Chapter 35
Reactive Programming Introduction

35.1 Introduction

In this chapter we will introduce the concept of Reactive Programming. Reactive programming is a way of write programs that allow the system to *reactive to* data being published to it. We will look at the RxPy library which provides a Python implementation of the ReactiveX approach to Reactive Programming.

35.2 What Is a Reactive Application?

A Reactive Application is one that must react to data; typically either to the presence of new data, or to changes in existing data. The *Reactive Manifesto* presents the key characteristics of Reactive Systems as:

- **Responsive**. This means that such systems respond in a timely manner. Here of course timely will differ depending upon the application and domain; in one situation a second may be timely in another it may be far too slow.
- **Resilient**. Such systems stay responsive in the face of failure. The systems must therefore be designed to handle failure gracefully and continue to work appropriately following the failure.
- **Elastic**. As the workload grows the system should continue to be responsive.
- **Message Driven**. Information is exchanged between elements of a reactive system using messages. This ensures loose coupling, isolation and location transparency between these components.

As an example, consider an application that lists a set of *Equity Stock Trade* values based on the latest market stick price data. This application might present the current value of each trade within a table. When new market stock price data is

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_35

published, then the application must update the value of the trade within the table. Such an application can be described as being reactive.

Reactive Programming is a programming style (typically supported by libraries) that allows code to be written that follow the ideas of reactive systems. Of course just because part of an application uses a Reactive Programming library does not make the whole application reactive; indeed it may only be necessary for part of an application to exhibit reactive behaviour.

35.3 The ReactiveX Project

ReactiveX is the best known implementation of the Reactive Programming paradigm.

ReactiveX is based on the *Observer-Observable* design pattern. However it is an extension to this design pattern as it extends the pattern such that the approach supports sequences of data and/or events and adds operators that allow developers to compose sequences together declaratively while abstracting away concerns associated with low-level threads, synchronisation, concurrent data structures and non-blocking I/O.

The ReactiveX project has implementations for many languages including RxJava, RxScala and RxPy; this last is the version we are looking at as it is for the Python language.

RxPy is described as:

> A library for composing asynchronous and event-based programs using Observable collections and query operator functions in Python

35.4 The Observer Pattern

The Observer Pattern is one of the *Gang of Four* set of Design Patterns. The Gang of Four Patterns (as originally described in Gamma et al. 1995) are so called because this book on design patterns was written by four very famous authors namely; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

The Observer Pattern provides a way of ensuring that a set of objects is notified whenever the state of another object changes. It has been widely used in a number of languages (such as Smalltalk and Java) and can also be used with Python.

The intent of the Observer Pattern is to manage a one to many relationship between an object and those objects interested in the state, and in particular state changes, of that object. Thus when the objects' state changes, the interested (dependent) objects are notified of that change and can take whatever action is appropriate.

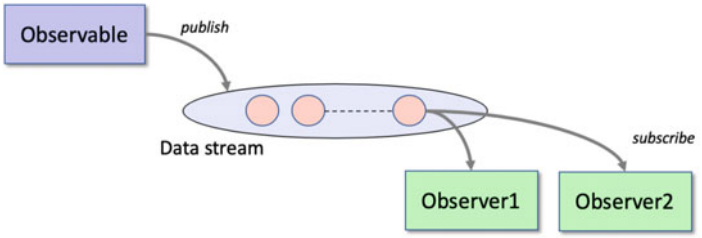
There are two key roles within the Observer Pattern, these are the Observable and the Observer roles.

- **Observable**. This is the object that is responsible for notifying other objects that a change in its state has occurred
- **Observer**. An Observer is an object that will be notified of the change in state of the Observable and can take appropriate action (such as triggering a change in their own state or performing some action).

In addition the state is typically represented explicitly:

- **State**. This role may be played by an object that is used to share information about the change in state that has occurred within the Observable. This might be as simple as a String indicating the new state of the Observable or it might be a data oriented object that provides more detailed information.

These roles are illustrated in the following figure.



In the above figure, the Observable object publishes data to a Data Stream. The data in the Data Stream is then sent to each of the Observers registered with the Observable. In this way data is broadcast to all Observers of an Observable.

It is common for an Observable to only publish data once there is an Observer available to process that data. The process of registering with an Observable is referred to as subscribing. Thus an Observable will have zero or more subscribers (Observers).

If the Observable publishes data at a faster rate than can be processed by the Observer then the data is queued via the Data Stream. This allows the Observer to process the data received one at a time at its own pace; without any concern for data loss (as long as sufficient memory is available for the data stream).

35.5 Hot and Cold Observables

Another concept that it is useful to understand is that of *Hot* and *Cold* Observables.

- Cold Observables are *lazy* Observables. That is, a Cold Observable will only publish data if at least one Observer is subscribed to it.

- Hot Observables, by contrast, publish data whether there is an Observer subscribed or not.

35.5.1 Cold Observables

A Cold Observable will not publish any data unless there is at least one Observer subscribed to process that data. In addition a cold Observable only provides data to an Observer when that Observer is ready to process the data; this is because the Observable-Observer relationship is more of a *pull* relationship. For example, given an Observable that will generate a set of values based on a range, then that Observable will generate each result lazily when requested by an Observer.

If the Observer takes some time to process the data emitted by the Observable, then the Observable will wait until the Observer is ready to process the data before emitting another value.

35.5.2 Hot Observables

Hot Observables by contrast publish data whether there is an Observer subscribed or not. When an Observer registers with the Observable, it will start to receive data at that point, as and when the Observable publishes new data. If the Observable has already published previous data items, then these will have been lost and the Observer will not receive that data.

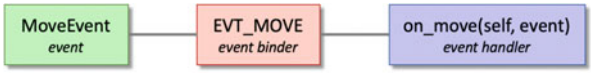
The most common situation in which a Hot Observable is created is when the source producer represents data that may be irrelevant if not processed immediately or may be superseded by subsequent data. For example, data published by a Stock Market Price data feed would fall into this category. When an Observable wraps around this data feed it can publish that data whether or not an Observer is subscribed.

35.5.3 Implications of Hot and Cold Observables

It is important to know whether you have a hot or cold Observable because this can impact on what you can assume about the data supplied to the Observers and thus how you need to design your application. If it is important that no data is lost then care is needed to ensure that the subscribers are in place before a Hot Observable starts to publish data (where as this is not a concern for a cold Observable).

35.6 Differences Between Event Driven Programming and Reactive Programming

In Event Driven programming, an event is generated in response too something happening; the event then represents this with any associated data. For example, if the user clicks the mouse then an associated `MouseClickEvent` might be generated. This object will usually hold information about the x and y coordinates of the mouse along with which button was clicked etc. It is then possible to associate some behaviour (such as a function or a method) with this event so that if the event occurs, then the associated operation is invoked and the event object is provided as a parameter. This is certainly the approach used in the `wxPython` library presented earlier in this book:



From the above diagram, when a MoveEvent is generated the `on_move()` method is called and the event is passed into the method.

In the Reactive Programming approach, an Observer is associated with an Observable. Any data generated by the Observable will be received and handled by the Observer. This is true whatever that data is, as the Observer is a handler of data generated by the Observable rather than a handler of a specific type of data (as with the Event driven approach).

Both approaches could be used in many situations. For example, we could have a scenario in which some data is to be processed whenever a stock price changes.

This could be implemented using a `StockPriceChangeEvent` associated with a `StockPriceEventHandler`. It could also be implemented via `StockPriceChangeObserverable` and a `StockPriceChangeObserver`. In either case one element handles the data generated by another element. However, the RxPy library simplifies this process and allows the Observer to run in the same thread as, or a separate thread from, the Observable with just a small change to the code.

35.7 Advantages of Reactive Programming

There are several advantages to the use of a Reactive Programming library these include:

- **It avoids multiple callback methods**. The problems associated with the use of callbacks are sometimes referred to as *callback hell*. This can occur when there are multiple callbacks, all defined to run in response to some data being generated or some operation completing. It can be hard to understand, maintain and debug such systems.

- **Simpler asynchronous, multi threaded execution**. The approach adopted by RxPy makes it very easy to execute operations/ behaviour within a multi threaded environment with independent asynchronous functions.
- **Available Operators**. The RxPy library comes pre built with numerous operators that make processing the data produced by an Observable much easier.
- **Data Composition**. It is straight forward to compose new data streams (Observables) from data supplied by two or more other Observables for asynchronous processing.

35.8 Disadvantages of Reactive Programming

Its easy to over complicate things when you start to chain operators together. If you use too many operators, or too complex a set of functions with the operators, it can become hard to understand what is going on.

Many developers think that Reactive programming is inherently multi-threaded; this is not necessarily the case; in fact RxPy (the library explored in the next two chapters) is single threaded by default. If an application needs the behaviour to execute asynchronously then it is necessary to explicitly indicate this.

Another issue for some Reactive programming frameworks is that it can become memory intensive to store streams of data so that Observers can processes that data when they are ready.

35.9 The RxPy Reactive Programming Framework

The RxPy library is a part of the larger ReactiveX project and provides an implementation of ReactiveX for Python. It is built on the concepts of Observables, Observers, Subjects and operators. In this book we use RxPy version 3.

In the next chapter we will discuss Observables, Observers, Subjects and subscriptions using the RxPy library. The following chapter will explore various RxPy operators.

35.10 Online Resources

See the following online resources for information on reactive programming:

- https://www.reactivemanifesto.org/ The Reactive Manifesto.
- http://reactivex.io/ The ReactiveX home page.
- https://en.wikipedia.org/wiki/Design_Patterns Wikipedia page on Design Patterns book.

35.11 Reference

For more information on the Observer Observable design pattern see the “Patterns” book by the Gang of Four

- E. Gamma, R. Helm, R. Johnson, J. Vlissades, Design patterns: elements of reusable object-oriented software, Addison-Wesley (1995).