**Dates and Times, Data Compression, Performance Measurement, and Quality Control**

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

>>>

>>> # dates are easily constructed and formatted

>>> from datetime import date

>>> now = date.today()

>>> now

datetime.date(2003, 12, 2)

>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")

'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic

>>> birthday = date(1964, 7, 31)

>>> age = now - birthday

>>> age.days

14368

**Data Compression**

Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, lzma, zipfile and tarfile.

>>>

>>> import zlib

>>> s = b'witch which has which witches wrist watch'

>>> len(s)

41

>>> t = zlib.compress(s)

>>> len(t)

37

>>> zlib.decompress(t)

b'witch which has which witches wrist watch'

>>> zlib.crc32(s)

226805979

**Performance Measurement**

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The timeit module quickly demonstrates a modest performance advantage:

>>>

>>> from timeit import Timer

>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()

0.57535828626024577

>>> Timer('a,b = b,a', 'a=1; b=2').timeit()

0.54962537085770791

In contrast to timeit’s fine level of granularity, the profile and pstats modules provide tools for identifying time critical sections in larger blocks of code.

**Quality Control**

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The doctest module provides a tool for scanning a module and validating tests embedded in a program’s docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

def average(values):

    """Computes the arithmetic mean of a list of numbers.

>>> print(average([20, 30, 70]))

40.0

"""

return sum(values) / len(values)

import doctest

doctest.testmod()   # automatically validate the embedded tests

The unittest module is not as effortless as the doctest module, but it allows a more comprehensive set of tests to be maintained in a separate file:

import unittest

class TestStatisticalFunctions(unittest.TestCase):

def test\_average(self):

        self.assertEqual(average([20, 30, 70]), 40.0)

        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)

     with self.assertRaises(ZeroDivisionError):

            average([])

     with self.assertRaises(TypeError):

            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests.