ThingFlow-Python Documentation

Release 2.3.0

MPI-SWS and Data-ken Research

Aug 08, 2017

Contents

ThingFlow is a (Python3) framework for building IoT event processing dataflows.[1] The goal of this framework is to support the creation of robust IoT systems from reusable components. These systems must account for noisy/missing sensor data, distributed computation, and the need for local (near the data source) processing.

The source repository for ThingFlow-python is at https://github.com/mpi-sws-rse/thingflow-python.

Sections 1 and 2 of this documentation cover how to get started. Sections 3 through 5 cover more advanced topics. Section 6 provides some more code examples. Section 7 covers our port of ThingFlow to MicroPython on the ESP8266. Section 8 documents some design decisions made during the evolution of ThingFlow. Finally, Section 9 contains reference documentation for the full ThingFlow-python API (extracted from the docstrings).

Contents:

---

[1] *ThingFlow* was originally known as *AntEvents*.

1. Introduction

The fundamental abstractions in ThingFlow are 1) *sensors*, which provide a means to sample a changing value representing some quanity in the physical world, 2) *event streams*, which are push-based sequences of sensor data readings, and 3) *things*, which are reusable components to generate, transform, or consume the events on these streams. Things can have simple, stateless logic (e.g. filter events based on a predicate) or implement more complex, stateful algorithms, such as Kalman filters or machine learning. Using ThingFlow, you describe the flow of data through these things rather than programming low-level behaviors.

Although ThingFlow presents a simple dataflow model to the user, internally it uses an event-driven programming model, building on Python's `asyncio` module. In addition to being a natural programming model for realtime sensor data, it reduces the potential resource consumption of Ant Events programs. The details of event scheduling are handled by the framework. Separate threads may be used on the "edges" of a dataflow, where elements frequently interact with external components that have blocking APIs.

ThingFlow integrates with standard Python data analytics frameworks, including NumPy, Pandas, and scikit-learn. This allows dataflows involving complex elements to be developed and refined offline and then deployed in an IoT environment using the same code base.

We call the implementation described here "ThingFlow-Python", as it should be possible to port the ideas of ThingFlow to other languages. Currently, one such port exists: "ThingFlow-MicroPython". This is a port of ThingFlow to MicroPython, a limited version of Python 3 that runs "bare metal" on embadded devices. The ThingFlow-MicroPython port is included in the ThingFlow-Python repository under the subdirectory `micropython`. It is documented in *Section 7* of this document.

Example

To give the flavor of ThingFlow, below is a short code snippet for the Raspberry Pi that reads a light sensor and then turns on an LED if the running average of the last five readings is greater than some threshold:

```python
lux = SensorAsOutputThing(LuxSensor())
lux.map(lambda e: e.val).running_avg(5).map(lambda v: v > THRESHOLD)\
    .GpioPinOut()
```

```
scheduler.schedule_periodic(lux, 60.0)
scheduler.run_forever()
```

The first line instantiates a light sensor object and wraps it in an *output thing* to handle sampling and progagation of events.

The next two lines create a pipeline of things to process the data from the sensor. We call things which have a single input and output *filters*, as they can be composed to process a stream of events. The `map` filter extracts the data value from the sensor event, the `running_avg` filter averages the last five values, and the next `map` filter converts the value to a a boolean based on the threshold. The `GpioPinOut` thing is an *adapter* to the outside world. It turns on the LED based on the value of its input boolean value.

Finally, the last two lines of the example schedule the sensor to be sampled at a sixty second interval and then start the scheduler's main loop.

Platforms

ThingFlow does not have any required external dependendencies, so, in theory at least, it can be run just about anywhere you can run Python 3. It has been tested on the Raspberry Pi (Rasbian distribution), Desktop Linux, and MacOSX. In a desktop environment, you might find the Anaconda Python distribution helpful, as it comes with many data analytics tools (e.g. Jupyter, NumPy, Pandas, and scikit-learn) pre-installed.

ThingFlow has been ported to Micropython, so that it can run on very small devices, like the ESP8266. Since these devices have stringent memory requirements, the code base has been stripped down to a core for the Micropython port. The port is in this repository under the `micropython` directory.

Installing ThingFlow

We recommend installing into a `virtualenv` rather than directly into the system's Python. To do so, first run the `activate` script of your chosen virtual environment. Next, you can either install from the Python Package Index website (pypi.python.org) or from the ThingFlow source tree.

Installing from the Python Package Index

The package name of thingflow-python on PyPi is `thingflow`. You can use the `pip` utility to install, as follows:

```
pip3 install thingflow
```

If you have activated your virtual environment, this should pick up the version of `pip3` associated with your environment, and install ThingFlow into your environment rather than into the system's Python install.

Installing from the source tree

Go to the `thingflow-python` directory and then run:

```
python3 setup.py install
```

If you have activated your virtual environment, this should pick up the version of `python3` associated with your environment, and install ThingFlow into your environment rather than into the system's Python install.

Using ThingFlow without installation

You can also run the ThingFlow code in-place from the git repository by adding the full path to the `thingflow-python` directory to your `PYTHONPATH`. This is how the tests and the examples are run.

Directory Layout

The layout of the files in the ThingFlow code repository (the `thingflow-python` directory) is as follows:

- `README.RST` - a short introduction and pointer to resources
- `Makefile` - builds the source distribution and documentation; can run the tests
- `setup.py` - used to install the core code into a python environment
- `thingflow/` - the core code. This is all that will get installed in a production system
  - `thingflow/base.py` - the core definitions and base classes of thingflow
  - `thingflow/adapters` - reader and writer things that talk to the outside world
  - `thingflow/filters` - elements for filter pipelines, in the style of Microsoft's Linq framework
- `docs/` - this documentation, build using Sphinx
- `tests/` - the tests. These can be run in-place.
- `examples/` - examples and other documentation.
- `micropython/` - port of the ThingFlow core to MicroPython

More Examples

Additional can be found throughout this document, particularly in *Section 6: More Examples*. There is also a separate repository with larger ThingFlow applications. It is at https://github.com/mpi-sws-rse/thingflow-examples. This repository includes an automated lighting application and a vehicle traffic analysis.

Next, we will go into more detail on ThingFlow with a *tutorial*.

2. Tutorial

To understand the core concepts in ThingFlow, let us build a simple app with a dummy sensor that generates random data and feeds it to a dummy LED. The final code for this example is at `thingflow-python/examples/tutorial.py`.

Input Things and Output Things

Each ThingFlow "thing" is either an *output thing*, which emits events and and puts the into the workflow, an *input thing*, which consumes events, accepting event streams from the workflow, or both.

An output thing may create multiple output event streams. Each output stream is associated with a named *output port*. Likewise, an input thing may accept input streams via named *input ports*. Input and output ports form the basis for interconnections in our data flows.

In general, we can connect an input port to an output port via an output thing's `connect()` method like this:

```
output_thing.connect(input_thing,
                     port_mapping=('output_port_name', 'input_port_name'))
```

There also exists a special *default* port, which is used when no port name is specified on a connection. If you leave off the port mapping parameter in the `connect()` call, it maps the default port of the output to the default port of the input:

```
output_thing.connect(input_thing)
```

Once connected through the `connect` call, a output and input thing interact through three methods on the input thing:

- `on_next`, which passes the next event in the stream to the input thing.

- `on_error`, which should be called at most once, if a fatal error occurs. The exception that caused the error is passed as the parameter.

- `on_completed`, which signals the end of the stream and takes no parameters.

Note that each output port may have multiple connections. The functionality in the `thingflow.base.OutputThing` base class handles dispatching the events to all downstream consumers.

More terms for specialized things

We call things which have a default input port and a default output port *filters*. Filters can be easily composed into pipelines. We talk more about filters *below*. A number of filters are defined by ThingFlow under the module `thingflow.filters`.

Some things interface to outside world, connecting ThingFlow to transports and data stores like MQTT, PostgreSQL, and flat CSV files. We call these things *adapters*. Several may be found under `thingflow.adapters`. We call an output thing that emits events coming from an outside source a *reader*. An input thing which accepts event and conveys them to an outside system a *writer*.

Sensors

Since ThingFlow is designed for Internet of Things applications, data capture from sensors is an important part of most applications. To this end, ThingFlow provides a *sensor* abstraction. A sensor is any python class that implements a `sample()` method and has a `sensor_id` property. The `sample()` method takes no arguments and returns the current value of the sensor. The `sensor_id` property is used to identify the sensor in downstream events. Optionally, a sensor can indicate that there is no more data available by thowing a `StopIteration` exception.

To plug sensors into the world of input and output things, ThingFlow provides the `SensorAsOutputThing` class. This class wraps any sensor, creating an output thing. When the thing is called by the scheduler, it calls the sensor's `sample()` method, wraps the value in an event (either `SensorEvent` or a custom event type), and pushes it to any connected input things. We will see `SensorAsOutputThing` in action below.

There are cases where this simple sensor abstraction is not sufficient to model a real-life sensor or you are outputting events that are not coming directly from a sensor (e.g. from a file or a message broker). In those situations, you can just create your own output thing class, subclassing from the base `OutputThing` class.

Implementing a Sensor

Now, we will implement a simple test sensor that generates random values. There is no base sensor class in ThingFlow, we just need a class that provides a `sensor_id` property and a `sample()` method. We'll take the `sensor_id` value as an argument to `__init__()`. The sample value will be a random number generated with a Gaussian distribution, via `random.gauss`. Here is the code for a simple version of our sensor:

```python
import random
random.seed()


class RandomSensor:
    def __init__(self, sensor_id, mean, stddev):
        """Generate a random value each time sample() is
        called, using the specified mean and standard
        deviation.
        """
        self.sensor_id = sensor_id
        self.mean = mean
        self.stddev = stddev

    def sample(self):
        return random.gauss(self.mean, self.stddev)
```

```
    def __str__(self):
        return "RandomSensor(%s, %s, %s)" % \
            (self.sensor_id, self.mean, self.stddev)
```

This sensor will generate a new random value each time it is called. If we run it with a scheduler, it will run forever (at least until the program is interrupted via Control-C). For testing, it would be helpful to stop the program after a certain number of events. We can do that, by passing an event limit to the constructor, counting down the events, and throwing a `StopIteration` exception when the limit has been reached. Here is an improved version of our sensor that can signal a stop after the specified number of events:

```python
import random
random.seed()
import time
from thingflow.base import SensorAsOutputThing


class RandomSensor:
    def __init__(self, sensor_id, mean, stddev, stop_after):
        """This sensor will signal it is completed after the
        specified number of events have been sampled.
        """
        self.sensor_id = sensor_id
        self.mean = mean
        self.stddev = stddev
        self.events_left = stop_after

    def sample(self):
        if self.events_left>0:
            data = random.gauss(self.mean, self.stddev)
            self.events_left -= 1
            return data
        else:
            raise StopIteration

    def __str__(self):
        return "RandomSensor(%s, %s, %s)" % \
            (self.sensor_id, self.mean, self.stddev)
```

Now, let's instantiate our sensor:

```python
from thingflow.base import SensorAsOutputThing
MEAN = 100
STDDEV = 10
sensor = SensorAsOutputThing(RandomSensor(1, MEAN, STDDEV, stop_after=5))
```

Implementing an Input Thing

Now, let us define a simple intput thing – a dummy LED actuator. The LED will inherit from the `thingflow.base.IntputThing` class, which defines the input thing interface for receiving events on the default port. Here is the code:

```python
from thingflow.base import InputThing
class LED(InputThing):
    def on_next(self, x):
        if x:
```

```
            print("On")
        else:
            print("Off")

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

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

    def __str__(self):
        return 'LED'
```

As you can see, the main logic is in `on_next` – if the event looks like a true value, we just print "On", otherwise we print "Off". We won't do anything special for the `on_error` and `on_completed` callbacks. Now, we can instantiate an LED:

```
led = LED()
```

Filters

A *filter* is a thing that as a single default input port and a single default output port. There is a base class for filters, `thingflow.base.Filter`, which subclasses from both `InputThing` and `OutputThing`. Although you can instantiate filter classes directly, ThingFlow makes use of some Python metaprogramming to dynamically add convenience methods to the base `OutputThing` class to create and return filtes. This allows filters can be easily chained together, implementing multi-step query pipelines without any glue code.

Let us now create a series of filters that connect together our dummy light sensor and our LED. Here is some code to look at each event and send `True` to the LED if the value exceeds the mean (provided to the sensor) and `False` otherwise:

```
import thingflow.filters.map
sensor.map(lambda evt: evt.val > MEAN).connect(led)
```

The `import` statement loads the code for the `map` filter. By loading it, it is added as a method to the `OutputThing` class. Since the sensor was wrapped in `SensorAsOutputThing`, which inherits from `OutputThing`, it gets this method as well. Calling the method creates a filter which runs the supplied anonymous function on each event. This filter is automatically connected to the sensor's default output port. The `map` call returns the filter, allowing it to be used in chained method calls. In this case, we `connect` the `led` to the filter's event stream.

Inside the Map filter

It is important to note that the call to a filter method returns a filter object and not an event. This call happens at initializaiton time. To get a better understanding of what's happening, let's take a look inside the `map` filter.

First, let us create a straightfoward implementation of our filter by subclassing from the base `Filter` class and then overridding the `on_next` method:

```
from thingflow.base import Filter, filtermethod
class MapFilter(Filter):
    def __init__(self, previous_in_chain, mapfun):
        super().__init__(previous_in_chain)
        self.mapfun = mapfun
```

```
    def on_next(self, x):
        next = self.mapfun(x)
        if next is not None:
            self._dispatch_net(next)


@filtermethod(OutputThing)
def map(this, mapfun):
    return MapFilter(this, mapfun)
```

In this case, the `on_next` method applies the provided `mapfun` mapping function to each incoming event and, if the result is not `None`, passes it on to the default output port via the method `dispatch_next` (whose implementation is inherited from the base `OutputThing` class).

In the `__init__` method of our filter, we accept a `previous_in_chain` argument and pass it to the parent class's constructor. As the name implies, this argument should be the previous filter in the chain which is acting as a source of events to this filter. `Filter.__init__` will perform a `previous_in_chain.connect(self)` call to establish the connection.

We can now wrap our filter in the function `map`, which takes the previous filter in the chain and our mapping function as arguments, returning a new instance of `MapFilter`. The decorator `functionfilter` is used to attach this function to `OutputThing` as a method. We can then make calls like `thing.map(mapfun)`.

The actual code for map``in ThingFlow map be found in the module ``thingflow.filters. map. It is written slightly differently, in a more functional style:

```
from thingflow.base import OutputThing, FunctionFilter, filtermethod

@filtermethod(OutputThing, alias="select")
def map(this, mapfun):
    def on_next(self, x):
        y = mapfun(x)
        if y is not None:
            self._dispatch_next(y)
    return FunctionFilter(this, on_next, name='map')
```

The `FunctionFilter` class is a subclass of `Filter` which takes its `on_next`, `on_error`, and `on_completed` method implementations as function parameters. In this case, we define `on_next` inside of our `map` filter. This avoids the need to even create a `MapFilter` class.

Sensor Events

ThingFlow provides a *namedtuple* called `thingflow.base.SensorEvent`, to serve as elements of our data stream. The first member of the tuple, called `sensor_id` is the sensor id property of the sensor from which the event originated. The second member of the event tuple, `ts`, is a timestamp of when the event was generated. The third member, `val`, is the value returned by the sensor's `sample()` method.

The `SensorAsOutputThing` wrapper class creates `SensorEvent` instances by default. However, you can provide an optional `make_sensor_event` callback to `SensorAsOutputThing` to override this behavior and provide your own event types.

Sensor Output Example

Imagine that the sensor defined above outputs the following three events, separated by 10 seconds each:

```
SensorEvent(1, 2016-06-21T17:43:25, 95)
SensorEvent(1, 2016-06-21T17:43:35, 101)
SensorEvent(1, 2016-06-21T17:43:45, 98)
```

The `select` filter would output the following:

```
False
True
False
```

The LED would print the following:

```
Off
On
Off
```

Some Debug Output

There are a number of approaches one can take to help understand the behavior of an event dataflow. First, can add an `output` thing to various points in the flow. The `output` thing just prints each event that it see. It is another filter that can be added to the base `OutputThing` class by importing the associated Python package. For example, here is how we add it as a connection to our sensor, to print out every event the sensor emits:

```python
import thingflow.filters.output
sensor.output()
```

Note that this does not actually print anything yet, we have to run the *scheduler* to start up our dataflow and begin sampling events from the sensor.

Another useful debugging tool is the `print_downstream` method on `OutputThing`. It can be called on any subclass to see a representation of the event tree rooted at the given thing. For example, here is what we get when we call it on the `sensor` at this point:

```
***** Dump of all paths from RandomSensor(1, 100, 10) *****
  RandomSensor(1, 100, 10) => select => LED
  RandomSensor(1, 100, 10) => output
************************************
```

Finally, the `OutputThing` class also provices a `trace_downstream` method. It will instument (transitively) all downstream connections. When the scheduler runs the thing, all events passing over these connections will be printed.

The Scheduler

As you can see, it is easy to create these pipelines. However, this sequence of things will do nothing until we hook it into the main event loop. In particular, any output thing that source events into the system (e.g. sensors) must be made known to the *scheduler*. Here is an example where we take the dataflow rooted at the light sensor, tell the scheduler to sample it once every second, and then start up the event loop:

```python
import asyncio
from thingflow.base import Scheduler
scheduler = Scheduler(asyncio.get_event_loop())
scheduler.schedule_periodic(sensor, 1.0) # sample once a second
scheduler.run_forever() # will run until there are no more active sensors
print("That's all folks!") # This will never get called in the current version
```

The output will look something like this:

```
Off
SensorEvent(sensor_id=1, ts=1466554963.321487, val=91.80221483640152)
On
SensorEvent(sensor_id=1, ts=1466554964.325713, val=105.20052817504502)
Off
SensorEvent(sensor_id=1, ts=1466554965.330321, val=97.78633493089245)
Off
SensorEvent(sensor_id=1, ts=1466554966.333975, val=90.08049816341648)
Off
SensorEvent(sensor_id=1, ts=1466554967.338074, val=89.52641383841595)
On
SensorEvent(sensor_id=1, ts=1466554968.342416, val=101.35659321534875)
...
```

The scheduler calls the _observe method of SensorAsOutputThing once every second. This method samples the sensor and calls _dispatch_next to pass it to any downstream things connected to the output port. In the program output above, we are seeing the On/Off output from the LED interleaved with the original events printed by the output element we connected directly to the sensor. Note that this will keep running forever, until you use Control-C to stop the program.

Stopping the Scheduler

As you saw in the last example, the run_forever method of the scheduler will keep on calling things as long as any have been scheduled. If we are just running a test, it would be nice to stop the program automatically ather than having to Control-C out of the running program. Our sensor class addresses this by including an optional stop_after parameter on the constuctor. When we instantiate our sensor, we can pass in this additional parameter:

```python
sensor = SensorAsOutputThing(RandomSensor(1, MEAN, STDDEV, stop_after=5))
```

The scheduler's run_forever() method does not really run forever – it only runs until there are no more schedulable actions. When our sensor throws the StopIteration exception, it causes the wrapping SensorAsOutputThing to deschedule the sensor. At that point, there are no more publishers being managed by the scheduler, so it exits the loop inside run_forever().

When we run the example this time, the program stops after five samples:

```
Off
SensorEvent(sensor_id=1, ts=1466570049.852193, val=87.42239337997071)
On
SensorEvent(sensor_id=1, ts=1466570050.856118, val=114.47614678277142)
Off
SensorEvent(sensor_id=1, ts=1466570051.860044, val=90.26934530230736)
On
SensorEvent(sensor_id=1, ts=1466570052.864378, val=102.70094730226809)
On
SensorEvent(sensor_id=1, ts=1466570053.868465, val=102.65381015942252)
LED Completed
```

```
Calling unschedule hook for RandomSensor(1, 100, 10)
No more active schedules, will exit event loop
That's all folks!
```

Next Steps

You have reached the end of the tutorial. To learn more, you might:

- Continue with this documentation. In the *next section*, we look at implementing output things.

- Take a look at the code under the `examples` directory.

- You can also read through the code in the `thingflow` proper – a goal of the project is to ensure that it is clearly commented.

3. Implementing an OutputThing

In most cases, one can simply wrap a sensor in the `SensorAsOutputThing` class and not worry about the details of how to implement output things. There are also several pre-defined *readers* under `thingflow.adapters` that can obtain events from external sources like message brokers, flat files, and databases.

The most likly reason for implmenting a new OutputThing is that you want to create a new adapter type that does not exist in the standard ThingFlow library. We will walk through the details in this document.

Subclassing

When implmenting an output thing, one subclasses from `thingflow.base.OutputThing`. To emit a new event, the subclass calls the `_dispatch_next` method with the event and port name. To signal an error or completion of the event stream, one calls `_dispatch_error` or `_dispatch_completed`, respectively. The base class implementation of these methods are responsible for calling the `on_next`, `on_error`, and `on_completed` methods for each of the connected things.

The code to call these `_dispatch` methods goes into a well-known method to be called by the scheduler. The specific method depends how the output thing will interact with the scheduler. There are two cases supported by ThingFlow and three associated mixin-classes that define the methods:

1. `DirectOutputThingMixin` defines an `_observe` method that can be called directly by the scheduler either in the main thread (via `Scheduler.schedule_period()` or `Scheduler.schedule_recurring()`) or in a separate thread (via `Scheduler.schedule_periodic_on_separate_thread()`).

2. `EventLoopOutputThingMixin` is used for an output thing that has its own separate event loop. This is run in a separate thread and the connected input things are called in the main thread.

Simple CSV Reader

OK, with all that out of the way, let us define a simple OutputThing. We will create a simple CSV-formatted spreadsheet file reader. Each row in the file corresponds to an event. Here is the class definition (found in `examples/`

`simple_csv_reader.py`):

```python
import csv
from thingflow.base import OutputThing, DirectOutputThingMixin,\
                            SensorEvent, FatalError


class SimpleCsvReader(OutputThing, DirectOutputThingMixin):
    def __init__(self, filename, has_header_row=True):
        super().__init__() # Make sure the output_thing class is initialized
        self.filename = filename
        self.file = open(filename, 'r', newline='')
        self.reader = csv.reader(self.file)
        if has_header_row:
            # swallow up the header row so it is not passed as data
            try:
                self.reader.__next__()
            except Exception as e:
                raise FatalError("Problem reading header row of csv file %s: %s" %
                                 (filename, e))

    def _observe(self):
        try:
            row = self.reader.__next__()
            event = SensorEvent(ts=float(row[0]), sensor_id=row[1],
                                val=float(row[2]))
            self._dispatch_next(event)
        except StopIteration:
            self.file.close()
            self._dispatch_completed()
        except FatalError:
            self._close()
            raise
        except Exception as e:
            self.file.close()
            self._dispatch_error(e)
```

The `SimpleCsvReader` class subclasses from both `OutputThing` and `DirectOutputThingMixin`. Subclassing from `OutputThing` provides the machinery needed to register connections and propagate events to downstream input things. `DirectOutputThingMixin` defines an empty `_observe()` method and indicates that the scheduler should call `_observe()` to dispatch events whenever the reader has been scheduled.

In the `__init__()` constructor, we first make sure that the base class infrastructure is initialized through `super().__init__()`. Next, we open the file, set up the csv reader, and read the header (if needed).

The main action is happening in `_observe()`. When scheduled, it reads the next row from the csv file and creates a `SensorEvent` from it. This event is passed on to the output port's connections via `_dispatch_next()`. If the end of the file has been reached (indicated by the `StopIteration` exception), we instead call `_dispatch_completed()`. There are two error cases:

1. If a `FatalError` exception is thrown, we close our connection and propagate the error up. This will lead to an early termination of the event loop.

2. If any other exception is thrown, we pass it downstream via `_dispatch_error()`. It will also close the event stream and cause the `SimpleCsvReader` to be de-scheduled. The main event loop may continue, assuming that there are other scheduled objects.

We could save some work in implementing our reader by subclassing from `thingflow.adapters.generic.DirectReader`. It provides the dispatch behavior common to most readers.

Reading a File

Now, let us create a simple data file `test.csv`:

```
ts,id,value
1,1,2
2,1,3
3,1,455
4,1,55
```

We can instantiate a `SimpleCsvReader` to read in the file via:

```
reader = SimpleCsvReader("test.csv")
```

Now, let's hook it to an printing input thing and then run it in the event loop:

```python
import asyncio
from thingflow.base import Scheduler
import thingflow.adapters.output # load the output method

reader.output()
scheduler = Scheduler(asyncio.get_event_loop())
scheduler.schedule_recurring(reader)
scheduler.run_forever()
```

We use `schedule_recurring()` instead of `schedule_periodic()`, as we expect all the data to be already present in the file. There is no sense in taking periodic samples.

The output looks as follows:

```
SensorEvent(sensor_id='1', ts=1.0, val=2.0)
SensorEvent(sensor_id='1', ts=2.0, val=3.0)
SensorEvent(sensor_id='1', ts=3.0, val=455.0)
SensorEvent(sensor_id='1', ts=4.0, val=55.0)
No more active schedules, will exit event loop
```

Note that the event loop terminates on its own. This is due to the call to `_dispatch_completed()` when the csv reader throws `StopIteration`.

Output Things with Private Event Loops

There can be cases when the underlying API to be called by the OutputThing requires its own event loop / event listener. To handle this situation, use the interface provided by `EventLoopOutputThingMixin`. Your main event loop for the output ting is implemented in the `_observe_event_loop()`. If you call the scheduler's `schedule_on_private_event_loop()` method, it will run this method in a separate thread and then dispatch any events to the scheduler's main event loop (running in the main thread).

To see some example code demonstrating an output thing using a private event loop, see `thingflow.adapters.mqtt.MQTTReader`.

4. Things with Non-default Ports

ThingFlow provides a general dataflow architecture. Output things can output events on different ports and input things can receive messages via different ports. Each `connect()` call can rename ports, allowing the interconnection of any compatible ports. For example, one might have code like:

```
output_thing.connect(input_thing,
                     port_mapping=('out_port_name', 'in_port_name'))
```

As you know, ThingFlow provides a special `default` port that does not need any mapping. This makes it convenient for building chains of filters and is good enough most of the time. However, when you need a more complex data flow, the more general mapping capability can be very helpful. We will now look at it in more detail.

Multiple Output Ports

To create an output thing which sends messaages on multiple output ports, one subclasses from `OutputThing` or one of its descendents. Here is a simple thing that accepts events on the default input port and sends values to one or more of three ports:

```python
class MultiPortOutputThing(OutputThing, InputThing):
    def __init__(self, previous_in_chain):
        super().__init__(ports=['divisible_by_two', 'divisible_by_three',
                                'other'])
        # connect to the previous filter
        self.disconnect_from_upstream = previous_in_chain.connect(self)

    def on_next(self, x):
        val = int(round(x.val))
        if (val%2)==0:
            self._dispatch_next(val, port='divisible_by_two')
        if (val%3)==0:
            self._dispatch_next(val, port='divisible_by_three')
        if (val%3)!=0 and (val%2)!=0:
            self._dispatch_next(val, port='other')
```

```python
    def on_completed(self):
        self._dispatch_completed(port='divisible_by_two')
        self._dispatch_completed(port='divisible_by_three')
        self._dispatch_completed(port='other')

    def on_error(self, e):
        self._dispatch_error(e, port='divisible_by_two')
        self._dispatch_error(e, port='divisible_by_three')
        self._dispatch_error(e, port='other')
```

In the _init__ constructor, we must be sure to call the super class's constructor, passing it the list of ports that will be used. If the list is not provided, it is initialized to the default port, and sending to any other port would be a runtime error.

This thing will accept events from the default input port, so we subclass from `InputThing` and process sensor values in the `on_next()` method. We first obtain a value from the event and round it to the nearest integer. Next, we see if it is divisible by 2. If so, we call `_dispatch_next()` to dispatch the value to the `divisible_by_two` port, passing the port name as the second parameter (it defaults to `default`). Next, we check for divisibity by three, and dispatch the value to the `divisible_by_three` port if it is divisible. Note that a number like six will get dispatched to both ports. Finally, if the value is not divisible by either two or three, we dispatch it to the `other` port.

For the `on_completed()` and `on_error()` events, we forward the notifications to each of the output ports, by calling `_dispatch_completed()` and `_dispatch_next()` three times. In general, each port can be viewed as a separate event stream with its own state. An output thing might decide to mark completed a subset of its ports while continuing to send new events on other ports.

Let us look at how this thing might be called:

```python
sensor = SensorAsOutputThing(RandomSensor(1, mean=10, stddev=5,
                                          stop_after_events=10))
mtthing = MultiPortOutputThing(sensor)
mtthing.connect(lambda v: print("even: %s" % v),
                port_mapping=('divisible_by_two', 'default'))
mtthing.connect(lambda v: print("divisible by three: %s" % v),
                port_mapping=('divisible_by_three', 'default'))
mtthing.connect(lambda v: print("not divisible: %s" % v),
                port_mapping=('other', 'default'))
scheduler.schedule_recurring(sensor)
scheduler.run_forever()
```

Here, we map a different anonymous print function to each output port of the thing. Internally, `connect` is wrapping the anonymous functions with `CallableAsInputThing`. This thing only listens on a default port, so we have to map the port names in the `connect()` calls.

The full code for this example is at `examples/multi_port_example.py`.

Multiple Input Ports

Now, let us consider a thing that supports incoming messages on multiple ports. Messages on non-default input ports are passed to different methods on an input thing. Specifically, given a port name `PORT`, events are dispatched to the method `on_PORT_next()`, completion of the port's stream is dispatched to `on_PORT_completed()`, and errors are dispatched to `on_PORT_error()`. Multiple ports are frequently useful when implementing state machines or filters that combine multiple inputs.

As an example, assume that we have a state machine that reads data from two sensors: a *left* sensor and a *right* sensor. Here is how the code might be structured:

```python
class StateMachine:
    def on_left_next(self, x):
        ...
    def on_left_completed(self):
        ...
    def on_left_error(self):
        ...
    def on_right_next(self, x):
        ...
    def on_right_completed(self):
        ...
    def on_right_error(self):
        ...
```

Here is how we might set up the connections to the sensors:

```python
left = SensorAsOutputThing(LuxSensor('left'))
right = SensorPsOutputThing(LuxSensor('right'))
state_machine = StateMachine()
left.connect(state_machine, port_mapping=('default', 'left'))
right.connect(state_machine, port_mapping=('default', 'right'))
```

Each sensor outputs its data on the default port, so we map the connections to the `left` and `right` ports on the state machine.

Multi-port Filters

A *filter* is an ThingFlow element that has both default input and default output ports. Filters can be easily connected into pipelines. Filters usually have a single input port and a single output port, but other topologies are possible (typically one-to-many or many-to-one). One particularly useful filter is the *dispatcher*. A dispatcher routes each incoming event (on the default input port) to one of several output ports, based on some criteria.

For example, consider the filter `thingflow.filters.dispatch.Dispatcher`. This filter is provided a set of routing rules in the form of (predicate function, output port) pairs. An output port is created for each rule (plus the default port). In the `on_next()` method of the filter's InputThing interface, an incoming event is tested on each of the predicate functions in order. When a predicate is found that returns true, the event is dispatched to the associated port and the rule search stops for that event. If an event fails all the predicate checks, it is passed to the `default` port.

Here is the most relevant parts of the filter code (see `dispatch.py` for the complete code):

```python
class Dispatcher(OutputThing, InputThing):
    def __init__(self, previous_in_chain, dispatch_rules):
        ports = [port for (pred, port) in dispatch_rules] + ['default']
        super().__init__(ports=ports)
        self.dispatch_rules = dispatch_rules
        self.disconnect = previous_in_chain.connect(self)

    def on_next(self, x):
        for (pred, port) in self.dispatch_rules:
            if pred(x):
                self._dispatch_next(x, port=port)
                return
        self._dispatch_next(x, port='default') # fallthrough case
```

We will use this dispatcher within a larger example in the subsection *Solar Water Heater*.

5. Functional API

Motivation

The primary API that ThingFlow provides for filters is a *fluent* API based on the concept of *method chaining*: each filter method on the `OutputThing` base class returns the last thing in the connection chain. This result can then be used for subsequent calls. For example, to apply a filter followed by a map, we might say:

```
thing.filter(lambda evt: evt.val > 300).map(lambda evt:evt.val)
```

Underneath the covers, the `filter()` call returns a `Filter` object (a subclass of `OutputThing`). The `map()` method call is then made against this object.

This approach is convenient when your processing pipeline really is a straight line. If you have parallel branches, or more complex structures, you end up having to break it up with assignment statements. For example, consider the following dataflow, based on the code in `examples/rpi/lux_sensor_example.py`:

```
lux = SensorPub(LuxSensor())
lux.output()
lux.csv_writer(os.path.expanduser('~/lux.csv'))
actions = lux.map(lambda event: event.val > threshold)
actions.subscribe(GpioPinOut())
actions.subscribe(lambda v: print('ON' if v else 'OFF'))
scheduler = Scheduler(asyncio.get_event_loop())
scheduler.schedule_periodic_on_separate_thread(lux, interval)
scheduler.run_forever()
```

In the above code, `lux` has three subscribers, and the output of the `map` filter has two subscribers.

Functional API

To simplfy these cases, we provide a *functional* API that can be used in place of (or along with) the *fluent* API. For each method added to the thing via the `@filtermethod` decorator (in `thingflow.base`), a function with the

same name is added to the module containing the definition (e.g. `thingflow.filters.output` has an `output` function and `thingflow.filters.map` has `map` and `select` functions). These functions take all the parameters of the associated method call (except for the implied `self` parameter of a bound method) and return what we call a *thunk*. In this case, a thunk is a function that accepts exactly one parameter, a output thing. The thunk subscribes one or more fitlers to the output thing and, if further downstream connections are permitted, returns the last filter in the chain. When composing filters, thunks can be used as follows:

1. The `Schedule` class has `schedule_sensor()` and `schedule_sensor_on_separate_thread()` methods. These take a sensor, wrap it in a `SensorAsOutputThing` instance, and then connect a sequence of filters to the output thing. Each filter can be passed in directly or passed indirectly via thunks.

2. The module `thingflow.filters.combinators` defines several functions that can be used to combine filters and thunks. These include `compose` (sequential composition), `parallel` (parallel composition), and `passthrough` (parallel composition of a single spur off the main chain).

Example

Now, let us look at the lux sensor example, using the functional API[1]:

```
scheduler = Scheduler(asyncio.get_event_loop())
scheduler.schedule_sensor(lux, interval,
                          passthrough(output()),
                          passthrough(csv_writer('/tmp/lux.csv')),
                          map(lambda event:event.val > THRESHOLD),
                          passthrouh(lambda v: print('ON' if v else 'OFF')),
                          GpioPinOut())
scheduler.run_forever()
```

Notice that we do not need to instantiate any intermediate variables. Everything happens in the `schedule_sensor()` call. The first argument to this call is the sensor (without being wrapped in `SensorAsOutputThing`) and the second argument is the sample interval. The rest of the arguments are a sequence of filters and thunks to be called. Using a bit of ASCII art, the graph created looks as follows:

```
          output
        /
LuxSensor – csv_writer
        \
         map – lambda v: print(...)
            \
             GpioPinOut
```

The lux sensor has three connections: `output`, `csv_writer`, and `map`. We get this fanout by using the `passthrough` combinator, which creates a spur off the main chain. A `passthrough` is then used with the output of the `map`, with the main chain finally ending at `GpioPinOut`.

Combining the Fluent and Functional APIs

You can use the functional API within a fluent API method chain. For example, let us include a sequence of filters in a `passthrough()`:

---

[1] A full, self-contained version of this example may be found at `examples/functional_api_example.py`.

```
sensor = SensorAsOutputThing(LuxSensor())
sensor.passthrough(compose(map(lambda event:event.val>THRESHOLD), output()))\
       .csv_writer('/tmp/lux.csv')
```

Here, we used `compose` to build a sequence of `map` followed by `output`. Note that the final `csv_writer` call is run against the original events output by the sensor, not on the mapped events. Here is the resuting graph:

```
          map - output
         /
LuxSensor - csvwriter
```

Internals

The linq-style functions of the fluent API are defined to be a kind of extension method – their first parameter, usually named `this`, is the output thing on which the method will eventually be attached (to borrow Smalltalk terminology, the "receiver"). The function takes zero or more additional parameters and returns a `Filter` object to be used for further chaining.

The decorator `thingflow.base.filtermethod` adds a linq-function as a method on a base class (usually `OutputThing`), effectively binding the `this` parameter and, thus, the receiver. To support the functional API, the `filtermethod` decorator also wraps the linq-function in a `_ThunkBuilder` object. This object, when called with the parameters intended for our linq-function, returns a *thunk* – a function that has all parameters bound except the `this` receiver. When a thunk is called (passing a output thing as a parameter), it calls the original linq-function with the output thing as the `this` receiver and the rest of the parameters coming from the original `_ThunkBuilder` call.

The functional API also needs some special handling in cases where we may make `connect` calls under the covers (e.g. the `Scheduler.schedule_sensor()` method or the various combinators in `thingflow.filters.combinators`). Depending on whether the input thing being passed in is a filter, a thunk, a thunk-builder, or a plain function, we need to handle it differently. For example, if we are given a filter `f`, we can connect it to our receiver `this` via `this.connect(f)`. However, if we are given a thunk `t`, we achieve the same thing via `t(this)`. All of this logic is cenralized in `thingflow.base._subscribe_thunk`.

6. More Examples

This section contains more examples of ThingFlow in action. The code for these examples may be found in the `examples` subdirectory of the `thingflow-python` repository.

Solar Water Heater

In this scenario, we have a solar water heater that provides hot water for a residence. There is a water temperature sensor on the output pipe of the heater. There is also an actuator which controls a bypass valve: if the actuator is ON, the hot water is redirected to a spa, instead of going to the house. The spa is acting as a heat sink, taking up the extra heat, so that the water in the house never gets too hot.

We will implement a state machine which looks at the data from the temperature sensor and turns on the bypass valve when the heated water is too hot. To avoid oscillations, we use the following logic:

1. If the running average of the temperature exceeds T_high, turn on the bypass

2. When the running average dips below T_low (where T_low<T_high), then turn off the bypass.

Here is a diagram of the ThingFlow flow which implements this application:
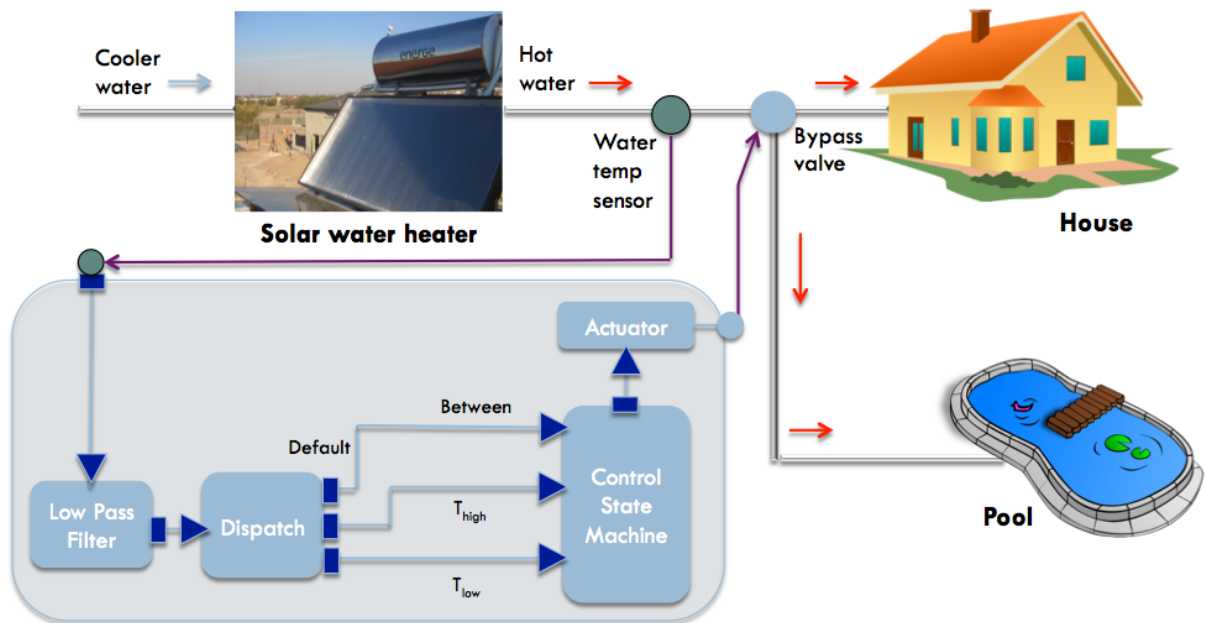
We see that the water sensor's output is run through a low pass filter to reduce noise in the reading. It is then passed to a dispatcher 1[#]_ which sends each event to one of several output ports, depending on how it compares to T_low and T_high. The control state machine determines when to turn on the actuator, based on the current state and the port of the input event.

Here is the ThingFlow code connecting everything together:

```python
T_high = 110 # Upper threshold (degrees fahrenheit)
T_low = 90 # Lower threshold
sensor = TempSensor(gpio_port=1)

# The dispatcher converts a sensor reading into
# threshold events
dispatcher = sensor.transduce(RunningAvg(5)) \
                .dispatch([(lambda v: v[2]>=T_high, 't_high'),
                           (lambda v: v[2]<=T_low, 't_low')])
controller = Controller()
dispatcher.connect(controller, port_mapping=('t_high','t_high'))
dispatcher.connect(controller, port_mapping=('t_low', 't_low'))
dispatcher.connect(controller, port_mapping=('default', 'between'))

actuator = Actuator()
controller.connect(actuator)
```

Here is a the state machine to be implemented by the `Controller` class:
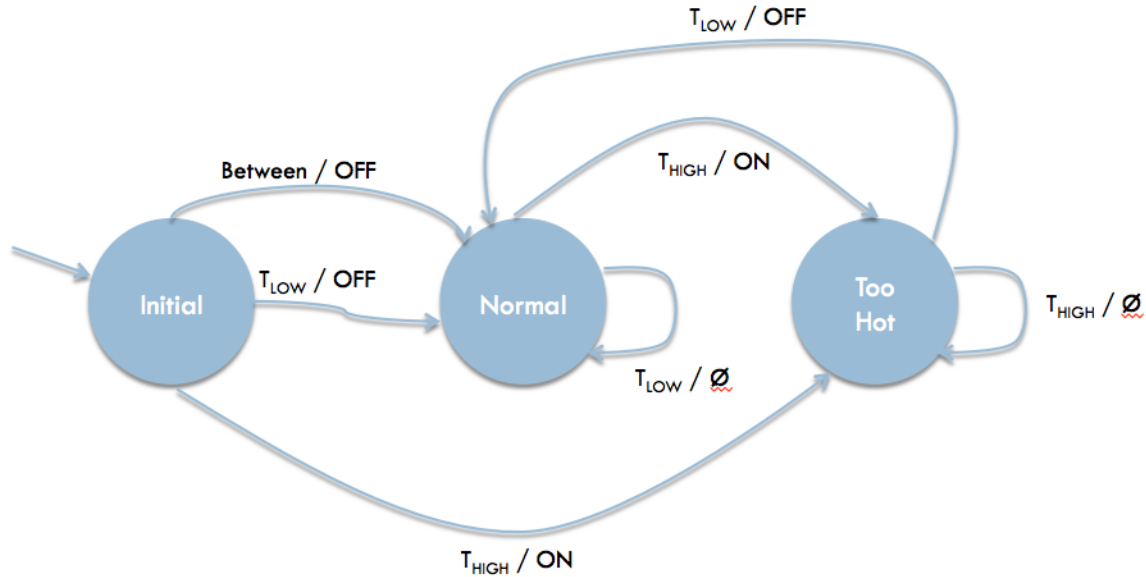
We start in the state `Initial`. If the first incoming event is `T_high`, we go to the `Normal` state. Othewise, we immediately go to `Too Hot`, which will turn off the actuator. After the initial event, we move from `Normal` to `Too Hot` if we receive a `T_high` event and then back to `Normal` when we receive `T_low`. Here is the code which implements this state machine as a ThingFlow filter:

```python
class Controller(OutputThing):
    def __init__(self):
        super().__init__()
        self.state = INITIAL
        self.completed = False

    def _make_event(self, val):
        return SensorEvent(ts=time.time(), sensor_id='Controller', val=val)

    def on_t_high_next(self, event):
        if self.state==NORMAL or self.state==INITIAL:
            self._dispatch_next(self._make_event("ON"))
            self.state = TOO_HOT
    def on_t_high_completed(self):
        if not self.completed:
            self._dispatch_completed()
            self.completed = True
    def on_t_high_error(self, e):
        pass

    def on_t_low_next(self, event):
        if self.state==TOO_HOT or self.state==INITIAL:
            self._dispatch_next(self._make_event("OFF"))
            self.state = NORMAL
    def on_t_low_completed(self):
        if not self.completed:
            self._dispatch_completed()
            self.completed = True
    def on_t_low_error(self, e):
        pass

    def on_between_next(self, x):
```

```
        if self.state==INITIAL:
            self.state = NORMAL
            self._dispatch_next(self._make_event("OFF"))
        else:
            pass # stay in current state
    def on_between_error(self, e):
        pass
    def on_between_completed(self):
        # don't want to pass this forward,
        # as it will happen after the first item
        pass
```

As you can see, we have `on_next`, `on_completed`, and `on_error` methods for each of the three input ports. A nice property of this design is that the state machine logic is isolated to a single class and does not ever deal with actual sensor readings. This makes it easy to test to test the controller logic independent of the physical sensor and actuator.

The full code for this example may be found at `examples/solar_heater_scenario.py`.

GE Predix Adapters

GE Digital's Predix platform is a public cloud service optimized for building IoT data analyses and applications. The Time Series Service supports the storage and retrieval of cloud sensor event data. ThingFlow events map very naturally to this service, and adapters are provided in the `thingflow.adapters.predix` module. This allows us to build ThingFlow applications that run "on the edge" and upload their event data to Predix for analysis.

Initial Configuration

Before using the Predix adapters, you will need to configure on Predix a UAA (User Authentication and Authorization) service and a Timeseries service. You will also need to install some client side CLI appliations to query and update Predix configurations. Instructions and hints on this may be found in a separate Github repository: https://github.com/jfischer/ge-predix-python-timeseries-example.

PredixWriter

The `PredixWriter` class is an `InputThing` which accepts ThingFlow events and sends them to the Predix Time Series Service via a websocket API. Here is some example code which instantiates a writer, connects it to a sensor and runs the resulting flow:

```
# The next three values are specific to your install
INGEST_URL = ...
PREDIX_ZONE_ID = ...
TOKEN = ...
# The data comes from a light sensor
sensor1 = SensorAsOutputThing(LuxSensor('test-sensor1'))
writer = PredixWriter(INGEST_URL, PREDIX_ZONE_ID, TOKEN,
                      batch_size=3)
sensor1.connect(writer)
scheduler = Scheduler(asyncio.get_event_loop())
scheduler.schedule_periodic(sensor1, 0.5)
scheduler.run_forever()
```

The `INGEST_URL`, `PREDIX_ZONE_ID`, and `TOKEN` parameters are described in the example repository's README file. The `batch_size` parameter indicates how many events to buffer in memory before sending them up within a single message.

By default, `PredixWriter` expects to receive instances of `SensorEvent`, which each include a Unix timestamp (in seconds as is the Python convention), a sensor id, and a value. If you want to sent data using a different representation in ThingFlow, you can subclass from `EventExtractor`, which defines how to obtain the Predix event fields from an incoming ThingFlow event. This extractor is then passed as an `extractor` keyword parameter to the `PredixWriter` constructor. Note that Predix timestamps are expressed in milliseconds rather than seconds.

PredixReader

The `PredixReader` class queries the Predix Time Series service for events from a specified sensor over a specified time range. These are then mapped to events in the ThingFlow world and passed to any connected things. Here is a short example which instantiates a `PredixReader` to query for events in the last minute, connects it `print`, schedules the reader to be run once every 60 seconds, and starts the flow.

```python
# The next three values are specific to your install
QUERY_URL = ...
PREDIX_ZONE_ID = ...
TOKEN = ...
reader = PredixReader(QUERY_URL, PREDIX_ZONE_ID, TOKEN,
                      'test-sensor1', start_time=time.time()-3600,
                      one_shot=False)
reader.connect(print)
scheduler.schedule_periodic(reader, 60)
scheduler.run_forever()
```

As with the writer, you must pass in a Query URL, Predix Zone Id, and bearer token as determined through your Predix configuration. The `start_time` parameter is the starting timestamp for the query. By specifying `one_shot` as `False`, we are requesting that the reader be run until the process is killed. If `one_shot` was set to `True`, the reader will close its event stream after one query.

If you want the reader to emit a different type of event, pass in a value for the `build_event_fn` keyword parameter of the `PredixReader` constructor. The function should take as arguments the sensor id, the predix timestamp (in milliseconds), the sensor value, and a quality code (which is dropped in the default implementation). The function should return a corresponding event for use in ThingFlow.

A complete version of this example may be found at `examples/predix_example.py`. This script can be executed from the command line.

7. ThingFlow-MicroPython Port

This section describes the port of ThingFlow to MicroPython, a bare-metal implementation of Python 3 for small processors. This port has been tested on the ESP8266 using version 1.8.7 of MicroPython.

MicroPython has only a subset of the libraries that come with the standard CPython implementation. For example, an event library, threading, and even logging are missing. This ThingFlow port currently only provides a subset of the ThingFlow functionality, due to the library limitation and memory limitations on the ESP8266.. The assumption is that processors like the ESP8266 are used primarily to sample sensor data and pass it on to a larger system (e.g. a Raspberry Pi or a server).

The code for this port may be found in the main ThingFlow-Python repository, under the `micropython` subdirectory. The core implementation is in `miropython/thingflow.py`. The other files (`logger.py`, `mqtt_writer.py`, and `wifi.py`) provide some additional utilities.

Installing

Just copy the python files in this directory to your MicroPython board. MicroPython's webrepl has experimental support for copying files. I instead used mpfshell to copy files to my ESP8266 board.

To free up more memory, I disabled the starting if the webrepl in the `boot.py` script.

Bug Workarounds

The thingflow code has a few workarounds for bugs in MicroPython (at least the ESP8266 port).

Clock wrapping

The clock on the ESP8266 wraps once every few hours. This causes problems when we wish to measure sleep time. The `utime.ticks_diff()` function is supposed to handle this, bug apparently is buggy. This leads to cases where the calculation::

```
int(round(utime.ticks_diff(end_ts, now)/1000))
```

yields 1,069,506 seconds instead of 59 seconds. Luckily, an assert in _advance_time caught the issue. The clock had clearly wrapped as end_ts (the earlier time) was 4266929 and the current timestamp was 30963.

Long variable names for keyword arguments

There is a bug in MicroPython where keyword argument names longer than 10 characters can result in a incorrect exception saying that keyword arguments are not implemented. I think this is related to MicroPython issue #1998.

Sensors

Sensor code for the MicroPython port are in the sensors subdirectory. See the README.rst file in that directory for details.

Design Notes

Scheduler Design

Since MicroPython does not provide an event scheduler,[1] we provide one directly in thingflow.py. This scheduler is optimized for minimal power consumption (by reducing wake-ups) rather than for robustness in the face of tight deadlines.

The scheduler has two layers: the *internal* layer (represented by the methods starting with an underscore) and the *public* layer. The public layer provides an API similar to the standard ThingFlow scheduler and is built on the internal layer.

For ease of testing and flexibility, the internal layer is designed such that the sensor sampling and the sleeps between events happen outside it. The internal layer is responsible for determining sleep times and the set of tasks to sample at each wakeup.

Time within the internal layer is measured in "ticks". Currently, one tick is 10 ms (1 "centisecond"). Fractional ticks are not allowed, to account for systems that cannot handle floating point sleep times. The scheduler tracks the current logical time in ticks and updates this based on elapsed time. To avoid issues with unexpected wrapping of the logical clock, it is automatically wrapped every 65535 ticks. The logical times for each for each scheduled task are then updated to reflect the wrapped clock.

When a task (output_thing) is added to the scheduler, a sample interval is specified. To optimize for wakeups, the following approaches are used:

1. Tasks with the same interval are always scheduled together. If a new task is added that matches an older task's interval, it will not be scheduled until the existing one is run.

2. If there are no tasks with the same interval, we look for the smallest interval that is either a factor or multiple of the new interval. We schedule the new interval to be coordinated with this one. For example, if we have a new interval 60 seconds and old intervals 30/45 seconds, we will schedule the new 60 second interval to first run on the next execution of the 30 second tasks. Thus, they will run at the same time for each execution of the 60 second interval.

---

[1] I have heard rumors about a port of the async library to MicroPython. However, it still makes sense to use this custom scheduler, as it is likely to be more power efficent due to its strategy of scheduling events together.

3. The next time for a task is maintained in logical ticks. If a task is run later than its interval, the next scheduled execution is kept as if the task had run at the correct time (by making the interval shorter). This avoids tasks getting out-of-sync when one misses a deadline.

The public API layer is a subset of the standard ThingFlow scheduler API, except for the additional `schedule_sensor` convenience method.

Logger

`logger.py` provides a very simple rotating file logger with an API that is a subset of Python's `logging` API. Given a log file `outputfile`, it will log events to that file until the length would exceed `max_len`. At that point, it renames the file to `outputfile`.1 and then starts a new logfile. Thus, the maximum size of the logs at any given time should be 2*``max_len``.

To use the logger, you need to first call `initialize_logging()`. You can then call `get_logger()` to get the logger instance. Note that there is a single global log level. The whole mess of handlers, formatters, and filters is not provided.

8. Design Notes

This section describes some design decisions in the ThingFlow API that are or were under discussion.

Closed Issues

These issues have already been decided, and any recommended changes implemented in the ThingFlow API. The text for each issue still uses the future tense, but we provide the outcome of the decision at the end of each section.

Basic Terminology

The terminology has evolved twice, once from the original *observer* and *observable* terms used by Mirosoft's RxPy to *subscribers* and *publishers*. Our underling communication model is really an internal publish/subscribe between the "things". This was the terminology used in our AntEvents framework.

We still found that a bit confusing and changed to the current terminology of *input things* and *output things*. Rather than topics, we have ports, which are connected rather than subscribed to. We think this better reflects the dataflow programming style.

** Outcome**: Changed

Output Things, Sensors, and the Scheduler

Today, sensors are just a special kind of output thing. Depending on whether it is intended to be blocking or non-blocking, it implements _observe or observe_and_enqueue. The reasoning behind this was to make it impossible to schedule a blocking sensor on the main thread. Perhaps this is not so important. If we relaxed this restriction, we could move the dispatch logic to the scheduler or the the base OutputThing class.

This change would also allow a single output thing implementation to be used with most sensors. We could then build a separate common interface for sensors, perhaps modeled after the Adafruit Unified Sensor Driver (https://github.com/adafruit/Adafruit_Sensor).

Outcome: Changed

We created the *sensor* abstraction and the `SensorAsOutputThing` wrapper class to adapt any sensor to the output thing API. We left the original output thing API, as there are still cases (e.g. adapters) that do not fit into the sensor sampling model.

Open Issues

At the end of each issue, there is a line that indicates the current bias for a decision, either **Keep as is** or **Change**.

Disconnecting

In the current system, the `OutputThing.connect` method returns a "disconnect" thunk that can be used to undo the connection. This is modeled after the `subscribe` method in Microsoft's Rx framework. Does this unnecessarily complicate the design? Will real dataflows use this to change their structure dynamically? If we eventually implement some kind of de-virtualization, it would be difficult to support disconnecting. Also, it might be more convenient for `connect` to return either the connected object or the output thing, to allow for method chaining like we do for filters (or is that going to be too confusing?).

As an argument for keeping the disconnect functionality, we may want to change scheduled output things so that, if they have no connections, they are unscheduled (or we could make it an option). That would make it easy to stop a sensor after a certain number of calls by disconnecting from it.

Bias: **Keep as is**

Terminology: Reader/Writer vs. Source/Sink

We introduced the *reader* and *writer* terms to refer to output things that introduce event streams into the system and input things that consume event streams with no output, respectively. A thing that accepts messages from an external source is a *output thing* in our system and an thing that emits messages to an external distination is an *input thing*. That is really confusing!

Reader/writer is better, but it might still be confusion that a reader is injecting messages into an ThingFlow dataflow. Perhaps the terms *source* and *sink* would be more obvious. Is it worth the change?

Bias: **Keep as is**

The `on_error` Callback

Borrowing from Microsoft's Rx framework, ThingFlow has three callbacks on each subscriber: `on_next`, `on_completed`, and `on_error`. The `on_error` callback is kind of strange: since it is defined to be called *at most once*, it is really only useful for fatal errors. A potentially intermittent sensor error would have to to be propagated in-band (or via another topic in ThingFlow). In that case, what is the value of an `on_error` callback over just throwing a fatal exception? ThingFlow does provide a `FatalError` exception class. Relying just on the `on_error` callbacks makes it too easy to accidently swallow a fatal error.

There are two reasons I can think of for `on_error`:

1. Provide downstream components a chance to release resources. However, if we going to stop operation due to a fatal error, we would probably just want to call it for all active things in the system (e.g. an unrelated thing may need to save some internal state). We could let the system keep running, but that may lead to a zombie situation. It is probably better to fail fast and let some higher level component resolve the issue (e.g. via a process restart).

2. If a sensor fails, we may want to just keep running and provide best guess data going forward in place of that sensor. The `on_error` callback gives us the opportunity to do that without impacting the downstream things. However, I am not sure how likely this use case is compared to the case where we have an intermittent error (e.g. a connection to a sensor node is lost, but we will keep retrying the connection).

In general, error handling needs more experience and thought.

Bias: **Change, but not sure what to**

Related Work

The architecture was heavily influenced by Microsoft's Rx (Reactive Extensions) framework and the Click modular router. We started by trying to simplfy Rx for the IoT case and remove some of the .NETisms. A key addition was the support for multiple ports, which makes more complex dataflows possible.

9. ThingFlow-Python API Reference

This is the main package for antevents. Directly within this package you fill find the following module:

- *base* - the core abstractions and classes of the system.

The rest of the functionality is in sub-packages:

- *adapters* - components to read/write events outside the system

- *internal* - some internal definitions

- *filters* - filters that allow linq-style query pipelines over event streams

- *sensors* - interfaces to sensors go here

thingflow.base

Base functionality for ThingFlow. All the core abstractions are defined here. Everything else is just subclassing or using these abstractions.

The key abstractions are:

- **Thing - a unit of computation in the data flow graph. Things can be** Filters (with inputs and outputs) or Adapters (with only inputs or only outputs).

- **OutputThing - Base class and interface for things that emit event streams on** output ports.

- **Sensor - an object that is (indirectly) connected to the physical world.** It can provide its current value through a sample() method. Sensors can be turned into Things by wrapping them with the SensorAs-InputThing class.

- **InputThing - interface for things that receive a stream of events on one or** more input ports.

- **Filter - a thing that is both an InputThing and an OutputThing, with one** input and one output. Filters transform data streams.

- **Scheduler - The scheduler wraps an event loop. It provides periodic and** one-time scheduling of Output-Things that originate events.

- **event - ThingFlow largely does not care about the particulars of the** events it processes. However, we define a generic SensorEvent datatype that can be used when the details of the event matter to a thing.

See the README.rst file for more details.

class thingflow.base.**BlockingInputThing**(*scheduler*, *ports=None*)

This implements a InputThing which may potential block when sending an event outside the system. The InputThing is run on a separate thread. We create proxy methods for each port that can be called directly - these methods just queue up the call to run in the worker thread.

The actual implementation of the InputThing goes in the _on_next, _on_completed, and _on_error methods. Note that we don't dispatch to separate methods for each port. This is because the port is likely to end up as just a message field rather than as a separate destination in the lower layers.

request_stop()

This can be called to stop the thread before it is automatically stopped when all ports are closed. The close() method will be called and the InputThing cannot be restarted later.

class thingflow.base.**CallableAsInputThing**(*on_next=None*, *on_error=None*, *on_completed=None*, *port=None*)

Wrap any callable with the InputThing interface. We only pass it the on_next() calls. on_error and on_completed can be passed in or default to noops.

class thingflow.base.**DirectOutputThingMixin**

This is the interface for OutputThings that should be directly scheduled by the scheduler (e.g. through schedule_recurring(), schedule_periodic(), or schedule_periodic_on_separate_thread).

class thingflow.base.**EventLoopOutputThingMixin**

OutputThing that gets messages from an event loop, either the same loop as the scheduler or a separate one.

exception thingflow.base.**ExcInDispatch**

Dispatching an event should not raise an error, other than a fatal error.

exception thingflow.base.**FatalError**

This is the base class for exceptions that should terminate the event loop. This should be for out-of-bound errors, not for normal errors in the data stream. Examples of out-of-bound errors include an exception in the infrastructure or an error in configuring or dispatching an event stream (e.g. publishing to a non-existant port).

class thingflow.base.**Filter**(*previous_in_chain*)

A filter has a default input port and a default output port. It is used for data transformations. The default implementations of on_next(), on_completed(), and on_error() just pass the event on to the downstream connection.

class thingflow.base.**FunctionFilter**(*previous_in_chain*, *on_next=None*, *on_completed=None*, *on_error=None*, *name=None*)

Implement a filter by providing functions that implement the on_next, on_completed, and one_error logic. This is useful when the logic is really simple or when a more functional programming style is more convenient.

Each function takes a "self" parameter, so it works almost like it was defined as a bound method. The signatures are then:

```
on_next(self, x)
on_completed(self)
on_error(self, e)
```

If a function is not provided to __init__, we just dispatch the call downstream.

class thingflow.base.**FunctionIteratorAsOutputThing**(*initial_state*, *condition*, *iterate*, *result_selector*)

Generates an OutputThing sequence by running a state-driven loop producing the sequence's elements. Example:

```
    res = GenerateOutputThing(0,
                              lambda x: x < 10,
                              lambda x: x + 1,
                              lambda x: x)

initial_state: Initial state.
condition: Condition to terminate generation (upon returning False).
iterate: Iteration step function.
result_selector: Selector function for results produced in the sequence.

Returns the generated sequence.
```

class thingflow.base.**InputThing**

This is the interface for the default input port of a Thing. Other (named) input ports will define similar methods with the names as on_PORT_next(), on_PORT_error(), and on_PORT_completed().

class thingflow.base.**IterableAsOutputThing**(*iterable*, *name=None*)

Convert any interable to an OutputThing. This can be used with the schedule_recurring() and schedule_periodic() methods of the scheduler.

class thingflow.base.**OutputThing**(*ports=None*)

Base class for event generators (output things). The non-underscore methods are the public end-user interface. The methods starting with underscores are for interactions with the scheduler.

connect(*input_thing*, *port_mapping=None*)

Connect the InputThing to events on a specific port. The port mapping is a tuple of the OutputThing's port name and InputThing's port name. It defaults to (default, default).

This returns a fuction that can be called to remove the connection.

pp_connections()

pretty print the set of connections

print_downstream()

Recursively print all the downstream paths. This is for debugging.

trace_downstream()

Install wrappers that print a trace message for each event on this thing and all downsteam things.

class thingflow.base.**Scheduler**(*event_loop*)

Wrap an asyncio event loop and provide methods for various kinds of periodic scheduling.

run_forever()

Call the event loop's run_forever(). We don't really run forever: the event loop is exited if we run out of scheduled events or if stop() is called.

schedule_on_main_event_loop(*output_thing*)

Schedule an OutputThing that runs on the main event loop. The OutputThing is assumed to implement EventLoopOutputThingMixin. Returns a callable that can be used to unschedule the OutputThing.

schedule_on_private_event_loop(*output_thing*)

Schedule an OutputThing that has its own event loop on another thread. The OutputThing is assumed to implement EventLoopOutputThingMixin. Returns a callable that can be used to unschedule the Output-Thing, by requesting that the event loop stop.

schedule_periodic(*output_thing*, *interval*)

Returns a callable that can be used to remove the OutputThing from the scheduler.

schedule_periodic_on_separate_thread(*output_thing*, *interval*)

Schedule an OutputThing to run in a separate thread. It should implement the DirectOutputThingMixin. Returns a callable that can be used to unschedule the OutputThing, by requesting that the child thread stop.

schedule_recurring(*output_thing*)

Takes a DirectOutputThingMixin and calls _observe() to get events. If, after the call, there are no down-stream connections, the scheduler will deschedule the output thing.

This variant is useful for something like an iterable. If the call to get the next event would block, don't use this! Instead, one of the calls that runs in a separate thread (e.g. schedule_recuring_separate_thread() or schedule_periodic_separate_thread()).

Returns a callable that can be used to remove the OutputThing from the scheduler.

schedule_sensor(*sensor*, *interval*, **input_thing_sequence*, *make_event_fn=<function make_sensor_event>*, *print_downstream=False*)

Create a OutputThing wrapper for the sensor and schedule it at the specified interval. Compose the specified connections (and/or thunks) into a sequence and connect the sequence to the sensor's OutputThing. Returns a thunk that can be used to remove the OutputThing from the scheduler.

schedule_sensor_on_separate_thread(*sensor*, *interval*, **input_thing_sequence*, *make_event_fn=<function make_sensor_event>*)

Create a OutputThing wrapper for the sensor and schedule it at the specified interval. Compose the specified connections (and/or thunks) into a sequence and connect the sequence to the sensor's OutputThing. Returns a thunk that can be used to remove the OutputThing from the scheduler.

stop()

Stop any active schedules for output things and then call stop() on the event loop.

class thingflow.base.**SensorAsOutputThing**(*sensor*, *make_event_fn=<function make_sensor_event>*)

OutputThing that samples a sensor upon its observe call, creates an event from the sample, and dispatches it forward. A sensor is just an object that has a sensor_id property and a sample() method. If the sensor wants to complete the stream, it should throw a StopIteration exception.

By default, it generates SensorEvent instances. This behavior can be changed by passing in a different function for make_event_fn.

class thingflow.base.**SensorEvent**(*sensor_id*, *ts*, *val*)

sensor_id

Alias for field number 0

ts

Alias for field number 1

val

Alias for field number 2

class thingflow.base.**XformOrDropFilter**(*previous_in_chain*)

Implements a slightly more complex filter protocol where events may be transformed or dropped. Subclasses just need to implement the _filter() and _complete() methods.

on_completed()

Passes on any final event and then passes the notification to the next Thing. If you need to clean up any state, do it in _complete().

on_error(*e*)

Passes on any final event and then passes the notification to the next Thing. If you need to clean up any state, do it in _complete().

on_next(*x*)

Calls _filter(x) to process the event. If _filter() returns None, nothing futher is done. Otherwise, the return value is passed to the downstream connection. This allows you to both transform as well as send only selected events.

> Errors other than FatalError are handled gracefully by calling self.on_error() and then disconnecing from the upstream OutputThing.

`thingflow.base.`**`filtermethod`**(*base*, *alias=None*)

> Function decorator that creates a linq-style filter out of the specified function. As described in the thingflow.linq documentation, it should take a OutputThing as its first argument (the source of events) and return a OutputThing (representing the end the filter sequence once the filter is included. The returned OutputThing is typically an instance of thingflow.base.Filter.
>
> The specified function is used in two places:
>
> 1. A method with the specified name is added to the specified class (usually the OutputThing base class). This is for the fluent (method chaining) API.
>
> 2. A function is created in the local namespace for use in the functional API. This function does not take the OutputThing as an argument. Instead, it takes the remaining arguments and then returns a function which, when passed a OutputThing, connects to it and returns a filter.
>
> Decorator arguments:
>
> • param T base: Base class to extend with method (usually thingflow.base.OutputThing)
>
> • **param string alias: an alias for this function or list of aliases** (e.g. map for select, etc.).
>
> • returns: A function that takes the class to be decorated.
>
> • rtype: func -> func
>
> This was adapted from the RxPy extensionmethod decorator.

`thingflow.base.`**`make_sensor_event`**(*sensor*, *sample*)

> Given a sensor object and a sample taken from that sensor, return a SensorEvent tuple.

thingflow.sensors

The sensors are not included in the auto-generated documentation, as importing the code requires external libraries (not possible for automated documentation generation). Here is a list of available sensor modules in the ThingFlow-Python distribution:

- `rpi.adxl345_py3` - interface to the adxl345 accelerometer

- `rpi.arduino` - interface an Arduino to the Raspberry Pi

- `rpi.gpio` - read from the Raspberry Pi GPIO pins

- `lux_sensor` - read from a TSL2591 lux sensor

Please see the source code for more details on these sensors.

thingflow.filters

This sub-module provides a collection of filters for providing linq-style programming (inspired by RxPy).

Each function appears as a method on the OutputThing base class, allowing for easy chaining of calls. For example:

```
sensor.where(lambda x: x > 100).select(lambda x: x*2)
```

If the @filtermethod decorator is used, then a standalone function is also defined that takes all the arguments except the publisher and returns a function which, when called, takes a publisher and subscribes to the publisher. We call this returned function a "thunk". Thunks can be used with combinators (like compose(), parallel(), and passthrough(), all defined in combinators.py) as well as directly with the scheduler. For example:

```
scheduler.schedule_sensor(sensor, where(lambda x: x> 100),
                                   select(lambda x: x*2))
```

The implementation code for a linq-style filter typically looks like the following:

```
@filtermethod(OutputThing)
def example(this, ...):
    def _filter(self, x):
        ....
    return FunctionFilter(this, _filter, name="example")
```

Note that, by convention, we use *this* as the first argument of the function, rather than self. The *this* parameter corresponds to the previous element in the chain, while the *self* parameter used in the _filter() function represents the current element in the chain. If you get these mixed up, you can get an infinite loop!

In general, a linq-style filter takes the previous OutputThing/filter in a chain as its first input, parameters to the filter as subsequent inputs, and returns a OutputThing/filter that should be used as the input to the next step in the filter chain.

thingflow.filters.buffer

class `thingflow.filters.buffer.`**BufferEventUntilTimeoutOrCount**(*previous_in_chain*, *event_watcher*, *scheduler*, *interval=None*, *count=None*)

A class that passes on the events on the default channel to a buffer (maintained by a BufferEventWatcher). When a timeout fires, the BufferEventWatcher returns the buffer of all events so far.

on_timeout_next(*x*)

We got the buffered events from the timeout – send it to the subscribers and reset the timer

thingflow.filters.combinators

This module defines combinators for linq-style functions: compose, parallel, and passthrough. A linq-style function takes the previous OutputThing/filter in a chain as its first input ("this"), parameters to the filter as subsequent inputs, and returns a OutputThing/filter that should be used as the input to the next step in the filter chain.

We use the term "thunk" for the special case where the linq-style function takes only a single input - the previous OutputThing/filter in the chain. The Scheduler.schedule_sensor() method and the functions below can accept thunks in place filters. If a linq-style filter F was defined using the @filtermethod decorator, then calling the function directly (not as a method of a OutputThing) returns a thunk.

`thingflow.filters.combinators.`**compose**(**thunks*)

Given a list of thunks and/or filters, compose them in a sequence and return a thunk.

`thingflow.filters.combinators.`**parallel**(**connectees*)

Take one or more InputThings/thunks and create a thunk that will connect all of them to "this" when evaluated. Note that the entire set of InputThings acts as spurs - the original OutputThing is returned as the next OutputThing in the chain.

thingflow.filters.dispatch

class `thingflow.filters.dispatch.`**`Dispatcher`**(*previous_in_chain*, *dispatch_rules*)
   Dispatch rules are a list of (predicate, port) pairs. See the documentation on the dispatch() extension method for details.

thingflow.filters.first

thingflow.filters.json

thingflow.filters.map

Transform each event in the stream. thingflow.filters.select and thingflow.filters.map have the same functionality. Just import one - the @filtermethod decorator will create the other as an alias.

thingflow.filters.never

class `thingflow.filters.never.`**`Never`**
   An OutputThing that never calls its connections: creates an empty stream that never goes away

thingflow.filters.output

thingflow.filters.scan

thingflow.filters.select

Transform each event in the stream. thingflow.filters.select and thingflow.filters.map have the same functionality. Just import one - the @filtermethod decorator will create the other as an alias.

thingflow.filters.skip

thingflow.filters.some

thingflow.filters.take

thingflow.filters.timeout

Timeout-related output things and filters.

class `thingflow.filters.timeout.`**`EventWatcher`**
   Watch the event stream and then produce an event for a timeout when asked. This can be subclassed to implement different policies.

class `thingflow.filters.timeout.`**`SupplyEventWhenTimeout`**(*previous_in_chain*,
                                                                    event_watcher,   *scheduler*,
                                                                    interval)
   This filter sits in a chain and passes incoming events through to its output. It also passes all events to the on_next() method of the event watcher. If no event arrives on the input after the interval has passed since the last event, event_watcher.produce_event_for_timeout() is called to get a dummy event, which is passed upstream.

on_timeout_completed()
> This won't get called, as the timeout thing does not propate any completions. We just use the primary event stream to figure out when things are done and clear any pending timeouts at that time.

on_timeout_error(*e*)
> This won't get called, as the Timeout thing does not republish any errors it receives.

on_timeout_next(*x*)
> This method is connected to the Timeout thing's output. If it gets called, the timeout has fired. We need to reschedule the timeout as well, so that we continue to produce events in the case of multiple consecutive timeouts.

class thingflow.filters.timeout.**Timeout**(*scheduler*, *timeout_thunk*)
> An output thing that can shedule timeouts for itself. When a timeout occurs, an event is sent on the default port. The timeout_thunk is called to get the actual event.

thingflow.filters.transducer

Transducers for streams. A transducer maintains internal state which is updated every time on_next is called. It implements a function f: Input X State -> Output X State

For those who speak automata, this is a Mealy machine.

class thingflow.filters.transducer.**PeriodicMedianTransducer**(*period=5*)
> Emit an event once every period input events. The value is the median of the inputs received since the last emission.

class thingflow.filters.transducer.**SensorSlidingMean**(*history_samples*)
> Given a stream of SensorEvents, output a new event representing the mean of the event values in the window. The state we keep is the sum of the .val fields within the window. We assume that all events are from the same sensor.

class thingflow.filters.transducer.**SlidingWindowTransducer**(*history_samples*)
> Transducer that processses a sliding window of events. The most recent history_samples events are kept internally in a deque. When an event arrives, it is pushed onto the deque and an old event is popped off. There are three cases: the very first event, events before the buffer is full, and events after the buffer is full. For each case, the new event, old event (if one is being popped off), and a accumulated state value are passed to a template method. The method returns the transduced event and a new value for the accumulated state. This makes it easy to efficently implement algorithms like a running average or min/max, etc.
>
> Note that the window here is based on the number of samples, not a time period.

thingflow.filters.where

thingflow.adapters

Adapters are components that connect ThingFlows to the external world. *Readers* are event output things which source an event stream into an ThingFlow process. *Writers* are input things that translate an event stream to a form used outside of the ThingFlow process. For example, *CsvReader* is a output thing that reads events from a CSV-formatted spreadsheet file and *CsvWriter* is an input thing that writes events to a CSV file.

Why don't we just call adapters OutputThings and InputThings? We want to avoid confusion do to the fact that an OutputThing is used to connect to external inputs while external outputs interface via InputThings.

thingflow.adapters.csv

Adapters for reading/writing event streams to CSV (spreadsheet) files.

class `thingflow.adapters.csv.`**`EventSpreadsheetMapping`**
Define the mapping between an event record and a spreadsheet.

> **`get_header_row`**`()`
> Return a list of header row column names.

class `thingflow.adapters.csv.`**`RollingCsvWriter`**(*previous_in_chain,*   *directory,*   *base_name,*   *mapper=<thingflow.adapters.csv.SensorEventMapping object>,*   *get_date=<function default_get_date_from_event>,*   *sub_port=None*)
Write an event stream to csv files, rolling to a new file daily. The filename is basename-yyyy-mm-dd.cvv. Typically, basename is the sensor id. If sub_port is specified, the writer will subscribe to the specified port in the previous filter, rather than the default port. This is helpful when connecting to a dispatcher.

class `thingflow.adapters.csv.`**`SensorEventMapping`**
A maping that works for SensorEvent tuples. We map the time values twice - as the raw timestamp and as an iso-formatted datetime.

thingflow.adapters.generic

Generic reader and writer classes, to be subclassed for specific adapters.

class `thingflow.adapters.generic.`**`DirectReader`**(*iterable, mapper, name=None*)
A reader that can be run in the current thread (does not block indefinitely). Reads rows from the iterable, converts them to events using the mapping and passes them on.

class `thingflow.adapters.generic.`**`EventRowMapping`**
Interface that converts between events and "rows"

> **`event_to_row`**(*event*)
> Convert an event to the row representation (usually a list of values).

> **`row_to_event`**(*row*)
> Convert a row to an event.

Other Adapters

Many adapters are not included in the auto-generated documentation, as importing the code requires external libraries (not possible for the auto document generation). Here is a list of additional adapters in the ThingFlow-Python distribution:

- `bokeh` - interface to the Bokeh visualization framework

- `influxdb` - interface to the InfluxDb time series database

- `mqtt` - interface to MQTT via `paho.mqtt`

- `mqtt_async` - interface to MQTT via `hbmqtt`

- `pandas` - convert ThingFlow events to Pandas `Series` data arrays

- `predix` - send and query data with the GE Predix Time Series API

- `postgres` - interface to the PostgreSQL database

- `rpi.gpio` - output on the Raspberry Pi GPIO pins

Please see the source code for more details on these adapters.

Indices and tables

- genindex
- modindex
- search

Python Module Index

t