# Chapter 36
# RxPy Observables, Observers and Subjects


## 36.1 Introduction

In this chapter we will discuss Observables, Observers and Subjects. We also consider how observers may or may not run concurrently.

In the remainder of this chapter we look at RxPy version 3 which is a major update from RxPy version 1 (you will therefore need to be careful if you are looking on the web for examples as some aspects have changed; most notably the way in which operators are chained).

## 36.2 Observables in RxPy

An `Observable` is a Python class that publishes data so that it can be processed by one or more Observers (potentially running in separate threads).

An `Observable` can be created to publish data from static data or from dynamic sources. Observables can be chained tougher to control how and when data is published, to transform data before it is published and to restrict what data is actually published.

For example, to create an `Observable` from a list of values we can use the `rx.from_list()` function. This function (also known as an RxPy operator) is used to create the new Observable object:

```
import rx
Observable = rx.from_list([2, 3, 5, 7])
```

© Springer Nature Switzerland AG 2019

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_36

## 36.3 Observers in RxPy

We can add an Observer to an `Observable` using the `subcribe()` method. This method can be supplied with a lambda function, a named function or an object whose class implements the *Observer protocol*.

For example, the simplest way to create an Observer is to use a *lambda* function:

```
# Subscribe a lambda function
observable.subscribe(lambda value: print('Lambda Received',
value))
```

When the `Observable` publishes data the `lambda` function will be invoked. Each data item published will be supplied independently to the function. The output from the above subscription for the previous `Observable` is:

```
Lambda Received 2
Lambda Received 3
Lambda Received 5
Lambda Received 7
```

We can also have used a standard or named function as an Observer:

```
def prime_number_reporter(value):
    print('Function Received', value)

# Subscribe a named function
observable.subscribe(prime_number_reporter)
```

Note that it is only the name of the function that is used with the `subscribe()` method (as this effectively passes a reference to the function into the method).

If we now run this code using the previous `Observable` we get:

```
Function Received 2
Function Received 3
Function Received 5
Function Received 7
```

In actual fact the `subscribe()` method takes four optional parameters. Thes are:

- `on_next` Action to invoke for each data item generated by the Observable.
- `on_error` Action to invoke upon exceptional termination of the Observable sequence.
- `on_completed` Action to invoke upon graceful termination of the Observable sequence.
- `Observer` The object that is to receive notifications. You may subscribe using an Observer or callbacks, not both.

Each of the above can be used as positional parameters or as keyword arguments, for example:

```
# Use lambdas to set up all three functions
observable.subscribe(
    on_next = lambda value: print('Received on_next', value),
    on_error = lambda exp: print('Error Occurred', exp),
    on_completed = lambda: print('Received completed
notification')
)
```

The above code defines three lambda functions that will be called depending upon whether data is supplied by the `Observable`, if an error occurs or when the data stream is terminated. The output from this is:

```
Received on_next 2
Received on_next 3
Received on_next 5
Received on_next 7
Received completed notification
```

Note that the `on_error` function is not run as no error was generated in this example.

The final optional parameter to the `subscribe()` method is an *Observer* object. An Observer object can implement the Observer protocol which has the following methods `on_next()`, `on_completed()` and `on_error()`, for example:

```
class PrimeNumberObserver:

    def on_next(self, value):
        print('Object Received', value)

    def on_completed(self):
        print('Data Stream Completed')

    def on_error(self, error):
        print('Error Occurred', error)
```

Instances of this class can now be used as an Observer via the `subscribe()` method:

```
# Subscribe an Observer object
observable.subscribe(PrimeNumberObserver())
```

The output from this example using the previous Observable is:

```
Object Received 2
Object Received 3
Object Received 5
Object Received 7
Data Stream Completed
```

Note that the `on_completed()` method is also called; however the `on_errror()` method is not called as there were no exceptions generated.

The `Observer` class must ensure that the methods implemented adhere to the Observer protocol (i.e. That the signatures of the `on_next()`, `on_completed()` and `on_error()` methods are correct).

## 36.4 Multiple Subscribers/Observers

An Observable can have multiple Observers subscribed to it. In this case each of the Observers is sent all of the data published by the Observable. Multiple Observers can be registered with an Observable by calling the subscribe method multiple times. For example, the following program has four subscribers as well as `on_error` and `on_completed` function registered:

```
# Create an observable using data in a list
observable = rx.from_list([2, 3, 5, 7])

class PrimeNumberObserver:
    """ An Observer class """

    def on_next(self, value):
        print('Object Received', value)

    def on_completed(self):
        print('Data Stream Completed')

    def on_error(self, error):
        print('Error Occurred', error)


def prime_number_reporter(value):
    print('Function Received', value)


print('Set up Observers / Subscribers')

# Subscribe a lambda function
observable.subscribe(lambda value: print('Lambda Received',
value))
# Subscribe a named function
observable.subscribe(prime_number_reporter)
# Subscribe an Observer object
observable.subscribe(PrimeNumberObserver())
# Use lambdas to set up all three functions
observable.subscribe(
    on_next=lambda value: print('Received on_next', value),
    on_error=lambda exp: print('Error Occurred', exp),
    on_completed=lambda: print('Received completed
notification')
)
```

The output from this program is:

```
Create the Observable object
Set up Observers / Subscribers
Lambda Received 2
Lambda Received 3
Lambda Received 5
Lambda Received 7
Function Received 2
Function Received 3
Function Received 5
Function Received 7
Object Received 2
Object Received 3
Object Received 5
Object Received 7
Data Stream Completed
Received on_next 2
Received on_next 3
Received on_next 5
Received on_next 7
Received completed notification
```

Note how each of the subscribers is sent all of the data before the next subscriber is sent their data (this is the default single threaded RxPy behaviour).

## 36.5 Subjects in RxPy

A *subject* is both an Observer and an Observable. This allows a subject to receive an item of data and then to republish that data or data derived from it.

For example, imagine a subject that receives stock market price data published by an external (to the organisation receiving the data) source. This subject might add a timestamp and source location to the data before republishing it to other internal Observers.

However, there is a subtle difference that should be noted between a *Subject* and a plain *Observable*. A subscription to an Observable will cause an independent execution of the Observable when data is published. Notice how in the previous section all the messages were sent to a specific Observer before the next Observer was sent any data at all.

However, a Subject shares the publication action with all of the subscribers and they will therefore all receive the same data item in a chain before the next data item.

In the class hierarchy the `Subject` class is a direct subclass of the `Observer` class.

The following example creates a `Subject` that enriches the data it receives by adding a *timestamp* to each data item. It then republishes the data item to any Observers that have subscribed to it.

```
import rx
from rx.subjects import Subject
from datetime import datetime

source = rx.from_list([2, 3, 5, 7])

class TimeStampSubject(Subject):

    def on_next(self, value):
        print('Subject Received', value)
        super().on_next((value, datetime.now()))

    def on_completed(self):
        print('Data Stream Completed')
        super().on_completed()

    def on_error(self, error):
        print('In Subject - Error Occurred', error)
        super().on_error(error)


def prime_number_reporter(value):
    print('Function Received', value)


print('Set up')

# Create the Subject
subject = TimeStampSubject()

# Set up multiple subscribers for the subject
subject.subscribe(prime_number_reporter)
subject.subscribe(lambda value: print('Lambda Received',
value))
subject.subscribe(
    on_next = lambda value: print('Received on_next', value),
    on_error = lambda exp: print('Error Occurred', exp),
    on_completed = lambda: print('Received completed
notification')
)

# Subscribe the Subject to the Observable source
source.subscribe(subject)

print('Done')
```

Note that in the above program the Observers are added to the `Subject` before the Subject is added to the source `Observable`. This ensures that the Observers are subscribed before the `Subject` starts to receive data published by the

`Observable`. If the `Subject` was subscribed to the `Observable` before the Observers were subscribed to the `Subject`, then all the data could have been published before the Observers were registered with the Subject.

The output from this program is:

```
Set up
Subject Received 2
Function Received (2, datetime.datetime(2019, 5, 21, 17, 0, 2,
196372))
Lambda Received (2, datetime.datetime(2019, 5, 21, 17, 0, 2,
196372))
Received on_next (2, datetime.datetime(2019, 5, 21, 17, 0, 2,
196372))
Subject Received 3
Function Received (3, datetime.datetime(2019, 5, 21, 17, 0, 2,
196439))
Lambda Received (3, datetime.datetime(2019, 5, 21, 17, 0, 2,
196439))
Received on_next (3, datetime.datetime(2019, 5, 21, 17, 0, 2,
196439))
Subject Received 5
Function Received (5, datetime.datetime(2019, 5, 21, 17, 0, 2,
196494))
Lambda Received (5, datetime.datetime(2019, 5, 21, 17, 0, 2,
196494))
Received on_next (5, datetime.datetime(2019, 5, 21, 17, 0, 2,
196494))
Subject Received 7
Function Received (7, datetime.datetime(2019, 5, 21, 17, 0, 2,
196548))
Lambda Received (7, datetime.datetime(2019, 5, 21, 17, 0, 2,
196548))
Received on_next (7, datetime.datetime(2019, 5, 21, 17, 0, 2,
196548))
Data Stream Completed
Received completed notification
Done
```

As can be seen from this output the numbers 2, 3, 5 and 7 are received by all of the Observers once the `Subject` has added the timestamp.

## 36.6 Observer Concurrency

By default RxPy uses a single threaded model; that is Observables and Observers execute in the same thread of execution. However, this is only the default as it is the simplest approach.

It is possible to indicate that when a Observer subscribes to an Observable that it should run in a separate thread using the `scheduler` keyword parameter on the

subscribe() method. This keyword is given an appropriate scheduler such as the rx.concurrency.NewThreadScheduler. This scheduler will ensure that the Observer runs in a separate thread.

To see the difference look at the following two programs. The main difference between the programs is the use of specific schedulers:

```
import rx

Observable = rx.from_list([2, 3, 5])

observable.subscribe(lambda v: print('Lambda1 Received', v))
observable.subscribe(lambda v: print('Lambda2 Received', v))
observable.subscribe(lambda v: print('Lambda3 Received', v))
```

The output from this first version is given below:

```
Lambda1 Received 2
Lambda1 Received 3
Lambda1 Received 5
Lambda2 Received 2
Lambda2 Received 3
Lambda2 Received 5
Lambda3 Received 2
Lambda3 Received 3
Lambda3 Received 5
```

The subscribe() method takes an optional keyword parameter called scheduler that allows a scheduler object to be provided.

Now if we specify a few different schedulers we will see that the effect is to run the Observers concurrently with the resulting output being interwoven:

```
import rx
from rx.concurrency import NewThreadScheduler,
ThreadPoolScheduler, ImmediateScheduler

Observable = rx.from_list([2, 3, 5])

observable.subscribe(lambda v: print('Lambda1 Received', v),
scheduler=ThreadPoolScheduler(3))
observable.subscribe(lambda v: print('Lambda2 Received', v),
scheduler=ImmediateScheduler())
observable.subscribe(lambda v: print('Lambda3 Received', v),
scheduler=NewThreadScheduler())

# As the Observable runs in a separate thread need
# ensure that the main thread does not terminate
input('Press enter to finish')
```

Note that we have to ensure that the main thread running the program does not terminate (as all the Observables are now running in their own threads) by waiting for user input. The output from this version is:

```
Lambda2 Received 2
Lambda1 Received 2
Lambda2 Received 3
Lambda2 Received 5
Lambda1 Received 3
Lambda1 Received 5
Press enter to finish
Lambda3 Received 2
Lambda3 Received 3
Lambda3 Received 5
```

By default the `scheduler` keyword on the `subscribe()` method defaults to `None` indicating that the current thread will be used for the subscription to the Observable.

### *36.6.1 Available Schedulers*

To support different scheduling strategies the RxPy library provides two modules that supply different schedulers; the `rx.concurrency` and `rx.currency.mainloopscheduler`. The modules contain a variety of schedulers including those listed below.

The following schedulers are available in the `rx.concurrency` module:

- `ImmediateScheduler` This schedules an action for immediate execution.
- `CurrentThreadScheduler` This schedules activity for the current thread.
- `TimeoutScheduler` This scheduler works via a timed callback.
- `NewThreadScheduler` creates a scheduler for each unit of work on a separate thread.
- `ThreadPoolScheduler`. This is a scheduler that utilises a thread pool to execute work. This scheduler can act as a way of throttling the amount of work carried out concurrently.

The `rx.concurrency.mainloopschduler` module also defines the following schedulers:

- `IOLoopScheduler` A scheduler that schedules work via the Tornado I/O main event loop.
- `PyGameScheduler` A scheduler that schedules works for PyGame.
- `WxScheduler` A scheduler for a wxPython event loop.

## 36.7 Online Resources

See the following online resources for information on RxPy:

- https://github.com/ReactiveX/RxPY The RxPy Git hub repository.
- https://rxpy.readthedocs.io/en/latest/ Documentation for the RxPy library.
- https://rxpy.readthedocs.io/en/latest/operators.html Lists of the available RxPy operators.

## 36.8 Exercises

Given the following set of tuples representing Stock/Equity prices:

```
stocks = (('APPL', 12.45), ('IBM', 15.55), ('MSFT', 5.66),
('APPL', 13.33))
```

Write a program that will create an Observable based on the stocks data.

Next subscribe three different observers to the Observable. The first should print out the stock price, the second should print out the name of the stock and the third should print out the entire tuple.