```
#!/usr/bin/env python

"""A sample implementation of MD5 in pure Python.

This is an implementation of the MD5 hash function, as specified by
RFC 1321, in pure Python. It was implemented using Bruce Schneier's
excellent book "Applied Cryptography", 2nd ed., 1996.

Surely this is not meant to compete with the existing implementation
of the Python standard library (written in C). Rather, it should be
seen as a Python complement that is more readable than C and can be
used more conveniently for learning and experimenting purposes in
the field of cryptography.

This module tries very hard to follow the API of the existing Python
standard library's "md5" module, but although it seems to work fine,
it has not been extensively tested! (But note that there is a test
module, test_md5py.py, that compares this Python implementation with
the C one of the Python standard library.

BEWARE: this comes with no guarantee whatsoever about fitness and/or
other properties! Specifically, do not use this in any production
code! License is Python License!

Special thanks to Aurelian Coman who fixed some nasty bugs!

Dinu C. Gherman
"""


__date__    = '2001-10-1'
__version__ = 0.9


import struct, string, copy


# ======================================================================
# Bit-Manipulation helpers
#
#   _long2bytes() was contributed by Barry Warsaw
#   and is reused here with tiny modifications.
# ======================================================================

def _long2bytes(n, blocksize=0):
    """Convert a long integer to a byte string.

    If optional blocksize is given and greater than zero, pad the front
    of the byte string with binary zeros so that the length is a multiple
    of blocksize.
    """

    # After much testing, this algorithm was deemed to be the fastest.
    s = ''
    pack = struct.pack
    while n > 0:
        ### CHANGED FROM '>I' TO '<I'. (DCG)
        s = pack('<I', n & 0xffffffffL) + s
        ### --------------------------
        n = n >> 32

    # Strip off leading zeros.
    for i in range(len(s)):
        if s[i] <> '\000':
            break
    else:
        # Only happens when n == 0.
        s = '\000'
        i = 0
```

```
    s = s[i:]

    # Add back some pad bytes. This could be done more efficiently
    # w.r.t. the de-padding being done above, but sigh...
    if blocksize > 0 and len(s) % blocksize:
        s = (blocksize - len(s) % blocksize) * '\000' + s

    return s


def _bytelist2long(list):
    "Transform a list of characters into a list of longs."

    imax = len(list)/4
    hl = [0L] * imax

    j = 0
    i = 0
    while i < imax:
        b0 = long(ord(list[j]))
        b1 = (long(ord(list[j+1]))) << 8
        b2 = (long(ord(list[j+2]))) << 16
        b3 = (long(ord(list[j+3]))) << 24
        hl[i] = b0 | b1 |b2 | b3
        i = i+1
        j = j+4

    return hl


def _rotateLeft(x, n):
    "Rotate x (32 bit) left n bits circularly."

    return (x << n) | (x >> (32-n))


# ======================================================================
# The real MD5 meat...
#
#   Implemented after "Applied Cryptography", 2nd ed., 1996,
#   pp. 436-441 by Bruce Schneier.
# ======================================================================

# F, G, H and I are basic MD5 functions.

def F(x, y, z):
    return (x & y) | ((~x) & z)

def G(x, y, z):
    return (x & z) | (y & (~z))

def H(x, y, z):
    return x ^ y ^ z

def I(x, y, z):
    return y ^ (x | (~z))


def XX(func, a, b, c, d, x, s, ac):
    """Wrapper for call distribution to functions F, G, H and I.

    This replaces functions FF, GG, HH and II from "Appl. Crypto.
    Rotation is separate from addition to prevent recomputation
    (now summed-up in one function).
    """

    res = 0L
    res = res + a + func(b, c, d)
    res = res + x
    res = res + ac
```

```
    res = res & 0xffffffffL
    res = _rotateLeft(res, s)
    res = res & 0xffffffffL
    res = res + b

    return res & 0xffffffffL


class MD5:
    "An implementation of the MD5 hash function in pure Python."

    def __init__(self):
        "Initialisation."

        # Initial 128 bit message digest (4 times 32 bit).
        self.A = 0L
        self.B = 0L
        self.C = 0L
        self.D = 0L

        # Initial message length in bits(!).
        self.length = 0L
        self.count = [0, 0]

        # Initial empty message as a sequence of bytes (8 bit characters).
        self.input = []

        # Length of the final hash (in bytes).
        self.HASH_LENGTH = 16

        # Length of a block (the number of bytes hashed in every transform).
        self.DATA_LENGTH = 64

        # Call a separate init function, that can be used repeatedly
        # to start from scratch on the same object.
        self.init()


    def init(self):
        "Initialize the message-digest and set all fields to zero."

        self.length = 0L
        self.input = []

        # Load magic initialization constants.
        self.A = 0x67452301L
        self.B = 0xefcdab89L
        self.C = 0x98badcfeL
        self.D = 0x10325476L


    def _transform(self, inp):
        """Basic MD5 step transforming the digest based on the input.

        Note that if the Mysterious Constants are arranged backwards
        in little-endian order and decrypted with the DES they produce
        OCCULT MESSAGES!
        """

        a, b, c, d = A, B, C, D = self.A, self.B, self.C, self.D

        # Round 1.

        S11, S12, S13, S14 = 7, 12, 17, 22

        a = XX(F, a, b, c, d, inp[ 0], S11, 0xD76AA478L) # 1
        d = XX(F, d, a, b, c, inp[ 1], S12, 0xE8C7B756L) # 2
        c = XX(F, c, d, a, b, inp[ 2], S13, 0x242070DBL) # 3
        b = XX(F, b, c, d, a, inp[ 3], S14, 0xC1BDCEEEL) # 4
        a = XX(F, a, b, c, d, inp[ 4], S11, 0xF57C0FAFL) # 5
```

```
        d = XX(F, d, a, b, c, inp[ 5], S12, 0x4787C62AL) # 6
        c = XX(F, c, d, a, b, inp[ 6], S13, 0xA8304613L) # 7
        b = XX(F, b, c, d, a, inp[ 7], S14, 0xFD469501L) # 8
        a = XX(F, a, b, c, d, inp[ 8], S11, 0x698098D8L) # 9
        d = XX(F, d, a, b, c, inp[ 9], S12, 0x8B44F7AFL) # 10
        c = XX(F, c, d, a, b, inp[10], S13, 0xFFFF5BB1L) # 11
        b = XX(F, b, c, d, a, inp[11], S14, 0x895CD7BEL) # 12
        a = XX(F, a, b, c, d, inp[12], S11, 0x6B901122L) # 13
        d = XX(F, d, a, b, c, inp[13], S12, 0xFD987193L) # 14
        c = XX(F, c, d, a, b, inp[14], S13, 0xA679438EL) # 15
        b = XX(F, b, c, d, a, inp[15], S14, 0x49B40821L) # 16

        # Round 2.

        S21, S22, S23, S24 = 5, 9, 14, 20

        a = XX(G, a, b, c, d, inp[ 1], S21, 0xF61E2562L) # 17
        d = XX(G, d, a, b, c, inp[ 6], S22, 0xC040B340L) # 18
        c = XX(G, c, d, a, b, inp[11], S23, 0x265E5A51L) # 19
        b = XX(G, b, c, d, a, inp[ 0], S24, 0xE9B6C7AAL) # 20
        a = XX(G, a, b, c, d, inp[ 5], S21, 0xD62F105DL) # 21
        d = XX(G, d, a, b, c, inp[10], S22, 0x02441453L) # 22
        c = XX(G, c, d, a, b, inp[15], S23, 0xD8A1E681L) # 23
        b = XX(G, b, c, d, a, inp[ 4], S24, 0xE7D3FBC8L) # 24
        a = XX(G, a, b, c, d, inp[ 9], S21, 0x21E1CDE6L) # 25
        d = XX(G, d, a, b, c, inp[14], S22, 0xC33707D6L) # 26
        c = XX(G, c, d, a, b, inp[ 3], S23, 0xF4D50D87L) # 27
        b = XX(G, b, c, d, a, inp[ 8], S24, 0x455A14EDL) # 28
        a = XX(G, a, b, c, d, inp[13], S21, 0xA9E3E905L) # 29
        d = XX(G, d, a, b, c, inp[ 2], S22, 0xFCEFA3F8L) # 30
        c = XX(G, c, d, a, b, inp[ 7], S23, 0x676F02D9L) # 31
        b = XX(G, b, c, d, a, inp[12], S24, 0x8D2A4C8AL) # 32

        # Round 3.

        S31, S32, S33, S34 = 4, 11, 16, 23

        a = XX(H, a, b, c, d, inp[ 5], S31, 0xFFFA3942L) # 33
        d = XX(H, d, a, b, c, inp[ 8], S32, 0x8771F681L) # 34
        c = XX(H, c, d, a, b, inp[11], S33, 0x6D9D6122L) # 35
        b = XX(H, b, c, d, a, inp[14], S34, 0xFDE5380CL) # 36
        a = XX(H, a, b, c, d, inp[ 1], S31, 0xA4BEEA44L) # 37
        d = XX(H, d, a, b, c, inp[ 4], S32, 0x4BDECFA9L) # 38
        c = XX(H, c, d, a, b, inp[ 7], S33, 0xF6BB4B60L) # 39
        b = XX(H, b, c, d, a, inp[10], S34, 0xBEBFBC70L) # 40
        a = XX(H, a, b, c, d, inp[13], S31, 0x289B7EC6L) # 41
        d = XX(H, d, a, b, c, inp[ 0], S32, 0xEAA127FAL) # 42
        c = XX(H, c, d, a, b, inp[ 3], S33, 0xD4EF3085L) # 43
        b = XX(H, b, c, d, a, inp[ 6], S34, 0x04881D05L) # 44
        a = XX(H, a, b, c, d, inp[ 9], S31, 0xD9D4D039L) # 45
        d = XX(H, d, a, b, c, inp[12], S32, 0xE6DB99E5L) # 46
        c = XX(H, c, d, a, b, inp[15], S33, 0x1FA27CF8L) # 47
        b = XX(H, b, c, d, a, inp[ 2], S34, 0xC4AC5665L) # 48

        # Round 4.

        S41, S42, S43, S44 = 6, 10, 15, 21

        a = XX(I, a, b, c, d, inp[ 0], S41, 0xF4292244L) # 49
        d = XX(I, d, a, b, c, inp[ 7], S42, 0x432AFF97L) # 50
        c = XX(I, c, d, a, b, inp[14], S43, 0xAB9423A7L) # 51
        b = XX(I, b, c, d, a, inp[ 5], S44, 0xFC93A039L) # 52
        a = XX(I, a, b, c, d, inp[12], S41, 0x655B59C3L) # 53
        d = XX(I, d, a, b, c, inp[ 3], S42, 0x8F0CCC92L) # 54
        c = XX(I, c, d, a, b, inp[10], S43, 0xFFEFF47DL) # 55
        b = XX(I, b, c, d, a, inp[ 1], S44, 0x85845DD1L) # 56
        a = XX(I, a, b, c, d, inp[ 8], S41, 0x6FA87E4FL) # 57
        d = XX(I, d, a, b, c, inp[15], S42, 0xFE2CE6E0L) # 58
        c = XX(I, c, d, a, b, inp[ 6], S43, 0xA3014314L) # 59
        b = XX(I, b, c, d, a, inp[13], S44, 0x4E0811A1L) # 60
```

```
        a = XX(I, a, b, c, d, inp[ 4], S41, 0xF7537E82L) # 61
        d = XX(I, d, a, b, c, inp[11], S42, 0xBD3AF235L) # 62
        c = XX(I, c, d, a, b, inp[ 2], S43, 0x2AD7D2BBL) # 63
        b = XX(I, b, c, d, a, inp[ 9], S44, 0xEB86D391L) # 64

        A = (A + a) & 0xffffffffL
        B = (B + b) & 0xffffffffL
        C = (C + c) & 0xffffffffL
        D = (D + d) & 0xffffffffL

        self.A, self.B, self.C, self.D = A, B, C, D


    # Down from here all methods follow the Python Standard Library
    # API of the md5 module.

    def update(self, inBuf):
        """Add to the current message.

        Update the md5 object with the string arg. Repeated calls
        are equivalent to a single call with the concatenation of all
        the arguments, i.e. m.update(a); m.update(b) is equivalent
        to m.update(a+b).
        """
        leninBuf = long(len(inBuf))

        # Compute number of bytes mod 64.
        index = (self.count[0] >> 3) & 0x3FL

        # Update number of bits.
        self.count[0] = self.count[0] + (leninBuf << 3)
        if self.count[0] < (leninBuf << 3):
            self.count[1] = self.count[1] + 1
        self.count[1] = self.count[1] + (leninBuf >> 29)

        partLen = 64 - index

        if leninBuf >= partLen:
            self.input[index:] = map(None, inBuf[:partLen])
            self._transform(_bytelist2long(self.input))
            i = partLen
            while i + 63 < leninBuf:
                self._transform(_bytelist2long(map(None, inBuf[i:i+64])))
                i = i + 64
            else:
                self.input = map(None, inBuf[i:leninBuf])
        else:
            i = 0
            self.input = self.input + map(None, inBuf)


    def digest(self):
        """Terminate the message-digest computation and return digest.

        Return the digest of the strings passed to the update()
        method so far. This is a 16-byte string which may contain
        non-ASCII characters, including null bytes.
        """

        A = self.A
        B = self.B
        C = self.C
        D = self.D
        input = [] + self.input
        count = [] + self.count

        index = (self.count[0] >> 3) & 0x3fL

        if index < 56:
            padLen = 56 - index
```

```
        else:
            padLen = 120 - index

        padding = ['\200'] + ['\000'] * 63
        self.update(padding[:padLen])

        # Append length (before padding).
        bits = _bytelist2long(self.input[:56]) + count

        self._transform(bits)

        # Store state in digest.
        digest = _long2bytes(self.A << 96, 16)[:4] + \
                 _long2bytes(self.B << 64, 16)[4:8] + \
                 _long2bytes(self.C << 32, 16)[8:12] + \
                 _long2bytes(self.D, 16)[12:]

        self.A = A
        self.B = B
        self.C = C
        self.D = D
        self.input = input
        self.count = count

        return digest


    def hexdigest(self):
        """Terminate and return digest in HEX form.

        Like digest() except the digest is returned as a string of
        length 32, containing only hexadecimal digits. This may be
        used to exchange the value safely in email or other non-
        binary environments.
        """

        d = map(None, self.digest())
        d = map(ord, d)
        d = map(lambda x:"%02x" % x, d)
        d = string.join(d, '')

        return d


    def copy(self):
        """Return a clone object.

        Return a copy ('clone') of the md5 object. This can be used
        to efficiently compute the digests of strings that share
        a common initial substring.
        """

        return copy.deepcopy(self)


# ======================================================================
# Mimick Python top-level functions from standard library API
# for consistency with the md5 module of the standard library.
# ======================================================================

def new(arg=None):
    """Return a new md5 object.

    If arg is present, the method call update(arg) is made.
    """

    md5 = MD5()
    if arg:
        md5.update(arg)
```

```
    return md5


def md5(arg=None):
    """Same as new().

    For backward compatibility reasons, this is an alternative
    name for the new() function.
    """

    return new(arg)
```