
RxPY Documentation

Release 3.1.1

Dag Brattli

Jul 16, 2020

CONTENTS

1	Installation	3
2	Rationale	5
3	Get Started	7
3.1	Operators and Chaining	8
3.2	Custom operator	9
3.3	Concurrency	10
4	Migration	15
4.1	Pipe Based Operator Chaining	15
4.2	Removal Of The Result Mapper	16
4.3	Scheduler Parameter In Create Operator	16
4.4	Removal Of List Of Observables	17
4.5	Blocking Observable	17
4.6	Back-Pressure	18
4.7	Time Is In Seconds	18
4.8	Packages Renamed	18
5	Operators	19
5.1	Creating Observables	19
5.2	Transforming Observables	19
5.3	Filtering Observables	20
5.4	Combining Observables	20
5.5	Error Handling	20
5.6	Utility Operators	21
5.7	Conditional and Boolean Operators	21
5.8	Mathematical and Aggregate Operators	21
5.9	Connectable Observable Operators	22
6	Additional Reading	23
6.1	Open Material	23
6.2	Commercial Material	23
7	Reference	25
7.1	Observable Factory	25
7.2	Observable	42
7.3	Subject	45
7.4	Schedulers	47
7.5	Operators	67
7.6	Typing	119

8	Contributing	123
9	The MIT License	125
	Python Module Index	127
	Index	129

RxPY is a library for composing asynchronous and event-based programs using observable collections and pipable query operators in Python.

INSTALLATION

RxPY v3.x runs on [Python 3](#). To install RxPY:

```
pip3 install rx
```

For Python 2.x you need to use version 1.6

```
pip install rx==1.6.1
```


RATIONALE

Reactive Extensions for Python (RxPY) is a set of libraries for composing asynchronous and event-based programs using observable sequences and pipable query operators in Python. Using Rx, developers represent asynchronous data streams with Observables, query asynchronous data streams using operators, and parameterize concurrency in data/event streams using Schedulers.

Using Rx, you can represent multiple asynchronous data streams (that come from diverse sources, e.g., stock quote, Tweets, computer events, web service requests, etc.), and subscribe to the event stream using the Observer object. The Observable notifies the subscribed Observer instance whenever an event occurs. You can put various transformations in-between the source Observable and the consuming Observer as well.

Because Observable sequences are data streams, you can query them using standard query operators implemented as functions that can be chained with the pipe operator. Thus you can filter, map, reduce, compose and perform time-based operations on multiple events easily by using these operators. In addition, there are a number of other reactive stream specific operators that allow powerful queries to be written. Cancellation, exceptions, and synchronization are also handled gracefully by using dedicated operators.

GET STARTED

An *Observable* is the core type in ReactiveX. It serially pushes items, known as *emissions*, through a series of operators until it finally arrives at an Observer, where they are consumed.

Push-based (rather than pull-based) iteration opens up powerful new possibilities to express code and concurrency much more quickly. Because an *Observable* treats events as data and data as events, composing the two together becomes trivial.

There are many ways to create an *Observable* that hands items to an Observer. You can use a `create()` factory and pass it functions that handle items:

- The `on_next` function is called each time the Observable emits an item.
- The `on_completed` function is called when the Observable completes.
- The `on_error` function is called when an error occurs on the Observable.

You do not have to specify all three event types. You can pick and choose which events you want to observe by providing only some of the callbacks, or simply by providing a single lambda for `on_next`. Typically in production, you will want to provide an `on_error` handler so that errors are explicitly handled by the subscriber.

Let's consider the following example:

```
from rx import create

def push_five_strings(observer, scheduler):
    observer.on_next("Alpha")
    observer.on_next("Beta")
    observer.on_next("Gamma")
    observer.on_next("Delta")
    observer.on_next("Epsilon")
    observer.on_completed()

source = create(push_five_strings)

source.subscribe(
    on_next = lambda i: print("Received {}".format(i)),
    on_error = lambda e: print("Error Occurred: {}".format(e)),
    on_completed = lambda: print("Done!"),
)
```

An Observable is created with `create`. On subscription, the `push_five_strings` function is called. This function emits five items. The three callbacks provided to the `subscribe` function simply print the received items and completion states. Note that the use of lambdas simplify the code in this basic example.

Output:

```
Received Alpha
Received Beta
Received Gamma
Received Delta
Received Epsilon
Done!
```

However, there are many *Observable factories* for common sources of emissions. To simply push five items, we can rid the `create()` and its backing function, and use `of()`. This factory accepts an argument list, iterates on each argument to emit them as items, and the completes. Therefore, we can simply pass these five Strings as arguments to it:

```
from rx import of

source = of("Alpha", "Beta", "Gamma", "Delta", "Epsilon")

source.subscribe(
    on_next = lambda i: print("Received {}".format(i)),
    on_error = lambda e: print("Error Occurred: {}".format(e)),
    on_completed = lambda: print("Done!"),
)
```

And a single parameter can be provided to the subscribe function if completion and error are ignored:

```
from rx import of

source = of("Alpha", "Beta", "Gamma", "Delta", "Epsilon")

source.subscribe(lambda value: print("Received {}".format(value)))
```

Output:

```
Received Alpha
Received Beta
Received Gamma
Received Delta
Received Epsilon
```

3.1 Operators and Chaining

You can also derive new Observables using over 130 operators available in RxPY. Each operator will yield a new *Observable* that transforms emissions from the source in some way. For example, we can `map()` each *String* to its length, then `filter()` for lengths being at least 5. These will yield two separate Observables built off each other.

```
from rx import of, operators as op

source = of("Alpha", "Beta", "Gamma", "Delta", "Epsilon")

composed = source.pipe(
    op.map(lambda s: len(s)),
    op.filter(lambda i: i >= 5)
)
composed.subscribe(lambda value: print("Received {}".format(value)))
```

Output:

```
Received 5
Received 5
Received 5
Received 7
```

Typically, you do not want to save Observables into intermediary variables for each operator, unless you want to have multiple subscribers at that point. Instead, you want to strive to inline and create an “Observable pipeline” of operations. That way your code is readable and tells a story much more easily.

```
from rx import of, operators as op

of("Alpha", "Beta", "Gamma", "Delta", "Epsilon").pipe(
    op.map(lambda s: len(s)),
    op.filter(lambda i: i >= 5)
).subscribe(lambda value: print("Received {}".format(value)))
```

3.2 Custom operator

As operators chains grow up, the chains must be split to make the code more readable. New operators are implemented as functions, and can be directly used in the *pipe* operator. When an operator is implemented as a composition of other operators, then the implementation is straightforward, thanks to the *pipe* function:

```
import rx
from rx import operators as ops

def length_more_than_5():
    return rx.pipe(
        ops.map(lambda s: len(s)),
        ops.filter(lambda i: i >= 5),
    )

rx.of("Alpha", "Beta", "Gamma", "Delta", "Epsilon").pipe(
    length_more_than_5()
).subscribe(lambda value: print("Received {}".format(value)))
```

In this example, the *map* and *filter* operators are grouped in a new *length_more_than_5* operator.

It is also possible to create an operator that is not a composition of other operators. This allows to fully control the subscription logic and items emissions:

```
import rx

def lowercase():
    def _lowercase(source):
        def subscribe(observer, scheduler = None):
            def on_next(value):
                observer.on_next(value.lower())

            return source.subscribe(
                on_next,
                observer.on_error,
                observer.on_completed,
                scheduler)

        return rx.create(subscribe)
    return _lowercase
```

(continues on next page)

(continued from previous page)

```
rx.of("Alpha", "Beta", "Gamma", "Delta", "Epsilon").pipe(
    lowercase()
).subscribe(lambda value: print("Received {0}".format(value)))
```

In this example, the *lowercase* operator converts all received items to lowercase. The structure of the *_lowercase* function is a very common way to implement custom operators: It takes a source Observable as input, and returns a custom Observable. The source observable is subscribed only when the output Observable is subscribed. This allows to chain subscription calls when building a pipeline.

Output:

```
Received alpha
Received beta
Received gamma
Received delta
Received epsilon
```

3.3 Concurrency

3.3.1 CPU Concurrency

To achieve concurrency, you use two operators: *subscribe_on()* and *observe_on()*. Both need a *Scheduler* which provides a thread for each subscription to do work (see section on Schedulers below). The *ThreadPoolScheduler* is a good choice to create a pool of reusable worker threads.

Attention: GIL has the potential to undermine your concurrency performance, as it prevents multiple threads from accessing the same line of code simultaneously. Libraries like NumPy can mitigate this for parallel intensive computations as they free the GIL. RxPy may also minimize thread overlap to some degree. Just be sure to test your application with concurrency and ensure there is a performance gain.

The *subscribe_on()* instructs the source *Observable* at the start of the chain which scheduler to use (and it does not matter where you put this operator). The *observe_on()*, however, will switch to a different *Scheduler* at that point in the *Observable* chain, effectively moving an emission from one thread to another. Some *Observable factories* and *operators*, like *interval()* and *delay()*, already have a default *Scheduler* and thus will ignore any *subscribe_on()* you specify (although you can pass a *Scheduler* usually as an argument).

Below, we run three different processes concurrently rather than sequentially using *subscribe_on()* as well as an *observe_on()*.

```
import multiprocessing
import random
import time
from threading import current_thread

import rx
from rx.scheduler import ThreadPoolScheduler
from rx import operators as ops

def intense_calculation(value):
```

(continues on next page)

(continued from previous page)

```

    # sleep for a random short duration between 0.5 to 2.0 seconds to simulate a long-
    ↪running calculation
    time.sleep(random.randint(5, 20) * 0.1)
    return value

# calculate number of CPUs, then create a ThreadPoolScheduler with that number of
↪threads
optimal_thread_count = multiprocessing.cpu_count()
pool_scheduler = ThreadPoolScheduler(optimal_thread_count)

# Create Process 1
rx.of("Alpha", "Beta", "Gamma", "Delta", "Epsilon").pipe(
    ops.map(lambda s: intense_calculation(s)), ops.subscribe_on(pool_scheduler)
).subscribe(
    on_next=lambda s: print("PROCESS 1: {0} {1}".format(current_thread().name, s)),
    on_error=lambda e: print(e),
    on_completed=lambda: print("PROCESS 1 done!"),
)

# Create Process 2
rx.range(1, 10).pipe(
    ops.map(lambda s: intense_calculation(s)), ops.subscribe_on(pool_scheduler)
).subscribe(
    on_next=lambda i: print("PROCESS 2: {0} {1}".format(current_thread().name, i)),
    on_error=lambda e: print(e),
    on_completed=lambda: print("PROCESS 2 done!"),
)

# Create Process 3, which is infinite
rx.interval(1).pipe(
    ops.map(lambda i: i * 100),
    ops.observe_on(pool_scheduler),
    ops.map(lambda s: intense_calculation(s)),
).subscribe(
    on_next=lambda i: print("PROCESS 3: {0} {1}".format(current_thread().name, i)),
    on_error=lambda e: print(e),
)

input("Press any key to exit\n")

```

OUTPUT:

```

Press any key to exit
PROCESS 1: Thread-1 Alpha
PROCESS 2: Thread-2 1
PROCESS 3: Thread-4 0
PROCESS 2: Thread-2 2
PROCESS 1: Thread-1 Beta
PROCESS 3: Thread-7 100
PROCESS 3: Thread-7 200
PROCESS 2: Thread-2 3
PROCESS 1: Thread-1 Gamma
PROCESS 1: Thread-1 Delta
PROCESS 2: Thread-2 4
PROCESS 3: Thread-7 300

```

3.3.2 IO Concurrency

IO concurrency is also supported for several asynchronous frameworks, in combination with associated RxPY schedulers. The following example implements a simple echo TCP server that delays its answers by 5 seconds. It uses AsyncIO as an event loop.

The TCP server is implemented in AsyncIO, and the echo logic is implemented as an RxPY operator chain. Futures allow the operator chain to drive the loop of the coroutine.

```
from collections import namedtuple
import asyncio
import rx
import rx.operators as ops
from rx.subject import Subject
from rx.scheduler.eventloop import AsyncIOScheduler

EchoItem = namedtuple('EchoItem', ['future', 'data'])

def tcp_server(sink, loop):
    def on_subscribe(observer, scheduler):
        async def handle_echo(reader, writer):
            print("new client connected")
            while True:
                data = await reader.readline()
                data = data.decode("utf-8")
                if not data:
                    break

                future = asyncio.Future()
                observer.on_next(EchoItem(
                    future=future,
                    data=data
                ))
                await future
                writer.write(future.result().encode("utf-8"))

            print("Close the client socket")
            writer.close()

        def on_next(i):
            i.future.set_result(i.data)

        print("starting server")
        server = asyncio.start_server(handle_echo, '127.0.0.1', 8888, loop=loop)
        loop.create_task(server)

        sink.subscribe(
            on_next=on_next,
            on_error=observer.on_error,
            on_completed=observer.on_completed)

    return rx.create(on_subscribe)

loop = asyncio.get_event_loop()
proxy = Subject()
source = tcp_server(proxy, loop)
```

(continues on next page)

(continued from previous page)

```
aio_scheduler = AsyncIOScheduler(loop=loop)

source.pipe(
    ops.map(lambda i: i._replace(data="echo: {}".format(i.data))),
    ops.delay(5.0)
).subscribe(proxy, scheduler=aio_scheduler)

loop.run_forever()
print("done")
loop.close()
```

Execute this code from a shell, and connect to it via telnet. Then each line that you type is echoed 5 seconds later.

```
telnet localhost 8888
Connected to localhost.
Escape character is '^]'.
foo
echo: foo
```

If you connect simultaneously from several clients, you can see that requests are correctly served, multiplexed on the AsyncIO event loop.

3.3.3 Default Scheduler

There are several ways to choose a scheduler. The first one is to provide it explicitly to each operator that supports a scheduler. However this can be annoying when a lot of operators are used. So there is a second way to indicate what scheduler will be used as the default scheduler for the whole chain: The scheduler provided in the subscribe call is the default scheduler for all operators in a pipe.

```
source.pipe(
    ...
).subscribe(proxy, scheduler=my_default_scheduler)
```

Operators that accept a scheduler select the scheduler to use in the following way:

- If a scheduler is provided for the operator, then use it.
- If a default scheduler is provided in subscribe, then use it.
- Otherwise use the default scheduler of the operator.

MIGRATION

RxPY v3 is a major evolution from RxPY v1. This release brings many improvements, some of the most important ones being:

- A better integration in IDEs via autocompletion support.
- New operators can be implemented outside of RxPY.
- Operator chains are now built via the *pipe* operator.
- A default scheduler can be provided in an operator chain.

4.1 Pipe Based Operator Chaining

The most fundamental change is the way operators are chained together. On RxPY v1, operators were methods of the *Observable* class. So they were chained by using the existing *Observable* methods:

```
from rx import Observable

Observable.of("Alpha", "Beta", "Gamma", "Delta", "Epsilon") \
    .map(lambda s: len(s)) \
    .filter(lambda i: i >= 5) \
    .subscribe(lambda value: print("Received {0}".format(value)))
```

Chaining in RxPY v3 is based on the *pipe* operator. This operator is now one of the only methods of the *Observable* class. In RxPY v3, operators are implemented as functions:

```
import rx
from rx import operators as ops

rx.of("Alpha", "Beta", "Gamma", "Delta", "Epsilon").pipe(
    ops.map(lambda s: len(s)),
    ops.filter(lambda i: i >= 5)
).subscribe(lambda value: print("Received {0}".format(value)))
```

The fact that operators are functions means that adding new operators is now very easy. Instead of wrapping custom operators with the *let* operator, they can be directly used in a pipe chain.

4.2 Removal Of The Result Mapper

The mapper function is removed in operators that combine the values of several observables. This change applies to the following operators: `combine_latest`, `group_join`, `join`, `with_latest_from`, `zip`, and `zip_with_iterable`.

In RxPY v1, these operators were used the following way:

```
from rx import Observable
import operator

a = Observable.of(1, 2, 3, 4)
b = Observable.of(2, 2, 4, 4)

a.zip(b, lambda a, b: operator.mul(a, b)) \
    .subscribe(print)
```

Now they return an Observable of tuples, with each item being the combination of the source Observables:

```
import rx
from rx import operators as ops
import operator

a = rx.of(1, 2, 3, 4)
b = rx.of(2, 2, 4, 4)

a.pipe(
    ops.zip(b), # returns a tuple with the items of a and b
    ops.map(lambda z: operator.mul(z[0], z[1]))
).subscribe(print)
```

Dealing with the tuple unpacking is made easier with the `starmap` operator that unpacks the tuple to args:

```
import rx
from rx import operators as ops
import operator

a = rx.of(1, 2, 3, 4)
b = rx.of(2, 2, 4, 4)

a.pipe(
    ops.zip(b),
    ops.starmap(operator.mul)
).subscribe(print)
```

4.3 Scheduler Parameter In Create Operator

The subscription function provided to the `create` operator now takes two parameters: An observer and a scheduler. The scheduler parameter is new: If a scheduler has been set in the call to subscribe, then this scheduler is passed to the subscription function. Otherwise this parameter is set to `None`.

One can use or ignore this parameter. This new scheduler parameter allows the create operator to use the default scheduler provided in the subscribe call. So scheduling item emissions with relative or absolute due-time is now possible.

4.4 Removal Of List Of Observables

The support of list of Observables as a parameter has been removed in the following operators: *merge*, *zip*, and *combine_latest*. For example in RxPY v1 the *merge* operator could be called with a list:

```
from rx import Observable

obs1 = Observable.from_([1, 2, 3, 4])
obs2 = Observable.from_([5, 6, 7, 8])

res = Observable.merge([obs1, obs2])
res.subscribe(print)
```

This is not possible anymore in RxPY v3. So Observables must be provided explicitly:

```
import rx, operator as op

obs1 = rx.from_([1, 2, 3, 4])
obs2 = rx.from_([5, 6, 7, 8])

res = rx.merge(obs1, obs2)
res.subscribe(print)
```

If for any reason the Observables are only available as a list, then they can be unpacked:

```
import rx
from rx import operators as ops

obs1 = rx.from_([1, 2, 3, 4])
obs2 = rx.from_([5, 6, 7, 8])

obs_list = [obs1, obs2]

res = rx.merge(*obs_list)
res.subscribe(print)
```

4.5 Blocking Observable

BlockingObservables have been removed from RxPY v3. In RxPY v1, blocking until an Observable completes was done the following way:

```
from rx import Observable

res = Observable.from_([1, 2, 3, 4]).to_blocking().last()
print(res)
```

This is now done with the *run* operator:

```
import rx

res = rx.from_([1, 2, 3, 4]).run()
print(res)
```

The *run* operator returns only the last value emitted by the source Observable. It is possible to use the previous blocking operators by using the standard operators before *run*. For example:

- Get first item: `obs.pipe(ops.first()).run()`
- Get all items: `obs.pipe(ops.to_list()).run()`

4.6 Back-Pressure

Support for back-pressure - and so `ControllableObservable` - has been removed in RxPY v3. Back-pressure can be implemented in several ways, and many strategies can be adopted. So we consider that such features are beyond the scope of RxPY. You are encouraged to provide independent implementations as separate packages so that they can be shared by the community.

List of community projects supporting backpressure can be found in [Additional Reading](#).

4.7 Time Is In Seconds

Operators that take time values as parameters now use seconds as a unit instead of milliseconds. This RxPY v1 example:

```
ops.debounce(500)
```

is now written as:

```
ops.debounce(0.5)
```

4.8 Packages Renamed

Some packages were renamed:

Old name	New name
<i>rx.concurrency</i>	<i>rx.scheduler</i>
<i>rx.disposables</i>	<i>rx.disposable</i>
<i>rx.subjects</i>	<i>rx.subject</i>

Furthermore, the package formerly known as *rx.concurrency.mainloopscheduler* has been split into two parts, *rx.scheduler.mainloop* and *rx.scheduler.eventloop*.

OPERATORS

5.1 Creating Observables

Operator	Description
<i>create</i>	Create an Observable from scratch by calling observer methods programmatically.
<i>empty</i>	Creates an Observable that emits no item and completes immediately.
<i>never</i>	Creates an Observable that never completes.
<i>throw</i>	Creates an Observable that terminates with an error.
<i>from_</i>	Convert some other object or data structure into an Observable.
<i>interval</i>	Create an Observable that emits a sequence of integers spaced by a particular time interval.
<i>just</i>	Convert an object or a set of objects into an Observable that emits that object or those objects.
<i>range</i>	Create an Observable that emits a range of sequential integers.
<i>repeat_value</i>	Create an Observable that emits a particular item or sequence of items repeatedly.
<i>start</i>	Create an Observable that emits the return value of a function.
<i>timer</i>	Create an Observable that emits a single item after a given delay.

5.2 Transforming Observables

Operator	Description
<i>buffer</i>	Periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time.
<i>flat_map</i>	Transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable.
<i>group_by</i>	Divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key.
<i>map</i>	Transform the items emitted by an Observable by applying a function to each item.
<i>scan</i>	Apply a function to each item emitted by an Observable, sequentially, and emit each successive value.
<i>window</i>	Periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time.

5.3 Filtering Observables

Operator	Description
<i>debounce</i>	Only emit an item from an Observable if a particular timespan has passed without it emitting another item.
<i>distinct</i>	Suppress duplicate items emitted by an Observable.
<i>element_at</i>	Emit only item n emitted by an Observable.
<i>filter</i>	Emit only those items from an Observable that pass a predicate test.
<i>first</i>	Emit only the first item, or the first item that meets a condition, from an Observable.
<i>ignore_elements</i>	Do not emit any items from an Observable but mirror its termination notification.
<i>last</i>	Emit only the last item emitted by an Observable.
<i>sample</i>	Emit the most recent item emitted by an Observable within periodic time intervals.
<i>skip</i>	Suppress the first n items emitted by an Observable.
<i>skip_last</i>	Suppress the last n items emitted by an Observable.
<i>take</i>	Emit only the first n items emitted by an Observable.
<i>take_last</i>	Emit only the last n items emitted by an Observable.

5.4 Combining Observables

Operator	Description
<i>combine_latest</i>	When an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function.
<i>join</i>	Combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable.
<i>merge</i>	Combine multiple Observables into one by merging their emissions.
<i>start_with</i>	Emit a specified sequence of items before beginning to emit the items from the source Observable.
<i>switch_latest</i>	Convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables.
<i>zip</i>	Combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function.

5.5 Error Handling

Operator	Description
<i>catch</i>	Continues observable sequences which are terminated with an exception by switching over to the next observable sequence.
<i>retry</i>	If a source Observable sends an onError notification, resubscribe to it in the hopes that it will complete without error.

5.6 Utility Operators

Operator	Description
<i>delay</i>	Shift the emissions from an Observable forward in time by a particular amount.
<i>do</i>	Register an action to take upon a variety of Observable lifecycle events.
<i>materialize</i>	Materializes the implicit notifications of an observable sequence as explicit notification values.
<i>dematerialize</i>	Dematerializes the explicit notification values of an observable sequence as implicit notifications.
<i>observe_on</i>	Specify the scheduler on which an observer will observe this Observable.
<i>subscribe</i>	Operate upon the emissions and notifications from an Observable.
<i>subscribe_on</i>	Specify the scheduler an Observable should use when it is subscribed to.
<i>time_interval</i>	Convert an Observable that emits items into one that emits indications of the amount of time elapsed between those emissions.
<i>timeout</i>	Mirror the source Observable, but issue an error notification if a particular period of time elapses without any emitted items.
<i>timestamp</i>	Attach a timestamp to each item emitted by an Observable.

5.7 Conditional and Boolean Operators

Operator	Description
<i>all</i>	Determine whether all items emitted by an Observable meet some criteria.
<i>amb</i>	Given two or more source Observables, emit all of the items from only the first of these Observables to emit an item.
<i>contains</i>	Determine whether an Observable emits a particular item or not.
<i>default_if_empty</i>	Emit items from the source Observable, or a default item if the source Observable emits nothing.
<i>sequence_equal</i>	Determine whether two Observables emit the same sequence of items.
<i>skip_until</i>	Discard items emitted by an Observable until a second Observable emits an item.
<i>skip_while</i>	Discard items emitted by an Observable until a specified condition becomes false.
<i>take_until</i>	Discard items emitted by an Observable after a second Observable emits an item or terminates.
<i>take_while</i>	Discard items emitted by an Observable after a specified condition becomes false.

5.8 Mathematical and Aggregate Operators

Operator	Description
<i>average</i>	Calculates the average of numbers emitted by an Observable and emits this average.
<i>concat</i>	Emit the emissions from two or more Observables without interleaving them.
<i>count</i>	Count the number of items emitted by the source Observable and emit only this value.
<i>max</i>	Determine, and emit, the maximum-valued item emitted by an Observable.
<i>min</i>	Determine, and emit, the minimum-valued item emitted by an Observable.
<i>reduce</i>	Apply a function to each item emitted by an Observable, sequentially, and emit the final value.
<i>sum</i>	Calculate the sum of numbers emitted by an Observable and emit this sum.

5.9 Connectable Observable Operators

Operator	Description
<code>connect</code>	Instruct a connectable Observable to begin emitting items to its subscribers.
<code>publish</code>	Convert an ordinary Observable into a connectable Observable.
<code>ref_count</code>	Make a Connectable Observable behave like an ordinary Observable.
<code>replay</code>	Ensure that all observers see the same sequence of emitted items, even if they subscribe after the Observable has begun emitting items.

ADDITIONAL READING

6.1 Open Material

The RxPY source repository contains [example notebooks](#).

The official ReactiveX website contains additional tutorials and documentation:

- [Introduction](#)
- [Tutorials](#)
- [Operators](#)

Several commercial contents have their associated example code available freely:

- [Packt Reactive Programming in Python](#)

RxPY 3.0.0 has removed support for backpressure here are the known community projects supporting backpressure:

- [rxbackpressure rxpy extension](#)
- [rxpy_backpressure observer decorators](#)

6.2 Commercial Material

O'Reilly Video

O'Reilly has published the video *Reactive Python for Data Science* which is available on both the [O'Reilly Store](#) as well as [O'Reilly Safari](#). This video teaches RxPY from scratch with applications towards data science, but should be helpful for anyone seeking to learn RxPY and reactive programming.

Packt Video

Packt has published the video *Reactive Programming in Python*, available on [Packt store](#). This video teaches how to write reactive GUI and network applications.

REFERENCE

7.1 Observable Factory

`rx.amb(*sources)`

Propagates the observable sequence that emits first. f87bbff3e9ad64d121cc9d2ab3d1aacd4d9015c8.png



marble\sphinxhyphen {}f87bbff3e9ad64d121cc9d2ab3d1aacd4d9015c8.png

Example

```
>>> winner = rx.amb(xs, ys, zs)
```

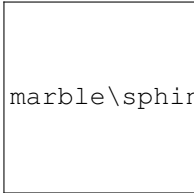
Parameters `sources` (*Observable*) – Sequence of observables to monitor for first emission.

Return type *Observable*

Returns An observable sequence that surfaces any of the given sequences, whichever emitted the first element.

`rx.case (mapper, sources, default_source=None)`

Uses mapper to determine which source in sources to use. 27349f4c42c4d91779a343361676839f6e081bc4.png



marble\sphinxhyphen {}27349f4c42c4d91779a343361676839f6e081bc4.png

Examples

```
>>> res = rx.case(mapper, { '1': obs1, '2': obs2 })
>>> res = rx.case(mapper, { '1': obs1, '2': obs2 }, obs0)
```

Parameters

- **mapper** (Callable[[], Any]) – The function which extracts the value for to test in a case statement.
- **sources** (Mapping) – An object which has keys which correspond to the case statement labels.
- **default_source** (Union[Observable, Future, None]) – [Optional] The observable sequence or Future that will be run if the sources are not matched. If this is not provided, it defaults to *empty()*.

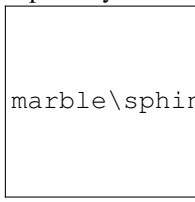
Return type *Observable*

Returns An observable sequence which is determined by a case statement.

rx.catch (*sources)

Continues observable sequences which are terminated with an exception by switching over to the next observable

sequence. d14c02dddb2af945932be6a8f5b44425cde47a95.png



marble\sphinxhyphen {}d14c02dddb2af945932be

Examples

```
>>> res = rx.catch(xs, ys, zs)
```

Parameters **sources** (*Observable*) – Sequence of observables.

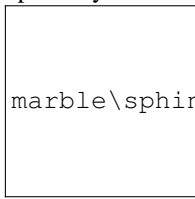
Return type *Observable*

Returns An observable sequence containing elements from consecutive observables from the sequence of sources until one of them terminates successfully.

rx.catch_with_iterable (sources)

Continues observable sequences that are terminated with an exception by switching over to the next observable

sequence. d14c02dddb2af945932be6a8f5b44425cde47a95.png



marble\sphinxhyphen {}d14c02dddb2af945932be

Examples

```
>>> res = rx.catch([xs, ys, zs])
>>> res = rx.catch(src for src in [xs, ys, zs])
```


Parameters **sources** (`Iterable[Observable]`) – An Iterable of observables; thus, a generator can also be used here.

Return type `Observable`

Returns An observable sequence containing elements from consecutive observables from the sequence of sources until one of them terminates successfully.

`rx.create(subscribe)`

Creates an observable sequence object from the specified subscription function.



aa0779a8a931350806ec603de7e4d48a00ab25d9.png


Parameters **subscribe** (`Callable[[Observer, Optional[Scheduler]], Disposable]`) – Subscription function.

Return type `Observable`

Returns An observable sequence that can be subscribed to via the given subscription function.

`rx.combine_latest(*sources)`

Merges the specified observable sequences into one observable sequence by creating a tuple whenever any of the



observable sequences emits an element. 86e55845149aaa02237e62b0ba0b70d2fbaf984f.png

Examples

```
>>> obs = rx.combine_latest(obs1, obs2, obs3)
```

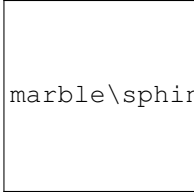
Parameters **sources** (`Observable`) – Sequence of observables.

Return type `Observable`

Returns An observable sequence containing the result of combining elements from each source in given sequence.

`rx.concat(*sources)`

Concatenates all of the specified observable sequences. 4544dd242d02df82ac059a82701a17b5b31135d4.png



marble\sphinxhyphen {}4544dd242d02df82ac059a82701a17b5b31135d4.png

Examples

```
>>> res = rx.concat(xs, ys, zs)
```

Parameters **sources** (*Observable*) – Sequence of observables.

Return type *Observable*

Returns An observable sequence that contains the elements of each source in the given sequence, in sequential order.

rx.concat_with_iterable (*sources*)

Concatenates all of the specified observable sequences. 4544dd242d02df82ac059a82701a17b5b31135d4.png



marble\sphinxhyphen {}4544dd242d02df82ac059a82701a17b5b31135d4.png

Examples

```
>>> res = rx.concat_with_iterable([xs, ys, zs])
>>> res = rx.concat_with_iterable(for src in [xs, ys, zs])
```

Parameters **sources** (*Iterable[Observable]*) – An Iterable of observables; thus, a generator can also be used here.


Return type *Observable*

Returns An observable sequence that contains the elements of each given sequence, in sequential order.

rx.defer (*factory*)

Returns an observable sequence that invokes the specified factory function whenever a new observer subscribes.

e9205b73ee187ab559557e85051afd341c421477.png



marble\sphinxhyphen {}e9205b73ee187ab559557e85051afd341c421477.png

Example


```
>>> res = rx.defer(lambda: of(1, 2, 3))
```

Parameters **factory** (Callable[[*Scheduler*], Union[*Observable*, Future]]) – Observable factory function to invoke for each observer which invokes *subscribe()* on the resulting sequence.

Return type *Observable*

Returns An observable sequence whose observers trigger an invocation of the given factory function.

`rx.empty(scheduler=None)`

Returns an empty observable sequence.  `b1ce6593ad9d9760c7d62a8f13c6cf965bef6a3e.png`

Example

```
>>> obs = rx.empty()
```

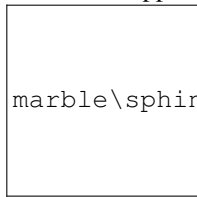
Parameters **scheduler** (Optional[*Scheduler*]) – [Optional] Scheduler instance to send the termination call on. By default, this will use an instance of *ImmediateScheduler*.

Return type *Observable*

Returns An observable sequence with no elements.

`rx.for_in(values, mapper)`

Concatenates the observable sequences obtained by running the specified result mapper for each element in the

specified values.  `2a208bb2fc1872ac5daa08dbdf2da490bd5bdab1.png`

Note: This is just a wrapper for `rx.concat(map(mapper, values))`

Parameters

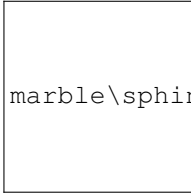
- **values** (Iterable[Any]) – An Iterable of values to turn into an observable source.
- **mapper** (Callable[[~T1], ~T2]) – A function to apply to each item in the values list to turn it into an observable sequence; this should return instances of *rx.Observable*.

Return type *Observable*

Returns An observable sequence from the concatenated observable sequences.

`rx.from_callable (supplier, scheduler=None)`

Returns an observable sequence that contains a single element generated by the given supplier, using the specified scheduler to send out observer messages. fb3e83a887145385d94d0bf013686975fe888f0d.png



marble\sphinxhyphen {}fb3e83a887145385d94d0bf013686975fe888f0d.png

Examples

```
>>> res = rx.from_callable(lambda: calculate_value())
>>> res = rx.from_callable(lambda: 1 / 0) # emits an error
```

Parameters

- **supplier** (Callable[[], Any]) – Function which is invoked to obtain the single element.
- **scheduler** (Optional[Scheduler]) – [Optional] Scheduler instance to schedule the values on. If not specified, the default is to use an instance of *CurrentThreadScheduler*.

Return type *Observable*

Returns An observable sequence containing the single element obtained by invoking the given supplier function.

`rx.from_callback (func, mapper=None)`

Converts a callback function to an observable sequence.

Parameters

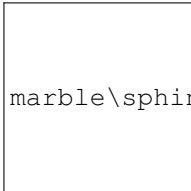
- **func** (Callable) – Function with a callback as the last argument to convert to an Observable sequence.
- **mapper** (Optional[Callable[[~T1], ~T2]]) – [Optional] A mapper which takes the arguments from the callback to produce a single item to yield on next.

Return type Callable[[], *Observable*]

Returns A function, when executed with the required arguments minus the callback, produces an Observable sequence with a single value of the arguments to the callback as a list.

`rx.from_future (future)`

Converts a Future to an Observable sequence b4a27b6a0e04aee0b97edc9b0f56546d34a0f3c3.png



marble\sphinxhyphen {}b4a27b6a0e04aee0b97edc9b0f56546d34a0f3c3.png

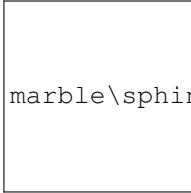
Parameters **future** (Future) – A Python 3 compatible future. <https://docs.python.org/3/library/asyncio-task.html#future> <http://www.tornadoweb.org/en/stable/concurrent.html#tornado.concurrent.Future>

Return type *Observable*

Returns An observable sequence which wraps the existing future success and failure.

`rx.from_iterable(iterable, scheduler=None)`

Converts an iterable to an observable sequence. 52acee3ae4a95d2cb92d12d732656a72119e0e8f.png



marble\sphinxhyphen {}52acee3ae4a95d2cb92d12d732656a72119e0e8f.png

Example

```
>>> rx.from_iterable([1,2,3])
```

Parameters

- **iterable** (Iterable) – An Iterable to change into an observable sequence.
- **scheduler** (Optional[*Scheduler*]) – [Optional] Scheduler instance to schedule the values on. If not specified, the default is to use an instance of *CurrentThreadScheduler*.

Return type *Observable*

Returns The observable sequence whose elements are pulled from the given iterable sequence.

`rx.from_(iterable, scheduler=None)`

Alias for `rx.from_iterable()`.

Return type *Observable*

`rx.from_list(iterable, scheduler=None)`

Alias for `rx.from_iterable()`.

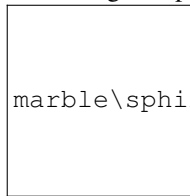
Return type *Observable*

`rx.from_marbles(string, timespan=0.1, scheduler=None, lookup=None, error=None)`

Convert a marble diagram string to a cold observable sequence, using an optional scheduler to enumerate the

marble\sphinxhyphen {}429e4bc8ef87062595ac19ce

events. 429e4bc8ef87062595ac19ce42c48d9420ddff90.png



Each character in the string will advance time by timespan (except for space). Characters that are not special (see the table below) will be interpreted as a value to be emitted. Numbers will be cast to int or float.

Special characters:

-	advance time by timespan
#	on_error()
	on_completed()
(open a group of marbles sharing the same timestamp
)	close a group of marbles
,	separate elements in a group
<space>	used to align multiple diagrams, does not advance time

In a group of elements, the position of the initial (determines the timestamp at which grouped elements will be emitted. E.g. -- (12, 3, 4) -- will emit 12, 3, 4 at 2 * timespan and then advance virtual time by 8 * timespan.

Examples

```
>>> from_marbles('--1--(2,3)-4--|')
>>> from_marbles('a--b--c-', lookup={'a': 1, 'b': 2, 'c': 3})
>>> from_marbles('a--b---#', error=ValueError('foo'))
```

Parameters

- **string** (str) – String with marble diagram
- **timespan** (Union[timedelta, float]) – [Optional] Duration of each character in seconds. If not specified, defaults to 0.1.
- **scheduler** (Optional[Scheduler]) – [Optional] Scheduler to run the the input sequence on. If not specified, defaults to the subscribe scheduler if defined, else to an instance of `NewThreadScheduler` `<rx.scheduler.NewThreadScheduler>`.
- **lookup** (Optional[Mapping]) – [Optional] A dict used to convert an element into a specified value. If not specified, defaults to {}.
- **error** (Optional[Exception]) – [Optional] Exception that will be use in place of the # symbol. If not specified, defaults to `Exception('error')`.

Return type *Observable*

Returns The observable sequence whose elements are pulled from the given marble diagram string.

`rx.cold(string, timespan=0.1, scheduler=None, lookup=None, error=None)`

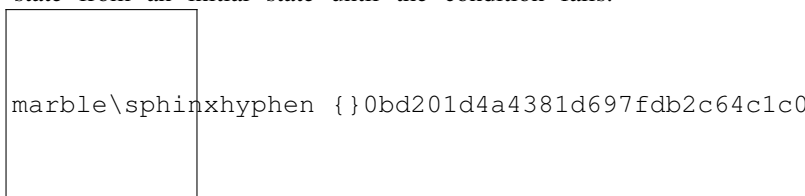
Alias for `rx.from_marbles()`.

Return type *Observable*

`rx.generate_with_relative_time(initial_state, condition, iterate, time_mapper)`

Generates an observable sequence by iterating a state from an initial state until the condition fails.

0bd201d4a4381d697fdb2c64c1c0b4494bbd8950.png



Example

```
>>> res = rx.generate_with_relative_time(0, lambda x: True, lambda x: x + 1,
↳ lambda x: 0.5)
```

Parameters

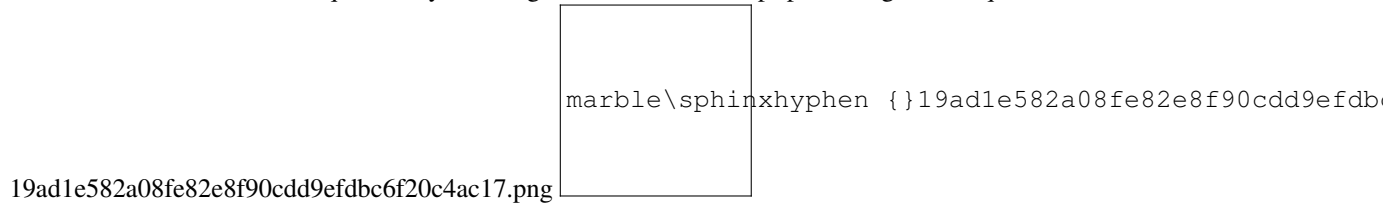
- **initial_state** (Any) – Initial state.
- **condition** (Callable[[~T1], bool]) – Condition to terminate generation (upon returning False).
- **iterate** (Callable[[~T1], ~T2]) – Iteration step function.
- **time_mapper** (Callable[[Any], Union[timedelta, float]]) – Time mapper function to control the speed of values being produced each iteration, returning relative times, i.e. either a float denoting seconds, or an instance of timedelta.

Return type *Observable*

Returns The generated sequence.

rx.generate (*initial_state*, *condition*, *iterate*)

Generates an observable sequence by running a state-driven loop producing the sequence's elements.



Example

```
>>> res = rx.generate(0, lambda x: x < 10, lambda x: x + 1)
```

Parameters

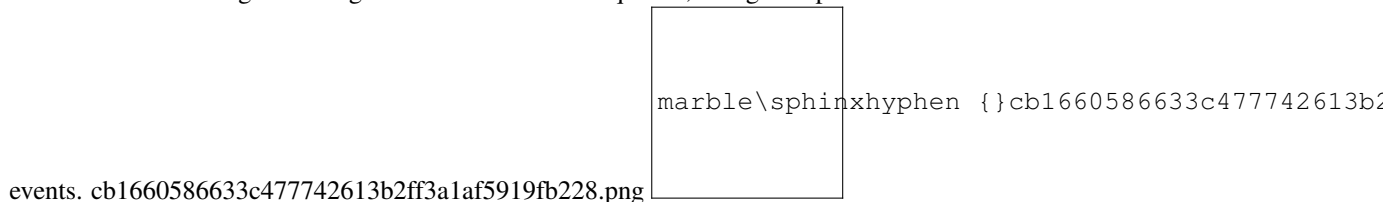
- **initial_state** (Any) – Initial state.
- **condition** (Callable[[~T1], bool]) – Condition to terminate generation (upon returning False).
- **iterate** (Callable[[~T1], ~T2]) – Iteration step function.

Return type *Observable*

Returns The generated sequence.

rx.hot (*string*, *timespan=0.1*, *duetime=0.0*, *scheduler=None*, *lookup=None*, *error=None*)

Convert a marble diagram string to a hot observable sequence, using an optional scheduler to enumerate the



Each character in the string will advance time by timespan (except for space). Characters that are not special (see the table below) will be interpreted as a value to be emitted. Numbers will be cast to int or float.

Special characters:

-	advance time by timespan
#	on_error()
	on_completed()
(open a group of elements sharing the same timestamp
)	close a group of elements
,	separate elements in a group
<space>	used to align multiple diagrams, does not advance time

In a group of elements, the position of the initial (determines the timestamp at which grouped elements will be emitted. E.g. -- (12, 3, 4) -- will emit 12, 3, 4 at 2 * timespan and then advance virtual time by 8 * timespan.

Examples

```
>>> hot("--1--(2,3)-4--|")
>>> hot("a--b--c-", lookup={'a': 1, 'b': 2, 'c': 3})
>>> hot("a--b---#", error=ValueError("foo"))
```

Parameters

- **string** (str) – String with marble diagram
- **timespan** (Union[timedelta, float]) – [Optional] Duration of each character in seconds. If not specified, defaults to 0.1.
- **duetime** (Union[datetime, timedelta, float]) – [Optional] Absolute datetime or timedelta from now that determines when to start the emission of elements.
- **scheduler** (Optional[Scheduler]) – [Optional] Scheduler to run the the input sequence on. If not specified, defaults to an instance of *NewThreadScheduler*.
- **lookup** (Optional[Mapping]) – [Optional] A dict used to convert an element into a specified value. If not specified, defaults to {}.
- **error** (Optional[Exception]) – [Optional] Exception that will be use in place of the # symbol. If not specified, defaults to `Exception('error')`.

Return type *Observable*

Returns The observable sequence whose elements are pulled from the given marble diagram string.

rx.if_then (condition, then_source, else_source=None)

Determines whether an observable collection contains values. b0833400c0d786a6f24a4a878882efe0852f5614.png



Examples

```
>>> res = rx.if_then(condition, obs1)
>>> res = rx.if_then(condition, obs1, obs2)
```

Parameters

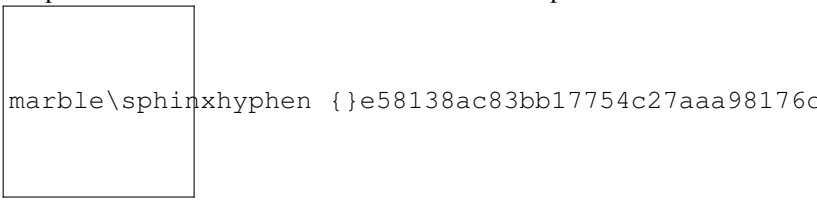
- **condition** (Callable[[], bool]) – The condition which determines if the then_source or else_source will be run.
- **then_source** (Union[Observable, Future]) – The observable sequence or Future that will be run if the condition function returns True.
- **else_source** (Union[Observable, Future, None]) – [Optional] The observable sequence or Future that will be run if the condition function returns False. If this is not provided, it defaults to `empty()`.

Return type *Observable*

Returns An observable sequence which is either the then_source or else_source.

rx.interval (*period*, *scheduler=None*)

Returns an observable sequence that produces a value after each period.



The diagram shows a horizontal timeline with a series of vertical lines representing discrete time intervals. Below each vertical line is a small circle containing a value. The values are represented by the placeholder text 'marble\sphinxhyphen {}e58138ac83bb17754c27aaa98176c' and 'e58138ac83bb17754c27aaa98176c76fe78bfca7.png'.

Example

```
>>> res = rx.interval(1.0)
```

Parameters

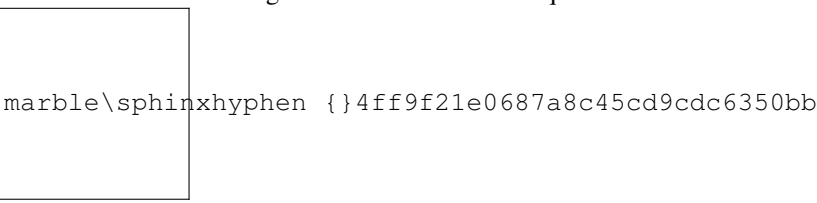
- **period** (Union[timedelta, float]) – Period for producing the values in the resulting sequence (specified as a float denoting seconds or an instance of timedelta).
- **scheduler** (Optional[Scheduler]) – Scheduler to run the interval on. If not specified, an instance of *TimeoutScheduler* is used.

Return type *Observable*

Returns An observable sequence that produces a value after each period.

rx.merge (**sources*)

Merges all the observable sequences into a single observable sequence.



The diagram shows a horizontal timeline with multiple vertical lines representing different observable sequences. These sequences merge into a single horizontal line, which then produces a series of values represented by the placeholder text 'marble\sphinxhyphen {}4ff9f21e0687a8c45cd9cdc6350bb' and '4ff9f21e0687a8c45cd9cdc6350bbe584661a43f.png'.

Example

```
>>> res = rx.merge(obs1, obs2, obs3)
```

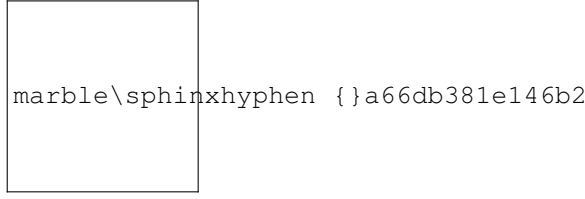
Parameters **sources** (*Observable*) – Sequence of observables.

Return type *Observable*

Returns The observable sequence that merges the elements of the observable sequences.

`rx.never()`

Returns a non-terminating observable sequence, which can be used to denote an infinite duration (e.g. when

using reactive joins). 

Return type *Observable*

Returns An observable sequence whose observers will never get called.

`rx.of(*args)`

This method creates a new observable sequence whose elements are taken from the arguments.



Note: This is just a wrapper for `rx.from_iterable(args)`

Example

```
>>> res = rx.of(1,2,3)
```

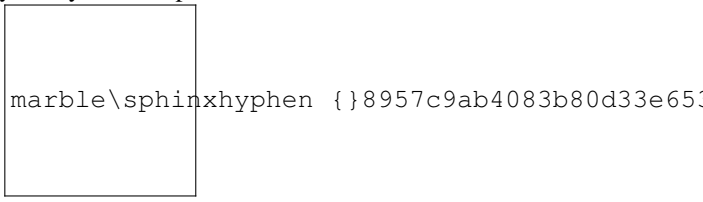
Parameters **args** (Any) – The variable number elements to emit from the observable.

Return type *Observable*

Returns The observable sequence whose elements are pulled from the given arguments

`rx.on_error_resume_next(*sources)`

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

sequence. 

Examples

```
>>> res = rx.on_error_resume_next(xs, ys, zs)
```

Parameters **sources** (Union[*Observable*, Future]) – Sequence of sources, each of which is expected to be an instance of either *Observable* or Future.

Return type *Observable*

Returns An observable sequence that concatenates the source sequences, even if a sequence terminates with an exception.

rx.range (*start*, *stop=None*, *step=None*, *scheduler=None*)

Generates an observable sequence of integral numbers within a specified range, using the specified scheduler to

marble\sphinxhyphen {}8ee4d4a3f5

send out observer messages. 8ee4d4a3f526f44dd889e96a4df0e5849b0e6a4f.png

Examples

```
>>> res = rx.range(10)
>>> res = rx.range(0, 10)
>>> res = rx.range(0, 10, 1)
```

Parameters

- **start** (int) – The value of the first integer in the sequence.
- **count** – The number of sequential integers to generate.
- **scheduler** (Optional[*Scheduler*]) – [Optional] The scheduler to schedule the values on. If not specified, the default is to use an instance of *CurrentThreadScheduler*.

Return type *Observable*

Returns An observable sequence that contains a range of sequential integral numbers.

rx.return_value (*value*, *scheduler=None*)

Returns an observable sequence that contains a single element, using the specified scheduler to send out observer

marble\sphinxhyphen {}cb

messages. There is an alias called 'just'. cbf658341a663218557b4c18fc5a96f46d58f18d.png

Examples

```
>>> res = rx.return_value(42)
>>> res = rx.return_value(42, timeout_scheduler)
```

Parameters **value** (Any) – Single element in the resulting observable sequence.

Return type *Observable*

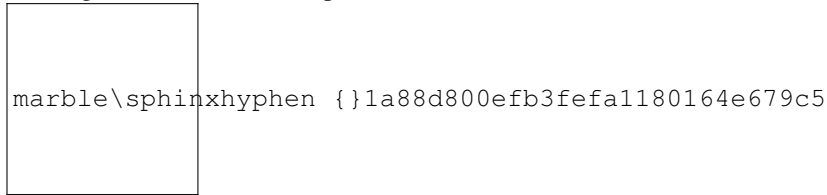
Returns An observable sequence containing the single specified element.

rx.just (*value*, *scheduler=None*)
Alias for *rx.return_value()*.

Return type *Observable*

rx.repeat_value (*value=None*, *repeat_count=None*)
Generates an observable sequence that repeats the given element the specified number of times.

1a88d800efb3fefa1180164e679c53cbadbceaa6.png



Examples

```
>>> res = rx.repeat_value(42)
>>> res = rx.repeat_value(42, 4)
```

Parameters

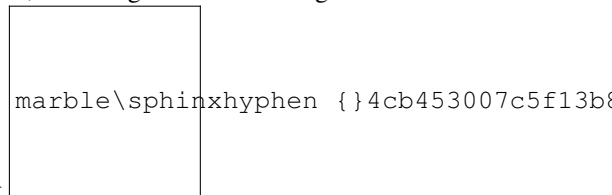
- **value** (Optional[Any]) – Element to repeat.
- **repeat_count** (Optional[int]) – [Optional] Number of times to repeat the element. If not specified, repeats indefinitely.

Return type *Observable*

Returns An observable sequence that repeats the given element the specified number of times.

rx.start (*func*, *scheduler=None*)
Invokes the specified function asynchronously on the specified scheduler, surfacing the result through an ob-

servable sequence. 4cb453007c5f13b8a0d39023bcc8c316c91f811e.png



Note: The function is called immediately, not during the subscription of the resulting sequence. Multiple subscriptions to the resulting sequence can observe the function's result.

Example

```
>>> res = rx.start(lambda: pprint('hello'))
>>> res = rx.start(lambda: pprint('hello'), rx.Scheduler.timeout)
```

Parameters

- **func** (Callable) – Function to run asynchronously.
- **scheduler** (Optional[Scheduler]) – [Optional] Scheduler to run the function on. If not specified, defaults to an instance of *TimeoutScheduler*.

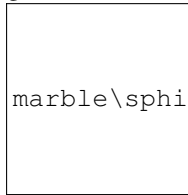
Return type *Observable*

Returns An observable sequence exposing the function's result value, or an exception.

rx.start_async (function_async)

Invokes the asynchronous function, surfacing the result through an observable sequence.

4f154949ac72b6ead68198137611e8c13c2e7c51.png



marble\sphinxhyphen {}4f154949ac72b6ead68198137611

Parameters **function_async** (Callable[[], Future]) – Asynchronous function which returns a Future to run.

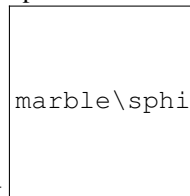
Return type *Observable*

Returns An observable sequence exposing the function's result value, or an exception.

rx.throw (exception, scheduler=None)

Returns an observable sequence that terminates with an exception, using the specified scheduler to send out the

single OnError message. 7ff5aaf3f68e18ba1b037f68e0fcc746936e6c5c.png



marble\sphinxhyphen {}7ff5aaf3f68e1

Example

```
>>> res = rx.throw(Exception('Error'))
```

Parameters

- **exception** (Exception) – An object used for the sequence's termination.
- **scheduler** (Optional[Scheduler]) – [Optional] Scheduler to schedule the error notification on. If not specified, the default is to use an instance of *ImmediateScheduler*.

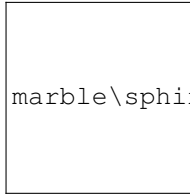
Return type *Observable*

Returns The observable sequence that terminates exceptionally with the specified exception object.

`rx.timer` (*duetime*, *period=None*, *scheduler=None*)

Returns an observable sequence that produces a value after *duetime* has elapsed and then after each period.

ee4501f9edf6effa1fbd83ea26f2d1045ec4f303.png



marble\sphinxhyphen-{}ee4501f9edf6effa1fbd83ea26f2d1

Examples

```
>>> res = rx.timer(datetime(...))
>>> res = rx.timer(datetime(...), 0.1)
>>> res = rx.timer(5.0)
>>> res = rx.timer(5.0, 1.0)
```

Parameters

- **duetime** (Union[datetime, timedelta, float]) – Absolute (specified as a date-time object) or relative time (specified as a float denoting seconds or an instance of timedelta) at which to produce the first value.
- **period** (Union[timedelta, float, None]) – [Optional] Period to produce subsequent values (specified as a float denoting seconds or an instance of timedelta). If not specified, the resulting timer is not recurring.
- **scheduler** (Optional[Scheduler]) – [Optional] Scheduler to run the timer on. If not specified, the default is to use an instance of *TimeoutScheduler*.

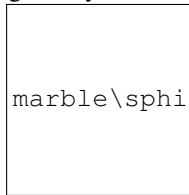
Return type *Observable*

Returns An observable sequence that produces a value after due time has elapsed and then each period.

`rx.to_async` (*func*, *scheduler=None*)

Converts the function into an asynchronous function. Each invocation of the resulting asynchronous function causes an invocation of the original synchronous function on the specified scheduler.

6d1bacb339a4300a6c3a617e57ada62f83e5a667.png



marble\sphinxhyphen-{}6d1bacb339a4300a6c3a617e57ada

Examples

```
>>> res = rx.to_async(lambda x, y: x + y)(4, 3)
>>> res = rx.to_async(lambda x, y: x + y, Scheduler.timeout)(4, 3)
>>> res = rx.to_async(lambda x: log.debug(x), Scheduler.timeout)('hello')
```

Parameters

- **func** (Callable) – Function to convert to an asynchronous function.

- **scheduler** (Optional[*Scheduler*]) – [Optional] Scheduler to run the function on. If not specified, defaults to an instance of *TimeoutScheduler*.

Return type *Callable*

Returns Asynchronous function.

`rx.using(resource_factory, observable_factory)`

Constructs an observable sequence that depends on a resource object, whose lifetime is tied to the resulting observable sequence's lifetime.

Example

```
>>> res = rx.using(lambda: AsyncSubject(), lambda: s: s)
```

Parameters

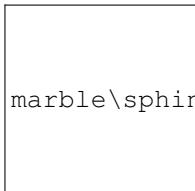
- **resource_factory** (*Callable*[[*Disposable*]]) – Factory function to obtain a resource object.
- **observable_factory** (*Callable*[[*Disposable*], *Observable*]) – Factory function to obtain an observable sequence that depends on the obtained resource.

Return type *Observable*

Returns An observable sequence whose lifetime controls the lifetime of the dependent resource object.

`rx.with_latest_from(*sources)`

Merges the specified observable sequences into one observable sequence by creating a tuple only when the first observable sequence produces an element. 1c1361f657a91ebf5ade83dbd9800bfa7ec5e7b5.png



marble\sphinxhyphen {}1c1361f657a91ebf5ade83dbd9800bfa7ec5e7b5.png

Examples

```
>>> obs = rx.with_latest_from(obs1)
>>> obs = rx.with_latest_from([obs1, obs2, obs3])
```

Parameters **sources** (*Observable*) – Sequence of observables.

Return type *Observable*


Returns An observable sequence containing the result of combining elements of the sources into a tuple.

`rx.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.

8f017bb0b30f69529ca9935a37b7146a7bb53865.png



marble\sphinxhyphen {}8f017bb0b30f69529ca9935a37b7

Example

```
>>> res = rx.zip(obs1, obs2)
```

Parameters **args** (*Observable*) – Observable sources to zip.

Return type *Observable*

Returns An observable sequence containing the result of combining elements of the sources as a tuple.

7.2 Observable

class `rx.Observable` (*subscribe=None*)

Observable base class.

Represents a push-style collection, which you can *pipe* into *operators*.

__init__ (*subscribe=None*)

Creates an observable sequence object from the specified subscription function.

Parameters **subscribe** (Optional[Callable[[*Observer*, Optional[*Scheduler*]], *Disposable*]]) – [Optional] Subscription function

subscribe (*observer=None*, *on_error=None*, *on_completed=None*, *on_next=None*, *, *scheduler=None*)

Subscribe an observer to the observable sequence.

You may subscribe using an observer or callbacks, not both; if the first argument is an instance of *Observer* or if it has a (callable) attribute named `on_next`, then any callback arguments will be ignored.

Examples

```
>>> source.subscribe()
>>> source.subscribe(observer)
>>> source.subscribe(observer, scheduler=scheduler)
>>> source.subscribe(on_next)
>>> source.subscribe(on_next, on_error)
>>> source.subscribe(on_next, on_error, on_completed)
>>> source.subscribe(on_next, on_error, on_completed, scheduler=scheduler)
```

Parameters

- **observer** (Union[*Observer*, Callable[[Any], None], None]) – [Optional] The object that is to receive notifications.

- **on_error** (Optional[Callable[[Exception], None]]) – [Optional] Action to invoke upon exceptional termination of the observable sequence.
- **on_completed** (Optional[Callable[[], None]]) – [Optional] Action to invoke upon graceful termination of the observable sequence.
- **on_next** (Optional[Callable[[Any], None]]) – [Optional] Action to invoke for each element in the observable sequence.
- **scheduler** (Optional[Scheduler]) – [Optional] The default scheduler to use for this subscription.

Return type *Disposable*

Returns Disposable object representing an observer's subscription to the observable sequence.

subscribe_ (on_next=None, on_error=None, on_completed=None, scheduler=None)
Subscribe callbacks to the observable sequence.

Examples

```
>>> source.subscribe_(on_next)
>>> source.subscribe_(on_next, on_error)
>>> source.subscribe_(on_next, on_error, on_completed)
```

Parameters

- **on_next** (Optional[Callable[[Any], None]]) – [Optional] Action to invoke for each element in the observable sequence.
- **on_error** (Optional[Callable[[Exception], None]]) – [Optional] Action to invoke upon exceptional termination of the observable sequence.
- **on_completed** (Optional[Callable[[], None]]) – [Optional] Action to invoke upon graceful termination of the observable sequence.
- **scheduler** (Optional[Scheduler]) – [Optional] The scheduler to use for this subscription.

Return type *Disposable*

Returns Disposable object representing an observer's subscription to the observable sequence.

```
pipe (*operators: Callable[['Observable'], 'Observable']) → 'Observable'
pipe () → 'Observable'
pipe (op1: Callable[['Observable'], A]) → A
pipe (op1: Callable[['Observable'], A], op2: Callable[[A], B]) → B
pipe (op1: Callable[['Observable'], A], op2: Callable[[A], B], op3: Callable[[B], C]) → C
pipe (op1: Callable[['Observable'], A], op2: Callable[[A], B], op3: Callable[[B], C], op4:
    Callable[[C], D]) → D
pipe (op1: Callable[['Observable'], A], op2: Callable[[A], B], op3: Callable[[B], C], op4:
    Callable[[C], D], op5: Callable[[D], E]) → E
pipe (op1: Callable[['Observable'], A], op2: Callable[[A], B], op3: Callable[[B], C], op4:
    Callable[[C], D], op5: Callable[[D], E], op6: Callable[[E], F]) → F
pipe (op1: Callable[['Observable'], A], op2: Callable[[A], B], op3: Callable[[B], C], op4:
    Callable[[C], D], op5: Callable[[D], E], op6: Callable[[E], F], op7: Callable[[F], G]) → G
```

Compose multiple operators left to right.

Composes zero or more operators into a functional composition. The operators are composed from left to right. A composition of zero operators gives back the original source.

Examples

```
>>> source.pipe() == source
>>> source.pipe(f) == f(source)
>>> source.pipe(g, f) == f(g(source))
>>> source.pipe(h, g, f) == f(g(h(source)))
```

Parameters **operators** (Callable[[*Observable*], Any]) – Sequence of operators.

Return type Any

Returns The composed observable.

run()

Run source synchronously.

Subscribes to the observable source. Then blocks and waits for the observable source to either complete or error. Returns the last value emitted, or throws exception if any error occurred.

Examples

```
>>> result = run(source)
```

Raises

- **SequenceContainsNoElementsError** – if observable completes (on_completed) without any values being emitted.
- **Exception** – raises exception if any error (on_error) occurred.

Return type Any

Returns The last element emitted from the observable.

__await__()

Awaits the given observable.

Return type Any

Returns The last item of the observable sequence.

__add__(other)

Pythonic version of *concat*.

Example

```
>>> zs = xs + ys
```

Parameters **other** – The second observable sequence in the concatenation.

Return type *Observable*

Returns Concatenated observable sequence.

__iadd__(other)

Pythonic use of *concat*.

Example

```
>>> xs += ys
```

Parameters **other** – The second observable sequence in the concatenation.

Return type *Observable*

Returns Concatenated observable sequence.

__getitem__ (*key*)

Pythonic version of *slice*.

Slices the given observable using Python slice notation. The arguments to slice are *start*, *stop* and *step* given within brackets *[]* and separated by the colons *:*.

It is basically a wrapper around the operators *skip*, *skip_last*, *take*, *take_last* and *filter*.

The following diagram helps you remember how slices works with streams. Positive numbers are relative to the start of the events, while negative numbers are relative to the end (close) of the stream.

r	---	e	---	a	---	c	---	t	---	i	---	v	---	e	---	!
0	1	2	3	4	5	6	7	8								
-8	-7	-6	-5	-4	-3	-2	-1	0								

Examples

```
>>> result = source[1:10]
>>> result = source[1:-2]
>>> result = source[1:-1:2]
```

Parameters **key** – Slice object

Return type *Observable*

Returns Sliced observable sequence.

Raises **TypeError** – If key is not of type *int* or *slice*

7.3 Subject

class `rx.subject.Subject`

Represents an object that is both an observable sequence as well as an observer. Each notification is broadcasted to all subscribed observers.

__init__ ()

Creates an observable sequence object from the specified subscription function.

Parameters **subscribe** – [Optional] Subscription function

on_next (*value*)

Notifies all subscribed observers with the value.

Parameters **value** (*Any*) – The value to send to all subscribed observers.

Return type *None*

on_error (*error*)

Notifies all subscribed observers with the exception.

Parameters **error** (*Exception*) – The exception to send to all subscribed observers.

Return type *None*

on_completed ()

Notifies all subscribed observers of the end of the sequence.

Return type *None*

dispose ()

Unsubscribe all observers and release resources.

Return type *None*

class `rx.subject.BehaviorSubject` (*value*)

Represents a value that changes over time. Observers can subscribe to the subject to receive the last (or initial) value and all subsequent notifications.

__init__ (*value*)

Initializes a new instance of the BehaviorSubject class which creates a subject that caches its last value and starts with the specified value.

Parameters **value** – Initial value sent to observers when no other value has been received by the subject yet.

dispose ()

Release all resources.

Releases all resources used by the current instance of the BehaviorSubject class and unsubscribe all observers.

Return type *None*

class `rx.subject.ReplaySubject` (*buffer_size=None, window=None, scheduler=None*)

Represents an object that is both an observable sequence as well as an observer. Each notification is broadcasted to all subscribed and future observers, subject to buffer trimming policies.

__init__ (*buffer_size=None, window=None, scheduler=None*)

Initializes a new instance of the ReplaySubject class with the specified buffer size, window and scheduler.

Parameters

- **buffer_size** (*Optional[int]*) – [Optional] Maximum element count of the replay buffer.
- **[Optional]** (*window*) – Maximum time length of the replay buffer.
- **scheduler** (*Optional[Scheduler]*) – [Optional] Scheduler the observers are invoked on.

dispose ()

Releases all resources used by the current instance of the ReplaySubject class and unsubscribe all observers.

Return type *None*

class `rx.subject.AsyncSubject`

Represents the result of an asynchronous operation. The last value before the close notification, or the error received through `on_error`, is sent to all subscribed observers.

__init__ ()

Creates a subject that can only receive one value and that value is cached for all future observations.

dispose()

Unsubscribe all observers and release resources.

Return type `None`

7.4 Schedulers

class `rx.scheduler.CatchScheduler` (*scheduler, handler*)

__init__ (*scheduler, handler*)

Wraps a scheduler, passed as constructor argument, adding exception handling for scheduled actions. The handler should return True to indicate it handled the exception successfully. Falsy return values will be taken to indicate that the exception should be escalated (raised by this scheduler).

Parameters

- **scheduler** (*Scheduler*) – The scheduler to be wrapped.
- **handler** (`Callable[[Exception], bool]`) – Callable to handle exceptions raised by wrapped scheduler.

property now

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type `datetime`

Returns The scheduler's current time, as a datetime instance.

schedule (*action, state=None*)

Schedules an action to be executed.

Parameters

- **action** (`Callable[[Scheduler, Optional[Disposable]], Optional[~TState]]`) – Action to be executed.
- **state** (`Optional[~TState]`) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime, action, state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (`Union[timedelta, float]`) – Relative time after which to execute the action.
- **action** (`Callable[[Scheduler, Optional[Disposable]], Optional[~TState]]`) – Action to be executed.
- **state** (`Optional[~TState]`) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime, action, state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable***Returns** The disposable object used to cancel the scheduled action (best effort).**schedule_periodic** (*period, action, state=None*)

Schedules a periodic piece of work.

Parameters

- **period** (Union[timedelta, float]) – Period in seconds or timedelta for running the work periodically.
- **action** (Callable[[Optional[~TState], Optional[~TState]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] Initial state passed to the action upon the first iteration.

Return type *Disposable***Returns** The disposable object used to cancel the scheduled recurring action (best effort).**class** rx.scheduler.**CurrentThreadScheduler**

Represents an object that schedules units of work on the current thread. You should never schedule timeouts using the *CurrentThreadScheduler*, as that will block the thread while waiting.

Each instance manages a number of trampolines (and queues), one for each thread that calls a *schedule* method. These trampolines are automatically garbage-collected when threads disappear, because they're stored in a weak key dictionary.

classmethod **singleton**()

Obtain a singleton instance for the current thread. Please note, if you pass this instance to another thread, it will effectively behave as if it were created by that other thread (separate trampoline and queue).

Return type *CurrentThreadScheduler***Returns** The singleton *CurrentThreadScheduler* instance.**__init__**()

Creates a scheduler that bounces its work off the trampoline.

class rx.scheduler.**EventLoopScheduler** (*thread_factory=None, exit_if_empty=False*)

Creates an object that schedules units of work on a designated thread.

__init__ (*thread_factory=None, exit_if_empty=False*)

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

schedule (*action, state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_periodic (*period*, *action*, *state=None*)

Schedules a periodic piece of work.

Parameters

- **period** (Union[timedelta, float]) – Period in seconds or timedelta for running the work periodically.
- **action** (Callable[[Optional[~TState]], Optional[~TState]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] Initial state passed to the action upon the first iteration.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled recurring action (best effort).

run ()

Event loop scheduled on the designated event loop thread. The loop is suspended/resumed using the condition which gets notified by calls to Schedule or calls to dispose.

Return type *None*

dispose ()

Ends the thread associated with this scheduler. All remaining work in the scheduler queue is abandoned.

Return type *None*

class rx.scheduler.HistoricalScheduler (*initial_clock=None*)

Provides a virtual time scheduler that uses datetime for absolute time and timedelta for relative time.

__init__ (*initial_clock=None*)

Creates a new historical scheduler with the specified initial clock value.

Parameters **initial_clock** (Optional[datetime]) – Initial value for the clock.

property now

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type datetime

Returns The scheduler's current time, as a datetime instance.

classmethod add (*absolute, relative*)

Adds a relative time value to an absolute time value.

Parameters

- **absolute** (Union[datetime, float]) – Absolute virtual time value.
- **relative** (Union[timedelta, float]) – Relative virtual time value to add.

Return type Union[datetime, float]

Returns The resulting absolute virtual time sum value.

class rx.scheduler.ImmediateScheduler

Represents an object that schedules units of work to run immediately, on the current thread. You're not allowed to schedule timeouts using the ImmediateScheduler since that will block the current thread while waiting. Attempts to do so will raise a `WouldBlockException`.

static **__new__** (*cls*)

Create and return a new object. See `help(type)` for accurate signature.

Return type ImmediateScheduler

schedule (*action, state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime, action, state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

class rx.scheduler.**NewThreadScheduler** (*thread_factory=None*)

Creates an object that schedules each unit of work on a separate thread.

__init__ (*thread_factory=None*)

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

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_periodic (*period*, *action*, *state=None*)

Schedules a periodic piece of work.

Parameters

- **period** (Union[timedelta, float]) – Period in seconds or timedelta for running the work periodically.
- **action** (Callable[[Optional[~TState]], Optional[~TState]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] Initial state passed to the action upon the first iteration.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled recurring action (best effort).

class rx.scheduler.ThreadPoolScheduler (*max_workers=None*)

A scheduler that schedules work via the thread pool.

class ThreadPoolThread (*executor*, *target*)

Wraps a concurrent future as a thread.

__init__ (*executor*, *target*)

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

start ()

Return type None

cancel ()

Return type None

__init__ (*max_workers=None*)

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

class rx.scheduler.TimeoutScheduler

A scheduler that schedules work via a timed callback.

static **__new__** (*cls*)

Create and return a new object. See help(type) for accurate signature.

Return type TimeoutScheduler

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

class rx.scheduler.TrampolineScheduler

Represents an object that schedules units of work on the trampoline. You should never schedule timeouts using the *TrampolineScheduler*, as it will block the thread while waiting.

Each instance has its own trampoline (and queue), and you can schedule work on it from different threads. Beware though, that the first thread to call a *schedule* method while the trampoline is idle will then remain occupied until the queue is empty.

__init__ ()

Creates a scheduler that bounces its work off the trampoline.

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_required ()

Test if scheduling is required.

Gets a value indicating whether the caller must call a schedule method. If the trampoline is active, then it returns False; otherwise, if the trampoline is not active, then it returns True.

Return type bool

ensure_trampoline (*action*)

Method for testing the TrampolineScheduler.

class rx.scheduler.**VirtualTimeScheduler** (*initial_clock=0*)

Virtual Scheduler. This scheduler should work with either datetime/timespan or ticks as int/int

__init__ (*initial_clock=0*)

Creates a new virtual time scheduler with the specified initial clock value.

Parameters **initial_clock** – Initial value for the clock.

property now

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type datetime

Returns The scheduler's current time, as a datetime instance.

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime, action, state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

start ()

Starts the virtual time scheduler.

Return type None

stop ()

Stops the virtual time scheduler.

Return type None

advance_to (*time*)

Advances the schedulers clock to the specified absolute time, running all work til that point.

Parameters **time** (Union[datetime, float]) – Absolute time to advance the schedulers clock to.

Return type None

advance_by (*time*)

Advances the schedulers clock by the specified relative time, running all work scheduled for that timespan.

Parameters **time** (Union[timedelta, float]) – Relative time to advance the schedulers clock by.

Return type None

sleep (*time*)

Advances the schedulers clock by the specified relative time.

Parameters **time** (Union[timedelta, float]) – Relative time to advance the schedulers clock by.

Return type None

abstract classmethod add (*absolute, relative*)

Adds a relative time value to an absolute time value.

Parameters

- **absolute** (Union[datetime, float]) – Absolute virtual time value.
- **relative** (Union[timedelta, float]) – Relative virtual time value to add.

Return type Union[datetime, float]

Returns The resulting absolute virtual time sum value.

class rx.scheduler.eventloop.**AsyncIOScheduler** (*loop*)

A scheduler that schedules work via the asyncio mainloop. This class does not use the asyncio threadsafe methods, if you need those please use the AsyncIOThreadSafeScheduler class.

__init__ (*loop*)

Create a new AsyncIOScheduler.

Parameters **loop** (AbstractEventLoop) – Instance of asyncio event loop to use; typically, you would get this by asyncio.get_event_loop()

schedule (*action, state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime, action, state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime, action, state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

property **now**

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type datetime

Returns The scheduler's current time, as a datetime instance.

class rx.scheduler.eventloop.**AsyncIOThreadSafeScheduler** (*loop*)

A scheduler that schedules work via the asyncio mainloop. This is a subclass of AsyncIOScheduler which uses the threadsafe asyncio methods.

__init__ (*loop*)

Create a new AsyncIOThreadSafeScheduler.

Parameters **loop** (AbstractEventLoop) – Instance of asyncio event loop to use; typically, you would get this by asyncio.get_event_loop()

schedule (*action, state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime, action, state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime, action, state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

class rx.scheduler.eventloop.**EventletScheduler** (*eventlet*)

A scheduler that schedules work via the eventlet event loop.

<http://eventlet.net/>

__init__ (*eventlet*)

Create a new EventletScheduler.

Parameters `eventlet` (Any) – The eventlet module to use; typically, you would get this by `import eventlet`

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

property now

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type *datetime*

Returns The scheduler's current time, as a datetime instance.

class `rx.scheduler.eventloop.GEventScheduler` (*gevent*)

A scheduler that schedules work via the GEvent event loop.

<http://www.gevent.org/>

__init__ (*gevent*)

Create a new GEventScheduler.

Parameters **gevent** (Any) – The gevent module to use; typically ,you would get this by import gevent

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

property now

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type datetime

Returns The scheduler’s current time, as a datetime instance.

class rx.scheduler.eventloop.**IOLoopScheduler** (*loop*)

A scheduler that schedules work via the Tornado I/O main event loop.

Note, as of Tornado 6, this is just a wrapper around the asyncio loop.

<http://tornado.readthedocs.org/en/latest/ioloop.html>

__init__ (*loop*)

Create a new IOLoopScheduler.

Parameters **loop** (Any) – The ioloop to use; typically, you would get this by `tornado import ioloop; ioloop.IOLoop.current()`

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

property now

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type *datetime*

Returns The scheduler's current time, as a datetime instance.

class `rx.scheduler.eventloop.TwistedScheduler` (*reactor*)

A scheduler that schedules work via the Twisted reactor mainloop.

__init__ (*reactor*)

Create a new TwistedScheduler.

Parameters **reactor** (Any) – The reactor to use; typically, you would get this by `from twisted.internet import reactor`

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

property now

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type *datetime*

Returns The scheduler's current time, as a datetime instance.

class rx.scheduler.mainloop.**GtkScheduler** (*glib*)

A scheduler that schedules work via the GLib main loop used in GTK+ applications.

See <https://wiki.gnome.org/Projects/PyGObject>

__init__ (*glib*)

Create a new GtkScheduler.

Parameters **glib** (Any) – The GLib module to use; typically, you would get this by import gi; gi.require_version('Gtk', '3.0'); from gi.repository import GLib

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_periodic (*period*, *action*, *state=None*)

Schedules a periodic piece of work to be executed in the loop.

Parameters

- **period** (Union[timedelta, float]) – Period in seconds for running the work repeatedly.
- **action** (Callable[[Optional[~TState]], Optional[~TState]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Returns: The disposable object used to cancel the scheduled action (best effort).

Return type *Disposable*

class rx.scheduler.mainloop.**TkinterScheduler** (*root*)

A scheduler that schedules work via the Tkinter main event loop.

<http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/universal.html> <http://effbot.org/tkinterbook/widget.htm>

__init__ (*root*)

Create a new TkinterScheduler.

Parameters **root** (Any) – The Tk instance to use; typically, you would get this by import tkinter; tkinter.Tk()

schedule (*action, state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime, action, state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime, action, state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

class rx.scheduler.mainloop.**PyGameScheduler** (*pygame*)

A scheduler that schedules works for PyGame.

Note that this class expects the caller to invoke run() repeatedly.

<http://www.pygame.org/docs/ref/time.html> <http://www.pygame.org/docs/ref/event.html>

__init__ (*pygame*)

Create a new PyGameScheduler.

Parameters **pygame** (Any) – The PyGame module to use; typically, you would get this by import pygame

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime. :type duetime: Union[timedelta, float] :param duetime: Relative time after which to execute the action. :type action: Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]] :param action: Action to be executed. :type state: Optional[~TState] :param state: [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

class rx.scheduler.mainloop.**QtScheduler** (*QtCore*)

A scheduler for a PyQt5/PySide2 event loop.

__init__ (*QtCore*)

Create a new QtScheduler.

Parameters **QtCore** (Any) – The QtCore instance to use; typically you would get this by either import PyQt5.QtCore or import PySide2.QtCore

schedule (*action*, *state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime, action, state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_periodic (*period, action, state=None*)

Schedules a periodic piece of work to be executed in the loop.

Parameters

- **period** (Union[timedelta, float]) – Period in seconds for running the work repeatedly.
- **action** (Callable[[Optional[~TState]], Optional[~TState]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Returns: The disposable object used to cancel the scheduled action (best effort).

Return type *Disposable*

class rx.scheduler.mainloop.**WxScheduler** (*wx*)

A scheduler for a wxPython event loop.

__init__ (*wx*)

Create a new WxScheduler.

Parameters **wx** (Any) – The wx module to use; typically, you would get this by import wx

cancel_all ()

Cancel all scheduled actions.

Should be called when destroying wx controls to prevent accessing dead wx objects in actions that might be pending.

Return type None

schedule (*action, state=None*)

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

schedule_periodic (*period*, *action*, *state=None*)

Schedules a periodic piece of work to be executed in the loop.

Parameters

- **period** (Union[timedelta, float]) – Period in seconds for running the work repeatedly.
- **action** (Callable[[Optional[~TState]], Optional[~TState]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

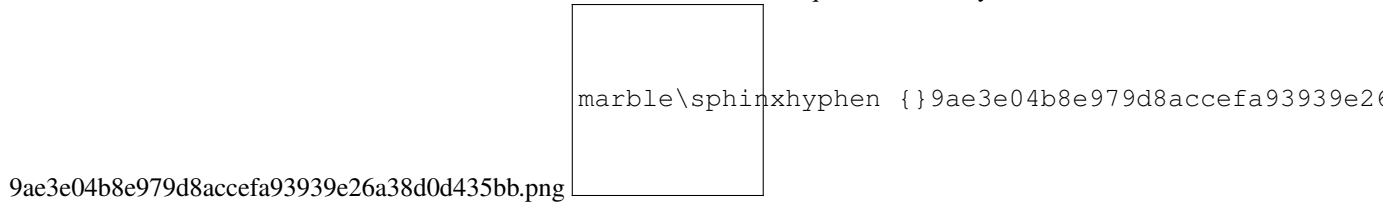
Returns: The disposable object used to cancel the scheduled action (best effort).

Return type *Disposable*

7.5 Operators

`rx.operators.all` (*predicate*)

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



Example

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

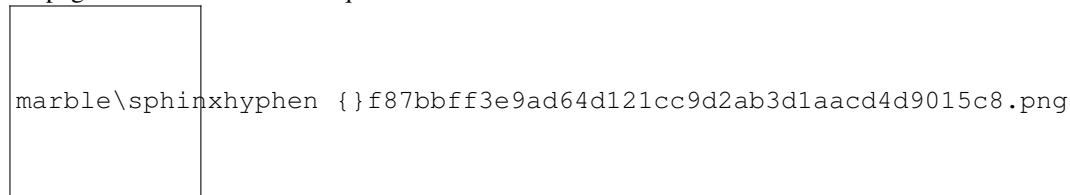
Parameters **predicate** (Callable[[~T1], bool]) – A function to test each element for a condition.

Return type Callable[[*Observable*], *Observable*]

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.

`rx.operators.amb` (*right_source*)

Propagates the observable sequence that reacts first. f87bbff3e9ad64d121cc9d2ab3d1aacd4d9015c8.png



Example

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

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.as_observable` ()

Hides the identity of an observable sequence.

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.average` (*key_mapper=None*)

The average operator.

Computes the average of an observable sequence of values that are in the sequence or obtained by invoking a transform function on each element of the input sequence if present.

b764a37a8ac08a8d9d9f8022185c88d1c8fb776b.png

marble\sphinxhyphen {}b764a37a8ac08a8d9d9f8022185c

Examples

```
>>> op = average()
>>> op = average(lambda x: x.value)
```

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

Return type Callable[[*Observable*], *Observable*]

Returns An operator function that takes an observable source and returns an observable sequence containing a single element with the average of the sequence of values.

`rx.operators.buffer` (*boundaries*)

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

81dbcd0cd7a21fb6c1f4939387f5ff16b1556101.png

marble\sphinxhyphen {}81dbcd0cd7a21fb6c1f4939387f5f

Examples

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

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

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.buffer_when` (*closing_mapper*)

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

1a971c0609b22ba1f5d5d63abfc62be378139340.png

marble\sphinxhyphen {}1a971c0609b22ba1f5d5d63abfc6

Examples

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

Parameters `closing_mapper` (`Callable[[Observable], Observable]`) – 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], Observable]`

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

`rx.operators.buffer_toggle` (`openings`, `closing_mapper`)

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

9c9ad906399c982a7ca5eb85fd1ce3d2c4ab39a5.png

marble\sphinxhyphen {}9c9ad906399c982a7ca5eb85fd1ce

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

Parameters

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

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

`rx.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

marble\sphinxhyphen {}58fe81f8f4a247bd8

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], Observable]`

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

`rx.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



marble\sphinxhyphen {}8dca4a83325669720b5e

information. 8dca4a83325669720b5e0100ff548f37f73038f.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

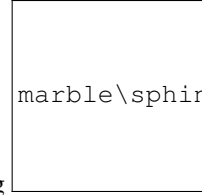
- **timespan** (`Union[timedelta, float]`) – Length of each buffer (specified as a float denoting seconds or an instance of `timedelta`).
- **timeshift** (`Union[timedelta, float, None]`) – [Optional] Interval between creation of consecutive buffers (specified as a float denoting seconds or an instance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

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

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

`rx.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. 

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

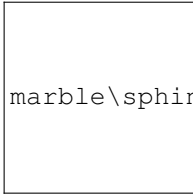
- **timespan** – Maximum time length of a buffer.
- **count** – Maximum element count of a buffer.
- **scheduler** – [Optional] Scheduler to run buffering timers on. If not specified, the timeout scheduler is used.

Return type Callable[[*Observable*], *Observable*]

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

`rx.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))
```

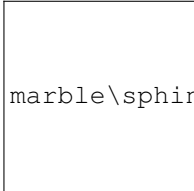
Parameters handler (Union[*Observable*, Callable[[Exception, *Observable*], *Observable*]]) – Second observable sequence used to produce results when an error occurred in the first sequence, or an exception handler function that returns an observable sequence given the error and source observable that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

Return type Callable[[*Observable*], *Observable*]

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.

`rx.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

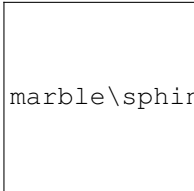
```
>>> obs = combine_latest(other)
>>> obs = combine_latest(obs1, obs2, obs3)
```

Return type Callable[[*Observable*], *Observable*]

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.

`rx.operators.concat(*sources)`

Concatenates all the observable sequences. 4544dd242d02df82ac059a82701a17b5b31135d4.png



marble\sphinxhyphen {}4544dd242d02df82ac059a82701a17b5b31135d4.png

Examples


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

Return type Callable[[*Observable*], *Observable*]

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.

`rx.operators.contains(value, comparer=None)`

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



marble\sphinxhyphen {}b55e762ff5a35f34ec54f1b2ec6c70
b55e762ff5a35f34ec54f1b2ec6c70eaf7775a3f.png

Examples

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

Parameters

- **value** (Any) – The value to locate in the source sequence.
- **comparer** (Optional[Callable[[~T1, ~T2], bool]]) – [Optional] An equality comparer to compare elements.

Return type Callable[[Observable], Observable]

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.

`rx.operators.count` (*predicate=None*)

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.

e210088b88c128750bb017ffb1374b2f463bff59.png

marble\sphinxhyphen {}e210088b88c128750bb017ffb1374

Examples

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

Parameters **predicate** (Optional[Callable[[~T1], bool]]) – A function to test each element for a condition.

Return type Callable[[Observable], Observable]

Returns An operator function that takes an observable source and returns an observable sequence containing a single element with a number that represents how many elements in the input sequence satisfy the condition in the predicate function if provided, else the count of items in the sequence.

`rx.operators.debounce` (*duetime, scheduler=None*)

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

ddf862178779eaa8593df342589be502be49ddb.png

marble\sphinxhyphen {}ddf862178779eaa8593df342589b

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[Scheduler]) – Scheduler to debounce values on.

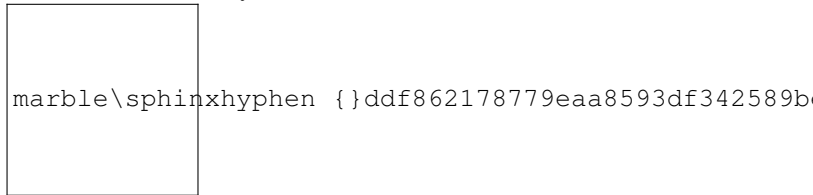
Return type Callable[[Observable], Observable]

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

`rx.operators.throttle_with_timeout(duetime, scheduler=None)`

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

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[Scheduler]) – Scheduler to debounce values on.

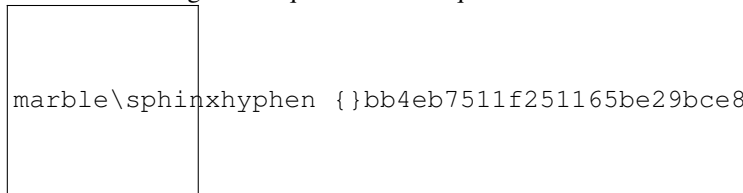
Return type Callable[[Observable], Observable]

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

`rx.operators.default_if_empty(default_value=None)`

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

empty. bb4eb7511f251165be29bce867fc77f1f2e56724.png



Examples

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

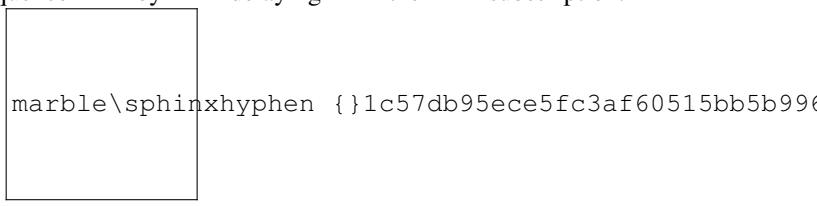
Parameters **default_value** (Optional[Any]) – The value to return if the sequence is empty. If not provided, this defaults to None.

Return type Callable[[Observable], Observable]

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.

`rx.operators.delay_subscription(duetime, scheduler=None)`

Time shifts the observable sequence by delaying the subscription.



Example

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

Parameters

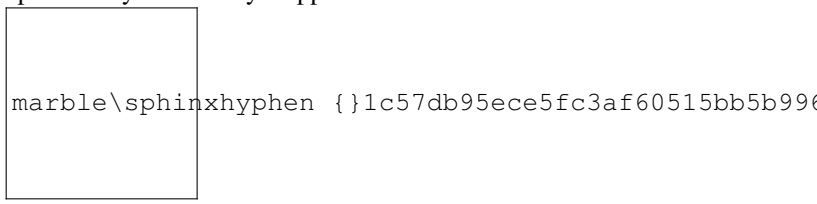
- **duetime** (Union[datetime, timedelta, float]) – Absolute or relative time to perform the subscription
- **at** . –
- **scheduler** (Optional[Scheduler]) – Scheduler to delay subscription on.

Return type Callable[[Observable], Observable]

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

`rx.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(rx.timer(2.0), lambda x: rx.timer(x))
```

Parameters

- **subscription_delay** – [Optional] Sequence indicating the delay for the subscription to the source.
- **delay_duration_mapper** – [Optional] Selector function to retrieve a sequence indicating the delay for each given element.

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.dematerialize()`
Dematerialize operator.

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

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.delay(duetime, scheduler=None)`

marble\sphinxhyphen {}1c57db95ece5fc3

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[*Scheduler*]) – [Optional] Scheduler to run the delay timers on. If not specified, the timeout scheduler is used.

Return type Callable[[*Observable*], *Observable*]

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

`rx.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 which can grow large. 1a8e43d967e0923c86f18e853fa7e87e43a808f9.png

marble\sphinxhyphen {}1a8e43d

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[[~T1], ~T2]]) – [Optional] A function to compute the comparison key for each element.
- **comparer** (Optional[Callable[[~T1, ~T2], bool]]) – [Optional] Used to compare items in the collection.

Return type Callable[[*Observable*], *Observable*]

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.

`rx.operators.distinct_until_changed` (*key_mapper=None, comparer=None*)

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

and the `comparer`. 3783ceec8ed2d9cdf16e748d7f8c2f8271ddbb2f.png

marble\sphinxhyphen {}3783ceec8ed2d9cdf

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[[~T1], ~T2]]) – [Optional] A function to compute the comparison key for each element. If not provided, it projects the value.
- **comparer** (Optional[Callable[[~T1, ~T2], bool]]) – [Optional] Equality comparer for computed key values. If not provided, defaults to an equality comparer function.

Return type Callable[[*Observable*], *Observable*]

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.

```
rx.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.

6e08c40414ac4aff5c3c9fcc512aa79bd7635234.png

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

Parameters `observer` (*Observer*) – Observer

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

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

```
rx.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.

dce15dead801908327ef799bfcdfa63a4c9f958d.png

Examples

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

Parameters

- **on_next** (Optional[Callable[[Any], 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], Observable]`

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

`rx.operators.do_while(condition)`

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

74f3910edce12fac1a74594adad08937786aea97.png

Parameters `condition` (`Callable[[~T1], bool]`) – The condition which determines if the source will be repeated.

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

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

`rx.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], Observable]`

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.

`rx.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 {}f6ff9e502837f3c34a2c23a40437f3

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[Any]`) – [Optional] The default value if the index is outside the bounds of the source sequence.

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

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.

`rx.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

marble\sphinxhyphen {}0718300e5a5496454a339cdc4721

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

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

`rx.operators.expand(mapper)`

Expands an observable sequence by recursively invoking *mapper*.

Parameters **mapper** (`Callable[[~T1], ~T2]`) – 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], Observable]`

Returns An observable sequence containing all the elements produced by the recursive expansion.

`rx.operators.filter(predicate)`

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

2ac5743543b7ddb0ba7b7dc3cedacda9de1a73eb.png

marble\sphinxhyphen {}2ac5743543b7ddb0ba7b7dc3ceda

Example

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

Parameters **predicate** (Callable[[~T1], bool]) – A function to test each source element for a condition.

Return type Callable[[Observable], Observable]

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.

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

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

915723ed35c180ce5428050d27cb2d4cbfc8ddf1.png

marble\sphinxhyphen {}915723ed35c180ce5428050d27cb2

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], Observable]

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.

`rx.operators.finally_action(action)`

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

092df87afded0d41cb3c9b7aa533dfd5b3bca776.png

marble\sphinxhyphen {}092df87afded0d41cb3c9b7aa533d

Example

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

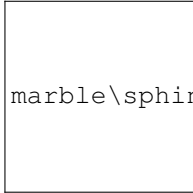
Parameters **action** (Callable) – Action to invoke after the source observable sequence terminates.

Return type Callable[[Observable], Observable]

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

`rx.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



marble\sphinxhyphen {}50c59c6e10555a1998fa10d51cbd9ffdaba3ac3d.png

Parameters `predicate` (Callable[[~T1], bool]) – The predicate that defines the conditions of the element to search for.

Return type Callable[[Observable], Observable]

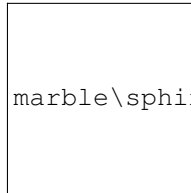
Returns An operator function that takes an observable source and returns an observable sequence with the first element that matches the conditions defined by the specified predicate, if found otherwise, None.

`rx.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.

marble\sphinxhyphen {}479c504797deac7666c2236ff162

479c504797deac7666c2236ff16287b5b001b465.png



Parameters `predicate` (Callable[[~T1], bool]) – The predicate that defines the conditions of the element to search for.

Return type Callable[[Observable], Observable]

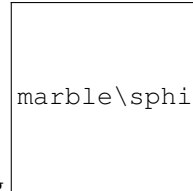
Returns An operator function that takes an observable source and returns an observable sequence with the zero-based index of the first occurrence of an element that matches the conditions defined by match, if found; otherwise, -1.

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

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

marble\sphinxhyphen {}f8ff77b2735d

first item in the sequence. f8ff77b2735db075f91ed29f2fe3fc0b1c918d49.png



Examples

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

Parameters **predicate** (Optional[Callable[[~T1], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.

Return type Callable[[Observable], Observable]

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.

`rx.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 {}e2da2e194120

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[[~T1], 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], Observable]

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.

`rx.operators.flat_map` (*mapper=None*)



marble\sphinxhyphen {}0796d3027064

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[Callable[[~T1], ~T2]]) – 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], Observable]

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.

`rx.operators.flat_map_indexed` (*mapper_indexed=None*)

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.

0796d302706411e6023b5671684563964c0afa1c.png

marble\sphinxhyphen {}0796d302706411e6023b56716845

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[Callable[[~T1, int], ~T2]]) – [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], Observable]

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.

`rx.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** (Callable[[~T1], ~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], Observable]

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.

`rx.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 {}e0cc64db1260f70482180126b622

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], Observable]

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.

`rx.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 lifetime request. c98393978f802bb0fe67988772a45adeda4060ab.png

marble\sphinxhyphen {}c98393978f802bb0fe67988772a45adeda4060ab.png

Examples

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

Parameters

- **key_mapper** (Callable[[~T1], ~T2]) – A function to extract the key for each element.
- **element_mapper** (Optional[Callable[[~T1], ~T2]]) – A function to map each source element to an element in an observable group.
- **duration_mapper** (Callable[[GroupedObservable], Observable]) – A function to signal the expiration of a group.
- **subject_mapper** (Optional[Callable[[], Subject]]) – A function that returns a subject used to initiate a grouped observable. Default mapper returns a Subject object.

Return type Callable[[Observable], Observable]

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.

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

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

marble\sphinxhyphen {}96c6bd5e72804426dd0d45e7be6f65a95cfc23bd.png

96c6bd5e72804426dd0d45e7be6f65a95cfc23bd.png

Parameters

- **right** (Observable) – The right observable sequence to join elements for.
- **left_duration_mapper** (Callable[[Any], Observable]) – 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]) – 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], Observable]

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.

`rx.operators.ignore_elements()`

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

marble\sphinxhyphen {}495529f1b8626a1641e59c8045dc

495529f1b8626a1641e59c8045dc3bfe4a46e52b.png

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

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.

`rx.operators.is_empty()`

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

marble\sphinxhyphen {}84685da521ceac3a1e3151b469388c793833bd91.png

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

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.

`rx.operators.join(right, left_duration_mapper, right_duration_mapper)`

Correlates the elements of two sequences based on overlapping durations.

marble\sphinxhyphen {}b3ff56065e83731d477f51b5fc600

b3ff56065e83731d477f51b5fc6003d2fc22a162.png

Parameters

- **right** (*Observable*) – The right observable sequence to join elements for.
- **left_duration_mapper** (`Callable[[Any], Observable]`) – 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]`) – 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], Observable]`

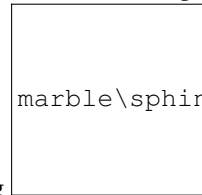
Returns An operator function that takes an observable source and returns an observable sequence that contains elements combined into a tuple from source elements that have an overlapping duration.

`rx.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

the last element. 7026868b150bc186397e1cda4dff4ff1bf30b8ca.png



Examples

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

Parameters **predicate** (Optional[Callable[[~T1], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.

Return type Callable[[Observable], Observable]

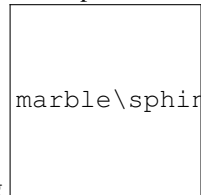
Returns An operator function that takes an observable source and returns an observable sequence containing the last element in the observable sequence that satisfies the condition in the predicate.

`rx.operators.last_or_default` (*predicate=None, default_value=None*)

The last_or_default operator.

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

if no such element exists. ac702793ffd235d9aecffd4dccc018f409b71c.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[[~T1], 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], Observable]

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.

`rx.operators.map (mapper=None)`

The map operator.

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

52e113b590f7fb5e7471c6d56c1829f94703f688.png

marble\sphinxhyphen {}52e113b590f7fb5e7471c6d56c182

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], Observable]

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.

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

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

6b178e548e0f0c5ecff8e62fc2591fb17b9d3973.png

marble\sphinxhyphen {}6b178e548e0f0c5ecff8e62fc2591

Example

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

Parameters `mapper_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], Observable]

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.

`rx.operators.materialize ()`

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

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

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

`rx.operators.max` (*comparer=None*)

Returns the maximum value in an observable sequence according to the specified comparer.

90c0d247c7fb08ce52df4fc3aeda0b963862f912.png

marble\sphinxhyphen-{}90c0d247c7fb08ce52df4fc3aeda0

Examples

```
>>> op = max()
>>> op = max(lambda x, y: x.value - y.value)
```

Parameters **comparer** (`Optional[Callable[[~T1, ~T2], bool]]`) – [Optional] Comparer used to compare elements.

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

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

`rx.operators.max_by` (*key_mapper*, *comparer=None*)

The max_by operator.

Returns the elements in an observable sequence with the maximum key value according to the specified com-

parer. 1ef072a3c5f1bb83ee48ada6c4083bcc73327e36.png

marble\sphinxhyphen-{}1ef072a3c5f1bb83ee48ada6

Examples

```
>>> res = max_by(lambda x: x.value)
>>> res = max_by(lambda x: x.value, lambda x, y: x - y)
```

Parameters

- **key_mapper** (`Callable[[~T1], ~T2]`) – Key mapper function.
- **comparer** (`Optional[Callable[[~T1, ~T2], bool]]`) – [Optional] Comparer used to compare key values.

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

Returns A partially applied operator function that takes an observable source and return an observable sequence containing a list of zero or more elements that have a maximum key value.

```
rx.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

sequence. 4ff9f21e0687a8c45cd9cdc6350bbe584661a43f.png

marble\sphinxhyphen {}4ff9f21e0687a8c45cd9cd

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], Observable]

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

```
rx.operators.merge_all()
```

The merge_all operator.

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

bf397132fa74dd0d17e17f08ce1314bbef0bbbd1.png

marble\sphinxhyphen {}bf397132fa74dd0d17e17f08ce131

Return type Callable[[Observable], Observable]

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

```
rx.operators.min(comparer=None)
```

The min operator.

Returns the minimum element in an observable sequence according to the optional comparer else a default

greater than less than check. 28a4004264d23d88a3a9c4b9822ada0e6dc3c0e6.png

marble\sphinxhyphen {}28a400426

Examples

```
>>> res = source.min()
>>> res = source.min(lambda x, y: x.value - y.value)
```

Parameters **comparer** (Optional[Callable[[~T1, ~T2], bool]]) – [Optional] Comparer used to compare elements.

Return type Callable[[Observable], Observable]

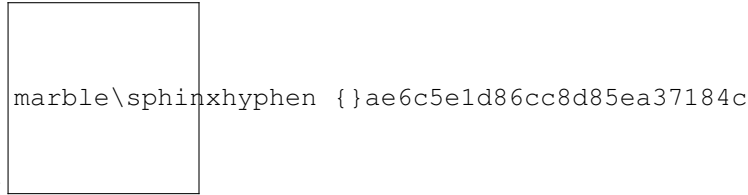
Returns An operator function that takes an observable source and returns an observable sequence containing a single element with the minimum element in the source sequence.

`rx.operators.min_by` (*key_mapper*, *comparer=None*)

The *min_by* operator.

Returns the elements in an observable sequence with the minimum key value according to the specified com-

parer. ae6c5e1d86cc8d85ea37184caa16746184e3b29a.png



Examples

```
>>> res = min_by(lambda x: x.value)
>>> res = min_by(lambda x: x.value, lambda x, y: x - y)
```

Parameters

- **key_mapper** (Callable[[~T1], ~T2]) – Key mapper function.
- **comparer** (Optional[Callable[[~T1, ~T2], bool]]) – [Optional] Comparer used to compare key values.

Return type Callable[[Observable], Observable]

Returns An operator function that takes an observable source and reuturns an observable sequence containing a list of zero or more elements that have a minimum key value.

`rx.operators.multicast` (*subject=None*, *subject_factory=None*, *mapper=None*)

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[*Scheduler*]], *Subject*]]) – Factory function to create an intermediate subject through which the source sequence’s elements will be multicast to the mapper function.
- **subject** (Optional[*Subject*]) – Subject to push source elements into.
- **mapper** (Optional[Callable[[ConnectableObservable], *Observable*]]) – [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*], Union[*Observable*, ConnectableObservable]]

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.

`rx.operators.observe_on(scheduler)`

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

Parameters **scheduler** (*Scheduler*) – 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*], *Observable*]

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

`rx.operators.on_error_resume_next(second)`

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

marble\sphinxhyphen {}cffe1529783c4bd3518f93

sequence. cffe1529783c4bd3518f934f74ac7f8960bd15ed.png

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

Return type Callable[[*Observable*], *Observable*]

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

`rx.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], Observable]`

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.

`rx.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 A marble diagram for the `partition` operator. It shows a single sequence of marbles (observations) being split into two separate sequences. The first sequence contains marbles for which the predicate returns true, and the second sequence contains marbles for which the predicate returns false. The diagram is represented by a box with the text "marble\sphinxhyphen {}9d1412bca63017f8545dfdf42baf36e766ddce27.png" inside.

Parameters

- **predicate** (`Callable[[~T1], bool]`) – The function to determine which output Observable
- **trigger a particular observation.** (*will*) –


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

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.

`rx.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.

9d1412bca63017f8545dfdf42baf36e766ddce27.png A marble diagram for the `partition_indexed` operator. It shows a single sequence of marbles (observations) being split into two separate sequences. The first sequence contains marbles for which the predicate returns true, and the second sequence contains marbles for which the predicate returns false. The diagram is represented by a box with the text "marble\sphinxhyphen {}9d1412bca63017f8545dfdf42baf36e766ddce27.png" inside.

Parameters

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

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

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

`rx.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` (Any) – The key to pluck.

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

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

`rx.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], Observable]`

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

`rx.operators.publish(mapper=None)`

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[[~T1], ~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], ConnectableObservable]`

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.

`rx.operators.publish_value(initial_value, mapper=None)`

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** (Any) – Initial value received by observers upon subscription.
- **mapper** (Optional[Callable[[~T1], ~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*], *Observable*]

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.

`rx.operators.reduce(accumulator, seed=<class 'rx.internal.utils.NotSet'>)`

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

marble\sphinxhyphen {}3f09a5a057c2de3bb200b642e329

Examples

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

Parameters

- **accumulator** (Callable[[~TState, ~T1], ~TState]) – An accumulator function to be invoked on each element.
- **seed** (Any) – Optional initial accumulator value.

Return type Callable[[*Observable*], *Observable*]

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.

`rx.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*], *Observable*]

`rx.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
- **repeats the sequence indefinitely.** (*provided,*) –

Return type Callable[[Observable], Observable]

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

`rx.operators.replay` (*mapper=None, buffer_size=None, window=None, scheduler=None*)

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[[~T1], ~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[Scheduler]) – [Optional] Scheduler the observers are invoked on.

Return type Callable[[*Observable*], Union[*Observable*, ConnectableObservable]]

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.

`rx.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)
```

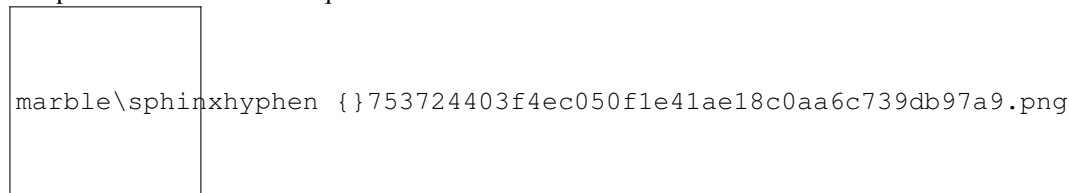
Parameters **retry_count** (Optional[int]) – [Optional] Number of times to retry the sequence. If not provided, retry the sequence indefinitely.

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.sample(sampler, scheduler=None)`

Samples the observable sequence at each interval. 



Examples

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

Parameters

- **sampler** (Union[timedelta, float, *Observable*]) – 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[*Scheduler*]) – Scheduler to use only when a time interval is given.


Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.scan(accumulator, seed=<class 'rx.internal.utils.NotSet'>)`

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()`. 

Examples

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

Parameters


- **accumulator** (Callable[[~TState, ~T1], ~TState]) – An accumulator function to be invoked on each element.
- **seed** (Any) – [Optional] The initial accumulator value.

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.sequence_equal(second, comparer=None)`

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

`comparer`. 

Examples

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

Parameters

- **second** (*Observable*) – Second observable sequence or array to compare.
- **comparer** (Optional[Callable[[~T1, ~T2], bool]]) – [Optional] Comparer used to compare elements of both sequences. No guarantees on order of comparer arguments.

Return type Callable[[*Observable*], *Observable*]

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.

`rx.operators.share()`

Share a single subscription among multiple observers.

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

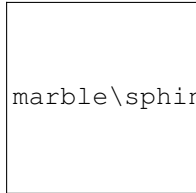
Return type `Callable[[Observable], Observable]`

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`.

`rx.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 {}59e02f94169eadf875bb292a59cd

59e02f94169eadf875bb292a59cda555b87d5092.png

Example

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

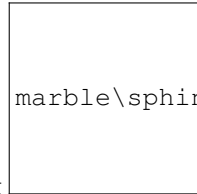
Parameters **predicate** (`Optional[Callable[[~T1], bool]]`) – [Optional] A predicate function to evaluate for elements in the source sequence.

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

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.

`rx.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 {}745696a3c38caf54674ca4831a80f

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[[~T1], 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.

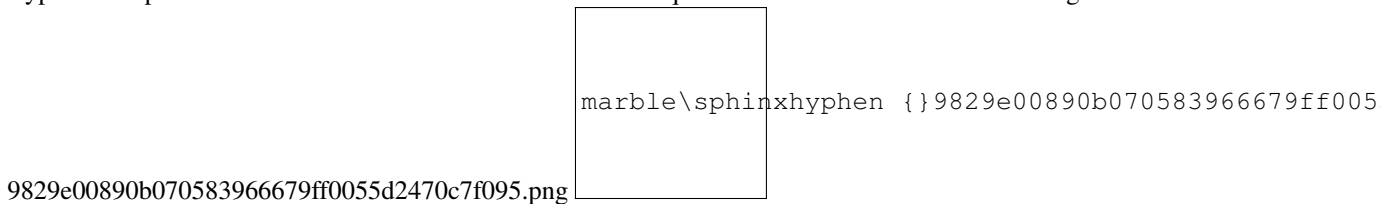
Return type `Callable[[Observable], Observable]`

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.

```
rx.operators.skip(count)
```

The skip operator.

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

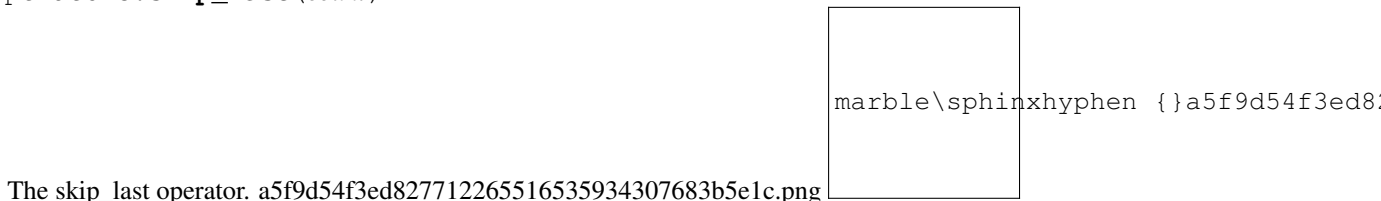


Parameters **count** (int) – The number of elements to skip before returning the remaining elements.

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

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.

```
rx.operators.skip_last (count)
```



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], Observable]`

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.

`rx.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

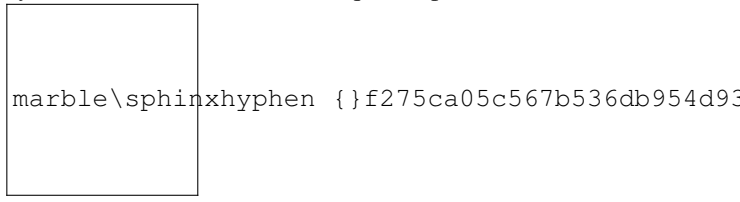
- **duration** (`Union[timedelta, float]`) – Duration for skipping elements from the end of the sequence.
- **scheduler** (`Optional[Scheduler]`) – Scheduler to use for time handling.

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

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

`rx.operators.skip_until(other)`

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

value. 

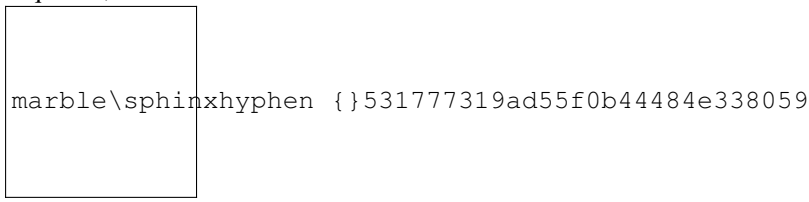
Parameters **other** (*Observable*) – The observable sequence that triggers propagation of elements of the source sequence.

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

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.

`rx.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.



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], Observable]

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

`rx.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 {}bf92f178aab29b95109adeb0c9c7b

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], Observable]

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.

`rx.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 {}bf92f178aab29b95109adeb0c9c7b

Example

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

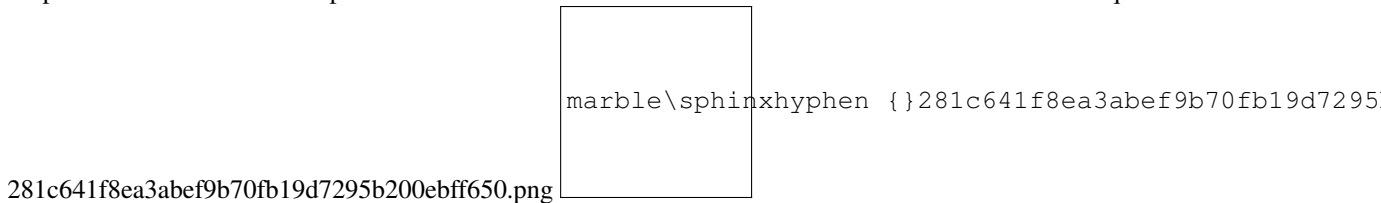
Parameters **predicate** (Callable[[~T1, int], bool]) – A function to test each element for a condition; the second parameter of the function represents the index of the source element.

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

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.

```
rx.operators.skip_with_time(duration, scheduler=None)
```

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

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

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.** –

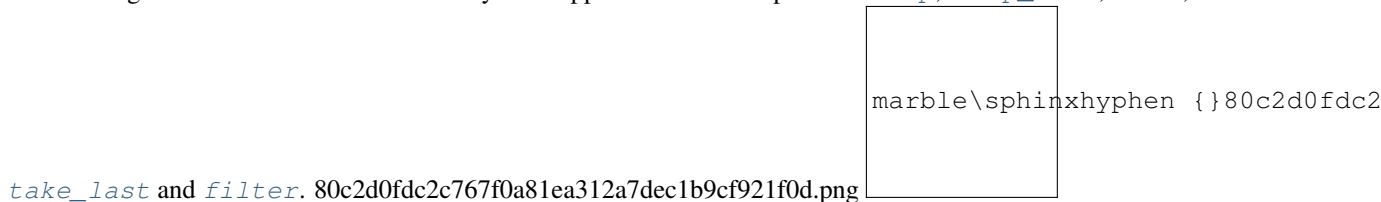
Return type `Callable[[Observable], Observable]`

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.

```
rx.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`,



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

Return type Callable[[*Observable*], *Observable*]

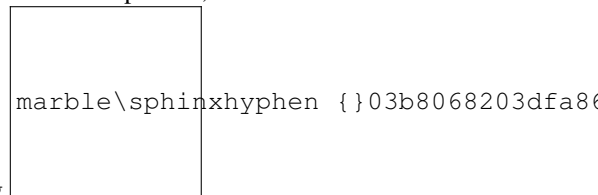
Returns An operator function that takes an observable source and returns a sliced observable sequence.

`rx.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



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*], *Observable*]

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.

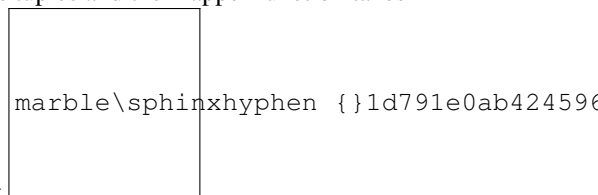
`rx.operators.starmap (mapper=None)`

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 arguments to the mapper is grouped as tuples and the mapper function takes

multiple arguments. 1d791e0ab4245961cd740c268a2a92fb24396ecf.png



Example

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

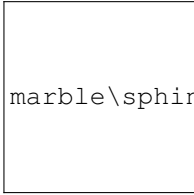
Parameters **mapper** (Optional[Callable[[~T1], ~T2]]) – A transform function to invoke with unpacked elements as arguments.

Return type Callable[[Observable], Observable]

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.

`rx.operators.starmap_indexed(mapper=None)`

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



marble\sphinxhyphen {}8239818b5b16fe0dd1257d653c34b42e0d963dbb.png

Example

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

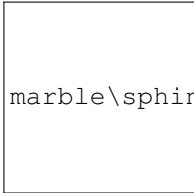
Parameters **mapper** (Optional[Callable[[~T1, int], ~T2]]) – A transform function to invoke with unpacked elements as arguments.

Return type Callable[[Observable], Observable]

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.

`rx.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], Observable]`

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

`rx.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** (*Scheduler*) – Scheduler to perform subscription and unsubscription actions on.

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

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

`rx.operators.sum(key_mapper=None)`

Computes the sum of a sequence of values that are obtained by invoking an optional transform function on each element of the input sequence, else if not specified computes the sum on each item in the sequence.

8ebbe43ade6e3245cea739bbd9cfd5697afa8bbd.png

marble\sphinxhyphen {}8ebbe43ade6e3245cea739bbd9cfd

Examples

```
>>> res = sum()
>>> res = sum(lambda x: x.value)
```

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

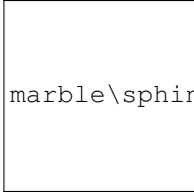
Return type `Callable[[Observable], Observable]`

Returns An operator function that takes a source observable and returns an observable sequence containing a single element with the sum of the values in the source sequence.

`rx.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



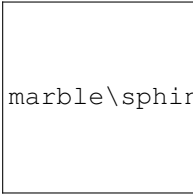
marble\sphinxhyphen {}b9329fb493471af771a2b890318e9a08d9bc0cfd.png

Return type Callable[[*Observable*], *Observable*]

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.

`rx.operators.take(count)`

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



marble\sphinxhyphen {}6e19e75a07bd399aae57e54367ef8
6e19e75a07bd399aae57e54367ef89cdf3f4abcb.png

Example

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

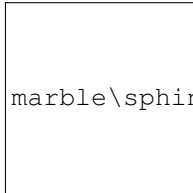
Parameters `count` (*int*) – The number of elements to return.

Return type Callable[[*Observable*], *Observable*]

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.

`rx.operators.take_last(count)`

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



marble\sphinxhyphen {}b90a41fff0fe1926f263bd7543265
b90a41fff0fe1926f263bd7543265b69545cd304.png

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*], *Observable*]

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.

`rx.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.

12b0f7a870f44770310e457d90a6a1e6c500aee2.png

marble\sphinxhyphen {}12b0f7a870f44770310e457d90a6a1e6c500aee2.png

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*], *Observable*]

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.

`rx.operators.take_last_with_time(duration, scheduler=None)`

Returns elements within the specified duration from the end of the observable source sequence.

d61e3083f22cf0310392c308c1ba7fbbe00d8c77.png

marble\sphinxhyphen {}d61e3083f22cf0310392c308c1ba7fbbe00d8c77.png

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], Observable]`

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.

`rx.operators.take_until` (*other*)

Returns the values from the source observable sequence until the other observable sequence produces a value.

A marble diagram showing a source sequence of marbles (circles) and a terminating sequence. The source sequence starts with a marble, followed by a gap, then another marble, and so on. The terminating sequence starts with a marble. The source sequence is cut off at the point where the terminating sequence starts. The diagram is labeled with a long hexadecimal string: f263d0fbb0af0ae1fdad8f9e5c3f9653fbf31f1d.png

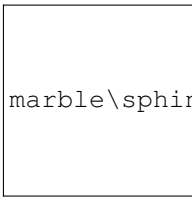
Parameters *other* (*Observable*) – Observable sequence that terminates propagation of elements of the source sequence.

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

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.

`rx.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.

A marble diagram showing a source sequence of marbles and a time-based termination. The source sequence starts with a marble, followed by a gap, then another marble, and so on. A vertical line indicates a specific end time. The source sequence is cut off at this time. The diagram is labeled with a long hexadecimal string: e57345367efb65a9a452388af6bfadbbb979b43e.png

Examples

```
>>> res = take_until_with_time(dt, [optional scheduler])
>>> res = take_until_with_time(5.0, [optional scheduler])
```

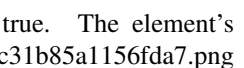
Parameters

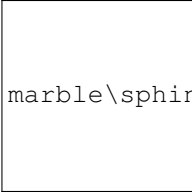
- **end_time** (`Union[datetime, timedelta, float]`) – Time to stop taking elements from the source sequence. If this value is less than or equal to `datetime.utcnow()`, the result stream will complete immediately.
- **scheduler** (`Optional[Scheduler]`) – Scheduler to run the timer on.

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

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

`rx.operators.take_while` (*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. A marble diagram showing a source sequence of marbles. The sequence is cut off at the first marble that does not satisfy the predicate function. The diagram is labeled with a long hexadecimal string: 7d8e208720017f8eb920b0a29c31b85a1156fda7.png



marble\sphinxhyphen {}7d8e208720017f8eb920b0a29c31b85a1156fda7.png

Example

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

Parameters

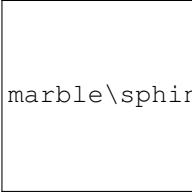
- **predicate** (Callable[[~T1], bool]) – A function to test each element for a condition.
- **inclusive** (bool) – [Optional] When set to True the value that caused the predicate function to return False will also be emitted. If not specified, defaults to False.

Return type Callable[[*Observable*], *Observable*]

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.

`rx.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 {}e150f5b604c299df12cc07f30c3c4d4816a44204.png

Example

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

Parameters

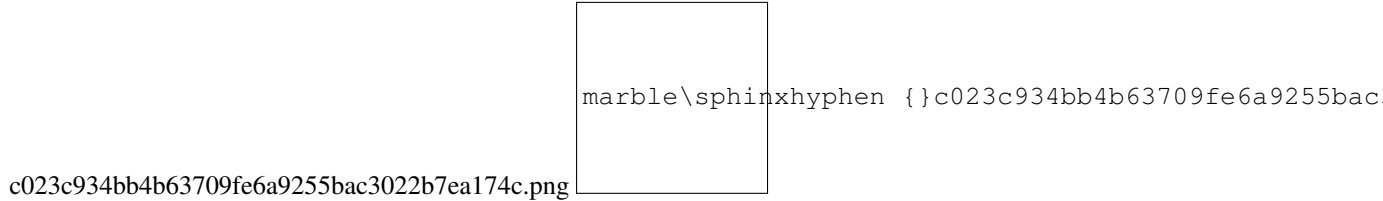
- **predicate** (Callable[[~T1, int], bool]) – A function to test each element for a condition; the second parameter of the function represents the index of the source element.
- **inclusive** (bool) – [Optional] When set to True the value that caused the predicate function to return False will also be emitted. If not specified, defaults to False.

Return type Callable[[*Observable*], *Observable*]

Returns An observable sequence that contains the elements from the input sequence that occur before the element at which the test no longer passes.

`rx.operators.take_with_time(duration, scheduler=None)`

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



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], Observable]`

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.

`rx.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], Observable]`

Returns An operator function that takes an observable source and returns an observable that performs the throttle operation.

`rx.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]`) – Mapper function to retrieve an
- **sequence indicating the throttle duration for each** (*observable*) –
- **element.** (*given*) –

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

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

```
rx.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*], *Observable*]

Returns A partially applied operator function that takes an observable source and returns an observable sequence with timestamp information on values.

```
rx.operators.timeout(duetime, other=None, scheduler=None)
```

Returns the source observable sequence or the other observable sequence if duetime elapses.

fc13941b8bf4aac2e5a18dbb00abe10e1a710e5e.png

marble\sphinxhyphen {}fc13941b8bf4aac2e5a18dbb00abe

Examples

```
>>> res = timeout(5.0)
>>> res = timeout(datetime(), return_value(42))
>>> res = timeout(5.0, return_value(42))
```

Parameters

- **duetime** (Union[datetime, 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*]) – Sequence to return in case of a timeout. If not specified, a timeout error throwing sequence will be used.
- **scheduler** (Optional[*Scheduler*]) –

Return type Callable[[*Observable*], *Observable*]

Returns An operator function that takes an observable source and returns the source sequence switching to the other sequence in case of a timeout.

```
rx.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(rx.timer(0.5))
>>> res = timeout_with_mapper(rx.timer(0.5), lambda x: rx.timer(0.2))
>>> res = timeout_with_mapper(rx.timer(0.5), lambda x: rx.timer(0.2)), rx.return_
↪value(42))
```

Parameters

- **first_timeout** (Optional[*Observable*]) – [Optional] Observable sequence that represents the timeout for the first element. If not provided, this defaults to rx.never().
- **timeout_duration_mapper** (Optional[Callable[[Any], *Observable*]]) – [Optional] Selector to retrieve an observable sequence that represents the timeout between the current element and the next element.
- **other** (Optional[*Observable*]) – [Optional] Sequence to return in case of a timeout. If not provided, this is set to rx.throw().

Return type Callable[[*Observable*], *Observable*]

Returns An operator function that takes an observable source and returns the source sequence switching to the other sequence in case of a timeout.

`rx.operators.time_interval(scheduler=None)`

Records the time interval between consecutive values in an observable sequence.

b0e3c487b07039af5b9dec98727bbaddf099f83d.png

marble\sphinxhyphen-{}b0e3c487b07039af5b9dec98727b

Examples

```
>>> res = time_interval()
```

Return type Callable[[*Observable*], *Observable*]

Returns An operator function that takes an observable source and returns an observable sequence with time interval information on values.

`rx.operators.to_dict(key_mapper, element_mapper=None)`

Converts the observable sequence to a Map if it exists.

Parameters

- **key_mapper** (Callable[[~T1], ~T2]) – A function which produces the key for the dictionary.
- **element_mapper** (Optional[Callable[[~T1], ~T2]]) – [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*], *Observable*]

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.

`rx.operators.to_future (future_ctor=None)`
Converts an existing observable sequence to a Future.

Example

```
op = to_future(asyncio.Future);
```

Parameters `future_ctor` (Optional[Callable[[], Future]]) – [Optional] The constructor of the future.

Return type Callable[[Observable], Future]

Returns An operator function that takes an observable source and returns a future with the last value from the observable sequence.

`rx.operators.to_iterable ()`
Creates an iterable from an observable sequence.

There is also an alias called `to_list`.

Return type Callable[[Observable], Observable]

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.

`rx.operators.to_list ()`
Creates an iterable from an observable sequence.

There is also an alias called `to_list`.

Return type Callable[[Observable], Observable]

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.

`rx.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[Scheduler]) – [Optional] The scheduler used to run the the input sequence on.

Return type Callable[[Observable], Observable]

Returns Observable stream.

`rx.operators.to_set ()`
Converts the observable sequence to a set.

Return type Callable[[Observable], Observable]

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.

`rx.operators.while_do (condition)`
Repeats source as long as condition holds emulating a while loop.

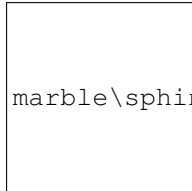
Parameters **condition** (Callable[[~T1], bool]) – The condition which determines if the source will be repeated.

Return type Callable[[*Observable*], *Observable*]

Returns An operator function that takes an observable source and returns an observable sequence which is repeated as long as the condition holds.

`rx.operators.window(boundaries)`

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



marble\sphinxhyphen-{}2c04b5c44b74070176595a3a4e68

2c04b5c44b74070176595a3a4e681bd1e8d6fcec.png

Examples

```
>>> res = window(rx.interval(1.0))
```

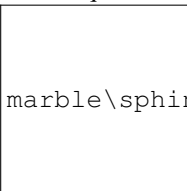
Parameters **boundaries** (*Observable*) – Observable sequence whose elements denote the creation and completion of non-overlapping windows.

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.window_when(closing_mapper)`

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



marble\sphinxhyphen-{}52e4fca35f1720c22c35083a9d2cd

52e4fca35f1720c22c35083a9d2cdc694a20aced.png

Examples

```
>>> res = window(lambda: rx.timer(0.5))
```

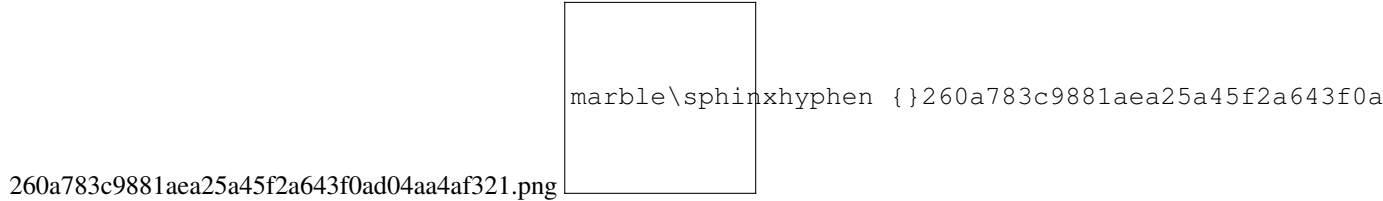
Parameters **closing_mapper** (Callable[[], *Observable*]) – A function invoked to define the closing of each produced window. It defines the boundaries of the produced windows (a window is started when the previous one is closed, resulting in non-overlapping windows).

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.window_toggle(openings, closing_mapper)`

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



```
>>> res = window(rx.interval(0.5), lambda i: rx.timer(i))
```

Parameters

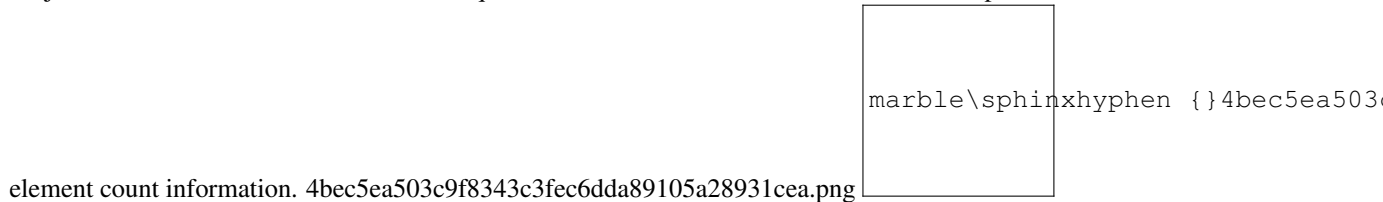
- **openings** (*Observable*) – Observable sequence whose elements denote the creation of windows.
- **closing_mapper** (Callable[[Any], *Observable*]) – A function invoked to define the closing of each produced window. Value from openings Observable that initiated the associated window is provided as argument to the function.

Return type Callable[[*Observable*], *Observable*]

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

`rx.operators.window_with_count(count, skip=None)`

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



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*], *Observable*]

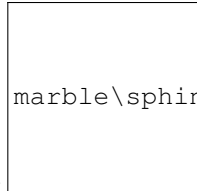
Returns An observable sequence of windows.

`rx.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 {}1c1361f657a91ebf5ade83dbd9

Examples

```
>>> op = with_latest_from(obs1)
>>> op = with_latest_from([obs1, obs2, obs3])
```

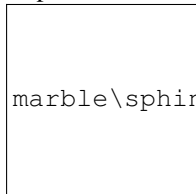
Return type Callable[[*Observable*], *Observable*]

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.

`rx.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.

8f017bb0b30f69529ca9935a37b7146a7bb53865.png



marble\sphinxhyphen {}8f017bb0b30f69529ca9935a37b7

Example

```
>>> res = zip(obs1, obs2)
```

Parameters `args` (*Observable*) – Observable sources to zip.

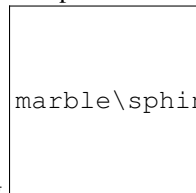
Return type Callable[[*Observable*], *Observable*]

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.

`rx.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.

45da698d9b059e9fb52b48e83e592327605a35d5.png



marble\sphinxhyphen {}45da698d9b059e9fb52b48e83e59

Example

```
>>> res = zip([1, 2, 3])
```

Parameters `second` (Iterable) – Iterable to zip with the source observable..

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

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.

`rx.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.

marble\spinxhyphen {}45da698d9b059e9fb52b48e83e59

45da698d9b059e9fb52b48e83e592327605a35d5.png

Example

```
>>> res = zip([1,2,3])
```

Parameters `second` (Iterable) – Iterable to zip with the source observable..

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

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.

7.6 Typing

class `rx.core.typing.Disposable`

Disposable abstract base class.

abstract `dispose()`

Dispose the object: stop whatever we're doing and release all of the resources we might be using.

Return type `None`

class `rx.core.typing.Scheduler`

Scheduler abstract base class.

abstract `property now`

Represents a notion of time for this scheduler. Tasks being scheduled on a scheduler will adhere to the time denoted by this property.

Return type `datetime`

Returns The scheduler's current time, as a datetime instance.

abstract `schedule(action, state=None)`

Schedules an action to be executed.

Parameters

- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type `Disposable`

Returns The disposable object used to cancel the scheduled action (best effort).

abstract schedule_relative (*duetime*, *action*, *state=None*)

Schedules an action to be executed after duetime.

Parameters

- **duetime** (Union[timedelta, float]) – Relative time after which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

abstract schedule_absolute (*duetime*, *action*, *state=None*)

Schedules an action to be executed at duetime.

Parameters

- **duetime** (Union[datetime, float]) – Absolute time at which to execute the action.
- **action** (Callable[[*Scheduler*, Optional[~TState]], Optional[*Disposable*]]) – Action to be executed.
- **state** (Optional[~TState]) – [Optional] state to be given to the action function.

Return type *Disposable*

Returns The disposable object used to cancel the scheduled action (best effort).

abstract classmethod to_seconds (*value*)

Converts time value to seconds. This method handles both absolute (datetime) and relative (timedelta) values. If the argument is already a float, it is simply returned unchanged.

Parameters **value** (Union[datetime, timedelta, float]) – the time value to convert to seconds.

Return type float

Returns The value converted to seconds.

abstract classmethod to_datetime (*value*)

Converts time value to datetime. This method handles both absolute (float) and relative (timedelta) values. If the argument is already a datetime, it is simply returned unchanged.

Parameters **value** (Union[datetime, timedelta, float]) – the time value to convert to datetime.

Return type datetime

Returns The value converted to datetime.

abstract classmethod to_timedelta (*value*)

Converts time value to timedelta. This method handles both absolute (datetime) and relative (float) values. If the argument is already a timedelta, it is simply returned unchanged. If the argument is an absolute time, the result value will be the timedelta since the epoch, January 1st, 1970, 00:00:00.

Parameters **value** (Union[datetime, timedelta, float]) – the time value to convert to timedelta.

Return type timedelta

Returns The value converted to timedelta.

class `rx.core.typing.PeriodicScheduler`

PeriodicScheduler abstract base class.

abstract `schedule_periodic` (*period*, *action*, *state=None*)

Schedules a periodic piece of work.

Parameters

- **period** (`Union[timedelta, float]`) – Period in seconds or timedelta for running the work periodically.
- **action** (`Callable[[Optional[~TState]], Optional[~TState]]`) – Action to be executed.
- **state** (`Optional[~TState]`) – [Optional] Initial state passed to the action upon the first iteration.

Return type `Disposable`

Returns The disposable object used to cancel the scheduled recurring action (best effort).

class `rx.core.typing.Observer` (**args*, ***kws*)

Observer abstract base class

An Observer is the entity that receives all emissions of a subscribed Observable.

abstract `on_next` (*value*)

Notifies the observer of a new element in the sequence.

Parameters **value** (*-T_in*) – The received element.

Return type `None`

abstract `on_error` (*error*)

Notifies the observer that an exception has occurred.

Parameters **error** (`Exception`) – The error that has occurred.

Return type `None`

abstract `on_completed` ()

Notifies the observer of the end of the sequence.

Return type `None`

class `rx.core.typing.Observable` (**args*, ***kws*)

Observable abstract base class.

Represents a push-style collection.

abstract `subscribe` (*observer=None*, ***, *scheduler=None*)

Subscribe an observer to the observable sequence.

Parameters

- **observer** (`Optional[Observer[+T_out]]`) – [Optional] The object that is to receive notifications.
- **scheduler** (`Optional[Scheduler]`) – [Optional] The default scheduler to use for this subscription.

Return type `Disposable`

Returns Disposable object representing an observer's subscription to the observable sequence.

class rx.core.typing.Subject (*args, **kws)

Subject abstract base class.

Represents an object that is both an observable sequence as well as an observer.

abstract subscribe (observer=None, *, scheduler=None)

Subscribe an observer to the observable sequence.

Parameters

- **observer** (Optional[Observer[+T_out]]) – [Optional] The object that is to receive notifications.
- **scheduler** (Optional[Scheduler]) – [Optional] The default scheduler to use for this subscription.

Return type Disposable

Returns Disposable object representing an observer’s subscription to the observable sequence.

abstract on_next (value)

Notifies the observer of a new element in the sequence.

Parameters value (-T_in) – The received element.

Return type None

abstract on_error (error)

Notifies the observer that an exception has occurred.

Parameters error (Exception) – The error that has occurred.

Return type None

abstract on_completed ()

Notifies the observer of the end of the sequence.

Return type None

CONTRIBUTING

You can contribute by reviewing and sending feedback on code checkins, suggesting and trying out new features as they are implemented, register issues and help us verify fixes as they are checked in, as well as submit code fixes or code contributions of your own.

The main repository is at [ReactiveX/RxPY](#). Please register any issues to [ReactiveX/RxPY/issues](#).

Please submit any pull requests against the [master](#) branch.

THE MIT LICENSE

Copyright 2013-2019, Dag Brattli, Microsoft Corp., and Contributors.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PYTHON MODULE INDEX

r

- `rx`, [25](#)
- `rx.core.typing`, [119](#)
- `rx.operators`, [67](#)
- `rx.scheduler`, [47](#)
- `rx.scheduler.eventloop`, [56](#)
- `rx.scheduler.mainloop`, [61](#)

Symbols

[__add__\(\) \(rx.Observable method\), 44](#)
[__await__\(\) \(rx.Observable method\), 44](#)
[__getitem__\(\) \(rx.Observable method\), 45](#)
[__iadd__\(\) \(rx.Observable method\), 44](#)
[__init__\(\) \(rx.Observable method\), 42](#)
[__init__\(\) \(rx.scheduler.CatchScheduler method\), 47](#)
[__init__\(\) \(rx.scheduler.CurrentThreadScheduler method\), 48](#)
[__init__\(\) \(rx.scheduler.EventLoopScheduler method\), 48](#)
[__init__\(\) \(rx.scheduler.HistoricalScheduler method\), 50](#)
[__init__\(\) \(rx.scheduler.NewThreadScheduler method\), 51](#)
[__init__\(\) \(rx.scheduler.ThreadPoolScheduler method\), 52](#)
[__init__\(\) \(rx.scheduler.ThreadPoolScheduler.ThreadPoolScheduler method\), 52](#)
[__init__\(\) \(rx.scheduler.TrampolineScheduler method\), 53](#)
[__init__\(\) \(rx.scheduler.VirtualTimeScheduler method\), 54](#)
[__init__\(\) \(rx.scheduler.eventloop.AsyncIOScheduler method\), 56](#)
[__init__\(\) \(rx.scheduler.eventloop.AsyncIOThreadSafeScheduler method\), 57](#)
[__init__\(\) \(rx.scheduler.eventloop.EventletScheduler method\), 57](#)
[__init__\(\) \(rx.scheduler.eventloop.GEventScheduler method\), 58](#)
[__init__\(\) \(rx.scheduler.eventloop.IOLoopScheduler method\), 59](#)
[__init__\(\) \(rx.scheduler.eventloop.TwistedScheduler method\), 60](#)
[__init__\(\) \(rx.scheduler.mainloop.GtkScheduler method\), 61](#)
[__init__\(\) \(rx.scheduler.mainloop.PyGameScheduler method\), 63](#)
[__init__\(\) \(rx.scheduler.mainloop.QtScheduler method\), 64](#)
[__init__\(\) \(rx.scheduler.mainloop.TkinterScheduler](#)

[method\), 62](#)
[__init__\(\) \(rx.scheduler.mainloop.WxScheduler method\), 65](#)
[__init__\(\) \(rx.subject.AsyncSubject method\), 46](#)
[__init__\(\) \(rx.subject.BehaviorSubject method\), 46](#)
[__init__\(\) \(rx.subject.ReplaySubject method\), 46](#)
[__init__\(\) \(rx.subject.Subject method\), 45](#)
[__new__\(\) \(rx.scheduler.ImmediateScheduler static method\), 50](#)
[__new__\(\) \(rx.scheduler.TimeoutScheduler static method\), 52](#)

A

[add\(\) \(rx.scheduler.HistoricalScheduler class method\), 50](#)
[add\(\) \(rx.scheduler.VirtualTimeScheduler class method\), 55](#)
[add_by\(\) \(rx.scheduler.VirtualTimeScheduler method\), 55](#)
[advance_to\(\) \(rx.scheduler.VirtualTimeScheduler method\), 55](#)
[all\(\) \(in module rx.operators\), 67](#)
[amb\(\) \(in module rx\), 25](#)
[amb\(\) \(in module rx.operators\), 67](#)
[as_observable\(\) \(in module rx.operators\), 67](#)
[AsyncIOScheduler \(class in rx.scheduler.eventloop\), 56](#)
[AsyncIOThreadSafeScheduler \(class in rx.scheduler.eventloop\), 57](#)
[AsyncSubject \(class in rx.subject\), 46](#)
[average\(\) \(in module rx.operators\), 67](#)

B

[BehaviorSubject \(class in rx.subject\), 46](#)
[buffer\(\) \(in module rx.operators\), 68](#)
[buffer_toggle\(\) \(in module rx.operators\), 69](#)
[buffer_when\(\) \(in module rx.operators\), 68](#)
[buffer_with_count\(\) \(in module rx.operators\), 69](#)
[buffer_with_time\(\) \(in module rx.operators\), 70](#)
[buffer_with_time_or_count\(\) \(in module rx.operators\), 70](#)

C

`cancel()` (*rx.scheduler.ThreadPoolScheduler.ThreadPoolThread* *rx.scheduler.eventloop*), 57
method), 52

`cancel_all()` (*rx.scheduler.mainloop.WxScheduler*
method), 65

`case()` (*in module rx*), 25

`catch()` (*in module rx*), 26

`catch()` (*in module rx.operators*), 71

`catch_with_iterable()` (*in module rx*), 26

`CatchScheduler` (*class in rx.scheduler*), 47

`cold()` (*in module rx*), 32

`combine_latest()` (*in module rx*), 27

`combine_latest()` (*in module rx.operators*), 71

`concat()` (*in module rx*), 27

`concat()` (*in module rx.operators*), 72

`concat_with_iterable()` (*in module rx*), 28

`contains()` (*in module rx.operators*), 72

`count()` (*in module rx.operators*), 73

`create()` (*in module rx*), 27

`CurrentThreadScheduler` (*class in rx.scheduler*),
48

D

`debounce()` (*in module rx.operators*), 73

`default_if_empty()` (*in module rx.operators*), 74

`defer()` (*in module rx*), 28

`delay()` (*in module rx.operators*), 76

`delay_subscription()` (*in module rx.operators*),
75

`delay_with_mapper()` (*in module rx.operators*), 75

`dematerialize()` (*in module rx.operators*), 76

`Disposable` (*class in rx.core.typing*), 119

`dispose()` (*rx.core.typing.Disposable* method), 119

`dispose()` (*rx.scheduler.EventLoopScheduler*
method), 49

`dispose()` (*rx.subject.AsyncSubject* method), 46

`dispose()` (*rx.subject.BehaviorSubject* method), 46

`dispose()` (*rx.subject.ReplaySubject* method), 46

`dispose()` (*rx.subject.Subject* method), 46

`distinct()` (*in module rx.operators*), 76

`distinct_until_changed()` (*in module*
rx.operators), 77

`do()` (*in module rx.operators*), 78

`do_action()` (*in module rx.operators*), 78

`do_while()` (*in module rx.operators*), 79

E

`element_at()` (*in module rx.operators*), 79

`element_at_or_default()` (*in module*
rx.operators), 79

`empty()` (*in module rx*), 29

`ensure_trampoline()`
(*rx.scheduler.TrampolineScheduler* method),
54

`EventletScheduler` (*class in*

rx.scheduler.eventloop), 57

`EventLoopScheduler` (*class in rx.scheduler*), 48

`exclusive()` (*in module rx.operators*), 80

`expand()` (*in module rx.operators*), 80

F

`filter()` (*in module rx.operators*), 80

`filter_indexed()` (*in module rx.operators*), 81

`finally_action()` (*in module rx.operators*), 81

`find()` (*in module rx.operators*), 82

`find_index()` (*in module rx.operators*), 82

`first()` (*in module rx.operators*), 82

`first_or_default()` (*in module rx.operators*), 83

`flat_map()` (*in module rx.operators*), 83

`flat_map_indexed()` (*in module rx.operators*), 84

`flat_map_latest()` (*in module rx.operators*), 85

`for_in()` (*in module rx*), 29

`from_()` (*in module rx*), 31

`from_callable()` (*in module rx*), 29

`from_callback()` (*in module rx*), 30

`from_future()` (*in module rx*), 30

`from_iterable()` (*in module rx*), 31

`from_list()` (*in module rx*), 31

`from_marbles()` (*in module rx*), 31

G

`generate()` (*in module rx*), 33

`generate_with_relative_time()` (*in module*
rx), 32

`GEventScheduler` (*class in rx.scheduler.eventloop*),
58

`group_by()` (*in module rx.operators*), 85

`group_by_until()` (*in module rx.operators*), 85

`group_join()` (*in module rx.operators*), 86

`GtkScheduler` (*class in rx.scheduler.mainloop*), 61

H

`HistoricalScheduler` (*class in rx.scheduler*), 49

`hot()` (*in module rx*), 33

I

`if_then()` (*in module rx*), 34

`ignore_elements()` (*in module rx.operators*), 87

`ImmediateScheduler` (*class in rx.scheduler*), 50

`interval()` (*in module rx*), 35

`IOLoopScheduler` (*class in rx.scheduler.eventloop*),
59

`is_empty()` (*in module rx.operators*), 87

J

`join()` (*in module rx.operators*), 87

`just()` (*in module rx*), 38

L

`last()` (in module `rx.operators`), 87
`last_or_default()` (in module `rx.operators`), 88

M

`map()` (in module `rx.operators`), 89
`map_indexed()` (in module `rx.operators`), 89
`materialize()` (in module `rx.operators`), 89
`max()` (in module `rx.operators`), 90
`max_by()` (in module `rx.operators`), 90
`merge()` (in module `rx`), 35
`merge()` (in module `rx.operators`), 90
`merge_all()` (in module `rx.operators`), 91
`min()` (in module `rx.operators`), 91
`min_by()` (in module `rx.operators`), 92
`module`
 `rx`, 25
 `rx.core.typing`, 119
 `rx.operators`, 67
 `rx.scheduler`, 47
 `rx.scheduler.eventloop`, 56
 `rx.scheduler.mainloop`, 61
`multicast()` (in module `rx.operators`), 92

N

`never()` (in module `rx`), 36
`NewThreadScheduler` (class in `rx.scheduler`), 51
`now()` (`rx.core.typing.Scheduler` property), 119
`now()` (`rx.scheduler.CatchScheduler` property), 47
`now()` (`rx.scheduler.eventloop.AsyncIOScheduler` property), 56
`now()` (`rx.scheduler.eventloop.EventletScheduler` property), 58
`now()` (`rx.scheduler.eventloop.GEventScheduler` property), 59
`now()` (`rx.scheduler.eventloop.IOLoopScheduler` property), 60
`now()` (`rx.scheduler.eventloop.TwistedScheduler` property), 61
`now()` (`rx.scheduler.HistoricalScheduler` property), 50
`now()` (`rx.scheduler.VirtualTimeScheduler` property), 54

O

`Observable` (class in `rx`), 42
`Observable` (class in `rx.core.typing`), 121
`observe_on()` (in module `rx.operators`), 93
`Observer` (class in `rx.core.typing`), 121
`of()` (in module `rx`), 36
`on_completed()` (`rx.core.typing.Observer` method), 121
`on_completed()` (`rx.core.typing.Subject` method), 122
`on_completed()` (`rx.subject.Subject` method), 46

`on_error()` (`rx.core.typing.Observer` method), 121
`on_error()` (`rx.core.typing.Subject` method), 122
`on_error()` (`rx.subject.Subject` method), 45
`on_error_resume_next()` (in module `rx`), 36
`on_error_resume_next()` (in module `rx.operators`), 93
`on_next()` (`rx.core.typing.Observer` method), 121
`on_next()` (`rx.core.typing.Subject` method), 122
`on_next()` (`rx.subject.Subject` method), 45

P

`pairwise()` (in module `rx.operators`), 93
`partition()` (in module `rx.operators`), 94
`partition_indexed()` (in module `rx.operators`), 94
`PeriodicScheduler` (class in `rx.core.typing`), 121
`pipe()` (`rx.Observable` method), 43
`pluck()` (in module `rx.operators`), 94
`pluck_attr()` (in module `rx.operators`), 95
`publish()` (in module `rx.operators`), 95
`publish_value()` (in module `rx.operators`), 95
`PyGameScheduler` (class in `rx.scheduler.mainloop`), 63

Q

`QtScheduler` (class in `rx.scheduler.mainloop`), 64

R

`range()` (in module `rx`), 37
`reduce()` (in module `rx.operators`), 96
`ref_count()` (in module `rx.operators`), 96
`repeat()` (in module `rx.operators`), 96
`repeat_value()` (in module `rx`), 38
`replay()` (in module `rx.operators`), 97
`ReplaySubject` (class in `rx.subject`), 46
`retry()` (in module `rx.operators`), 98
`return_value()` (in module `rx`), 37
`run()` (`rx.Observable` method), 44
`run()` (`rx.scheduler.EventLoopScheduler` method), 49
`rx`
 `module`, 25
`rx.core.typing`
 `module`, 119
`rx.operators`
 `module`, 67
`rx.scheduler`
 `module`, 47
`rx.scheduler.eventloop`
 `module`, 56
`rx.scheduler.mainloop`
 `module`, 61

S

`sample()` (in module `rx.operators`), 98
`scan()` (in module `rx.operators`), 98

```

schedule() (rx.core.typing.Scheduler method), 119
schedule() (rx.scheduler.CatchScheduler method), 47
schedule() (rx.scheduler.eventloop.AsyncIOScheduler
method), 56
schedule() (rx.scheduler.eventloop.AsyncIOThreadSafeScheduler
method), 57
schedule() (rx.scheduler.eventloop.EventletScheduler
method), 58
schedule() (rx.scheduler.eventloop.GEventScheduler
method), 59
schedule() (rx.scheduler.eventloop.IOLoopScheduler
method), 60
schedule() (rx.scheduler.eventloop.TwistedScheduler
method), 60
schedule() (rx.scheduler.EventLoopScheduler
method), 48
schedule() (rx.scheduler.ImmediateScheduler
method), 50
schedule() (rx.scheduler.mainloop.GtkScheduler
method), 61
schedule() (rx.scheduler.mainloop.PyGameScheduler
method), 63
schedule() (rx.scheduler.mainloop.QtScheduler
method), 64
schedule() (rx.scheduler.mainloop.TkinterScheduler
method), 63
schedule() (rx.scheduler.mainloop.WxScheduler
method), 65
schedule() (rx.scheduler.NewThreadScheduler
method), 51
schedule() (rx.scheduler.TimeoutScheduler method),
52
schedule() (rx.scheduler.TrampolineScheduler
method), 53
schedule() (rx.scheduler.VirtualTimeScheduler
method), 54
schedule_absolute() (rx.core.typing.Scheduler
method), 120
schedule_absolute()
(rx.scheduler.CatchScheduler method), 47
schedule_absolute()
(rx.scheduler.eventloop.AsyncIOScheduler
method), 56
schedule_absolute()
(rx.scheduler.eventloop.AsyncIOThreadSafeScheduler
method), 57
schedule_absolute()
(rx.scheduler.eventloop.EventletScheduler
method), 58
schedule_absolute()
(rx.scheduler.eventloop.GEventScheduler
method), 59
schedule_absolute()
(rx.scheduler.eventloop.IOLoopScheduler
method), 60
schedule_absolute()
(rx.scheduler.eventloop.TwistedScheduler
method), 61
schedule_absolute()
(rx.scheduler.EventLoopScheduler method), 49
schedule_absolute()
(rx.scheduler.ImmediateScheduler method), 51
schedule_absolute()
(rx.scheduler.mainloop.GtkScheduler method),
62
schedule_absolute()
(rx.scheduler.mainloop.PyGameScheduler
method), 64
schedule_absolute()
(rx.scheduler.mainloop.QtScheduler method),
65
schedule_absolute()
(rx.scheduler.mainloop.TkinterScheduler
method), 63
schedule_absolute()
(rx.scheduler.mainloop.WxScheduler method),
66
schedule_absolute()
(rx.scheduler.NewThreadScheduler method),
51
schedule_absolute()
(rx.scheduler.TimeoutScheduler method),
53
schedule_absolute()
(rx.scheduler.TrampolineScheduler method),
54
schedule_absolute()
(rx.scheduler.VirtualTimeScheduler method),
55
schedule_periodic()
(rx.core.typing.PeriodicScheduler method),
121
schedule_periodic()
(rx.scheduler.CatchScheduler method), 48
schedule_periodic()
(rx.scheduler.EventLoopScheduler method), 49
schedule_periodic()
(rx.scheduler.mainloop.GtkScheduler method),
62
schedule_periodic()
(rx.scheduler.mainloop.QtScheduler method),
65
schedule_periodic()
(rx.scheduler.mainloop.WxScheduler method),
66
schedule_periodic()
(rx.scheduler.NewThreadScheduler method),
52

```


`schedule_relative()` (*rx.core.typing.Scheduler* method), 120
`schedule_relative()` (*rx.scheduler.CatchScheduler* method), 47
`schedule_relative()` (*rx.scheduler.eventloop.AsyncIOScheduler* method), 56
`schedule_relative()` (*rx.scheduler.eventloop.AsyncIOThreadSafeScheduler* method), 57
`schedule_relative()` (*rx.scheduler.eventloop.EventletScheduler* method), 58
`schedule_relative()` (*rx.scheduler.eventloop.GEventScheduler* method), 59
`schedule_relative()` (*rx.scheduler.eventloop.IOLoopScheduler* method), 60
`schedule_relative()` (*rx.scheduler.eventloop.TwistedScheduler* method), 61
`schedule_relative()` (*rx.scheduler.EventLoopScheduler* method), 49
`schedule_relative()` (*rx.scheduler.ImmediateScheduler* method), 50
`schedule_relative()` (*rx.scheduler.mainloop.GtkScheduler* method), 62
`schedule_relative()` (*rx.scheduler.mainloop.PyGameScheduler* method), 64
`schedule_relative()` (*rx.scheduler.mainloop.QtScheduler* method), 64
`schedule_relative()` (*rx.scheduler.mainloop.TkinterScheduler* method), 63
`schedule_relative()` (*rx.scheduler.mainloop.WxScheduler* method), 66
`schedule_relative()` (*rx.scheduler.NewThreadScheduler* method), 51
`schedule_relative()` (*rx.scheduler.TimeoutScheduler* method), 52
`schedule_relative()` (*rx.scheduler.TrampolineScheduler* method), 53
`schedule_relative()` (*rx.scheduler.VirtualTimeScheduler* method), 54
`schedule_required()` (*rx.scheduler.TrampolineScheduler* method), 54
`sequence_equal()` (*in module rx.operators*), 99
`share()` (*in module rx.operators*), 100
`single()` (*in module rx.operators*), 100
`single_or_default()` (*in module rx.operators*), 100
`singleton()` (*rx.scheduler.CurrentThreadScheduler* class method), 48
`skip()` (*in module rx.operators*), 101
`skip_last()` (*in module rx.operators*), 101
`skip_last_with_time()` (*in module rx.operators*), 102
`skip_until()` (*in module rx.operators*), 102
`skip_until_with_time()` (*in module rx.operators*), 102
`skip_while()` (*in module rx.operators*), 103
`skip_while_indexed()` (*in module rx.operators*), 103
`skip_with_time()` (*in module rx.operators*), 104
`sleep()` (*rx.scheduler.VirtualTimeScheduler* method), 55
`slice()` (*in module rx.operators*), 104
`some()` (*in module rx.operators*), 105
`starmap()` (*in module rx.operators*), 105
`starmap_indexed()` (*in module rx.operators*), 106
`start()` (*in module rx*), 38
`start()` (*rx.scheduler.ThreadPoolScheduler.ThreadPoolThread* method), 52
`start()` (*rx.scheduler.VirtualTimeScheduler* method), 55
`start_async()` (*in module rx*), 39
`start_with()` (*in module rx.operators*), 106
`stop()` (*rx.scheduler.VirtualTimeScheduler* method), 55
`Subject` (*class in rx.core.typing*), 121
`Subject` (*class in rx.subject*), 45
`subscribe()` (*rx.core.typing.Observable* method), 121
`subscribe()` (*rx.core.typing.Subject* method), 122
`subscribe()` (*rx.Observable* method), 42
`subscribe_()` (*rx.Observable* method), 43
`subscribe_on()` (*in module rx.operators*), 107
`sum()` (*in module rx.operators*), 107
`switch_latest()` (*in module rx.operators*), 107

T

`take()` (*in module rx.operators*), 108
`take_last()` (*in module rx.operators*), 108
`take_last_buffer()` (*in module rx.operators*), 109
`take_last_with_time()` (*in module rx.operators*), 109
`take_until()` (*in module rx.operators*), 110
`take_until_with_time()` (*in module rx.operators*), 110

`take_while()` (in module `rx.operators`), 110
`take_while_indexed()` (in module `rx.operators`), 111
`take_with_time()` (in module `rx.operators`), 111
`ThreadPoolScheduler` (class in `rx.scheduler`), 52
`ThreadPoolScheduler.ThreadPoolThread` (class in `rx.scheduler`), 52
`throttle_first()` (in module `rx.operators`), 112
`throttle_with_mapper()` (in module `rx.operators`), 112
`throttle_with_timeout()` (in module `rx.operators`), 74
`throw()` (in module `rx`), 39
`time_interval()` (in module `rx.operators`), 114
`timeout()` (in module `rx.operators`), 113
`timeout_with_mapper()` (in module `rx.operators`), 113
`TimeoutScheduler` (class in `rx.scheduler`), 52
`timer()` (in module `rx`), 39
`timestamp()` (in module `rx.operators`), 113
`TkinterScheduler` (class in `rx.scheduler.mainloop`), 62
`to_async()` (in module `rx`), 40
`to_datetime()` (`rx.core.typing.Scheduler` class method), 120
`to_dict()` (in module `rx.operators`), 114
`to_future()` (in module `rx.operators`), 115
`to_iterable()` (in module `rx.operators`), 115
`to_list()` (in module `rx.operators`), 115
`to_marbles()` (in module `rx.operators`), 115
`to_seconds()` (`rx.core.typing.Scheduler` class method), 120
`to_set()` (in module `rx.operators`), 115
`to_timedelta()` (`rx.core.typing.Scheduler` class method), 120
`TrampolineScheduler` (class in `rx.scheduler`), 53
`TwistedScheduler` (class in `rx.scheduler.eventloop`), 60

U

`using()` (in module `rx`), 41

V

`VirtualTimeScheduler` (class in `rx.scheduler`), 54

W

`while_do()` (in module `rx.operators`), 115
`window()` (in module `rx.operators`), 116
`window_toggle()` (in module `rx.operators`), 116
`window_when()` (in module `rx.operators`), 116
`window_with_count()` (in module `rx.operators`), 117
`with_latest_from()` (in module `rx`), 41
`with_latest_from()` (in module `rx.operators`), 117

`WxScheduler` (class in `rx.scheduler.mainloop`), 65

Z

`zip()` (in module `rx`), 41
`zip()` (in module `rx.operators`), 118
`zip_with_iterable()` (in module `rx.operators`), 118
`zip_with_list()` (in module `rx.operators`), 119