**How to Convert a Time Series to a Supervised Learning Problem in Python**

by [**Jason Brownlee**](https://machinelearningmastery.com/author/jasonb/) on May 8, 2017 in [**Time Series**](https://machinelearningmastery.com/category/time-series/)

Machine learning methods like deep learning can be used for time series forecasting.

Before machine learning can be used, time series forecasting problems must be re-framed as supervised learning problems. From a sequence to pairs of input and output sequences.

In this tutorial, you will discover how to transform univariate and multivariate time series forecasting problems into supervised learning problems for use with machine learning algorithms.

After completing this tutorial, you will know:

* How to develop a function to transform a time series dataset into a supervised learning dataset.
* How to transform univariate time series data for machine learning.
* How to transform multivariate time series data for machine learning.

Let’s get started.



How to Convert a Time Series to a Supervised Learning Problem in Python  
Photo by [Quim Gil](https://www.flickr.com/photos/quimgil/8490510169/), some rights reserved.

**Time Series vs Supervised Learning**

Before we get started, let’s take a moment to better understand the form of time series and supervised learning data.

A time series is a sequence of numbers that are ordered by a time index. This can be thought of as a list or column of ordered values.

For example:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 0  1  2  3  4  5  6  7  8  9 |

A supervised learning problem is comprised of input patterns (*X*) and output patterns (*y*), such that an algorithm can learn how to predict the output patterns from the input patterns.

For example:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | X, y  1 2  2, 3  3, 4  4, 5  5, 6  6, 7  7, 8  8, 9 |

For more on this topic, see the post:

* [Time Series Forecasting as Supervised Learning](http://machinelearningmastery.com/time-series-forecasting-supervised-learning/)

**Pandas shift() Function**

A key function to help transform time series data into a supervised learning problem is the Pandas [shift()](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shift.html) function.

Given a DataFrame, the *shift()* function can be used to create copies of columns that are pushed forward (rows of NaN values added to the front) or pulled back (rows of NaN values added to the end).

This is the behavior required to create columns of lag observations as well as columns of forecast observations for a time series dataset in a supervised learning format.

Let’s look at some examples of the shift function in action.

We can define a mock time series dataset as a sequence of 10 numbers, in this case a single column in a DataFrame as follows:



|  |  |
| --- | --- |
| 1  2  3  4 | from pandas import DataFrame  df = DataFrame()  df['t'] = [x for x in range(10)]  print(df) |

Running the example prints the time series data with the row indices for each observation.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | t  0  0  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 |

We can shift all the observations down by one time step by inserting one new row at the top. Because the new row has no data, we can use NaN to represent “no data”.

The shift function can do this for us and we can insert this shifted column next to our original series.



|  |  |
| --- | --- |
| 1  2  3  4  5 | from pandas import DataFrame  df = DataFrame()  df['t'] = [x for x in range(10)]  df['t-1'] = df['t'].shift(1)  print(df) |

Running the example gives us two columns in the dataset. The first with the original observations and a new shifted column.

We can see that shifting the series forward one time step gives us a primitive supervised learning problem, although with *X* and *y* in the wrong order. Ignore the column of row labels. The first row would have to be discarded because of the NaN value. The second row shows the input value of 0.0 in the second column (input or *X*) and the value of 1 in the first column (output or *y*).



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | t  t-1  0  0  NaN  1  1  0.0  2  2  1.0  3  3  2.0  4  4  3.0  5  5  4.0  6  6  5.0  7  7  6.0  8  8  7.0  9  9  8.0 |

We can see that if we can repeat this process with shifts of 2, 3, and more, how we could create long input sequences (*X*) that can be used to forecast an output value (*y*).

The shift operator can also accept a negative integer value. This has the effect of pulling the observations up by inserting new rows at the end. Below is an example:



|  |  |
| --- | --- |
| 1  2  3  4  5 | from pandas import DataFrame  df = DataFrame()  df['t'] = [x for x in range(10)]  df['t+1'] = df['t'].shift(-1)  print(df) |

Running the example shows a new column with a NaN value as the last value.

We can see that the forecast column can be taken as an input (*X*) and the second as an output value (*y*). That is the input value of 0 can be used to forecast the output value of 1.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | t  t+1  0  0  1.0  1  1  2.0  2  2  3.0  3  3  4.0  4  4  5.0  5  5  6.0  6  6  7.0  7  7  8.0  8  8  9.0  9  9  NaN |

Technically, in time series forecasting terminology the current time (*t*) and future times (*t+1*, *t+n*) are forecast times and past observations (*t-1*, *t-n*) are used to make forecasts.

We can see how positive and negative shifts can be used to create a new DataFrame from a time series with sequences of input and output patterns for a supervised learning problem.

This permits not only classical *X -> y* prediction, but also *X -> Y* where both input and output can be sequences.

Further, the shift function also works on so-called multivariate time series problems. That is where instead of having one set of observations for a time series, we have multiple (e.g. temperature and pressure). All variates in the time series can be shifted forward or backward to create multivariate input and output sequences. We will explore this more later in the tutorial.

**The series\_to\_supervised() Function**

We can use the *shift()* function in Pandas to automatically create new framings of time series problems given the desired length of input and output sequences.

This would be a useful tool as it would allow us to explore different framings of a time series problem with machine learning algorithms to see which might result in better performing models.

In this section, we will define a new Python function named *series\_to\_supervised()* that takes a univariate or multivariate time series and frames it as a supervised learning dataset.

The function takes four arguments:

* **data**: Sequence of observations as a list or 2D NumPy array. Required.
* **n\_in**: Number of lag observations as input (*X*). Values may be between [1..len(data)] Optional. Defaults to 1.
* **n\_out**: Number of observations as output (*y*). Values may be between [0..len(data)-1]. Optional. Defaults to 1.
* **dropnan**: Boolean whether or not to drop rows with NaN values. Optional. Defaults to True.

The function returns a single value:

* **return**: Pandas DataFrame of series framed for supervised learning.

The new dataset is constructed as a DataFrame, with each column suitably named both by variable number and time step. This allows you to design a variety of different time step sequence type forecasting problems from a given univariate or multivariate time series.

Once the DataFrame is returned, you can decide how to split the rows of the returned DataFrame into *X* and *y* components for supervised learning any way you wish.

The function is defined with default parameters so that if you call it with just your data, it will construct a DataFrame with *t-1* as *X* and *t* as *y*.

The function is confirmed to be compatible with Python 2 and Python 3.

The complete function is listed below, including function comments.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | from pandas import DataFrame  from pandas import concat    def series\_to\_supervised(data, n\_in=1, n\_out=1, dropnan=True):  """  Frame a time series as a supervised learning dataset.  Arguments:  data: Sequence of observations as a list or NumPy array.  n\_in: Number of lag observations as input (X).  n\_out: Number of observations as output (y).  dropnan: Boolean whether or not to drop rows with NaN values.  Returns:  Pandas DataFrame of series framed for supervised learning.  """  n\_vars = 1 if type(data) is list else data.shape[1]  df = DataFrame(data)  cols, names = list(), list()  # input sequence (t-n, ... t-1)  for i in range(n\_in, 0, -1):  cols.append(df.shift(i))  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n\_vars)]  # forecast sequence (t, t+1, ... t+n)  for i in range(0, n\_out):  cols.append(df.shift(-i))  if i == 0:  names += [('var%d(t)' % (j+1)) for j in range(n\_vars)]  else:  names += [('var%d(t+%d)' % (j+1, i)) for j in range(n\_vars)]  # put it all together  agg = concat(cols, axis=1)  agg.columns = names  # drop rows with NaN values  if dropnan:  agg.dropna(inplace=True)  return agg |

Can you see obvious ways to make the function more robust or more readable?  
Please let me know in the comments below.

Now that we have the whole function, we can explore how it may be used.

**One-Step Univariate Forecasting**

It is standard practice in time series forecasting to use lagged observations (e.g. t-1) as input variables to forecast the current time step (t).

This is called one-step forecasting.

The example below demonstrates a one lag time step (t-1) to predict the current time step (t).



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | from pandas import DataFrame  from pandas import concat    def series\_to\_supervised(data, n\_in=1, n\_out=1, dropnan=True):  """  Frame a time series as a supervised learning dataset.  Arguments:  data: Sequence of observations as a list or NumPy array.  n\_in: Number of lag observations as input (X).  n\_out: Number of observations as output (y).  dropnan: Boolean whether or not to drop rows with NaN values.  Returns:  Pandas DataFrame of series framed for supervised learning.  """  n\_vars = 1 if type(data) is list else data.shape[1]  df = DataFrame(data)  cols, names = list(), list()  # input sequence (t-n, ... t-1)  for i in range(n\_in, 0, -1):  cols.append(df.shift(i))  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n\_vars)]  # forecast sequence (t, t+1, ... t+n)  for i in range(0, n\_out):  cols.append(df.shift(-i))  if i == 0:  names += [('var%d(t)' % (j+1)) for j in range(n\_vars)]  else:  names += [('var%d(t+%d)' % (j+1, i)) for j in range(n\_vars)]  # put it all together  agg = concat(cols, axis=1)  agg.columns = names  # drop rows with NaN values  if dropnan:  agg.dropna(inplace=True)  return agg      values = [x for x in range(10)]  data = series\_to\_supervised(values)  print(data) |

Running the example prints the output of the reframed time series.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var1(t-1)  var1(t)  1        0.0        1  2        1.0        2  3        2.0        3  4        3.0        4  5        4.0        5  6        5.0        6  7        6.0        7  8        7.0        8  9        8.0        9 |

We can see that the observations are named “*var1*” and that the input observation is suitably named (t-1) and the output time step is named (t).

We can also see that rows with NaN values have been automatically removed from the DataFrame.

We can repeat this example with an arbitrary number length input sequence, such as 3. This can be done by specifying the length of the input sequence as an argument; for example:



|  |  |
| --- | --- |
| 1 | data = series\_to\_supervised(values, 3) |

The complete example is listed below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | from pandas import DataFrame  from pandas import concat    def series\_to\_supervised(data, n\_in=1, n\_out=1, dropnan=True):  """  Frame a time series as a supervised learning dataset.  Arguments:  data: Sequence of observations as a list or NumPy array.  n\_in: Number of lag observations as input (X).  n\_out: Number of observations as output (y).  dropnan: Boolean whether or not to drop rows with NaN values.  Returns:  Pandas DataFrame of series framed for supervised learning.  """  n\_vars = 1 if type(data) is list else data.shape[1]  df = DataFrame(data)  cols, names = list(), list()  # input sequence (t-n, ... t-1)  for i in range(n\_in, 0, -1):  cols.append(df.shift(i))  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n\_vars)]  # forecast sequence (t, t+1, ... t+n)  for i in range(0, n\_out):  cols.append(df.shift(-i))  if i == 0:  names += [('var%d(t)' % (j+1)) for j in range(n\_vars)]  else:  names += [('var%d(t+%d)' % (j+1, i)) for j in range(n\_vars)]  # put it all together  agg = concat(cols, axis=1)  agg.columns = names  # drop rows with NaN values  if dropnan:  agg.dropna(inplace=True)  return agg      values = [x for x in range(10)]  data = series\_to\_supervised(values, 3)  print(data) |

Again, running the example prints the reframed series. We can see that the input sequence is in the correct left-to-right order with the output variable to be predicted on the far right.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var1(t-3)  var1(t-2)  var1(t-1)  var1(t)  3        0.0        1.0        2.0        3  4        1.0        2.0        3.0        4  5        2.0        3.0        4.0        5  6        3.0        4.0        5.0        6  7        4.0        5.0        6.0        7  8        5.0        6.0        7.0        8  9        6.0        7.0        8.0        9 |

**Multi-Step or Sequence Forecasting**

A different type of forecasting problem is using past observations to forecast a sequence of future observations.

This may be called sequence forecasting or multi-step forecasting.

We can frame a time series for sequence forecasting by specifying another argument. For example, we could frame a forecast problem with an input sequence of 2 past observations to forecast 2 future observations as follows:



|  |  |
| --- | --- |
| 1 | data = series\_to\_supervised(values, 2, 2) |

The complete example is listed below:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | from pandas import DataFrame  from pandas import concat    def series\_to\_supervised(data, n\_in=1, n\_out=1, dropnan=True):  """  Frame a time series as a supervised learning dataset.  Arguments:  data: Sequence of observations as a list or NumPy array.  n\_in: Number of lag observations as input (X).  n\_out: Number of observations as output (y).  dropnan: Boolean whether or not to drop rows with NaN values.  Returns:  Pandas DataFrame of series framed for supervised learning.  """  n\_vars = 1 if type(data) is list else data.shape[1]  df = DataFrame(data)  cols, names = list(), list()  # input sequence (t-n, ... t-1)  for i in range(n\_in, 0, -1):  cols.append(df.shift(i))  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n\_vars)]  # forecast sequence (t, t+1, ... t+n)  for i in range(0, n\_out):  cols.append(df.shift(-i))  if i == 0:  names += [('var%d(t)' % (j+1)) for j in range(n\_vars)]  else:  names += [('var%d(t+%d)' % (j+1, i)) for j in range(n\_vars)]  # put it all together  agg = concat(cols, axis=1)  agg.columns = names  # drop rows with NaN values  if dropnan:  agg.dropna(inplace=True)  return agg      values = [x for x in range(10)]  data = series\_to\_supervised(values, 2, 2)  print(data) |

Running the example shows the differentiation of input (t-n) and output (t+n) variables with the current observation (t) considered an output.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var1(t-2)  var1(t-1)  var1(t)  var1(t+1)  2        0.0        1.0        2        3.0  3        1.0        2.0        3        4.0  4        2.0        3.0        4        5.0  5        3.0        4.0        5        6.0  6        4.0        5.0        6        7.0  7        5.0        6.0        7        8.0  8        6.0        7.0        8        9.0 |

**Multivariate Forecasting**

Another important type of time series is called multivariate time series.

This is where we may have observations of multiple different measures and an interest in forecasting one or more of them.

For example, we may have two sets of time series observations obs1 and obs2 and we wish to forecast one or both of these.

We can call *series\_to\_supervised()* in exactly the same way.

For example:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | from pandas import DataFrame  from pandas import concat    def series\_to\_supervised(data, n\_in=1, n\_out=1, dropnan=True):  """  Frame a time series as a supervised learning dataset.  Arguments:  data: Sequence of observations as a list or NumPy array.  n\_in: Number of lag observations as input (X).  n\_out: Number of observations as output (y).  dropnan: Boolean whether or not to drop rows with NaN values.  Returns:  Pandas DataFrame of series framed for supervised learning.  """  n\_vars = 1 if type(data) is list else data.shape[1]  df = DataFrame(data)  cols, names = list(), list()  # input sequence (t-n, ... t-1)  for i in range(n\_in, 0, -1):  cols.append(df.shift(i))  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n\_vars)]  # forecast sequence (t, t+1, ... t+n)  for i in range(0, n\_out):  cols.append(df.shift(-i))  if i == 0:  names += [('var%d(t)' % (j+1)) for j in range(n\_vars)]  else:  names += [('var%d(t+%d)' % (j+1, i)) for j in range(n\_vars)]  # put it all together  agg = concat(cols, axis=1)  agg.columns = names  # drop rows with NaN values  if dropnan:  agg.dropna(inplace=True)  return agg      raw = DataFrame()  raw['ob1'] = [x for x in range(10)]  raw['ob2'] = [x for x in range(50, 60)]  values = raw.values  data = series\_to\_supervised(values)  print(data) |

Running the example prints the new framing of the data, showing an input pattern with one time step for both variables and an output pattern of one time step for both variables.

Again, depending on the specifics of the problem, the division of columns into *X* and *Y*components can be chosen arbitrarily, such as if the current observation of *var1* was also provided as input and only *var2* was to be predicted.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | var1(t-1)  var2(t-1)  var1(t)  var2(t)  1        0.0       50.0        1       51  2        1.0       51.0        2       52  3        2.0       52.0        3       53  4        3.0       53.0        4       54  5        4.0       54.0        5       55  6        5.0       55.0        6       56  7        6.0       56.0        7       57  8        7.0       57.0        8       58  9        8.0       58.0        9       59 |

You can see how this may be easily used for sequence forecasting with multivariate time series by specifying the length of the input and output sequences as above.

For example, below is an example of a reframing with 1 time step as input and 2 time steps as forecast sequence.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | from pandas import DataFrame  from pandas import concat    def series\_to\_supervised(data, n\_in=1, n\_out=1, dropnan=True):  """  Frame a time series as a supervised learning dataset.  Arguments:  data: Sequence of observations as a list or NumPy array.  n\_in: Number of lag observations as input (X).  n\_out: Number of observations as output (y).  dropnan: Boolean whether or not to drop rows with NaN values.  Returns:  Pandas DataFrame of series framed for supervised learning.  """  n\_vars = 1 if type(data) is list else data.shape[1]  df = DataFrame(data)  cols, names = list(), list()  # input sequence (t-n, ... t-1)  for i in range(n\_in, 0, -1):  cols.append(df.shift(i))  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n\_vars)]  # forecast sequence (t, t+1, ... t+n)  for i in range(0, n\_out):  cols.append(df.shift(-i))  if i == 0:  names += [('var%d(t)' % (j+1)) for j in range(n\_vars)]  else:  names += [('var%d(t+%d)' % (j+1, i)) for j in range(n\_vars)]  # put it all together  agg = concat(cols, axis=1)  agg.columns = names  # drop rows with NaN values  if dropnan:  agg.dropna(inplace=True)  return agg      raw = DataFrame()  raw['ob1'] = [x for x in range(10)]  raw['ob2'] = [x for x in range(50, 60)]  values = raw.values  data = series\_to\_supervised(values, 1, 2)  print(data) |

Running the example shows the large reframed DataFrame.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var1(t-1)  var2(t-1)  var1(t)  var2(t)  var1(t+1)  var2(t+1)  1        0.0       50.0        1       51        2.0       52.0  2        1.0       51.0        2       52        3.0       53.0  3        2.0       52.0        3       53        4.0       54.0  4        3.0       53.0        4       54        5.0       55.0  5        4.0       54.0        5       55        6.0       56.0  6        5.0       55.0        6       56        7.0       57.0  7        6.0       56.0        7       57        8.0       58.0  8        7.0       57.0        8       58        9.0       59.0 |

Experiment with your own dataset and try multiple different framings to see what works best.