Chapter 37
RxPy Operators

37.1 Introduction

In this chapter we will look at the types of operators provided by RxPy that can be applied to the data emitted by an Observable.

37.2 Reactive Programming Operators

Behind the interaction between an Observable and an Observer is a data stream. That is the Observable supplies a data stream to an Observer that consumes/processes that stream. It is possible to apply an operator to this data stream that can be used to to filter, transform and generally refine how and when the data is supplied to the Observer.

The operators are mostly defined in the `rx.operators` module, for example `rx.operators.average()`. However it is common to use an alias for this such that the `operators` module is called `op`, such as

```
from rx import operators as op
```

This allows for a short hand form to be used when referencing an operator, such as `op.average()`.

Many of the RxPy operators execute a function which is applied to each of the data items produced by an Observable. Others can be used to create an initial Observable (indeed you have already seen these operators in the form of the `from_list()` operator). Another set of operators can be used to generate a result based on data produced by the Observable (such as the `sum()` operator).

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_37

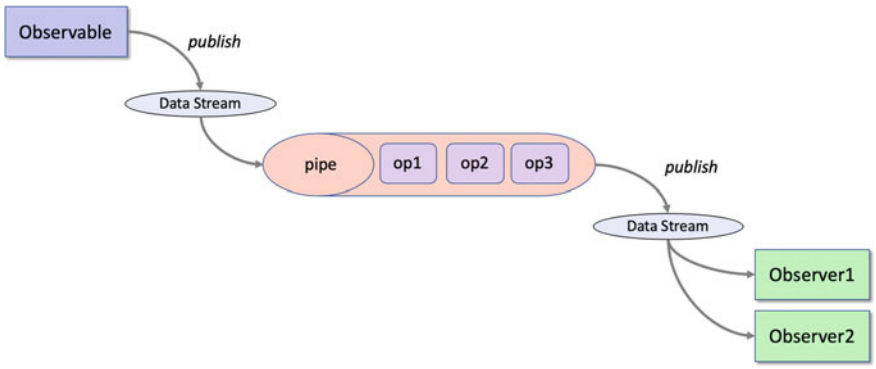
In fact RxPy provides a wide variety of operators and these operators can be categorised as follows:

- Creational,
- Transformational,
- Combinatorial,
- Filters,
- Error handlers,
- Conditional and Boolean operators,
- Mathematical,
- Connectable.

Examples of some of these categories are presented in the rest of this section.

37.3 Piping Operators

To apply an operator other than a creational operator to an Observable it is necessary to create a pipe. A Pipe is essentially a series of one or more operations that can be applied to the data stream generated by the Observable. The result of applying the pipe is that a new data stream is generated that represents the results produced following the application of each operator in turn. This is illustrated below:



To create a pipe the `Observable.pipe()` method is used. This method takes a comma delimited list of one or more operators and returns a data stream. Observers can then subscribe to the pipe's data stream. This can be seen in the examples given in the rest of this chapter for transformations, filters, mathematical operators etc.

37.4 Creational Operators

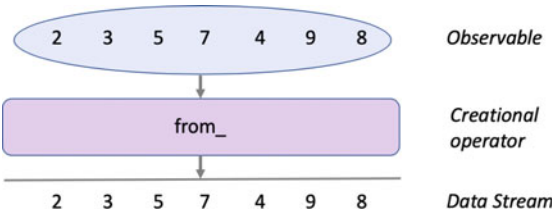
You have already seen an example of a creational operator in the examples presented earlier in this chapter. This is because the `rx.from_list()` operator is an example of a creational operator. It is used to create a new Observable based on data held in a *list like* structure.

A more generic version of `from_list()` is the `from_()` operator. This operator takes an iterable and generates an Observable based on the data provided by the iterable. Any object that implements the iterable protocol can be used including user defined types. There is also an operator `from_iterable()`. All three operators do the same thing and you can choose which to use based on which provides the most semantic meaning in your context.

All three of the following statements have the same effect:

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

This is illustrated pictorially below:



Another creational operator is the `rx.range()` operator. This operator generates an observable for a range of integer numbers. The range can be specified with our without a starting value and with or within an increment. However the maximum value in the range must always be provided, for example:

```
obs1 = rx.range(10)
obs2 = rx.range(0, 10)
obs3 = rx.range(0, 10, 1)
```

37.5 Transformational Operators

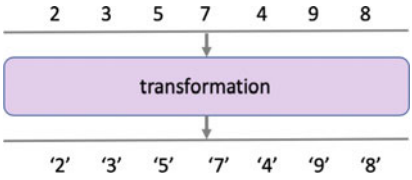
There are several transformational operators defined in the `rx.operators` module including `rx.operators.map()` and `rx.operators.flat_map()`.

The `rx.operators.map()` operator applies a function to each data item generated by an Observable.

The `rx.operators.flat_map()` operator also applies a function to each data item but then applies a *flatten* operation to the result. For example, if the result is a list of lists then `flat_map` will flatten this into a single list. In this section we will focus on the `rx.operators.map()` operator.

The `rx.operators.map()` operator allows a function to be applied to all data items generated by an Observable. The result of this function is then returned as the result of the `map()` operators Observable. The function is typically used to perform some form of transformation to the data supplied to it. This could be adding one to all integer values, converting the format of the data from XML to JSON, enriching the data with additional information such as the time the data was acquired and who the data was supplied by etc.

In the example given below we are transforming the set of integer values supplied by the original `Observable` into strings. In the diagram these strings include quotes around them to highlight they are in fact a string:



This is typical of the use of a transformation operator; that is to change the data from one format to another or to add information to the data.

The code used to implement this scenario is given below. Note the use of the `pipe()` method to apply the operator to the data stream generated by the Observable:

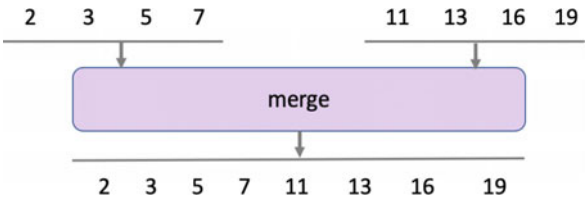
```
# Apply a transformation to a data source to convert
# integers into strings
import rx
from rx import operators as op

# Set up a source with a map function
source = rx.from_list([2, 3, 5, 7]).pipe(
    op.map(lambda value: "'" + str(value) + "'")
)

# Subscribe a lambda function
source.subscribe(lambda value: print('Lambda Received',
                                     value,
                                     ' is a string ',
                                     isinstance(value, str)))
```

The output from this program is:

```
Lambda Received '2'  is a string  True
Lambda Received '3'  is a string  True
Lambda Received '5'  is a string  True
Lambda Received '7'  is a string  True
```

37.6 Combinatorial Operators

Combinatorial operators combine together multiple data items in some way. One example of a combinatorial operator is the `rx.merge()` operator. This operator merges the data produced by two Observables into a single Observable data stream. For example:



In the above diagram two Observables are represented by the sequence 2, 3, 5, 7 and the sequence 11, 13, 16, 19. These Observables are supplied to the merge operator that generates a single Observable that will supply data generated from both of the original Observables. This is an example of an operator that does not take a function but instead takes two Observables.

The code representing the above scenario is given below:

```
# An example illustrating how to merge two data sources
import rx

# Set up two sources
source1 = rx.from_list([2, 3, 5, 7])
source2 = rx.from_list([10, 11, 12])

# Merge two sources into one
rx.merge(source1, source2)\
    .subscribe(lambda v: print(v, end=','))
```

Notice that in this case we have subscribed directly to the Observable returned by the `merge()` operator and have not stored this in an intermediate variable (this was a design decision and either approach is acceptable).

The output from this program is presented below:

```
2,3,5,7,10,11,12,
```

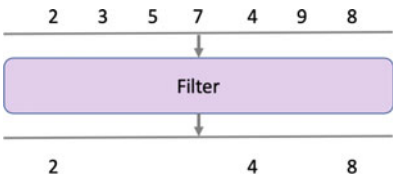
Notice from the output the way in which the data held in the original Observables is intertwined in the output of the Observable generated by the `merge()` operator.

37.7 Filtering Operators

There are several operators in this category including `rx.operators.filter()`, `rx.operators.first()`, `rx.operators.last()` and `rx.operators.distinct()`.

The `filter()` operator only allows those data items to pass through that pass some test expression defined by the function passed into the filter. This function must return `True` or `False`. Any data item that causes the function to return `True` is allowed to pass through the filter.

For example, let us assume that the function passed into `filter()` is designed to only allow even numbers through. If the data stream contains the numbers 2, 3, 5, 7, 4, 9 and 8 then the `filter()` will only emit the numbers 2, 4 and 8. This is illustrated below:



The following code implements the above scenario:

```
# Filter source for even numbers
import rx
from rx import operators as op

# Set up a source with a filter
source = rx.from_list([2, 3, 5, 7, 4, 9, 8]).pipe(
    op.filter(lambda value: value % 2 == 0)
)

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

In the above code the `rx.operators.filter()` operator takes a lambda function that will verify if the current value is even or not (note this could have been a named function or a method on an object etc.). It is applied to the data stream generated by the Observable using the `pipe()` method. The output generated by this example is:

```
Lambda Received 2
Lambda Received 4
Lambda Received 8
```

The `first()` and `last()` operators emit only the first and last data item published by the Observable.

The `distinct()` operator suppresses duplicate items being published by the Observable. For example, in the following list used as the data for the Observable, the numbers 2 and 3 are duplicated:

```
# Use distinct to suppress duplicates
source = rx.from_list([2, 3, 5, 2, 4, 3, 2]).pipe(
    op.distinct()
)

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

However, when the output is generated by the program all duplicates have been suppressed:

```
Received 2
Received 3
Received 5
Received 4
```

37.8 Mathematical Operators

Mathematical and aggregate operators perform calculations on the data stream provided by an Observable. For example, the `rx.operators.average()` operator can be used to calculate the average of a set of numbers published by an Observable. Similarly `rx.operators.max()` can select the maximum value, `rx.operators.min()` the minimum value and `rx.operators.sum()` will total all the numbers published etc.

An example using the `rx.operators.sum()` operator is given blow:

```
# Example of summing all the values in a data stream
import rx
from rx import operators as op

# Set up a source and apply sum
rx.from_list([2, 3, 5, 7]).pipe(
    op.sum()
).subscribe(lambda v: print(v))
```

The output from the `rx.operators.sum()` operator is the total of the data items published by the Observable (in this case the total of 2, 3, 5 and 7). The Observer function that is subscribed to the `rx.operators.sum()` operators Observable will print out this value:

```
17
```

However, in some cases it may be useful to be notified of the intermediate running total as well as the final value so that other operators down the chain can react to these subtotals. This can be achieved using the `rx.operators.scan()` operator. The `rx.operators.scan()` operator is actually a transformational operator but can be used in this case to provide a mathematical operation. The `scan()` operator applies a function to each data item published by an Observable and generates its own data item for each value received. Each generated value is passed to the next invocation of the `scan()` function as well as being published to the `scan()` operators Observable data stream. The running total can thus be generated from the previous sub total and the new value obtained. This is shown below:

```
import rx
from rx import operators as op
# Rolling or incremental sum
rx.from_([2, 3, 5, 7]).pipe(
          op.scan(lambda subtotal, i: subtotal+i)
).subscribe(lambda v: print(v))
```

The output from this example is:

```
2
5
10
17
```

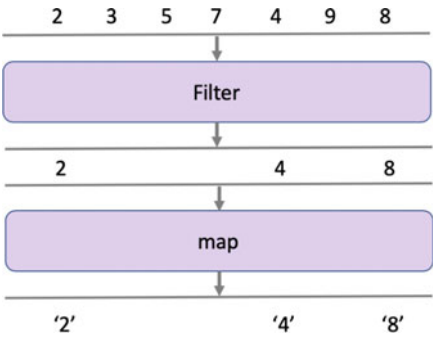
This means that each subtotal is published as well as the final total.

37.9 Chaining Operators

An interesting aspect of the RxPy approach to data stream processing is that it is possible to apply multiple operators to the data stream produced by an Observable.

The operators discussed earlier actually return another Observable. This new Observable can supply its own data stream based on the original data stream and the result of applying the operator. This allows another operator to be applied in sequence to the data produced by the new Observable. This allows the operators to be chained together to provide sophisticated processing of the data published by the original Observable.

For example, we might first start off by filtering the output from an Observable such that only certain data items are published. We might then apply a transformation in the form of a `map()` operator to that data, as shown below:



Note the the order in which we have applied the operators; we first filter out data that is not of interest and then apply the transformation. This is more efficient than apply the operators the other way around as in the above example we do not need to transform the odd values. It is therefore common to try and push the filter operators as high up the chain as possible.

The code used to generate the chained set of operators is given below. In this case we have used lambda functions to define the `filter()` function and the `map()` function. The operators are applied to the Observable obtained from the list supplied. The data stream generated by the Observable is processed by each of the operators defined in the pipe. As there are now two operators the pipe contains both operators and acts a pipe down which the data flows.

The list used as the initial source of the Observables data contains a sequence of event and odd numbers. The `filter()` function selects only even numbers and the `map()` function transforms the integer values into strings. We then subscribe an Observer function to the Observable produced by the transformational `map()` operator.

```
# Example of chaining operators together
import rx
from rx import operators as op

# Set up a source with a filter
source = rx.from_list([2, 3, 5, 7, 4, 9, 8])
pipe = source.pipe(
    op.filter(lambda value: value % 2 == 0),
    op.map(lambda value: "'" + str(value) + "'")
)

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

The output from this application is given below:

```
Received '2'
Received '4'
Received '8'
```

This makes it clear that only the three even numbers (2, 4 and 8) are allowed through to the `map()` function.

37.10 Online Resources

See the following online resources for information on RxPy:

- 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.

37.11 Exercises

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

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

Provide solutions to the following:

- Select all the 'APPL' stocks
- Select all stocks with a price over 15.00
- Find the average price of all 'APPL' stocks.

Now use the second set of tuples and merge them with the first set of stock prices:

```
stocks2 = (('GOOG', 8.95), ('APPL', 7.65), ('APPL', 12.45),
('MSFT', 5.66), ('GOOG', 7.56), ('IBM', 12.76))
```

Convert each tuple into a list and calculate how much 25 shares in that stock would be, print this out as the result).

- Find the highest value stock.
- Find the lowest value stock.
- Only publish unique data times (I.e. Suppress duplicates).