1

Content

2

Introduction to Reactive Programming

> A programming paradigm for asynchronous data streams.
> Key Features
  – Asynchronous data flow
  – Propagation of changes
  – Event-driven architecture

Why Use Reactive Programming?

> Scalability: Handles high data volumes
> Simplicity: Simplifies complex event handling
> Flexibility: Adapts well to dynamic environments
> Examples: Real-time updates, UIs, and asynchronous APIs

What is RxPY?

> A Python library for Reactive Extensions (Rx)
  – Handles asynchronous and event-based programming.
  – Inspired by RxJS

5

Features of RxPY

> Works with Observables and Observers
> Provides operators for transforming data streams
> Supports event-driven and functional programming styles
> Combines different data sources

6

RxPy

CORE PROGRAMMING FEATURES

Importing the Rx module

```
import reactivex
from reactivex import operators as ops
from reactivex import Observer
```

Generating a sequence

- There are many ways to generate a sequence of events.
- The easiest way to get started is to use the **`from_iterable()`** operator that is also called just **`from_`**.
- Other operators you may use to generate a sequence
  - **`just`**, **`generate`**, **`create`** and **`range`**.

Generating a sequence

```
class MyObserver(Observer[int]):
    def on_next(self, value: int):
        print("Got: %s" % value)

    def on_error(self, error: Exception):
        print("Got error: %s" % error)

    def on_completed(self):
        print("Sequence completed")

xs = reactivex.from_iterable(range(10))
d = xs.subscribe(MyObserver())
```

Filtering a sequence

```
import reactivex
from reactivex import operators as ops
from reactivex import Observer

xs = reactivex.from_(range(10))
d = xs.pipe(
    ops.filter(
        lambda x: x % 2
    )).subscribe(print)
```


Transforming a sequence

```
import reactivex
from reactivex import operators as ops
from reactivex import Observer

xs = reactivex.from_(range(10))
d = xs.pipe(
    ops.map(
        lambda x: x * 2
    )).subscribe(print)
```

Transforming a sequence

```
import reactivex
from reactivex import operators as ops
from reactivex import Observer

xs = reactivex.from_(range(10, 20, 2))
d = xs.pipe(
    ops.map_indexed(
        lambda x, i: "%s: %s" % (i, x * 2)
    )).subscribe(print)
```


Merge

```
import reactivex
from reactivex import operators as ops
from reactivex import Observer

xs = reactivex.range(1, 5)
ys = reactivex.from_("abcde")
zs = xs.pipe(ops.merge(ys)).subscribe(print)
```

The Spacetime of Rx

> The events are only separated by ordering.
> This confuses many newcomers to Rx since the result of the merge operation above may have several valid results such as:
  - a1b2c3d4e5
  - 1a2b3c4d5e
  - ab12cd34e5
  - abcde12345

The Spacetime of Rx

> The only guarantee you have is that **1** will be before **2** in **xs**, but **1** in **xs** can be before or after **a** in **ys**.
> It's up the the sort stability of the scheduler to decide which event should go first.
> For real time data streams this will not be a problem since the events will be separated by actual time.
> To make sure you get the results you "expect", it's always a good idea to add some time between the events when playing with Rx.

Marbles and Marble Diagrams

```
import reactivex
from reactivex import operators as ops
from reactivex import Observer

xs = reactivex.from_marbles("a-b-c-|")
my_list = xs.pipe(ops.to_list()).run()
print(my_list)
```


Subjects and Streams

> A simple way to create an observable stream is to use a subject

> a Subject is both an Observable and an Observer, so you can both subscribe to it and on_next it with events.

– publish values into an observable stream for processing

Subjects and Streams

```
from reactivex.subject import Subject

stream = Subject[int]()
stream.on_next(41)

d = stream.subscribe(lambda x: print("Got: %s" % x))

stream.on_next(42)

d.dispose()
stream.on_next(43)
```


Core Concepts in RxPY

- **Observables**: Represents a data stream
- **Observers**: Subscribes to Observables and reacts to emitted data
- **Operators**: Transform data streams (e.g., map, filter)
- **Schedulers**: Manage concurrency

Anatomy of an Observable

```
import reactivex as rx

# Create an Observable
def observable_example(observer, scheduler):
    observer.on_next("Hello")
    observer.on_next("World")
    observer.on_completed()

source = rx.create(observable_example)
source.subscribe(print)
```


Creating Observables

- From Iterables: **`from_iterable()`**
- From Events: **`from_callback()`**
- Using Create: **`Observable.create()`**

Observables Lifecycle

1. **Creation**: Define the data stream
2. **Subscription**: Observer subscribes to receive data
3. **Emission**: Observable emits data
4. **Completion/Termination**: Ends the stream

What Are Observers?

> Definition: Consumers of data emitted by Observables.

> Implement these methods:

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

Example: Observer

```
import reactivex as rx

def push_data(observer, scheduler):
    observer.on_next(1)
    observer.on_next(2)
    observer.on_completed()

source = rx.create(push_data)
source.subscribe(
    on_next=lambda x: print(f"Received: {x}"),
    on_error=lambda e: print(f"Error: {e}"),
    on_completed=lambda: print("Done")
)
```


Operators in RxPY

- Transform and manipulate streams
- Categories:
  - Filtering: **`filter()`**, **`distinct()`**
  - Transformation: **`map()`**, **`flat_map()`**
  - Combining: **`merge()`**, **`zip()`**

Example: Filtering Data

```
import reactivex as rx
from reactivex import operators as ops

source = rx.from_iterable(range(10))
source.pipe(
   ops.filter(lambda x: x % 2 == 0)
).subscribe(print)
```


Schedulers

- Manage threading and concurrency.
- Types
  - ImmediateScheduler
  - NewThreadScheduler
  - ThreadPoolScheduler

Asyncio Integration

> Manage threading and concurrency.
> Types
  – ImmediateScheduler
  – NewThreadScheduler
  – ThreadPoolScheduler

Asyncio Integration

> Asyncio
  – Handles asynchronous I/O-bound tasks using an event loop.
> RxPY
  – Provides tools to work with asynchronous and event-driven systems using Observables, which can emit and process sequences of data over time.

Key Components of the Integration

> RxPY has built-in support for asyncio
  – Convert Observables to asyncio constructs (like Futures or Tasks).
  – Convert asyncio coroutines or Futures into Observables.
  – Schedule RxPY Observables on asyncio's event loop.

Key Tools

> **`AsyncioScheduler`**
  – Schedules RxPY Observables to run on an asyncio event loop
> **`from_future`**
  – Converts an **`asyncio.Future`** or coroutine into an Observable
> **`to_future`**
  – Converts an Observable into an **`asyncio.Future`**

RxPy

WEBSOCKET: COMMUNICATING BETWEEN TWO OBSERVERS

Example

```
import asyncio
import websockets
import json
from rx import operators as ops
from rx.scheduler.eventloop import AsyncIOScheduler
from rx.subject import Subject
```

Example

```
def process_message(msg):
    """Process the received message from Binance."""
    try:
        data = json.loads(msg)
        if "data" in data and "p" in data["data"]:
            return float(data["data"]["p"])
    except Exception as e:
        print(f"Error processing message: {e}")
    return None
```


Example

```
async def main():
    trade_pairs = ["btcusdt@trade", "ethusdt@trade"]

    # RxPy Subject to publish WebSocket messages
    subject = Subject()

    # Asynchronous WebSocket stream
    async def websocket_task():
        async for message in binance_ws_stream(trade_pairs):
            subject.on_next(message)

    # Reactive stream processing
    scheduler = AsyncIOScheduler(asyncio.get_event_loop())
```

```
    subject.pipe(
        ops.map(process_message),            # Map: Extract price
        ops.filter(lambda x: x is not None), # Filter: Valid messages
        ops.buffer_with_time(30, scheduler=scheduler),
        ops.map(lambda prices: {
            "count": len(prices),
            "average": sum(prices) / len(prices) if prices else None,
            "min": min(prices) if prices else None,
            "max": max(prices) if prices else None,
        }) # Map: Aggregate results
    ).subscribe(
        lambda stats: print(f"Stats in 30s window: {stats}"),
        lambda e: print(f"Error: {e}"),
        lambda: print("Stream completed.")
    )
```


Example

```
    # Run WebSocket task
    await websocket_task()

if __name__ == "__main__":
    asyncio.run(main())
```

RxPy

WEBSOCKET AND RABBITMQ

Example

```
import asyncio
import websockets
import json
import pika
from rx import operators as ops
from rx.subject import Subject
from rx.scheduler.eventloop import AsyncIOScheduler
from datetime import datetime
```

Example

```
BINANCE_WS_URL = "wss://stream.binance.com:9443/ws/btcusdt@trade"
RABBITMQ_QUEUE = "filtered_trades"

# RabbitMQ Connection Setup
connection =
pika.BlockingConnection(pika.ConnectionParameters("localhost"))
channel = connection.channel()
channel.queue_declare(queue=RABBITMQ_QUEUE)

# Reactive Stream Setup
subject = Subject()
```


Example

```
async def binance_websocket():
    """Connect to Binance WebSocket and emit data to the subject."""
    async with websockets.connect(BINANCE_WS_URL) as websocket:
        async for message in websocket:
            trade = json.loads(message)
            subject.on_next(trade)
```

Example

```
def process_trades():
    """Filter, map, reduce and send results to RabbitMQ."""
    scheduler = AsyncIOScheduler(asyncio.get_event_loop())  # Attach
scheduler to event loop

    subject.pipe(
        ops.filter(lambda trade: float(trade['p']) > 30000),  #
Example filter: price > 30,000
        ops.map(lambda trade: {
            "symbol": trade["s"],
            "price": float(trade["p"]),
            "quantity": float(trade["q"]),
            "timestamp": trade["T"]
        }),
        ops.buffer_with_time(30.0, scheduler=scheduler),  # 30-second
```


Example

```
def process_trades():
    # Attach scheduler to event loop
    scheduler = AsyncIOScheduler(asyncio.get_event_loop())

    subject.pipe(
        ops.filter(lambda trade: float(trade['p']) > 50000),
        ops.map(lambda trade: {
            "symbol": trade["s"],
            "price": float(trade["p"]),
            "quantity": float(trade["q"]),
            "timestamp": trade["T"]
        }),
        # 30-second time window
        ops.buffer_with_time(30.0, scheduler=scheduler),
```

Example

```
    ops.map(lambda trades: {
         "average_price": sum(t["price"] * t["quantity"] for t in
trades) / sum(t["quantity"] for t in trades) if trades else None,
         "total_quantity": sum(t["quantity"] for t in trades),
         "window_start": trades[0]["timestamp"] if trades else None,
         "window_end": trades[-1]["timestamp"] if trades else None,
        }),
    ).subscribe(
        on_next=lambda result: send_to_rabbitmq(result),
        on_error=lambda e: print(f"Error: {e}"),
        on_completed=lambda: print("Processing completed."),
    )
```


Example

```
def send_to_rabbitmq(result):
    """Send processed trade data to RabbitMQ."""
    if result["average_price"] is not None:
        channel.basic_publish(
            exchange="",
            routing_key=RABBITMQ_QUEUE,
            body=json.dumps(result)
        )
        print(f"Sent to RabbitMQ: {result}")
```

Example

```
async def main():
    # Start WebSocket connection
    websocket_task = asyncio.create_task(binance_websocket())

    # Start processing trades
    process_trades()

    # Run WebSocket task indefinitely
    await websocket_task

asyncio.run(main())
```


Example

```
async def main():
    """Main event loop."""
    # Start WebSocket connection
    websocket_task = asyncio.create_task(binance_websocket())

    # Start processing trades
    process_trades()

    # Run WebSocket task indefinitely
    await websocket_task


if __name__ == "__main__":
    try:
```