**UNIT-5**

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, or physics. Anything that is observed or measured at many points in time forms a time series. Many time series are *fixed frequency*, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be *irregular* without a fixed unit or time or offset between units. How you mark and refer to time series data depends on the application and you may have one of the following:

* *Timestamps*, specific instants in time
* Fixed *periods*, such as the month January 2007 or the full year 2010
* *Intervals* of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals
* Experiment or elapsed time; each timestamp is a measure of time relative to a particular start time. For example, the diameter of a cookie baking each second since being placed in the oven.

In this chapter, I am mainly concerned with time series in the first 3 categories, though many of the techniques can be applied to experimental time series where the index may be an integer or floating point number indicating elapsed time from the start of the experiment. The simplest and most widely used kind of time series are those indexed by timestamp.

pandas provides a standard set of time series tools and data algorithms. With this, you can efficiently work with very large time series and easily slice and dice, aggregate, and resample irregular and fixed frequency time series. As you might guess, many of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.

**Date and Time Data Types and Tools**

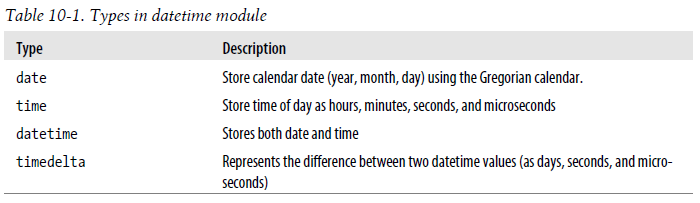
The Python standard library includes data types for date and time data, as well as calendar-related functionality. The datetime, time, and calendar modules are the main places to start. The datetime.datetime type, or simply datetime, is widely used:

## datetime Module:

Python has a module named **datetime** to work with dates and times.

Commonly used classes in the datetime module are:

* date Class
* time Class
* datetime Class
* timedelta Class



## datetime.datetime

The datetime module has a class named datetime class that can contain information from both **date** and **time** objects. It contains year, month, day, hour, minute, second, microseconds.

In [317]: from datetime import datetime

In [318]: now = datetime.now()

In [319]: now

Out[319]: datetime.datetime(2012, 8, 4, 17, 9, 21, 832092)

Here, we have imported **datetime** module using import datetime statement. One of the classes defined in the datetime module is datetime class. We then used now() method to create a datetime object containing the current local date and time.

In [320]: now.year, now.month, now.day

Out[320]: (2012, 8, 4)

**datetime.date Class**

You can instantiate date objects from the date class. A date object represents a date (year, month and day).

### Example : Date object to represent a date

import datetime

d = datetime.date(2019, 4, 13)

print(d)

**Output**:

2019-04-13

If you are wondering, date() in the above example is a constructor of the date class. The constructor takes three arguments: year, month and day.

### Example : Get current date

You can create a date object containing the current date by using a classmethod named today(). Here's how:

from datetime import date

today = date.today()

print("Current date =", today)

**Output**:

Current date = 2019-04-13

## datetime.time

A time object instantiated from the time class represents the local time.

### Example 8: Print hour, minute, second and microsecond

Once you create a time object, you can easily print its attributes such as *hour*, *minute* etc.

from datetime import time

a = time(11, 34, 56)

print("time =", a)

print("hour =", a.hour)

print("minute =", a.minute)

print("second =", a.second)

print("microsecond =", a.microsecond)

**Output**:

Time = 11:34:56.0

hour = 11

minute = 34

second = 56

microsecond = 0

## datetime.timedelta

A timedelta object represents the difference between two dates or times.

datetime stores both the date and time down to the microsecond. datetime.timedelta represents the temporal difference between two datetime objects:

delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

delta

**Output:**

datetime.timedelta(926, 56700)

In [323]: delta.days

Out[323]: 926

In [324]: delta.seconds

Out[324]: 56700

You can add (or subtract) a timedelta or multiple thereof to a datetime object to yield a new shifted object:

In [325]: from datetime import timedelta

In [326]: start = datetime(2011, 1, 7)

In [327]: start + timedelta(12)

Out[327]: datetime.datetime(2011, 1, 19, 0, 0)

In [328]: start - 2 \* timedelta(12)

Out[328]: datetime.datetime(2010, 12, 14, 0, 0)

**Converting between string and datetime**

datetime objects and pandas Timestamp objects, which I’ll introduce later, can be formatted as strings using str or the strftime method, passing a format specification:

In [329]: stamp = datetime(2011, 1, 3)

In [330]: str(stamp)

Out[330]: '2011-01-03 00:00:00'

In [331]: stamp.strftime('%Y-%m-%d')

Out[331]: '2011-01-03'

See Table 10-2 for a complete list of the format codes. These same format codes can be used to convert strings to dates using datetime.strptime:

In [332]: value = '2011-01-03'

In [333]: datetime.strptime(value, '%Y-%m-%d')

Out[333]: datetime.datetime(2011, 1, 3, 0, 0)

In [334]: datestrs = ['7/6/2011', '8/6/2011']

In [335]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]

Out[335]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]

datetime.strptime is the best way to parse a date with a known format. However, it can be a bit annoying to have to write a format spec each time, especially for common date formats. In this case, you can use the **parser.parse** method in the third party dateutil package:

In [336]: from dateutil.parser import parse

In [337]: parse('2011-01-03')

Out[337]: datetime.datetime(2011, 1, 3, 0, 0)

dateutil is capable of parsing almost any human-intelligible date representation:

In [338]: parse('Jan 31, 1997 10:45 PM')

Out[338]: datetime.datetime(1997, 1, 31, 22, 45)

In international locales, day appearing before month is very common, so you can pass dayfirst=True to indicate this:

In [339]: parse('6/12/2011', dayfirst=True)

Out[339]: datetime.datetime(2011, 12, 6, 0, 0)

pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame. The to\_datetime method parses many different kinds of date representations. Standard date formats like ISO8601 can be parsed very quickly.

In [340]: datestrs

Out[340]: ['7/6/2011', '8/6/2011']

In [341]: pd.to\_datetime(datestrs)

Out[341]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2011-07-06 00:00:00, 2011-08-06 00:00:00]

Length: 2, Freq: None, Timezone: None

It also handles values that should be considered missing (None, empty string, etc.):

In [342]: idx = pd.to\_datetime(datestrs + [None])

In [343]: idx

Out[343]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2011-07-06 00:00:00, ..., NaT]

Length: 3, Freq: None, Timezone: None

In [344]: idx[2]

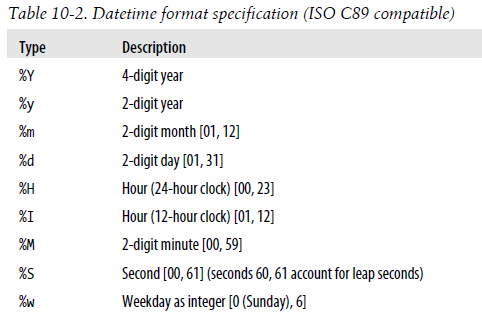
Out[344]: NaT

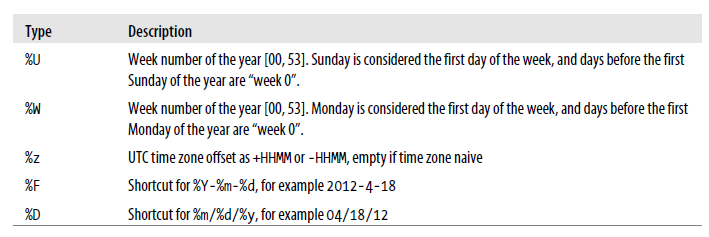
In [345]: pd.isnull(idx)

Out[345]: array([False, False, True], dtype=bool)

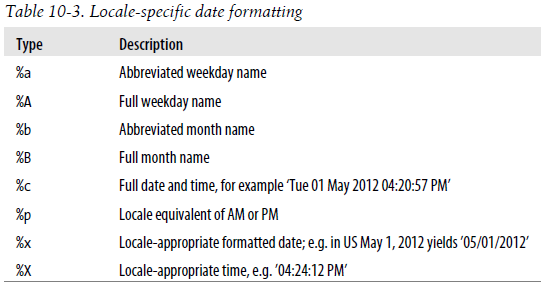
NaT (Not a Time) is pandas’s NA value for timestamp data.

Note: dateutil.parser is a useful, but not perfect tool. Notably, it will recognize some strings as dates that you might prefer that it didn’t, like '42' will be parsed as the year 2042 with today’s calendar date.





datetime objects also have a number of locale-specific formatting options for systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems.



**Time Series Basics**

The most basic kind of time series object in pandas is a Series indexed by timestamps, which is often represented external to pandas as Python strings or datetime objects:

In [346]: from datetime import datetime

In [347]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5), datetime(2011, 1, 7),

.....: datetime(2011, 1, 8), datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [348]: ts = Series(np.random.randn(6), index=dates)

In [349]: ts

Out[349]:

2011-01-02 0.690002

2011-01-05 1.001543

2011-01-07 -0.503087

2011-01-08 -0.622274

2011-01-10 -0.921169

2011-01-12 -0.726213

Under the hood, these datetime objects have been put in a DatetimeIndex, and the variable ts is now of type TimeSeries:

In [350]: type(ts)

Out[350]: pandas.core.series.TimeSeries

In [351]: ts.index

Out[351]:

<class 'pandas.tseries.index.DatetimeIndex'>

[2011-01-02 00:00:00, ..., 2011-01-12 00:00:00]

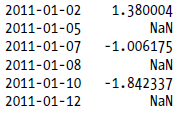
Length: 6, Freq: None, Timezone: None

**NOTE:** It’s not necessary to use the TimeSeries constructor explicitly; when creating a Series with a DatetimeIndex, pandas knows that the object is a time series.

Like other Series, arithmetic operations between differently-indexed time series automatically

align on the dates:

In [352]: ts + ts[::2]



pandas stores timestamps using NumPy’s datetime64 data type at the nanosecond resolution:

In [353]: ts.index.dtype

Out[353]: dtype('datetime64[ns]')

Scalar values from a DatetimeIndex are pandas Timestamp objects

In [354]: stamp = ts.index[0]

In [355]: stamp

Out[355]: <Timestamp: 2011-01-02 00:00:00>

A Timestamp can be substituted anywhere you would use a datetime object. Additionally, it can store frequency information (if any) and understands how to do time zone conversions and other kinds of manipulations. More on both of these things later.

**Indexing, Selection, Subsetting**

TimeSeries is a subclass of Series and thus behaves in the same way with regard to indexing and selecting data based on label:

In [356]: stamp = ts.index[2]

In [357]: ts[stamp]

Out[357]: -0.50308739136034464

As a convenience, you can also pass a string that is interpretable as a date:

In [358]: ts['1/10/2011']

Out[358]: -0.92116860801301081

In [359]: ts['20110110']

Out[359]: -0.92116860801301081

# pandas.date\_range() method :

**pandas.date\_range()** is one of the general functions in Pandas which is used to return a fixed frequency DatetimeIndex.

pandas.date\_range(*start=None*, *end=None*, *periods=None*, *freq=None*, *tz=None*)

**Parameters:**  
**start :** Left bound for generating dates.  
**end :** Right bound for generating dates.  
**periods :** Number of periods to generate.  
**freq :** Frequency strings can have multiples, e.g. ‘5H’.

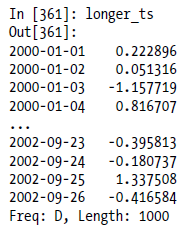
**tz :** Time zone name for returning localized DatetimeIndex. By default, the resulting DatetimeIndex is timezone-naive.

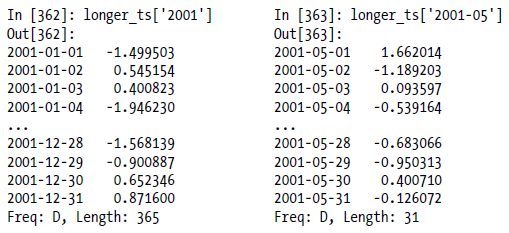
**Examples:**

For longer time series, a year or only a year and month can be passed to easily select slices of data:

In [360]: longer\_ts = Series(np.random.randn(1000),

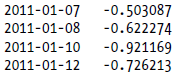
.....: index=pd.date\_range('1/1/2000', periods=1000))



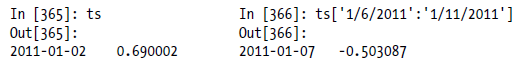


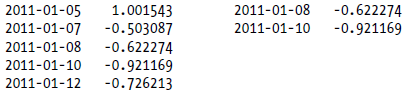
Slicing with dates works just like with a regular Series:

In [364]: ts[datetime(2011, 1, 7):]



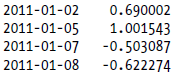
Because most time series data is ordered chronologically, you can slice with timestamps not contained in a time series to perform a range query:





As before you can pass either a string date, datetime, or Timestamp. Remember that slicing in this manner produces views on the source time series just like slicing NumPy arrays. There is an equivalent instance method truncate which slices a TimeSeries between two dates:

In [367]: ts.truncate(after='1/9/2011')



All of the above holds true for DataFrame as well, indexing on its rows:

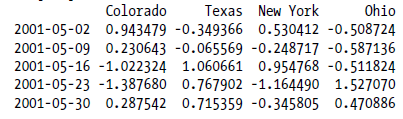
In [368]: dates = pd.date\_range('1/1/2000', periods=100, freq='W-WED')

In [369]: long\_df = DataFrame(np.random.randn(100, 4),

.....: index=dates,

.....: columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [370]: long\_df.ix['5-2001']



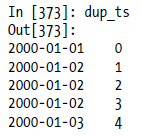
**Time Series with Duplicate Indices**

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

In [371]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',

.....: '1/3/2000'])

In [372]: dup\_ts = Series(np.arange(5), index=dates)

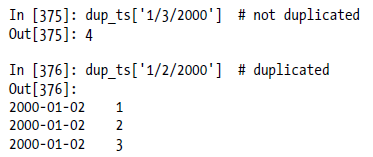


We can tell that the index is not unique by checking its is\_unique property:

In [374]: dup\_ts.index.is\_unique

Out[374]: False

Indexing into this time series will now either produce scalar values or slices depending on whether a timestamp is duplicated:



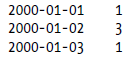
Suppose you wanted to aggregate the data having non-unique timestamps. One way to do this is to use groupby and pass level=0 (the only level of indexing!):

In [377]: grouped = dup\_ts.groupby(level=0)

In [378]: grouped.mean()



grouped.count()



**Time Zone Handling**

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. In particular, daylight savings time (DST) transitions are a common source of complication. As such, many time series users choose to work with time series in *coordinated universal time* or *UTC*, which is the successor to Greenwich Mean Time and is the current international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight savings time and 5 hours the rest of the year.

In Python, time zone information comes from the 3rd party pytz library, which exposes the *Olson database*, a compilation of world time zone information. This is especially important for historical data because the DST transition dates (and even UTC offsets) have been changed numerous times depending on the whims of local governments. In the United States,the DST transition times have been changed many times since 1900!

**What is PYTZ in Python?**

**pytz** brings the Olson tz database into **Python**. This library allows accurate and cross platform timezone calculations using **Python** 2.4 or higher. It also solves the issue of ambiguous times at the end of daylight saving time.

**UTC:**

UTC is the time standard commonly used across the world. The world's timing centers have agreed to keep their time scales closely synchronized - or coordinated - therefore the name Coordinated Universal Time.

In [418]: import pytz

In [419]: pytz.common\_timezones[-5:]

Out[419]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']

To get a time zone object from pytz, use pytz.timezone:

In [420]: tz = pytz.timezone('US/Eastern')

In [421]: tz

Out[421]: <DstTzInfo 'US/Eastern' EST-1 day, 19:00:00 STD>

Methods in pandas will accept either time zone names or these objects. I recommend just using the names.

**Localization and Conversion**

By default, time series in pandas are *time zone naive*.

# pandas.date\_range() method :

**pandas.date\_range()** is one of the general functions in Pandas which is used to return a fixed frequency DatetimeIndex.

pandas.date\_range(*start=None*, *end=None*, *periods=None*, *freq=None*, *tz=None*)

**Parameters:**  
**start :** Left bound for generating dates.  
**end :** Right bound for generating dates.  
**periods :** Number of periods to generate.  
**freq :** Frequency strings can have multiples, e.g. ‘5H’.

**tz :** Time zone name for returning localized DatetimeIndex. By default, the resulting DatetimeIndex is timezone-naive.

Example: Consider the following time series:

rng = pd.date\_range('3/9/2012 9:30', periods=6, freq='D')

ts = Series(np.random.randn(len(rng)), index=rng)

The index’s tz field is None:

In [423]: print(ts.index.tz)

None

Date ranges can be generated with a time zone set:

In [424]: pd.date\_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC') // D= day

Out[424]:

<class 'pandas.tseries.index.DatetimeIndex'>

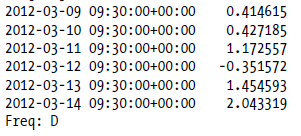
[2012-03-09 09:30:00, ..., 2012-03-18 09:30:00]

Length: 10, Freq: D, Timezone: UTC

Conversion from naive to *localized* is handled by the tz\_localize method:

In [425]: ts\_utc = ts.tz\_localize('UTC')

In [426]: ts\_utc



**Example-2**

|  |
| --- |
| import pandas as pd    per1 = pd.date\_range(start ='1-1-2018',           end ='1-03-2018', freq ='5H') // H = Hours    for val in per1:      print(val) |

**Output:**

