diagnostics

Release 0.6.0

Timo Lesterhuis

Aug 22, 2019

CONTENTS

diagnostics is a Python module designed to make analysis of diagnostic data easier. It comes with a couple of clear data-structures with automatic quality checks, easy Boolean logic operators and built-in bookkeeping. To top that off, it's built on numpy!

diagnostics is free to use (MIT license), open source (GitHub), works with python 3.5+ and is available for Linux, OSX, and Windows.

CHAPTER

ONE

INTRODUCTION

This is the Introduction! more info will come soon.

1.1 Features

I'll sum these up at some point

1.2 Why should I use diagnostics instead of X?

Because it's way cooler, you know.

CHAPTER

TWO

GETTING STARTED

Note: You can also try out this quickstart as a jupyter notebook on Binder!

2.1 Installation

To install diagnostics, please use the following command:

```
pip install pydiagnostics
```

Alternatively, you can clone the repository and use `setup.py` to install:

```
git clone https://github.com/tim00w/diagnostics.git
cd diagnostics
python setup.py install
```

2.2 Basic usage

2.2.1 TimeSeries

Diagnostic events are derived from from real occurances. For instance, your phone will probably generate a message (event) if your battery is running low (percentage below threshold value). The diagnostics library has a `TimeSerie` class that can capture these occurances.

For example, a `TimeSerie` representing your battery life, which drains 0.01% each second:

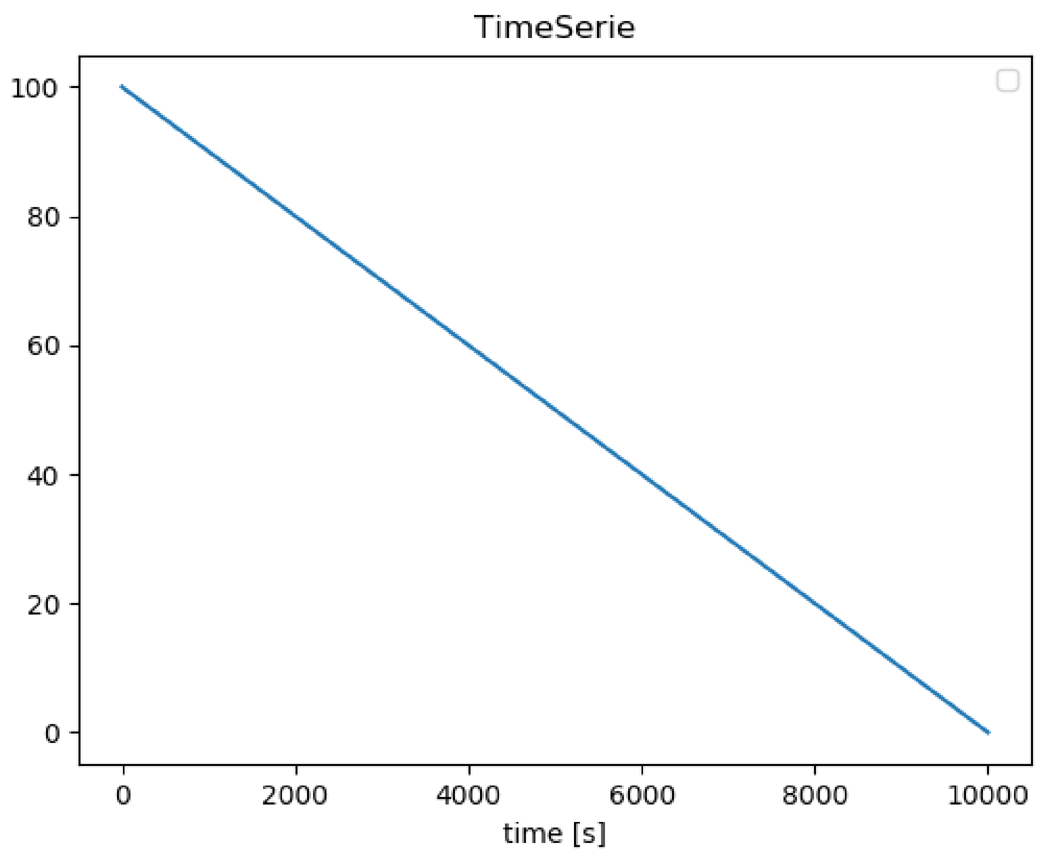
```
import numpy as np
import diagnostics as ds

battery_life = ds.TimeSerie(np.arange(100, 0, -0.01), fs=1)
```

the first argument is consists of a data array (both `list()` and `numpy.array()` are supported), and additionally you can provide some keyword parameters. Here we've provided the sample frequency (`fs`) which is 1 Hz, because we said our battery drains 0.01% each second. In this particular case we could've left `fs` out, since the default value of `fs` is also 1.

Now that we've got our data, we can easily visualize this:

```
battery_life.plot()
```

Which will show the following matplotlib figure:



There are other keyword parameters that we can use as well, such as t0 (start time of `TimeSerie` in posixtime or a `datetime` object), and a name (default is an empty string).

```
from datetime import datetime

battery_life = ds.TimeSerie(np.arange(100, 0, -0.01),
                            fs=1,
                            t0=datetime(2019,1,1,8,5), # 2019-01-01 08:05
                            name='battery life')
```

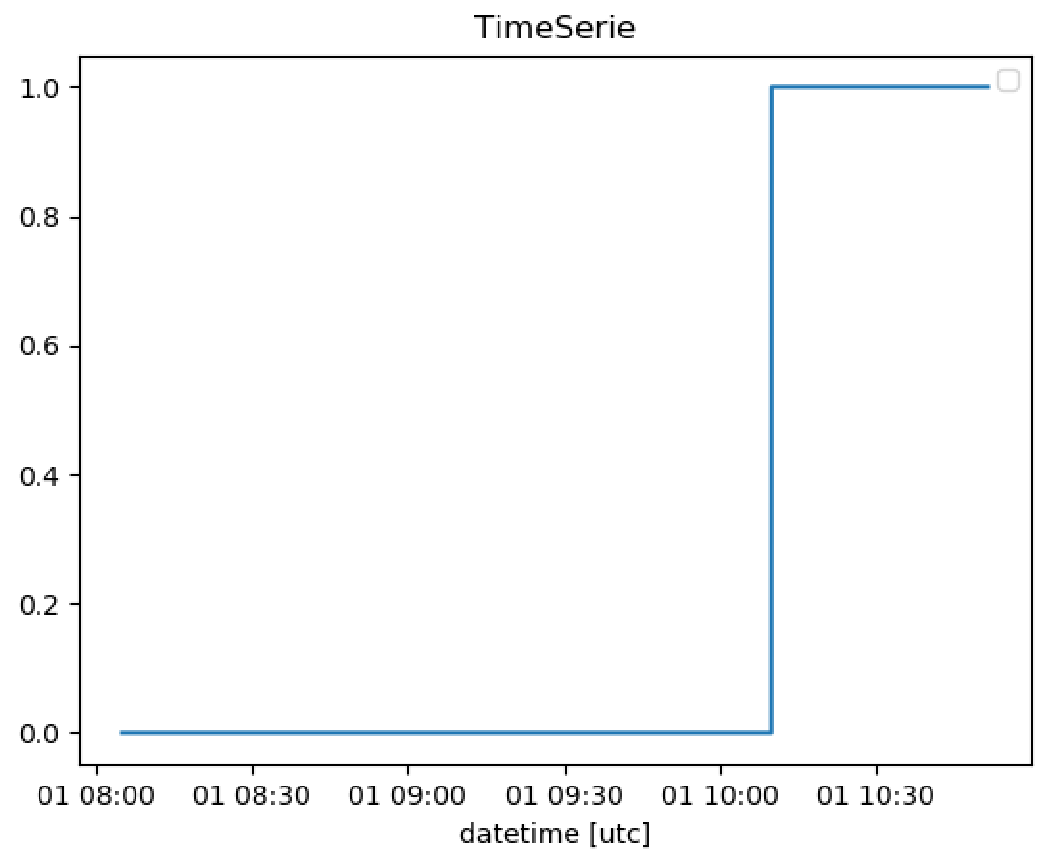
Now we've got our battery life set to a specific (start-)datetime, and gave it a name. Both will come in handy later.

2.2.2 BooleanTimeSeries

Let's be honest, the battery percentage of your phone does not really matter to you, unless it goes below a certain threshold. Luckily for us, our `TimeSerie` can easily be converted to a `BooleanTimeSerie`, which only contains boolean values of when the percentage reaches below 25%:

```
battery_below25 = battery_life <= 25

battery_below25.plot(as_dt=True, show=True)
```

Now that's easy! We can see that our battery goes below 25% around 10:10:



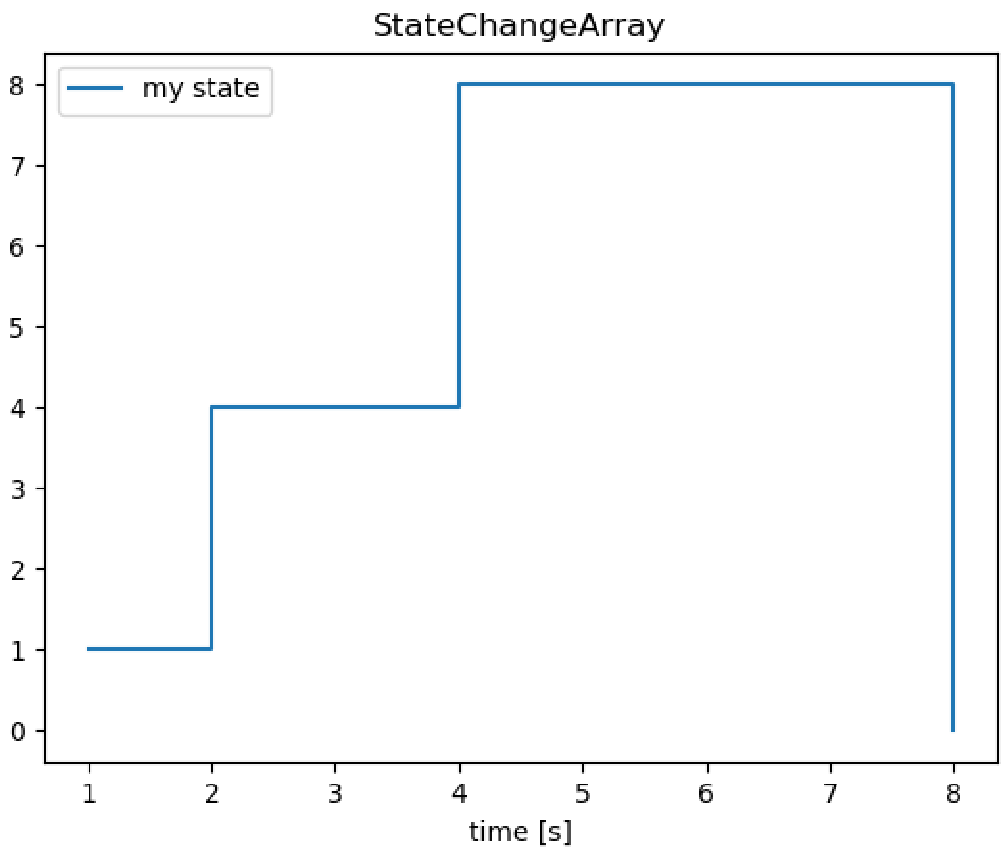
2.2.3 StateChangeArray

You could argue that our `BooleanTimeSerie` contains a lot of data points with the same value. I'd agree with you, and therefore introduce a class that only keeps track of the changes in data points, the `StateChangeArray`:

```
battery_low_state = battery_below25.to_statechangearray()
```

Alternatively, we can create a `StateChangeArray` (or `BooleanStateChangeArray`, you can probably guess the difference :smile:) from scratch:

```
s = ds.StateChangeArray([1, 4, 8, 0], t=[1,2,4,8], name='my state')
b = ds.BooleanStateChangeArray([True, False, True, False], t=[1,3,6,9], name='b')

s.plot(show=True)
```



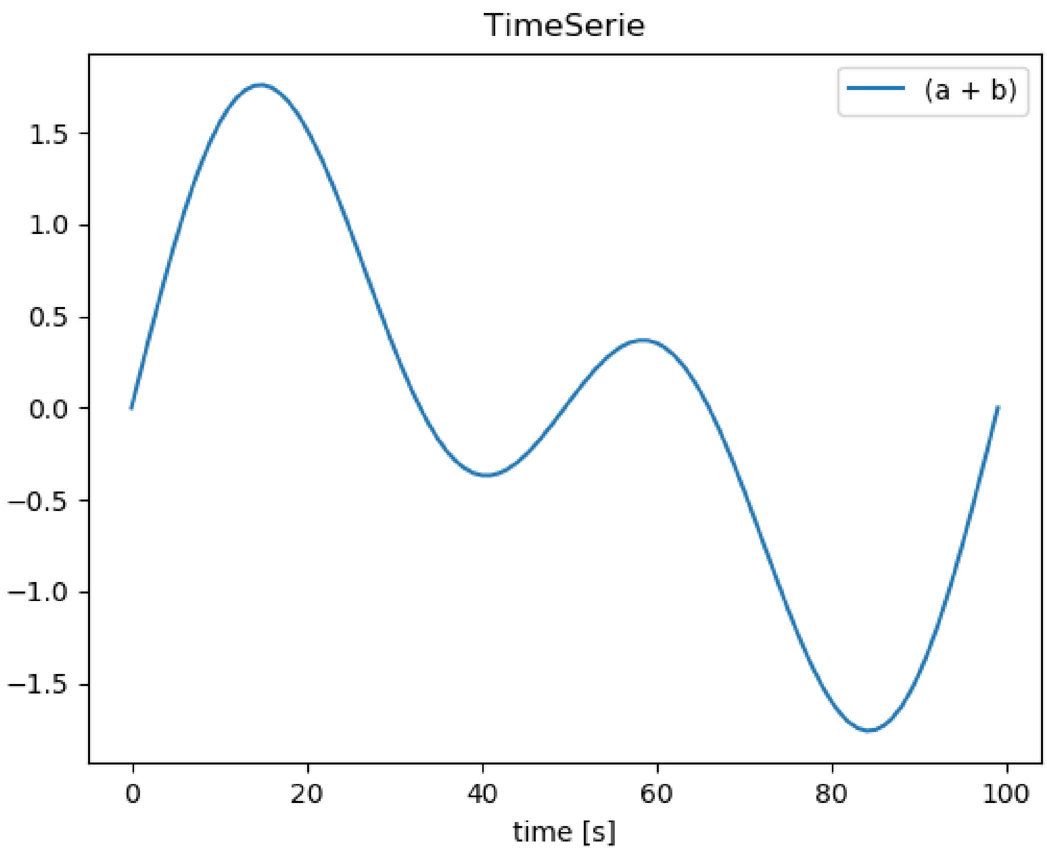
Both the data array as the values for time (`t`) can be `list()` or `np.array()`. When a list of `datetime` objects is given, they are converted to posixtime. Otherwise, the input is considered as posixtime. For now it is not possible to give a datetimearray as an input, but this wil be implemented in the near future.

2.2.4 Comparing TimeSeries and StateChangeArrays

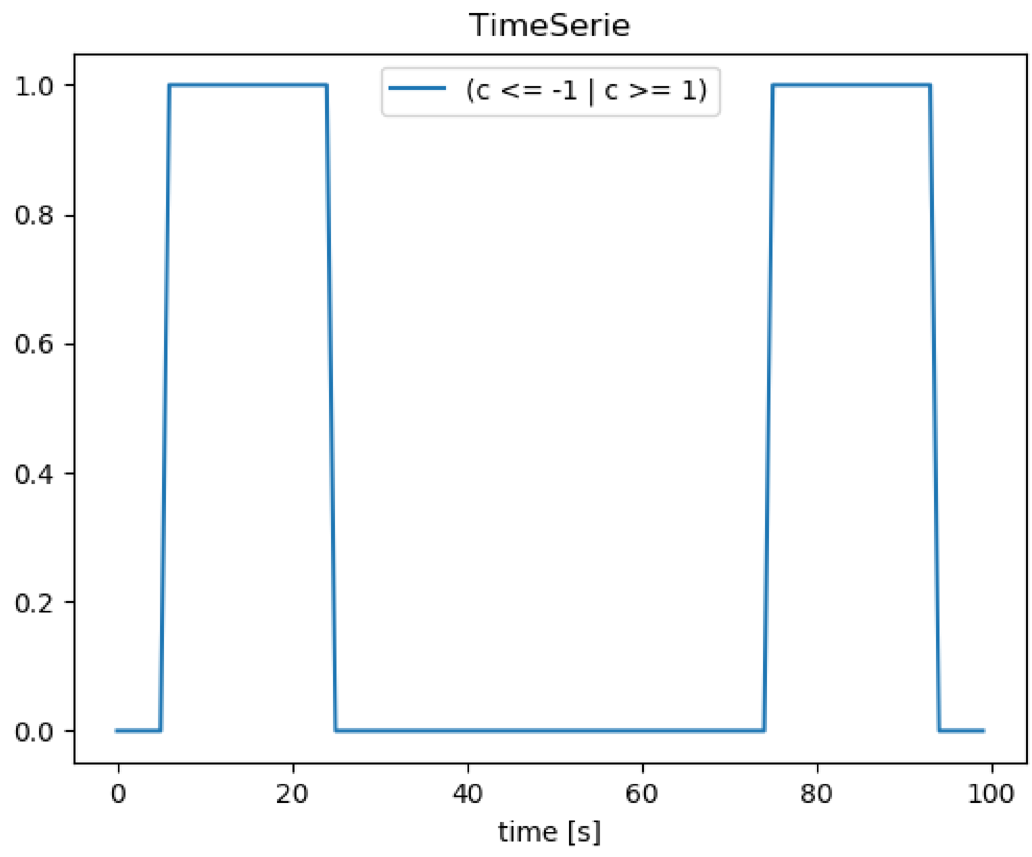
There are more classes besides TimeSeries and StateChangearrays, each with their own advantages and disadvantages. The power of this module lies in clear transformations from one class to another (we've already shown the `TimeSerie.to_statechangearray()` method), and the comparison of multiple classes.

To start with TimeSeries, if two (or more) have the same array_length, `t0` and `fs`, we can easily do calculations with them!

```
# create two TimeSerie objects that we'll combine
a = ds.TimeSerie(np.sin(np.linspace(0, 2*np.pi, 100)), t0=0, fs=1, name='a')
b = ds.TimeSerie(np.sin(2* np.linspace(0, 2*np.pi, 100)), t0=0, fs=1, name='b')

# It's this easy!
c = a + b
```



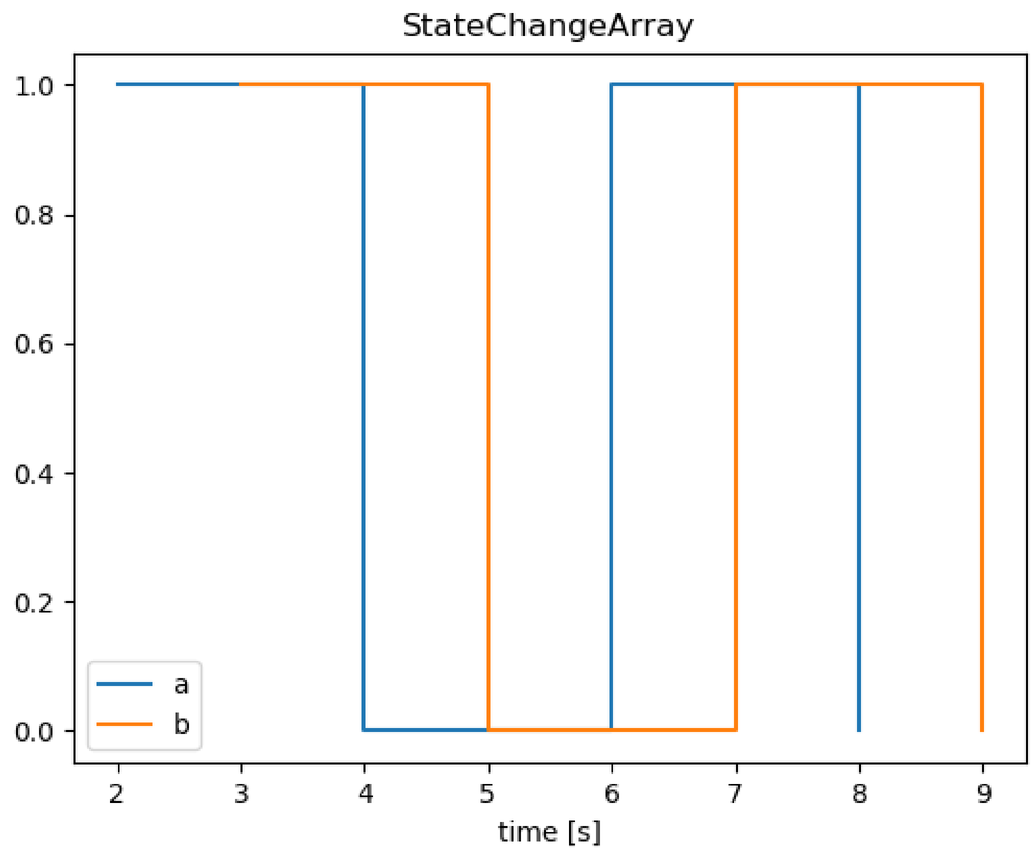
```
# We're interested in the more extreme values, lets create TimeSeries for these:
d = c <= -1
e = c >=  1

# we'll name them to keep our bookkeeping up to date
d.name = 'c <= -1'
e.name = 'c >= 1'

# and find when one of the above conditions is True!
f = d | e

# when performing boolean operators ('~', '^', '&', '|'), the library
# does it's own bookkeeping:
print(f.name)
f.plot(show=True)
```

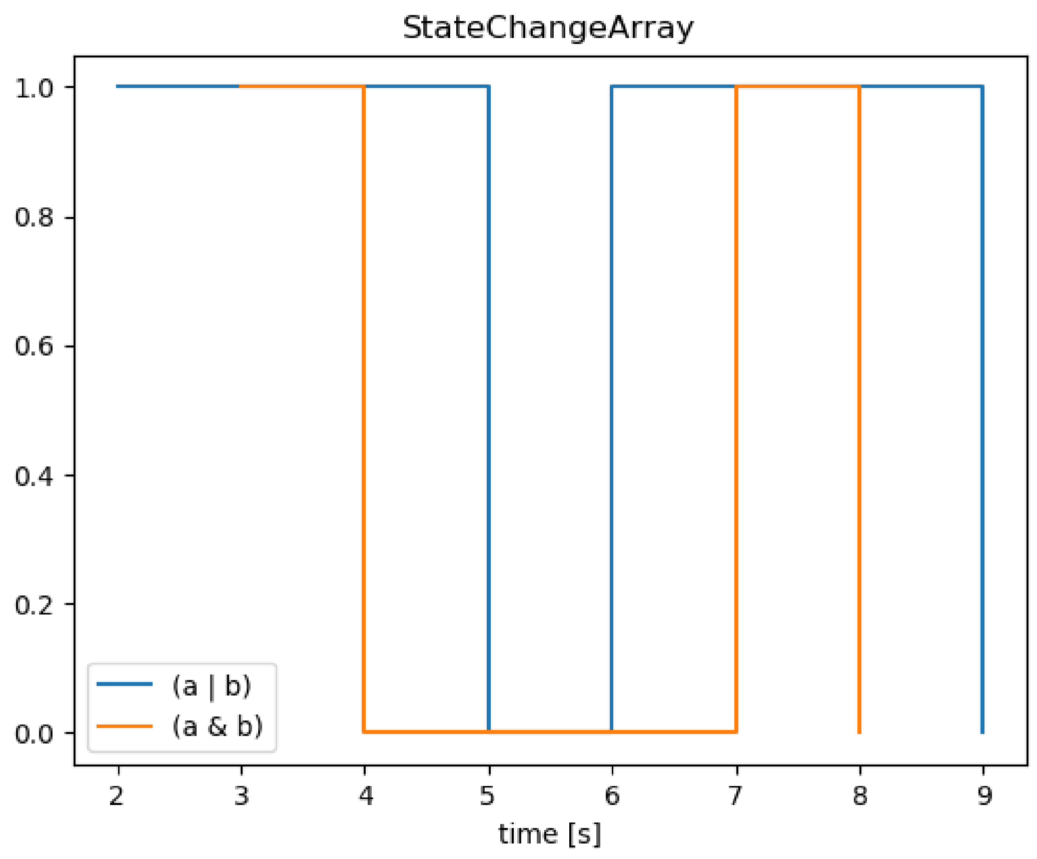


Comparing StateChangeArrays would normally be a bit tricky, since the data is most likely non-linearly spaced. This means that we can't just perform vectorized boolean operations, but we'll need to combine both data values as well as their respective points in time.

Luckily for us, the `StateChangeArray` has this functionality built in:

```
a = ds.StateChangeArray([True, False, True, False], t=[2,4,6,8], name='a')
b = ds.StateChangeArray([True, False, True, False], t=[3,5,7,9], name='b')

c = a | b
d = a & b
e = ~a
f = a ^ a
g = a ^ e

a.plot(other=b)
c.plot(other=d)
```


StateChangeArray
1.0
0.8
0.6
0.4
0.2
0.0
a
b
2
3
4
5
6
7
8
9
time [s]



That’s pretty great right?

Note: plotting is still an experimental feature, and the API/implementation might change in the near future.

2.2.5 Reports & Events

WIP

2.3 What to read next

I’m not sure either, and I’m the one that has to write it first!

CHAPTER

THREE

ADVANCED USAGE

For now there are no advanced usages! It’s an easy library :)

CHAPTER

FOUR

CHANGELOG

Everything is still rapidly changing, after a 1.0 release this will become important

WIP

CHAPTER FIVE

TROUBLESHOOTING

Honestly, what can go wrong?

CHAPTER

SIX

FAQ

Ask away!

CHAPTER

SEVEN

CONTRIBUTING

For now, just yours truly!

CHAPTER

EIGHT

API

More will be added soon!

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search