]) – [Optional] Action to invoke on graceful termination of the observable sequence.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes the source observable an returns the source sequence with the side-effecting behavior applied.

giving.operators.do_while(*condition*)

Repeats source as long as condition holds emulating a do while loop.

74f3910edce12fac1a74594adad08937786aea97.png

marble\sphinxhyphen {}74f3910edce12fac1a74594adad08937786a

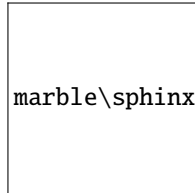
Parameters **condition** (Callable[[Observable[~_T]], bool]) – The condition which determines if the source will be repeated.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An observable sequence which is repeated as long as the condition holds.

giving.operators.**element_at**(*index*)

Returns the element at a specified index in a sequence. 57e895b2baf9d4fe3d4067802334f4ef144153a0.png



marble\sphinxhyphen {}57e895b2baf9d4fe3d4067802334f4ef144153a0.png

Example

```
>>> res = source.element_at(5)
```

Parameters **index** (int) – The zero-based index of the element to retrieve.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence that produces the element at the specified position in the source sequence.

giving.operators.**element_at_or_default**(*index*, *default_value=None*)

Returns the element at a specified index in a sequence or a default value if the index is out of range.

f6ff9e502837f3c34a2c23a40437f3ab7ecc9835.png

marble\sphinxhyphen {}f6ff9e502837f3c34a2c23a40437f3ab7ecc9

Example

```
>>> res = source.element_at_or_default(5)
>>> res = source.element_at_or_default(5, 0)
```

Parameters

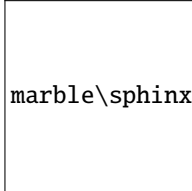
- **index** (int) – The zero-based index of the element to retrieve.
- **default_value** (Optional[~_T]) – [Optional] The default value if the index is outside the bounds of the source sequence.

Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns A function that takes an observable source and returns an observable sequence that produces the element at the specified position in the source sequence, or a default value if the index is outside the bounds of the source sequence.

`giving.operators.exclusive()`

Performs a exclusive waiting for the first to finish before subscribing to another observable. Observables that come in between subscriptions will be dropped on the floor. 0718300e5a5496454a339cdc4721f9f4e0411a17.png



Return type `Callable[[Observable[Observable[~_T]], Observable[~_T]]`

Returns An exclusive observable with only the results that happen when subscribed.

`giving.operators.expand(mapper)`

Expands an observable sequence by recursively invoking mapper.

Parameters **mapper** (`Callable[[~_T], Observable[~_T]]`) – Mapper function to invoke for each produced element, resulting in another sequence to which the mapper will be invoked recursively again.

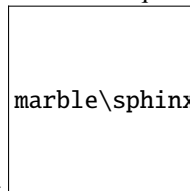
Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns An observable sequence containing all the elements produced

by the recursive expansion.

`giving.operators.filter(predicate)`

Filters the elements of an observable sequence based on a predicate.



2ac5743543b7ddb0ba7b7dc3cedacda9de1a73eb.png

Example

```
>>> op = filter(lambda value: value < 10)
```

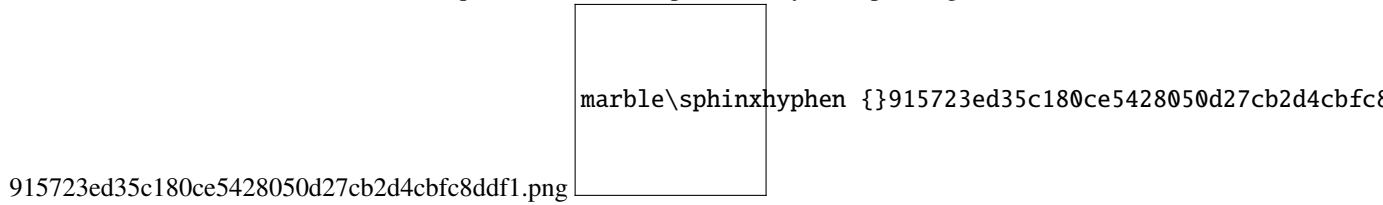
Parameters **predicate** (`Callable[[~_T], bool]`) – A function to test each source element for a condition.

Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns An operator function that takes an observable source and returns an observable sequence that contains elements from the input sequence that satisfy the condition.

`giving.operators.filter_indexed(predicate_indexed=None)`

Filters the elements of an observable sequence based on a predicate by incorporating the element's index.



Example

```
>>> op = filter_indexed(lambda value, index: (value + index) < 10)
```

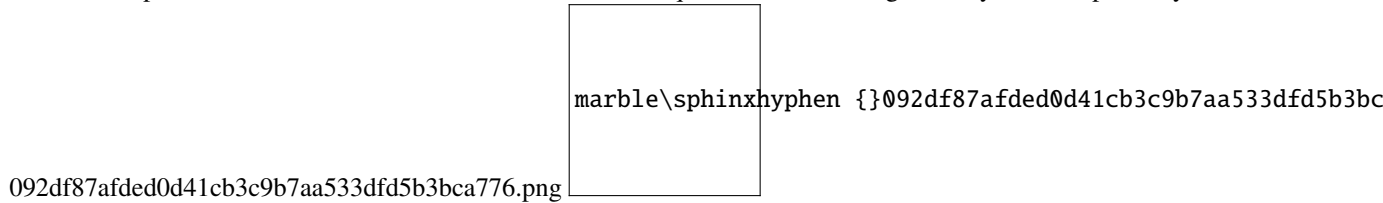
Parameters **predicate** – A function to test each source element for a condition; the second parameter of the function represents the index of the source element.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence that contains elements from the input sequence that satisfy the condition.

`giving.operators.finally_action(action)`

Invokes a specified action after the source observable sequence terminates gracefully or exceptionally.



Example

```
>>> res = finally_action(lambda: print('sequence ended'))
```

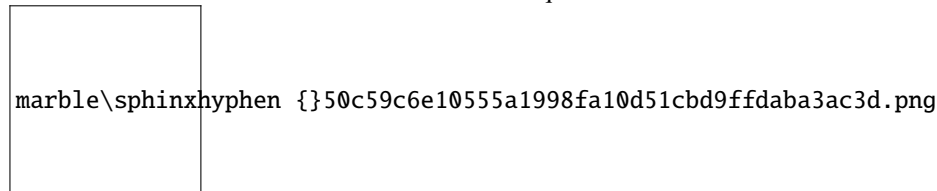
Parameters **action** (Callable[[], None]) – Action to invoke after the source observable sequence terminates.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence with the action-invoking termination behavior applied.

`giving.operators.find(predicate)`

Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire Observable sequence. 50c59c6e10555a1998fa10d51cbd9ffdaba3ac3d.png



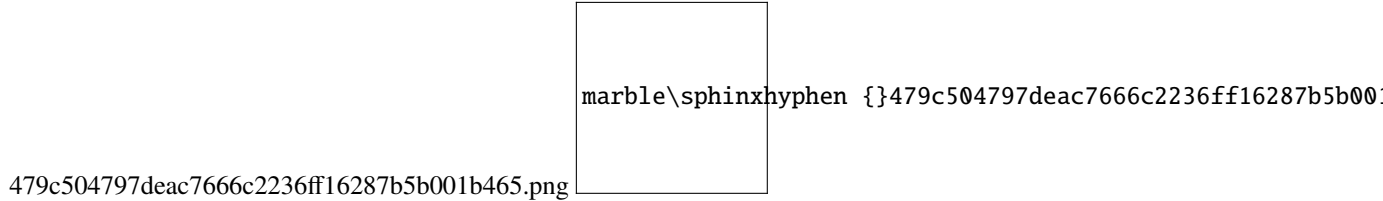
Parameters **predicate** (Callable[[~T, int, Observable[~T]], bool]) – The predicate that defines the conditions of the element to search for.

Return type Callable[[Observable[~T]], Observable[Optional[~T]]]

Returns An operator function that takes an observable source and returns an observable sequence with the first element that matches the conditions defined by the specified predicate, if found otherwise, None.

`giving.operators.find_index(predicate)`

Searches for an element that matches the conditions defined by the specified predicate, and returns an Observable sequence with the zero-based index of the first occurrence within the entire Observable sequence.



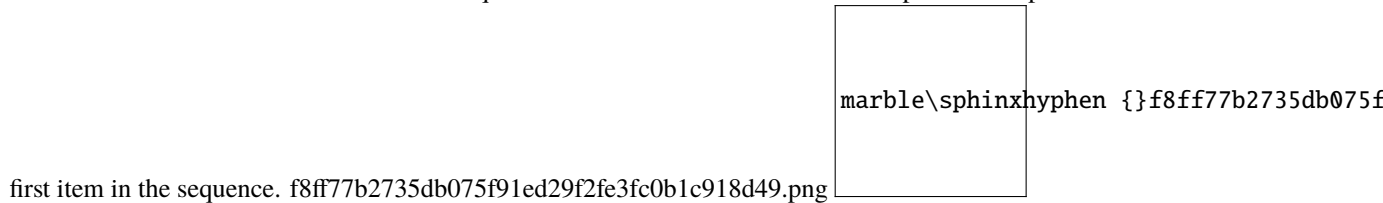
Parameters **predicate** (Callable[[~T, int, Observable[~T]], bool]) – The predicate that defines the conditions of the element to search for.

Return type Callable[[Observable[~T]], Observable[Optional[int]]]

Returns An operator function that takes an observable source and returns an observable sequence with the zero-based index of the first occurrence of an element that matches the conditions defined by match, if found; otherwise, -1.

`giving.operators.first(predicate=None)`

Returns the first element of an observable sequence that satisfies the condition in the predicate if present else the



Examples

```
>>> res = res = first()
>>> res = res = first(lambda x: x > 3)
```

Parameters **predicate** (Optional[Callable[[~T], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns A function that takes an observable source and returns an observable sequence containing the first element in the observable sequence that satisfies the condition in the predicate if provided, else the first item in the sequence.

`giving.operators.first_or_default(predicate=None, default_value=None)`

Returns the first element of an observable sequence that satisfies the condition in the predicate, or a default value

marble\sphinxhyphen {}e2da2e19412d614c

if no such element exists. e2da2e19412d614c899fa5b6ff0b46b0b28a65f2.png

Examples

```
>>> res = first_or_default()
>>> res = first_or_default(lambda x: x > 3)
>>> res = first_or_default(lambda x: x > 3, 0)
>>> res = first_or_default(None, 0)
```

Parameters

- **predicate** (Optional[Callable[[~T], bool]]) – [optional] A predicate function to evaluate for elements in the source sequence.
- **default_value** (Optional[~T]) – [Optional] The default value if no such element exists. If not specified, defaults to None.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns A function that takes an observable source and returns an observable sequence containing the first element in the observable sequence that satisfies the condition in the predicate, or a default value if no such element exists.

```
giving.operators.flat_map(mapper: Optional[Iterable[reactivex.operators._T2]] = None) →
    Callable[[reactivex.observable.observable.Observable[Any]],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]
giving.operators.flat_map(mapper:
    Optional[reactivex.observable.observable.Observable[reactivex.operators._T2]] =
    None) → Callable[[reactivex.observable.observable.Observable[Any]],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]
giving.operators.flat_map(mapper: Optional[Callable[[reactivex.operators._T1],
    Iterable[reactivex.operators._T2]]] = None) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]
giving.operators.flat_map(mapper: Optional[Callable[[reactivex.operators._T1],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]] = None)
    →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]
```

marble\sphinxhyphen {}0796d302706411e60

The flat_map operator. 0796d302706411e6023b5671684563964c0afa1c.png

One of the Following: Projects each element of an observable sequence to an observable sequence and merges the resulting observable sequences into one observable sequence.

Example

```
>>> flat_map(lambda x: Observable.range(0, x))
```

Or: Projects each element of the source observable sequence to the other observable sequence and merges the resulting observable sequences into one observable sequence.

Example

```
>>> flat_map(Observable.of(1, 2, 3))
```

Parameters **mapper** (Optional[Any]) – A transform function to apply to each element or an observable sequence to project each element from the source sequence onto.

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes a source observable and returns an observable sequence whose elements are the result of invoking the one-to-many transform function on each element of the input sequence.

```
giving.operators.flat_map_indexed(mapper_indexed: Optional[Iterable[reactivex.operators._T2]] = None)
    → Callable[[reactivex.observable.observable.Observable[Any]],
reactivex.observable.observable.Observable[reactivex.operators._T2]]
giving.operators.flat_map_indexed(mapper_indexed: Op-
    tional[reactivex.observable.observable.Observable[reactivex.operators._T2]]
    = None) →
    Callable[[reactivex.observable.observable.Observable[Any]],
reactivex.observable.observable.Observable[reactivex.operators._T2]]
giving.operators.flat_map_indexed(mapper_indexed: Optional[Callable[[reactivex.operators._T1, int],
    Iterable[reactivex.operators._T2]]] = None) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
reactivex.observable.observable.Observable[reactivex.operators._T2]]
giving.operators.flat_map_indexed(mapper_indexed: Optional[Callable[[reactivex.operators._T1, int],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]]
    = None) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
reactivex.observable.observable.Observable[reactivex.operators._T2]]
```

The *flat_map_indexed* operator.

One of the Following: Projects each element of an observable sequence to an observable sequence and merges the resulting observable sequences into one observable sequence.

marble\sphinxhyphen-{}0796d302706411e6023b5671684563964c0

0796d302706411e6023b5671684563964c0afa1c.png

Example

```
>>> source.flat_map_indexed(lambda x, i: Observable.range(0, x))
```

Or: Projects each element of the source observable sequence to the other observable sequence and merges the resulting observable sequences into one observable sequence.

Example

```
>>> source.flat_map_indexed(Observable.of(1, 2, 3))
```

Parameters `mapper_indexed` (Optional[Any]) – [Optional] A transform function to apply to each element or an observable sequence to project each element from the source sequence onto.

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable source and returns an observable sequence whose elements are the result of invoking the one-to-many transform function on each element of the input sequence.

giving.operators.flat_map_latest(*mapper*)

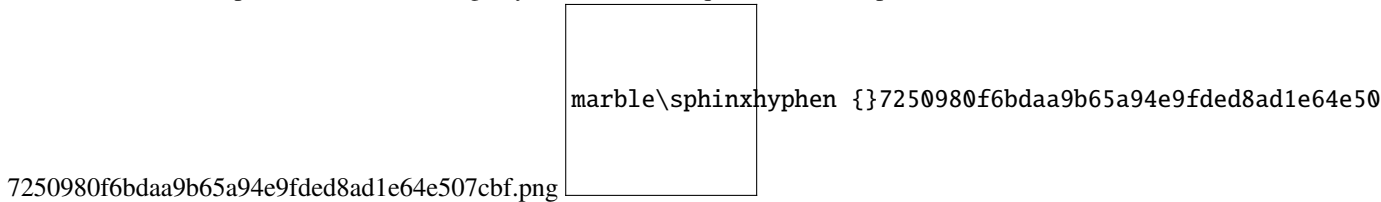
Projects each element of an observable sequence into a new sequence of observable sequences by incorporating the element's index and then transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence.

Parameters `mapper` – A transform function to apply to each source element. The second parameter of the function represents the index of the source element.

Returns An operator function that takes an observable source and returns an observable sequence whose elements are the result of invoking the transform function on each element of source producing an observable of Observable sequences and that at any point in time produces the elements of the most recent inner observable sequence that has been received.

giving.operators.fork_join(**others*)

Wait for observables to complete and then combine last values they emitted into a tuple. Whenever any of that observables completes without emitting any value, result sequence will complete at that moment as well.



Examples

```
>>> res = fork_join(obs1)
>>> res = fork_join(obs1, obs2, obs3)
```

Return type Callable[[Observable[Any]], Observable[Tuple[Any, ...]]]

Returns An operator function that takes an observable source and return an observable sequence containing the result of combining last element from each source in given sequence.

`giving.operators.format(string, raw=False, skip_missing=False)`

Format an object using a format string.

- If the data is a dict, it is passed as `*kwargs` to `str.format`, unless `raw=True`
- If the data is a tuple, it is passed as `*args` to `str.format`, unless `raw=True`

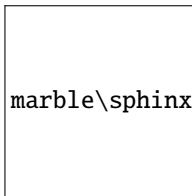
Parameters

- **string** – The format string.
- **raw** – Whether to pass the data as `*args` or `**kwargs` if it is a tuple or dict.
- **skip_missing** – Whether to ignore KeyErrors due to missing entries in the format.

`giving.operators.getitem(*keys, strict=False)`

Extract one or more keys from a dictionary.

If more than one key is given, a stream of tuples is produced. 84c8bf7c08c782ad854edc424348e0212df9d0d3.png



Parameters

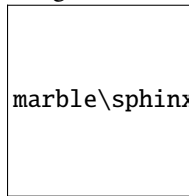
- **keys** – Names of the keys to index with.
- **strict** – If true, every element in the stream is required to contains this key.

`giving.operators.group_by(key_mapper, element_mapper=None, subject_mapper=None)`

Groups the elements of an observable sequence according to a specified key mapper function and comparer and selects the resulting elements by using a specified function.

marble\sphinxhyphen {}e0cc64db1260f70482180126b622340bc48

e0cc64db1260f70482180126b622340bc48163ba.png



Examples

```
>>> group_by(lambda x: x.id)
>>> group_by(lambda x: x.id, lambda x: x.name)
>>> group_by(lambda x: x.id, lambda x: x.name, lambda: ReplaySubject())
```

Keyword Arguments

- **key_mapper** – A function to extract the key for each element.
- **element_mapper** – [Optional] A function to map each source element to an element in an observable group.
- **subject_mapper** – A function that returns a subject used to initiate a grouped observable. Default mapper returns a Subject object.

Return type Callable[[Observable[~_T]], Observable[GroupedObservable[~_TKey, ~_TValue]]]

Returns An operator function that takes an observable source and returns a sequence of observable groups, each of which corresponds to a unique key value, containing all elements that share that same key value.

giving.operators.group_by_until(key_mapper, element_mapper, duration_mapper, subject_mapper=None)
Groups the elements of an observable sequence according to a specified key mapper function. A duration mapper function is used to control the lifetime of groups. When a group expires, it receives an OnCompleted notification. When a new element with the same key value as a reclaimed group occurs, the group will be reborn with a new

marble\sphinxhyphen {}c98393978f802bb0fe67988

lifetime request. c98393978f802bb0fe67988772a45adeda4060ab.png

Examples

```
>>> group_by_until(lambda x: x.id, None, lambda : reactivex.never())
>>> group_by_until(
    lambda x: x.id, lambda x: x.name, lambda grp: reactivex.never()
)
>>> group_by_until(
    lambda x: x.id,
    lambda x: x.name,
    lambda grp: reactivex.never(),
    lambda: ReplaySubject()
)
```

Parameters

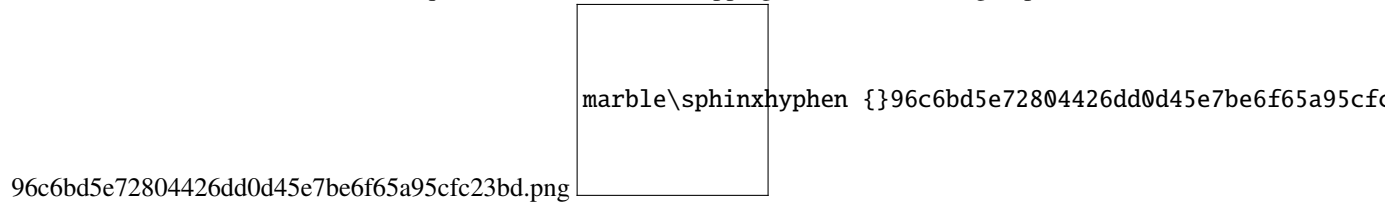
- **key_mapper** (Callable[[~_T], ~_TKey]) – A function to extract the key for each element.
- **element_mapper** (Optional[Callable[[~_T], ~_TValue]]) – A function to map each source element to an element in an observable group.
- **duration_mapper** (Callable[[GroupedObservable[~_TKey, ~_TValue]], Observable[Any]]) – A function to signal the expiration of a group.
- **subject_mapper** (Optional[Callable[[], Subject[~_TValue]]]) – A function that returns a subject used to initiate a grouped observable. Default mapper returns a Subject object.

Return type Callable[[Observable[~_T]], Observable[GroupedObservable[~_TKey, ~_TValue]]]

Returns An operator function that takes an observable source and returns a sequence of observable groups, each of which corresponds to a unique key value, containing all elements that share that same key value. If a group's lifetime expires, a new group with the same key value can be created once an element with such a key value is encountered.

`giving.operators.group_join(right, left_duration_mapper, right_duration_mapper)`

Correlates the elements of two sequences based on overlapping durations, and groups the results.



Parameters

- **right** (`Observable[_TRight]`) – The right observable sequence to join elements for.
- **left_duration_mapper** (`Callable[[_TLeft], Observable[Any]]`) – A function to select the duration (expressed as an observable sequence) of each element of the left observable sequence, used to determine overlap.
- **right_duration_mapper** (`Callable[[_TRight], Observable[Any]]`) – A function to select the duration (expressed as an observable sequence) of each element of the right observable sequence, used to determine overlap.

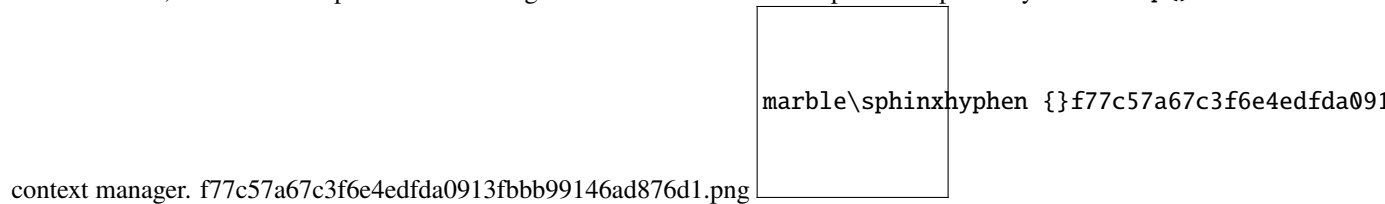
Return type `Callable[[Observable[_TLeft], Observable[_TRight]], Observable[Tuple[_TLeft, Observable[_TRight]]]]`

Returns An operator function that takes an observable source and returns an observable sequence that contains elements combined into a tuple from source elements that have an overlapping duration.

`giving.operators.group_wrap(name, **conditions)`

Return a stream of observables for wrapped groups.

In this schema, B and E correspond to the messages sent in the enter and exit phases respectively of the `wrap()`



Example

```
results = []

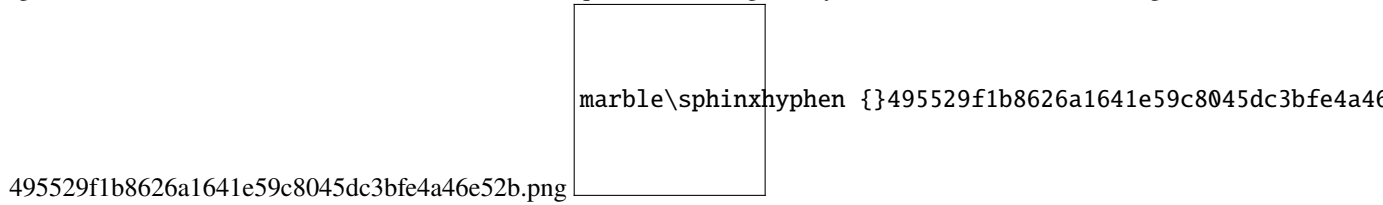
@obs.group_wrap().subscribe
def _(obs2):
    obs2["a"].sum() >> results
```

Parameters

- **name** – Name of the wrap block to group on.
- **conditions** – Maps a key to the value it must be associated to in the dictionary of the wrap statement, or to a predicate function on the value.

giving.operators.ignore_elements()

Ignores all elements in an observable sequence leaving only the termination messages.

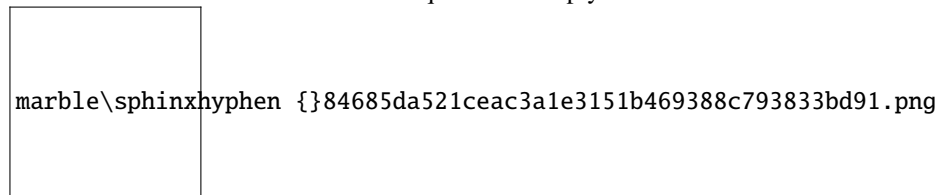


Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns An operator function that takes an observable source and returns an empty observable sequence that signals termination, successful or exceptional, of the source sequence.

giving.operators.is_empty()

Determines whether an observable sequence is empty. 84685da521ceac3a1e3151b469388c793833bd91.png

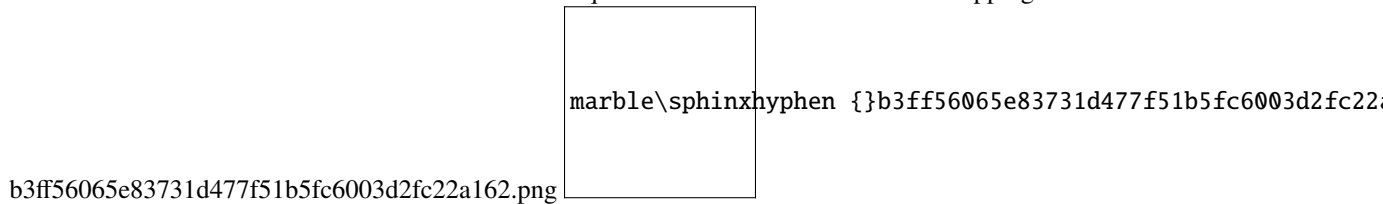


Return type `Callable[[Observable[Any]], Observable[bool]]`

Returns An operator function that takes an observable source and returns an observable sequence containing a single element determining whether the source sequence is empty.

giving.operators.join(right, left_duration_mapper, right_duration_mapper)

Correlates the elements of two sequences based on overlapping durations.



Parameters

- **right** (`Observable[~_T2]`) – The right observable sequence to join elements for.
- **left_duration_mapper** (`Callable[[Any], Observable[Any]]`) – A function to select the duration (expressed as an observable sequence) of each element of the left observable sequence, used to determine overlap.
- **right_duration_mapper** (`Callable[[Any], Observable[Any]]`) – A function to select the duration (expressed as an observable sequence) of each element of the right observable sequence, used to determine overlap.

Return type `Callable[[Observable[~_T1]], Observable[Tuple[~_T1, ~_T2]]]`

Returns An operator function that takes an observable source and returns an observable sequence that contains elements combined into a tuple from source elements that have an overlapping duration.

giving.operators.**keep**(*keys, **remap)

marble\sphinxhyphen {}c98cc

Keep certain dict keys and remap others. c98ccfe24e41b0a6275c872ca31cb7faf6ea1bbd.png

Parameters

- **keys** – Keys that must be kept
- **remap** – Keys that must be renamed

giving.operators.**kfilter**(fn)

Filter a stream of dictionaries.

Example

```
# [{"x": 1, "y": 2}, {"x": 100, "y": 50}] => [{"x": 100, "y": 50}]
gv.kfilter(lambda x, y: x > y)
```

Parameters **fn** – A function that will be called for each element, passing the element using ****kwargs**.

Note: If the dict has elements that are not in the function's arguments list and the function does not have a ****kwargs** argument, these elements will be dropped and no error will occur.

giving.operators.**kmap**(_fn=None, **_fns)

Map a dict, passing keyword arguments.

kmap either takes a positional function argument or keyword arguments serving to build a new dict.

Example

```
# [{"x": 1, "y": 2}] => [3]
gv.kmap(lambda x, y: x + y)

# [{"x": 1, "y": 2}] => [{"z": 3}]
gv.kmap(z=lambda x, y: x + y)
```

Parameters

- **_fn** – A function that will be called for each element, passing the element using ****kwargs**.

Note: If the dict has elements that are not in the function's arguments list and the function does not have a ****kwargs** argument, these elements will be dropped and no error will occur.

- **_fns** – Alternatively, build a new dict with each key associated to a function with the same interface as **fn**.

giving.operators.**kmerge**(scan=False)

Merge the dictionaries in the stream. b34c157ccad6cac31836d7d4accd5124cbf7c711.png

marble\sphinxhyphen {}b34c1

marble\sphinxhyphen {}b15b90a1f22863b59ff1f11838d8d9f7358f3

b15b90a1f22863b59ff1f11838d8d9f7358f38ef.png

giving.operators.**kscan**()

Alias for **kmerge**(scan=True).

giving.operators.**last**(predicate=None)

The last operator.

Returns the last element of an observable sequence that satisfies the condition in the predicate if specified, else

marble\sphinxhyphen {}7026868b150bc186397e1cda

the last element. 7026868b150bc186397e1cda4dff4ff1bf30b8ca.png

Examples

```
>>> op = last()
>>> op = last(lambda x: x > 3)
```

Parameters **predicate** (Optional[Callable[[~T], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns an observable sequence containing the last element in the observable sequence that satisfies the condition in the predicate.

giving.operators.**last_or_default**() →

Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
reactivex.observable.observable.Observable[Optional[reactivex.operators._T]]]

giving.operators.**last_or_default**(default_value: reactivex.operators._T) →

Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
reactivex.observable.observable.Observable[reactivex.operators._T]]

giving.operators.**last_or_default**(default_value: reactivex.operators._T, predicate:

Callable[[reactivex.operators._T], bool]) →
Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
reactivex.observable.observable.Observable[reactivex.operators._T]]

The last_or_default operator.

Returns the last element of an observable sequence that satisfies the condition in the predicate, or a default value

marble\sphinxhyphen {}ac702793ffd235d9ae

if no such element exists. ac702793ffd235d9aecffd4dcce018f409b71c.png

Examples

```
>>> res = last_or_default()
>>> res = last_or_default(lambda x: x > 3)
>>> res = last_or_default(lambda x: x > 3, 0)
>>> res = last_or_default(None, 0)
```

Parameters

- **predicate** (Optional[Callable[[~T], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.
- **default_value** (Optional[Any]) – [Optional] The default value if no such element exists. If not specified, defaults to None.

Return type Callable[[Observable[~T]], Observable[Any]]

Returns An operator function that takes an observable source and returns an observable sequence containing the last element in the observable sequence that satisfies the condition in the predicate, or a default value if no such element exists.

giving.operators.**map**(mapper=None)

The map operator.

Project each element of an observable sequence into a new form. 52e113b590f7fb5e7471c6d56c1829f94703f688.png

marble\sphinxhyphen {}52e113b590f7fb5e7471c6d56c1829f94703f688.png

Example

```
>>> map(lambda value: value * 10)
```

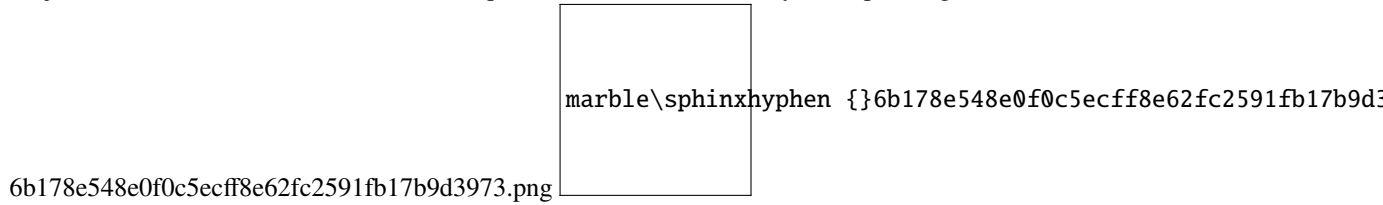
Parameters **mapper** (Optional[Callable[[~T1], ~T2]]) – A transform function to apply to each source element.

Return type Callable[[Observable[~T1]], Observable[~T2]]

Returns A partially applied operator function that takes an observable source and returns an observable sequence whose elements are the result of invoking the transform function on each element of the source.

`giving.operators.map_indexed(mapper_indexed=None)`

Project each element of an observable sequence into a new form by incorporating the element's index.



Example

```
>>> ret = map_indexed(lambda value, index: value * value + index)
```

Parameters `map_indexed` (Optional[Callable[[~T1, int], ~T2]]) – A transform function to apply to each source element. The second parameter of the function represents the index of the source element.

Return type Callable[[Observable[~T1]], Observable[~T2]]

Returns A partially applied operator function that takes an observable source and returns an observable sequence whose elements are the result of invoking the transform function on each element of the source.

`giving.operators.materialize()`

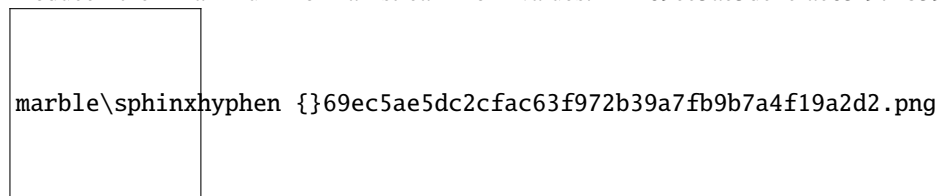
Materializes the implicit notifications of an observable sequence as explicit notification values.

Return type Callable[[Observable[~T]], Observable[Notification[~T]]]

Returns An operator function that takes an observable source and returns an observable sequence containing the materialized notification values from the source sequence.

`giving.operators.max(*, key=None, comparer=None, scan=False)`

Produce the maximum of a stream of values. 69ec5ae5dc2cfac63f972b39a7fb9b7a4f19a2d2.png



Parameters

- **key** – A key mapping function or a string.
- **comparer** – A function of two elements that returns -1 if the first is smaller than the second, 0 if they are equal, 1 if the second is larger.
- **scan** – If True, generate the current maximum on every element.
- **seed** – First element of the reduction.

`giving.operators.merge(*sources, max_concurrent=None)`

Merges an observable sequence of observable sequences into an observable sequence, limiting the number of concurrent subscriptions to inner sequences. Or merges two observable sequences into a single observable se-

quence. 4ff9f21e0687a8c45cd9cdc6350bbe584661a43f.png

marble\sphinxhyphen {}4ff9f21e0687a8c45cd9cdc6350bbe

Examples

```
>>> op = merge(max_concurrent=1)
>>> op = merge(other_source)
```

Parameters **max_concurrent** (Optional[int]) – [Optional] Maximum number of inner observable sequences being subscribed to concurrently or the second observable sequence.

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable source and returns the observable sequence that merges the elements of the inner sequences.

giving.operators.**merge_all**()

The merge_all operator.

Merges an observable sequence of observable sequences into an observable sequence.

bf397132fa74dd0d17e17f08ce1314bbef0bbbd1.png

marble\sphinxhyphen {}bf397132fa74dd0d17e17f08ce1314bbef0b

Return type Callable[[Observable[Observable[_T]]], Observable[_T]]

Returns A partially applied operator function that takes an observable source and returns the observable sequence that merges the elements of the inner sequences.

giving.operators.**min**(*, key=None, comparer=None, scan=False)

Produce the minimum of a stream of values. a3a11956650c120c73d13399527412e361144e31.png

marble\sphinxhyphen {}a3a11956650c120c73d13399527412e361144e31.png

Parameters

- **key** – A key mapping function or a string.
- **comparer** – A function of two elements that returns -1 if the first is smaller than the second, 0 if they are equal, 1 if the second is larger.
- **scan** – If True, generate the current minimum on every element.
- **seed** – First element of the reduction.

```

giving.operators.multicast() →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
             reactivex.observable.connectableobservable.ConnectableObservable[reactivex.operators._T]]
giving.operators.multicast(subject: reactivex.abc.subject.SubjectBase[reactivex.operators._T]) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
             reactivex.observable.connectableobservable.ConnectableObservable[reactivex.operators._T]]
giving.operators.multicast(*, subject_factory: Callable[[Optional[reactivex.abc.scheduler.SchedulerBase]],
             reactivex.abc.subject.SubjectBase[reactivex.operators._T]], mapper: Op-
             tional[Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
             reactivex.observable.observable.Observable[reactivex.operators._T2]]] =
             'None') →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
             reactivex.observable.observable.Observable[reactivex.operators._T2]]

```

Multicasts the source sequence notifications through an instantiated subject into all uses of the sequence within a mapper function. Each subscription to the resulting sequence causes a separate multicast invocation, exposing the sequence resulting from the mapper function's invocation. For specializations with fixed subject types, see Publish, PublishLast, and Replay.

Examples

```

>>> res = multicast(observable)
>>> res = multicast(
    subject_factory=lambda scheduler: Subject(), mapper=lambda x: x
)

```

Parameters

- **subject_factory** (Optional[Callable[[Optional[SchedulerBase]], SubjectBase[~_T]]]) – Factory function to create an intermediate subject through which the source sequence's elements will be multicast to the mapper function.
- **subject** (Optional[SubjectBase[~_T]]) – Subject to push source elements into.
- **mapper** (Optional[Callable[[Observable[~_T]], Observable[~_T2]]]) – [Optional] Mapper function which can use the multicasted source sequence subject to the policies enforced by the created subject. Specified only if subject_factory is a factory function.

Return type Callable[[Observable[~_T]], Union[Observable[~_T2], ConnectableObservable[~_T]]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a mapper function.

giving.operators.observe_on(scheduler)

Wraps the source sequence in order to run its observer callbacks on the specified scheduler.

Parameters **scheduler** (SchedulerBase) – Scheduler to notify observers on.

This only invokes observer callbacks on a scheduler. In case the subscription and/or unsubscription actions have side-effects that require to be run on a scheduler, use subscribe_on.

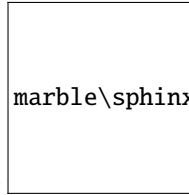
Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns the source sequence whose observations happen on the specified scheduler.

`giving.operators.on_error_resume_next(second)`

Continues an observable sequence that is terminated normally or by an exception with the next observable se-

quence. `cffe1529783c4bd3518f934f74ac7f8960bd15ed.png`



marble\sphinxhyphen {}cffe1529783c4bd3518f934f74ac7f

Keyword Arguments **second** – Second observable sequence used to produce results after the first sequence terminates.

Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns An observable sequence that concatenates the first and second sequence, even if the first sequence terminates exceptionally.

`giving.operators.pairwise()`

The pairwise operator.

Returns a new observable that triggers on the second and subsequent triggerings of the input observable. The Nth triggering of the input observable passes the arguments from the N-1th and Nth triggering as a pair. The argument passed to the N-1th triggering is held in hidden internal state until the Nth triggering occurs.

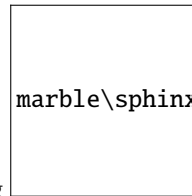
Return type `Callable[[Observable[~_T]], Observable[Tuple[~_T, ~_T]]]`

Returns An operator function that takes an observable source and returns an observable that triggers on successive pairs of observations from the input observable as an array.

`giving.operators.partition(predicate)`

Returns two observables which partition the observations of the source by the given function. The first will trigger observations for those values for which the predicate returns true. The second will trigger observations for those values where the predicate returns false. The predicate is executed once for each subscribed observer. Both also propagate all error observations arising from the source and each completes when the source completes.

`9d1412bca63017f8545dfdf42baf36e766ddce27.png`



marble\sphinxhyphen {}9d1412bca63017f8545dfdf42baf36e766dd

Parameters

- **predicate** (`Callable[[~_T], bool]`) – The function to determine which output Observable
- **observation.** (*will trigger a particular*) –

Return type `Callable[[Observable[~_T]], List[Observable[~_T]]]`

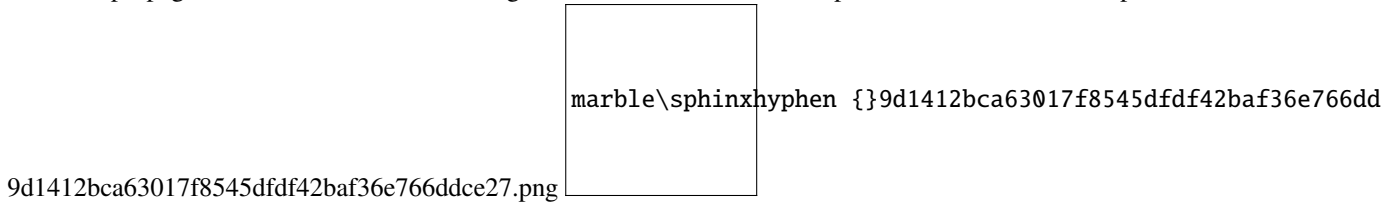
Returns An operator function that takes an observable source and returns a list of observables. The first triggers when the predicate returns True, and the second triggers when the predicate returns False.

`giving.operators.partition_indexed(predicate_indexed)`

The indexed partition operator.

Returns two observables which partition the observations of the source by the given function. The first will trigger observations for those values for which the predicate returns true. The second will trigger observations for those values where the predicate returns false. The predicate is executed once for each subscribed observer.

Both also propagate all error observations arising from the source and each completes when the source completes.



Parameters

- **predicate** – The function to determine which output Observable
- **observation.** (*will trigger a particular*) –

Return type Callable[[Observable[~_T]], List[Observable[~_T]]]

Returns A list of observables. The first triggers when the predicate returns True, and the second triggers when the predicate returns False.

giving.operators.**pluck**(key)

Retrieves the value of a specified key using dict-like access (as in element[key]) from all elements in the Observable sequence.

To pluck an attribute of each element, use pluck_attr.

Parameters **key** (~_TKey) – The key to pluck.

Return type Callable[[Observable[Dict[~_TKey, ~_TValue]], Observable[~_TValue]]

Returns An operator function that takes an observable source and returns a new observable sequence of key values.

giving.operators.**pluck_attr**(prop)

Retrieves the value of a specified property (using getattr) from all elements in the Observable sequence.

To pluck values using dict-like access (as in element[key]) on each element, use pluck.

Parameters **property** – The property to pluck.

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable source and returns a new observable sequence of property values.

giving.operators.**publish**() →

Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1],
reactivex.observable.connectableobservable.ConnectableObservable[reactivex.operators._T1]]

giving.operators.**publish**(mapper:

Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
reactivex.observable.observable.Observable[reactivex.operators._T2]]) →
Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1],
reactivex.observable.observable.Observable[reactivex.operators._T2]]

The *publish* operator.

Returns an observable sequence that is the result of invoking the mapper on a connectable observable sequence that shares a single subscription to the underlying sequence. This operator is a specialization of Multicast using a regular Subject.

Example

```
>>> res = publish()
>>> res = publish(lambda x: x)
```

Parameters **mapper** (Optional[Callable[[Observable[~_T1]], Observable[~_T2]]]) – [Optional] Selector function which can use the multicasted source sequence as many times as needed, without causing multiple subscriptions to the source sequence. Subscribers to the given source will receive all notifications of the source from the time of the subscription on.

Return type Callable[[Observable[~_T1]], Union[Observable[~_T2], ConnectableObservable[~_T1]]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a mapper function.

```
giving.operators.publish_value(initial_value: reactivex.operators._T1) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
    reactivex.observable.connectableobservable.ConnectableObservable[reactivex.operators._T1]]
giving.operators.publish_value(initial_value: reactivex.operators._T1, mapper:
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
    reactivex.observable.observable.Observable[reactivex.operators._T2]]
```

Returns an observable sequence that is the result of invoking the mapper on a connectable observable sequence that shares a single subscription to the underlying sequence and starts with initial_value.

This operator is a specialization of Multicast using a BehaviorSubject.

Examples

```
>>> res = source.publish_value(42)
>>> res = source.publish_value(42, lambda x: x.map(lambda y: y * y))
```

Parameters

- **initial_value** (~_T1) – Initial value received by observers upon subscription.
- **mapper** (Optional[Callable[[Observable[~_T1]], Observable[~_T2]]]) – [Optional] Optional mapper function which can use the multicasted source sequence as many times as needed, without causing multiple subscriptions to the source sequence. Subscribers to the given source will receive immediately receive the initial value, followed by all notifications of the source from the time of the subscription on.

Return type Callable[[Observable[~_T1]], Union[Observable[~_T2], ConnectableObservable[~_T1]]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a mapper function.

```

giving.operators.reduce(accumulator: Callable[[reactivex.operators._TState, reactivex.operators._T],
    reactivex.operators._TState]) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
    reactivex.observable.observable.Observable[reactivex.operators._T]]
giving.operators.reduce(accumulator: Callable[[reactivex.operators._TState, reactivex.operators._T],
    reactivex.operators._TState], seed: reactivex.operators._TState) →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
    reactivex.observable.observable.Observable[reactivex.operators._TState]]

```

The reduce operator.

Applies an accumulator function over an observable sequence, returning the result of the aggregation as a single element in the result sequence. The specified seed value is used as the initial accumulator value.

For aggregation behavior with incremental intermediate results, see *scan*.

3f09a5a057c2de3bb200b642e3295f510b010da1.png

Examples

```

>>> res = reduce(lambda acc, x: acc + x)
>>> res = reduce(lambda acc, x: acc + x, 0)

```

Parameters

- **accumulator** (Callable[[~_TState, ~_T], ~_TState]) – An accumulator function to be invoked on each element.
- **seed** (Union[~_TState, Type[NotSet]]) – Optional initial accumulator value.

Return type Callable[[Observable[~_T]], Observable[Any]]

Returns A partially applied operator function that takes an observable source and returns an observable sequence containing a single element with the final accumulator value.

giving.operators.ref_count()

Returns an observable sequence that stays connected to the source as long as there is at least one subscription to the observable sequence.

Return type Callable[[ConnectableObservable[~_T]], Observable[~_T]]

giving.operators.repeat(repeat_count=None)

Repeats the observable sequence a specified number of times. If the repeat count is not specified, the sequence

repeats indefinitely. 500cd215f2c580628cf9a7e611f44d42054a0e0a.png

Examples

```
>>> repeated = repeat()
>>> repeated = repeat(42)
```

Parameters

- **repeat_count** (Optional[int]) – Number of times to repeat the sequence. If not
- **provided** –
- **indefinitely.** (*repeats the sequence*) –

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable sources and returns an observable sequence producing the elements of the given sequence repeatedly.

```
giving.operators.replay(buffer_size: Optional[int] = None, window: Optional[Union[datetime.timedelta,
float]] = None, *, scheduler: Optional[reactivex.abc.scheduler.SchedulerBase] =
'None') →
Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
reactivex.observable.connectableobservable.ConnectableObservable[reactivex.operators._T1]]
giving.operators.replay(buffer_size: Optional[int] = None, window: Optional[Union[datetime.timedelta,
float]] = None, *, mapper: Op-
tional[Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
reactivex.observable.observable.Observable[reactivex.operators._T2]]], scheduler:
Optional[reactivex.abc.scheduler.SchedulerBase] = 'None') →
Callable[[reactivex.observable.observable.Observable[reactivex.operators._T1]],
reactivex.observable.observable.Observable[reactivex.operators._T2]]
```

The *replay* operator.

Returns an observable sequence that is the result of invoking the mapper on a connectable observable sequence that shares a single subscription to the underlying sequence replaying notifications subject to a maximum time length for the replay buffer.

This operator is a specialization of Multicast using a ReplaySubject.

Examples

```
>>> res = replay(buffer_size=3)
>>> res = replay(buffer_size=3, window=0.5)
>>> res = replay(None, 3, 0.5)
>>> res = replay(lambda x: x.take(6).repeat(), 3, 0.5)
```

Parameters

- **mapper** (Optional[Callable[[Observable[~_T1]], Observable[~_T2]]]) – [Optional] Selector function which can use the multicasted source sequence as many times as needed, without causing multiple subscriptions to the source sequence. Subscribers to the given source will receive all the notifications of the source subject to the specified replay buffer trimming policy.
- **buffer_size** (Optional[int]) – [Optional] Maximum element count of the replay buffer.
- **window** (Union[timedelta, float, None]) – [Optional] Maximum time length of the replay buffer.

- **scheduler** (Optional[SchedulerBase]) – [Optional] Scheduler the observers are invoked on.

Return type Callable[[Observable[_T1]], Union[Observable[_T2], ConnectableObservable[_T1]]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements of a sequence produced by multicasting the source sequence within a mapper function.

giving.operators.**retry**(*retry_count=None*)

Repeats the source observable sequence the specified number of times or until it successfully terminates. If the retry count is not specified, it retries indefinitely.

Examples

```
>>> retried = retry()
>>> retried = retry(42)
```

Parameters **retry_count** (Optional[int]) – [Optional] Number of times to retry the sequence. If not provided, retry the sequence indefinitely.

Return type Callable[[Observable[_T]], Observable[_T]]

Returns An observable sequence producing the elements of the given sequence repeatedly until it terminates successfully.

giving.operators.**roll**(*n, reduce=None, seed=<class 'reactivex.internal.utils.NotSet'>*)

Group the last *n* elements, giving a sequence of overlapping sequences.

For example, this can be used to compute a rolling average of the 100 last elements (however, `average(scan=100)` is better optimized).

```
op.roll(100, lambda xs: sum(xs) / len(xs))
```

marble\sphinxhyphen-{}d1b21b1be01b397c21c66f8aea675c3dcf0d8575.png

d1b21b1be01b397c21c66f8aea675c3dcf0d8575.png

Parameters

- **n** – The number of elements to group together.
- **reduce** – A function to reduce the group.
It should take five arguments:
 - **last**: The last result.
 - **add**: The element that was just added. It is the last element in the elements list.
 - **drop**: The element that was dropped to make room for the added one. It is *not* in the elements argument. If the list of elements is not yet of size *n*, there is no need to drop anything and drop is None.
 - **last_size**: The window size on the last invocation.

– **current_size**: The window size on this invocation.

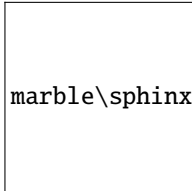
Defaults to returning the deque of elements directly.

Note: The same reference is returned each time in order to save memory, so it should be processed immediately.

- **seed** – The first element of the reduction.

giving.operators.**sample**(*sampler, scheduler=None*)

Samples the observable sequence at each interval. 753724403f4ec050f1e41ae18c0aa6c739db97a9.png



Examples

```
>>> res = sample(sample_observable) # Sampler tick sequence
>>> res = sample(5.0) # 5 seconds
```

Parameters

- **sampler** (Union[timedelta, float, Observable[Any]]) – Observable used to sample the source observable **or** time interval at which to sample (specified as a float denoting seconds or an instance of timedelta).
- **scheduler** (Optional[SchedulerBase]) – Scheduler to use only when a time interval is given.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns a sampled observable sequence.

```
giving.operators.scan(accumulator: Callable[[reactivex.operators._T, reactivex.operators._T],
reactivex.operators._T]) →
Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
reactivex.observable.observable.Observable[reactivex.operators._T]]
giving.operators.scan(accumulator: Callable[[reactivex.operators._TState, reactivex.operators._T],
reactivex.operators._TState], seed: Union[reactivex.operators._TState,
Type[reactivex.internal.utils.NotSet]]) →
Callable[[reactivex.observable.observable.Observable[reactivex.operators._T]],
reactivex.observable.observable.Observable[reactivex.operators._TState]]
```

The scan operator.

Applies an accumulator function over an observable sequence and returns each intermediate result. The optional seed value is used as the initial accumulator value. For aggregation behavior with no intermediate results, see

aggregate() or *Observable()*. bc4ff8e48a531690a2cf5b2a166dfd22fd92290c.png

marble\sphinxhyphen-{}bc4ff8e48a5316

Examples

```
>>> scanned = source.scan(lambda acc, x: acc + x)
>>> scanned = source.scan(lambda acc, x: acc + x, 0)
```

Parameters

- **accumulator** (Callable[[~TState, ~T], ~TState]) – An accumulator function to be invoked on each element.
- **seed** (Union[~TState, Type[NotSet]]) – [Optional] The initial accumulator value.

Return type Callable[[Observable[~T]], Observable[~TState]]

Returns A partially applied operator function that takes an observable source and returns an observable sequence containing the accumulated values.

giving.operators.**sequence_equal**(*second*, *comparer=None*)

Determines whether two sequences are equal by comparing the elements pairwise using a specified equality

comparer. 9aae3d37d38bfe67409f48b0b2e47fa1b19cc626.png

marble\sphinxhyphen-{}9aae3d37d38bfe67409f48b0b2e4

Examples

```
>>> res = sequence_equal([1,2,3])
>>> res = sequence_equal([{"value": 42}], lambda x, y: x.value == y.value)
>>> res = sequence_equal(reactivex.return_value(42))
>>> res = sequence_equal(
    reactivex.return_value({"value": 42}), lambda x, y: x.value == y.value)
```

Parameters

- **second** (Union[Observable[~T], Iterable[~T]]) – Second observable sequence or iterable to compare.
- **comparer** (Optional[Callable[[~T, ~T], bool]]) – [Optional] Comparer used to compare elements of both sequences. No guarantees on order of comparer arguments.

Return type Callable[[Observable[~T]], Observable[bool]]

Returns An operator function that takes an observable source and returns an observable sequence that contains a single element which indicates whether both sequences are of equal length and their corresponding elements are equal according to the specified equality comparer.

`giving.operators.share()`

Share a single subscription among multiple observers.

This is an alias for a composed `publish()` and `ref_count()`.

Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns An operator function that takes an observable source and returns a new Observable that multicasts (shares) the original Observable. As long as there is at least one Subscriber this Observable will be subscribed and emitting data. When all subscribers have unsubscribed it will unsubscribe from the source Observable.

`giving.operators.single(predicate=None)`

The single operator.

Returns the only element of an observable sequence that satisfies the condition in the optional predicate, and reports an exception if there is not exactly one element in the observable sequence.

marble\sphinxhyphen {}59e02f94169eadf875bb292a59cda555b87d5092.png

59e02f94169eadf875bb292a59cda555b87d5092.png

Example

```
>>> res = single()
>>> res = single(lambda x: x == 42)
```

Parameters `predicate` (Optional[Callable[[~_T], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.

Return type `Callable[[Observable[~_T]], Observable[~_T]]`

Returns An operator function that takes an observable source and returns an observable sequence containing the single element in the observable sequence that satisfies the condition in the predicate.

`giving.operators.single_or_default(predicate=None, default_value=None)`

Returns the only element of an observable sequence that matches the predicate, or a default value if no such element exists this method reports an exception if there is more than one element in the observable sequence.

marble\sphinxhyphen {}745696a3c38caf54674ca4831a80f237d3a54efe.png

745696a3c38caf54674ca4831a80f237d3a54efe.png

Examples

```
>>> res = single_or_default()
>>> res = single_or_default(lambda x: x == 42)
>>> res = single_or_default(lambda x: x == 42, 0)
>>> res = single_or_default(None, 0)
```

Parameters

- **predicate** (Optional[Callable[[~T], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.
- **default_value** (Optional[Any]) – [Optional] The default value if the index is outside the bounds of the source sequence.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns an observable sequence containing the single element in the observable sequence that satisfies the condition in the predicate, or a default value if no such element exists.

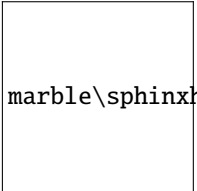
giving.operators.**single_or_default_async**(*has_default=False, default_value=None*)

Return type Callable[[Observable[~T]], Observable[~T]]

giving.operators.**skip**(*count*)

The skip operator.

Bypasses a specified number of elements in an observable sequence and then returns the remaining elements.

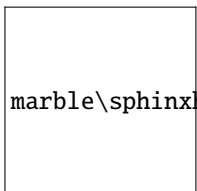
9829e00890b070583966679ff0055d2470c7f095.png  marble\sphinxhyphen-{}9829e00890b070583966679ff0055d2470c7f095.png

Parameters **count** (int) – The number of elements to skip before returning the remaining elements.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements that occur after the specified index in the input sequence.

giving.operators.**skip_last**(*count*)

The skip_last operator. a5f9d54f3ed827712265516535934307683b5e1c.png  marble\sphinxhyphen-{}a5f9d54f3ed827712265516535934307683b5e1c.png

Bypasses a specified number of elements at the end of an observable sequence.

This operator accumulates a queue with a length enough to store the first *count* elements. As more elements are received, elements are taken from the front of the queue and produced on the result sequence. This causes elements to be delayed.

Parameters

- **count** (int) – Number of elements to bypass at the end of the source
- **sequence.** –

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence containing the source sequence elements except for the bypassed ones at the end.

giving.operators.**skip_last_with_time**(duration, scheduler=None)

Skips elements for the specified duration from the end of the observable source sequence.

Example

```
>>> res = skip_last_with_time(5.0)
```

This operator accumulates a queue with a length enough to store elements received during the initial duration window. As more elements are received, elements older than the specified duration are taken from the queue and produced on the result sequence. This causes elements to be delayed with duration.

Parameters

- **duration** (Union[timedelta, float]) – Duration for skipping elements from the end of the sequence.
- **scheduler** (Optional[SchedulerBase]) – Scheduler to use for time handling.

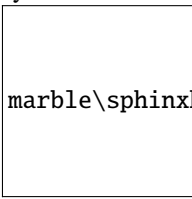
Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An observable sequence with the elements skipped during the specified duration from the end of the source sequence.

giving.operators.**skip_until**(other)

Returns the values from the source observable sequence only after the other observable sequence produces a

value. f275ca05c567b536db954d93cc4a27a8309a01d8.png



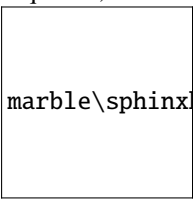
Parameters other – The observable sequence that triggers propagation of elements of the source sequence.

Returns An operator function that takes an observable source and returns an observable sequence containing the elements of the source sequence starting from the point the other sequence triggered propagation.

giving.operators.**skip_until_with_time**(start_time, scheduler=None)

Skips elements from the observable source sequence until the specified start time. Errors produced by the source sequence are always forwarded to the result sequence, even if the error occurs before the start time.

531777319ad55f0b44484e3380596082c25027fc.png



Examples

```
>>> res = skip_until_with_time(datetime())
>>> res = skip_until_with_time(5.0)
```

Parameters `start_time` (Union[datetime, timedelta, float]) – Time to start taking elements from the source sequence. If this value is less than or equal to `datetime.utcnow()`, no elements will be skipped.

Return type Callable[[Observable[_T]], Observable[_T]]

Returns An operator function that takes an observable source and returns an observable sequence with the elements skipped until the specified start time.

giving.operators.**skip_while**(*predicate*)

The *skip_while* operator.

Bypasses elements in an observable sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.

bf92f178aab29b95109adeb0c9c7bbdfc7965209.png

marble\sphinxhyphen-{}bf92f178aab29b95109adeb0c9c7bbdfc796

Example

```
>>> skip_while(lambda value: value < 10)
```

Parameters `predicate` (Callable[[_T], bool]) – A function to test each element for a condition; the second parameter of the function represents the index of the source element.

Return type Callable[[Observable[_T]], Observable[_T]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements from the input sequence starting at the first element in the linear series that does not pass the test specified by predicate.

giving.operators.**skip_while_indexed**(*predicate*)

Bypasses elements in an observable sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function.

bf92f178aab29b95109adeb0c9c7bbdfc7965209.png

marble\sphinxhyphen-{}bf92f178aab29b95109adeb0c9c7bbdfc796

Example

```
>>> skip_while(lambda value, index: value < 10 or index < 10)
```

Parameters `predicate` (Callable[[~T, int], bool]) – A function to test each element for a condition; the second parameter of the function represents the index of the source element.

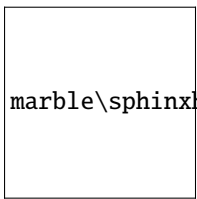
Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements from the input sequence starting at the first element in the linear series that does not pass the test specified by predicate.

giving.operators.**skip_with_time**(duration, scheduler=None)

Skips elements for the specified duration from the start of the observable source sequence.

281c641f8ea3abef9b70fb19d7295b200ebff650.png



Parameters `skip_with_time` (>>> res =) –

Specifying a zero value for duration doesn't guarantee no elements will be dropped from the start of the source sequence. This is a side-effect of the asynchrony introduced by the scheduler, where the action that causes callbacks from the source sequence to be forwarded may not execute immediately, despite the zero due time.

Errors produced by the source sequence are always forwarded to the result sequence, even if the error occurs before the duration.

Parameters

- **duration** (Union[timedelta, float]) – Duration for skipping elements from the start of the
- **sequence.** –

Return type Callable[[Observable[~T]], Observable[~T]]


Returns An operator function that takes an observable source and returns an observable sequence with the elements skipped during the specified duration from the start of the source sequence.

giving.operators.**slice**(start=None, stop=None, step=None)

The slice operator.

Slices the given observable. It is basically a wrapper around the operators `skip`, `skip_last`, `take`, `take_last`

and filter. 80c2d0fdc2c767f0a81ea312a7dec1b9cf921f0d.png



Examples

```
>>> result = source.slice(1, 10)
>>> result = source.slice(1, -2)
>>> result = source.slice(1, -1, 2)
```

Parameters

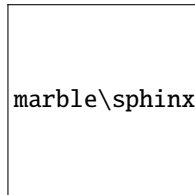
- **start** (Optional[int]) – First element to take of skip last
- **stop** (Optional[int]) – Last element to take of skip last
- **step** (Optional[int]) – Takes every step element. Must be larger than zero

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns a sliced observable sequence.

giving.operators.sole(*, keep_key=False, exclude=[])

Extract values from a stream of dicts with one entry each. 2051e2a94e433846e59e2b559e0e65bacac7ad31.png



marble\sphinxhyphen {}2051e2a94e433846e59e2b559e0e65bacac7ad31.png

If, after removing keys from the exclusion set, any dict is empty or has a length superior to 1, that is an error.

Parameters

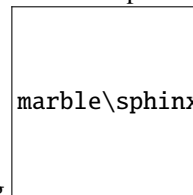
- **keep_key** – If True, return a (key, value) tuple, otherwise only return the value. Defaults to False.
- **exclude** – Keys to exclude.

giving.operators.some(predicate=None)

The some operator.

Determines whether some element of an observable sequence satisfies a condition if present, else if some items

are in the sequence. 03b8068203dfa863141e2d946ee5dfa724782216.png



marble\sphinxhyphen {}03b8068203dfa863141e

Examples

```
>>> result = source.some()
>>> result = source.some(lambda x: x > 3)
```

Parameters **predicate** (Optional[Callable[[~_T], bool]]) – A function to test each element for a condition.

Return type Callable[[Observable[~_T], Observable[bool]]

Returns An operator function that takes an observable source and returns an observable sequence containing a single element determining whether some elements in the source sequence pass the test in the specified predicate if given, else if some items are in the sequence.

```
giving.operators.starmap(mapper: Callable[[reactivex.operators._A, reactivex.operators._B],
reactivex.operators._T]) →
Callable[[reactivex.observable.observable.Observable[Tuple[reactivex.operators._A,
reactivex.operators._B]]],
reactivex.observable.observable.Observable[reactivex.operators._T]]
giving.operators.starmap(mapper: Callable[[reactivex.operators._A, reactivex.operators._B,
reactivex.operators._C], reactivex.operators._T]) →
Callable[[reactivex.observable.observable.Observable[Tuple[reactivex.operators._A,
reactivex.operators._B, reactivex.operators._C]]],
reactivex.observable.observable.Observable[reactivex.operators._T]]
giving.operators.starmap(mapper: Callable[[reactivex.operators._A, reactivex.operators._B,
reactivex.operators._C, reactivex.operators._D], reactivex.operators._T]) →
Callable[[reactivex.observable.observable.Observable[Tuple[reactivex.operators._A,
reactivex.operators._B, reactivex.operators._C, reactivex.operators._D]]],
reactivex.observable.observable.Observable[reactivex.operators._T]]
```

The starmap operator.

Unpack arguments grouped as tuple elements of an observable sequence and return an observable sequence of values by invoking the mapper function with star applied unpacked elements as positional arguments.

Use instead of `map()` when the the arguments to the mapper is grouped as tuples and the mapper function takes

marble\sphinxhyphen-{}1d791e0ab4245961cd74

multiple arguments. 1d791e0ab4245961cd740c268a2a92fb24396ecf.png

Example

```
>>> starmap(lambda x, y: x + y)
```

Parameters **mapper** (Optional[Callable[..., Any]]) – A transform function to invoke with unpacked elements as arguments.

Return type Callable[[Observable[Any], Observable[Any]]

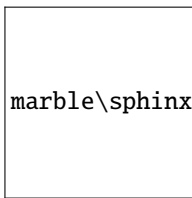
Returns An operator function that takes an observable source and returns an observable sequence containing the results of invoking the mapper function with unpacked elements of the source.

```

giving.operators.starmap_indexed(mapper: Callable[[reactivex.operators._A, int], reactivex.operators._T])
    →
    Callable[[reactivex.observable.observable.Observable[reactivex.operators._A]],
    reactivex.observable.observable.Observable[reactivex.operators._T]]
giving.operators.starmap_indexed(mapper: Callable[[reactivex.operators._A, reactivex.operators._B, int],
    reactivex.operators._T]) →
    Callable[[reactivex.observable.observable.Observable[Tuple[reactivex.operators._A,
    reactivex.operators._B]]],
    reactivex.observable.observable.Observable[reactivex.operators._T]]
giving.operators.starmap_indexed(mapper: Callable[[reactivex.operators._A, reactivex.operators._B,
    reactivex.operators._C, int], reactivex.operators._T]) →
    Callable[[reactivex.observable.observable.Observable[Tuple[reactivex.operators._A,
    reactivex.operators._B, reactivex.operators._C]]],
    reactivex.observable.observable.Observable[reactivex.operators._T]]
giving.operators.starmap_indexed(mapper: Callable[[reactivex.operators._A, reactivex.operators._B,
    reactivex.operators._C, reactivex.operators._D, int],
    reactivex.operators._T]) →
    Callable[[reactivex.observable.observable.Observable[Tuple[reactivex.operators._A,
    reactivex.operators._B, reactivex.operators._C, reactivex.operators._D]]],
    reactivex.observable.observable.Observable[reactivex.operators._T]]

```

Variant of `starmap()` which accepts an indexed mapper. 8239818b5b16fe0dd1257d653c34b42e0d963dbb.png



marble\sphinxhyphen {}8239818b5b16fe0dd1257d653c34b42e0d963dbb.png

Example

```
>>> starmap_indexed(lambda x, y, i: x + y + i)
```

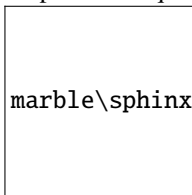
Parameters `mapper` (Optional[Callable[..., Any]]) – A transform function to invoke with unpacked elements as arguments.

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable source and returns an observable sequence containing the results of invoking the indexed mapper function with unpacked elements of the source.

```
giving.operators.start_with(*args)
```

Prepends a sequence of values to an observable sequence. 2a04dfbc6e7eaf822fbe38ae7023896bc704e6de.png



marble\sphinxhyphen {}2a04dfbc6e7eaf822fbe38ae7023896bc704e6de.png

Example

```
>>> start_with(1, 2, 3)
```

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes a source observable and returns the source sequence prepended with the specified values.

`giving.operators.subscribe_on(scheduler)`

Subscribe on the specified scheduler.

Wrap the source sequence in order to run its subscription and unsubscription logic on the specified scheduler. This operation is not commonly used; see the remarks section for more information on the distinction between `subscribe_on` and `observe_on`.

This only performs the side-effects of subscription and unsubscription on the specified scheduler. In order to invoke observer callbacks on a scheduler, use `observe_on`.

Parameters `scheduler` (SchedulerBase) – Scheduler to perform subscription and unsubscription actions on.

Return type Callable[[Observable[~_T]], Observable[~_T]]

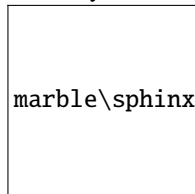
Returns An operator function that takes an observable source and returns the source sequence whose subscriptions and un-subscriptions happen on the specified scheduler.

`giving.operators.sum(*, scan=False)`

`giving.operators.switch_latest()`

The `switch_latest` operator.

Transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence. [b9329fb493471af771a2b890318e9a08d9bc0cfd.png](#)



marble\sphinxxyphen {}b9329fb493471af771a2b890318e9a08d9bc0cfd.png

Returns A partially applied operator function that takes an observable source and returns the observable sequence that at any point in time produces the elements of the most recent inner observable sequence that has been received.

`giving.operators.tag(group="", field='$word', group_field='$group')`

Tag each dict or object with a unique word.

If the item is a dict, do `item[field] = <new_word>`, otherwise attempt to do `setattr(item, field, <new_word>)`.

These tags are displayed specially by the `display()` method and they can be used to determine breakpoints with the `breakword()` method.

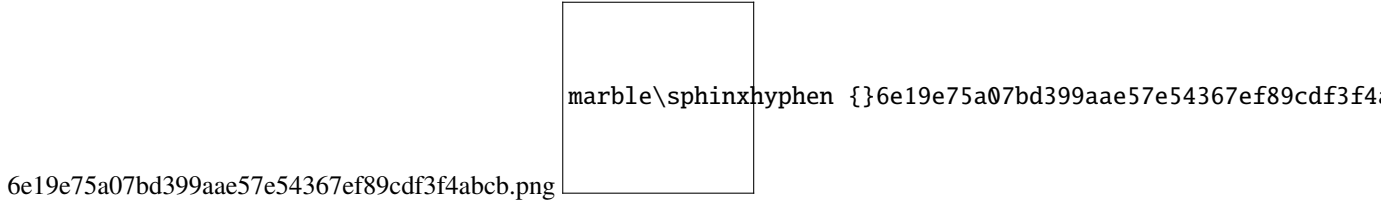
Parameters

- **group** – An arbitrary group name that corresponds to an independent sequence of words. It determines the color in display.
- **field** – The field name in which to put the word (default: `$word`).

- **group_field** – The field name in which to put the group (default: \$group).

giving.operators.**take**(count)

Returns a specified number of contiguous elements from the start of an observable sequence.



Example

```
>>> op = take(5)
```

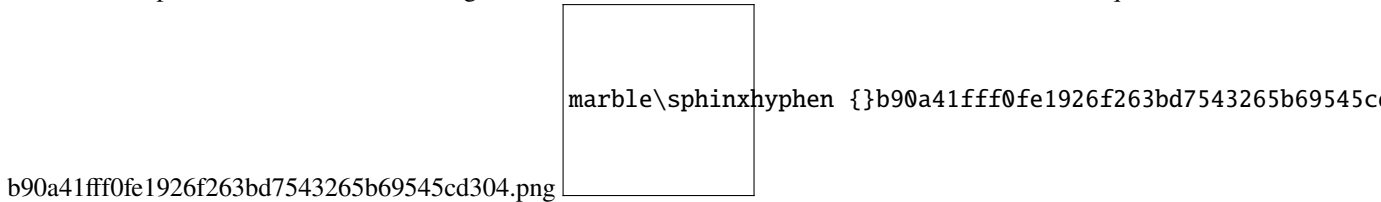
Parameters **count** (int) – The number of elements to return.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the specified number of elements from the start of the input sequence.

giving.operators.**take_last**(count)

Returns a specified number of contiguous elements from the end of an observable sequence.



Example

```
>>> res = take_last(5)
```

This operator accumulates a buffer with a length enough to store elements count elements. Upon completion of the source sequence, this buffer is drained on the result sequence. This causes the elements to be delayed.

Parameters

- **count** (int) – Number of elements to take from the end of the source
- **sequence.** –

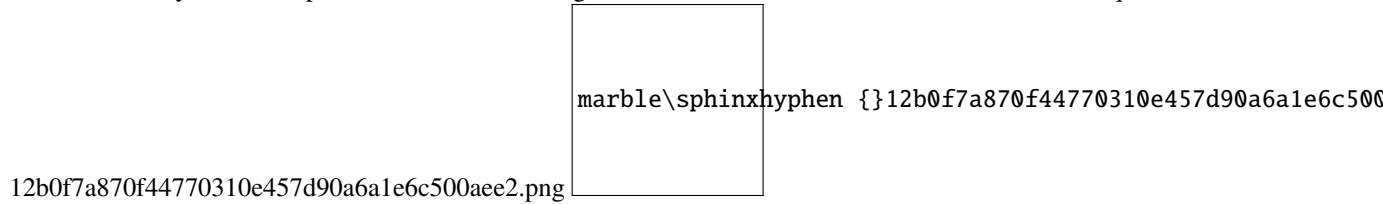
Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence containing the specified number of elements from the end of the source sequence.

giving.operators.**take_last_buffer**(count)

The *take_last_buffer* operator.

Returns an array with the specified number of contiguous elements from the end of an observable sequence.



Example

```
>>> res = source.take_last(5)
```

This operator accumulates a buffer with a length enough to store elements count elements. Upon completion of the source sequence, this buffer is drained on the result sequence. This causes the elements to be delayed.

Parameters

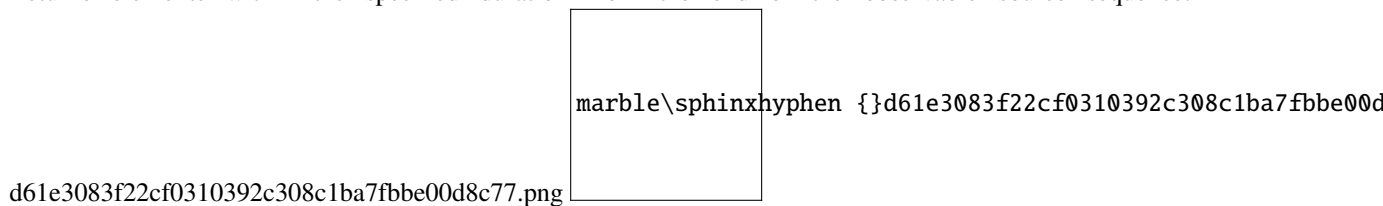
- **count** (int) – Number of elements to take from the end of the source
- **sequence.** –

Return type Callable[[Observable[~_T]], Observable[List[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence containing a single list with the specified number of elements from the end of the source sequence.

giving.operators.**take_last_with_time**(duration, scheduler=None)

Returns elements within the specified duration from the end of the observable source sequence.



Example

```
>>> res = take_last_with_time(5.0)
```

This operator accumulates a queue with a length enough to store elements received during the initial duration window. As more elements are received, elements older than the specified duration are taken from the queue and produced on the result sequence. This causes elements to be delayed with duration.

Parameters

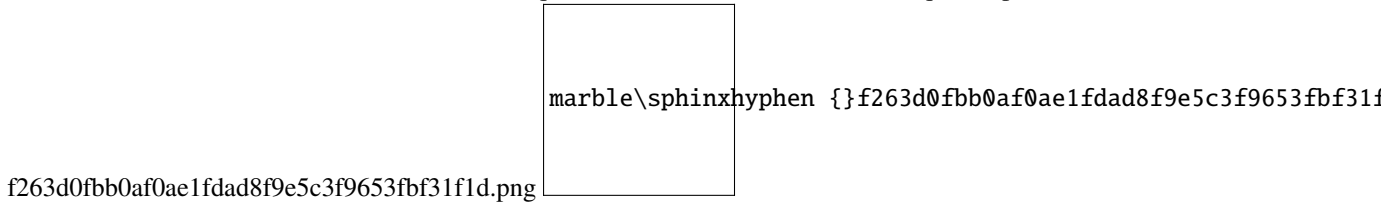
- **duration** (Union[timedelta, float]) – Duration for taking elements from the end of the
- **sequence.** –

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence with the elements taken during the specified duration from the end of the source sequence.

giving.operators.**take_until**(*other*)

Returns the values from the source observable sequence until the other observable sequence produces a value.



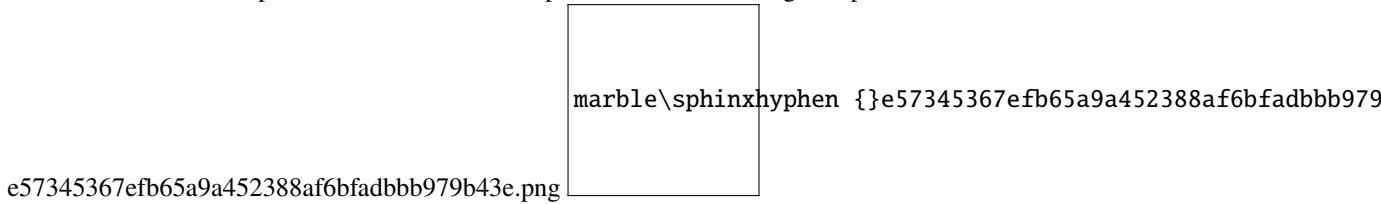
Parameters **other** (Observable[Any]) – Observable sequence that terminates propagation of elements of the source sequence.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns as observable sequence containing the elements of the source sequence up to the point the other sequence interrupted further propagation.

giving.operators.**take_until_with_time**(*end_time*, *scheduler=None*)

Takes elements for the specified duration until the specified end time, using the specified scheduler to run timers.



Examples

```
>>> res = take_until_with_time(dt, [optional scheduler])
>>> res = take_until_with_time(5.0, [optional scheduler])
```

Parameters

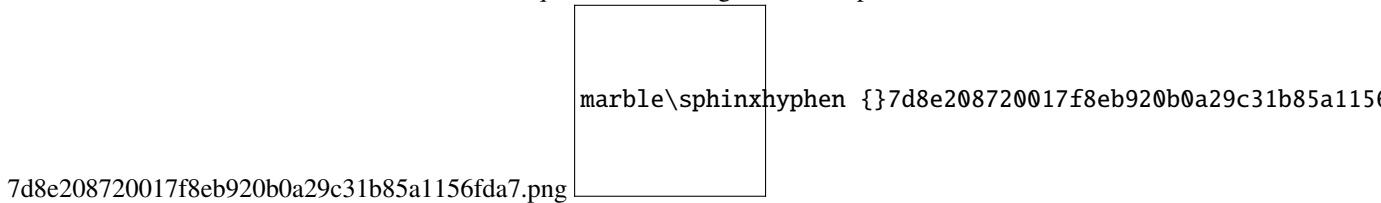
- **end_time** (Union[datetime, timedelta, float]) – Time to stop taking elements from the source sequence. If this value is less than or equal to `datetime.utcnow()`, the result stream will complete immediately.
- **scheduler** (Optional[SchedulerBase]) – Scheduler to run the timer on.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns an observable sequence with the elements taken until the specified end time.

giving.operators.**take_while**(*predicate*, *inclusive=False*)

Returns elements from an observable sequence as long as a specified condition is true.



Example

```
>>> take_while(lambda value: value < 10)
```

Parameters

- **predicate** (Callable[[~_T], bool]) – A function to test each element for a condition.
- **inclusive** (bool) – [Optional] When set to True the value that caused the predicate function to return False will also be emitted. If not specified, defaults to False.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence that contains the elements from the input sequence that occur before the element at which the test no longer passes.

giving `operators.take_while_indexed(predicate, inclusive=False)`

Returns elements from an observable sequence as long as a specified condition is true. The element's index is used

in the logic of the predicate function. e150f5b604c299df12cc07f30c3c4d4816a44204.png

marble\sphinxhyphen-{}e150f5b

Example

```
>>> take_while_indexed(lambda value, index: value < 10 or index < 10)
```

Parameters

- **predicate** (Callable[[~_T, int], bool]) – A function to test each element for a condition; the second parameter of the function represents the index of the source element.
- **inclusive** (bool) – [Optional] When set to True the value that caused the predicate function to return False will also be emitted. If not specified, defaults to False.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An observable sequence that contains the elements from the input sequence that occur before the element at which the test no longer passes.

giving `operators.take_with_time(duration, scheduler=None)`

Takes elements for the specified duration from the start of the observable source sequence.

c023c934bb4b63709fe6a9255bac3022b7ea174c.png

marble\sphinxhyphen-{}c023c934bb4b63709fe6a9255bac3022b7ea

Example

```
>>> res = take_with_time(5.0)
```

This operator accumulates a queue with a length enough to store elements received during the initial duration window. As more elements are received, elements older than the specified duration are taken from the queue and produced on the result sequence. This causes elements to be delayed with duration.

Parameters **duration** (Union[timedelta, float]) – Duration for taking elements from the start of the sequence.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns an observable sequence with the elements taken during the specified duration from the start of the source sequence.

giving.operators.**throttle**(window_duration, scheduler=None)
throttle() is an alias of *throttle_first()*

giving.operators.**throttle_first**(window_duration, scheduler=None)

Returns an Observable that emits only the first item emitted by the source Observable during sequential time windows of a specified duration.

Parameters **window_duration** (Union[timedelta, float]) – time to wait before emitting another item after emitting the last item.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns an observable that performs the throttle operation.

giving.operators.**throttle_with_mapper**(throttle_duration_mapper)

The throttle_with_mapper operator.

Ignores values from an observable sequence which are followed by another value within a computed throttle duration.

Example

```
>>> op = throttle_with_mapper(lambda x: rx.Scheduler.timer(x+x))
```

Parameters

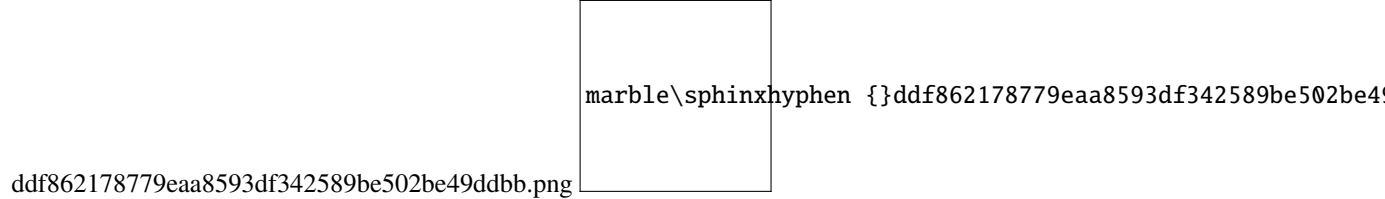
- **throttle_duration_mapper** (Callable[[Any], Observable[Any]]) – Mapper function to retrieve an
- **each** (observable sequence indicating the throttle duration for) –
- **element.** (given) –

Return type Callable[[Observable[~T]], Observable[~T]]

Returns A partially applied operator function that takes an observable source and returns the throttled observable sequence.

`giving.operators.throttle_with_timeout(duetime, scheduler=None)`

Ignores values from an observable sequence which are followed by another value before duetime.



Example

```
>>> res = debounce(5.0) # 5 seconds
```

Parameters

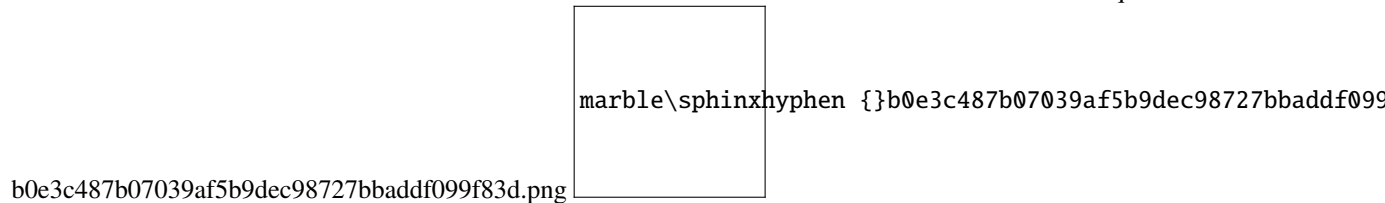
- **duetime** (Union[timedelta, float]) – Duration of the throttle period for each value (specified as a float denoting seconds or an instance of timedelta).
- **scheduler** (Optional[SchedulerBase]) – Scheduler to debounce values on.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes the source observable and returns the debounced observable sequence.

`giving.operators.time_interval(scheduler=None)`

Records the time interval between consecutive values in an observable sequence.



Examples

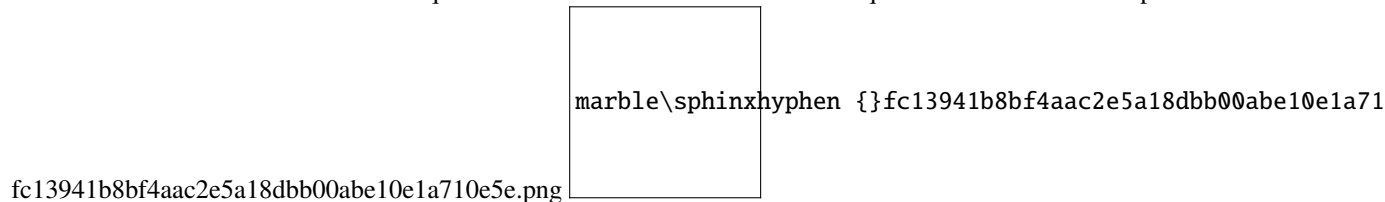
```
>>> res = time_interval()
```

Return type Callable[[Observable[~_T]], Observable[ForwardRef]]

Returns An operator function that takes an observable source and returns an observable sequence with time interval information on values.

`giving.operators.timeout(duetime, other=None, scheduler=None)`

Returns the source observable sequence or the other observable sequence if duetime elapses.



Examples

```
>>> res = timeout(5.0)
>>> res = timeout(datetime(), return_value(42))
>>> res = timeout(5.0, return_value(42))
```

Parameters

- **duetime** (Union[datetime, timedelta, float]) – Absolute (specified as a datetime object) or relative time (specified as a float denoting seconds or an instance of timedelta) when a timeout occurs.
- **other** (Optional[Observable[~T]]) – Sequence to return in case of a timeout. If not specified, a timeout error throwing sequence will be used.
- **scheduler** (Optional[SchedulerBase]) –

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns the source sequence switching to the other sequence in case of a timeout.

giving `operators.timeout_with_mapper(first_timeout=None, timeout_duration_mapper=None, other=None)`
Returns the source observable sequence, switching to the other observable sequence if a timeout is signaled.

Examples

```
>>> res = timeout_with_mapper(reactivex.timer(0.5))
>>> res = timeout_with_mapper(
    reactivex.timer(0.5), lambda x: reactivex.timer(0.2)
)
>>> res = timeout_with_mapper(
    reactivex.timer(0.5),
    lambda x: reactivex.timer(0.2),
    reactivex.return_value(42)
)
```

Parameters

- **first_timeout** (Optional[Observable[Any]]) – [Optional] Observable sequence that represents the timeout for the first element. If not provided, this defaults to `reactivex.never()`.
- **timeout_duration_mapper** (Optional[Callable[[~T], Observable[Any]]]) – [Optional] Selector to retrieve an observable sequence that represents the timeout between the current element and the next element.
- **other** (Optional[Observable[~T]]) – [Optional] Sequence to return in case of a timeout. If not provided, this is set to `reactivex.throw()`.

Return type Callable[[Observable[~T]], Observable[~T]]

Returns An operator function that takes an observable source and returns the source sequence switching to the other sequence in case of a timeout.

giving `operators.timestamp(scheduler=None)`
The timestamp operator.

Records the timestamp for each value in an observable sequence.

Examples

```
>>> timestamp()
```

Produces objects with attributes *value* and *timestamp*, where *value* is the original value.

Return type `Callable[[Observable[~_T]], Observable[ForwardRef]]`

Returns A partially applied operator function that takes an observable source and returns an observable sequence with timestamp information on values.

`giving.operators.to_dict(key_mapper, element_mapper=None)`

Converts the observable sequence to a Map if it exists.

Parameters

- **key_mapper** (`Callable[[~_T], ~_TKey]`) – A function which produces the key for the dictionary.
- **element_mapper** (`Optional[Callable[[~_T], ~_TValue]]`) – [Optional] An optional function which produces the element for the dictionary. If not present, defaults to the value from the observable sequence.

Return type `Callable[[Observable[~_T]], Observable[Dict[~_TKey, ~_TValue]]]`

Returns An operator function that takes an observable source and returns an observable sequence with a single value of a dictionary containing the values from the observable sequence.

`giving.operators.to_future(future_ctor=None)`

Converts an existing observable sequence to a Future.

Example

```
op = to_future(asyncio.Future);
```

Parameters **future_ctor** – [Optional] The constructor of the future.

Returns An operator function that takes an observable source and returns a future with the last value from the observable sequence.

`giving.operators.to_iterable()`

Creates an iterable from an observable sequence.

There is also an alias called `to_list`.

Return type `Callable[[Observable[~_T]], Observable[List[~_T]]]`

Returns An operator function that takes an observable source and returns an observable sequence containing a single element with an iterable containing all the elements of the source sequence.

`giving.operators.to_list()`

Creates an iterable from an observable sequence.

There is also an alias called `to_list`.

Return type `Callable[[Observable[~_T]], Observable[List[~_T]]]`

Returns An operator function that takes an observable source and returns an observable sequence containing a single element with an iterable containing all the elements of the source sequence.

`giving.operators.to_marbles(timespan=0.1, scheduler=None)`

Convert an observable sequence into a marble diagram string.

Parameters

- **timespan** (Union[timedelta, float]) – [Optional] duration of each character in second. If not specified, defaults to 0.1s.
- **scheduler** (Optional[SchedulerBase]) – [Optional] The scheduler used to run the the input sequence on.

Return type Callable[[Observable[Any]], Observable[str]]

Returns Observable stream.

`giving.operators.to_set()`

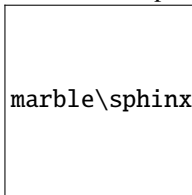
Converts the observable sequence to a set.

Return type Callable[[Observable[_T]], Observable[Set[_T]]]

Returns An operator function that takes an observable source and returns an observable sequence with a single value of a set containing the values from the observable sequence.

`giving.operators.top(n=10, key=None)`

Return the top n values, sorted in descending order. 7b6c1e8599749d074746f03a2e29ba534b8e454c.png



top may emit less than n elements, if there are less than n elements in the original sequence.

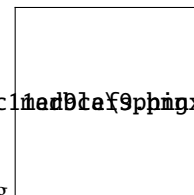
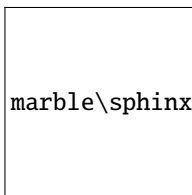
Parameters

- **n** – The number of top entries to return.
- **key** – The comparison key function to use or a string.

`giving.operators.variance(*, scan=False)`

`giving.operators.where(*keys, **conditions)`

Filter entries with the given keys meeting the given conditions. 224f5f799ed36630eb603290c5ccedc11ed9caf9.png



Example

```
where("x", "!y", z=True, w=lambda x: x > 0)
```

Parameters

- **keys** – Keys that must be present in the dictionary or, if a key starts with “!”, it must *not* be present.
- **conditions** – Maps a key to the value it must be associated to in the dictionary, or to a predicate function on the value.

giving.operators.**where_any**(*keys)

marble\sphinxhyphen {}52122

Filter entries with any of the given keys. 52122fe2fd7d5a4bf51af04628f21eb74171ce22.png

Parameters keys – Keys that must be present in the dictionary.

giving.operators.**while_do**(condition)

Repeats source as long as condition holds emulating a while loop.

Parameters condition (Callable[[Observable[~_T]], bool]) – The condition which determines if the source will be repeated.

Return type Callable[[Observable[~_T]], Observable[~_T]]

Returns An operator function that takes an observable source and returns an observable sequence which is repeated as long as the condition holds.

giving.operators.**window**(boundaries)

Projects each element of an observable sequence into zero or more windows.

marble\sphinxhyphen {}2c04b5c44b74070176595a3a4e681bd1e8d

2c04b5c44b74070176595a3a4e681bd1e8d6fcec.png

Examples

```
>>> res = window(reactivex.interval(1.0))
```

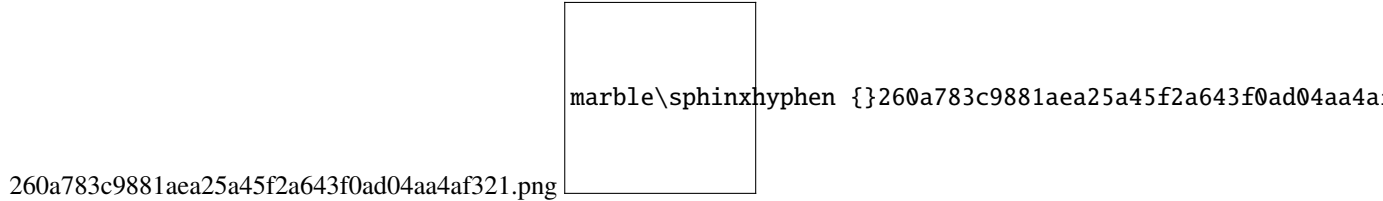
Parameters boundaries (Observable[Any]) – Observable sequence whose elements denote the creation and completion of non-overlapping windows.

Return type Callable[[Observable[~_T]], Observable[Observable[~_T]]]

Returns An operator function that takes an observable source and returns an observable sequence of windows.

giving.operators.**window_toggle**(*openings*, *closing_mapper*)

Projects each element of an observable sequence into zero or more windows.



```
>>> res = window(reactivex.interval(0.5), lambda i: reactivex.timer(i))
```

Parameters

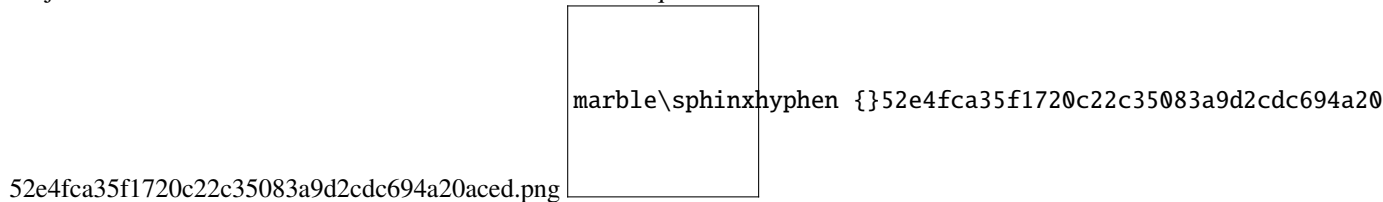
- **openings** (Observable[Any]) – Observable sequence whose elements denote the creation of windows.
- **closing_mapper** (Callable[[Any], Observable[Any]]) – A function invoked to define the closing of each produced window. Value from openings Observable that initiated the associated window is provided as argument to the function.

Return type Callable[[Observable[~T]], Observable[Observable[~T]]]

Returns An operator function that takes an observable source and returns an observable sequence of windows.

giving.operators.**window_when**(*closing_mapper*)

Projects each element of an observable sequence into zero or more windows.



Examples

```
>>> res = window(lambda: reactivex.timer(0.5))
```

Parameters **closing_mapper** (Callable[[], Observable[Any]]) – A function invoked to define the closing of each produced window. It defines the boundaries of the produced windows (a window is started when the previous one is closed, resulting in non-overlapping windows).

Return type Callable[[Observable[~T]], Observable[Observable[~T]]]

Returns An operator function that takes an observable source and returns an observable sequence of windows.

giving.operators.**window_with_count**(*count*, *skip=None*)

Projects each element of an observable sequence into zero or more windows which are produced based on element

count information. 4bec5ea503c9f8343c3fec6dda89105a28931cea.png

marble\sphinxhyphen {}4bec5ea503c9f8343c3fec6dda89105a28931cea.png

Examples

```
>>> window_with_count(10)
>>> window_with_count(10, 1)
```

Parameters

- **count** (int) – Length of each window.
- **skip** (Optional[int]) – [Optional] Number of elements to skip between creation of consecutive windows. If not specified, defaults to the count.

Return type Callable[[Observable[~_T]], Observable[Observable[~_T]]]

Returns An observable sequence of windows.

giving.operators.**window_with_time**(*timespan*, *timeshift=None*, *scheduler=None*)

Return type Callable[[Observable[~_T]], Observable[Observable[~_T]]]

giving.operators.**window_with_time_or_count**(*timespan*, *count*, *scheduler=None*)

Return type Callable[[Observable[~_T]], Observable[Observable[~_T]]]

giving.operators.**with_latest_from**(**sources*)

The *with_latest_from* operator.

Merges the specified observable sequences into one observable sequence by creating a tuple only when the first observable sequence produces an element. The observables can be passed either as separate arguments or as a

list. 1c1361f657a91ebf5ade83dbd9800bfa7ec5e7b5.png

marble\sphinxhyphen {}1c1361f657a91ebf5ade83dbd9800bfa7ec5e7b5.png

Examples

```
>>> op = with_latest_from(obs1)
>>> op = with_latest_from([obs1, obs2, obs3])
```

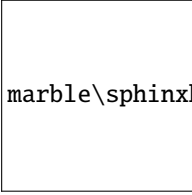
Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable source and returns an observable sequence containing the result of combining elements of the sources into a tuple.

`giving.operators.wmap(name, fn=None, pass_keys=True)`

Map each begin/end pair of a `give.wrap`.

In this schema, B and E correspond to the messages sent in the enter and exit phases respectively of the `wrap()`



marble\sphinxhyphen {}ae5183a952d0eb15135cf12

context manager. ae5183a952d0eb15135cf129cf63a68f11f34e80.png

Example

```
def _wrap(x):
    yield
    return x * 10

with given() as gv:
    results = gv.wmap("block", _wrap).accum()

    with give.wrap("block", x=3):
        with give.wrap("block", x=4):
            pass


assert results == [40, 30]
```

Parameters

- **name** – Name of the wrap block to group on.
- **fn** – A generator function that yields exactly once.
- **pass_keys** – Whether to pass the arguments to `give.wrap()` as keyword arguments at the start (defaults to True).

`giving.operators.zip(*args)`

Merges the specified observable sequences into one observable sequence by creating a tuple whenever all of the observable sequences have produced an element at a corresponding index.



marble\sphinxhyphen {}2dcc60783a078d2fa3e925688f6b6740cbb

2dcc60783a078d2fa3e925688f6b6740cbb65d36.png

Example

```
>>> res = zip(obs1, obs2)
```

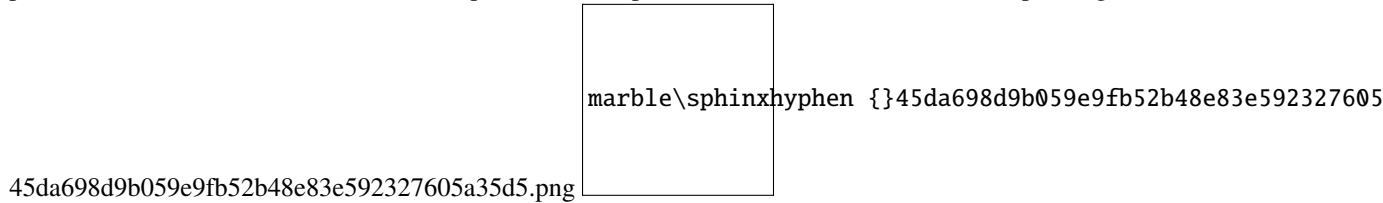
Parameters **args** (Observable[Any]) – Observable sources to zip.

Return type Callable[[Observable[Any]], Observable[Any]]

Returns An operator function that takes an observable source and returns an observable sequence containing the result of combining elements of the sources as a tuple.

giving.operators.**zip_with_iterable**(*second*)

Merges the specified observable sequence and list into one observable sequence by creating a tuple whenever all of the observable sequences have produced an element at a corresponding index.



Example

```
>>> res = zip([1,2,3])
```

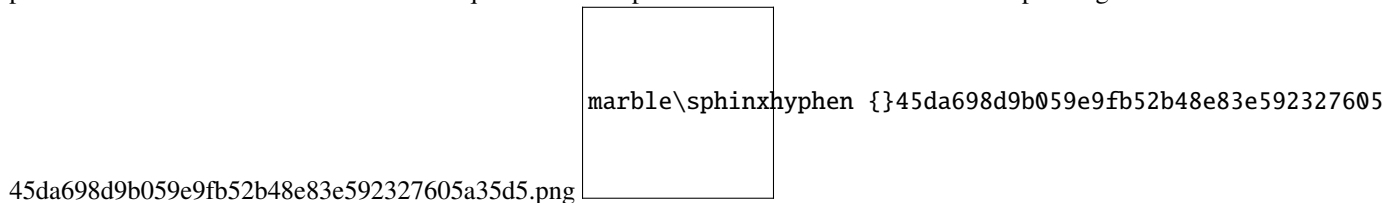
Parameters **second** (Iterable[~T2]) – Iterable to zip with the source observable..

Return type Callable[[Observable[~T1]], Observable[Tuple[~T1, ~T2]]]

Returns An operator function that takes and observable source and returns an observable sequence containing the result of combining elements of the sources as a tuple.

giving.operators.**zip_with_list**(*second*)

Merges the specified observable sequence and list into one observable sequence by creating a tuple whenever all of the observable sequences have produced an element at a corresponding index.



Example

```
>>> res = zip([1,2,3])
```

Parameters **second** (Iterable[~T2]) – Iterable to zip with the source observable..

Return type Callable[[Observable[~T1]], Observable[Tuple[~T1, ~T2]]]

Returns An operator function that takes and observable source and returns an observable sequence containing the result of combining elements of the sources as a tuple.

6.3 ptera.interpret

Create events on transformed functions based on selectors.

There are some extra features here compared to the standard Probe interface, namely *Total* which can accumulate multiple values for non-focus variables and is only triggered when the selector's outer function finishes.

```
class ptera.interpret.BaseAccumulator(*, selector, intercept=None, trigger=None, close=None,  
                                     parent=None, template=True, check=True, pass_info=False)
```

Accumulates the values of variables in Capture objects.

Under certain conditions, call user-provided event functions.

Any function given to the constructor must take one argument which is the dictionary of captures.

Parameters

- **selector** – The selector to use.
- **trigger** – The function to call when the focus variable is set.
- **intercept** – The function to call to override the value of the focus variable.
- **close** – The function to call when the selector is closed.
- **parent** – The parent Accumulator.
- **template** – Whether the Accumulator is a “template” and should be cloned prior to accumulating anything.
- **check** – Whether to filter that the values are correct in a selector such as $f(x=1) > y$. Otherwise the $=1$ would be ignored.
- **pass_info** – Whether to pass the accumulator and current triggered element to trigger or intercept.

build()

Build the dictionary of captures.

The built dictionary includes captures from the parents.

fork(selector=None)

Fork the Accumulator, possibly with a new selector.

Children Accumulators can accumulate new data while sharing what is accumulated by their parents.

getcap(element)

Get the Capture object for a leaf element.

```
class ptera.interpret.Capture(element)
```

Represents captured values for a variable.

Parameters **element** – The selector element for which we are capturing.

element

The selector element for which we are capturing.

capture

The variable name or alias corresponding to the capture (same as `element.capture`).

names

The list of names of the variables that match the element.

values

The list of values taken by matching variables.

accum(*varname, value*)

Accumulate a variable name and value.

property name

Name of the capture.

For a generic element such as `$x`, there may be multiple names, in which case the `.names` attribute should be used instead.

set(*varname, value*)

Set a variable name and value, overwriting the previous capture.

snapshot()

Return a snapshot of the capture at this moment.

property value

Value of the capture.

This only works if there is a unique value. Otherwise, you must use `.values`.

class `ptera.interpret.Immediate`(*selector, trigger=None, **kwargs*)

Accumulator triggered when the focus variable is set.

The Immediate accumulator only keeps the last value of each variable in the selector.

Any function given to the constructor must take one argument which is the dictionary of captures.

Parameters

- **selector** – The selector to use.
- **trigger** – The function to call when the focus variable is set.
- **intercept** – The function to call to override the value of the focus variable.
- **close** – The function to call when the selector is closed.

class `ptera.interpret.Interactor`(*fn, accumulators=None*)

Represents an interactor for a tooled function.

Define an `interact` method called by the tooled function when variables are changed.

exit()

Exit the interactor.

This triggers the close function on available accumulators.

interact(*varname, key, category, value, overridable*)

Interaction function called when setting a variable in a tooled function.

Parameters

- **varname** – The variable's name.
- **key** – The attribute or index set on the variable (as a Key object)
- **category** – The variable's category or tag (annotation)
- **value** – The value given to the variable in the original code.
- **overridable** – Whether the value can be overridden.

Returns The value to actually set the variable to.

register(*acc, captures, close_at_exit*)

Register an accumulator for a certain set of captures.

Parameters

- **acc** – An Accumulator.
- **captures** – A dictionary of elements to sets of matching variable names for which the accumulator will be triggered.
- **close_at_exit** – Whether to call the accumulator’s close function when the interactor exits.

work_on(varname, key, category)

Return a *WorkingFrame* for the given variable.

Parameters

- **varname** – The name of the variable.
- **key** – The key (attribute or index) that is being set on the variable.
- **category** – The variable’s category or tag.

exception ptera.interpret.OverrideException

Exception raised when trying to override a closure variable.

class ptera.interpret.Total(selector, close, trigger=None, **kwargs)

Accumulator usually triggered when the selector’s outer function ends.

The Total accumulator keeps all values taken by the variables in the selector for each value taken by the focus variable. For example, if the selector is $f(x) > g(!y) > h(z)$ and h is called multiple times for multiple values of z , they will all be accumulated together. However, if y is set multiple times, there will be multiple events.

Any function given to the constructor must take one argument which is the dictionary of captures.

Parameters

- **selector** – The selector to use.
- **close** – The function to call when the selector is closed.
- **trigger** – The function to call when the focus variable is set.
- **intercept** – The function to call to override the value of the focus variable.

class ptera.interpret.WorkingFrame(varname, key, category, accumulators)

Context manager to facilitate working on a variable.

intercept(tentative)

Execute the intercepts of all matching accumulators.

The last intercept that does not return ABSENT wins.

Parameters **tentative** – The tentative value for the variable, as provided in the original code.

Returns The value the intercepted variable should take.

log(value)

Log a value for the variable.

trigger()

Trigger an event using what was accumulated.

6.4 ptera.opparse

Parser for the selector syntax.

class ptera.opparse.**ASTNode**(*parts*)

Node that results from parsing.

args

List of arguments.

ops

List of operators. Generally one less than the number of arguments.

key

String key that represents the kind of operation we are dealing with: which arguments are non-null and what the ops are. If args == [a, b, None] and ops == [+ , -] then key == "X + X - _".

location

Location of the node in the source code. It encompasses the locations of all args and ops.

class ptera.opparse.**Lexer**(*definitions*)

The Lexer splits source code into Tokens.

class ptera.opparse.**OperatorPrecedenceTower**(*operators*)

Compare operators using a simple operator tower.

resolve(*op*)

Resolve the priority tuple for a given op.

class ptera.opparse.**Parser**(*lexer, order*)

Operator precedence parser.

finalize(*parts*)

Clean up a list of parts that form a completed ASTNode.

- If the parts are [None, op, None], this is just the op.
- If the parts are [arg1, op1, arg2, op2, ... argn] then we create an ASTNode with the given args and ops.

process(*tokens*)

Process a list of tokens.

class ptera.opparse.**Token**(*value, type, source, start, end*)

Token produced by the lexer.

value

Textual value of the token.

type

Type of the token.

location

Location of the token.

ptera.opparse.**cbrack**(*prio*)

Create a priority tuple for a closing bracket.

ptera.opparse.**lassoc**(*prio*)

Create a priority tuple for a left-associative operator.

ptera.opparse.**obrack**(*prio*)

Create a priority tuple for an opening bracket.

`ptera.opparse.rassoc(prio)`
 Create a priority tuple for a right-associative operator.

6.5 ptera.overlay

class `ptera.overlay.BaseOverlay(*handlers)`

An Overlay contains a set of selectors and associated rules.

When used as a context manager, the rules are applied within the with block.

Parameters `handlers` – A collection of handlers, each typically an instance of either *Immediate* or *Total*.

add(**handlers*)

Add new handlers.

fork()

Create a clone of this overlay.

class `ptera.overlay.HandlerCollection(handler_pairs=None)`

List of (selector, accumulator) pairs.

The selector in the pair may not be the same as accumulator.selector. When processing a selector such as `f > g > x`, after entering `f`, we may map the `g > x` selector to the same accumulator in a new collection that represents what should be done inside `f`.

plus(*handler_pairs*)

Clone this collection with additional (selector, accumulator) pairs.

proceed(*fn*)

Proceed into a call to `fn` with this collection.

Considers each selector to see if it matches `fn`. Returns an Interactor object for the call and a new HandlerCollection with the selectors to use inside the call.

class `ptera.overlay.Overlay(*handlers)`

An Overlay contains a set of selectors and associated rules.

When used as a context manager, the rules are applied within the with block.

Rules can be given in the constructor or built using helper methods such as `on`, `tapping` or `tweaking`.

Parameters `handlers` – A collection of handlers, each typically an instance of either *Immediate* or *Total*.

on(*selector, **kwargs*)

Make a decorator for a function to trigger on a selector.

Parameters

- **selector** – The selector to use.
- **full** – (default False) Whether to return a dictionary of Capture objects.
- **all** – (default False) If not full, whether to return a list of results for each variable or a single value.
- **immediate** – (default True) Whether to use an *Immediate()* accumulator. If False, use a *Total()* accumulator.

register(*selector, fn, full=False, all=False, immediate=True*)

Register a function to trigger on a selector.

Parameters

- **selector** – The selector to use.
- **fn** – The function to register.
- **full** – (default False) Whether to return a dictionary of Capture objects.
- **all** – (default False) If not full, whether to return a list of results for each variable or a single value.
- **immediate** – (default True) Whether to use an *Immediate* accumulator. If False, use a *Total* accumulator.

rewrite(*rewriters*, *full=False*)

Override the focus variables of selectors.

Parameters **rewriters** – A {selector: override_function} dictionary.

rewriting(*values*, *full=False*)

Fork this Overlay and *rewrite()*.

Can be called on the class (with *Overlay.rewriting(...)*).

tap(*selector*, *dest=None*, ***kwargs*)

Tap values from a selector into a list.

Parameters

- **selector** – The selector to use.
- **dest** – The list in which to append. If None, a list is created.

Returns The list in which to append.

tapping(*selector*, *dest=None*, ***kwargs*)

Context manager yielding a list in which results will be accumulated.

Can be called on the class (with *Overlay.tapping(...)*).

tweak(*values*)

Override the focus variables of selectors.

Parameters **values** – A {selector: value} dictionary.

tweaking(*values*)

Fork this Overlay and *tweak()*.

Can be called on the class (with *Overlay.tweaking(...)*).

ptera.overlay.**autotool**(*selector*, *undo=False*)

Automatically tool functions inplace.

Parameters **selector** – The selector to use as a basis for the tooling. Any function it refers to will be tooled.

ptera.overlay.**fits_selector**(*pfn*, *selector*)

Check whether a PteraFunction matches a selector.

Parameters

- **pfn** – The PteraFunction.
- **selector** – The selector. We are trying to match the outer scope.

class ptera.overlay.**proceed**(*fn*)

Context manager to wrap execution of a function.

This uses the current [HandlerCollection](#) to proceed through the current selectors.

Parameters *fn* – The function that will be executed.

Yields An Interactor that will be used by Ptera.

ptera.overlay.**tooled**(*fn*)

Tool a function so that it can report changes in its variables to Ptera.

@tooled can be used as a decorator.

Note: You may write @tooled.inplace as a decorator to tool a function inplace.

Parameters *fn* – The function to tool.

6.6 ptera.probe

This module defines the probing functionality. The interface for probes is built on [giving](#)

ptera.probe.**probing**(**selectors*, *raw=False*, *probe_type=None*, *env=None*, *overridable=False*)

Probe that can be used as a context manager.

Example:

```
>>> def f(x):
...     a = x * x
...     return a
```

```
>>> with probing("f > a").print():
...     f(4) # Prints {"a": 16}
```

Parameters

- **selectors** – The selector strings describing the variables to probe (at least one).
- **raw** – Defaults to False. If True, produce a stream of [Capture](#) objects that contain extra information about the capture.
- **probe_type** – Either “immediate”, “total”, or None (the default).
 - If “immediate”, use [Immediate](#).
 - If “total”, use [Total](#).
 - If None, determine what to use based on whether the selector has a focus or not.
- **env** – A dictionary that will be used to resolve symbols in the selector. If it is not provided, ptera will seek the locals and globals dictionaries of the scope where this function is called.
- **overridable** – Whether to include the override/koverride methods.

ptera.probe.**global_probe**(**selectors*, *raw=False*, *probe_type=None*, *env=None*)

Set a probe globally.

Example:

```
>>> def f(x):
...     a = x * x
...     return a
```

```
>>> probe = global_probe("f > a")
>>> probe["a"].print()
>>> f(4) # Prints 16
```

Parameters

- **selectors** – The selector strings describing the variables to probe (at least one).
- **raw** – Defaults to False. If True, produce a stream of *Capture* objects that contain extra information about the capture.
- **probe_type** – Either “immediate”, “total”, or None (the default).
 - If “immediate”, use *Immediate*.
 - If “total”, use *Total*.
 - If None, determine what to use based on whether the selector has a focus or not.
- **env** – A dictionary that will be used to resolve symbols in the selector. If it is not provided, ptera will seek the locals and globals dictionaries of the scope where this function is called.

class ptera.probe.**Probe**(*selectors, raw=False)

Observable which generates a stream of values from program variables.

Probes should be created with *probing()* or *global_probe()*.

Note: In the documentation for some methods you may see calls to *give()* or *given()*, but that’s because they come from the documentation for the *giving* package (on top of which Probe is built).

give() is equivalent to what Ptera does when a variable of interest is set, *given()* yields an object that has the same interface as *Probe* (the superclass to *Probe*, in fact). Take variables named *gv* to be probes.

Parameters

- **selectors** – The selector strings describing the variables to probe (at least one).
- **raw** – Defaults to False. If True, produce a stream of *Capture* objects that contain extra information about the capture. Mostly relevant for advanced selectors such as *f > \$x:@Parameter* which captures the value of any variable with the Parameter tag under the generic name “x”. When raw is True, the actual name of the variable is preserved in a Capture object associated to x.
- **probe_type** – Either “immediate”, “total”, or None (the default).
 - If “immediate”, use *Immediate*.
 - If “total”, use *Total*.
 - If None, determine what to use based on whether the selector has a focus or not.
- **env** – A dictionary that will be used to resolve symbols in the selector. If it is not provided, ptera will seek the locals and globals dictionaries of the scope where this function is called.

accum(*obj=None*)

Accumulate into a list or set.

Parameters **obj** – The object in which to accumulate, either a list or a set. If not provided, a new list is created.

Returns The object in which the values will be accumulated.

allow_empty()

Suppresses SequenceContainsNoElementsError.

This can be chained to `reduce()`, `min()`, `first()`, etc. of an empty sequence to allow the output of these operations to be empty. Otherwise, these operations would raise `rx.internal.exceptions.SequenceContainsNoElementsError`.

breakpoint(*args, skip=[], **kwargs)

Trigger a breakpoint on every entry.

Parameters **skip** – A list of globs corresponding to modules to skip during debugging, for example `skip=["giving.*"]` would skip all frames that are in the `giving` module.

breakword(*args, skip=[], **kwargs)

Trigger a breakpoint using `breakword`.

This feature requires the `breakword` package to be installed, and the `tag()` operator to be applied.

```
gvt = gv.tag()
gvt.display()
gvt.breakword()
```

The above will display words next to each entry. Set the `BREAKWORD` environment to one of these words to set a breakpoint when it is printed.

Parameters

- **skip** – A list of globs corresponding to modules to skip during debugging, for example `skip=["giving.*"]` would skip all frames that are in the `giving` module.
- **word** – Only trigger the breakpoint on the given word.

display(*, breakword=None, skip=[], **kwargs)

Pretty-print each element.

Parameters

- **colors** – Whether to colorize the output or not.
- **time_format** – How to format the time (if present), e.g. `"%Y-%m-%d %H:%M:%S"`
- **breakword** – If not `None`, run `self.breakword(word=breakword)`.
- **skip** – If `breakword` is not `None`, pass `skip` to the debugger.

eval(fn, *args, **kwargs)

Run a function in the context of this Given and get the values.

```
def main():
    give(x=1)
    give(x=2)

values = given()["x"].eval(main)
assert values == [1, 2]
```


Parameters

- **fn** – The function to run.
- **args** – Positional arguments to pass to fn.
- **kwargs** – Keyword arguments to pass to fn.

exec(fn, *args, **kwargs)

Run a function in the context of this Given.

```
def main():
    give(x=1)
    give(x=2)

gv = given()
gv["x"].print()
gv.exec(main) # prints 1, 2
```

Parameters

- **fn** – The function to run.
- **args** – Positional arguments to pass to fn.
- **kwargs** – Keyword arguments to pass to fn.

fail(*args, skip=[], **kwargs)

Raise an exception if the stream produces anything.

Parameters

- **message** – The exception message (format).
- **exc_type** – The exception type to raise. Will be passed the next data element, and the result is raised. Defaults to **Failure**.
- **skip** – Modules to skip in the traceback. Defaults to “giving.*” and “rx.*”.

fail_if_empty(message=None, exc_type=<class 'giving.gvn.Failure'>, skip=['giving.*', 'rx.*'])

Raise an exception if the stream is empty.

Parameters **exc_type** – The exception type to raise. Defaults to **Failure**.

fail_if_false(*args, skip=[], **kwargs)

Raise an exception if any element of the stream is falsey.

False, 0, [], etc. are falsey.

Parameters **exc_type** – The exception type to raise. Defaults to **Failure**.

give(*keys, **extra)

Give each element.

This calls **give()** for each value in the stream.

Be careful using this method because it could easily lead to an infinite loop.

Parameters

- **keys** – Key(s) under which to give the elements.
- **extra** – Extra key/value pairs to give along with the rest.

ksubscribe(*fn*)

Subscribe a function called with keyword arguments.

Note: The function passed to `ksubscribe` is wrapped with `lax_function()`, so it is not necessary to add a `**kwargs` argument for keys that you do not need.

```
gv.ksubscribe(lambda x, y=None, z=None: print(x, y, z))
give(x=1, z=2, abc=3)  # Prints 1, None, 2
```

Parameters *fn* – The function to call.

pipe(ops*)**

Pipe one or more operators.

Returns: An ObservableProxy.

print(*format=None, skip_missing=False*)

Print each element of the stream.

Parameters

- **format** – A format string as would be used with `str.format`.
- **skip_missing** – Whether to ignore KeyErrors due to missing entries in the format.

subscribe(*observer=None, on_next=None, on_error=None, on_completed=None*)

Subscribe a function to this Observable stream.

```
with given() as gv:
    gv.subscribe(print)

results = []
gv["x"].subscribe(results.append)

give(x=1)  # prints {"x": 1}
give(x=2)  # prints {"x": 2}

assert results == [1, 2]
```

Parameters

- **observer** – The object that is to receive notifications.
- **on_error** – Action to invoke upon exceptional termination of the observable sequence.
- **on_completed** – Action to invoke upon graceful termination of the observable sequence.
- **on_next** – Action to invoke for each element in the observable sequence.

Returns An object representing the subscription with a `dispose()` method to remove it.

values()

Context manager to accumulate the stream into a list.

```
with given()["?x"].values() as results:
    give(x=1)
    give(x=2)
```

(continues on next page)

(continued from previous page)

```
assert results == [1, 2]
```

Note that this will activate the root `given()` including all subscriptions that it has (directly or indirectly).

wrap(*name*, *fn*=None, *pass_keys*=False, *return_function*=False)

Subscribe a context manager, corresponding to `wrap()`.

```
@gv.wrap("main")
def _():
    print("<")
    yield
    print(">")

with give.wrap("main"):    # prints <
    ...
    with give.wrap("sub"):
        ...
    ...
    # prints >
```

Parameters

- **name** – The name of the wrap block to subscribe to.
- **fn** – The wrapper function OR an object with an `__enter__` method. If the wrapper is a generator, it will be wrapped with `contextmanager(fn)`. If a function, it will be called with no arguments, or with the arguments given to `give.wrap` if `pass_keys=True`.
- **pass_keys** – Whether to pass the arguments to `give.wrap` to this function as keyword arguments. You may use `kwargs` as a shortcut to `pass_keys=True`.

__or__(*other*)

Alias for `merge()`.

Merge this ObservableProxy with another.

__rshift__(*subscription*)

Alias for `subscribe()`.

If `subscription` is a list or a set, accumulate into it.

__getitem__(*item*)

Mostly an alias for `getitem()`.

Extra feature: if the item starts with "?", `getitem` is called with `strict=False`.

Other methods: All operators in the *operator list* have a corresponding method on Probe.

- `affix()`
- `all()`
- `amb()`
- `as_()`
- `as_observable()`
- `augment()`

- *average()*
- *average_and_variance()*
- *bottom()*
- *buffer()*
- *buffer_toggle()*
- *buffer_when()*
- *buffer_with_count()*
- *buffer_with_time()*
- *buffer_with_time_or_count()*
- *catch()*
- *collect_between()*
- *combine_latest()*
- *concat()*
- *contains()*
- *count()*
- *debounce()*
- *default_if_empty()*
- *delay()*
- *delay_subscription()*
- *delay_with_mapper()*
- *dematerialize()*
- *distinct()*
- *distinct_until_changed()*
- *do()*
- *do_action()*
- *do_while()*
- *element_at()*
- *element_at_or_default()*
- *exclusive()*
- *expand()*
- *filter()*
- *filter_indexed()*
- *finally_action()*
- *find()*
- *find_index()*
- *first()*

- `first_or_default()`
- `flat_map()`
- `flat_map_indexed()`
- `flat_map_latest()`
- `fork_join()`
- `format()`
- `getitem()`
- `group_by()`
- `group_by_until()`
- `group_join()`
- `group_wrap()`
- `ignore_elements()`
- `is_empty()`
- `join()`
- `keep()`
- `kfilter()`
- `kmap()`
- `kmerge()`
- `kscan()`
- `last()`
- `last_or_default()`
- `map()`
- `map_indexed()`
- `materialize()`
- `max()`
- `merge()`
- `merge_all()`
- `min()`
- `multicast()`
- `observe_on()`
- `on_error_resume_next()`
- `pairwise()`
- `partition()`
- `partition_indexed()`
- `pluck()`
- `pluck_attr()`

- `publish()`
- `publish_value()`
- `reduce()`
- `ref_count()`
- `repeat()`
- `replay()`
- `retry()`
- `roll()`
- `sample()`
- `scan()`
- `sequence_equal()`
- `share()`
- `single()`
- `single_or_default()`
- `single_or_default_async()`
- `skip()`
- `skip_last()`
- `skip_last_with_time()`
- `skip_until()`
- `skip_until_with_time()`
- `skip_while()`
- `skip_while_indexed()`
- `skip_with_time()`
- `slice()`
- `some()`
- `sole()`
- `starmap()`
- `starmap_indexed()`
- `start_with()`
- `subscribe_on()`
- `sum()`
- `switch_latest()`
- `tag()`
- `take()`
- `take_last()`
- `take_last_buffer()`

- `take_last_with_time()`
- `take_until()`
- `take_until_with_time()`
- `take_while()`
- `take_while_indexed()`
- `take_with_time()`
- `throttle()`
- `throttle_first()`
- `throttle_with_mapper()`
- `throttle_with_timeout()`
- `time_interval()`
- `timeout()`
- `timeout_with_mapper()`
- `timestamp()`
- `to_dict()`
- `to_future()`
- `to_iterable()`
- `to_list()`
- `to_marbles()`
- `to_set()`
- `top()`
- `variance()`
- `where()`
- `where_any()`
- `while_do()`
- `window()`
- `window_toggle()`
- `window_when()`
- `window_with_count()`
- `window_with_time()`
- `window_with_time_or_count()`
- `with_latest_from()`
- `zip()`
- `zip_with_iterable()`
- `zip_with_list()`

class ptera.probe.OverridableProbe(*selectors, raw=False)

Probe that allows overriding the values of variables.

OverridableProbe works essentially like [Probe](#), but it also has the [override\(\)](#) and [koverride\(\)](#) methods, which can be used to inject different values in the probed variables.

OverridableProbe is triggered before Probe, so a Probe will see the changes of an OverridableProbe. However, if there are multiple operations on an OverridableProbe, these operations will proceed using the old, non-overridden values.

koverride(setter)

Override the value of the focus variable using a setter function with kwargs.

```
def f(x):
    ...
    y = 123
    ...

# This will basically override y = 123 to become y = x + 123
OverridableProbe("f(x) > y").koverride(lambda x, y: x + y)
```

Note: Important: [override\(\)](#) only overrides the **focus variable**. The focus variable is the one to the right of `>`, or the one prefixed with `!`.

See [override\(\)](#).

Parameters **setter** – A function that takes a value from the pipeline as keyword arguments and produces the value to set the focus variable to.

override(setter=<function _identity>)

Override the value of the focus variable using a setter function.

```
# Increment a whenever it is set (will not apply recursively)
OverridableProbe("f > a")["a"].override(lambda value: value + 1)
```

Note: Important: [override\(\)](#) only overrides the **focus variable**. The focus variable is the one to the right of `>`, or the one prefixed with `!`.

This is because a Ptera selector is triggered when the focus variable is set, so realistically it is the only one that it makes sense to override.

Be careful, because it is easy to write misleading code:

```
# THIS WILL SET y = x + 1, NOT x
OverridableProbe("f(x) > y")["x"].override(lambda x: x + 1)
```

Note: [override](#) will only work at the end of a synchronous pipe (map/filter are OK, but not e.g. sample)

Parameters **setter** – A function that takes a value from the pipeline and produces the value to set the focus variable to.

- If not provided, the value from the stream is used as-is.

- If not callable, set the variable to the value of setter

6.7 ptera.selector

Specifications for call paths.

class ptera.selector.**Call**(**kwargs)

Represents a call in the call stack.

encode()

Return a string representation of the selector.

problems()

Return a list of problems with this selector.

- Wildcards are not allowed for cuntings.
- All functions should be tooled.
- All captured variables should exist in their respective functions.
- For wildcard variables that specify a tag/category, at least one variable should match.

specialize(specializations)

Replace \$variables in the selector using a specializations dict.

class ptera.selector.**Element**(**kwargs)

Represents a variable or some other atom.

encode()

Return a string representation of the selector.

specialize(specializations)

Replace \$variables in the selector using a specializations dict.

class ptera.selector.**Evaluator**

Evaluator that transforms the parse tree into a Selector.

class ptera.selector.**Selector**(**kwargs)

Represents a selector for variables in a call stack.

exception ptera.selector.**SelectorError**

Error raised for invalid selectors.

ptera.selector.**check_element**(el, name, category)

Check if Element el matches the given name and category.

ptera.selector.**dict_resolver**(env)

Resolve a symbol from a dictionary, e.g. the globals directory.

ptera.selector.**select**(s, env=None, skip_modules=[], skip_frames=0, strict=False)

Create a selector from a string.

Parameters

- **s** – The string to compile to a Selector, or a Selector to return unchanged.
- **env** – The environment to use to evaluate symbols in the selector. If not given, the environment chosen is the parent scope.
- **skip_modules** – Modules to skip when looking for an environment. We will go up through the stack until we get to a scope that is outside these modules.

- **skip_frames** – Number of frames to skip when looking for an environment.
- **strict** – Whether to require functions and variables in the selector to be statically resolvable (will give better errors).

`ptera.selector.verify(selector, display=None)`

Verify that the selector is resolvable.

This raises an exception if `problems()` returns any problems.

- Wildcards are not allowed for cuntions.
- All functions should be tooled.
- All captured variables should exist in their respective functions.
- For wildcard variables that specify a tag/category, at least one variable should match.

6.8 ptera.tags

Tag system for variables.

Variables can be tagged as e.g. `x: ptera.tag.Important` and the selectors `x:Important` or `*:Important` would match it. Alternatively, Ptera recognizes `x: "@Important"` as referring to these tags.

class `ptera.tags.Tag(name)`

Tag for a variable, to be used as an annotation.

Parameters `name` – The name of the tag.

class `ptera.tags.TagSet(members)`

Set of multiple tags.

`ptera.tags.get_tags(*tags)`

Build a Tag or TagSet from strings.

`ptera.tags.match_tag(to_match, tg)`

Return whether two Tags or TagSets match.

Only tg can be a TagSet.

6.9 ptera.transform

Code transform that instruments probed functions.

class `ptera.transform.ExternalVariableCollector(tree, comments, closure_vars)`

Collect variables referred to but not defined in the given AST.

The attributes are filled after the object is created.

used

Set of used variable names (does not include the names of inner functions).

assigned

Set of assigned variable names.

vardoc

Dict that maps variable names to matching comments.

provenance

Dict that maps variable names to “body” or “argument” if they are defined as variables in the body or as function arguments.

funcnames

Set of function names defined in the body.

class ptera.transform.**Key**(*type, value*)

Represents an attribute or index on a variable.

type

Either “attr” or “index”.

value

The value of the attribute or index.

affix_to(*sym*)

Return a string representing getting the key from sym.

```
>>> Key("attr", "y").affix_to("x")
"x.y"
>>> Key("index", "y").affix_to("x")
"x['y']"
```

exception ptera.transform.**PteraNameError**(*varname, function*)

The Ptera equivalent of a NameError, which gives more information.

info()

Return information about the missing variable.

class ptera.transform.**PteraTransformer**(*tree, evc, lib, filename, glb, to_instrument*)

Transform the AST of a function to instrument it with ptera.

The *result* field is set to the AST of the transformed function after instantiation of the PteraTransformer.

make_interaction(*target, ann, value, orig=None, expression=False*)

Create code for setting the value of a variable.

visit_Assign(*node*)

Rewrite an annotated assignment statement.

Before: x: int

After: x: int = _ptera_interact('x', int)

visit_Assign(*node*)

Rewrite an assignment statement.

Before: x = y + z

After: x = _ptera_interact('x', None, y + z)

visit_Import(*node*)

Rewrite an import statement.

Before: import kangaroo

After: import kangaroo kangaroo = _ptera_interact('kangaroo', None, kangaroo)

visit_ImportFrom(*node*)

Rewrite an import statement.

Before: from kangaroo import jump

After: from kangaroo import jump jump = _ptera_interact('jump', None, jump)

visit_NamedExpr(*node*)

Rewrite an assignment expression.

Before: `x := y + z`

After: `x := _ptera_interact('x', None, y + z)`

class `ptera.transform.SimpleVariableCollector`(*tree*)

`ptera.transform.name_error`(*varname*, *function*, *pop_frames=1*)

Raise a `PteraNameError` pointing to the right location.

`ptera.transform.transform`(*fn*, *proceed*, *to_instrument=True*, *set_conformer=True*)

Return an instrumented version of *fn*.

The transform roughly works as follows.

```
def f(x: int):
    y = x * x
    return y + 1
```

Becomes:

```
def f(x: int):
    with proceed(f) as FR:
        FR.interact("#enter", None, None, True, False)
        x = FR.interact("x", None, int, x, True)
        y = FR.interact("y", None, None, x * x, True)
        VALUE = FR.interact("#value", None, None, y + 1, True)
    return VALUE
```

Parameters

- **fn** – The function to instrument.
- **proceed** – A context manager that will wrap the function body and which should yield some object that has an `interact` method. Whenever a variable is changed, the `interact` method receives the arguments (symbol, key, category, value, overridable). See [proceed](#) and [interact](#).
- **to_instrument** – List of [Element](#) representing the variables to instrument, or `True`. If `True` (or if one `Element` is a generic), all variables are instrumented.
- **set_conformer** – Whether to set a “conformer” on the resulting function which will update the code when the original code is remapped through the `codefind` module (e.g. if you use `jurigged` to change source while it is running, the conformer will update the instrumentation to correspond to the new version of the function). Mostly for internal use.

Returns

A new function that is an instrumented version of the old one. The function has the following properties set:

- `__ptera_info__`: An info dictionary about all variables used in the function, their provenance, annotations and comments.
- `__ptera_token__`: The name of the global variable in which the function is tucked so that it can refer to itself.

6.10 ptera.utils

Miscellaneous utilities.

exception `ptera.utils.CodeNotFoundError`

class `ptera.utils.Named(name)`

A named object.

This class can be used to construct objects with a name that will be used for the string representation.

class `ptera.utils.autocreate(fn)`

Automatically create an instance when called on the class.

Basically makes it so that `Klass.f()` is equivalent to `Klass().f()`.

class `ptera.utils.cached_property(fn)`

Property that caches its value when we get it for the first time.

`ptera.utils.is_toolled(fn)`

Return whether a function has been tooled for Ptera.

`ptera.utils.keyword_decorator(deco)`

Wrap a decorator to optionally takes keyword arguments.

`ptera.utils.refstring(fn)`

Return the canonical reference string to select fn.

For example, if fn is called `bloop` and is located in module `squid.game`, the refstring will be `/squid.game/bloop`.

INDICES AND TABLES

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

PYTHON MODULE INDEX

g

`giving.operators`, 30

p

`ptera.interpret`, 92

`ptera.opparse`, 95

`ptera.overlay`, 96

`ptera.probe`, 98

`ptera.selector`, 109

`ptera.tags`, 110

`ptera.transform`, 110

`ptera.utils`, 113

Symbols

`__getitem__()` (*ptera.probe.Probe* method), 103
`__or__()` (*ptera.probe.Probe* method), 103
`__rshift__()` (*ptera.probe.Probe* method), 103

A

`accum()` (*ptera.interpret.Capture* method), 92
`accum()` (*ptera.probe.Probe* method), 99
`add()` (*ptera.overlay.BaseOverlay* method), 96
`affix()` (in module *giving.operators*), 30
`affix_to()` (*ptera.transform.Key* method), 111
`all()` (in module *giving.operators*), 31
`allow_empty()` (*ptera.probe.Probe* method), 100
`amb()` (in module *giving.operators*), 31
`args` (*ptera.opparse.ASTNode* attribute), 95
`as_()` (in module *giving.operators*), 31
`as_observable()` (in module *giving.operators*), 32
`assigned` (*ptera.transform.ExternalVariableCollector* attribute), 110
`ASTNode` (class in *ptera.opparse*), 95
`augment()` (in module *giving.operators*), 32
`autocreate` (class in *ptera.utils*), 113
`autotool()` (in module *ptera.overlay*), 97
`average()` (in module *giving.operators*), 32
`average_and_variance()` (in module *giving.operators*), 32

B

`BaseAccumulator` (class in *ptera.interpret*), 92
`BaseOverlay` (class in *ptera.overlay*), 96
`bottom()` (in module *giving.operators*), 33
`breakpoint()` (*ptera.probe.Probe* method), 100
`breakword()` (*ptera.probe.Probe* method), 100
`buffer()` (in module *giving.operators*), 33
`buffer_toggle()` (in module *giving.operators*), 33
`buffer_when()` (in module *giving.operators*), 34
`buffer_with_count()` (in module *giving.operators*), 34
`buffer_with_time()` (in module *giving.operators*), 35
`buffer_with_time_or_count()` (in module *giving.operators*), 35
`build()` (*ptera.interpret.BaseAccumulator* method), 92

C

`cached_property` (class in *ptera.utils*), 113
`Call` (class in *ptera.selector*), 109
`Capture` (class in *ptera.interpret*), 92
`capture` (*ptera.interpret.Capture* attribute), 92
`catch()` (in module *giving.operators*), 36
`cbrack()` (in module *ptera.opparse*), 95
`check_element()` (in module *ptera.selector*), 109
`CodeNotFoundError`, 113
`collect_between()` (in module *giving.operators*), 36
`combine_latest()` (in module *giving.operators*), 37
`concat()` (in module *giving.operators*), 37
`contains()` (in module *giving.operators*), 38
`count()` (in module *giving.operators*), 38

D

`debounce()` (in module *giving.operators*), 38
`default_if_empty()` (in module *giving.operators*), 39
`delay()` (in module *giving.operators*), 39
`delay_subscription()` (in module *giving.operators*), 40
`delay_with_mapper()` (in module *giving.operators*), 40
`dematerialize()` (in module *giving.operators*), 41
`dict_resolver()` (in module *ptera.selector*), 109
`display()` (*ptera.probe.Probe* method), 100
`distinct()` (in module *giving.operators*), 41
`distinct_until_changed()` (in module *giving.operators*), 42
`do()` (in module *giving.operators*), 42
`do_action()` (in module *giving.operators*), 43
`do_while()` (in module *giving.operators*), 43

E

`Element` (class in *ptera.selector*), 109
`element` (*ptera.interpret.Capture* attribute), 92
`element_at()` (in module *giving.operators*), 44
`element_at_or_default()` (in module *giving.operators*), 44
`encode()` (*ptera.selector.Call* method), 109
`encode()` (*ptera.selector.Element* method), 109
`eval()` (*ptera.probe.Probe* method), 100
`Evaluator` (class in *ptera.selector*), 109

exclusive() (in module giving.operators), 45
 exec() (ptera.probe.Probe method), 101
 exit() (ptera.interpret.Interactor method), 93
 expand() (in module giving.operators), 45
 ExternalVariableCollector (class
 ptera.transform), 110

F

fail() (ptera.probe.Probe method), 101
 fail_if_empty() (ptera.probe.Probe method), 101
 fail_if_false() (ptera.probe.Probe method), 101
 filter() (in module giving.operators), 45
 filter_indexed() (in module giving.operators), 45
 finalize() (ptera.opparse.Parser method), 95
 finally_action() (in module giving.operators), 46
 find() (in module giving.operators), 46
 find_index() (in module giving.operators), 47
 first() (in module giving.operators), 47
 first_or_default() (in module giving.operators), 47
 fits_selector() (in module ptera.overlay), 97
 flat_map() (in module giving.operators), 48
 flat_map_indexed() (in module giving.operators), 49
 flat_map_latest() (in module giving.operators), 50
 fork() (ptera.interpret.BaseAccumulator method), 92
 fork() (ptera.overlay.BaseOverlay method), 96
 fork_join() (in module giving.operators), 50
 format() (in module giving.operators), 50
 funcnames (ptera.transform.ExternalVariableCollector
 attribute), 111

G

get_tags() (in module ptera.tags), 110
 getcap() (ptera.interpret.BaseAccumulator method), 92
 getitem() (in module giving.operators), 51
 give() (ptera.probe.Probe method), 101
 giving.operators
 module, 30
 global_probe() (in module ptera.probe), 98
 group_by() (in module giving.operators), 51
 group_by_until() (in module giving.operators), 52
 group_join() (in module giving.operators), 52
 group_wrap() (in module giving.operators), 53

H

HandlerCollection (class in ptera.overlay), 96

I

ignore_elements() (in module giving.operators), 53
 Immediate (class in ptera.interpret), 93
 info() (ptera.transform.PteraNameError method), 111
 interact() (ptera.interpret.Interactor method), 93
 Interactor (class in ptera.interpret), 93
 intercept() (ptera.interpret.WorkingFrame method),
 94

is_empty() (in module giving.operators), 54
 is_toolled() (in module ptera.utils), 113

J

in join() (in module giving.operators), 54

K

keep() (in module giving.operators), 54
 Key (class in ptera.transform), 111
 key (ptera.opparse.ASTNode attribute), 95
 keyword_decorator() (in module ptera.utils), 113
 kfilter() (in module giving.operators), 55
 kmap() (in module giving.operators), 55
 kmerge() (in module giving.operators), 55
 koverride() (ptera.probe.OverridableProbe method),
 108
 kscan() (in module giving.operators), 56
 ksubscribe() (ptera.probe.Probe method), 101

L

lassoc() (in module ptera.opparse), 95
 last() (in module giving.operators), 56
 last_or_default() (in module giving.operators), 56
 Lexer (class in ptera.opparse), 95
 location (ptera.opparse.ASTNode attribute), 95
 location (ptera.opparse.Token attribute), 95
 log() (ptera.interpret.WorkingFrame method), 94

M

make_interaction() (ptera.transform.PteraTransformer
 method), 111
 map() (in module giving.operators), 57
 map_indexed() (in module giving.operators), 57
 match_tag() (in module ptera.tags), 110
 materialize() (in module giving.operators), 58
 max() (in module giving.operators), 58
 merge() (in module giving.operators), 58
 merge_all() (in module giving.operators), 59
 min() (in module giving.operators), 59
 module
 giving.operators, 30
 ptera.interpret, 92
 ptera.opparse, 95
 ptera.overlay, 96
 ptera.probe, 98
 ptera.selector, 109
 ptera.tags, 110
 ptera.transform, 110
 ptera.utils, 113
 multicast() (in module giving.operators), 59

N

name (ptera.interpret.Capture property), 93

`name_error()` (in module *ptera.transform*), 112
`Named` (class in *ptera.utils*), 113
`names` (*ptera.interpret.Capture* attribute), 92

O

`obrack()` (in module *ptera.opparse*), 95
`observe_on()` (in module *giving.operators*), 60
`on()` (*ptera.overlay.Overlay* method), 96
`on_error_resume_next()` (in module *giving.operators*), 60
`OperatorPrecedenceTower` (class in *ptera.opparse*), 95
`ops` (*ptera.opparse.ASTNode* attribute), 95
`Overlay` (class in *ptera.overlay*), 96
`OverridableProbe` (class in *ptera.probe*), 107
`override()` (*ptera.probe.OverridableProbe* method), 108
`OverrideException`, 94

P

`pairwise()` (in module *giving.operators*), 61
`Parser` (class in *ptera.opparse*), 95
`partition()` (in module *giving.operators*), 61
`partition_indexed()` (in module *giving.operators*), 61
`pipe()` (*ptera.probe.Probe* method), 102
`pluck()` (in module *giving.operators*), 62
`pluck_attr()` (in module *giving.operators*), 62
`plus()` (*ptera.overlay.HandlerCollection* method), 96
`print()` (*ptera.probe.Probe* method), 102
`Probe` (class in *ptera.probe*), 99
`probing()` (in module *ptera.probe*), 98
`problems()` (*ptera.selector.Call* method), 109
`proceed()` (in module *ptera.overlay*), 97
`proceed()` (*ptera.overlay.HandlerCollection* method), 96
`process()` (*ptera.opparse.Parser* method), 95
`provenance` (*ptera.transform.ExternalVariableCollector* attribute), 110
`ptera.interpret`
 module, 92
`ptera.opparse`
 module, 95
`ptera.overlay`
 module, 96
`ptera.probe`
 module, 98
`ptera.selector`
 module, 109
`ptera.tags`
 module, 110
`ptera.transform`
 module, 110
`ptera.utils`
 module, 113
`PteraNameError`, 111

`PteraTransformer` (class in *ptera.transform*), 111
`publish()` (in module *giving.operators*), 62
`publish_value()` (in module *giving.operators*), 63

R

`rassoc()` (in module *ptera.opparse*), 95
`reduce()` (in module *giving.operators*), 63
`ref_count()` (in module *giving.operators*), 64
`refstring()` (in module *ptera.utils*), 113
`register()` (*ptera.interpret.Interactor* method), 93
`register()` (*ptera.overlay.Overlay* method), 96
`repeat()` (in module *giving.operators*), 64
`replay()` (in module *giving.operators*), 65
`resolve()` (*ptera.opparse.OperatorPrecedenceTower* method), 95
`retry()` (in module *giving.operators*), 66
`rewrite()` (*ptera.overlay.Overlay* method), 97
`rewriting()` (*ptera.overlay.Overlay* method), 97
`roll()` (in module *giving.operators*), 66

S

`sample()` (in module *giving.operators*), 67
`scan()` (in module *giving.operators*), 67
`select()` (in module *ptera.selector*), 109
`Selector` (class in *ptera.selector*), 109
`SelectorError`, 109
`sequence_equal()` (in module *giving.operators*), 68
`set()` (*ptera.interpret.Capture* method), 93
`share()` (in module *giving.operators*), 68
`SimpleVariableCollector` (class in *ptera.transform*), 112
`single()` (in module *giving.operators*), 69
`single_or_default()` (in module *giving.operators*), 69
`single_or_default_async()` (in module *giving.operators*), 70
`skip()` (in module *giving.operators*), 70
`skip_last()` (in module *giving.operators*), 70
`skip_last_with_time()` (in module *giving.operators*), 71
`skip_until()` (in module *giving.operators*), 71
`skip_until_with_time()` (in module *giving.operators*), 71
`skip_while()` (in module *giving.operators*), 72
`skip_while_indexed()` (in module *giving.operators*), 72
`skip_with_time()` (in module *giving.operators*), 73
`slice()` (in module *giving.operators*), 73
`snapshot()` (*ptera.interpret.Capture* method), 93
`sole()` (in module *giving.operators*), 74
`some()` (in module *giving.operators*), 74
`specialize()` (*ptera.selector.Call* method), 109
`specialize()` (*ptera.selector.Element* method), 109
`starmap()` (in module *giving.operators*), 75
`starmap_indexed()` (in module *giving.operators*), 75

start_with() (in module giving.operators), 76
 subscribe() (ptera.probe.Probe method), 102
 subscribe_on() (in module giving.operators), 77
 sum() (in module giving.operators), 77
 switch_latest() (in module giving.operators), 77

T

Tag (class in ptera.tags), 110
 tag() (in module giving.operators), 77
 TagSet (class in ptera.tags), 110
 take() (in module giving.operators), 78
 take_last() (in module giving.operators), 78
 take_last_buffer() (in module giving.operators), 78
 take_last_with_time() (in module giving.operators), 79
 take_until() (in module giving.operators), 79
 take_until_with_time() (in module giving.operators), 80
 take_while() (in module giving.operators), 80
 take_while_indexed() (in module giving.operators), 81
 take_with_time() (in module giving.operators), 81
 tap() (ptera.overlay.Overlay method), 97
 tapping() (ptera.overlay.Overlay method), 97
 throttle() (in module giving.operators), 82
 throttle_first() (in module giving.operators), 82
 throttle_with_mapper() (in module giving.operators), 82
 throttle_with_timeout() (in module giving.operators), 82
 time_interval() (in module giving.operators), 83
 timeout() (in module giving.operators), 83
 timeout_with_mapper() (in module giving.operators), 84
 timestamp() (in module giving.operators), 84
 to_dict() (in module giving.operators), 85
 to_future() (in module giving.operators), 85
 to_iterable() (in module giving.operators), 85
 to_list() (in module giving.operators), 85
 to_marbles() (in module giving.operators), 85
 to_set() (in module giving.operators), 86
 Token (class in ptera.opparse), 95
 tooled() (in module ptera.overlay), 98
 top() (in module giving.operators), 86
 Total (class in ptera.interpret), 94
 transform() (in module ptera.transform), 112
 trigger() (ptera.interpret.WorkingFrame method), 94
 tweak() (ptera.overlay.Overlay method), 97
 tweaking() (ptera.overlay.Overlay method), 97
 type (ptera.opparse.Token attribute), 95
 type (ptera.transform.Key attribute), 111

U

used (ptera.transform.ExternalVariableCollector at-

tribute), 110

V

value (ptera.interpret.Capture property), 93
 value (ptera.opparse.Token attribute), 95
 value (ptera.transform.Key attribute), 111
 values (ptera.interpret.Capture attribute), 92
 values() (ptera.probe.Probe method), 102
 vardoc (ptera.transform.ExternalVariableCollector attribute), 110
 variance() (in module giving.operators), 86
 verify() (in module ptera.selector), 110
 visit_AnnAssign() (ptera.transform.PteraTransformer method), 111
 visit_Assign() (ptera.transform.PteraTransformer method), 111
 visit_Import() (ptera.transform.PteraTransformer method), 111
 visit_ImportFrom() (ptera.transform.PteraTransformer method), 111
 visit_NamedExpr() (ptera.transform.PteraTransformer method), 112

W

where() (in module giving.operators), 86
 where_any() (in module giving.operators), 87
 while_do() (in module giving.operators), 87
 window() (in module giving.operators), 87
 window_toggle() (in module giving.operators), 87
 window_when() (in module giving.operators), 88
 window_with_count() (in module giving.operators), 88
 window_with_time() (in module giving.operators), 89
 window_with_time_or_count() (in module giving.operators), 89
 with_latest_from() (in module giving.operators), 89
 wmap() (in module giving.operators), 89
 work_on() (ptera.interpret.Interactor method), 94
 WorkingFrame (class in ptera.interpret), 94
 wrap() (ptera.probe.Probe method), 103

Z

zip() (in module giving.operators), 90
 zip_with_iterable() (in module giving.operators), 91
 zip_with_list() (in module giving.operators), 91