RXpy

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

RxPY is a python library to support Reactive Programming. RxPy stands for **Reactive Extensions for Python**. It is a library that uses observables to work with reactive programming that deals with asynchronous data calls, callbacks and event-based programs. This tutorial will give you enough understanding on various functionalities of RxPY with suitable examples.

Audience

The tutorial is designed for software programmers who want to learn the basics of RxPY i.e. Reactive extension for Python and its programming concepts in simple and easy ways.

Prerequisites

We suggest you to go through tutorials related to Python, before proceeding with this tutorial.

Copyright & Disclaimer

Table of Contents

1. RxPY – Overview

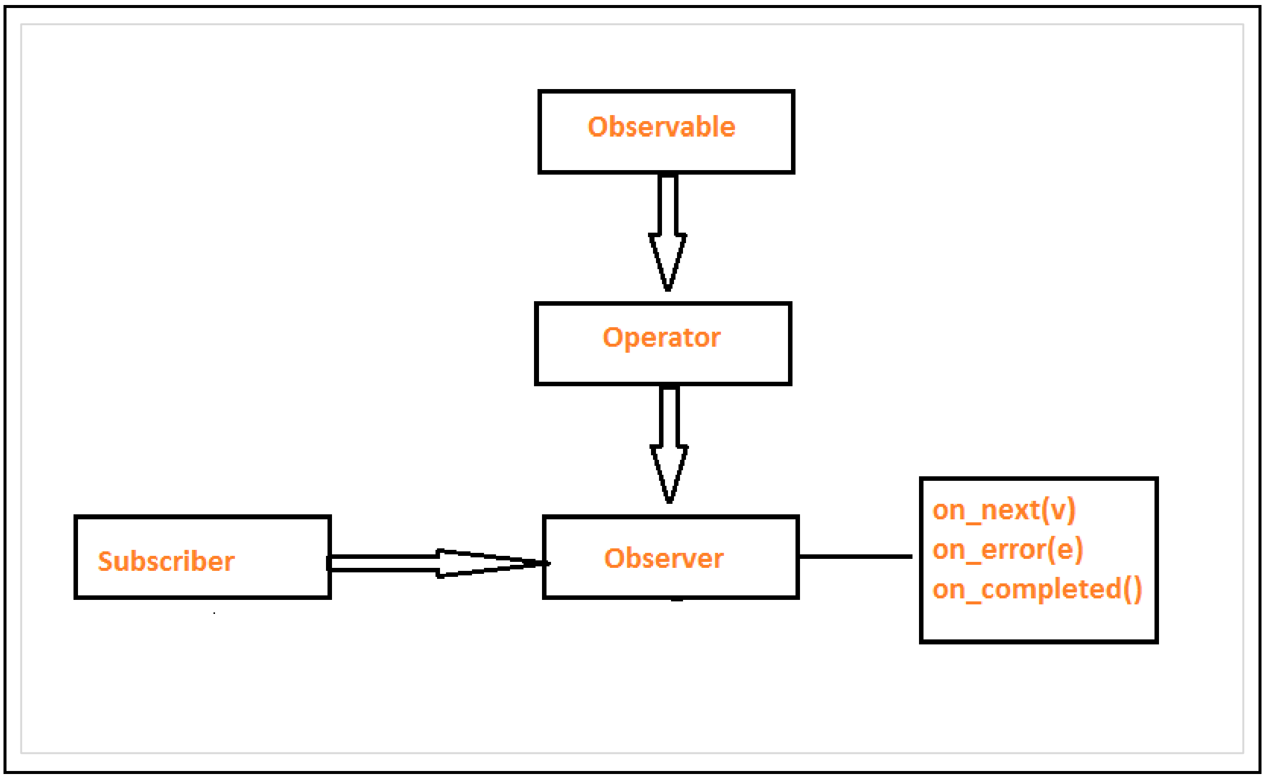
This chapter explains what is reactive programming, what is RxPY, its operators, features, advantages and disadvantage.

What is Reactive Programming?

Reactive programming is a programming paradigm, that deals with data flow and the propagation of change. It means that, when a data flow is emitted by one component, the change will be propagated to other components by a reactive programming library. The propagation of change will continue until it reaches the final receiver.

By using RxPY, you have good control on the asynchronous data streams, for example, a request made to URL can be traced by using observable, and use the observer to listen to when the request is complete for response or error.

RxPY offers you to handle asynchronous data streams using **Observables**, query the data streams using **Operators** i.e. filter, sum, concat, map and also make use of concurrency for the data streams using **Schedulers**. Creating an Observable, gives an observer object with on_next(v), on_error(e) and on_completed() methods, that needs to be **subscribed** so that we get a notification when an event occurs.



The Observable can be queried using multiple operators in a chain format by using the pipe operator.

RxPY offers operators in various categories like: -

- Mathematical operators
- Transformation operators
- Filtering operators
- Error handling operators
- Utility operators
- Conditional operators
- Creation operators
- Connectable operators

These operators are explained in detail in this tutorial.

What is RxPy?

RxPY is defined as **a library for composing asynchronous and event-based programs using observable collections and pipable query operators in Python** as per the official website of RxPy, which is https://rxpy.readthedocs.io/en/latest/.

RxPY is a python library to support Reactive Programming. RxPy stands for **Reactive Extensions for Python**. It's a library that uses observables to work with reactive programming that deals with asynchronous data calls, callbacks and event-based programs.

Features of RxPy

In RxPy, following concepts take care of handling the asynchronous task:

Observable

An observable is a function that creates an observer and attaches it to the source having data streams that are expected from, for example, Tweets, computer-related events, etc.

Observer

It is an object with on_next(), on_error() and on_completed() methods, that will get called when there is interaction with the observable i.e. the source interacts for an example incoming Tweets, etc.

Subscription

When the observable is created, to execute the observable we need to subscribe to it.

Operators

An operator is a pure function that takes in observable as input and the output is also an observable. You can use multiple operators on an observable data by using the pipe operator.

Subject

A subject is an observable sequence as well as an observer that can multicast, i.e. talk to many observers that have subscribed. The subject is a cold observable, i.e. the values will be shared between the observers that have been subscribed.

Schedulers

One important feature of RxPy is concurrency i.e. to allow the task to execute in parallel. To make that happen RxPy has two operators subscribe_on() and observe_on() that works with schedulers and will decide the execution of the subscribed task.

Advantages of using RxPY

The following are the advantages of RxPy:

- RxPY is an awesome library when it comes to the handling of async data streams and events. RxPY uses observables to work with reactive programming that deals with asynchronous data calls, callbacks and event-based programs.
- RxPY offers a huge collection of operators in mathematical, transformation, filtering, utility, conditional, error handling, join categories that makes life easy when used with reactive programming.
- Concurrency i.e. working of multiple tasks together is achieved using schedulers in RxPY.
- The performance is improved using RxPY as handling of async task and parallel processing is made easy.

Disadvantage of using RxPY

- Debugging the code with observables is a little difficult.

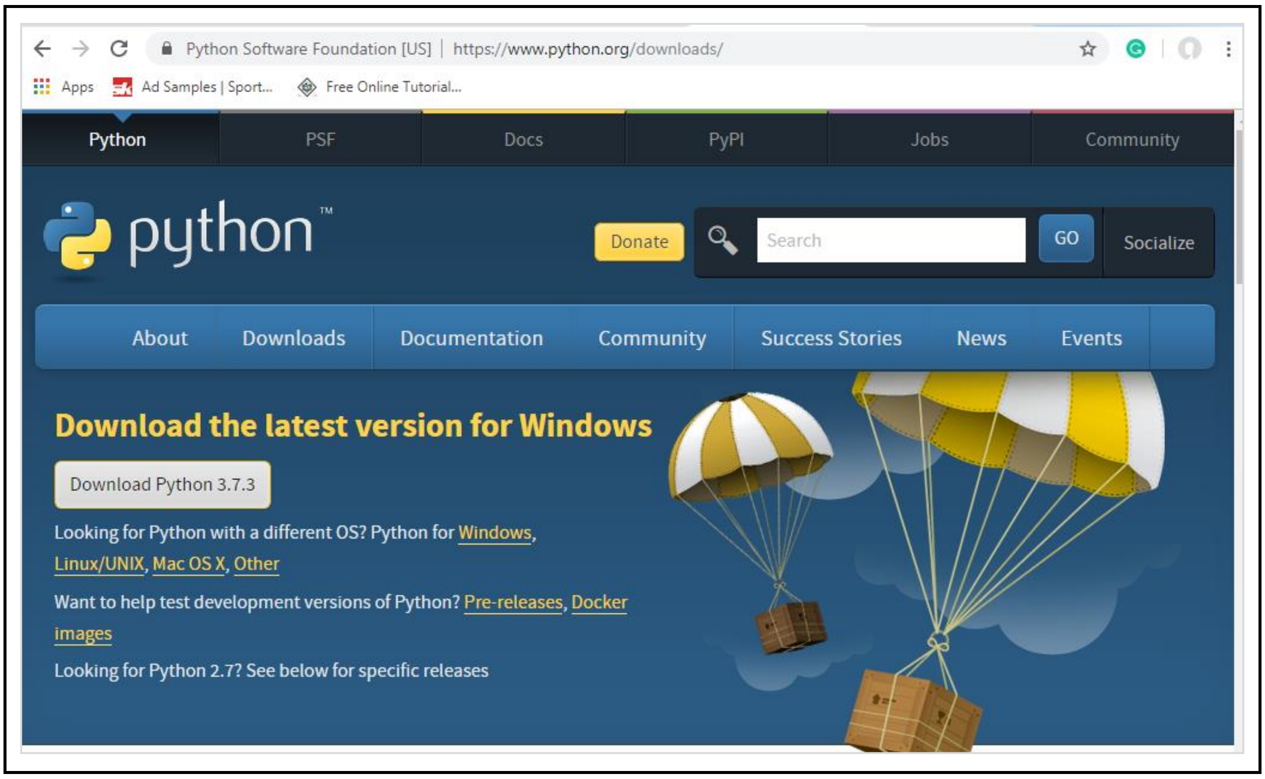
2. RxPY — Environment Setup

In this chapter, we will work on the installation of RxPy. To start working with RxPY, we need to install Python first. So, we are going to work on the following:

- Install Python
- Install RxPy

Installing Python

Go to the Python official site: https://www.python.org/downloads/ as shown below, and click on the latest version available for Windows, Linux/Unix, and mac os. Download Python as per your 64 or 32-bit OS available with you.



Download the latest version for Windows

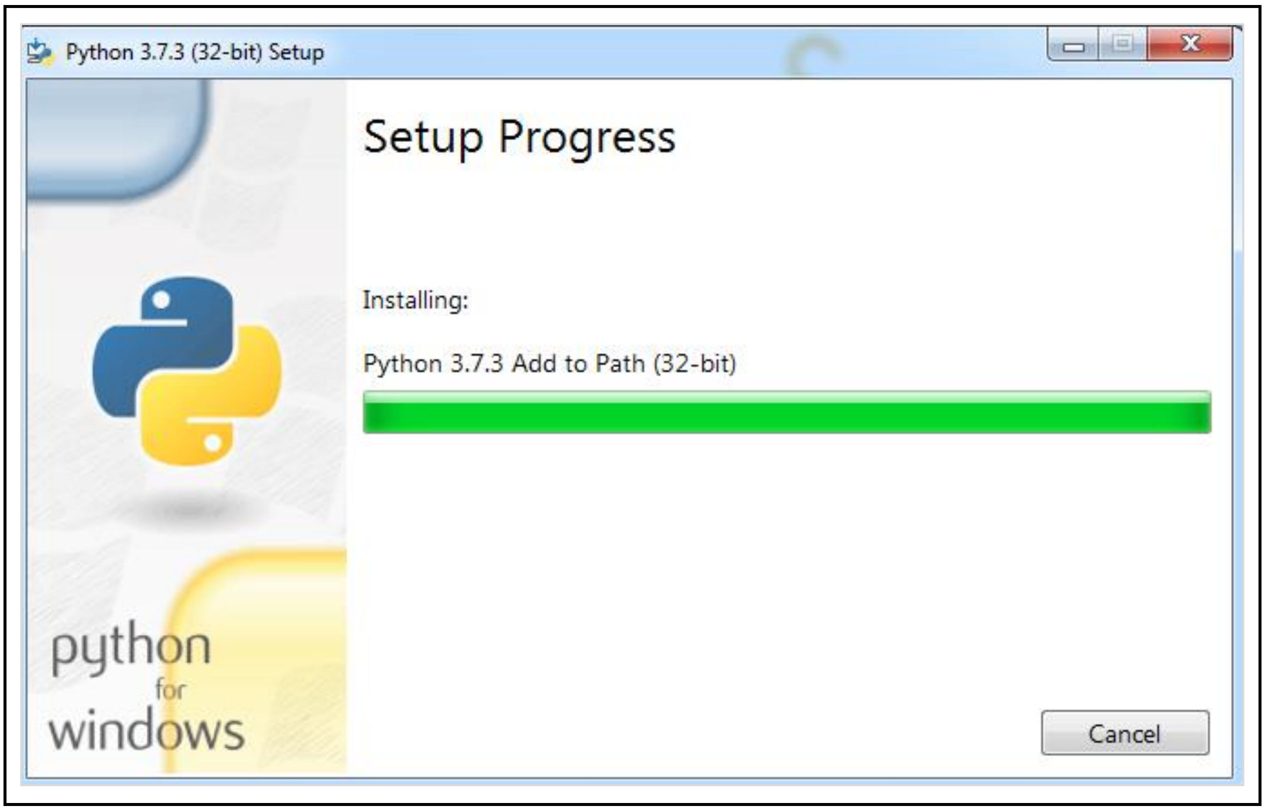
Download Python 3.7.3

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Pre-releases, Docker images

Looking for Python 2.7? See below for specific releases

Once you have downloaded, click on the **.exe file** and follow the steps to install python on your system.


Python 3.7.3 (32-bit) Setup

Setup Progress

Installing:

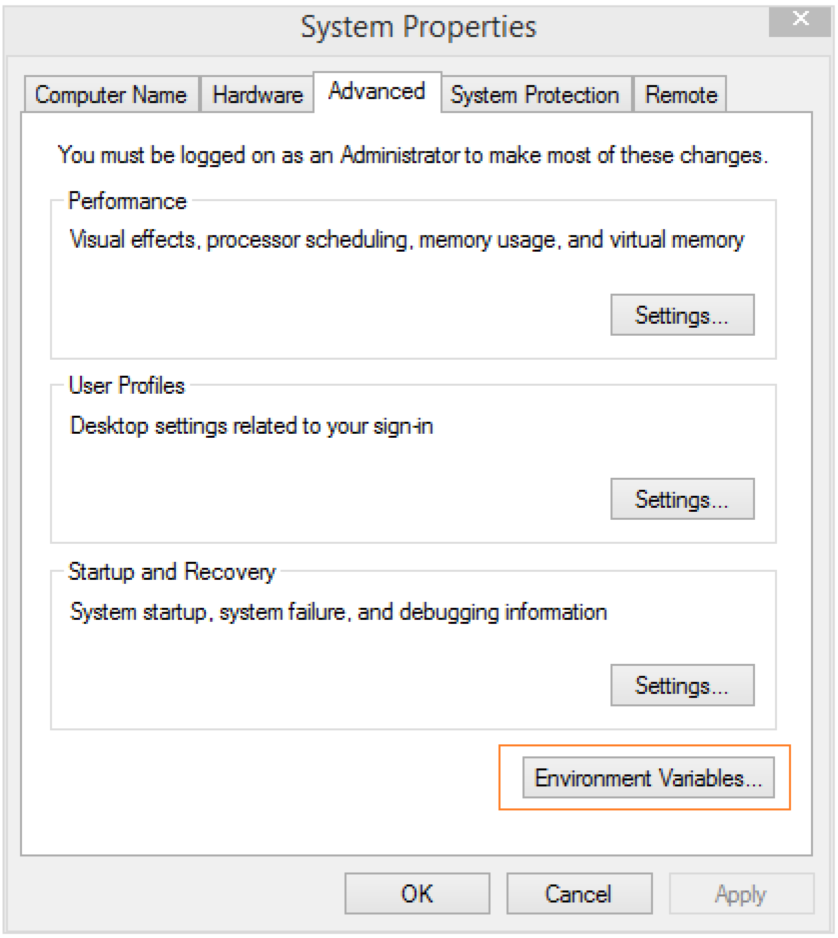
Python 3.7.3 Add to Path (32-bit)

The python package manager, i.e. pip will also get installed by default with the above installation. To make it work globally on your system, directly add the location of python to the PATH variable, the same is shown at the start of the installation, to remember to check the checkbox, which says ADD to PATH. In case, you forget to check it, please follow the below given steps to add to PATH.

To add to PATH follow the below steps:

Right-click on your Computer icon and click on properties -> Advanced System Settings.

It will display the screen as shown below:



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

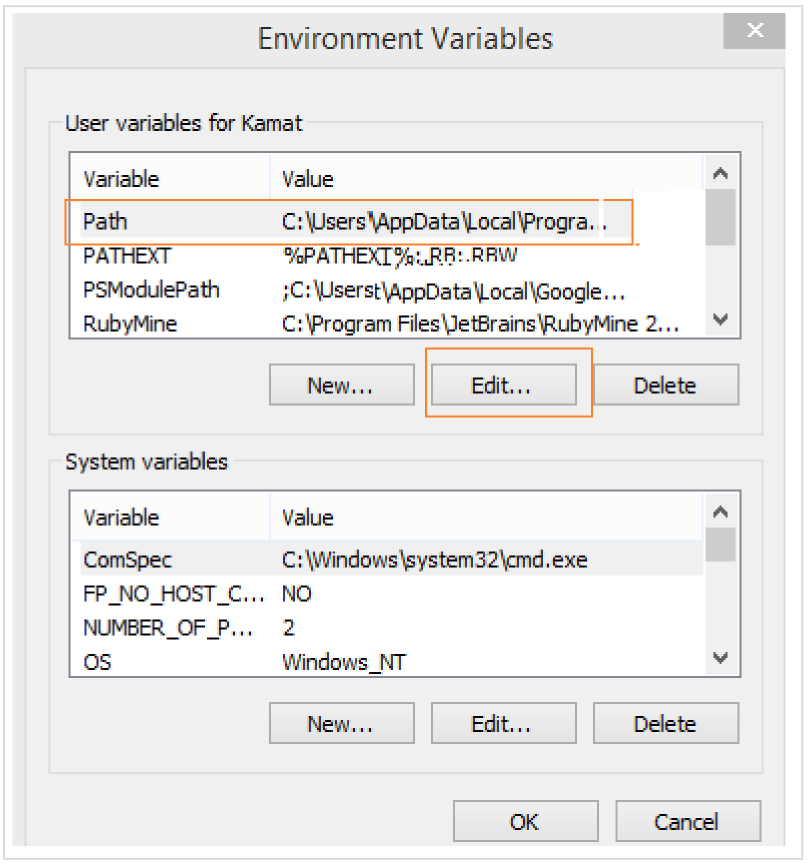
You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your sign-in

Startup and Recovery
System startup, system failure, and debugging information

Click on Environment Variables as shown above. It will display the screen as shown below:



User variables for Kamat

Variable	Value
Path	C:\Users\AppData\Local\Progra...
PATHEXT	%PATHEXT%;.RB;.RBW
PSModulePath	;C:\Userst\AppData\Local\Google...
RubyMine	C:\Program Files\JetBrains\RubyMine 2...

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT

Select Path and click on Edit button, add the location path of your python at the end. Now, let’s check the python version.

Checking for python version

```
E:\pyrx>python --version

Python 3.7.3
```

Install RxPY

Now, that we have python installed, we are going to install RxPy.

Once python is installed, python package manager, i.e. pip will also get installed. Following is the command to check pip version:

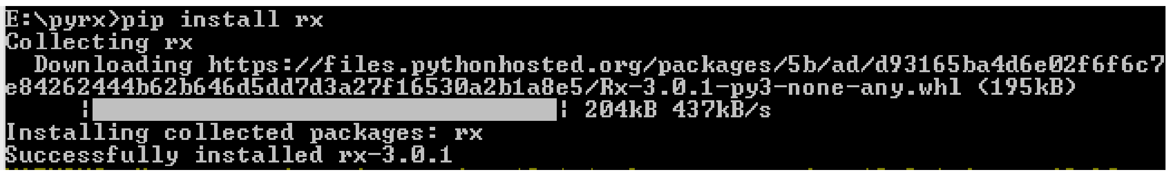
```
E:\pyrx>pip --version

pip 19.1.1 from c:\users\xxxx\appdata\local\programs\python\python37\lib\site-packages\pip (python 3.7)
```

We have pip installed and the version is **19.1.1**. Now, we will use pip to install RxPy

The command is as follows:

```
pip install rx
```



```
E:\pyrx>pip install rx
Collecting rx
  Downloading https://files.pythonhosted.org/packages/5b/ad/d93165ba4d6e02f6f6c7
e84262444b62b646d5dd7d3a27f16530a2b1a8e5/Rx-3.0.1-py3-none-any.whl (195kB)
     |████████████████████████████████| 204kB 437kB/s
Installing collected packages: rx
Successfully installed rx-3.0.1
```

3. RxPY — Latest Release Updates

In this tutorial, we are using RxPY version 3 and python version 3.7.3. The working of RxPY version 3 differs a little bit with the earlier version, i.e. RxPY version 1.

In this chapter, we are going to discuss the differences between the 2 versions and changes that need to be done in case you are updating Python and RxPY versions.

Observable in RxPY

In RxPy version 1, Observable was a separate class:

```
from rx import Observable
```

To use the Observable, you have to use it as follows:

```
Observable.of(1,2,3,4,5,6,7,8,9,10)
```

In RxPy version 3, Observable is directly a part of the rx package.

Example:

```
import rx
rx.of(1,2,3,4,5,6,7,8,9,10)
```

Operators in RxPy

In version 1, the operator was methods in the Observable class. For example, to make use of operators we have to import Observable as shown below:

```
from rx import Observable
```

The operators are used as Observable.operator, for example, as shown below:

```
Observable.of(1,2,3,4,5,6,7,8,9,10)\
    .filter(lambda i: i %2 == 0) \
    .sum() \
    .subscribe(lambda x: print("Value is {0}".format(x)))
```

In the case of RxPY version 3, operators are function and are imported and used as follows:

```
import rx

from rx import operators as ops
```

```
rx.of(1,2,3,4,5,6,7,8,9,10).pipe(
     ops.filter(lambda i: i %2 == 0),
    ops.sum()
).subscribe(lambda x: print("Value is {0}".format(x)))
```

Chaining Operators Using Pipe() method

In RxPy version 1, in case you had to use multiple operators on an observable, it had to be done as follows:

Example

```
from rx import Observable


Observable.of(1,2,3,4,5,6,7,8,9,10)\
    .filter(lambda i: i %2 == 0) \
    .sum() \
    .subscribe(lambda x: print("Value is {0}".format(x)))
```

But, in case of RxPY version 3, you can use pipe() method and multiple operators as shown below:

Example

```
import rx
from rx import operators as ops


rx.of(1,2,3,4,5,6,7,8,9,10).pipe(
    ops.filter(lambda i: i %2 == 0),
    ops.sum()
).subscribe(lambda x: print("Value is {0}".format(x)))
```

4. RxPY — Working with Observables

An observable, is a function that creates an observer and attaches it to the source where values are expected, for example, clicks, mouse events from a dom element, etc.

The topics mentioned below will be studied in detail in this chapter.

- Create Observables
- Subscribe and Execute an Observable

Create observables

To create an observable we will use **create()** method and pass the function to it that has the following items.

- **on_next()**: This function gets called when the Observable emits an item.
- **on_completed()**: This function gets called when the Observable is complete.
- **on_error()**: This function gets called when an error occurs on the Observable

To work with create() method first import the method as shown below:

```
from rx import create
```

Here is a working example, to create an observable:
testrx.py

```
from rx import create


deftest_observable(observer, scheduler):
    observer.on_next("Hello")
    observer.on_error("Error")
    observer.on_completed()


source = create(test_observable).
```

Subscribe and Execute an Observable

To subscribe to an observable, we need to use subscribe() function and pass the callback function on_next, on_error and on_completed.

Here is a working example:
testrx.py

```
from rx import create
```

```
deftest_observable(observer, scheduler):
    observer.on_next("Hello")
    observer.on_completed()

source = create(test_observable)

source.subscribe(
    on_next = lambda i: print("Got - {0}".format(i)),
    on_error = lambda e: print("Error : {0}".format(e)),
    on_completed = lambda: print("Job Done!"),
)
```

The subscribe() method takes care of executing the observable. The callback function **on_next**, **on_error** and **on_completed** has to be passed to the subscribe method. Call to subscribe method, in turn, executes the test_observable() function.

It is not mandatory to pass all three callback functions to the subscribe() method. You can pass as per your requirements the on_next(), on_error() and on_completed().

The lambda function is used for on_next, on_error and on_completed. It will take in the arguments and execute the expression given.

Here is the output, of the observable created:

```
E:\pyrx>python testrx.py
Got - Hello
Job Done!
```

5. RxPY — Operators

This chapter explains about the operators in RxPY in detail. These operators include: -

- Working with Operators
- Mathematical operators
- Transformation operators
- Filtering operators
- Error handling operators
- Utility operators
- Conditional operators
- Creation operators
- Connectable operators
- Combining operators

Reactive (Rx) python has almost lots of operators, that make life easy with python coding. You can use these multiple operators together, for example, while working with strings you can use map, filter, merge operators.

Working with Operators

You can work with multiple operators together using pipe() method. This method allows chaining multiple operators together.

Here, is a working example of using operators:

```
test = of(1,2,3) // an observable


subscriber = test.pipe(
    op1(),
    op2(),
    op3()
)
```

In the above example, we have created an observable using of() method that takes in values 1, 2 and 3. Now, on this observable, you can perform a different operation, using any numbers of operators using pipe() method as shown above. The execution of operators will go on sequentially on the observable given.

To work with operators, first import it as shown below:

```
from rx import of, operators as op
```

Here, is a working example:

testrx.py

```
from rx import of, operators as op


test = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)


sub1 = test.pipe(
    op.filter(lambda s: s%2==0),
    op.reduce(lambda acc, x: acc + x)
)
sub1.subscribe(lambda x: print("Sum of Even numbers is {0}".format(x)))
```

In the above example, there is a list of numbers, from which we are filtering even numbers using a filter operator and later adding it using a reduce operator.

Output

```
E:\pyrx>python testrx.py
Sum of Even numbers is 30
```

Here is a list of Operators, that we are going to discuss:

- Creating Observables
- Mathematical operators
- Transformation operators
- Filtering operators
- Error handling operators
- Utility operators
- Conditional
- Connectable
- Combining Observables

Creating Observables

Following are the observables, we are going to discuss in Creation category

Observable	Description
create	This method is used to create an observable.
empty	This observable will not output anything and directly emit the complete state.
never	This method creates an observable that will never reach the complete state.

throw	This method will create an observable that will throw an error.
from_	This method will convert the given array or object into an observable.
interval	This method will give a series of values produced after a timeout.
just	This method will convert given value into an observable.
range	This method will give a range of integers based on the input given.
repeat_value	This method will create an observable that will repeat the given value as per the count is given.
start	This method takes in a function as an input and returns an observable that will return value from the input function.
timer	This method will emit the values in sequence after the timeout is done.

create

This method is used to create an observable. It will have the observer method, i.e.

- **on_next()**: This function gets called, when the Observable emits an item.
- **on_completed()**: This function gets called, when the Observable is complete.
- **on_error()**: This function gets called, when an error occurs on the Observable.

Here, is a working example:

testrx.py

```
from rx import create

def test_observable(observer, scheduler):
    observer.on_next("Hello")
    observer.on_error("Error occured")
    observer.on_completed()

source = create(test_observable)

source.subscribe(
    on_next = lambda i: print("Got - {0}".format(i)),
    on_error = lambda e: print("Error : {0}".format(e)),
    on_completed = lambda: print("Job Done!"),
)
```

Here, is the **output** of the observable created:

```
E:\pyrx>python testrx.py
Got - Hello
Job Done!
```

empty

This observable will not output anything and directly emit the complete state.

Syntax

```
empty()
```

Return value

It will return an observable with no elements.

Example

```
from rx import empty


test = empty()


test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```
E:\pyrx>python testrx.py
Job Done!
```

never

This method creates an observable that will never reach the complete state.

Syntax

```
never()
```

Return value

It will return an observable that will never complete.

Example

```
from rx import never


test = never()


test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```
It does not show any output.
```

throw

This method will create an observable that will throw an error.

Syntax

```
throw(exception)
```

Parameters

exception: an object that has error details.

Return value

An observable is returned with error details.

Example

```
from rx import throw



test = throw(Exception('There is an Error!'))
```

```
test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```
E:\pyrx>python testrx.py
Error: There is an Error!
```

from_

This method will convert the given array or object into an observable.

Syntax

```
from_(iterator)
```

Parameters

iterator: This is an object or array.

Return value

This will return an observable for the given iterator.

Example

```
from rx import from_

test = from_([1,2,3,4,5,6,7,8,9,10])

test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```
E:\pyrx>python testrx.py
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
The value is 7
The value is 8
The value is 9
The value is 10
Job Done!
```

interval

This method will give a series of values produced after a timeout.

Syntax

```
interval(period)
```

Parameters

period: to start the integer sequence.

Return value

It returns an observable with all the values in sequential order.

Example

```
import rx
from rx import operators as ops

rx.interval(1).pipe(
    ops.map(lambda i: i * i)
).subscribe(lambda x: print("The value is {0}".format(x)))

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

Output

```
E:\pyrx>python testrx.py
Press any key to exit
The value is 0
The value is 1
The value is 4
The value is 9
The value is 16
The value is 25
The value is 36
The value is 49
The value is 64
The value is 81
The value is 100
The value is 121
The value is 144
The value is 169
The value is 196
The value is 225
The value is 256
The value is 289


The value is 324
The value is 361
The value is 400
```

just

This method will convert given value into an observable.

Syntax

```
just(value)
```

Parameters

value: to be converted to an observable.

Return value

It will return an observable with the given values.

Example

```
from rx import just
```

```
test = just([15, 25,50, 55])


test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```
E:\pyrx>python testrx.py
The value is [15, 25, 50, 55]
Job Done!
```

range

This method will give a range of integers based on the input given.

Syntax

```
range(start, stop=None)
```

Parameters

start: the first value from which the range will start.

stop: optional, the last value for the range to stop.

Return value

This will return an observable with integer value based on the input given.

Example

```
from rx import range


test = range(0,10)


test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```
E:\pyrx>python testrx.py
The value is 0
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
The value is 7
The value is 8
The value is 9
Job Done!
```

repeat_value

This method will create an observable that will repeat the given value as per the count is given.

Syntax

```
repeat_value(value=None, repeat_count=None)
```

Parameters

value: optional. The value to be repeated.

repeat_count: optional. The number of times the given value to be repeated.

Return value

It will return an observable that will repeat the given value as per the count is given.

Example

```
from rx import repeat_value




test = repeat_value(44,10)


test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```

E:\pyrx>python testrx.py
The value is 44
The value is 44
The value is 44
The value is 44
The value is 44
The value is 44
The value is 44
The value is 44
The value is 44
The value is 44
Job Done!
```

start

This method takes in a function as an input, and returns an observable that will return value from the input function.

Syntax

```
start(func)
```

Parameters

func: a function that will be called.

Return value

It returns an observable that will have a return value from the input function.

Example

```
from rx import start



```

```
test = start(lambda : "Hello World")



test.subscribe(
lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)),
on_completed = lambda: print("Job Done!")
)
```

Output

```
E:\pyrx>python testrx.py
The value is Hello World
Job Done!
```

timer

This method will emit the values in sequence after the timeout is done.

Syntax

```
timer(duetime)
```

Parameters

duetime: time after which it should emit the first value.

Return value

It will return an observable with values emitted after duetime.

Example

```
import rx
from rx import operators as ops

rx.timer(5.0, 10).pipe(
    ops.map(lambda i: i * i)
).subscribe(lambda x: print("The value is {0}".format(x)))

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

Output

```
E:\pyrx>python testrx.py
Press any key to exit
The value is 0


The value is 1
The value is 4
The value is 9


The value is 16
The value is 25
The value is 36
The value is 49
The value is 64
```

Mathematical operators

The operators we are going to discuss in Mathematical operator category are as follows: -

Operator	Description
average	This operator will calculate the average from the source observable given and output an observable that will have the average value.
concat	This operator will take in two or more observables and given a single observable with all the values in the sequence.
count	This operator takes in an Observable with values and converts it into an Observable that will have a single value. The count function takes in predicate function as an optional argument. The function is of type boolean and will add value to the output only if it satisfies the condition.
max	This operator will give an observable with max value from the source observable.
min	This operator will give an observable with min value from the source observable.

reduce	This operator takes in a function called accumulator function that is used on the values coming from the source observable, and it returns the accumulated values in the form of an observable, with an optional seed value passed to the accumulator function.
sum	This operator will return an observable with the sum of all the values from source observables.

average

This operator will calculate the average from the source observable given and output an observable that will have the average value.

Syntax

```
average()
```

Return value

It returns an observable that will have the average value.

Example

```
from rx import of, operators as op


test = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)


sub1 = test.pipe(
    op.average()
)
sub1.subscribe(lambda x: print("Average is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
Average is 5.5
```

concat

This operator will take in two or more observables and give a single observable with all the values in the sequence.

Syntax

```
concat(observable1, observable2...)
```

Parameters

Observables: List of observables to be concatenated.

Return value

An observable is returned with a single value merged from the values of the source observable.

Example

testrx.py

```
from rx import of, operators as op


test = of(2, 4, 6, 8, 10)
test2 = of(3,6,9,12,15)



sub1 = test.pipe(
    op.concat(test2)
)
sub1.subscribe(lambda x: print("Final value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
Final value is 2
Final value is 4
Final value is 6
Final value is 8
Final value is 10
Final value is 3
Final value is 6
Final value is 9
Final value is 12
Final value is 15
```

count

This operator takes in an observable with values, and converts it into an observable that will have a single value. The count function takes in predicate function as an optional argument. The function is of type Boolean, and will add value to the output only, if it satisfies the condition.

Syntax

```
count(predicate_function=None)
```

Parameters

The count function takes in predicate function as an optional argument. The function is of type Boolean, and will add value to the output only, if it satisfies the condition.

Return value

It will return an observable with a single value, i.e. the count from the source observable.

Example 1

```
from rx import of, operators as op



test = of(1,2,3, 4,5, 6,7, 8,9, 10)



sub1 = test.pipe(
    op.count()
)
sub1.subscribe(lambda x: print("The count is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The count is 10
```

Example 2: Using a predicate function

```
from rx import of, operators as op


test = of(1,2,3, 4,5, 6,7, 8,9, 10)

```

```
sub1 = test.pipe(
    op.count(lambda x : x %2 == 0)
)
sub1.subscribe(lambda x: print("The count of even numbers is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The count of even numbers is 5
```

max

This operator will give an observable with max value from the source observable.

Syntax

```
max(comparer_function=None)
```

Parameters

comparer_function: optional param. This function is used on source observables to compare values.

Return value

It returns an observable with max value from the source observable.

Example 1

```
from rx import of, operators as op

test = of(12,32,41,50,280,250)


sub1 = test.pipe(
    op.max()
)
sub1.subscribe(lambda x: print("Max value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
Max value is 280
```

Example 2: comparer_function

```
from rx import of, operators as op


test = of(12,32,41,50,280,250)


sub1 = test.pipe(
    op.max(lambda a, b : a - b)
)
sub1.subscribe(lambda x: print("Max value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
Max value is 280
```

min

This operator will give an observable with min value from the source observable.

Syntax

```
min(comparer_function=None)
```

Parameters

comparer_function: optional param. This function is used on source observables to compare values.

Return value

It returns an observable with min value from the source observable.

Example 1

```
from rx import of, operators as op




test = of(12,32,41,50,280,250)



sub1 = test.pipe(
    op.min()
)
sub1.subscribe(lambda x: print("Min value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
Min value is 12
```

Example 2: Using comparer_function

```
from rx import of, operators as op

test = of(12,32,41,50,280,250)

sub1 = test.pipe(
    op.min(lambda a, b : a - b)
)
sub1.subscribe(lambda x: print("Min value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
Min value is 12
```

reduce

This operator takes in a function called accumulator function, that is used on the values coming from the source observable, and it returns the accumulated values in the form of an observable, with an optional seed value passed to the accumulator function.

Syntax

```
reduce(accumulator_func, seed=notset)
```

Parameters

accumulator_func: A function that is used on the values coming from the source observable, and it returns the accumulated values in the form of an observable.

seed: optional. The default value is not set. It is the initial value, to be used inside the accumulator function.

Return value

It returns an observable, with a single value as output from the accumulator function applied on each value of the source observable.

Example

```
from rx import of, operators as op



test = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)


sub1 = test.pipe(
    op.reduce(lambda acc, x: acc + x)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 55
```

sum

This operator will return an observable with the sum of all the values from source observables.

Syntax

```
sum(key_mapper=none)
```

Parameters

key_mapper: optional. This is the function, that is applied to the values coming from the source observable.

Return value

It returns an observable with the sum of all the values from the source observable.

Example 1

```
from rx import of, operators as op


test = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
sub1 = test.pipe(
    op.sum()

)
sub1.subscribe(lambda x: print("The sum is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The sum is 55
```

Example 2: using key_mapper function

```
from rx import of, operators as op

test = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

sub1 = test.pipe(
    op.sum(lambda a: a+1)
)
sub1.subscribe(lambda x: print("The sum is {0}".format(x)))
```

Using key_mapper function, we are adding all the values by 1 and getting the sum of it.

```
E:\pyrx>python testrx.py
The sum is 65
```

Transformation operators

The operators we are going to discuss in the Transformation operator category are mentioned below:

Operator	Category
buffer	This operator will collect all the values from the source observable, and emit them at regular intervals once the given boundary condition is satisfied.
ground_by	This operator will group the values coming from the source observable based on the key_mapper function given.

map	This operator will change each value from the source observable into a new value based on the output of the mapper_func given.
scan	This operator will apply an accumulator function to the values coming from the source observable and return an observable with new values.

buffer

This operator will collect all the values, from the source observable and emit them at regular intervals once the given boundary condition is satisfied.

Syntax

```
buffer(boundaries)
```

Parameters

boundaries: The input is observable that will decide when to stop so that the collected values are emitted.

Return value

The return value is observable, that will have all the values collected from source observable based and that is time duration is decided by the input observable taken.

Example

```
from rx import of, interval, operators as op
from datetime import date
test = of(1, 2,3,4,5,6,7,8,9,10)

sub1 = test.pipe(
    op.buffer(interval(1.0))
)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python test1.py
The elements are [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

ground_by

This operator will group the values coming from the source observable based on the key_mapper function given.

Syntax

```
group_by(key_mapper)
```

Parameters

key_mapper: This function will take care of extracting keys from the source observable.

Return value

It returns an observable with values grouped based on the key_mapper function.

Example

```
from rx import from_, interval, operators as op


test = from_(["A", "B", "C", "D"])
sub1 = test.pipe(
    op.group_by(lambda v: v[0])


)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py


The element is <rx.core.observable.groupedobservable.GroupedObservable object
at
 0x000000C99A2E6550>
The element is <rx.core.observable.groupedobservable.GroupedObservable object
at


 0x000000C99A2E65C0>
The element is <rx.core.observable.groupedobservable.GroupedObservable object
at
 0x000000C99A2E6588>
The element is <rx.core.observable.groupedobservable.GroupedObservable object
at
```

```
0x000000C99A2E6550>
```

map

This operator will change each value from the source observable into a new value based on the output of the mapper_func given.

Syntax

```
map(mapper_func:None)
```

Parameters

mapper_func: (optional) It will change the values from the source observable based on the output coming from this function.

Example

```
from rx import of, interval, operators as op
test = of(1, 2,3,4,5,6,7,8,9,10)




sub1 = test.pipe(
    op.map(lambda x :x*x)
)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The element is 1
The element is 4
The element is 9
The element is 16
The element is 25

The element is 36
The element is 49
The element is 64

The element is 81
The element is 100
```

scan

This operator will apply an accumulator function to the values coming from the source observable and return an observable with new values.

Syntax

```
scan(accumulator_func, seed=NotSet)
```

Parameters

accumulator_func: This function is applied to all the values from the source observable.

seed:(optional) The initial value to be used inside the accumular_func.

Return value

This operator will return an observable that will have new values based on the accumulator function applied on each value of the source observable.

Example

```
from rx import of, interval, operators as op
test = of(1, 2,3,4,5,6,7,8,9,10)


sub1 = test.pipe(
    op.scan(lambda acc, a: acc + a, 0)


)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py


The element is 1
The element is 3
The element is 6
The element is 10
The element is 15
The element is 21
The element is 28
The element is 36
The element is 45
The element is 55
```

Filtering operators

The operators we are going to discuss in Filtering operator category are given below:

Operator	Category
debounce	This operator will give the values from the source observable, until the timespan given and ignore the rest of the time passes.
distinct	This operator will give all the values that are distinct from the source observable.
element_at	This operator will give an element from the source observable for the index given.
filter	This operator will filter values from the source observable based on the predicate function given.
first	This operator will give the first element from the source observable.
ignore_elements	This operator will ignore all the values from the source observable and only execute calls to complete or error callback functions.
last	This operator will give the last element from the source observable.
skip	This operator will give back an observable that will skip the first occurrence of count items taken as input.
skip_last	This operator will give back an observable that will skip the last occurrence of count items taken as input.
take	This operator will give a list of source values in continuous order based on the count given.

take_last	This operator will give a list of source values in continuous order from last based on the count given.

debounce

This operator will give the values from the source observable, until the timespan given and ignore the rest of the values if time passes.

Syntax

```
debounce(duetime)
```

Parameters

duetime: this will value in seconds or instances of time, the duration that will decide the values to be returned from the source observable.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6,7,8,9,10)

sub1 = test.pipe(
    op.debounce(2.0)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 10
```

distinct

This operator will give all the values that are distinct from the source observable.

Syntax

```
distinct()
```

Return value

It will return an observable, where it will have distinct values from the source observable.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)



sub1 = test.pipe(
    op.distinct()
)
sub1.subscribe(lambda x: print("The distinct value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The distinct value is 1


The distinct value is 6
The distinct value is 15
The distinct value is 10


The distinct value is 40
The distinct value is 58
The distinct value is 20
```

element_at

This operator will give an element from the source observable for the index given.

Syntax

```
element_at(index)
```

Parameters

index: the number starting from zero for which you need the element from the source observable.

Return value

It will return an observable with the value from the source observable with the index given.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)

sub1 = test.pipe(
    op.element_at(5)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 6
```

filter

This operator will filter values from the source observable based on the predicate function given.

Syntax

```
filter(predicate_func)
```

Parameters

predicate_func: This function will decide the values to be filtered from the source observable.

Return value

It will return an observable that will have the filtered values from the source observable based on predicate function.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)

sub1 = test.pipe(
    op.filter(lambda x : x %2==0)
)
sub1.subscribe(lambda x: print("The filtered value is {0}".format(x)))
```

In the example, we have filtered all the even numbers.

Output

```
E:\pyrx>python testrx.py
The filtered value is 6
The filtered value is 10
The filtered value is 6
The filtered value is 40
The filtered value is 10
The filtered value is 58
The filtered value is 20
The filtered value is 40
```

first

This operator will give the first element from the source observable.

Syntax

```
first(predicate_func=None)
```

Parameters

predicate_func: (optional) This function will decide the first element to be picked based on the condition if passed.

Return value

It will return an observable with the first value from the source observable.

Example

```
from rx import of, operators as op
from datetime import date


test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)


sub1 = test.pipe(
    op.first()
)
sub1.subscribe(lambda x: print("The first element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The first element is 1
```

Example 2: using predicate_func

```
from rx import of, operators as op
from datetime import date


test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)


sub1 = test.pipe(
    op.first(lambda x : x%2==0)
)
sub1.subscribe(lambda x: print("The first element is {0}".format(x)))
```

Output

```
E:\pyrx>python test1.py
The first element is 6
```

ignore_elements

This operator will ignore all the values from the source Observable, and only execute calls to complete or error callback functions.

Syntax

```
ignore_elements()
```

Return value

It returns an observable that will call complete or error based on the source observable.

Example

```
from rx import of, operators as op
from datetime import date


test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)



sub1 = test.pipe(
    op.ignore_elements()
)
sub1.subscribe(lambda x: print("The first element is {0}".format(x)),
lambda e: print("Error : {0}".format(e)),
lambda: print("Job Done!"))
```

Output

```
E:\pyrx>python testrx.py
Job Done!
```

last

This operator will give the last element from the source observable.

Syntax

```
last(predicate_func=None)
```

Parameters

predicate_func: (optional) This function will decide the last element to be picked based on the condition if passed.

Return value

It will return an observable with the last value from the source observable.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)

sub1 = test.pipe(
    op.last()
)
sub1.subscribe(lambda x: print("The last element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The last element is 40
```

Example 2: using predicate_func

```
from rx import of, operators as op
from datetime import date

test = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40)

sub1 = test.pipe(
    op.last(lambda x : x%2==0)
)
sub1.subscribe(lambda x: print("The last element is {0}".format(x)))
```

Output

```
E:\pyrx>python test1.py
The last element is 40
```

skip

This operator will give back an observable, that will skip the first occurrence of count items taken as input.

Syntax

```
skip(count)
```

Parameters

count: The count is the number of times that the items will be skipped from the source observable.

Return value

It will return an observable that skips values based on the count given.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1, 2,3,4,5,6,7,8,9,10)

sub1 = test.pipe(
    op.skip(5)
)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The element is 6


The element is 7
The element is 8
The element is 9
The element is 10
```

skip_last

This operator will give back an observable, that will skip the last occurrence of count items taken as input.

Syntax

```
skip_last(count)
```

Parameters

count: The count is the number of times, that the items will be skipped from the source observable.

Return value

It will return an observable that skips values based on the count given from last.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1, 2,3,4,5,6,7,8,9,10)


sub1 = test.pipe(
    op.skip_last(5)
)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The element is 1
The element is 2
The element is 3
The element is 4
The element is 5
```

take

This operator will give a list of source values in continuous order based on the count given.

Syntax

```
take(count)
```

Parameters

count: The count is the number of items, that will be given from the source observable.

Return value

It will return an observable that has the values in continuous order based on count given.

Example

```
from rx import of, operators as op


from datetime import date
test = of(1, 2,3,4,5,6,7,8,9,10)


sub1 = test.pipe(
    op.take(5)
)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py


The element is 1
The element is 2
The element is 3
The element is 4
The element is 5
```

take_last

This operator will give a list of source values, in continuous order from last based on the count given.

Syntax

```
take_last(count)
```

Parameters

count: The count is the number of items, that will be given from the source observable.

Return value

It will return an observable, that has the values in continuous order from last based on count given.

Example

```
from rx import of, operators as op


from datetime import date


test = of(1, 2,3,4,5,6,7,8,9,10)


sub1 = test.pipe(
    op.take_last(5)
)
sub1.subscribe(lambda x: print("The element is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py


The element is 6
The element is 7
The element is 8
The element is 9
The element is 10
```

Error handling operators

The operators we are going to discuss in the Error handling operator category are: -

Operator	Description
catch	This operator will terminate the source observable when there is an exception.
retry	This operator will retry on the source observable when there is an error and once the retry count is done it will terminate.

catch

This operator will terminate the source observable when there is an exception.

Syntax

```
catch(handler)
```

Parameters

handler: This observable will be emitted, when the source observable has an error.

Return value

It will return an observable, that will have values from the source observable before the error, followed by values from the handler observable.

Example

```
from rx import of, operators as op
from datetime import date


test = of(1,2,3,4,5,6)


handler = of(11,12,13,14)


def casetest(e):
    if (e==4):
        raise Exception('err')
    else:


        return e


sub1 = test.pipe(
    op.map(lambda e : casetest(e)),
    op.catch(handler)

)

sub1.subscribe(lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)))
```

In this example, we have created an exception, when the source value from the observable is 4, so the first observable is terminated there and later followed by the values from the handler.

Output

```
E:\pyrx>python testrx.py
The value is 1
The value is 2
The value is 3
The value is 11
The value is 12
The value is 13
The value is 14
```

retry

This operator will retry on the source observable when there is an error and once the retry count is done it will terminate.

Syntax

```
retry(count)
```

Parameters

count: the number of times to retry if there is an error from the source observable.

Return value

It will return an observable from the source observable in repeated sequence as per the retry count given.

Example

```
from rx import of, operators as op
test = of(1,2,3,4,5,6)
```

```
def casetest(e):
    if (e==4):
        raise Exception('There is error cannot proceed!')
    else:



        return e

sub1 = test.pipe(
    op.map(lambda e : casetest(e)),
    op.retry(2)
)

sub1.subscribe(lambda x: print("The value is {0}".format(x)),
on_error = lambda e: print("Error : {0}".format(e)))
```

Output

```
E:\pyrx>python testrx.py
The value is 1
The value is 2
The value is 3
The value is 1
The value is 2
The value is 3
Error: There is error cannot proceed!
```

Utility operators

The following are the operators we are going to discuss in the Utility operator category.

Operator	Description
delay	This operator will delay the source observable emission as per the time or date is given.
materialize	This operator will convert the values from the source observable with the values

	emitted in the form of explicit notification values.
time_interval	This operator will give the time elapsed between the values from the source observable.
timeout	This operator will give all the values from the source observable after the elapsed time or else will trigger an error.
timestamp	This operator will attach a timestamp to all the values from the source observable.

delay

This operator will delay the source observable emission as per the time or date given.

Syntax

```
delay(timespan)
```

Parameters

timespan: this will be the time in seconds or date.

Return value

It will give back an observable with source values emitted after the timeout.

Example

```
from rx import of, operators as op
import datetime


test1 = of(1,2,3,4,5)


sub1 = test1.pipe(
    op.delay(5.0)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))


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

Output

```
E:\pyrx>python testrx.py
```

```
Press any key to exit
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
```

materialize

This operator will convert the values from the source observable with the values emitted in the form of explicit notification values.

Syntax

```
materialize()
```

Return value

This will give back an observable with the values emitted in the form of explicit notification values.

Example

```
from rx import of, operators as op
import datetime

test1 = of(1,2,3,4,5)

sub1 = test1.pipe(
    op.materialize()
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is OnNext(1.0)
The value is OnNext(2.0)
The value is OnNext(3.0)
The value is OnNext(4.0)
The value is OnNext(5.0)
The value is OnCompleted()
```

time_interval

This operator will give the time elapsed between the values from the source observable.

Syntax

```
time_interval()
```

Return value

It will return an observable that will have the time elapsed, between the source value emitted.

Example

```
from rx import of, operators as op
from datetime import date


test = of(1,2,3,4,5,6)


sub1 = test.pipe(


    op.time_interval()
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is TimeInterval(value=1,
interval=datetime.timedelta(microseconds=1000
))
The value is TimeInterval(value=2, interval=datetime.timedelta(0))
The value is TimeInterval(value=3, interval=datetime.timedelta(0))
The value is TimeInterval(value=4,
interval=datetime.timedelta(microseconds=1000
))
The value is TimeInterval(value=5, interval=datetime.timedelta(0))
The value is TimeInterval(value=6, interval=datetime.timedelta(0))
```

timeout

This operator will give all the values from the source observable, after the elapsed time or else will trigger an error.

Syntax

```
timeout(duetime)
```

Parameters

duetime: the time given in seconds.

Return value

It will give back on observable with all values from the source observable.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6)

sub1 = test.pipe(
    op.timeout(5.0)



)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
```

timestamp

This operator will attach a timestamp to all the values from the source observable.

Syntax

```
timestamp()
```

Return value

It will give back an observable with all values from the source observable along with a timestamp.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6)

sub1 = test.pipe(
    op.timestamp()
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is Timestamp(value=1, timestamp=datetime.datetime(2019, 11, 4, 4, 57,
44, 667243))
The value is Timestamp(value=2, timestamp=datetime.datetime(2019, 11, 4, 4, 57,
44, 668243))
The value is Timestamp(value=3, timestamp=datetime.datetime(2019, 11, 4, 4, 57,
44, 668243))

The value is Timestamp(value=4, timestamp=datetime.datetime(2019, 11, 4, 4, 57,

44, 668243))

The value is Timestamp(value=5, timestamp=datetime.datetime(2019, 11, 4, 4, 57,
44, 669243))
The value is Timestamp(value=6, timestamp=datetime.datetime(2019, 11, 4, 4, 57,
44, 669243))
```

Conditional and Boolean Operators

The operators we are going to discuss in Conditional and Boolean Operator category are as given below:

Operator	Description
all	This operator will check if all the values from the source observable satisfy the condition given.
contains	This operator will return an observable with the value true or false if the given value is present and if it is the value of the source observable.
default_if_empty	This operator will return a default value if the source observable is empty.
sequence_equal	This operator will compare two sequences of observables or an array of values and return an observable with the value true or false.
skip_until	This operator will discard values from the source observable until the second observable emits a value.
skip_while	This operator will return an observable with values from the source observable that satisfies the condition passed.
take_until	This operator will discard values from the source observable after the second observable emits a value or is terminated.
take_while	This operator will discard values from the source observable when the condition fails.

all

This operator will check if all the values from the source observable satisfy the condition given.

Syntax

```
all(predicate)
```

Parameters

predicate: boolean. This function will be applied to all the values, from the source observable and will return true or false based on the condition given.

Return value

The return value is an observable, which will have the boolean value true or false, based on the condition applied on all the values of the source observable.

Example 1

```
from rx import of, operators as op



test = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)


sub1 = test.pipe(
    op.all(lambda a: a<10)
)
sub1.subscribe(lambda x: print("The result is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The result is False
```

Example 2

```
from rx import of, operators as op

test = of(1, 2, 3, 4, 5, 6, 7, 8, 9)

sub1 = test.pipe(
    op.all(lambda a: a<10)
)
sub1.subscribe(lambda x: print("The result is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The result is True
```

contains

This operator will return an observable with the value true or false if the given value is present is the values of the source observable.

Syntax

```
contains(value, comparer=None)
```

Parameters

value: The value to be checked if present in the source observable

comparer: optional. This is a comparer function to be applied to the values present in the source observable for comparison.

Example

```
from rx import of, operators as op



test = of(17, 25, 34, 56, 78)


sub1 = test.pipe(
    op.contains(34)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is True
```

Example 2: Using comparer

```
from rx import of, operators as op



test = of(17, 25, 34, 56, 78)


sub1 = test.pipe(
    op.contains(34, lambda x, y: x == y)


)
sub1.subscribe(lambda x: print("The valus is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is True
```

default_if_empty

This operator will return a default value if the source observable is empty.

Syntax

```
default_if_empty(default_value=None)
```

Parameters

default_value: optional. It will give the output, as None is nothing is passed as default_value, else it will give whatever value passed.

Return value

It will return an observable with a default value if the source observable is empty.

Example 1

```
from rx import of, operators as op



test = of()


sub1 = test.pipe(
    op.default_if_empty()
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is None
```

Example 2: default_value passed

```
from rx import of, operators as op


test = of()

```

```
sub1 = test.pipe(

    op.default_if_empty("Empty!")
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is Empty!
```

sequence_equal

This operator will compare two sequences of observables, or an array of values and return an observable with the value true or false.

Syntax

```
sequence_equal(second_seq, comparer=None)
```

Parameters

second_seq: observable or array to be compared with the first observable.

comparer: optional. Comparer function to be applied to compare values in both sequences.

Example

```
from rx import of, operators as op

test = of(1,2,3)

test1 = of(1,2,3)

sub1 = test.pipe(
    op.sequence_equal(test1)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is True
```

Example: using a comparer function

```
from rx import of, operators as op


test = of(1,2,3)
test1 = of(1,2,3)


sub1 = test.pipe(
    op.sequence_equal(test1, lambda x, y : x == y)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is True
```

skip_until

This operator will discard values from the source observable until the second observable emits a value.

Syntax

```
skip_until(observable)
```

Parameters

observable: the second observable which when emits a value will trigger the source observable.

Return value

It will return an observable which will have values from the source observable until the second observable emits a value.

Example

```
from rx import interval,range, operators as op
from datetime import date
test = interval(0)
test1 = range(10)


sub1 = test1.pipe(
    op.skip_until(test)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 0
The value is 1

The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
The value is 7
The value is 8
The value is 9
```

skip_while

This operator will return an observable with values from the source observable that satisfies the condition passed.

Syntax

```
skip_while(predicate_func)
```

Parameters

predicate_func: This function will be applied to all the values of the source observable, and return the values which satisfy the condition.

Return value

It will return an observable with values from the source observable that satisfies the condition passed.

Example

```
from rx import of, operators as op
from datetime import date

test = of(1,2,3,4,5,6,7,8,9,10)
```

```

sub1 = test.pipe(
    op.skip_while(lambda x : x<5)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 5
The value is 6
The value is 7

The value is 8
The value is 9
The value is 10
```

take_until

This operator will discard values from the source observable after the second observable emits a value or is terminated.

Syntax

```
take_until(observable)
```

Parameters

observable: the second observable which, when emits a value will terminate the source observable.

Return value

It will return an observable, which will have values from the source observable only, when the second observable used emits a value.

Example

```
from rx import timer,range, operators as op
from datetime import date
test = timer(0.01)
test1 = range(500)

sub1 = test1.pipe(
    op.take_until(test)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

In this example, you will get the values emitted from range. But, once the timer is done, it will stop the source observable from emitting further.

Output

```
E:\pyrx>python testrx.py
The value is 0


The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
The value is 7


The value is 8
The value is 9
The value is 10
The value is 11
The value is 12
The value is 13


The value is 14
The value is 15
The value is 16
The value is 17
The value is 18
The value is 19
The value is 20
The value is 21
The value is 22
The value is 23
The value is 24
The value is 25
The value is 26
```

take_while

This operator will discard values from the source observable when the condition fails.

Syntax

```
take_while(predicate_func)
```

Parameters

predicate_func: this function will evaluate each value of the source observable.

Return value

It will return an observable with values till the predicate function satisfies.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6,7,8,9,10)

sub1 = test.pipe(
    op.take_while(lambda a : a<5)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 1

The value is 2
The value is 3
The value is 4
```

Connectable Operators

The operators we are going to discuss in Connectable Operator category are:

Operator	Description

publish	This method will convert the observable into a connectable observable.
ref_count	This operator will make the observable a normal observable.
replay	This method works similar to the replaySubject. This method will return the same values, even if the observable has already emitted and some of the subscribers are late in subscribing.

publish

This method will convert the observable into a connectable observable.

Syntax

```
publish(mapper=None)
```

Parameters

mapper: optional. A function used to multicast source values multiple times, without having to do multiple subscriptions.

Example

```
from rx import create, range, operators as op
import random


def test_observable(observer, scheduler):
    observer.on_next(random.random())
    observer.on_completed()


source = create(test_observable).pipe(op.publish())
test1 = source.subscribe(on_next = lambda i: print("From subscriber 1 -
{0}".format(i)))


test2 = source.subscribe(on_next = lambda i: print("From subscriber 2 –


{0}".format(i)))
source.connect()
```

Output

```
E:\pyrx>python testrx.py
```

```
From subscriber 1 - 0.14751607273318490
From subscriber 2 - 0.1475160727331849
```

ref_count

This operator will make the observable a normal observable.

Syntax

```
ref_count()
```

Example

```
from rx import create, operators as op
import random

def test_observable(observer, scheduler):
    observer.on_next(random.random())

source = create(test_observable).pipe(op.publish(),op.ref_count())
test1 = source.subscribe(on_next = lambda i: print("From subscriber 1 - {0}".format(i)))
test2 = source.subscribe(on_next = lambda i: print("From subscriber 2 - {0}".format(i)))
```

Output

```
E:\pyrx>python testrx.py
From subscriber 1 - 0.8230640432381131
```

replay

This method works similar to the replaySubject. This method will return the same values, even if the observable has already emitted, and some of the subscribers are late in subscribing.

Syntax

```
replay()
```

Example

```
from rx import create, range, operators as op
import random
```

```
from threading import Timer


def test_observable(observer, scheduler):
    observer.on_next(random.random())
    observer.on_completed()


source = create(test_observable).pipe(op.replay())
test1 = source.subscribe(on_next = lambda i: print("From subscriber 1 -
{0}".format(i)))
test2 = source.subscribe(on_next = lambda i: print("From subscriber 2 -
{0}".format(i)))


source.connect()


print("subscriber called after delay ")


def last_subscriber():
    test3 = source.subscribe(on_next = lambda i: print("From subscriber 3 -
{0}".format(i)))


t = Timer(5.0, last_subscriber)
t.start()
```

Output

```
E:\pyrx>python testrx.py
From subscriber 1 - 0.8340998157725388
From subscriber 2 - 0.8340998157725388
subscriber called after delay
From subscriber 3 - 0.8340998157725388
```

Combining Operators

The following are the operators we are going to discuss in the Combining operator category.

Operator	Description
combine_latest	This operator will create a tuple for the observable given as input.

merge	This operator will merge given observables.
start_with	This operator will take in the given values and add at the start of the source observable return back the full sequence.
zip	This operator returns an observable with values in a tuple form which is formed by taking the first value of the given observable and so on.

combine_latest

This operator will create a tuple, for the observable given as input.

Syntax

```
combine_latest(observable1,observable2,.....)
```

Parameters

Observable: An observable.

Return value

It returns an observable with the values from the source observable converted to a tuple.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6)
test2 = of(11,12,13,14,15,16)
test3 = of(111,112,113,114,115,116)



sub1 = test.pipe(
    op.combine_latest(test2, test3)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is (6, 16, 111)
The value is (6, 16, 112)



The value is (6, 16, 113)
The value is (6, 16, 114)
The value is (6, 16, 115)
The value is (6, 16, 116)
```

merge

This operator will merge given observables.

Syntax

```
merge(observable)
```

Parameters

Observable: an observable.

Return value

It will return an observable with one sequence from the given observables.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6)
test2 = of(11,12,13,14,15,16)

sub1 = test.pipe(
    op.merge(test2)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is 1

The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
The value is 11
The value is 12
The value is 13


The value is 14


The value is 15
The value is 16
```

start_with

This operator will take in the given values, and add at the start of the source observable return back the full sequence.

Syntax

```
start_with(values)
```

Parameters

values: The values you want to prefix at the start.

Return value

It returns an observable with given values prefixed at the start followed by the values from the source observable.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6)

sub1 = test.pipe(
    op.start_with(-2,-1,0)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is -2
The value is -1


The value is 0
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
```

zip

This operator returns an observable with values in a tuple form, which is formed by taking the first value of the given observable and so on.

Syntax

```
zip(observable1, observable2...)
```

Parameters

Observable: an observable

Return value

It returns an observable with values in tuple format.

Example

```
from rx import of, operators as op
from datetime import date
test = of(1,2,3,4,5,6)
test1 = of(4,8,12,16,20)
test2 = of(5,10,15,20,25)


sub1 = test.pipe(
    op.zip(test1, test2)
)
sub1.subscribe(lambda x: print("The value is {0}".format(x)))
```

Output

```
E:\pyrx>python testrx.py
The value is (1, 4, 5)
The value is (2, 8, 10)
The value is (3, 12, 15)
The value is (4, 16, 20)
The value is (5, 20, 25)
```

6.RxPY — Working with Subject

A subject is an observable sequence, as well as, an observer that can multicast, i.e. talk to many observers that have subscribed.

We are going to discuss the following topics on subject:

- Create a subject
- Subscribe to a subject
- Passing data to subject
- BehaviorSubject
- ReplaySubject
- AsyncSubject

Create a subject

To work with a subject, we need to import Subject as shown below:

```
from rx.subject import Subject
```

You can create a subject-object as follows:

```
subject_test = Subject()
```

The object is an observer that has three methods:

- on_next(value)
- on_error(error) and
- on_completed()

Subscribe to a Subject

You can create multiple subscription on the subject as shown below:

```
subject_test.subscribe(
 lambda x: print("The value is {0}".format(x))
)
subject_test.subscribe(
 lambda x: print("The value is {0}".format(x))
)
```

Passing Data to Subject

You can pass data to the subject created using the on_next(value) method as shown below:

```
subject_test.on_next("A")
subject_test.on_next("B")
```

The data will be passed to all the subscription, added on the subject.

Here, is a working example of the subject.

Example

```
from rx.subject import Subject


subject_test = Subject()


subject_test.subscribe(
 lambda x: print("The value is {0}".format(x))
)
subject_test.subscribe(
 lambda x: print("The value is {0}".format(x))
)


subject_test.on_next("A")
subject_test.on_next("B")
```

The subject_test object is created by calling a Subject(). The subject_test object has reference to on_next(value), on_error(error) and on_completed() methods. The output of the above example is shown below:

Output

```
E:\pyrx>python testrx.py
The value is A
The value is A
The value is B
The value is B
```

We can use the on_completed() method, to stop the subject execution as shown below.

Example

```
from rx.subject import Subject



subject_test = Subject()


subject_test.subscribe(
 lambda x: print("The value is {0}".format(x))
)
subject_test.subscribe(
 lambda x: print("The value is {0}".format(x))
)


subject_test.on_next("A")
subject_test.on_completed()
subject_test.on_next("B")
```

Once we call complete, the next method called later is not invoked.

Output

```
E:\pyrx>python testrx.py
The value is A
The value is A
```

Let us now see, how to call on_error(error) method.

Example

```
from rx.subject import Subject


subject_test = Subject()


subject_test.subscribe(
on_error = lambda e: print("Error : {0}".format(e))
)
subject_test.subscribe(
on_error = lambda e: print("Error : {0}".format(e))
)


subject_test.on_error(Exception('There is an Error!'))
```

Output

```
E:\pyrx>python testrx.py


Error: There is an Error!

Error: There is an Error!
```

BehaviorSubject

BehaviorSubject will give you the latest value when called. You can create behavior subject as shown below:

```
from rx.subject import BehaviorSubject


behavior_subject = BehaviorSubject("Testing Behaviour Subject");  //
initialized the behaviour subject with value:Testing Behaviour Subject
```

Here, is a working example to use Behaviour Subject

Example

```
from rx.subject import BehaviorSubject


behavior_subject = BehaviorSubject("Testing Behaviour Subject");


behavior_subject.subscribe(

  lambda x: print("Observer A : {0}".format(x))

)


behavior_subject.on_next("Hello")


behavior_subject.subscribe(

  lambda x: print("Observer B : {0}".format(x))

)

behavior_subject.on_next("Last call to Behaviour Subject")
```

Output

```
E:\pyrx>python testrx.py

Observer A : Testing Behaviour Subject

Observer A : Hello

Observer B : Hello
Observer A : Last call to Behaviour Subject
Observer B : Last call to Behaviour Subject
```

Replay Subject

A replaysubject is similar to behavior subject, wherein, it can buffer the values and replay the same to the new subscribers. Here, is a working example of replay subject.

Example

```
from rx.subject import ReplaySubject


replay_subject = ReplaySubject(2)


replay_subject.subscribe(lambda x: print("Testing Replay Subject A:
{0}".format(x)))


replay_subject.on_next(1)
replay_subject.on_next(2)
replay_subject.on_next(3)



replay_subject.subscribe(lambda x: print("Testing Replay Subject B:
{0}".format(x)));


replay_subject.on_next(5)
```

The buffer value used is 2 on the replay subject. So, the last two values will be buffered and used for the new subscribers called.

Output

```
E:\pyrx>python testrx.py
Testing Replay Subject A: 1
Testing Replay Subject A: 2
Testing Replay Subject A: 3
Testing Replay Subject B: 2
Testing Replay Subject B: 3
Testing Replay Subject A: 5
Testing Replay Subject B: 5
```

AsyncSubject

In the case of AsyncSubject, the last value called is passed to the subscriber, and it will be done only after the complete() method is called.

Example

```
from rx.subject import AsyncSubject
async_subject = AsyncSubject()


async_subject.subscribe(lambda x: print("Testing Async Subject A:
{0}".format(x)))


async_subject.on_next(1)
async_subject.on_next(2)
async_subject.on_completed()


async_subject.subscribe(lambda x: print("Testing Async Subject B:
{0}".format(x)))
```

Here, before complete is called, the last value passed to the subject is 2, and the same is given to the subscribers.

Output

```
E:\pyrx>python testrx.py
Testing Async Subject A: 2
Testing Async Subject B: 2
```

7. RxPY — Concurrency using Scheduler

One important feature of RxPy is concurrency, i.e. to allow the task to execute in parallel. To make that happen, we have two operators subscribe_on() and observe_on() that will work with a scheduler, that will decide the execution of the subscribed task.

Here, is a working example, that shows the need for subscibe_on(), observe_on() and scheduler.

Example

```
import random
import time
import rx
from rx import operators as ops


def adding_delay(value):
    time.sleep(random.randint(5, 20) * 0.1)
    return value


# Task 1
rx.of(1,2,3,4,5).pipe(
    ops.map(lambda a: adding_delay(a))
).subscribe(
    lambda s: print("From Task 1: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 1 complete")
)


# Task 2
rx.range(1, 5).pipe(
    ops.map(lambda a: adding_delay(a))
).subscribe(
    lambda s: print("From Task 2: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 2 complete")


)
```

```


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

In the above example, I have 2 tasks: Task 1 and Task 2. The execution of the task is in sequence. The second task starts only, when the first task is done.

Output

```
E:\pyrx>python testrx.py
From Task 1: 1
From Task 1: 2
From Task 1: 3
From Task 1: 4
From Task 1: 5
Task 1 complete
From Task 2: 1
From Task 2: 2
From Task 2: 3
From Task 2: 4
Task 2 complete
```

RxPy supports many Scheduler, and here, we are going to make use of ThreadPoolScheduler. ThreadPoolScheduler mainly will try to manage with the CPU threads available.

In the example, we have seen earlier, we are going to make use of a multiprocessing module that will give us the cpu_count. The count will be given to the ThreadPoolScheduler that will manage to get the task working in parallel based on the threads available.

Here, is a working example:

```
import multiprocessing
import random
import time
from threading import current_thread
import rx
from rx.scheduler import ThreadPoolScheduler
from rx import operators as ops



# calculate cpu count, using which will  create a ThreadPoolScheduler
thread_count = multiprocessing.cpu_count()
```

```
thread_pool_scheduler = ThreadPoolScheduler(thread_count)

print("Cpu count is : {0}".format(thread_count))

def adding_delay(value):
    time.sleep(random.randint(5, 20) * 0.1)
    return value

# Task 1
rx.of(1,2,3,4,5).pipe(
    ops.map(lambda a: adding_delay(a)),
    ops.subscribe_on(thread_pool_scheduler)
).subscribe(
    lambda s: print("From Task 1: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 1 complete")
)

# Task 2
rx.range(1, 5).pipe(
    ops.map(lambda a: adding_delay(a)),
    ops.subscribe_on(thread_pool_scheduler)
).subscribe(
    lambda s: print("From Task 2: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 2 complete")
)

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

In the above example, I have 2 tasks and the cpu_count is 4. Since, the task is 2 and threads available with us are 4, both the task can start in parallel.
Output:

```
E:\pyrx>python testrx.py

Cpu count is : 4
```

```
Press any key to exit
From Task 1: 1
From Task 2: 1
From Task 1: 2
From Task 2: 2
From Task 2: 3
From Task 1: 3
From Task 2: 4
Task 2 complete
From Task 1: 4
From Task 1: 5
Task 1 complete
```

If you see the output, both the task has started in parallel.

Now, consider a scenario, where the task is more than the CPU count i.e. CPU count is 4 and tasks are 5. In this case, we would need to check if any thread has got free after task completion, so that, it can be assigned to the new task available in the queue.

For this purpose, we can use the observe_on() operator which will observe the scheduler if any threads are free. Here, is a working example using observe_on()

Example

```
import multiprocessing
import random
import time
from threading import current_thread
import rx
from rx.scheduler import ThreadPoolScheduler
from rx import operators as ops

# calculate cpu count, using which will  create a ThreadPoolScheduler
thread_count = multiprocessing.cpu_count()
thread_pool_scheduler = ThreadPoolScheduler(thread_count)



print("Cpu count is : {0}".format(thread_count))
```

```
def adding_delay(value):

    time.sleep(random.randint(5, 20) * 0.1)
    return value

# Task 1
rx.of(1,2,3,4,5).pipe(
    ops.map(lambda a: adding_delay(a)),
    ops.subscribe_on(thread_pool_scheduler)
).subscribe(
    lambda s: print("From Task 1: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 1 complete")
)

# Task 2
rx.range(1, 5).pipe(
    ops.map(lambda a: adding_delay(a)),
    ops.subscribe_on(thread_pool_scheduler)
).subscribe(
    lambda s: print("From Task 2: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 2 complete")
)

#Task 3
rx.range(1, 5).pipe(
    ops.map(lambda a: adding_delay(a)),
    ops.subscribe_on(thread_pool_scheduler)
).subscribe(


    lambda s: print("From Task 3: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 3 complete")
)


#Task 4
rx.range(1, 5).pipe(
    ops.map(lambda a: adding_delay(a)),
    ops.subscribe_on(thread_pool_scheduler)
).subscribe(
    lambda s: print("From Task 4: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 4 complete")
)


#Task 5
rx.range(1, 5).pipe(
    ops.map(lambda a: adding_delay(a)),
    ops.observe_on(thread_pool_scheduler)
).subscribe(
    lambda s: print("From Task 5: {0}".format(s)),
    lambda e: print(e),
    lambda: print("Task 5 complete")
)



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

Output

```
E:\pyrx>python testrx.py
Cpu count is : 4
From Task 4: 1
From Task 4: 2




From Task 1: 1
From Task 2: 1
From Task 3: 1
From Task 1: 2
From Task 3: 2

From Task 4: 3
From Task 3: 3
From Task 2: 2

From Task 1: 3
From Task 4: 4
Task 4 complete
From Task 5: 1
From Task 5: 2
From Task 5: 3
From Task 3: 4
Task 3 complete
From Task 2: 3
Press any key to exit
From Task 5: 4
Task 5 complete
From Task 1: 4
From Task 2: 4
Task 2 complete
From Task 1: 5
Task 1 complete
```

If you see the output, the moment task 4 is complete, the thread is given to the next task i.e., task 5 and the same starts executing.

8. RxPy — Examples

In this chapter, we will discuss the following topics in detail:

- Basic Example showing the working of observable, operators, and subscribing to the observer.
- Difference between observable and subject.
- Understanding cold and hot observables.

Given below is a basic example showing the working of observable, operators, and subscribing to the observer.

Example

test.py

```
import requests
import rx
import json
from rx import operators as ops

def filternames(x):
    if (x["name"].startswith("C")):
        return x["name"]
    else :
        return ""

content = requests.get('https://jsonplaceholder.typicode.com/users')
y = json.loads(content.text)

source = rx.from_(y)

case1 = source.pipe(
    ops.filter(lambda c: filternames(c)),
    ops.map(lambda a:a["name"])
)
case1.subscribe(

    on_next = lambda i: print("Got - {0}".format(i)),
    on_error = lambda e: print("Error : {0}".format(e)),
    on_completed = lambda: print("Job Done!"),
)
```

Here, is a very simple example, wherein, I am getting user data from this URL: https://jsonplaceholder.typicode.com/users.

Filtering the data, to give the names starting with "C", and later using the map to return the names only. Here is the output for the same:

```
E:\pyrx\examples>python test.py
Got - Clementine Bauch
Got - Chelsey Dietrich
Got - Clementina DuBuque
Job Done!
```

Difference between observable and subject

In this example, we will see the difference between an observable and a subject.

```
from rx import of, operators as op
import random

test1 = of(1,2,3,4,5)

sub1 = test1.pipe(
    op.map(lambda a : a+random.random())
)
print("From first subscriber")
subscriber1 = sub1.subscribe(lambda i: print("From sub1 {0}".format(i)))
print("From second subscriber")
subscriber2 = sub1.subscribe(lambda i: print("From sub2 {0}".format(i)))
```

Output

```
E:\pyrx>python testrx.py
From first subscriber
From sub1 1.610450821095726

From sub1 2.9567564032037335
From sub1 3.933217537811936
From sub1 4.82444905626622
From sub1 5.929414892567188
From second subscriber
From sub2 1.8573813517529874
From sub2 2.902433239469483
From sub2 3.2289868093016825
From sub2 4.050413890694411
From sub2 5.226515068012821
```

In the above example, every time you subscribe to the observable, it will give you new values.

Subject Example

```
from rx import of, operators as op
import random
from rx.subject import Subject


subject_test = Subject()
subject_test.subscribe(
 lambda x: print("From sub1 {0}".format(x))
)
subject_test.subscribe(
 lambda x: print("From sub2 {0}".format(x))
)


test1 = of(1,2,3,4,5)
sub1 = test1.pipe(
    op.map(lambda a : a+random.random())
)
subscriber = sub1.subscribe(subject_test)
```

Output

```
E:\pyrx>python testrx.py
From sub1 1.1789422863284509
From sub2 1.1789422863284509




From sub1 2.5525627903260153
From sub2 2.5525627903260153
From sub1 3.4191549324778325
From sub2 3.4191549324778325
From sub1 4.644042420199624
From sub2 4.644042420199624
From sub1 5.079896897489065
From sub2 5.079896897489065
```

If you see the values are shared, between both subscribers using the subject.

Understanding Cold and Hot Observables

An observable is classified as

- Cold Observables
- Hot Observables

The difference in observables will be noticed when multiple subscribers are subscribing.

Cold Observables

Cold observables, are observable that are executed, and renders data each time it is subscribed. When it is subscribed, the observable is executed and the fresh values are given.

The following example gives the understanding of cold observable.

```
from rx import of, operators as op
import random

test1 = of(1,2,3,4,5)

sub1 = test1.pipe(
    op.map(lambda a : a+random.random())
)
print("From first subscriber")
subscriber1 = sub1.subscribe(lambda i: print("From sub1 {0}".format(i)))

print("From second subscriber")
subscriber2 = sub1.subscribe(lambda i: print("From sub2 {0}".format(i)))
```

Output

```
E:\pyrx>python testrx.py
From first subscriber
From sub1 1.610450821095726
From sub1 2.9567564032037335
From sub1 3.933217537811936
From sub1 4.82444905626622
From sub1 5.929414892567188
From second subscriber
From sub2 1.8573813517529874
From sub2 2.902433239469483
From sub2 3.2289868093016825
From sub2 4.050413890694411
From sub2 5.226515068012821
```

In the above example, every time you subscribe to the observable, it will execute the observable and emit values. The values can also differ from subscriber to subscriber as shown in the example above.

Hot Observables

In the case of hot observable, they will emit the values when they are ready and will not always wait for a subscription. When the values are emitted, all the subscribers will get the same value.

You can make use of hot observable when you want values to emitted when the observable is ready, or you want to share the same values to all your subscribers.

An example of hot observable is Subject and connectable operators.

```
from rx import of, operators as op
import random
from rx.subject import Subject

subject_test = Subject()
subject_test.subscribe(
 lambda x: print("From sub1 {0}".format(x))

)
subject_test.subscribe(

 lambda x: print("From sub2 {0}".format(x))

)


test1 = of(1,2,3,4,5)
sub1 = test1.pipe(
    op.map(lambda a : a+random.random())
)
subscriber = sub1.subscribe(subject_test)
```

Output

```
E:\pyrx>python testrx.py
From sub1 1.1789422863284509
From sub2 1.1789422863284509
From sub1 2.5525627903260153
From sub2 2.5525627903260153
From sub1 3.4191549324778325
From sub2 3.4191549324778325
From sub1 4.644042420199624
From sub2 4.644042420199624
From sub1 5.079896897489065
From sub2 5.079896897489065
```

If you see, the same value is shared between the subscribers. You can achieve the same using publish () connectable observable operator.