Reactive Python for Data

Part IV - The Observable

4.1A - Creating an `Observable`

An `Observable` pushes items. It can push a finite or infinite series of items over time. To create an `Observable` that pushes 5 text strings, you can declare it like this:

```
from rx import Observable

letters = Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"])
```

We create an `Observable` using the `from_()` function, and pass it a list of five strings. It will take the list and **emit** (or push) each item from it. The `Observable.from_()` will work with any iterable.

However, running this does nothing more than save an `Observable` to a variable called `letters`. For the items to actually get pushed, we need a `Subscriber`.

4.1B - Subscribing to an `Observable`

To receive emissions from an `Observable`, we need to create a `Subscriber` by implementing an `Observer`. An `Observer` implements three functions `on_next()` which receives an emission, `on_completed()` which is called when there are no more items, and `on_error()` which receives an error in the event one occurs.

Then we can pass an implementation of this `Observer` to the Observable's `subscribe()` function. It will then fire the emissions to our `Subscriber`.

```
from rx import Observable, Observer

letters = Observable.from_(["Alpha","Beta","Gamma","Delta
","Epsilon"])


class MySubscriber(Observer):
    def on_next(self, value):
        print(value)

    def on_completed(self):
        print("Completed!")

    def on_error(self, error):
        print("Error occured: {0}".format(error))


```

```
letters.subscribe(MySubscriber())
```

OUTPUT:

```
Received: Alpha
Received: Beta
Received: Gamma
Received: Delta
Received: Epsilon
Completed!
```

Example 3.1C - Subscribing Shorthand with Lambdas

Implementing a `Subscriber` is a bit verbose, so we also have the option of passing more concise lambda arguemnts to the `subscribe()` function. Then it will use those lambas to create the `Subscriber` for us.

```
from rx import Observable
31
letters = Observable.from_(["Alpha","Beta","Gamma","Delta
","Epsilon"])

letters.subscribe(on_next = lambda value: print(value),
                  on_completed = lambda: print("Completed
```

```
!"),
                    on_error = lambda error: print("Error o
ccurred: {0}".format(error)))
```

You do not even have to supply all the lambda arguments. You can leave out the `on_completed` and `on_error`, but for production code you should try to have an `on_error` so errors are not quietly swallowed.

```
letters.subscribe(on_next = lambda value: print(value))

# or

letters.subscribe(lambda value: print("Received: {0}".for
mat(value)))
```

We will be using lambdas constantly as we do reactive programming.

4.2A - Some Basic Operators

RxPy has approximately 130 operators to powerfully express business logic, transformations, and concurrency behaviors. For now we will start with two basic ones: `map()` and `filter()` and cover more in the next section.

For instance, we can `map()` each `String` to its lenth, and then filter only to lengths that are at least 5.

```
from rx import Observable

letters = Observable.from_(["Alpha","Beta","Gamma","Delta
","Epsilon"])

mapped = letters.map(lambda s: len(s))

filtered = mapped.filter(lambda i: i >= 5)

filtered.subscribe(lambda value: print(value))
```

OUTPUT:

```
Received: 5
Received: 5
Received: 5
Received: 7
```

Each operator yields a new `Observable` emitting that transformation. We can save each one to a variable if we want and then `subscribe()` to the one we want, but oftentimes you will likely want to call them all in a single chain.

```
from rx import Observable

Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
```

```
]) \
    .map(lambda s: len(s)) \
    .filter(lambda i: i >= 5) \
    .subscribe(lambda value: print(value))
```

> *If you are using an IDE like PyCharm, operators like `filter()` and `map()` will unfortunately not be available for auto-complete. The reason is RxPy will add these operators to the `Observable` at runtime. For PyCharm, you may want to disable Unresolved References under Settings -> Editor -> Inspection -> Python so you do not get any warnings.*

4.2B Using Observable.range()

There are other ways to create an `Observable`. For instance, you can emit a range of numbers:

```
from rx import Observable

letters = Observable.range(1,10)

letters.subscribe(lambda value: print(value))
```

OUTPUT:

```
Received: 1
```

```
Received: 2
Received: 3
Received: 4
Received: 5
Received: 6
Received: 7
Received: 8
Received: 9
Received: 10
```

You can also use `Observable.just()` to emit a single item.

```
from rx import Observable, Observer

greeting = Observable.just("Hello World!")

greeting.subscribe(lambda value: print(value))
```

OUTPUT:

```
Received: Hello World!
```

3.2C - Using Observable.empty()

You can also create an `Observable` that emits nothing and call

`on_completed()` immediately via `Observable.empty()`. While this may not seem useful, an empty `Observable` is the reactive equivalent to `None`, `null`, or an empty collection so you will encounter it.

```
from rx import Observable

Observable.empty() \
    .subscribe(on_next= lambda s: print(s),
               on_completed= lambda: print("Done!")
               )
```

OUTPUT:

```
Done!
```

3.3A - Creating an Observable from Scratch

You can also create an `Observable` source from scratch. Using `Observable.create()`. you can pass a function with an `observer` argument, and call it's `on_next()`, `on_completed()`, and `on_error()` to pass items or events to the `Subscriber` or the next operator in the chain.

```
from rx import Observable, Observer

```

```
def push_numbers(observer):
    observer.on_next(100)
    observer.on_next(300)
    observer.on_next(500)
    observer.on_completed()

Observable.create(push_numbers).subscribe(on_next = lambd
a i: print(i))
```

OUTPUT:

```
100
300
500
```

4.3B - An Interval Observable

Observables do not have to strictly emit data. They can also emit events. Remember our definition that *events are data, and data are events*? Events and data are treated the same way in ReactiveX. They both can be pushed through an `Observable`.

For instance, we can use `Observable.interval()` to emit a consecutive integer every 1 second.

```
from rx import Observable
```

```
Observable.interval(1000) \
    .map(lambda i: "{0} Mississippi".format(i)) \
    .subscribe(lambda s: print(s))

# Keep application alive until user presses a key
input("Press any key to quit")
```

OUTPUT:

```
0 Mississippi
1 Mississippi
2 Mississippi
3 Mississippi
4 Mississippi
5 Mississippi
6 Mississippi
7 Mississippi
8 Mississippi
```

Notice how the `Observable` in fact has a notion of time? It is emitting an integer every second, and each emission is both data and an event. Observables can be created to emit button clicks for a UI, server requests, new Tweets, and any other event while repsresenting that event as data.

Note also we had to use `input()` to make the main thread pause until the user presses a key. If we did not do this, the

`Observable.interval()` would not have a chance to fire because the application will exit. The reason for this is the `Observable.interval()` has to operate on a separate thread and create a separate workstream driven by a timer. The Python code will finish and terminate before it has a chance to fire.

3.3C - Unsubscribing from an Observable

When you `subscribe()` to an `Observable` it returns a `Disposable` so you can disconnect the `Subscriber` from the `Observable` at any time.

```
from rx import Observable
import time

disposable = Observable.interval(1000) \
    .map(lambda i: "{0} Mississippi".format(i)) \
    .subscribe(lambda s: print(s))

# sleep 5 seconds so Observable can fire
time.sleep(5)

# disconnect the Subscriber
print("Unsubscribing!")
disposable.dispose()
```

```
# sleep a bit longer to prove no more emissions are comin
g
time.sleep(5)
```

OUTPUT:

```
0 Mississippi
1 Mississippi
2 Mississippi
3 Mississippi
Unsubscribing!
```

Unsubscribing/disposing is usually not necessary for Observables that are finite and quick (they will unsubscribe themselves), but it can be necessary for long-running or infinite Observables.

3.4 - An Observable emitting Tweets

Later we will learn how to create Observables that emit Tweets for a given topic, but here is a preview of what's to come. Using Tweepy and `Observable.create()`, we can create a function that yields an `Observable` emitting Tweets for specified topics. For instance, here is how to get a live stream of text bodies from Tweets for "Britain" and "France".

4.4A - A Twitter Observable

```
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
import json
from rx import Observable

# Variables that contains the user credentials to access
Twitter API
access_token = "CONFIDENTIAL"
access_token_secret = "CONFIDENTIAL"
consumer_key = "CONFIDENTIAL"
consumer_secret = "CONFIDENTIAL"


def tweets_for(topics):

    def observe_tweets(observer):
        class TweetListener(StreamListener):
            def on_data(self, data):
                observer.on_next(data)
                return True

            def on_error(self, status):
                observer.on_error(status)
```

```
        # This handles Twitter authetification and the co
nnection to Twitter Streaming API
        l = TweetListener()
        auth = OAuthHandler(consumer_key, consumer_secret
)
        auth.set_access_token(access_token, access_token_
secret)
        stream = Stream(auth, l)
        stream.filter(track=topics)

    return Observable.create(observe_tweets).share()


topics = ['Britain','France']

tweets_for(topics).map(lambda d: json.loads(d)) \
    .filter(lambda map: "text" in map) \
    .map(lambda map: map["text"].strip()) \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
RT @YourAnonCentral: The five biggest international arms e
xports suppliers in 2008–12 were the #US,#Russia, #German
y, #France and #China. ht…
```

```
RT @parismarx: Marine Le Pen believes France "will provid
e the third stage of a global political uprising" followi
ng Brexit &amp;amp; Trump https:/…
Attentats du 13-Novembre: des rescapés racontent leur vie
 un an après https://t.co/VMM5rlsoQu via @RFI
RT @AOLNews: 1 year after the Paris attacks, France's sta
te of emergency remains: https://t.co/PD0U6mXHcN https://
t.co/QUHWRSCLxt
 https://t.co/4ImUajYSq2 @HuffPostJapan
RT @CPIF_: #France Interdit cette année, les islamistes t
entent de convertir les femmes en faisant l'expérience du
 voile à…
RT @StewartWood: This week our Government should remember
 &amp;amp; make clear that Britain's alliances must be based
on our values, not our values…
RT @MaxAbrahms: "Britain will spend the next two months t
rying to convince Mr Trump's team of the need to remove P
resident Assad." https://…
RT @Bassounov: #Trump est devenu présidentiable grâce à 1
0 ans de #téléPoubelle. En 2022 en France, la présidence
se jouera entre #Hanouna…
# Panoramix #Radio #Station
...
```

4.4B Cold vs Hot Observables

Observables that emit data typically are **cold Observables**, meaning

they will replay emissions to each individual `Subscriber`. For instance, this `Observable` below will emit all five strings to both Subscribers individually.

```
from rx import Observable

source = Observable.from_(["Alpha","Beta","Gamma","Delta"
,"Epsilon"])

source.subscribe(lambda s: print("Subscriber 1: {0}".form
at(s)))
source.subscribe(lambda s: print("Subscriber 2: {0}".form
at(s)))
```

OUTPUT:

```
Subscriber 1: Alpha
Subscriber 1: Beta
Subscriber 1: Gamma
Subscriber 1: Delta
Subscriber 1: Epsilon
Subscriber 2: Alpha
Subscriber 2: Beta
Subscriber 2: Gamma
Subscriber 2: Delta
Subscriber 2: Epsilon
```

However, **hot Observables** will not replay emissions for tardy subscribers that come later. Our Twitter `Observable` is an example of a hot `Observable`. If a second `Subscriber` subscribes to a Tweet feed 5 seconds after the first `Subscriber`, it will miss all Tweets that occurred in that window. We will explore this later.

Part V - Operators

In this section, we will learn some of the 130 operators available in RxPy. Learning these operators can be overwhelming, so the best approach is to seek the right operators out of need. The key to being productive with RxPy and unleashing its potential is to find the key operators that help you with the tasks you encounter. With practice, you will become fluent in composing them together.

The best way to see what operators are available in RxPy is to look through them on GitHub
https://github.com/ReactiveX/RxPY/tree/master/rx/linq/observable

You can also view the ReactiveX operators page which has helpful marble diagrams showing each operator's behavior
http://reactivex.io/documentation/operators.html

You can also explore various operators using the interactive RxMarbles website
http://rxmarbles.com/

5.1 Suppressing Emissions

Here are some operators that can be helpful for supressing emissions that fail to meet a criteria in some form.

5.1A `filter()`

You have already seen the `filter()`. It supresses emissions that fail to meet a condition specified by you. For instance, only allowing emissions forward that are at least length 5.

```
Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .filter(lambda s: len(s) >= 5) \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
Alpha
Gamma
Delta
Epsilon
```

5.1B `take()`

You can also use `take()` to cut off at a certain number of emissions

and call `on_completed()`. For instance, calling `take(2)` like below will only allow the first two emissions coming out of the `filter()` to come through.

```
Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .filter(lambda s: len(s) >= 5) \
    .take(2) \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
Alpha
Gamma
```

`take()` will not throw an error if it fails to get the number of items it wants. It will just emit what it does capture. For instance, when `take(10)` only recieves 4 emissions (and not 10), it will just emit those 4 emissions.

```
from rx import Observable

Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .filter(lambda s: len(s) >= 5) \
    .take(10) \
```

```
    .subscribe(on_next = lambda s: print(s), on_error = l
ambda e: print(e))
```

OUTPUT:

```
Alpha
Beta
Gamma
Delta
Epsilon
```

5.1C `take_while()`

`take_while()` and `take_until()` will keep passing emissions based on a condition. For instance if we have an `Observable` emitting some integers, we can keep taking integers while they are less than 100. We can achieve this using a `take_while()`.

```
from rx import Observable

Observable.from_([2,5,21,5,2,1,5,63,127,12]) \
    .take_while(lambda i: i < 100) \
    .subscribe(on_next = lambda i: print(i), on_completed
 = lambda: print("Done!"))
```

When the `127` is encountered, the `take_while()` specified as above

with the condition `i < 100` will trigger `on_completed()` to be called to the `Subscriber`, and unsubscription will prevent any more emissions from occurring.

4.2 Distinct Operators

5.2A `distinct()`

You can use `distinct()` to suppress redundant emissions. If an item has been emitted before (based on its equality logic via its `__eq__` implementation), it will not be emitted.

This will emit the distinct lengths

```
from rx import Observable

Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Eps
ilon"]) \
    .map(lambda s: len(s)) \
    .distinct() \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
5
4
7
```

5.2B `distinct()` with mapping

You can also pass a lambda specifying what you want to distinct on. If we want to emit the `String` rather than its length, but use distinct logic on its length, you can leverage a lambda argument.

```
from rx import Observable


Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Eps
ilon"]) \
    .distinct(lambda s: len(s)) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
Alpha
Beta
Epsilon
```

5.2C `distinct_until_changed()`

The `distinct_until_changed()` will prevent *consecutive* duplicates from emitting.

```
from rx import Observable


```

```
Observable.from_(["Alpha", "Theta", "Kappa", "Beta", "Gam
ma", "Delta", "Epsilon"]) \
    .map(lambda s: len(s)) \
    .distinct_until_changed() \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
5
4
5
7
```

Just like `distinct()`, you can also provide a lambda to distinct on an attribute.

```
from rx import Observable

Observable.from_(["Alpha", "Theta", "Kappa", "Beta", "Gam
ma", "Delta", "Epsilon"]) \
    .distinct_until_changed(lambda s: len(s)) \
    .subscribe(lambda i: print(i))
```

```
Alpha
Beta
Gamma
```

```
Epsilon
```

4.3 Aggregating Operators

When working with data, there will be many instances where we want to consolidate emissions into a single emission to reflect some form of an aggregated result.

With the exception of `scan()`, one thing to be careful about when aggregating emissions is they rely on `on_completed()` to be called. Infinite Observables will cause an aggregation operator to work forever aggregating an infinite series of emissions.

5.3A - `count()`

The simplest aggregation to an `Observable` is to simply `count()` the number of emisssions, and then push that count forward as a single emission once `on_completed()` is called. If we want to count the number of text strings that are not 5 characters, we can achieve it like this:

```
from rx import Observable

Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .filter(lambda s: len(s) != 5) \
    .count() \
```

```
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
2
```

5.3B `reduce()`

The `reduce()` allows you to define a custom aggregation operation to "fold" each value into a rolling value. For instance, you can find the sum of numeric emissions (less than 100) using `reduce()` in this manner.

```
from rx import Observable

Observable.from_([4,76,22,66,881,13,35]) \
    .filter(lambda i: i < 100) \
    .reduce(lambda total, value: total + value) \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
216
```

You can use this to consolidate emissions in your own custom way for

most cases. Keep in mind that there are already built in mathematical aggregators like `sum()` (which could replace this `reduce()`) as well as `min()`, `max()`, and `average()`. These only work on numeric emissions, however.

5.3C `scan()`

The `scan()` is almost identical to `reduce()`, but it will emit each rolling total for each emission that is received. Therefore, it can work with infinite Observables such as Twitter streams and other events.

```
from rx import Observable

Observable.from_([4,76,22,66,881,13,35]) \
    .scan(lambda total, value: total + value) \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
4
80
102
168
1049
1062
1097
```

Each accumulation is emitted every time an emission is added to our running total. We start with `4`, then `4` + `76` which is `80`, then `80` + `22` which is `102`, etc...

4.4 Collecting Operators

You can consolidate emissions by collecting them into a `List` or `Dict`, and then pushing that collection forward as a single emission.

5.4A - `to_list()`

`to_list()` will collect the emissions into a single `List` until `on_completed()` is called, then it will push that `List` forward as a single emission.

```
from rx import Observable

Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .to_list() \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
['Alpha', 'Beta', 'Gamma', 'Delta', 'Epsilon']
```

Typically you want avoid excessively collecting things into Lists unless business logic requires it. Prefer to keep emissions flowing forward one-at-a-time in a reactive manner when possible, rather than stopping the flow and collecting emissions into Lists.

5.4B - `to_dict()`

The `to_dict()` will collect emissions into a `Dict` and you specify a lambda that derives the key. For instance, if you wanted to key each String off its first letter and collect them into a `Dict`, do the following:

```
from rx import Observable

Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Eps
ilon"]) \
    .to_dict(lambda s: s[0]) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
{'B': 'Beta', 'E': 'Epsilon', 'A': 'Alpha', 'G': 'Gamma',
 'D': 'Delta'}
```

You can optionally provide a second lambda argument to specify a value other than the emission itself. If we wanted to map the first letter to the length of the String instead, we can do this:

```
from rx import Observable

Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Eps
ilon"]) \
    .to_dict(lambda s: s[0], lambda s: len(s)) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
{'A': 5, 'B': 4, 'D': 5, 'G': 5, 'E': 7}
``
# Section VI - Combining Observables

We can combine multiple Observables into a single `Observ
able`, and bring their emissions together in various ways
.

# 6.1 Merge

We can merge two or more Observables using the `Observabl
e.merge()` function, and this will yield a new `Observabl
e` pushing emissions from all of them.

## 6.1A - Observable.merge()

```python
from rx import Observable
```
```

```
source1 = Observable.from_(["Alpha","Beta","Gamma","Delta
","Epsilon"])
source2 = Observable.from_(["Zeta","Eta","Theta","Iota"])

Observable.merge(source1,source2) \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
Alpha
Zeta
Beta
Eta
Gamma
Theta
Delta
Iota
Epsilon
```

Notice that although emissions from both Observable are now a single stream, the emissions are interleaved and jumbled. This is because `Observable.merge()` will fire emissions from all the Observables at once rather than sequentially one-at-a-time.

6.1B - Observable.merge()

(Continued)

If you want this sequential ordered guarantee, you will want to use `Observable.concat()` which is discussed later. But the `Observable.merge()` can be helpful for merging multiple event streams.

```
from rx import Observable

source1 = Observable.interval(1000).map(lambda i: "Source
 1: {0}".format(i))
source2 = Observable.interval(500).map(lambda i: "Source
2: {0}".format(i))
source3 = Observable.interval(300).map(lambda i: "Source
3: {0}".format(i))

Observable.merge(source1, source2, source3) \
    .subscribe(lambda s: print(s))

# keep application alive until user presses a key
input("Press any key to quit\n")
```

OUTPUT:

```
Source 3: 0
Source 2: 0
Source 3: 1
```

```
Source 3: 2
Source 1: 0
Source 2: 1
Source 3: 3
Source 2: 2
Source 3: 4
Source 3: 5
Source 2: 3
Source 1: 1
etc...
```

Three infinite Observables above are emitting a consecutive integer at different intervals (1000 milliseconds, 500 milliseconds, and 300 milliseconds), and putting each integer into a String labeling the source. But we merged these three infinite Observables into one using `Observable.merge()`.

6.1C - `merge_all()`

Another way to accomplish this is to make a List containing all three Observables, and then passing it to `Observable.from_()`. This will make an Observable emitting Observables, then you can call `merge_all()` to turn each one into its emissions.

```
from rx import Observable

source1 = Observable.interval(1000).map(lambda i: "Source
```

```
 1: {0}".format(i))
source2 = Observable.interval(500).map(lambda i: "Source
2: {0}".format(i))
source3 = Observable.interval(300).map(lambda i: "Source
3: {0}".format(i))

Observable.from_([source1,source2,source3]) \
    .merge_all() \
    .subscribe(lambda s: print(s))

# keep application alive until user presses a key
input("Press any key to quit\n")
```

6.1D - `merge_all()` (Continued)

If you are creating an `Observable` off each emission on-the-fly, `merge_all()` can be helpful here as well. Say you have a list of Strings containing numbers separated by `/`. You can map each String to be `split()` and then pass those separated values to an `Observable.from_()`. Then you can call `merge_all()` afterwards.

```
from rx import Observable

items = ["134/34/235/132/77", "64/22/98/112/86/11", "66/0
8/34/778/22/12"]

Observable.from_(items) \
```

```
    .map(lambda s: Observable.from_(s.split("/"))) \
    .merge_all() \
    .map(lambda s: int(s)) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
134
34
64
235
22
66
132
98
8
77
112
34
86
778
11
22
12
```

6.1E - `flat_map()`

An alternative way of expressing the previous example (5.1D) is using `flat_map()`. It will consolidate mapping to an `Observable` and calling `merge_all()` into a single operator.

```
from rx import Observable

items = ["134/34/235/132/77", "64/22/98/112/86/11", "66/0
8/34/778/22/12"]

Observable.from_(items) \
    .flat_map(lambda s: Observable.from_(s.split("/"))) \
    .map(lambda s: int(s)) \
    .subscribe(lambda i: print(i))
```

We will try to prefer the `flat_map()` over the `map()`/`merge_all()` from now on since it is much more succinct.

6.2 Concat and Zip

`Observable.concat()` and the `concat_all()` operator are simliar to `Observable.merge()` and the `merge_all()` operator. The only difference is they will emit items from each `Observable` *sequentially*. It will fire off each `Observable` in order and one-at-a-time. Therefore, this not something you want to use with infinite Observables, because the first infinite `Observable` will occupy its place in the queue forever

and stop the `Observables` behind it from firing. They are helpful for finite data sets though.

6.2A - `concat()`

Our previous `merge()` example can now emit items in order:

```
from rx import Observable

source1 = Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"])
source2 = Observable.from_(["Zeta","Eta","Theta","Iota"])

Observable.concat(source1,source2) \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
Alpha
Beta
Gamma
Delta
Epsilon
Zeta
Eta
Theta
Iota
```

6.2B - `concat_all()`

We can make our earlier example splitting Strings ordered using `concat_all()` instead of `merge_all()`.

```
from rx import Observable

items = ["134/34/235/132/77", "64/22/98/112/86/11", "66/0
8/34/778/22/12"]

Observable.from_(items) \
    .map(lambda s: Observable.from_(s.split("/"))) \
    .concat_all() \
    .map(lambda s: int(s)) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
134
34
235
132
77
64
22
98
112
```

```
86
11
66
08
34
778
22
12
```

If you do not care about ordering, it is recommend to use `merge_all()` or `flat_map()`. `concat_all()` can behave unpredictably with certain operators like `group_by()`, which we will cover later.

6.2C - Zip

Zipping pairs emissions from two or more sources and turns them into a single `Observable`.

```
from rx import Observable

letters = Observable.from_(["A","B","C","D","E","F"])
numbers = Observable.range(1,5)

Observable.zip(letters,numbers, lambda l,n: "{0}-{1}".for
mat(l,n)) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
A-1
B-2
C-3
D-4
E-5
```

You can alternatively express this as an operator.

```
letters.zip(numbers, lambda l,n: "{0}-{1}".format(l,n)) \
    .subscribe(lambda i: print(i))
```

6.3D - Using Zip to Space Emissions

Zip can also be helpful to space out emissions by zipping an Observable with an `Observable.interva()`. For instnance, we can space out five emissions by one second intervals.

```
from rx import Observable

letters = Observable.from_(["Alpha","Beta","Gamma","Delta
","Epsilon"])
intervals = Observable.interval(1000)
```

```
Observable.zip(letters,intervals, lambda s,i: s) \
    .subscribe(lambda s: print(s))

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

Note that `zip()` can get overwhelmed with infinite hot Observables where one produces emissions faster than another. You might want to consider using `combine_latest()` or `with_latest_from()` instead of `zip()`, which will pair with the latest emission from each source. For the sake of brevity, we will not cover this in this course. But you can read more about it in the ReactiveX documentation.

6.4 group_by

For the purposes of data science, one of the most powerful operators in ReactiveX is `group_by()`. It will yield an Observable emitting GroupedObservables, where each `GroupedObservable` pushes items with a given key. It behaves just like any other `Observable`, but it has a `key` property which we will leverage in a moment.

But first, let's group some `String` emissions by keying on their lengths. Then let's collect emissions for each grouping into a `List`. Then we can call `flat_map()` to yield all the Lists.

```
from rx import Observable

items = ["Alpha", "Beta", "Gamma", "Delta", "Epsilon"]
```

```
Observable.from_(items) \
    .group_by(lambda s: len(s)) \
    .flat_map(lambda grp: grp.to_list()) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
['Alpha', 'Gamma', 'Delta']
['Beta']
['Epsilon']
```

`group_by()` is efficient because it is still 100% reactive and pushing items one-at-a-time through the different GroupedObservables. You can also leverage the `key` property and tuple it up with an aggregated value. This is helpful if you want to create `Dict` that holds aggregations by key values.

For instance, if you want to find the count of each word length occurrence, you can create a `Dict` like this:

```
from rx import Observable

items = ["Alpha", "Beta", "Gamma", "Delta", "Epsilon"]

Observable.from_(items) \
    .group_by(lambda s: len(s)) \
```

```
    .flat_map(lambda grp:
        grp.count().map(lambda ct: (grp.key, ct))
    ) \
    .to_dict(lambda key_value: key_value[0], lambda key_v
alue: key_value[1]) \
    .subscribe(lambda i: print(i))
```

OUTPUT:

```
{4: 1, 5: 3, 7: 1}
```

You can interpret the returned `Dict` above as “for length 4 there are one occurrences, for length 5 there are 3 occurrences, etc”.

`group_by()` is somewhat abstract but it is a powerful and efficient way to perform aggregations on a given key. It also works with infinite Observables assuming you use infinite-friendly operators on each `GroupedObservable`. We will use `group_by()` a few more times in this course.

Section VII - Reading and Analyzing data

In this chapter we will look over basic ways to reactively read data and analyze data from text files, URL’s, and SQL. We will also integrate concepts we previously learned to create a reactive word counter that

runs on a schedule and detects changes to a file.

One catch with using `Observable.from_()` with a data source iterable is it only iterates once, causing multiple Subscribers to not receieve data after the first Subscriber. To get around this we will use functions to create a new `Observable` each time we need to subscribe to a data source. A slightly more advanced way to solve this issue is to use `Observable.defer()` which we will not cover here, but you can read about it in the Appendix.

It is good to leverage functions that return Observables anyway. You can accept arguments to build the Observable chain that is returned and increase reusability.

7.1A - Reading a Text File

As stated earlier, anything that is iterable can be turned into an `Observable` using `Observable.from_()`. We can emit the lines from a text file in this manner. If I have a raw text file called `bbc_news_article.txt` in my Python project, I can emit the lines like this:

```
from rx import Observable


def read_lines(file_name):
    file = open(file_name)
```

```
    return Observable.from_(file) \
        .map(lambda l: l.strip()) \
        .filter(lambda l: l != "")


read_lines("bbc_news_article.txt").subscribe(lambda s: pr
int(s))
```

OUTPUT:

```
Giant waves damage S Asia economy
Governments, aid agencies, insurers and travel firms are
among those counting the cost of the massive earthquake a
nd waves that hammered southern Asia.
The worst-hit areas are Sri Lanka, India, Indonesia and T
hailand, with at least 23,000 people killed. Early estima
tes from the World Bank put the amount of aid needed at a
bout $5bn (£2.6bn), similar to the cash offered Central A
merica after Hurricane Mitch. Mitch killed about 10,000 p
eople and caused damage of about $10bn in 1998. World Ban
k spokesman Damien
...
```

I use the `map()` and `filter()` operators to strip any leading and trailing whitespace for each line, as well as rid lines that are empty.

We will use this example for a project at the end of this section.

7.1B - Reading a URL

You can also read content from the web in a similar manner. This can be a powerful way to do web scraping and data wrangling, especially if you reactively push multiple URL's or URL arguments and scrape the content off each page. Just be kind and don't tax somebody's system!

I saved a simple raw text page of the 50 U.S. states on a Gist page. You can view it with this URL: https://goo.gl/rIaDyM.

If you want to read the lines off the response, you can do it like this:

```
from rx import Observable
from urllib.request import urlopen


def read_request(link):
    f = urlopen(link)

    return Observable.from_(f) \
        .map(lambda s: s.decode("utf-8").strip()) \

read_request("https://goo.gl/rIaDyM") \
    .subscribe(lambda s: print(s))
```

OUTPUT:

```
Alabama
Alaska
Arizona
Arkansas
California
Colorado
Connecticut
Delaware
...
```

In the map we have to decode the bytes and convert them to UTF-8 Strings. Then we also clean leading and trailing whitespace with `strip()`. then finally we print each line.

7.2 - Reading a SQL Query

SQLAlchemy is the go-to Python library for SQL querying, and since it is iterable it can easily support Rx. In this example, I am using a SQLite database file which you can download at https://goo.gl/9DYXPS. You can also download it on my *Getting Started with SQL* GitHub page.

7.2A - Emitting a query

When you set up your engine, statement, and connection, you can reactively emit each result (which will be a tuple) from a query using `Observable.from_()`. Since a SQL query result set can only be iterated once, it is easiest to use a function to create a new one and

return it in an `Observable` each time. That way multiple subscribers can be accommodated easily.

```
from sqlalchemy import create_engine, text
from rx import Observable

engine = create_engine('sqlite:///C:\\Users\\thoma\\Dropbox\\rexon_metals.db')
conn = engine.connect()


def get_all_customers():
    stmt = text("SELECT * FROM CUSTOMER")
    return Observable.from_(conn.execute(stmt))


get_all_customers().subscribe(lambda r: print(r))
```

OUTPUT:

```
(1, 'LITE Industrial', 'Southwest', '729 Ravine Way', 'Irving', 'TX', 75014)
(2, 'Rex Tooling Inc', 'Southwest', '6129 Collie Blvd', 'Dallas', 'TX', 75201)
(3, 'Re-Barre Construction', 'Southwest', '9043 Windy Dr', 'Irving', 'TX', 75032)
(4, 'Prairie Construction', 'Southwest', '264 Long Rd', '
```

```
Moore', 'OK', 62104)
(5, 'Marsh Lane Metal Works', 'Southeast', '9143 Marsh Ln
', 'Avondale', 'LA', 79782)
```

7.2B - Merging multiple queries

You can create some powerful reactive patterns when working with databases. For instance, say you wanted to query for customers with ID's 1, 3, and 5. Of course you can do this in raw SQL like so:

```
SELECT * FROM CUSTOMER WHERE CUSTOMER_ID in (1,3,5)
```

However, let's leverage Rx to keep our API simple and minimize the number of query functions it needs.

You can create a single `customer_for_id()` function that returns an `Observable` emitting a customer for a given `customer_id`. You can compose it into a reactive chain by using `merge_all()` or `flat_map()`. Do this by emitting the desired ID's, mapping them to the `customer_for_id()`, and then calling `merge_all()` to consolidate the results from all three queries.

```
from sqlalchemy import create_engine, text
from rx import Observable

engine = create_engine('sqlite:///C:\\Users\\thoma\\Dropb
ox\\rexon_metals.db')
```

```
conn = engine.connect()

def get_all_customers():
    stmt = text("SELECT * FROM CUSTOMER")
    return Observable.from_(conn.execute(stmt))

def customer_for_id(customer_id):
    stmt = text("SELECT * FROM CUSTOMER WHERE CUSTOMER_ID
 = :id")
    return Observable.from_(conn.execute(stmt, id=custome
r_id))

# Query customers with IDs 1, 3, and 5
Observable.from_([1, 3, 5]) \
    .flat_map(lambda id: customer_for_id(id)) \
    .subscribe(lambda r: print(r))
```

OUTPUT:

```
(1, 'LITE Industrial', 'Southwest', '729 Ravine Way', 'Ir
ving', 'TX', 75014)
(3, 'Re-Barre Construction', 'Southwest', '9043 Windy Dr'
, 'Irving', 'TX', 75032)
(5, 'Marsh Lane Metal Works', 'Southeast', '9143 Marsh Ln
', 'Avondale', 'LA', 79782)
```

You can also use Rx to write data to databases using operators or Subscribers. This is beyond the scope of this course, but just remember that any iterable can be turned into an `Observable`!

7.3 - A Scheduled Reactive Word Counter

Let's apply everything we have learned so far to create a reactive word counter process.

7.3A - Emitting words from a text file

Let's start by creating a function that returns an `Observable` emitting and cleaning the words in a text file, ridding punctuation, empty lines, and making all words lower case.

```
from rx import Observable
import re



def words_from_file(file_name):
    file = open(file_name)

    # parse, clean, and push words in text file
    return Observable.from_(file) \
        .flat_map(lambda s: Observable.from_(s.split())) \
```

```
        .map(lambda w: re.sub(r'[^\w]', '', w)) \
        .filter(lambda w: w != "") \
        .map(lambda w: w.lower())

article_file = "bbc_news_article.txt"
words_from_file(article_file).subscribe(lambda w: print(w
))
```

OUTPUT:

```
giant
waves
damage
governments
s
aid
asia
agencies
the
economy
...
```

7.3B - Counting Word Occurrences

Let's create another function called `word_counter()`. It will leverage the existing `words_from_file()` then use `group_by()` to count the word occurrances, then tuple the word with the count.

```
from rx import Observable
import re


def words_from_file(file_name):
    file = open(file_name)

    # parse, clean, and push words in text file
    return Observable.from_(file) \
        .flat_map(lambda s: Observable.from_(s.split())) \
        .map(lambda w: re.sub(r'[^\w\s]', '', w)) \
        .filter(lambda w: w != "") \
        .map(lambda w: w.lower()) \


def word_counter(file_name):

    # count words using `group_by()`
    # tuple the word with the count
    return words_from_file(file_name) \
        .group_by(lambda word: word) \
        .flat_map(lambda grp: grp.count().map(lambda ct: (grp.key, ct)))

article_file = "bbc_news_article.txt"
```

```
word_counter(article_file).subscribe(lambda w: print(w))
```

OUTPUT:

```
('giant', 1)
('waves', 3)
('damage', 6)
('governments', 3)
('s', 1)
('aid', 10)
('asia', 6)
('agencies', 3)
('the', 78)
('economy', 1)
...
```

7.3C - Scheduling the Word Count And Notifying of Changes

Finally, let's schedule this word count to occur every 3 seconds and collect them into a `Dict`. We can use `distinct_until_changed()` to only emit `Dict` items that have changed due to the text file being edited.

```
# Schedules a reactive process that counts the words in a
 text file every three seconds,
```

```
# but only prints it as a dict if it has changed

from rx import Observable
import re


def words_from_file(file_name):
    file = open(file_name)

    # parse, clean, and push words in text file
    return Observable.from_(file) \
        .flat_map(lambda s: Observable.from_(s.split())) \
        .map(lambda w: re.sub(r'[^\w\s]', '', w)) \
        .filter(lambda w: w != "") \
        .map(lambda w: w.lower()) \



def word_counter(file_name):

    # count words using `group_by()`
    # tuple the word with the count
    return words_from_file(file_name) \
        .group_by(lambda word: word) \
        .flat_map(lambda grp: grp.count().map(lambda ct:
(grp.key, ct)))

```

```
# composes the above word_counter() into a dict
def word_counter_as_dict(file_name):
    return word_counter(file_name).to_dict(lambda t: t[0]
, lambda t: t[1])


# Schedule to create a word count dict every three second
s an article
# But only re-print if text is edited and word counts cha
nge

article_file = "bbc_news_article.txt"

# create a dict every three seconds, but only push if it
changed
Observable.interval(3000) \
    .flat_map(lambda i: word_counter_as_dict(article_file
))
    .distinct_until_changed() \
    .subscribe(lambda word_ct_dict: print(word_ct_dict))

# Keep alive until user presses any key
input("Starting, press any key to quit\n")
```

OUTPUT:

```
Starting, press any key to quit
{'a': 7, 'governments': 3, 'first': 1, 'getting': 1, 'off
ered': 1, ...
```

Every time the file is edited and words are added, modified, or removed, it should push a new `Dict` reflecting these changes. This can be helpful to run a report on a schedule, and you can only emit a new report to an output if the data has changed.

Ideally, it is better to hook onto the change event itself rather than running a potentially expensive process every 3 seconds. We will learn how to do this with Twitter in the next section.

> *If you want to see an intensive reactive data analysis example, see my social media example on Gist*

Section VIII - Hot Observables

In this section we will learn how to create an `Observable` emitting Tweets for a set of topics. We will wrap an `Observable.create()` around the Tweepy API. But first, let's cover multicasting.

8.1A - Creating a `ConnectableObservable`

Remember how cold Observables will replay data to each Subscriber

like a music CD?

```
from rx import Observable

source = Observable.from_(["Alpha","Beta","Gamma","Delta"
,"Epsilon"])

source.subscribe(lambda s: print("Subscriber 1: {0}".form
at(s)))
source.subscribe(lambda s: print("Subscriber 2: {0}".form
at(s)))
```

OUTPUT:

```
Subscriber 1: Alpha
Subscriber 1: Beta
Subscriber 1: Gamma
Subscriber 1: Delta
Subscriber 1: Epsilon
Subscriber 2: Alpha
Subscriber 2: Beta
Subscriber 2: Gamma
Subscriber 2: Delta
Subscriber 2: Epsilon
```

This is often what we want so no data is missed for each Subscriber.

But there are times we will want to force cold Observables to become

hot Observables. We can do this by calling `publish()` which will return a `ConnectableObservable`. Then we can subscribe our Subscribers to it, then call `connect()` to fire emissions to all Subscribers at once.

```
from rx import Observable

source = Observable.from_(["Alpha","Beta","Gamma","Delta"
,"Epsilon"]).publish()

source.subscribe(lambda s: print("Subscriber 1: {0}".form
at(s)))
source.subscribe(lambda s: print("Subscriber 2: {0}".form
at(s)))

source.connect()
```

OUTPUT:

```
Subscriber 1: Alpha
Subscriber 2: Alpha
Subscriber 1: Beta
Subscriber 2: Beta
Subscriber 1: Gamma
Subscriber 2: Gamma
Subscriber 1: Delta
Subscriber 2: Delta
```

```
Subscriber 1: Epsilon
Subscriber 2: Epsilon
```

This is known as multicasting. Notice how the emissions are now interleaved? This is because each emission is going to both subscribers. This is helpful if “replaying” the data is expensive or we just simply want all Subscribers to get the emissions simultaneously.

8.1B - Sharing an Interval Observable

`Observable.interval()` is actually a cold Observable too. If one Subscriber subscribes to it, and 5 seconds later another Subscriber comes in, that second subscriber will receive its own emissions that “start over”.

```
from rx import Observable
import time

source = Observable.interval(1000)

source.subscribe(lambda s: print("Subscriber 1: {0}".form
at(s)))

# sleep 5 seconds, then add another subscriber
time.sleep(5)
source.subscribe(lambda s: print("Subscriber 2: {0}".form
```

```
at(s)))


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

OUTPUT:

```
Subscriber 1: 0
Subscriber 1: 1
Subscriber 1: 2
Subscriber 1: 3
Press any key to exit
Subscriber 1: 4
Subscriber 2: 0
Subscriber 1: 5
Subscriber 2: 1
Subscriber 1: 6
Subscriber 2: 2
Subscriber 1: 7
Subscriber 2: 3
```

Subscriber 2 starts at `0` while Subscriber 2 is already at `4`. If we want both to be on the same timer, we can use `publish()` to create a `ConnectableObservable`.

```
from rx import Observable
import time
```

```
source = Observable.interval(1000).publish()

source.subscribe(lambda s: print("Subscriber 1: {0}".form
at(s)))
source.connect()

# sleep 5 seconds, then add another subscriber
time.sleep(5)
source.subscribe(lambda s: print("Subscriber 2: {0}".form
at(s)))

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

OUTPUT:

```
Subscriber 1: 0
Subscriber 1: 1
Subscriber 1: 2
Subscriber 1: 3
Press any key to exit
Subscriber 1: 4
Subscriber 2: 4
Subscriber 1: 5
Subscriber 2: 5
```

8.1C - Autoconnecting

We can have our `ConnectableObservable` automatically `connect()` itself when it gets a Subscriber by calling `ref_count()` on it.

```
from rx import Observable
import time

source = Observable.interval(1000).publish().ref_count()

source.subscribe(lambda s: print("Subscriber 1: {0}".form
at(s)))

# sleep 5 seconds, then add another subscriber
time.sleep(5)
source.subscribe(lambda s: print("Subscriber 2: {0}".form
at(s)))

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

You can also call an alias for `publish().ref_count()` by calling `share()` instead.

```
source = Observable.interval(1000).share()
```

Again, multicasting is helpful when you want all Subscribers to receive

the same emissions simultaneously

and prevent redundant, expensive work for each Subscriber.

8.2 - Querying Live Twitter Feeds

You can use `Observable.create()` to wrangle and analyze a live Twitter feed.

You will need to create your own application and access keys/tokens at https://apps.twitter.com.

If we want to query a live stream of Tweets pertaining to the topics of “Britain” or “France”, we can do it like this:

```
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
import json
from rx import Observable

# Variables that contains the user credentials to access
Twitter API
access_token = "PUT YOURS HERE"
access_token_secret = "PUT YOURS HERE"
consumer_key = "PUT YOURS HERE"
consumer_secret = "PUT YOURS HERE"
```

```
def tweets_for(topics):
    def observe_tweets(observer):
        class TweetListener(StreamListener):
            def on_data(self, data):
                observer.on_next(data)
                return True

            def on_error(self, status):
                observer.on_error(status)

        # This handles Twitter authetification and the co
nnection to Twitter Streaming API
        l = TweetListener()
        auth = OAuthHandler(consumer_key, consumer_secret
)
        auth.set_access_token(access_token, access_token_
secret)
        stream = Stream(auth, l)
        stream.filter(track=topics)

    return Observable.create(observe_tweets).share()


topics = ['Britain', 'France']

tweets_for(topics) \
    .map(lambda d: json.loads(d)) \
    .subscribe(on_next=lambda s: print(s), on_error=lambd
```

```
a e: print(e))
```

IX - Concurrency

(Refer to slides to cover concurrency concepts).

9.1 - Using `subscribe_on()`

9.1A - Two Long-Running Processes

We will not dive too deep into concurrency topics, but we will learn enough to make it useful and speed up slow processes.

> *Keep in mind your output may be different than mine, because concurrency tends to shuffle emissions of multiple sources. Output is almost never deterministic when multiple threads are doing work simultaneously and being merged.*

Below, we create two Observables we will call “Process 1” and “Process 2”. The first Observable is emitting five strings and the other emits numbers in a range. These Observables will fire quickly when subscribed to, but concurrency is more useful and apparent with long-running tasks. To emulate long-running expensive processes, we will need to exaggerate and slow down emissions. We can use a `intense_calculation()` function that sleeps for a short random duration (between 0.5 to 2.0 seconds) before returning the value it was

given. Then we can use this in a `map()` operator for each `Observable`.

We will use `current_thread().name` to identify the thread that is calling each `on_next()` in the `Subscriber`. Python will label each thread it creates consecutively as “Thread-1”, “Thread-2”, “Thread-3”, etc.

Before “Process 2” can start, it must wait for “Process 1” to call `on_completed()` because by default both are on the `ImmediateScheduler`. This scheduler uses the same `MainThread` that runs our Python program.

```
from rx import Observable
from threading import current_thread
import multiprocessing, time, random

def intense_calculation(value):
    # sleep for a random short duration between 0.5 to 2.
0 seconds to simulate a long-running calculation
    time.sleep(random.randint(5,20) * .1)
    return value

# Create Process 1
Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe(on_next=lambda s: print("PROCESS 1: {0} {1
```

```
}".format(current_thread().name, s)),
                on_error=lambda e: print(e),
                on_completed=lambda: print("PROCESS 1 done
!"))

# Create Process 2
Observable.range(1,10) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe(on_next=lambda i: print("PROCESS 2: {0} {1
}".format(current_thread().name, i)),
                on_error=lambda e: print(e),
                on_completed=lambda: print("PROCESS 2 done
!"))

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

OUTPUT (May not match yours):

```
PROCESS 1: MainThread Alpha
PROCESS 1: MainThread Beta
PROCESS 1: MainThread Gamma
PROCESS 1: MainThread Delta
PROCESS 1: MainThread Epsilon
PROCESS 1 done!
PROCESS 2: MainThread 1
PROCESS 2: MainThread 2
PROCESS 2: MainThread 3
```

```
PROCESS 2: MainThread 4
PROCESS 2: MainThread 5
PROCESS 2: MainThread 6
PROCESS 2: MainThread 7
PROCESS 2: MainThread 8
PROCESS 2: MainThread 9
PROCESS 2: MainThread 10
PROCESS 2 done!
```

9.1B - Kicking off both processes simultaneously

This would go much faster if we kick off both “Process 1” and “Process 2” simultaneously. We can kick off the Subscription in “Process 1” and then immediately move on to kicking off “Process 2”. We will kick off both of their subscriptions simultaneously.

In advance, we can create a `ThreadPoolScheduler` that holds a number of threads equaling the *number of CPU’s on your computer* + 1. If your computer has 4 cores, the `ThreadPoolScheduler` will have 5 threads. The reason for the extra thread is to utilize any idle time of the other threads. To make the Observables work on this `ThreadPoolScheduler`, we can pass it to a `subscribe_on()` operator anywhere in the chain. The `subscribe_on()`, no matter where it is in the chain, will instruct the source Observable what thread to push items on.

> *You are welcome to experiment and specify your own arbitrary number of threads. Just keep in mind there will be a point of diminishing return.*

The code below will execute all the above:

```
from rx import Observable
from rx.concurrency import ThreadPoolScheduler
from threading import current_thread
import multiprocessing, time, random


def intense_calculation(value):
    # sleep for a random short duration between 0.5 to 2.
0 seconds to simulate a long-running calculation
    time.sleep(random.randint(5,20) * .1)
    return value

# calculate number of CPU's and add 1, then create a Thre
adPoolScheduler with that number of threads
optimal_thread_count = multiprocessing.cpu_count() + 1
pool_scheduler = ThreadPoolScheduler(optimal_thread_count
)

print("We are using {0} threads".format(optimal_thread_co
unt))
```

```
# Create Process 1
Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe_on(pool_scheduler) \
    .subscribe(on_next=lambda s: print("PROCESS 1: {0} {1
}".format(current_thread().name, s)),
               on_error=lambda e: print(e),
               on_completed=lambda: print("PROCESS 1 done
!"))

# Create Process 2
Observable.range(1,10) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe_on(pool_scheduler) \
    .subscribe(on_next=lambda i: print("PROCESS 2: {0} {1
}".format(current_thread().name, i)),
               on_error=lambda e: print(e),
               on_completed=lambda: print("PROCESS 2 done
!"))

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

OUTPUT (May not match yours):

```
PROCESS 1: Thread-1 Alpha
```

```
PROCESS 2: Thread-2 1
PROCESS 1: Thread-1 Beta
PROCESS 1: Thread-1 Gamma
PROCESS 2: Thread-2 2
PROCESS 2: Thread-2 3
PROCESS 1: Thread-1 Delta
PROCESS 2: Thread-2 4
PROCESS 1: Thread-1 Epsilon
PROCESS 1 done!
PROCESS 2: Thread-2 5
PROCESS 2: Thread-2 6
PROCESS 2: Thread-2 7
PROCESS 2: Thread-2 8
PROCESS 2: Thread-2 9
PROCESS 2: Thread-2 10
PROCESS 2 done!
```

We use the `input()` function to hold the `MainThread` and keep the application alive until a key is pressed, allowing the Observables to fire. Notice how the emissions between Process 1 and Process 2 are interleaved, indicating they are both working at the same time. If we did not have the `subscribe_on()` calls, “Process 1” would have to finish before “Process 2” can start, because they both would use the default `ImmediateScheduler` as shown earlier.

Notice also that “Process 1” requested a thread from our `ThreadPoolScheduler` and got `Thread-1`, and “Process 2” got

`Thread 2`. They both will continue to use these threads until `on_completed()` is called on their Subscribers. Then the threads will be given back to the `ThreadPoolScheduler` so they can be used again later.

9.2 - Using `observe_on()` to redirect in the middle of the chain

Not all source Observables will respect a `subscribe_on()` you specify. This is especially true for time-driven sources like `Observable.interval()` which will use the `TimeoutScheduler` and effectively ignore any `subscribe_on()` you try to call. However, although you cannot instruct the source to emit on a different scheduler, you can specify a different scheduler to be used *at a certain point* in the `Observable` chain by using `observe_on()`.

Let's create a third process called "Process 3". The source will be an `Observable.interval()` which will emit on the `TimeoutScheduler`. After each emitted number is multiplied by 100, the emission is then moved to the `ThreadPoolScheduler` via the `observe_on()` operator. This means for the remaining operators, the emissions will be passed on the `ThreadPoolScheduler`. Unlike `subscribe_on()`, the placement of `observe_on()` does matter as it will redirect to a different executor *at that point* in the chain.

```
from rx import Observable
```

```
from rx.concurrency import ThreadPoolScheduler
from threading import current_thread
import multiprocessing, time, random

def intense_calculation(value):
    # sleep for a random short duration between 0.5 to 2.
0 seconds to simulate a long-running calculation
    time.sleep(random.randint(5,20) * .1)
    return value

# calculate number of CPU's and add 1, then create a Thre
adPoolScheduler with that number of threads
optimal_thread_count = multiprocessing.cpu_count() + 1
pool_scheduler = ThreadPoolScheduler(optimal_thread_count
)

# Create Process 1
Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
]) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe_on(pool_scheduler) \
    .subscribe(on_next=lambda s: print("PROCESS 1: {0} {1
}".format(current_thread().name, s)),
               on_error=lambda e: print(e),
              on_completed=lambda: print("PROCESS 1 done!
"))

# Create Process 2
```

```
Observable.range(1,10) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe_on(pool_scheduler) \
    .subscribe(on_next=lambda i: print("PROCESS 2: {0} {1
}".format(current_thread().name, i)), on_error=lambda e:
print(e), on_completed=lambda: print("PROCESS 2 done!"))

# Create Process 3, which is infinite
Observable.interval(1000) \
    .map(lambda i: i * 100) \
    .observe_on(pool_scheduler) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe(on_next=lambda i: print("PROCESS 3: {0} {1
}".format(current_thread().name, i)), on_error=lambda e:
print(e))

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

OUTPUT (May not match yours):

```
PROCESS 2: Thread-2 1
PROCESS 1: Thread-1 Alpha
PROCESS 1: Thread-1 Beta
PROCESS 3: Thread-4 0
PROCESS 2: Thread-2 2
PROCESS 1: Thread-1 Gamma
PROCESS 3: Thread-4 100
```

```
PROCESS 1: Thread-1 Delta
PROCESS 2: Thread-2 3
PROCESS 3: Thread-6 200
PROCESS 1: Thread-1 Epsilon
PROCESS 1 done!
PROCESS 3: Thread-13 300
PROCESS 2: Thread-2 4
PROCESS 3: Thread-15 400
PROCESS 2: Thread-2 5
PROCESS 3: Thread-4 500
PROCESS 2: Thread-2 6
PROCESS 3: Thread-4 600
PROCESS 2: Thread-2 7
PROCESS 3: Thread-4 700
PROCESS 2: Thread-2 8
PROCESS 3: Thread-4 800
PROCESS 2: Thread-2 9
PROCESS 3: Thread-4 900
PROCESS 3: Thread-4 1000
PROCESS 2: Thread-2 10
PROCESS 2 done!
PROCESS 3: Thread-4 1100
PROCESS 3: Thread-4 1200
PROCESS 3: Thread-4 1300
PROCESS 3: Thread-4 1400
...
```

Unlike `subscribe_on()`, the `observe_on()` may use a different thread for each emission rather than reserving one thread for all emissions. You can use as many `observe_on()` calls as you like in an `Observable` chain to redirect emissions to different thread pools at different points in the chain. But you can only have one `subscribe_on()`.

> *You can use the `do_action()` to essentially put Subscribers in the middle of the Observable chain, often for debugging purposes. This can be helpful to print the current thread at different points in the `Observable` chain. Refer to the Appendix to learn more.*

9.3 - Parallelization

An `Observable` will only process one item at a time. However, we can use a `subscribe_on()` or an `observe_on()` in a `flat_map()` and do multiple operations in parallel *within* that `flat_map()`.

For instance, say I have 10 Strings I need to process. Because our `intense_calculation()` will take 0.5 to 2.0 seconds to process each emission, this could take up to 20 seconds.

```
from rx import Observable
from rx.concurrency import ThreadPoolScheduler
from threading import current_thread
import multiprocessing, time, random


```

```
def intense_calculation(value):
    # sleep for a random short duration between 0.5 to 2.
0 seconds to simulate a long-running calculation
    time.sleep(random.randint(5,20) * .1)
    return value

# Create Parallel Process
Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
,"Zeta","Eta","Theta","Iota","Kappa"]) \
    .map(lambda s: intense_calculation(s)) \
    .subscribe(on_next=lambda s: print("{0} {1}".format(c
urrent_thread().name, s)),
               on_error=lambda e: print(e),
               on_completed=lambda: print("PROCESS 1 done
!"))


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

This would go much faster if we processed multiple emissions at a time rather than one at a time. Let's set

My computer has 8 cores, but let's use Python to count the number of cores dynamically. Let's set a `ThreadPoolScheduler` to have that many threads (plus one) according to our rough optimal formula. Rather than process 1 item at a time, I can now process 9 at a time which will yield a much faster completion. I just need to make sure the

expensive operators happen within a `flat_map()`, starting with that single emission wrapped in an `Observable.just()` and scheduled using `subscribe_on()`.

```
from rx import Observable
from rx.concurrency import ThreadPoolScheduler
from threading import current_thread
import multiprocessing, time, random

def intense_calculation(value):
    # sleep for a random short duration between 0.5 to 2.
0 seconds to simulate a long-running calculation
    time.sleep(random.randint(5,20) * .1)
    return value

# calculate number of CPU's and add 1, then create a Thre
adPoolScheduler with that number of threads
optimal_thread_count = multiprocessing.cpu_count() + 1
pool_scheduler = ThreadPoolScheduler(optimal_thread_count
)

# Create Parallel Process
Observable.from_(["Alpha","Beta","Gamma","Delta","Epsilon"
,"Zeta","Eta","Theta","Iota","Kappa"]) \
    .flat_map(lambda s:
        Observable.just(s).subscribe_on(pool_scheduler).m
ap(lambda s: intense_calculation(s))
    ) \
```

```
    .subscribe(on_next=lambda i: print("{0} {1}".format(c
urrent_thread().name, i)),
               on_error=lambda e: print(e),
               on_completed=lambda: print("PROCESS 1 done
!"))


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

OUTPUT:

```
Press any key to exit
Thread-4 Delta
Thread-6 Zeta
Thread-1 Alpha
Thread-2 Beta
Thread-9 Iota
Thread-3 Gamma
Thread-8 Theta
Thread-4 Kappa
Thread-7 Eta
Thread-5 Epsilon
PROCESS done!
```

Now this takes less than 3 seconds! Of course the 10 items are now racing each other and complete in a random order. Only 9 threads are

available, thus a 10th item must wait for one of the first 9 to complete. It looks like this item was `Kappa` which received `Thread-4` from `Delta` after it was done.

Parallelization using `flat_map()` (or `merge_all()`) can greatly increase performance if each emission must go through an expensive operation. Just wrap that emission into an `Observable.just()`, schedule it with `subscribe_on()` or `observe_on()` (preferably `subscribe_on()` if possible), and then make all the expensive operations happen inside the `flat_map()`.

The reason each emission must be broken into its own `Observable` is because an `Observable` is sequential and cannot be parallelized. But you can take multiple Observables and merge them into a single Observable, even if they are working on a different threads. The merged Observable will only push out items on one thread, but the items inside `flat_map()` can process in parallel.

9.4 - Redirecting Work with `switch_map()`

Imagine you have an `Observable` and you use `flat_map()` to yield a emissions from another `Observable`. However, say you wanted to *only* puruse the `Observable` for the latest emission, and kill any previous Observables to stop their emissions coming out of `flat_map()`.

You can achieve this with `switch_map()`. It operates much like a

`flat_map()`, but will only fire items for the latest emission. All previous Observables derived from previous emissions will be unsubscribed.

This example is slightly contrived, but let's say we have a finite `Observable` emitting Strings. We want an `Observable.interval()` to emit every 6 seconds, and have each emission flat map to our `Observable` of strings which are artificially slowed by `intense_calculation()`. But instead of using `flat_map()`, we can use `switch_map()` to only chase after the latest `Observable` created off each interval emission and unsubscribe previous ones.

We also need to parallelize using `subscribe_on()` so each Observable within the `switch_map()` happens on a different thread.

```
from rx import Observable
from rx.concurrency import ThreadPoolScheduler
from threading import current_thread
import multiprocessing, time, random


def intense_calculation(value):
    # sleep for a random short duration between 0.5 to 2.
0 seconds to simulate a long-running calculation
    time.sleep(random.randint(5, 20) * .1)
    return value

# calculate number of CPU's and add 1, then create a Thre
```

```
adPoolScheduler with that number of threads
optimal_thread_count = multiprocessing.cpu_count() + 1
pool_scheduler = ThreadPoolScheduler(optimal_thread_count
)

strings = Observable.from_(["Alpha", "Beta", "Gamma", "De
lta", "Epsilon", "Zeta", "Eta", "Theta", "Iota", "Kappa"]
)

Observable.interval(6000) \
    .switch_map(lambda i: strings.map(lambda s: intense_c
alculation(s)).subscribe_on(pool_scheduler)) \
    .subscribe(on_next = lambda s: print("Received {0} on
 {1}".format(s, current_thread().name)),
               on_error = lambda e: print(e))


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

OUTPUT (May Vary):

```
Press any key to exit
Received Alpha on Thread-2
Received Beta on Thread-2
Received Gamma on Thread-2
Received Delta on Thread-2
Received Alpha on Thread-4
```

```
Received Beta on Thread-4
Received Gamma on Thread-4
Received Alpha on Thread-6
Received Beta on Thread-6
Received Gamma on Thread-6
Received Delta on Thread-6
Received Epsilon on Thread-6
Received Alpha on Thread-2
...
```

Using `switch_map()` is a convenient way to cancel current work when new work comes in, rather than queuing up work. This is desirable if you are only concerned with the latest data or want to cancel obsolete processing. If you are scraping web data on a schedule using `Observable.interval()`, but a scrape instance takes too long and a new scrape requests comes in, you can cancel that scrape and start the next one.

Appendix

1 - Deferred Observables

A behavior to be aware of with `Observable.from_()` and other functions that create Observables is they may not reflect changes that happen to their sources, such as a `List`.

If we build an `Observable` off a `List`, `subscribe` to it, add “Delta”,

then `subscribe()` again, we will not see that “Delta” item emitted.

```
from rx import Observable

items = ["Alpha", "Beta", "Gamma"]
source = Observable.from_(items)

source.subscribe(lambda s: print(s))

print("\nAdding Delta!\n")
items.append("Delta")

source.subscribe(lambda s: print(s))
```

OUTPUT:

```
Alpha
Beta
Gamma

Adding Delta!

Alpha
Beta
Gamma
Delta
```

Using `Observable.defer()` allows you to create a new `Observable` from scratch each time it is subscribed, and therefore capturing anything that might have changed about its source.

```
from rx import Observable

items = ["Alpha", "Beta", "Gamma"]
source = Observable.defer(lambda: Observable.from_(items)
)

source.subscribe(lambda s: print(s))

print("\nAdding Delta!\n")
items.append("Delta")

source.subscribe(lambda s: print(s))
```

OUTPUT:

```
Alpha
Beta
Gamma

Adding Delta!

Alpha
Beta
```

```
Gamma
Delta
```

The lambda argument ensures the `Observable` source declaration is rebuilt each time it is subscribed to. This is especially helpful to use with data sources that can only be iterated once, as opposed to calling a helper function for each Subscriber (this was covered in Section VII):

```
def get_all_customers():
    stmt = text("SELECT * FROM CUSTOMER")
    return Observable.from_(conn.execute(stmt))
```

We can actually create an `Obserable` that is truly reusable for multiple Subscribers.

```
stmt = text("SELECT * FROM CUSTOMER")

# Will suppport multiple subscribers and coldly replay to
 each one
all_customers =  Observable.defer(lambda: Observable.from
_(conn.execute(stmt)))
```

2 - Debugging with `do_action()`

A helpful operator that provides insight into any point in the

`Observable` chain is the `do_action()`. This essentially allows us to insert a `Subscriber` after any operator we want, and pass one or more of `on_next()`, `on_completed()`, and `on_error()` actions.

```
from rx import Observable

Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Epsilon"]) \
    .map(lambda s: len(s)) \
    .do_action(on_next=lambda i: print("Receiving {0} from map()".format(i)),
               on_completed=lambda: print("map() is done!")) \
    .to_list() \
    .subscribe(on_next=lambda l: print("Subscriber received {0}".format(l)),
               on_completed=lambda: print("Subscriber done!"))
```

OUTPUT:

```
Receiving 5 from map()
Receiving 4 from map()
Receiving 5 from map()
Receiving 5 from map()
Receiving 7 from map()
```

```
map() is done!
Subscriber received [5, 4, 5, 5, 7]
Subscriber done!
```

Above, we declare a `do_action` right after the `map()` operation emitting the lengths. We print each length emission before it goes to the `to_list()`. Finally, `on_completed` is called and prints a notification that `map()` is not giving any more items. Then it pushes the completion event to the `to_list()` which then pushes the `List` to the `Subscriber`. Then `to_list()` calls `on_completed()` up to the `Subscriber` *after* the `List` is emitted.

Use `do_action()` when you need to "peek" inside any point in the `Observable` chain, either for debugging or quickly call actions at that point.

3 - Subjects

Another way to create an `Observable` is by declaring a `Subject`. A `Subject` is both an `Observable` and `Observer`, and you can call its `Observer` functions to push items through it and up to any Subscribers at any time.

```
from rx.subjects import Subject

subject = Subject()

```

```
subject.filter(lambda i: i < 100) \
    .map(lambda i: i * 1000) \
    .subscribe(lambda i: print(i))

subject.on_next(10)
subject.on_next(50)
subject.on_next(105)
subject.on_next(87)

subject.on_completed()
```

OUTPUT:

```
10000
50000
87000
```

While they seem convenient, Subjects are often discouraged from being used. They can easily encourage antipatterns and are prone to abuse. They also are difficult to compose against and do not respect `subscribe_on()`. It is better to create Observables that strictly come from one defined source, rather than be openly mutable and have anything push items to it at anytime. Use Subjects with discretion.

4. Error Recovery

There are a number of error recovery operators, but we will cover two

helpful ones. Say you have an `Observable` operation that will ultimately attempt to divide by zero and therefore throw an error.

```
from rx import Observable

Observable.from_([5, 6, 2, 0, 1, 35]) \
    .map(lambda i: 5 / i) \
    .subscribe(on_next=lambda i: print(i), on_error=lambd
a e: print(e))
```

OUTPUT:

```
1.0
0.8333333333333334
2.5
division by zero
```

There are multiple ways to handle this. Of course, the best way is to be proactive and use `filter()` to hold back any `0` value emissions. But for the sake of example, let's say we did not expect this error and we want a way to handle any errors we have not considered.

One way is to use `on_error_resume_next()` which will switch to an alternate `Observable` source in the event there is an error. This is somewhat contrived, but if we encounter an error we can switch to emitting an `Observable.range()`.

```
from rx import Observable

Observable.from_([5, 6, 2, 0, 1, 35]) \
    .map(lambda i: 5 / i) \
    .on_error_resume_next(Observable.range(1,10)) \
    .subscribe(on_next=lambda i: print(i), on_error=lambd
a e: print(e))
```

OUTPUT:

```
1.0
0.8333333333333334
2.5
1
2
3
4
5
6
7
8
9
10
```

It probably would be more realistic to pass an `Observable.empty()` instead to simply stop emissions once an error happens.

```
from rx import Observable

Observable.from_([5, 6, 2, 0, 1, 35]) \
    .map(lambda i: 5 / i) \
    .on_error_resume_next(Observable.empty()) \
    .subscribe(on_next=lambda i: print(i), on_error=lambd
a e: print(e))
```

OUTPUT:

```
1.0
0.8333333333333334
2.5
```

Although this is not a good example to use it, you can also use `retry()` to re-attempt subscribing to the `Observable` and hope the next set of emissions are successful without error. You typically should pass an integer argument to specify the number of retry attempts before it gives up and lets the error go to the `Subscriber`. If you do not, it will retry an infinite number of times.

```
from rx import Observable

Observable.from_([5, 6, 2, 0, 1, 35]) \
    .map(lambda i: 5 / i) \
    .retry(3) \
    .subscribe(on_next=lambda i: print(i), on_error=lambd
```

```
a e: print(e))
```

OUTPUT:

```
1.0
0.8333333333333334
2.5
1.0
0.8333333333333334
2.5
1.0
0.8333333333333334
2.5
division by zero
```

You can also use this in combination with the `delay()` operator to hold off subscribing for a fixed time period, which can be helpful for intermittent connectivity problems.

5. combine_latest()

There is one operation for merging multiple Observables together we did not cover: `combine_latest()`. It behaves much like `zip()` but will only combine the *latest* emissions for each source in the event one of them emits something. This is helpful for hot event sources especially, such as user inputs in a UI, where do you not care what the previous emissions are.

Below, we have two interval sources put in `combine_latest()`: `source1` emitting every 3 seconds and `source2` every 1 second. Notice that `source2` is going to emit a lot faster, but rather than get queued up like in `zip()` waiting for an emission from `source1`, it is going to pair with only the latest emission from `source1`. It is not going to wait for any emission to be zipped with. Conversely, when `source1` does emit something it is going to pair with the latest emission from `source2`, not wait for an emission.

```
from rx import Observable

source1 = Observable.interval(3000).map(lambda i: "SOURCE
 1: {0}".format(i))
source2 = Observable.interval(1000).map(lambda i: "SOURCE
 2: {0}".format(i))

Observable.combine_latest(source1, source2, lambda s1,s2:
 "{0}, {1}".format(s1,s2)) \
    .subscribe(lambda s: print(s))

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

OUTPUT:

```
Press any key to quit
SOURCE 1: 0, SOURCE 2: 1
SOURCE 1: 0, SOURCE 2: 2
```

```
SOURCE 1: 0, SOURCE 2: 3
SOURCE 1: 0, SOURCE 2: 4
SOURCE 1: 1, SOURCE 2: 4
SOURCE 1: 1, SOURCE 2: 5
SOURCE 1: 1, SOURCE 2: 6
SOURCE 1: 1, SOURCE 2: 7
SOURCE 1: 2, SOURCE 2: 7
SOURCE 1: 2, SOURCE 2: 8
SOURCE 1: 2, SOURCE 2: 9
SOURCE 1: 2, SOURCE 2: 10
SOURCE 1: 3, SOURCE 2: 10
SOURCE 1: 3, SOURCE 2: 11
SOURCE 1: 3, SOURCE 2: 12
SOURCE 1: 3, SOURCE 2: 13
```

Again, this is a helpful alternative for `zip()` if you want to emit the *latest combinations* from two or more Observables.