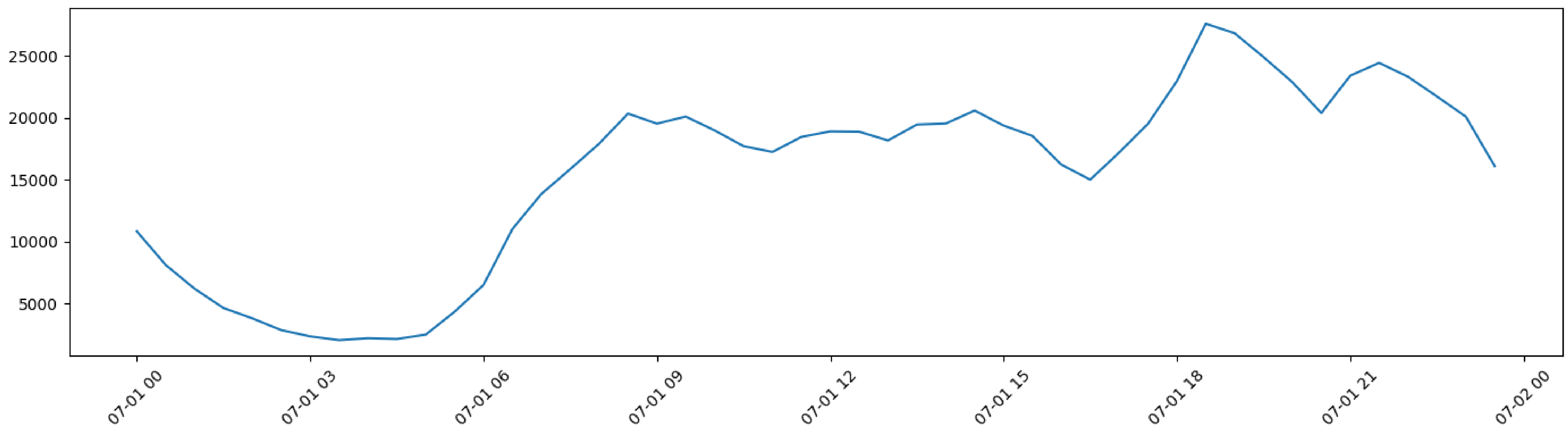
Sliding Windows

Temporal Correlations

Let's have a closer look at our time series

In [2]:
```
nab.plot_series(data.iloc[:48], figsize=figsize)
```



- Nearby points tend to have similar values
- ...Meaning they are correlated

Determine the Period

How can we study such correlation?

A useful tool: autocorrelation plots

- Consider a range of possible lags
- For each lag value l:
  - Make a copy of the series and shift it by l time steps
  - Compute the Pearson Correlation Coefficient with the original series
- Plot the correlation coefficients over the lag values

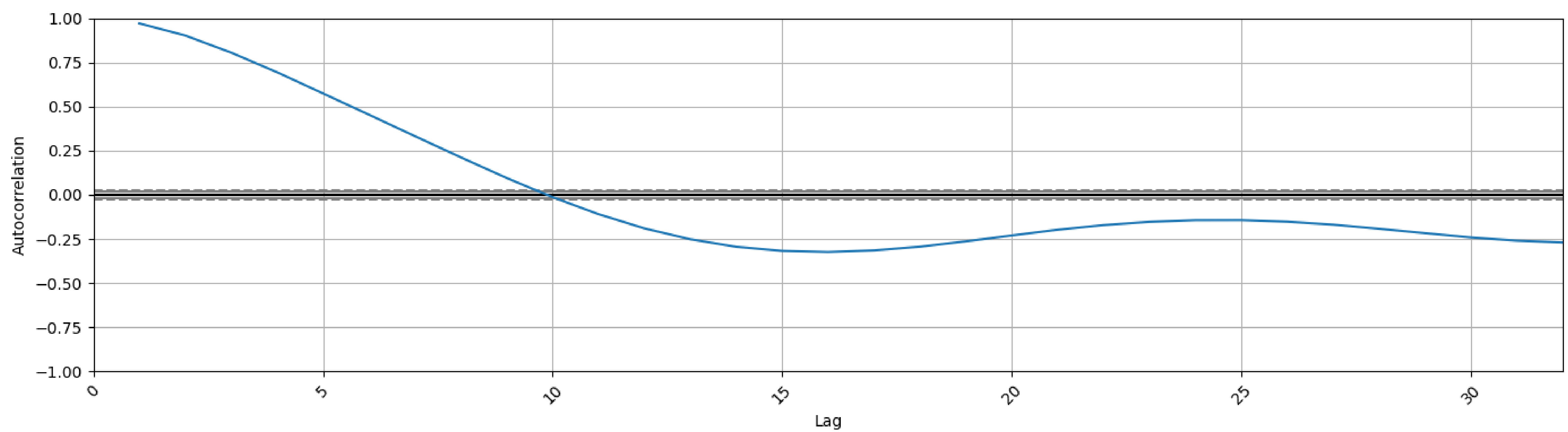
Then we look at the resulting plot:

- Where the curve is far from zero, there is a significant correlation
- Where it gets close to zero, no significant correlation exists

Temporal Correlations

Let's have a look at our plot

In [3]:
```
nab.plot_autocorrelation(data, max_lag=32, 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

How can we take advantage of them?

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

How can we take advantage of them?

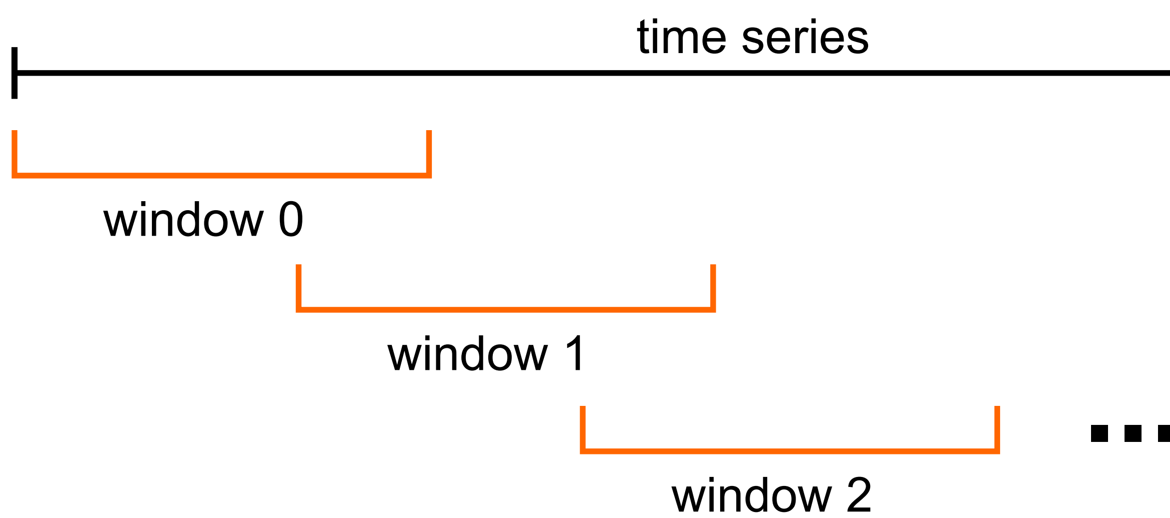
For example, rather then feeding our model with individual observations

We can use sequences of observations as input

- This is a very common approach in time series
- ...And in many cases it's a good idea

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

	$\mathbf{s_0}$	$\mathbf{s_1}$	$\dots$	$\mathbf{s_{w-1}}$
$\mathbf{t_{w-1}}$	x_0	x_1	$\dots$	x_{w-1}
$\mathbf{t_w}$	x_1	x_2	$\dots$	x_w
$\mathbf{t_{w+1}}$	x_2	x_3	$\dots$	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 [4]: wlen = 10
        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    10844
Name: value, dtype: int64
timestamp
2014-07-01 00:00:00    10844
2014-07-01 00:30:00     8127
Name: value, dtype: int64
timestamp
2014-07-01 00:00:00    10844
2014-07-01 00:30:00     8127
2014-07-01 01:00:00     6210
Name: value, dtype: int64
```

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, and we wrap everything in a `DataFrame`

```
In [5]: %%time
        rows = []
        for i, w in enumerate(data['value'].rolling(wlen)):
            if i >= wlen-1: rows.append(w.values)

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

```
CPU times: user 449 ms, sys: 5.18 ms, total: 455 ms
Wall time: 454 ms
```

- 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:

	$\mathbf{s_0}$	$\mathbf{s_1}$	$\dots$	$\mathbf{s_{w-1}}$
$\mathbf{t_{w-1}}$	x_0	x_1	$\dots$	x_{w-1}
$\mathbf{t_w}$	x_1	x_2	$\dots$	x_w
$\mathbf{t_{w+1}}$	x_2	x_3	$\dots$	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 [6]: 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  10844
2014-07-01 00:30:00   8127
2014-07-01 01:00:00   6210
                     value
timestamp
2014-07-01 00:30:00   8127
2014-07-01 01:00:00   6210
2014-07-01 01:30:00   4656
```

- `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 [7]:

```
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=range(wlen), data=lc)
wdata.head()
```

Out[7]:

timestamp	0	1	2	3	4	5	6	7	8	9
2014-07-01 04:30:00	10844	8127	6210	4656	3820	2873	2369	2064	2221	2158
2014-07-01 05:00:00	8127	6210	4656	3820	2873	2369	2064	2221	2158	2515
2014-07-01 05:30:00	6210	4656	3820	2873	2369	2064	2221	2158	2515	4364
2014-07-01 06:00:00	4656	3820	2873	2369	2064	2221	2158	2515	4364	6526
2014-07-01 06:30:00	3820	2873	2369	2064	2221	2158	2515	4364	6526	11039

Sliding Window in pandas

We can wrap this approach in a function:

```
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
```

```
In [8]: %%time
        wdata = nab.sliding_window_1D(data, wlen=wlen)

CPU times: user 1.71 ms, sys: 0 ns, total: 1.71 ms
Wall time: 1.25 ms
```

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