Mastering Oracle+Python, Part 2: Working with Times and Dates

by Przemyslaw Piotrowski

An introduction to Oracle and Python date handling

Published September 2007

See series TOC

Starting with the Python 2.4 release, cx_Oracle handles DATE and TIMESTAMP datatypes natively, mapping values of such columns to Python datetime objects from the datetime module. This offers certain advantages as datetime objects support arithmetic operations in-place. Built-in time zone support and several dedicated modules make Python a real time machine. The transition between Python and Oracle date/time datatypes is completely transparent to developers thanks to cx_Oracle's mapping mechanisms.

Python developers might find Oracle's date arithmetic a bit odd at first, but only with a few tips it becomes completely clear and very reasonable. This part of the series will give you an in-depth understanding of date arithmetic from both Oracle and Python's point of view. Each of them offers rich support for handling date/time datatypes, so it is the programmer's choice which one to rely on. If you tend to put application logic inside the database or whether you prefer to encapsulate date/time operations in the application itself, the seamless integration of Oracle with Python offers you maximum flexibility with limited programming effort.

Oracle
Oracle features top-of-the-line support of time zones and date arithmetic. Basic Oracle datatypes for handling time and date include:

DATE - Date and time information that includes the century, year, month, date, hour, minute and second. Columns of this type support values from January 1, 4712 B.C. up to December 31, 9999.
TIMESTAMP - The granularity of the DATE datatype is to the second. TIMESTAMP fields hold all the information DATE does, plus the fraction of a second at a given precision (up to nine places). The default precision is 6.
TIMESTAMP WITH TIME ZONE - In addition to information stored in a TIMESTAMP column, this variation also includes a time zone offset which is a difference between the local time and UTC (Coordinated Universal Time). The precision properties are same as above.
TIMESTAMP WITH LOCAL TIME ZONE - Contrary to TIMESTAMP WITH TIME ZONE this type doesn't hold time a zone offset in its value, but one that is determined by the user's local session time zone.
Datetimes consist of a number of fields, determined by the datatype's granularity and variant. These fields can be extracted in SQL queries with the EXTRACT statement. For details about the available fields in datatypes and intervals consult the Datatypes section of *Oracle Database SQL Language Reference*. Let's see how this works:

```
SQL> SELECT EXTRACT(YEAR FROM hire_date) FROM employees ORDER BY 1;

EXTRACT(YEARFROMHIRE_DATE)
--------------------------------------------------
                      1987
                      1987
                         ⋮
                      2000
107 rows selected.
```

Using this method, and Oracle's date arithmetic, you can also grab intervals between two dates:

```
SQL> SELECT hire_date, SYSDATE, EXTRACT(YEAR FROM (SYSDATE-hire_date) YEAR TO MONTH) "Years"
  2  FROM employees WHERE ROWNUM <= 5;
HIRE_DATE          SYSDATE                Years
------------------          ------------------              ----------
17-JUN-87          23-FEB-07                 19
21-SEP-89          23-FEB-07                 17
13-JAN-93          23-FEB-07                 14
03-JAN-90          23-FEB-07                 17
21-MAY-91          23-FEB-07                 15
5 rows selected.
```

Another datatype involved in date operations is INTERVAL, which represents a period of time. At the time of writing Python doesn't support the INTERVAL datatype returned as part of a query; the only way to do that is by extracting the required information from the interval with EXTRACT. Queries involving intervals that return a TIMESTAMP type work fine, though.
There are two variants of INTERVAL type:

INTERVAL YEAR TO MONTH - stores the information about a number of years and number of months. Year precision could be specified manually. The default is (INTERVAL YEAR(2) TO MONTH).
Error—Base class for all of the exceptions mentioned here except for Warning
INTERVAL DAY TO SECOND - In cases where more precision is required, this type holds information about days, hours, minutes, and seconds as a period of time. Both the day and second precision can be stated explicitly with ranges 0 to 9. The default is INTERVAL DAY(2) TO SECOND(6)

SYSDATE+1 Equals Tomorrow

Now we'll examine how Oracle solves date arithmetic. When working with datetime columns, Oracle considers 1 to be a single day. This approach turns to be very intuitive indeed. If you want to work with smaller units then you need to divide: a minute is 1/1440 because there are 60*24 minutes in a single day, an hour is 1/24, a second is 1/86400 and so on.

Fifteen minutes from now would be queried as SELECT SYSDATE+15/1440 FROM dual.

Formatting Dates

Oracle natively displays dates as strings, as shown by the examples above. The formatting depends on parameters inherited from the environment or set explicitly. To see the formatting parameters of your database use this query:

```
SQL> SELECT * FROM v$nls_parameters WHERE REGEXP_LIKE(parameter, 'NLS_(DATE|TIME).*');

PARAMETER                      VALUE
------------------------------              ------------------------------------
```

```
NLS_DATE_FORMAT                RR/MM/DD
NLS_DATE_LANGUAGE              POLISH
NLS_TIME_FORMAT                HH24:MI:SSXFF
NLS_TIMESTAMP_FORMAT           RR/MM/DD HH24:MI:SSXFF
NLS_TIME_TZ_FORMAT             HH24:MI:SSXFF TZR
NLS_TIMESTAMP_TZ_FORMAT        RR/MM/DD HH24:MI:SSXFF TZR


6 rows selected.
```

A set of Oracle functions (TO_DATE, TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_YMINTERVAL, TO_DSINTERVAL) is available to the developer for converting character values into datetimes. The TO_CHAR function is used in the opposite direction. Notice that these conversions aren't often necessary for transition between Oracle and Python because we are working on types convertible in both directions. Still, Oracle itself allows great flexibility in formatting datatimes:

```
SQL> SELECT TO_CHAR(TO_DATE('04-2007-07', 'DD-YYYY-MM'), 'DD/MM/YYYY') FROM dual;
TO_CHAR(TO_DATE('04-2007-07','DD-YYYY-MM'),'DD/MM/YYYY')
----------
04/07/2007
```

Obtaining current time and time zone information is as easy. Two new formatting models, TZH and TZM, are introduced:

```
SQL> SELECT TO_CHAR(SYSTIMESTAMP, 'HH24:MI TZH:TZM') FROM dual;
TO_CHAR(SYSTIMESTAMP,'HH24:MI TZH:TZM')
------------
16:24 +01:00
```

For a complete list of available formatting models, please refer to the Format Models section of the *Oracle Database SQL Language Reference*, which also includes numerous examples of how to use them. In many situations you might find it useful to set the format model for the current session permanently. This can be achieved with the ALTER SESSION statement:

```
Connected to:
Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production
SQL> SELECT SYSDATE FROM dual;
SYSDATE
--------------
23-FEB-07
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
Session altered.
SQL> SELECT SYSDATE FROM dual;
SYSDATE
-------------------
2007-02-23 17:50:15
```

The process of setting National Language Support (NLS) parameters is described in detail in the *Oracle Database Globalization Support Guide* . *Oracle Database SQL Language Reference* contains more information related to the subject; in particular see Datetime/Interval Arithmetic and Datetime Functions sections to find out more about date arithmetic and built-in date functions.

Python

Python gives you freedom to choose between low- and high-level interfaces when working with times and dates. To leverage the full potential of Python's standard library, we will focus on the datetime module, which is also the foundation for date/time arithmetic. This module holds five core types: date, time, datetime, timedelta, and tzinfo.

```
>>> import datetime
>>> d = datetime.datetime.now()
>>> print d
2007-03-03 16:48:27.734000
>>> print type(d)
<type 'datetime.datetime'>
>>> print d.hour, d.minute, d.second
(16, 48, 27)
```

As presented above, datetime objects are granular to microseconds, exposing a set of attributes corresponding to day, hour, second, and so on. The fastest way in Python to learn more about the attributes and methods an object has is to use the built-in dir() function. Python's standard library is also heavily documented so at any point you can use the help() function to see a brief description of the object—in most cases sufficient for getting started right away.

```
>>> dir(datetime.datetime)
['__add__', '__class__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__ne__', '__new__', '__radd__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rsub__', '__setattr__', '__str__',
 '__sub__', 'astimezone', 'combine', 'ctime', 'date', 'day', 'dst',
 'fromordinal', 'fromtimestamp', 'hour', 'isocalendar', 'isoformat',
 'isoweekday', 'max', 'microsecond', 'min', 'minute', 'month', 'now',
  'replace', 'resolution', 'second', 'strftime', 'strptime', 'time',
   'timetuple', 'timetz', 'today', 'toordinal', 'tzinfo', 'tzname',
    'utcfromtimestamp', 'utcnow', 'utcoffset', 'utctimetuple',
        'weekday', 'year']
>>> help(datetime.datetime.weekday)
Help on method_descriptor:

weekday(...)
    Return the day of the week represented by the date.
    Monday == 0 ... Sunday == 6
```

When only one component of datetime is needed (date or time), you can use date() or time() methods of the datetime.datetime object to return a datetime.date or datetime.time object, respectively.

Python Date Arithmetic

Differences between dates don't need to be calculated by hand because the datetime module already supports such arithmetic with timedelta objects. Time deltas represents durations, which internally store the number of days, seconds and microseconds. Milliseconds, minutes, hours and weeks are the rest of the attributes timedelta exposes. These are calculated from the internal representation. The supported ranges are: -999999999 <= days <= 999999999, 0 <= seconds < 86400 and 0 <= microseconds < 1000000. Notice that timedelta objects are represented by as many fields required. Thus timedelta(hours=1) is output as timedelta(0, 3600) i.e. 0 days and 3600 seconds. There are no milliseconds here because they are not required to represent one hour.

The most frequently used and supported arithmetic operations on datetime/timedelta objects in Python are shown below.

Object type	Operation	Example and Equivalent Result
datetime.timedelta	td2 + td3	timedelta(minutes=10) + timedelta(hours=2) == timedelta(0, 7800)
	td2 - td3	timedelta(weeks=3) - timedelta(hours=72) == timedelta(18)
	td2 * n	timedelta(minutes=5) * 5 == timedelta(0, 1500)
	td2 / n	timedelta(weeks=1) / 7 == timedelta(1)
datetime.date	d2 + td	date(2007, 12, 31) + timedelta(days=1) == date(2008, 1, 1)
	d2 - td	date(2007, 12, 31) - timedelta(weeks=52) == date(2007, 1, 1)

	d1 - d2	date(2007, 1, 1) - date(2006, 1, 1) == timedelta(365)
		# leap year 2004 had 366 days and Python is aware of that:
		date(2005, 1, 1) - date(2004, 1, 1) == timedelta(366)

datetime.datetime	d2 + td	datetime(2007, 12, 31, 23, 59) + timedelta(minutes=1) == datetime(2008, 1, 1, 0, 0)
	d2 - td	datetime(2007, 12, 31) - timedelta(weeks=52, seconds=1) == datetime(2006, 12, 31, 23, 59, 59)
	d1 - d2	datetime(2007, 1, 1) - datetime(2008, 1, 1) == timedelta(-365)

Don't try to fool Python into trying to get February 29, 2007, because there's no such date—the interpreter will raise a ValueError exception with a "day is out of range for month" message. At any time you can also use datetime.datetime.today() to obtain a datetime object for the current date and time.

If you need to parse existing strings into date(time) objects you can use the strptime() method of a datetime object:

```
>>> from datetime import datetime
>>> datetime.strptime("2007-12-31 23:59:59", "%Y-%m-%d %H:%M:%S")
datetime.datetime(2007, 12, 31, 23, 59, 59)
```

The complete list of format strings for the strptime function is included in the documentation of the time module in Python Library Reference.

Time Zones

Python ships with time zone support available through the datetime module but it is not ready for prime time right away; it is only a skeleton provided for your implementation. You need to create your own class inherited from datetime.tzinfo and put the desired logic inside. Python Library Reference covers this subject extensively.

TO_DATE(Python)

Unsurprisingly, cx_Oracle makes the transition between Oracle and Python datatypes completely invisible. Queries involving DATE or TIMESTAMP columns return datetime objects.

Let's see whether Oracle's INTERVAL arithmetic and Python's timedelta calculations are equivalent:

```
>>> import cx_Oracle
>>> db = cx_Oracle.connect('hr/hrpwd@localhost:1521/XE')
>>> cursor = db.cursor()
>>> r = cursor.execute("SELECT end_date-start_date diff, end_date,
        start_date FROM job_history")
>>> for diff, end_date, start_date in cursor:
...     print diff, '\t', (end_date-start_date).days
...
2018    2018
1497    1497

1644    1644
```

Great! They do match.

The next example will demonstrate that it is up to you whether to put date/time logic on the database or Python side. This flexibility makes it possible to fit into any scenario. Let's find all employees who were hired in Q4/1998:

```
>>> Q4 = (datetime.date(1998, 10, 1), datetime.date(1998, 12, 31))
>>> r = cursor.execute("""
SELECT last_name||' '||first_name name, TO_CHAR(hire_date, 'YYYY-MM-DD')
FROM employees  WHERE hire_date BETWEEN :1 AND :2  ORDER BY hire_date ASC """, Q4)
```

```
>>> for row in cursor:
... print row
...
('Sewall Sarath', '1998-11-03')
('Himuro Guy', '1998-11-15')
('Cambrault Nanette', '1998-12-09')
```

Python datetime.date, datetime.time, and datetime.datetime objects can be safely used as bind variables for querying dates. You can choose which level of granularity is required but keep in mind that when working with fractions of a second you need to instruct cx_Oracle that a fractional part is being passed. Regular queries return perfectly valid timestamps with fractional parts of a second, this concerns using bind variables only. Of course, you can always use the strptime() function of Python in combination with TO_DATE() in Oracle, but really, why bother?

Let's create a simple table of the following structure:

```
CREATE TABLE python_tstamps (
  ts TIMESTAMP(6)
);
```

The following example illustrates the problem. ts has a fractional part that gets truncated during the insertion of ts = datetime.datetime.now():

```
>>> ts = datetime.datetime.now()
>>> print ts
2007-03-10 20:01:24.046000
>>> cursor.execute("INSERT INTO python_tstamps VALUES(:t)", {'t':ts})
>>> db.commit()
```

```
SQL> SELECT ts FROM python_tstamps;

TS
---------------------------------------------------------------------------
10-MAR-07 08.01.56.000000 PM
10-MAR-07 08.12.02.109000 PM
```

The solution is to use the setinputsizes() method between the prepare and execute phases. It instructs cx_Oracle how to handle particular bind variables. Internally it predefines areas in memory for those objects. It can also be used to preallocate memory areas for strings of a certain length—they need to be given as integers representing their length.

Let's rewrite the insert now:

```
>>>  ts = datetime.datetime.now()
>>>  print ts
2007-03-10 20:12:02.109000
>>>  cursor.prepare("INSERT INTO python_tstamps VALUES(:t_val)")
>>>  cursor.setinputsizes(t_val=cx_Oracle.TIMESTAMP)
cursor.setinputsizes(t_val=cx_Oracle.TIMESTAMP)
>>>  cursor.execute(None, {'t_val':ts})
>>>  db.commit()
```

```
SQL>  SELECT ts FROM python_tstamps;

TS
---------------------------------------------------------------------------
10-MAR-07 08.01.56.000000 PM
10-MAR-07 08.12.02.109000 PM
```

Conclusion

Important things to remember about cx_Oracle 4.3 in the datetime context:

no support for INTERVAL and TIMESTAMP WITH (LOCAL) TIME ZONE
Fractional seconds of a date/time value passed as a bind variable will be truncated unless the setinputsizes() method is used between prepare() and execute()
For time zone support either opt for writing your own implementation using the datetime.tzinfo class from the standard library or choose the pytz module available from SourceForge. Be prepared however that there is no smooth transition between WITH TIME ZONE column types and Python datetime objects.
The Python standard library offers other facilities for date/time work including:

A calendar module for displaying calendars in text and HTML formats as well as writing your own, derived implementations
A timeit module for profiling and benchmarking Python code
A sched module which does the same job as the cron utility under Linux/Unix
After finishing this tutorial you should be familiar with the concepts that rule Oracle and Python date handling. Getting acquainted with seamlessly integrated datetime datatypes you can now inject them in your calendar-aware applications, gaining Python for your development toolbox.

Need a calendar quickly? import calendar; print calendar.calendar(2007).

Przemyslaw Piotrowski is an information technology specialist working with emerging technologies and dynamic, agile development environments. Having a strong IT background that includes administration, development and design, he finds many paths of software interoperability.