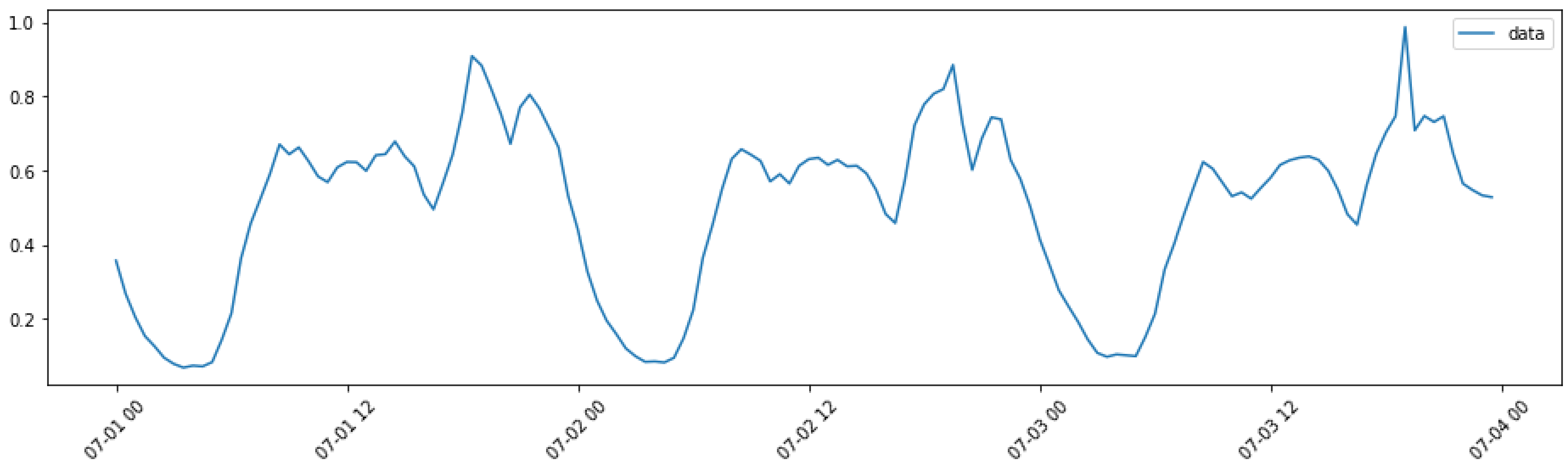
Sliding Windows

Temporal Correlations

We have show how our time series is almost periodical

In [3]:
```
nab.plot_series(data.iloc[:3*48], figsize=figsize)
```

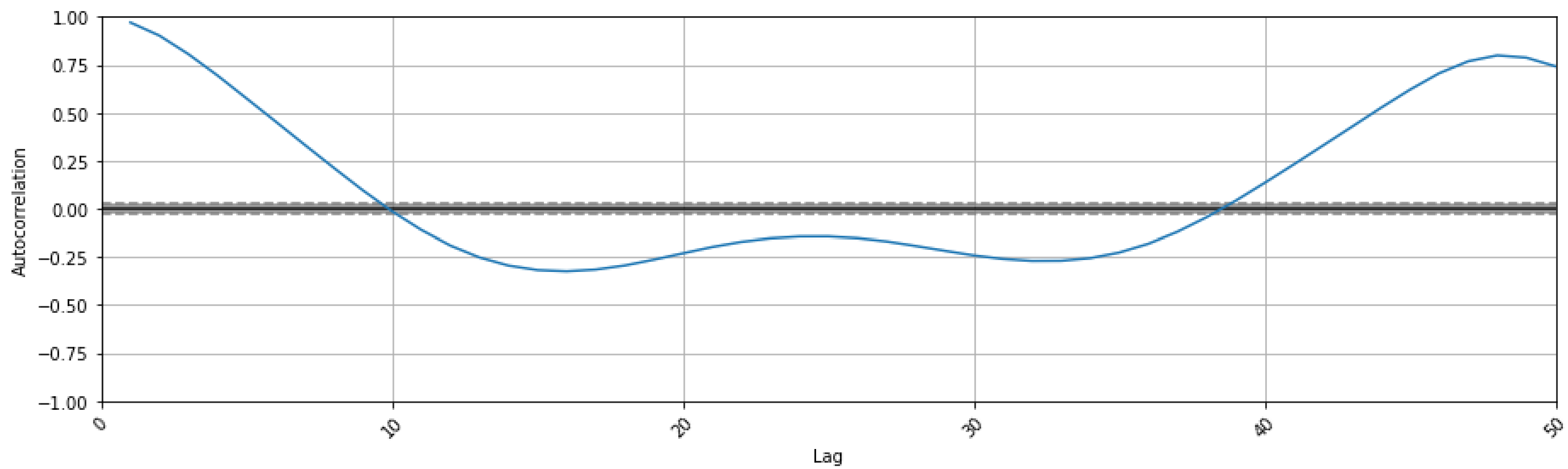


- ...But the period is not the only temporal correlation!
- There are patterns (down, up, stable) even between nearby time points

Temporal Correlations

Another way to see that: let's check again the autocorrelation plot

In [4]:
```
nab.plot_autocorrelation(data, max_lag=50, figsize=figsize)
```



- The correlation is strong up to 4-5 lags

Temporal Correlations

These correlations are a source of information

- They could be exploited to improve our estimated probabilities
- ...But our models so far make no use of them

If we want to take advantage of them:

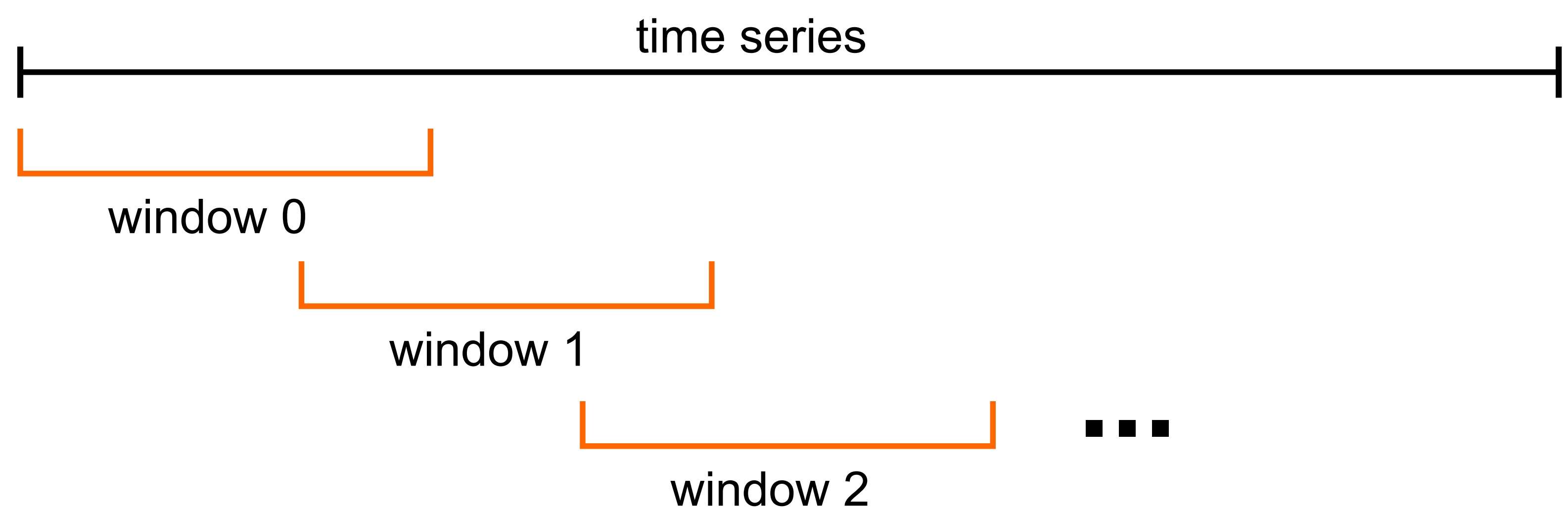
- We need to feed sequences to our estimators
- ...Rather than individual observations

Our previous approach for combining multiple observations is not enough:

- It was based on assuming i.i.d. observations
- ...And here we want to exploit dependencies!

Sliding Window

A common approach consist in using a sliding window



- We choose a window length w, i.e. the length of each sub-sequence
- We place the "window" at the beginning of the series
- ...We extract the corresponding observations
- Then, we move the forward by a certain stride and we repeat

Sliding Window

The result is a table

Let m be the number of examples and w be the window length

	$\mathbf{s_0}$	$\mathbf{s_1}$	$\ldots$	$\mathbf{s_{w-1}}$
$\mathbf{t_{w-1}}$	x_0	x_1	$\ldots$	x_{w-1}
$\mathbf{t_w}$	x_1	x_2	$\ldots$	x_w
$\mathbf{t_{w+1}}$	x_2	x_3	$\ldots$	x_{w+1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathbf{t_{m-1}}$	x_{m-w}	x_{m-w+1}	$\vdots$	x_{m-1}

- The first window includes observations from x_0 to x_{w-1}
- The second from x_1 to x_w and so on
- t_i is the time window index (where it was applied)
- s_j is the position of an observation within a window

Sliding Window in pandas

pandas provides a sliding window iterator

```
DataFrame.rolling(window, ...)
```

In [5]:
```
wlen = 48
for i, w in enumerate(data['value'].rolling(wlen)):
    print(w)
    if i == 2: break # We print the first three windows
```

```
timestamp
2014-07-01    0.357028
Name: value, dtype: float64
timestamp
2014-07-01 00:00:00    0.357028
2014-07-01 00:30:00    0.267573
Name: value, dtype: float64
timestamp
2014-07-01 00:00:00    0.357028
2014-07-01 00:30:00    0.267573
2014-07-01 01:00:00    0.204458
Name: value, dtype: float64
```

Notice how the first windows are not full (shorter than `wlen`)

Sliding Window in pandas

We can build our dataset using the `rolling` iterator

- We discard the first `wlen-1` (incomplete) applications
- Then we store each window in a list
- Finally we wrap everything in a `DataFrame`

In [6]:
```
rows = []
for i, w in enumerate(data['value'].rolling(wlen)):
    if i >= wlen-1: rows.append(w.values)

wdata_index = data.index[wlen-1:]
cols = range(wlen)
wdata = pd.DataFrame(index=wdata_index, columns=cols, data=rows)
```

- The `values` field allows access to the `Series` content as a numpy array
- We use it to discard the index
- ...Since the series for multiple iterations have inconsistent indexes

Sliding Window in pandas

This method works, but it's a bit slow

- We are building our table by rows...
- ...But it is usually faster to do it by columns!
- After all, there are usually fewer columns than rows

Let us look again at our table:

	$\mathbf{s_0}$	$\mathbf{s_1}$	$\ldots$	$\mathbf{s_{w-1}}$
$\mathbf{t_{w-1}}$	x_0	x_1	$\ldots$	x_{w-1}
$\mathbf{t_w}$	x_1	x_2	$\ldots$	x_w
$\mathbf{t_{w+1}}$	x_2	x_3	$\ldots$	x_{w+1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathbf{t_{m-1}}$	x_{m-w}	x_{m-w+1}	$\vdots$	x_{m-1}

Sliding Window in pandas

We can build the columns by slicing the original `DataFrame`

```
In [7]: m = len(data)
        c0 = data.iloc[0:m-wlen+1]    # first column
        c1 = data.iloc[1:m-wlen+1+1] # second column
        print(c0.iloc[0:3])
        print(c1.iloc[0:3])

                        value
timestamp
2014-07-01 00:00:00  0.357028
2014-07-01 00:30:00  0.267573
2014-07-01 01:00:00  0.204458
                        value
timestamp
2014-07-01 00:30:00  0.267573
2014-07-01 01:00:00  0.204458
2014-07-01 01:30:00  0.153294
```

- `iloc` in pandas allows to address a `DataFrame` by position

Sliding Window in pandas

Now we collect all columns in a list and we stack them

In [8]:
```
lc = [data.iloc[i:m-wlen+i+1].values for i in range(0, wlen)]
lc = np.hstack(lc)
wdata = pd.DataFrame(index=wdata_index, columns=cols, data=lc)
wdata.head()
```

Out[8]:

	0	1	2	3	4	5	6	7	8	9	...	38	
timestamp													
2014-07-01 23:30:00	0.357028	0.267573	0.204458	0.153294	0.125770	0.094591	0.077997	0.067955	0.073124	0.071050	...	0.883252	0.8
2014-07-02 00:00:00	0.267573	0.204458	0.153294	0.125770	0.094591	0.077997	0.067955	0.073124	0.071050	0.082804	...	0.819939	0.7
2014-07-02 00:30:00	0.204458	0.153294	0.125770	0.094591	0.077997	0.067955	0.073124	0.071050	0.082804	0.143680	...	0.753136	0.6
2014-07-02 01:00:00	0.153294	0.125770	0.094591	0.077997	0.067955	0.073124	0.071050	0.082804	0.143680	0.214862	...	0.671452	0.7
2014-07-02 01:30:00	0.125770	0.094591	0.077997	0.067955	0.073124	0.071050	0.082804	0.143680	0.214862	0.363448	...	0.770454	0.8

5 rows × 48 columns

Sliding Window in pandas

We can wrap this approach in a function:

```python
def sliding_window_1D(data, wlen):
    m = len(data)
    lc = [data.iloc[i:m-wlen+i+1] for i in range(0, wlen)]
    wdata = np.hstack(lc)
    wdata = pd.DataFrame(index=data.index[wlen-1:],
            data=wdata, columns=range(wlen))
    return wdata
```

```python
In [9]: wdata = nab.sliding_window_1D(data, wlen=wlen)
```

- This is available in the (updated)) `nab` module
- The function works for univariate data

Considerations

Some considerations and take-home messages:

It's very common to use sliding windows with time series

- In fact, it's one of their more recognizable peculiarities

Applying a time window with `rolling` in pandas is quite easy

- ...But building the result by column is faster!
- Speed can be extremely important in Data Science tasks
    - At training time, it make exploring ideas more convenient
    - At deployment time, there may me latency constraints

The approaches we have discussed work for univariate series:

- We will see how to handle multivariate time series later in the course