reactivex Documentation

Release 4.0.4.post32.dev0

Dag Brattli <dag@brattli.net>

Mar 04, 2023

CONTENTS

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

CHAPTER

ONE

INSTALLATION

ReactiveX for Python (RxPY) v4.x runs on Python 3. To install:

```
pip3 install reactivex
```

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
```

CHAPTER

TWO

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.

CHAPTER

THREE

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 reactivex 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 {0}".format(i)),
    on_error = lambda e: print("Error Occurred: {0}".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 reactivex import of

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

source.subscribe(
    on_next = lambda i: print("Received {0}".format(i)),
    on_error = lambda e: print("Error Occurred: {0}".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 reactivex import of

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

source.subscribe(lambda value: print("Received {0}".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 reactivex 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 {0}".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 reactivex 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 {0}".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 reactivex
from reactivex import operators as ops

def length_more_than_5():
    # In v4 rx.pipe has been renamed to `compose`
    return reactivex.compose(
        ops.map(lambda s: len(s)),
        ops.filter(lambda i: i >= 5),
    )

reactivex.of("Alpha", "Beta", "Gamma", "Delta", "Epsilon").pipe(
    length_more_than_5()
).subscribe(lambda value: print("Received {0}".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 reactivex

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

            return source.subscribe(
```

(continues on next page)

(continued from previous page)

```
                on_next,
                observer.on_error,
                observer.on_completed,
                scheduler=scheduler)
        return reactivex.create(subscribe)
    return _lowercase

reactivex.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
```

(continues on next page)

(continued from previous page)

```
from threading import current_thread

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


def intense_calculation(value):
    # 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
reactivex.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
reactivex.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
reactivex.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 Enter key to exit\n")
```

OUTPUT:

```
Press Enter key to exit
PROCESS 1: Thread-1 Alpha
```

(continues on next page)

(continued from previous page)

```
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 reactivex
import reactivex.operators as ops
from reactivex.subject import Subject
from reactivex.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()
```

(continues on next page)

(continued from previous page)

```
        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.create_task(server)

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

    return reactivex.create(on_subscribe)


loop = asyncio.new_event_loop()
proxy = Subject()
source = tcp_server(proxy, loop)
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.

CHAPTER

FOUR

TESTING

Using the tools provided in *reactivex.testing*, it is possible to create tests for your own observables, custom operators and subscriptions.

Additionally, tests can be used to help understand the behaviors of existing operators.

4.1 Basic example

```
# This assumes that you are using pytest but unittest or others would work just as well
# Import the testing tools
from reactivex.testing import ReactiveTest, TestScheduler
from reactivex import operators

def test_double():
    # Create a scheduler
    scheduler = TestScheduler()
    # Define one or more source
    source = scheduler.create_hot_observable(
        ReactiveTest.on_next(250, 3),
        ReactiveTest.on_next(350, 5),
    )

    # Define how the observable/operator is used on the source
    def create():
        return source.pipe(operators.map(lambda x: 2 * x))

    # trigger subscription and record emissions
    results = scheduler.start(create)

    # check the messages and potentially subscriptions
    assert results.messages == [
        ReactiveTest.on_next(250, 6),
        ReactiveTest.on_next(350, 10),
    ]
```

4.2 Testing a custom operator

Whether your custom operator is created using a *composition* of operators or with full control, you can easily test various situations and combinations

Surprised about the timestamps (@500, @600, ...) for the result messages? Then read below about the timeline.

4.3 Timeline

When `scheduler.start` is called, the test scheduler starts moving its virtual clock forward. Some important timestamps are however hidden as defaults, as listed below. These values can be modified using *kwargs* in the `scheduler.start(...)` call:

1. `created` [100]: When is the observable created. That is when the `create` function seen in the basic example is called.
2. `subscribed` [200]: When does the subscription occur. This explains the above emission timestamps: consider the first emission @500; given that we are using a cold observable, and subscribe to it at 200, the *source*'s timeline starts at 200 and only 300 ticks later, it emits.
3. `disposed` [1000]: When the subscription is disposed

Gotchas when modifying these values:

1. Do not use *0* as values for created/subscribed since the code would ignore it.
2. If you change `subscribed` to be lower than 100, you need to change `created` as well, otherwise nothing will happen.

4.4 An alternative using marbles

As we saw in the previous section, we can use *reactivex.from_marbles* to create observables for our tests.

An example of using *to_marbles* for the assertion is shown in *test_hot*

There is a simplified flow available in *reactivex.testing.marbles* and here's an example:

```
def test_start_with():
    from reactivex.testing.marbles import marbles_testing
    with marbles_testing() as (start, cold, hot, exp):
        source = cold('------1-2-3-|')
        outcome = exp('a-----1-2-3-|', {"a": None})  # can use lookups if needed
        obs = source.pipe(
            operators.start_with(None)
        )
        # Note that start accepts the observable directly,
        # without the need for a "create" function
        results = start(obs)

        assert results == outcome
```

This method makes for very quick to write, and easy to read, tests. At this moment however, it does not allow for testing subscriptions.

4.5 Testing an observable factory

An observable created directly from *Observable* can be just as easily tested.

In this example, we will additionally test a case where a `Disposable` is used.

```
def test_my_observable_factory():
    from reactivex.disposable import Disposable, CompositeDisposable
    a = 42
    def factory(observer: Observer, scheduler=None):
        def increment():
            nonlocal a
            a += 1
        sub = Disposable(action=increment)
        return CompositeDisposable(
            sub,
            reactivex.timer(20, scheduler=scheduler).subscribe(observer)
        )

    scheduler = TestScheduler()
    result = scheduler.start(lambda: Observable(factory))
    assert result.messages == [
        on_next(220, 0),
        on_completed(220)
    ]
    assert a == 43  # shows that our Disposable's action was as expected
```

4.6 Testing errors

Going back to the *in_sequence_or_throw* operator, we did not test the error case; Let's remedy that below.

```
def test_in_sequence_or_throw_error():
    scheduler = TestScheduler()
    source = reactivex.from_marbles('--1-4-3-', timespan=50, scheduler=scheduler)
    result = scheduler.start(lambda: source.pipe(
        in_sequence_or_throw(),
    ), created=1, subscribed=30)

    assert result.messages == [
        on_next(30+100, 1),
        on_error(230, ValueError('Sequence error'))
    ]
    # At times it's better not to test the exact exception,
    # maybe its message changes with time or other reasons
    # We can test a specific notification's details as follows:
    first_notification, error_notification = result.messages
    assert first_notification.time == 130
    assert error_notification.time == 230
    assert first_notification.value.kind == 'N'  # Notification
    assert error_notification.value.kind == 'E'  # E for errors
    assert first_notification.value.value == 1
```

(continues on next page)

(continued from previous page)

```
    assert type(error_notification.value.exception) == ValueError  # look at .exception␣
↪for errors
```

4.7 Testing subscriptions, multiple observables, hot observables

`scheduler.start` only allows for a single subscription. Some cases like e.g. `operators.partition` require more. The examples below showcase some less commonly needed testing tools.

```
def test_multiple():
    scheduler = TestScheduler()
    source = reactivex.from_marbles('-1-4-3-|', timespan=50, scheduler=scheduler)
    odd, even = source.pipe(
        operators.partition(lambda x: x % 2),
    )
    steven = scheduler.create_observer()
    todd = scheduler.create_observer()

    even.subscribe(steven)
    odd.subscribe(todd)

    # Note! Since the subscription is not created within
    # `scheduler.start` below, the usual `subscribed` delay of t=200
    # is not in effect. The subscriptions therefore occur at t=0
    scheduler.start()

    assert steven.messages == [
        on_next(150, 4),
        on_completed(350)
    ]
    assert todd.messages == [
        on_next(50, 1),
        on_next(250, 3),
        on_completed(350)
    ]
```

```
from reactivex.testing.subscription import Subscription
def test_subscriptions():
    scheduler = TestScheduler()
    source = scheduler.create_cold_observable()  # "infinite"
    subs = []
    shared = source.pipe(
        operators.share()
    )
    # Creating our story:
    # first sub is set to occur at t=200; this creates a sub on source
    scheduler.schedule_relative(200, lambda *_: subs.append(shared.
↪subscribe(scheduler=scheduler)))
    # second sub does not create a new sub on source, due to the `share` operator
    scheduler.schedule_relative(300, lambda *_: subs.append(shared.
↪subscribe(scheduler=scheduler)))
```

(continues on next page)

(continued from previous page)

```
    # second sub ends
    scheduler.schedule_relative(500, lambda *_: subs[1].dispose())
    # first sub ends... and since there is no sub remaining, the only sub on source␣
↪should be disposed too
    scheduler.schedule_relative(600, lambda *_: subs[0].dispose())
    # no existing sub on source, therefore this will create a new one
    # we never dispose of it; we will test that infinite sub in the assertions
    scheduler.schedule_relative(900, lambda *_: subs.append(shared.
↪subscribe(scheduler=scheduler)))

    scheduler.start()
    # Check that the submissions on the source are as expected
    assert source.subscriptions == [
        Subscription(200, 600), # only one sub from 200 to 600
        Subscription(900),  # represents an infinite subscription
    ]
```

```
def test_hot():
    scheduler = TestScheduler()
    # hot starts at 0 but sub starts at 200 so we'll miss 190
    source = scheduler.create_hot_observable(
        on_next(190, 5),
        on_next(300, 42),
        on_completed(500)
    )
    result = scheduler.start(lambda: source.pipe(
        operators.to_marbles(timespan=20, scheduler=scheduler)
    ))

    message = result.messages[0]
    # the subscription starts at 200;
    # since `source` is a hot observable, the notification @190 will not be caught
    # the next notification is at 300 ticks,
    # which, on our subscription, will show at 100 ticks (300-200 from subscription␣
↪delay)
    # or 5 "-" each representing 20 ticks (timespan=20 in `to_marbles`).
    # Then the "42" notification is received
    # and then nothing for another 200 ticks, which is equal to 10 "-", before complete
    assert message.value.value == '-----(42)----------|'
```

CHAPTER

FIVE

MIGRATION V4

ReactiveX for Python v4 is an evolution of RxPY v3 to modernize it to current Python standards:

- Project main module renamed from `rx` to `reactivex`. This is done to give it a unique name different from the obsolete Reactive Extensions (RxPY)
- Generic type annotations. Code now type checks with pyright / pylance at strict settings. It also mostly type checks with mypy. Mypy should eventually catch up.
- The `pipe` function has been renamed to `compose`. There is now a new function `pipe` that works similar to the `pipe` method.
- RxPY is now a modern Python project using `pyproject.toml` instead of `setup.py`, and using modern tools such as Poetry, Black formatter and isort.

```
import reactivex as rx
from reactivex 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)))
```

CHAPTER

SIX

MIGRATION V3

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.

6.1 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.

6.2 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)
```

6.3 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.

6.4 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)
```

6.5 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()

6.6 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*.

6.7 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)
```

6.8 Packages Renamed

Some packages were renamed:

Old name	New name
rx.concurrency	*reactivex.scheduler*
rx.disposables	*rx.disposable*
rx.subjects	*rx.subject*

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

CHAPTER

SEVEN

OPERATORS

7.1 Creating Observables

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

7.2 Transforming Observables

7.3 Filtering Observables

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

7.4 Combining Observables

Operator	Description
combine_latest	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.
join	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.
merge	Combine multiple Observables into one by merging their emissions.
start_with	Emit a specified sequence of items before beginning to emit the items from the source Observable.
switch_latest	Convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables.
zip	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.
fork_join	Wait for Observables to complete and then combine last values they emitted into a tuple.

7.5 Error Handling

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

7.6 Utility Operators

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

7.7 Conditional and Boolean Operators

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

7.8 Mathematical and Aggregate Operators

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

7.9 Connectable Observable Operators

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

CHAPTER

EIGHT

ADDITIONAL READING

8.1 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

8.2 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.

CHAPTER

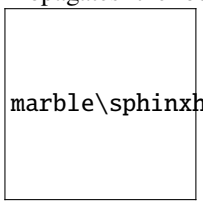
NINE

REFERENCE

9.1 Observable Factory

reactivex.**amb**(**sources*)

Propagates the observable sequence that emits first. f87bbff3e9ad64d121cc9d2ab3d1aacd4d9015c8.png



Example

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

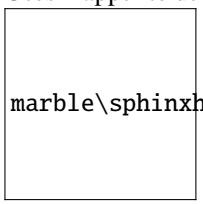
Parameters
sources (*Observable*[TypeVar(_T)]) – Sequence of observables to monitor for first emission.

Return type
Observable[TypeVar(_T)]

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

reactivex.**case**(*mapper*, *sources*, *default_source=None*)

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



Examples

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

Parameters

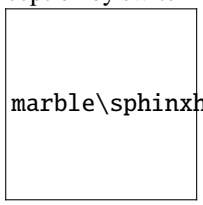
- **mapper** – The function which extracts the value for to test in a case statement.
- **sources** – An object which has keys which correspond to the case statement labels.
- **default_source** – [Optional] The observable sequence or Future that will be run if the sources are not matched. If this is not provided, it defaults to *empty()*.

Returns

An observable sequence which is determined by a case statement.

reactivex.**catch**(**sources*)

Continues observable sequences which are terminated with an exception by switching over to the next observable sequence. d14c02dddb2af945932be6a8f5b44425cde47a95.png



Examples

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

Parameters

sources (*Observable*[TypeVar(_T)]) – Sequence of observables.

Return type

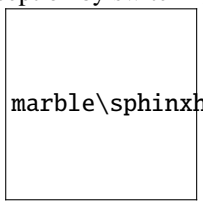
Observable[TypeVar(_T)]

Returns

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

reactivex.**catch_with_iterable**(*sources*)

Continues observable sequences that are terminated with an exception by switching over to the next observable sequence. d14c02dddb2af945932be6a8f5b44425cde47a95.png



Examples

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

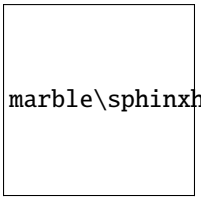
Parameters
sources (Iterable[*Observable*[TypeVar(_T)]]) – An Iterable of observables; thus, a generator can also be used here.

Return type
Observable[TypeVar(_T)]

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

reactivex.**create**(*subscribe*)

Creates an observable sequence object from the specified
subscription function.



aa0779a8a931350806ec603de7e4d48a00ab25d9.png

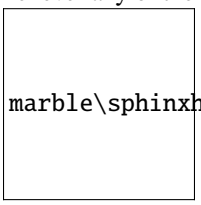
Parameters
subscribe (Callable[[ObserverBase[TypeVar(_T)], Optional[SchedulerBase]], DisposableBase]) – Subscription function.

Return type
Observable[TypeVar(_T)]

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

reactivex.**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 – Sequence of observables.

Return type
Observable[Any]

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

reactivex.**compose**(**operators*)

Compose multiple operators left to right.

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

Examples

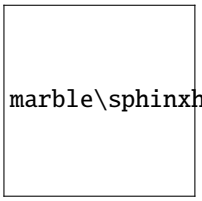
```
>>> pipe()(source) == source
>>> pipe(f)(source) == f(source)
>>> pipe(f, g)(source) == g(f(source))
>>> pipe(f, g, h)(source) == h(g(f(source)))
...
```

Return type
Callable[[Any], Any]

Returns
The composed observable.

reactivex.**concat**(**sources*)

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



marble\sphinxhyphen {}4544dd242d02df82ac059a82701a17b5b31135d4.png

Examples

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

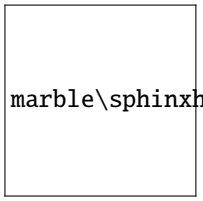
Parameters
sources (*Observable*[TypeVar(_T)]) – Sequence of observables.

Return type
Observable[TypeVar(_T)]

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

reactivex.**concat_with_iterable**(*sources*)

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



Examples

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

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

Return type
Observable[TypeVar(_T)]

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

class reactivex.**ConnectableObservable**(*source*, *subject*)

Represents an observable that can be connected and disconnected.

__init__(*source*, *subject*)

Creates an observable sequence object from the specified subscription function.

Parameters
subscribe – [Optional] Subscription function

connect(*scheduler=None*)

Connects the observable.

Return type
Optional[DisposableBase]

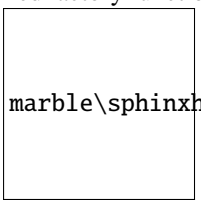
auto_connect(*subscriber_count=1*)

Returns an observable sequence that stays connected to the source indefinitely to the observable sequence. Providing a subscriber_count will cause it to connect() after that many subscriptions occur. A subscriber_count of 0 will result in emissions firing immediately without waiting for subscribers.

Return type
Observable[TypeVar(_T)]

reactivex.**defer**(*factory*)

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



e9205b73ee187ab559557e85051afd341c421477.png

Example

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

Parameters
factory – Observable factory function to invoke for each observer which invokes *subscribe()* on the resulting sequence. The factory takes a single argument, the scheduler used.

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

reactivex.**empty**(*scheduler=None*)

Returns an empty observable sequence. b1ce6593ad9d9760c7d62a8f13c6cf965bef6a3e.png

Example

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

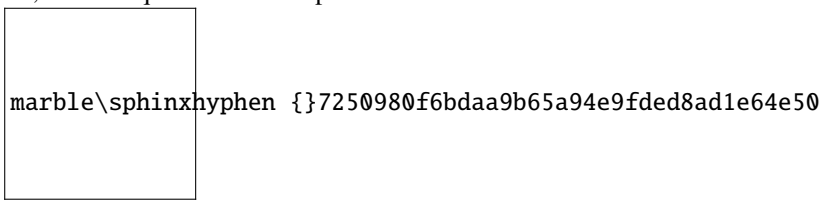
Parameters
scheduler (Optional[SchedulerBase]) – [Optional] Scheduler instance to send the termination call on. By default, this will use an instance of *ImmediateScheduler*.

Return type
Observable[Any]

Returns
An observable sequence with no elements.

reactivex.**fork_join**(**sources*)

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



Examples

```
>>> obs = reactivex.fork_join(obs1, obs2, obs3)
```

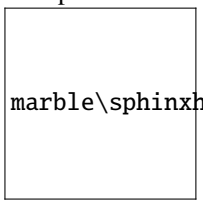
Parameters
sources (*Observable*[Any]) – Sequence of observables.

Return type
Observable[Any]

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

reactivex.**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



Examples

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

Parameters

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

Return type
Observable[TypeVar(_T)]

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

reactivex.**from_callback**(*func*, *mapper=None*)

Converts a callback function to an observable sequence.

Parameters

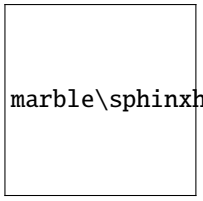
- **func** (Callable[..., Callable[..., None]]) – Function with a callback as the last argument to convert to an Observable sequence.
- **mapper** (Optional[Callable[[Any], Any]]) – [Optional] A mapper which takes the arguments from the callback to produce a single item to yield on next.

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

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.

reactivex.**from_future**(*future*)

Converts a Future to an Observable sequence b4a27b6a0e04aee0b97edc9b0f56546d34a0f3c3.png

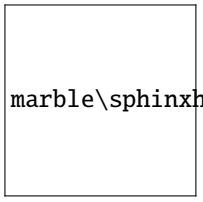


Parameters
future – A Python 3 compatible future. https://docs.python.org/3/library/asyncio-task.html#future

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

reactivex.**from_iterable**(*iterable*, *scheduler=None*)

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



Example

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

Parameters

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

Return type
Observable[TypeVar(_T)]

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

class reactivex.**GroupedObservable**(*key*, *underlying_observable*, *merged_disposable=None*)

__init__(*key*, *underlying_observable*, *merged_disposable=None*)

Creates an observable sequence object from the specified subscription function.

Parameters
subscribe – [Optional] Subscription function

reactivex.**never**()

Returns a non-terminating observable sequence, which can be used to denote an infinite duration (e.g. when using reactive joins). a66db381e146b21dc6913ea77287d2b73ff2a341.png 

Return type
Observable[Any]

Returns
An observable sequence whose observers will never get called.

class reactivex.**Notification**

Represents a notification to an observer.

__init__()

Default constructor used by derived types.

accept(*on_next*, *on_error=None*, *on_completed=None*)

Invokes the delegate corresponding to the notification or an observer and returns the produced result.

Examples

```
>>> notification.accept(observer)
>>> notification.accept(on_next, on_error, on_completed)
```

Parameters

- **on_next** (Union[Callable[[TypeVar(_T)], None], ObserverBase[TypeVar(_T)]]) – Delegate to invoke for an OnNext notification.
- **on_error** (Optional[Callable[[Exception], None]]) – [Optional] Delegate to invoke for an OnError notification.
- **on_completed** (Optional[Callable[[], None]]) – [Optional] Delegate to invoke for an OnCompleted notification.

Return type
None

Returns
Result produced by the observation.

to_observable(*scheduler=None*)

Returns an observable sequence with a single notification, using the specified scheduler, else the immediate scheduler.

Parameters
scheduler (Optional[SchedulerBase]) – [Optional] Scheduler to send out the notification calls on.

Return type
ObservableBase[TypeVar(_T)]

Returns
An observable sequence that surfaces the behavior of the notification upon subscription.

equals(*other*)

Indicates whether this instance and a specified object are equal.

Return type
bool

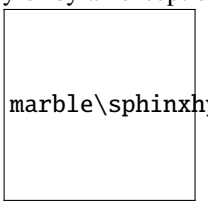
__eq__(*other*)

Return self==value.

Return type
bool

__hash__ = None

reactivex.**on_error_resume_next**(**sources*)

Continues an observable sequence that is terminated normally or by an exception with the next observable sequence. 8957c9ab4083b80d33e6535cfc81f21ade3c7bdb.png 

Examples

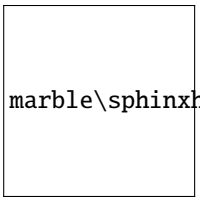
```
>>> res = reactivex.on_error_resume_next(xs, ys, zs)
```

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

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

reactivex.**of**(**args*)

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

a7133f2107f6ae9c75399b06e29d317fd8ce28d4.png 

Note: This is just a wrapper for *reactivex.from_iterable(args)*

Example

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

Parameters
args (TypeVar(_T)) – The variable number elements to emit from the observable.

Return type
Observable[TypeVar(_T)]

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

class reactivex.**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[[ObserverBase[TypeVar(_T_out, covariant=True)], Optional[SchedulerBase]], DisposableBase]]) – [Optional] Subscription function

subscribe(*on_next=None*, *on_error=None*, *on_completed=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** – [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** (Union[ObserverBase[TypeVar(_T_out, covariant=True)], Callable[[TypeVar(_T_out, covariant=True)], None], None]) – [Optional] Action to invoke for each element in the observable sequence.
- **scheduler** (Optional[SchedulerBase]) – [Optional] The default scheduler to use for this subscription.

Return type
DisposableBase

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

pipe(**operators*)

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[[Any], 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
Generator[Any, None, TypeVar(_T_out, covariant=True)]

Returns
The last item of the observable sequence.

__add__(*other*)

Pythonic version of *concat*.

Example

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

Parameters
other (*Observable*[TypeVar(_T_out, covariant=True)]) – The second observable sequence in the concatenation.

Return type
Observable[TypeVar(_T_out, covariant=True)]

Returns
Concatenated observable sequence.

__iadd__(*other*)

Pythonic use of *concat*.

Example

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

Parameters
other (*Observable*[TypeVar(_T_out, covariant=True)]) – The second observable sequence in the concatenation.

Return type
Observable[_T_out]

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 (Union[slice, int]) – Slice object

Return type
Observable[TypeVar(_T_out, covariant=True)]

Returns
Sliced observable sequence.

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

class reactivex.**Observer**(*on_next=None*, *on_error=None*, *on_completed=None*)

Base class for implementations of the Observer class. This base class enforces the grammar of observers where OnError and OnCompleted are terminal messages.

__init__(*on_next=None*, *on_error=None*, *on_completed=None*)

on_next(*value*)

Notify the observer of a new element in the sequence.

Return type
None

on_error(*error*)

Notify the observer that an exception has occurred.

Parameters
error (Exception) – The error that occurred.

Return type
None

on_completed()

Notifies the observer of the end of the sequence.

Return type
None

dispose()

Disposes the observer, causing it to transition to the stopped state.

Return type
None

to_notifier()

Creates a notification callback from an observer.

Returns the action that forwards its input notification to the underlying observer.

Return type
Callable[[*Notification*[TypeVar(_T_in, contravariant=True)]], None]

as_observer()

Hides the identity of an observer.

Returns an observer that hides the identity of the specified observer.

Return type
ObserverBase[TypeVar(_T_in, contravariant=True)]

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

Returns an observable sequence that contains a single element, using the specified scheduler to send out observer messages. There is an alias called 'just'. cbf658341a663218557b4c18fc5a96f46d58f18d.png marble\sphinxhyphen {}cbf65

Examples

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

Parameters
value (TypeVar(_T)) – Single element in the resulting observable sequence.

Return type
Observable[TypeVar(_T)]

Returns
An observable sequence containing the single specified element.

reactivex.**pipe**(*__value*, **fns*)

Functional pipe (|>)

Allows the use of function argument on the left side of the function.

Return type
Any

Example

```
>>> pipe(x, fn) == __fn(x)  # Same as x |> fn
>>> pipe(x, fn, gn) == gn(fn(x))  # Same as x |> fn |> gn
...
```

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

Generates an observable sequence of integral numbers within a specified range, using the specified scheduler to send out observer messages. 8ee4d4a3f526f44dd889e96a4df0e5849b0e6a4f.png marble\sphinxhyphen {}8ee4d4a3f526f4

Examples

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

Parameters

- **start** (int) – The value of the first integer in the sequence.
- **stop** (Optional[int]) – [Optional] Generate number up to (exclusive) the stop value. Default is *sys.maxsize*.
- **step** (Optional[int]) – [Optional] The step to be used (default is 1).
- **scheduler** (Optional[SchedulerBase]) – [Optional] The scheduler to schedule the values on. If not specified, the default is to use an instance of *CurrentThreadScheduler*.

Return type
Observable[int]

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

reactivex.**repeat_value**(*value*, *repeat_count=None*)

Generates an observable sequence that repeats the given element the specified number of times. marble\sphinxhyphen {}1a88d800efb3fefa1180164e679c53cbadbeeaa6.png

Examples

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

Parameters

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

Return type
Observable[TypeVar(_T)]

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

class reactivex.**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 (TypeVar(_T)) – 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.

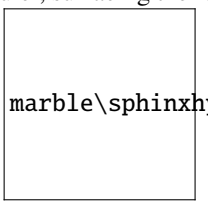
Return type
None

dispose()

Unsubscribe all observers and release resources.

Return type
None

reactivex.**start**(*func*, *scheduler=None*)

Invokes the specified function asynchronously on the specified scheduler, surfacing the result through an observable 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 = reactivex.start(lambda: pprint('hello'))
>>> res = reactivex.start(lambda: pprint('hello'), rx.Scheduler.timeout)
```

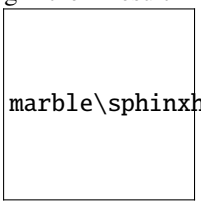
Parameters

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

Return type
Observable[TypeVar(_T)]

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

reactivex.**start_async**(*function_async*)

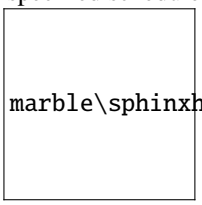
Invokes the asynchronous function, surfacing the result through an observable sequence. 

4f154949ac72b6ead68198137611e8c13c2e7c51.png

Parameters
function_async – Asynchronous function which returns a Future to run.

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

reactivex.**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 

Example

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

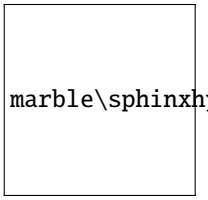
Parameters

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

Return type
Observable[Any]

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

reactivex.**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 

Examples

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

Parameters

- **duetime** (Union[datetime, timedelta, float]) – Absolute (specified as a datetime object) or relative time (specified as a float denoting seconds or an instance of timedelta) at which to produce the first value.

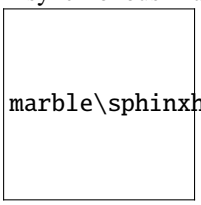
- **period** (Union[timedelta, float, None]) – [Optional] Period to produce subsequent values (specified as a float denoting seconds or an instance of timedelta). If not specified, the resulting timer is not recurring.
- **scheduler** (Optional[SchedulerBase]) – [Optional] Scheduler to run the timer on. If not specified, the default is to use an instance of *TimeoutScheduler*.

Return type
Observable[int]

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

reactivex.**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

Examples

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

Parameters

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

Return type
Callable[..., *Observable*[TypeVar(_T)]]

Returns
Asynchronous function.

reactivex.**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 = reactivex.using(lambda: AsyncSubject(), lambda: s: s)
```

Parameters

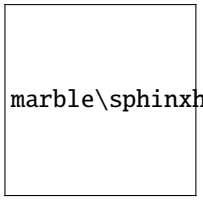
- **resource_factory** (Callable[[], DisposableBase]) – Factory function to obtain a resource object.
- **observable_factory** (Callable[[DisposableBase], *Observable*[TypeVar(_T)]]) – Factory function to obtain an observable sequence that depends on the obtained resource.

Return type
Observable[TypeVar(_T)]

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

reactivex.**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



Examples

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

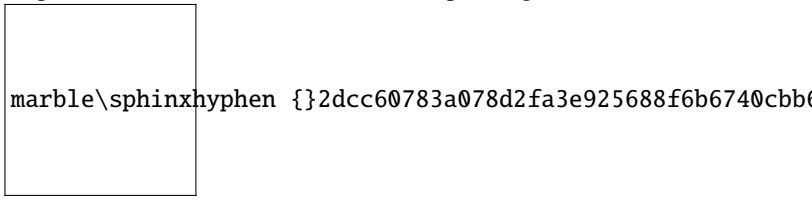
Parameters
sources (*Observable*[Any]) – Sequence of observables.

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

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

reactivex.**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.



2dcc60783a078d2fa3e925688f6b6740cbb65d36.png

Example

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

Parameters
args (*Observable*[Any]) – Observable sources to zip.

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

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

9.2 Observable

class reactivex.**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[[ObserverBase[TypeVar(_T_out, covariant=True)], Optional[SchedulerBase]], DisposableBase]]) – [Optional] Subscription function

subscribe(*on_next=None*, *on_error=None*, *on_completed=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** – [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** (Union[ObserverBase[TypeVar(_T_out, covariant=True)], Callable[[TypeVar(_T_out, covariant=True)], None], None]) – [Optional] Action to invoke for each element in the observable sequence.
- **scheduler** (Optional[SchedulerBase]) – [Optional] The default scheduler to use for this subscription.

Return type
DisposableBase

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

pipe(**operators*)

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[[Any], 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
`Generator`[`Any`, `None`, `TypeVar`(`_T_out`, covariant=True)]

Returns
The last item of the observable sequence.

`__add__`(*other*)

Pythonic version of *concat*.

Example

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

Parameters
other (*Observable*[`TypeVar`(`_T_out`, covariant=True)]) – The second observable sequence in the concatenation.

Return type
Observable[`TypeVar`(`_T_out`, covariant=True)]

Returns
Concatenated observable sequence.

`__iadd__`(*other*)

Pythonic use of *concat*.

Example

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

Parameters
other (*Observable*[`TypeVar`(`_T_out`, covariant=True)]) – The second observable sequence in the concatenation.

Return type
Observable[_T_out]

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 (Union[slice, int]) – Slice object

Return type
Observable[TypeVar(_T_out, covariant=True)]

Returns
Sliced observable sequence.

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

9.3 Subject

class reactivex.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 (TypeVar(_T)) – 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 reactivex.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 (TypeVar(_T)) – 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 reactivex.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[SchedulerBase]) – [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 reactivex.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

9.4 Schedulers

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

property now: datetime

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

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

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.

- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[TypeVar(_TState)]], Optional[TypeVar(_TState)]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] Initial state passed to the action upon the first iteration.

Return type
DisposableBase

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

class reactivex.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__`()

class reactivex.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*)

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[TypeVar(_TState)]], Optional[TypeVar(_TState)]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] Initial state passed to the action upon the first iteration.

Return type
DisposableBase

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 reactivex.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.

class reactivex.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*)

Return type
ImmediateScheduler

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

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

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

__init__(*thread_factory=None*)

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[TypeVar(_TState)]], Optional[TypeVar(_TState)]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] Initial state passed to the action upon the first iteration.

Return type
DisposableBase

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

`class reactivex.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*)

__init__(*max_workers=None*)

class reactivex.scheduler.TimeoutScheduler

A scheduler that schedules work via a timed callback.

static __new__(*cls*)

Return type
TimeoutScheduler

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

class reactivex.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__()

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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.

Return type
Optional[DisposableBase]

class reactivex.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 (Union[datetime, float]) – Initial value for the clock.

property **now: datetime**

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

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

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.

- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

start()

Starts the virtual time scheduler.

Return type
Any

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

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 reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

property now: datetime

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

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

class reactivex.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.

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

Schedules an action to be executed.

Parameters

- **action** (Callable[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

property now: datetime

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

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

class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

property now: datetime

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

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

class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

property now: datetime

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

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

class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

property now: datetime

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

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

class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[TypeVar(_TState)]], Optional[TypeVar(_TState)]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

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

Return type
DisposableBase

class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]] :param action: Action to be executed. :type state: Optional[TypeVar(_TState)] :param state: [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

`class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[TypeVar(_TState)]], Optional[TypeVar(_TState)]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

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

Return type
DisposableBase

`class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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

class reactivex.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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[[SchedulerBase, Optional[TypeVar(_TState)]], Optional[DisposableBase]]) – Action to be executed.
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

Return type
DisposableBase

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[TypeVar(_TState)]], Optional[TypeVar(_TState)]]) – Action to be executed.

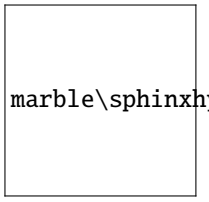
- **state** (Optional[TypeVar(_TState)]) – [Optional] state to be given to the action function.

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

Return type
DisposableBase

9.5 Operators

reactivex.operators.**all**(*predicate*)

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

Example

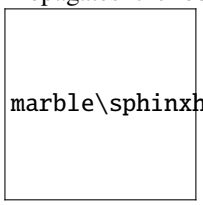
```
>>> op = all(lambda value: value.length > 3)
```

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[bool]]

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

reactivex.operators.**amb**(*right_source*)

Propagates the observable sequence that reacts first. f87bbff3e9ad64d121cc9d2ab3d1aacd4d9015c8.png


Example

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**as_observable**()

Hides the identity of an observable sequence.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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.

marble\sphinxhyphen {}b764a37a8ac08a8d9d9f8022185c88d1c8fb776b.png

Examples

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

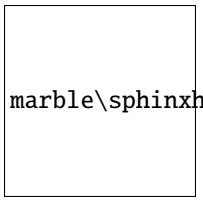
Parameters
key_mapper (Optional[Callable[[TypeVar(_T)], float]]) – [Optional] A transform function to apply to each element.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[float]]

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.

reactivex.operators.**buffer**(*boundaries*)

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

81dbcd0cd7a21fb6c1f4939387f5ff16b1556101.png 

Examples

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

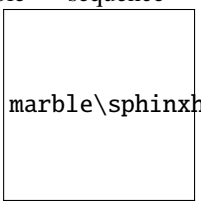
Parameters
boundaries (*Observable*[Any]) – Observable sequence whose elements denote the creation and completion of buffers.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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

reactivex.operators.**buffer_when**(*closing_mapper*)

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

1a971c0609b22ba1f5d5d63abfc62be378139340.png 

Examples

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

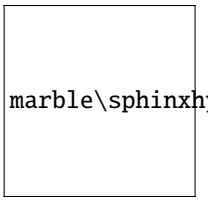
Parameters
closing_mapper (Callable[[], *Observable*[Any]]) – A function invoked to define the closing of each produced buffer. A buffer is started when the previous one is closed, resulting in non-overlapping buffers. The buffer is closed when one item is emitted or when the observable completes.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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

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

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



9c9ad906399c982a7ca5eb85fd1ce3d2c4ab39a5.png

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

Parameters

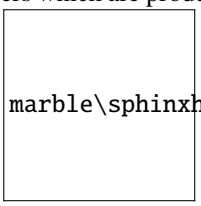
- **openings** (*Observable*[Any]) – Observable sequence whose elements denote the creation of buffers.
- **closing_mapper** (Callable[[Any], *Observable*[Any]]) – A function invoked to define the closing of each produced buffer. Value from openings Observable that initiated the associated buffer is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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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

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

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



count information. 58fe81f8f4a247bd862f50ae3cc219a98197ff80.png

Examples

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

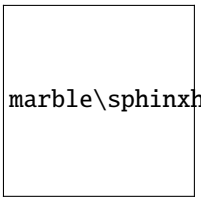
Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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

reactivex.operators.**buffer_with_time**(*timespan*, *timeshift=None*, *scheduler=None*)

Projects each element of an observable sequence into zero or more buffers which are produced based on timing information. 8dca4a83325669720b5ee0100ff548f37f73038f.png 

Examples

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

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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

reactivex.operators.**buffer_with_time_or_count**(*timespan*, *count*, *scheduler=None*)

Projects each element of an observable sequence into a buffer that is completed when either it's full or a given amount of time has elapsed. c8e0d068c8d51a7ab5499296538b2c74025dfe40.png 

Examples

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

Parameters

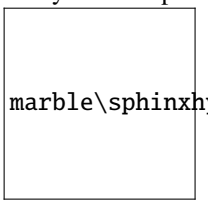
- **timespan** (Union[timedelta, float]) – Maximum time length of a buffer.
- **count** (int) – Maximum element count of a buffer.
- **scheduler** (Optional[SchedulerBase]) – [Optional] Scheduler to run buffering timers on. If not specified, the timeout scheduler is used.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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

reactivex.operators.**catch**(*handler*)

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



d14c02dddb2af945932be6a8f5b44425cde47a95.png

Examples

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

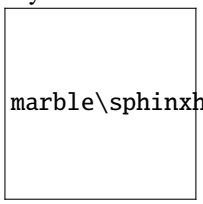
Parameters
handler (Union[*Observable*[TypeVar(_T)], Callable[[Exception, *Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]]) – Second observable sequence used to produce results when an error occurred in the first sequence, or an exception handler function that returns an observable sequence given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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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



Examples

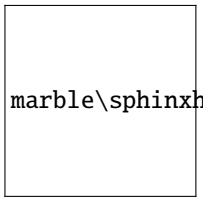
```
>>> obs = combine_latest(other)
>>> obs = combine_latest(obs1, obs2, obs3)
```

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

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

reactivex.operators.**concat**(**sources*)

Concatenates all the observable sequences. 4544dd242d02df82ac059a82701a17b5b31135d4.png



Examples

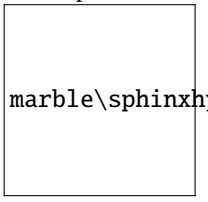
```
>>> op = concat(xs, ys, zs)
```

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**contains**(*value*, *comparer=None*)

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



b55e762ff5a35f34ec54f1b2ec6c70eaf7775a3f.png

Examples

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

Parameters

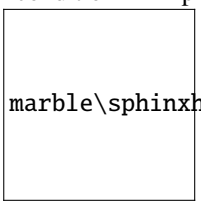
- **value** (TypeVar(_T)) – The value to locate in the source sequence.
- **comparer** (Optional[Callable[[TypeVar(_T), TypeVar(_T)], bool]]) – [Optional] An equality comparer to compare elements.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[bool]]

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

reactivex.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

Examples

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

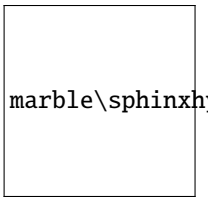
Parameters
predicate (Optional[Callable[[TypeVar(_T)], bool]]) – A function to test each element for a condition.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[int]]

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.

reactivex.operators.**debounce**(*duetime*, *scheduler=None*)

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

marble\sphinxhyphen {}ddf862178779eaa8593df342589be502be49ddbb.png 

Example

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

Parameters

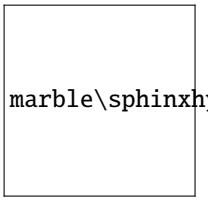
- **duetime** (Union[timedelta, float]) – Duration of the throttle period for each value (specified as a float denoting seconds or an instance of timedelta).
- **scheduler** (Optional[SchedulerBase]) – Scheduler to debounce values on.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**throttle_with_timeout**(*duetime*, *scheduler=None*)

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

marble\sphinxhyphen {}ddf862178779eaa8593df342589be502be49ddbb.png 

Example

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

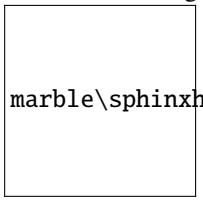
Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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

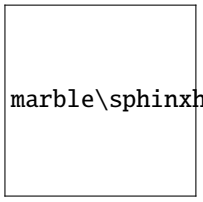
Callable[[*Observable*[Any]], *Observable*[Any]]

Returns

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

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

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



Example

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

Parameters

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

Return type

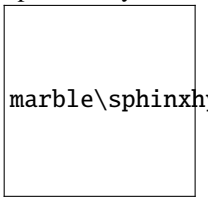
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns

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

reactivex.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.



1c57db95ece5fc3af60515bb5b99603381d3f000.png

Examples

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

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**dematerialize**()

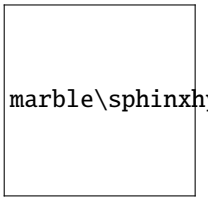
Dematerialize operator.

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

Return type
Callable[[*Observable*[*Notification*[TypeVar(_T)]]], *Observable*[TypeVar(_T)]]

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

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



The delay operator. 1c57db95ece5fc3af60515bb5b99603381d3f000.png

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

Examples

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

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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 {}1a8e43d967e0923c86f

Examples

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

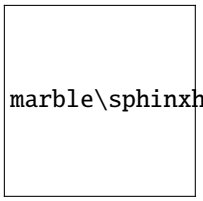
Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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 

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)
```

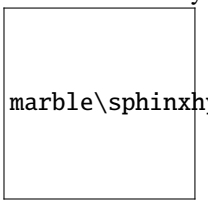
Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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)
```

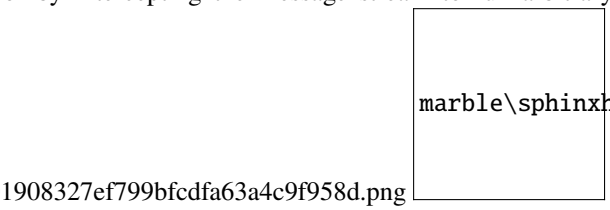
Parameters
observer (ObserverBase[TypeVar(_T)]) – Observer

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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

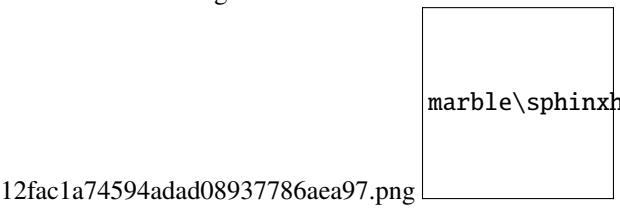
- **on_next** (Optional[Callable[[TypeVar(_T)], None]]) – [Optional] Action to invoke for each element in the observable sequence.
- **on_error** (Optional[Callable[[Exception], None]]) – [Optional] Action to invoke on exceptional termination of the observable sequence.
- **on_completed** (Optional[Callable[[],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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**do_while**(*condition*)

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



74f3910edce12fac1a74594adad08937786aea97.png

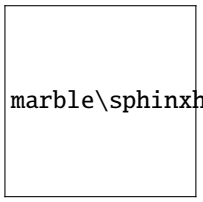
Parameters
condition (Callable[[*Observable*[TypeVar(_T)]], bool]) – The condition which determines if the source will be repeated.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**element_at**(*index*)

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



Example

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

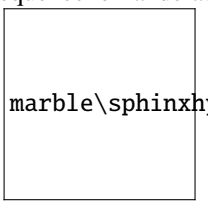
Parameters
index (int) – The zero-based index of the element to retrieve.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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



Example

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

Parameters

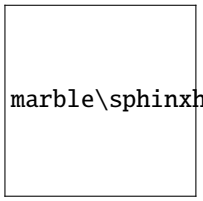
- **index** (int) – The zero-based index of the element to retrieve.
- **default_value** (Optional[TypeVar(_T)]) – [Optional] The default value if the index is outside the bounds of the source sequence.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**exclusive**()

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



Return type
Callable[[*Observable*[*Observable*[TypeVar(_T)]]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**expand**(*mapper*)

Expands an observable sequence by recursively invoking mapper.

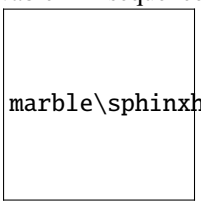
Parameters
mapper (Callable[[TypeVar(_T)], *Observable*[TypeVar(_T)]]) – Mapper function to invoke for each produced element, resulting in another sequence to which the mapper will be invoked recursively again.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An observable sequence containing all the elements produced

by the recursive expansion.

reactivex.operators.**filter**(*predicate*)

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

2ac5743543b7ddb0ba7b7dc3cedacda9de1a73eb.png

Example

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

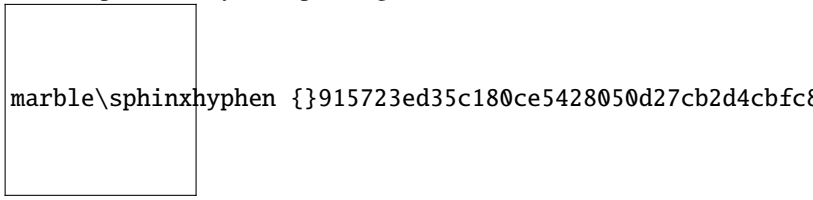
Parameters
predicate (Callable[[TypeVar(_T)], bool]) – A function to test each source element for a condition.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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

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

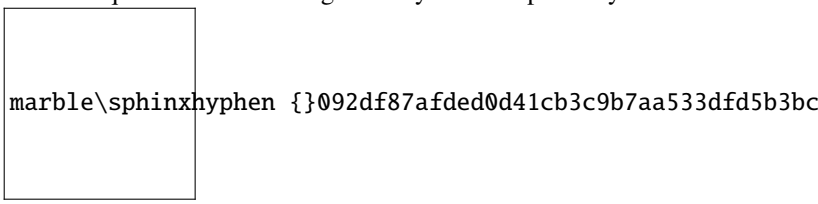
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns

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

reactivex.operators.**finally_action**(*action*)

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



092df87afded0d41cb3c9b7aa533dfd5b3bca776.png

Example

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

Parameters

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

Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

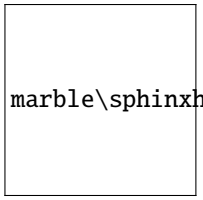
Returns

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

reactivex.operators.**find**(*predicate*)

Searches for an element that matches the conditions defined by the specified predicate, and returns the

first occurrence within the entire Observable sequence. 50c59c6e10555a1998fa10d51cbd9ffdaba3ac3d.png



Parameters

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

Return type

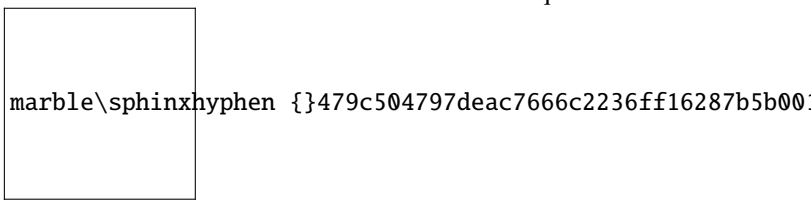
Callable[[*Observable*[TypeVar(_T)]], *Observable*[Optional[TypeVar(_T)]]]

Returns

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

reactivex.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.



479c504797deac7666c2236ff16287b5b001b465.png

Parameters

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

Return type

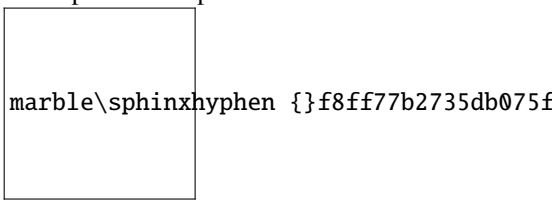
Callable[[*Observable*[TypeVar(_T)]], *Observable*[Optional[int]]]

Returns

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

reactivex.operators.**first**(*predicate=None*)

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



first item in the sequence. f8ff77b2735db075f91ed29f2fe3fc0b1c918d49.png

Examples

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

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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 if no such element exists. e2da2e19412d614c899fa5b6ff0b46b0b28a65f2.png marble\sphinxhyphen {}e2da2e19412d614c8

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)
```

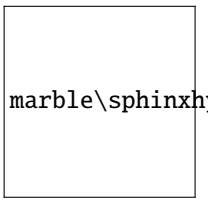
Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**flat_map**(*mapper=None*)



The flat_map operator. 0796d302706411e6023b5671684563964c0afa1c.png

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

Example

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

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

Example

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

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

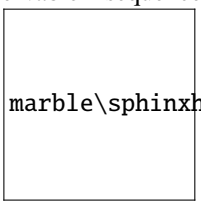
Return type
Callable[[*Observable*[Any]], *Observable*[Any]]

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

reactivex.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

Example

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

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

Example

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

Parameters

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

Return type

Callable[[*Observable*[Any]], *Observable*[Any]]

Returns

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

reactivex.operators.**flat_map_latest**(*mapper*)

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

Parameters

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

Returns

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

reactivex.operators.**fork_join**(**others*)

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

Examples

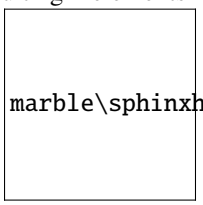
```
>>> res = fork_join(obs1)
>>> res = fork_join(obs1, obs2, obs3)
```

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

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

reactivex.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.



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

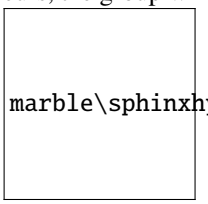
- **key_mapper** – A function to extract the key for each element.
- **element_mapper** – [Optional] A function to map each source element to an element in an observable group.
- **subject_mapper** – A function that returns a subject used to initiate a grouped observable. Default mapper returns a Subject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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[*GroupedObservable*[TypeVar(_TKey), TypeVar(_TValue)]]]

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

reactivex.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

Examples

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

Parameters

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

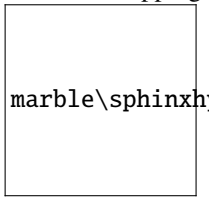
Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[*GroupedObservable*[TypeVar(_TKey), TypeVar(_TValue)]]]

Returns

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

reactivex.operators.**group_join**(*right*, *left_duration_mapper*, *right_duration_mapper*)

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

96c6bd5e72804426dd0d45e7be6f65a95cfc23bd.png

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_TLeft)]], *Observable*[Tuple[TypeVar(_TLeft), *Observable*[TypeVar(_TRight)]]]]

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

reactivex.operators.**ignore_elements**()

Ignores all elements in an observable sequence leaving only the termination messages. marble\sphinxhyphen {}495529f1b8626a1641e59c8045dc3bfe4a46e52b.png

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**is_empty**()

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

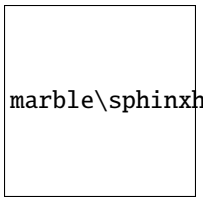
marble\sphinxhyphen {}84685da521ceac3a1e3151b469388c793833bd91.png

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

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

reactivex.operators.**join**(*right*, *left_duration_mapper*, *right_duration_mapper*)

Correlates the elements of two sequences based on overlapping durations.



b3ff56065e83731d477f51b5fc6003d2fc22a162.png

Parameters

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

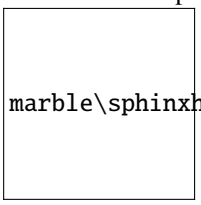
Return type
Callable[[*Observable*[TypeVar(_T1)]], *Observable*[Tuple[TypeVar(_T1), TypeVar(_T2)]]]

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

reactivex.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[[TypeVar(_T)], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.

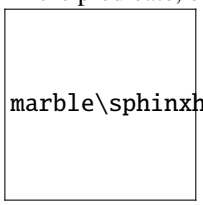
Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**last_or_default**(*default_value=None*, *predicate=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. ac702793ffd235d9aecfffd4dccce018f409b71c.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[[TypeVar(_T)], bool]]) – [Optional] A predicate function to evaluate for elements in the source sequence.
- **default_value** (Optional[Any]) – [Optional] The default value if no such element exists. If not specified, defaults to None.

Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[Any]]

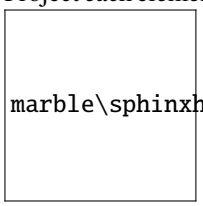
Returns

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

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

The map operator.

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



Example

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

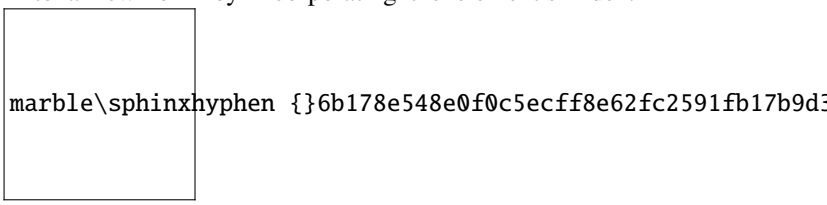
Parameters
mapper (Optional[Callable[[TypeVar(_T1)], TypeVar(_T2)]]) – A transform function to apply to each source element.

Return type
Callable[[*Observable*[TypeVar(_T1)]], *Observable*[TypeVar(_T2)]]

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

reactivex.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

Example

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

Parameters
mapper_indexed (Optional[Callable[[TypeVar(_T1), int], TypeVar(_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*[TypeVar(_T1)]], *Observable*[TypeVar(_T2)]]

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

reactivex.operators.**materialize**()

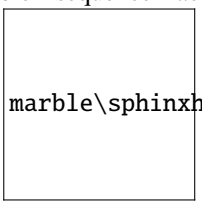
Materializes the implicit notifications of an observable sequence as explicit notification values.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[*Notification*[TypeVar(_T)]]]

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

reactivex.operators.**max**(*comparer=None*)

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



90c0d247c7fb08ce52df4fc3aeda0b963862f912.png

Examples

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

Parameters

comparer (Optional[Callable[[TypeVar(_T), TypeVar(_T)], bool]]) – [Optional] Comparer used to compare elements.

Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

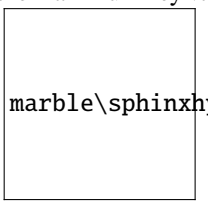
Returns

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.

reactivex.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 comparer.



1ef072a3c5f1bb83ee48ada6c4083bcc73327e36.png

Examples

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

Parameters

- **key_mapper** (Callable[[TypeVar(_T)], TypeVar(_TKey)]) – Key mapper function.
- **comparer** (Optional[Callable[[TypeVar(_TKey), TypeVar(_TKey)], bool]]) – [Optional] Comparer used to compare key values.

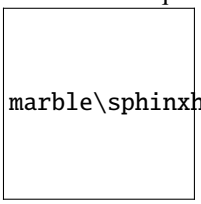
Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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.

reactivex.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 

Examples

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

Parameters

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

Return type

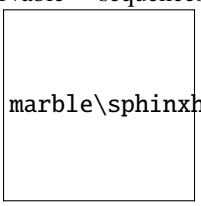
Callable[[*Observable*[Any]], *Observable*[Any]]

Returns

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

reactivex.operators.**merge_all**()

The merge_all operator.

Merges an observable sequence of observable sequences into an observable sequence. bf397132fa74dd0d17e17f08ce1314bbef0bbbd1.png 

Return type

Callable[[*Observable*[*Observable*[TypeVar(_T)]]], *Observable*[TypeVar(_T)]]

Returns

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

reactivex.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 

Examples

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

Parameters
comparer (Optional[Callable[[TypeVar(_T), TypeVar(_T)], bool]]) – [Optional] Comparer used to compare elements.

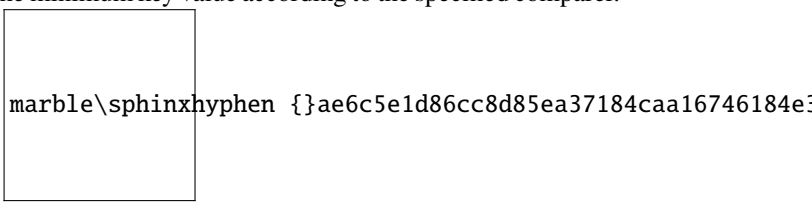
Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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.

reactivex.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 comparer.



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[[TypeVar(_T)], TypeVar(_TKey)]) – Key mapper function.
- **comparer** (Optional[Callable[[TypeVar(_TKey), TypeVar(_TKey)], bool]]) – [Optional] Comparer used to compare key values.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

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.

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

Return type
Callable[[*Observable*[TypeVar(_T)]], Union[*Observable*[TypeVar(_T2)], *ConnectableObservable*[TypeVar(_T)]]]

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

reactivex.operators.**observe_on**(*scheduler*)

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

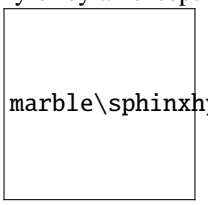
Parameters
scheduler (SchedulerBase) – Scheduler to notify observers on.

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**on_error_resume_next**(*second*)

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

cffe1529783c4bd3518f934f74ac7f8960bd15ed.png

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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*[TypeVar(_T)]], *Observable*[Tuple[TypeVar(_T), TypeVar(_T)]]]

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

reactivex.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

Parameters

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

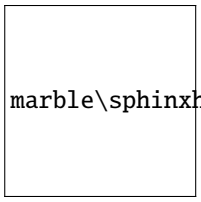
Return type
Callable[[*Observable*[TypeVar(_T)]], List[*Observable*[TypeVar(_T)]]]

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

reactivex.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.

marble\sphinxhyphen {}9d1412bca63017f8545dfdf42baf36e766dd 9d1412bca63017f8545dfdf42baf36e766ddce27.png 

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], List[*Observable*[TypeVar(_T)]]]

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

reactivex.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 (TypeVar(_TKey)) – The key to pluck.

Return type
Callable[[*Observable*[Dict[TypeVar(_TKey), TypeVar(_TValue)]]], *Observable*[TypeVar(_TValue)]]

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

reactivex.operators.**pluck_attr**(*prop*)

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

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

Parameters
property – The property to pluck.

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

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

reactivex.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[[*Observable*[TypeVar(_T1)]], *Observable*[TypeVar(_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*[TypeVar(_T1)]], Union[*Observable*[TypeVar(_T2)], *ConnectableObservable*[TypeVar(_T1)]]]

Returns

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

reactivex.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** (TypeVar(_T1)) – Initial value received by observers upon subscription.
- **mapper** (Optional[Callable[[*Observable*[TypeVar(_T1)]], *Observable*[TypeVar(_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*[TypeVar(_T1)]], Union[*Observable*[TypeVar(_T2)], *ConnectableObservable*[TypeVar(_T1)]]]

Returns

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

reactivex.operators.**reduce**(*accumulator*, *seed=<class 'reactivex.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

Examples

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

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[Any]]

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

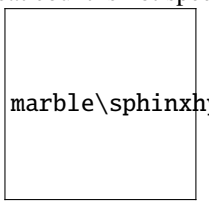
reactivex.operators.**ref_count**()

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*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

reactivex.operators.**repeat**(*repeat_count=None*)

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



500cd215f2c580628cf9a7e611f44d42054a0e0a.png

Examples

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

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

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

Return type
Callable[[*Observable*[TypeVar(_T1)]], Union[*Observable*[TypeVar(_T2)], *ConnectableObservable*[TypeVar(_T1)]]]

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

reactivex.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*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

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

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

marble\sphinxhyphen {}753724403f4ec050f1e41ae18c0aa6c739db97a9.png

Examples

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

Parameters

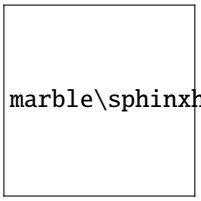
- **sampler** (Union[timedelta, float, *Observable*[Any]]) – Observable used to sample the source observable **or** time interval at which to sample (specified as a float denoting seconds or an instance of timedelta).
- **scheduler** (Optional[SchedulerBase]) – Scheduler to use only when a time interval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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**scan**(*accumulator*, *seed=<class 'reactivex.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()*. bc4ff8e48a531690a2cf5b2a166dfd22fd92290c.png 

Examples

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

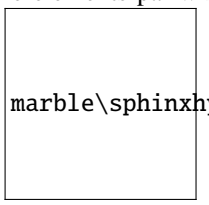
Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_TState)]]

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

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

Determines whether two sequences are equal by comparing the elements pairwise using a specified equality comparer. 9aae3d37d38bfe67409f48b0b2e47fa1b19cc626.png 

Examples

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

Parameters

- **second** (Union[*Observable*[TypeVar(_T)], Iterable[TypeVar(_T)]]) – Second observable sequence or iterable to compare.

- **comparer** (Optional[Callable[[TypeVar(_T), TypeVar(_T)], bool]]) – [Optional] Comparer used to compare elements of both sequences. No guarantees on order of comparer arguments.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[bool]]

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

reactivex.operators.**share**()

Share a single subscription among multiple observers.

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

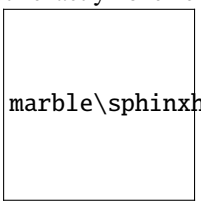
Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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.



59e02f94169eadf875bb292a59cda555b87d5092.png

Example

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

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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.

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

Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

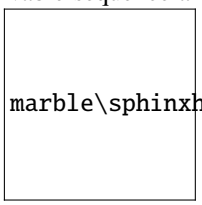
Returns

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

reactivex.operators.**skip**(*count*)

The skip operator.

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



9829e00890b070583966679ff0055d2470c7f095.png

Parameters

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

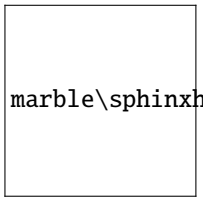
Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns

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

reactivex.operators.**skip_last**(*count*)



The skip_last operator. a5f9d54f3ed827712265516535934307683b5e1c.png

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

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

Parameters

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

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

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

Example

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

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

Parameters

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

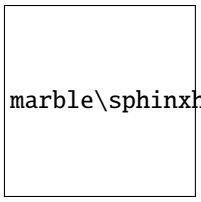
Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An observable sequence with the elements skipped during the

specified duration from the end of the source sequence.

reactivex.operators.**skip_until**(*other*)

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



value. f275ca05c567b536db954d93cc4a27a8309a01d8.png

Parameters

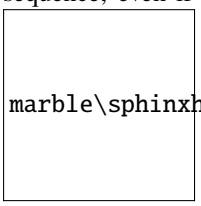
other – The observable sequence that triggers propagation of elements of the source sequence.

Returns

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

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

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



531777319ad55f0b44484e3380596082c25027fc.png

Examples

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

Parameters

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

Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns

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

reactivex.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

Example

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

Parameters
predicate (Callable[[TypeVar(_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*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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

Example

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

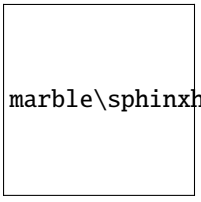
Parameters
predicate (Callable[[TypeVar(_T), int], bool]) – A function to test each element for a condition; the second parameter of the function represents the index of the source element.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

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

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



281c641f8ea3abef9b70fb19d7295b200ebff650.png

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

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

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

Parameters

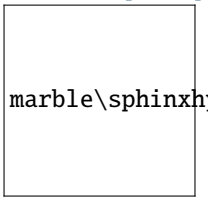
- **duration** (Union[timedelta, float]) – Duration for skipping elements from the start of the
- **sequence.** –

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

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

The slice operator.

Slices the given observable. It is basically a wrapper around the operators *skip*, *skip_last*, *take*, *take_last* and *filter*. 80c2d0fdc2c767f0a81ea312a7dec1b9cf921f0d.png 

Examples

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

Parameters

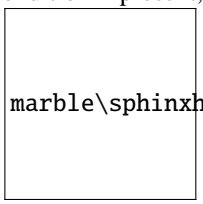
- **start** (Optional[int]) – First element to take of skip last
- **stop** (Optional[int]) – Last element to take of skip last
- **step** (Optional[int]) – Takes every step element. Must be larger than zero

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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[[TypeVar(_T)], bool]]) – A function to test each element for a condition.

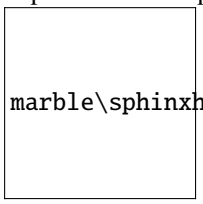
Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[bool]]

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

reactivex.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 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[..., Any]]) – A transform function to invoke with unpacked elements as arguments.

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

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

reactivex.operators.**starmap_indexed**(*mapper=None*)

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

Example

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

Parameters

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

Return type

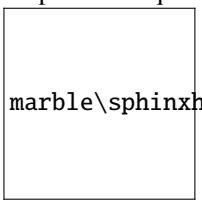
Callable[[*Observable*[Any]], *Observable*[Any]]

Returns

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

reactivex.operators.**start_with**(**args*)

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



Example

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

Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns

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

reactivex.operators.**subscribe_on**(*scheduler*)

Subscribe on the specified scheduler.

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

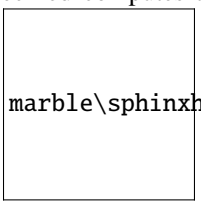
This only performs the side-effects of subscription and unsubscription on the specified scheduler. In order to invoke observer callbacks on a scheduler, use observe_on.

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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

Examples

```
>>> res = sum()
>>> res = sum(lambda x: x.value)
```

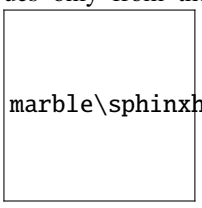
Parameters
key_mapper (Optional[Callable[[Any], float]]) – [Optional] A transform function to apply to each element.

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

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.

reactivex.operators.**switch_latest**()

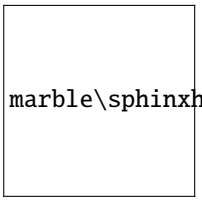
The switch_latest operator.

Transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence. b9329fb493471af771a2b890318e9a08d9bc0cfd.png


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.

reactivex.operators.**take**(*count*)

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



6e19e75a07bd399aae57e54367ef89cdf3f4abcb.png

Example

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

Parameters

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

Return type

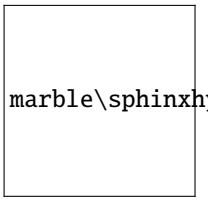
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns

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

reactivex.operators.**take_last**(*count*)

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



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*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

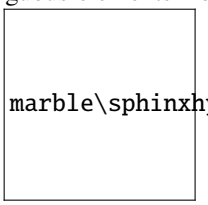
Returns

An operator function that takes an observable source and returns an observable sequence containing the specified number of elements from the end of the source sequence.

reactivex.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

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*[TypeVar(_T)]], *Observable*[List[TypeVar(_T)]]]

Returns

An operator function that takes an observable source and returns an observable sequence containing a single list with the specified number of elements from the end of the source sequence.

reactivex.operators.**take_last_with_time**(*duration*, *scheduler=None*)

Returns elements within the specified duration from the end of the observable source sequence.

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.** –

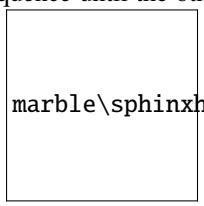
Return type

Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**take_until**(*other*)

Returns the values from the source observable sequence until the other observable sequence produces a value.



f263d0fbb0af0ae1fdad8f9e5c3f9653fbf31f1d.png

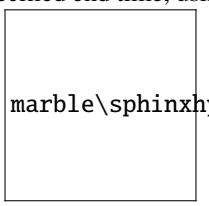
Parameters
other (*Observable*[Any]) – Observable sequence that terminates propagation of elements of the source sequence.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An operator function that takes an observable source and returns as observable sequence containing the elements of the source sequence up to the point the other sequence interrupted further propagation.

reactivex.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.



e57345367efb65a9a452388af6bfadbbb979b43e.png

Examples

```
>>> res = take_until_with_time(dt, [optional scheduler])
>>> res = take_until_with_time(5.0, [optional scheduler])
```

Parameters

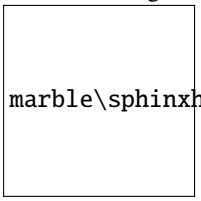
- **end_time** (Union[datetime, timedelta, float]) – Time to stop taking elements from the source sequence. If this value is less than or equal to *datetime.utcnow()*, the result stream will complete immediately.
- **scheduler** (Optional[SchedulerBase]) – Scheduler to run the timer on.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.operators.**take_while**(*predicate*, *inclusive=False*)

Returns elements from an observable sequence as long as a specified condition is true.



7d8e208720017f8eb920b0a29c31b85a1156fda7.png

Example

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

Parameters

- **predicate** (Callable[[TypeVar(_T)], bool]) – A function to test each element for a condition.
- **inclusive** (bool) – [Optional] When set to True the value that caused the predicate function to return False will also be emitted. If not specified, defaults to False.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An operator function that takes an observable source and returns an observable sequence that contains the elements from the input sequence that occur before the element at which the test no longer passes.

reactivex.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

Example

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

Parameters

- **predicate** (Callable[[TypeVar(_T), int], bool]) – A function to test each element for a condition; the second parameter of the function represents the index of the source element.
- **inclusive** (bool) – [Optional] When set to True the value that caused the predicate function to return False will also be emitted. If not specified, defaults to False.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An observable sequence that contains the elements from the input sequence that occur before the element at which the test no longer passes.

reactivex.operators.**take_with_time**(*duration*, *scheduler=None*)

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

c023c934bb4b63709fe6a9255bac3022b7ea174c.png

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*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An operator function that takes an observable source and returns an observable that performs the throttle operation.

reactivex.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: reactivex.timer(x+x))
```

Parameters

- **throttle_duration_mapper** (Callable[[Any], *Observable*[Any]]) – Mapper function to retrieve an
- **each** (*observable sequence indicating the throttle duration for*) –
- **element.** (*given*) –

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

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

reactivex.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*[TypeVar(_T)]], *Observable*[Timestamp[TypeVar(_T)]]]

Returns
A partially applied operator function that takes an observable source and returns an observable sequence with timestamp information on values.

reactivex.operators.**timeout**(*duetime*, *other=None*, *scheduler=None*)

Returns the source observable sequence or the other observable sequence if duetime elapses.

Examples

```
>>> res = timeout(5.0)
>>> res = timeout(datetime(), return_value(42))
>>> res = timeout(5.0, return_value(42))
```

Parameters

- **duetime** (Union[datetime, timedelta, float]) – Absolute (specified as a datetime object) or relative time (specified as a float denoting seconds or an instance of timedetla) when a timeout occurs.
- **other** (Optional[*Observable*[TypeVar(_T)]]) – Sequence to return in case of a timeout. If not specified, a timeout error throwing sequence will be used.
- **scheduler** (Optional[SchedulerBase]) –

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An operator function that takes and observable source and returns the source sequence switching to the other sequence in case of a timeout.

reactivex.operators.**timeout_with_mapper**(*first_timeout=None*, *timeout_duration_mapper=None*, *other=None*)

Returns the source observable sequence, switching to the other observable sequence if a timeout is signaled.

Examples

```
>>> res = timeout_with_mapper(reactivex.timer(0.5))
>>> res = timeout_with_mapper(
    reactivex.timer(0.5), lambda x: reactivex.timer(0.2)
)
>>> res = timeout_with_mapper(
    reactivex.timer(0.5),
    lambda x: reactivex.timer(0.2),
    reactivex.return_value(42)
)
```

Parameters

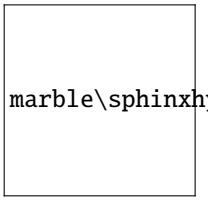
- **first_timeout** (Optional[*Observable*[Any]]) – [Optional] Observable sequence that represents the timeout for the first element. If not provided, this defaults to reactivex.never().
- **timeout_duration_mapper** (Optional[Callable[[TypeVar(_T)], *Observable*[Any]]]) – [Optional] Selector to retrieve an observable sequence that represents the timeout between the current element and the next element.
- **other** (Optional[*Observable*[TypeVar(_T)]]) – [Optional] Sequence to return in case of a timeout. If not provided, this is set to reactivex.throw().

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An operator function that takes an observable source and returns the source sequence switching to the other sequence in case of a timeout.

reactivex.operators.**time_interval**(*scheduler=None*)

Records the time interval between consecutive values in an observable sequence.



b0e3c487b07039af5b9dec98727bbaddf099f83d.png

Examples

```
>>> res = time_interval()
```

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TimeInterval[TypeVar(_T)]]]

Returns
An operator function that takes an observable source and returns an observable sequence with time interval information on values.

reactivex.operators.**to_dict**(*key_mapper*, *element_mapper=None*)

Converts the observable sequence to a Map if it exists.

Parameters

- **key_mapper** (Callable[[TypeVar(_T)], TypeVar(_TKey)]) – A function which produces the key for the dictionary.
- **element_mapper** (Optional[Callable[[TypeVar(_T)], TypeVar(_TValue)]]) – [Optional] An optional function which produces the element for the dictionary. If not present, defaults to the value from the observable sequence.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[Dict[TypeVar(_TKey), TypeVar(_TValue)]]]

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

reactivex.operators.**to_future**(*future_ctor=None*)

Converts an existing observable sequence to a Future.

Example

op = to_future(asyncio.Future);

Parameters
future_ctor – [Optional] The constructor of the future.

Returns
An operator function that takes an observable source and returns a future with the last value from the observable sequence.

reactivex.operators.**to_iterable**()

Creates an iterable from an observable sequence.

There is also an alias called `to_list`.

Return type
`Callable[[Observable[TypeVar(_T)]], Observable[List[TypeVar(_T)]]]`

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

reactivex.operators.**to_list**()

Creates an iterable from an observable sequence.

There is also an alias called `to_list`.

Return type
`Callable[[Observable[TypeVar(_T)]], Observable[List[TypeVar(_T)]]]`

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

reactivex.operators.**to_marbles**(*timespan=0.1*, *scheduler=None*)

Convert an observable sequence into a marble diagram string.

Parameters

- **timespan** (`Union[timedelta, float]`) – [Optional] duration of each character in second. If not specified, defaults to 0.1s.
- **scheduler** (`Optional[SchedulerBase]`) – [Optional] The scheduler used to run the the input sequence on.

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

Returns
Observable stream.

reactivex.operators.**to_set**()

Converts the observable sequence to a set.

Return type
`Callable[[Observable[TypeVar(_T)]], Observable[Set[TypeVar(_T)]]]`

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

reactivex.operators.**while_do**(*condition*)

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

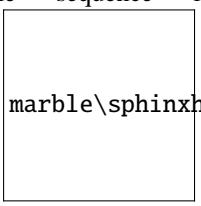
Parameters
condition (Callable[[*Observable*[TypeVar(_T)]], bool]) – The condition which determines if the source will be repeated.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[TypeVar(_T)]]

Returns
An operator function that takes an observable source and returns an observable sequence which is repeated as long as the condition holds.

reactivex.operators.**window**(*boundaries*)

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



2c04b5c44b74070176595a3a4e681bd1e8d6fcec.png

Examples

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

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

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[*Observable*[TypeVar(_T)]]]

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

reactivex.operators.**window_when**(*closing_mapper*)

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

52e4fca35f1720c22c35083a9d2cdc694a20aced.png

Examples

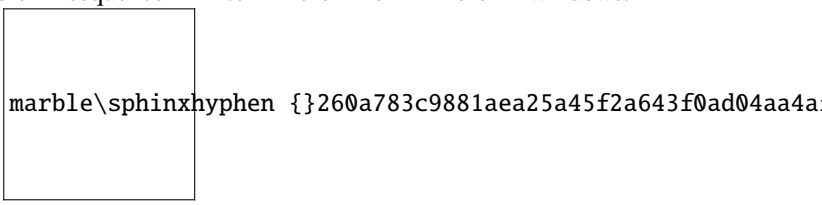
```
>>> res = window(lambda: reactivex.timer(0.5))
```

Parameters
closing_mapper (Callable[[], *Observable*[Any]]) – A function invoked to define the closing of each produced window. It defines the boundaries of the produced windows (a window is started when the previous one is closed, resulting in non-overlapping windows).

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[*Observable*[TypeVar(_T)]]]

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

reactivex.operators.**window_toggle**(*openings*, *closing_mapper*)

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

260a783c9881aea25a45f2a643f0ad04aa4af321.png

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

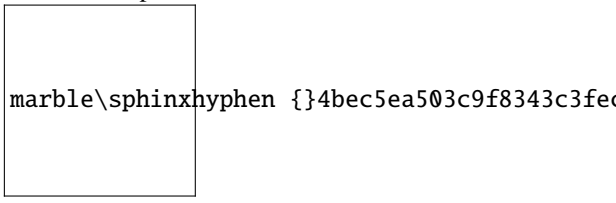
Parameters

- **openings** (*Observable*[Any]) – Observable sequence whose elements denote the creation of windows.
- **closing_mapper** (Callable[[Any], *Observable*[Any]]) – A function invoked to define the closing of each produced window. Value from openings Observable that initiated the associated window is provided as argument to the function.

Return type
Callable[[*Observable*[TypeVar(_T)]], *Observable*[*Observable*[TypeVar(_T)]]]

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

reactivex.operators.**window_with_count**(*count*, *skip=None*)

Projects each element of an observable sequence into zero or more windows which are produced based on element count information. 

4bec5ea503c9f8343c3fec6dda89105a28931cea.png

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*[TypeVar(_T)]], *Observable*[*Observable*[TypeVar(_T)]]]

Returns
An observable sequence of windows.

reactivex.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 {}1c1361f657a91ebf5ade83dbd9800bfa7

Examples

```
>>> op = with_latest_from(obs1)
>>> op = with_latest_from([obs1, obs2, obs3])
```

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

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

reactivex.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. 2dcc60783a078d2fa3e925688f6b6740cbb65d36.png marble\sphinxhyphen {}2dcc60783a078d2fa3e925688f6b6740cbb6

Example

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

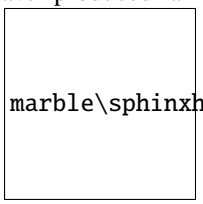
Parameters
args (*Observable*[Any]) – Observable sources to zip.

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

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

reactivex.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.



45da698d9b059e9fb52b48e83e592327605a35d5.png

Example

```
>>> res = zip([1,2,3])
```

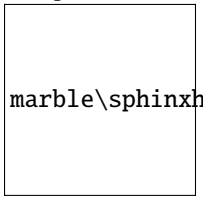
Parameters
second (Iterable[TypeVar(_T2)]) – Iterable to zip with the source observable..

Return type
Callable[[*Observable*[TypeVar(_T1)]], *Observable*[Tuple[TypeVar(_T1), TypeVar(_T2)]]]

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

reactivex.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

Example

```
>>> res = zip([1,2,3])
```

Parameters
second (Iterable[TypeVar(_T2)]) – Iterable to zip with the source observable..

Return type
Callable[[*Observable*[TypeVar(_T1)]], *Observable*[Tuple[TypeVar(_T1), TypeVar(_T2)]]]

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

9.6 Typing

CHAPTER

TEN

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.

CHAPTER

ELEVEN

THE MIT LICENSE

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

S

T

U

V

W

Z