Manipulation Functions:

```
def leftrotate_64(x, c):
    """ Left rotate the number x by c bytes, for 64-bits numbers."""
    x &= 0xFFFFFFFFFFFFFFFF
    return ((x << c) | (x >> (64 - c))) & 0xFFFFFFFFFFFFFFFF

def rightrotate_64(x, c):
    """ Right rotate the number x by c bytes, for 64-bits numbers."""
    x &= 0xFFFFFFFFFFFFFFFF
    return ((x >> c) | (x << (64 - c))) & 0xFFFFFFFFFFFFFFFF

def leftshift(x, c):
    """ Left shift the number x by c bytes."""
    return x << c

def rightshift(x, c):
    """ Right shift the number x by c bytes."""
    return x >> c
```

SHA512 Class Declaration

```
class SHA512(object):

    def __init__(self):
        self.name        = "SHA512"
        self.byteorder   = 'big'
        self.block_size  = 128
        self.digest_size = 64

        # Initialize hash values:
        # (The second 64 bits of the fractional parts of the square
roots of the first 8 primes 2..19)
        h0 = 0x6a09e667f3bcc908
        h1 = 0xbb67ae8584caa73b
        h2 = 0x3c6ef372fe94f82b
        h3 = 0xa54ff53a5f1d36f1
        h4 = 0x510e527fade682d1
        h5 = 0x9b05688c2b3e6c1f
        h6 = 0x1f83d9abfb41bd6b
        h7 = 0x5be0cd19137e2179

        # Initialize array of round constants:
        # (first 64 bits of the fractional parts of the cube roots of
the first 80 primes 2..409):
        self.k = [
            0x428a2f98d728ae22, 0x7137449123ef65cd,
0xb5c0fbcfec4d3b2f, 0xe9b5dba58189dbbc, 0x3956c25bf348b538,
            0x59f111f1b605d019, 0x923f82a4af194f9b,
0xab1c5ed5da6d8118, 0xd807aa98a3030242, 0x12835b0145706fbe,
```

```
            0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2,
0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235,
            0xc19bf174cf692694, 0xe49b69c19ef14ad2,
0xefbe4786384f25e3, 0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,
            0x2de92c6f592b0275, 0x4a7484aa6ea6e483,
0x5cb0a9dcbd41fbd4, 0x76f988da831153b5, 0x983e5152ee66dfab,
            0xa831c66d2db43210, 0xb00327c898fb213f,
0xbf597fc7beef0ee4, 0xc6e00bf33da88fc2, 0xd5a79147930aa725,
            0x06ca6351e003826f, 0x142929670a0e6e70,
0x27b70a8546d22ffc, 0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed,
            0x53380d139d95b3df, 0x650a73548baf63de,
0x766a0abb3c77b2a8, 0x81c2c92e47edaee6, 0x92722c851482353b,
            0xa2bfe8a14cf10364, 0xa81a664bbc423001,
0xc24b8b70d0f89791, 0xc76c51a30654be30, 0xd192e819d6ef5218,
            0xd69906245565a910, 0xf40e35855771202a,
0x106aa07032bbd1b8, 0x19a4c116b8d2d0c8, 0x1e376c085141ab53,
            0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8,
0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb, 0x5b9cca4f7763e373,
            0x682e6ff3d6b2b8a3, 0x748f82ee5defb2fc,
0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
            0x90befffa23631e28, 0xa4506cebde82bde9,
0xbef9a3f7b2c67915, 0xc67178f2e372532b, 0xca273eceea26619c,
            0xd186b8c721c0c207, 0xeada7dd6cde0eb1e,
0xf57d4f7fee6ed178, 0x06f067aa72176fba, 0x0a637dc5a2c898a6,
            0x113f9804bef90dae, 0x1b710b35131c471b,
0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc,
            0x431d67c49c100d4c, 0x4cc5d4becb3e42b6,
0x597f299cfc657e2a, 0x5fcb6fab3ad6faec, 0x6c44198c4a475817
        ]

        # Store them
        self.hash_pieces = [h0, h1, h2, h3, h4, h5, h6, h7]

    def update(self, arg):
        h0, h1, h2, h3, h4, h5, h6, h7 = self.hash_pieces
        # 1. Pre-processing
        data = bytearray(arg)
        orig_len_in_bits = (8 * len(data)) &
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
        # 1.a. Add a single '1' bit at the end of the input bits
        data.append(0x80)
        # 1.b. Padding with zeros as long as the input bits length ≡
896 (mod 1024)
        while len(data) % 128 != 112:
            data.append(0)
        # 1.c. append original length in bits mod (2 pow 128) to
message
        data += orig_len_in_bits.to_bytes(16, byteorder='big')
        assert len(data) % 128 == 0, "Error in padding"
        # 2. Computations
```

```
        # Process the message in successive 1024-bit = 128-bytes
chunks:
        for offset in range(0, len(data), 128):
            # 2.a. 1024-bits = 128-bytes chunks
            chunks = data[offset : offset + 128]
            w = [0 for i in range(80)]
            # 2.b. Break chunk into sixteen 128-bit = 8-bytes words
w[i], 0 ≤ i ≤ 15
            for i in range(16):
                w[i] = int.from_bytes(chunks[8*i : 8*i + 8],
byteorder='big')
            # 2.c.  Extend the first 16 words into the remaining 64
            #       words w[16..79] of the message schedule array:
            for i in range(16, 80):
                s0 = (rightrotate_64(w[i-15], 1) ^ rightrotate_64(w[i-
15], 8) ^ rightshift(w[i-15], 7)) & 0xFFFFFFFFFFFFFFFF
                s1 = (rightrotate_64(w[i-2], 19) ^ rightrotate_64(w[i-
2], 61) ^ rightshift(w[i-2], 6)) & 0xFFFFFFFFFFFFFFFF
                w[i] = (w[i-16] + s0 + w[i-7] + s1) &
0xFFFFFFFFFFFFFFFF
            # 2.d. Initialize hash value for this chunk
            a, b, c, d, e, f, g, h = h0, h1, h2, h3, h4, h5, h6, h7
            # 2.e. Main loop
            for i in range(80):
                S1 = (rightrotate_64(e, 14) ^ rightrotate_64(e, 18) ^
rightrotate_64(e, 41)) & 0xFFFFFFFFFFFFFFFF
                ch = ((e & f) ^ ((~e) & g)) & 0xFFFFFFFFFFFFFFFF
                temp1 = (h + S1 + ch + self.k[i] + w[i]) &
0xFFFFFFFFFFFFFFFF
                S0 = (rightrotate_64(a, 28) ^ rightrotate_64(a, 34) ^
rightrotate_64(a, 39)) & 0xFFFFFFFFFFFFFFFF
                maj = ((a & b) ^ (a & c) ^ (b & c)) &
0xFFFFFFFFFFFFFFFF
                temp2 = (S0 + maj) & 0xFFFFFFFFFFFFFFFF

                new_a = (temp1 + temp2) & 0xFFFFFFFFFFFFFFFF
                new_e = (d + temp1) & 0xFFFFFFFFFFFFFFFF
                # Rotate the 8 variables
                a, b, c, d, e, f, g, h = new_a, a, b, c, new_e, e, f,
g

            # Add this chunk's hash to result so far:
            h0 = (h0 + a) & 0xFFFFFFFFFFFFFFFF
            h1 = (h1 + b) & 0xFFFFFFFFFFFFFFFF
            h2 = (h2 + c) & 0xFFFFFFFFFFFFFFFF
            h3 = (h3 + d) & 0xFFFFFFFFFFFFFFFF
            h4 = (h4 + e) & 0xFFFFFFFFFFFFFFFF
            h5 = (h5 + f) & 0xFFFFFFFFFFFFFFFF
            h6 = (h6 + g) & 0xFFFFFFFFFFFFFFFF
            h7 = (h7 + h) & 0xFFFFFFFFFFFFFFFF
```

```
        # 3. Conclusion
        self.hash_pieces = [h0, h1, h2, h3, h4, h5, h6, h7]

    def digest(self):
        # h0 append h1 append h2 append h3 append h4 append h5 append
h6 append h7
        return sum(leftshift(x, 64 * i) for i, x in
enumerate(self.hash_pieces[::-1]))

    def hexdigest(self):
        """ Returns a string object of double length, containing only
hexadecimal digits."""
        digest = self.digest()
        raw = digest.to_bytes(self.digest_size,
byteorder=self.byteorder)
        format_str = '{:0' + str(2 * self.digest_size) + 'x}'
        return format_str.format(int.from_bytes(raw, byteorder='big'))
```

Shortcut Function:

```
def hash_SHA512(data):
    """ Shortcut function to directly receive the hex digest from
SHA512(data)."""
    h = SHA512()
    if isinstance(data, str):
        data = bytes(data, encoding='utf8')
    h.update(data)
    return h.hexdigest()
```

Test Cases

```
test_cases={
    "abc":
"ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a21929
92a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f",

    "":
"cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d
13c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327af927da3e",

    "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq":

"204a8fc6dda82f0a0ced7beb8e08a41657c16ef468b228a8279be331a703c33596fd1
5c13b1b07f9aa1d3bea57789ca031ad85c7a71dd70354ec631238ca3445",


"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklm
nopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu":

"8e959b75dae313da8cf4f72814fc143f8f7779c6eb9f7fa17299aeadb6889018501d2
89e4900f7e4331b99dec4b5433ac7d329eeb6dd26545e96e55b874be909",
```

```
    }

for i in test_cases:
    g= hash_SHA512(i)
    if g ==test_cases[i]:
        print ("test case "+i+" is sattisfied and the hash is:\n"+ g
+"\n")
    else:
        print ("test case "+i+" is not sattisfied and the hash: \n"+ g
+ "\n does not match test hash \n" + test_cases[i]+ "\n")
```

```
test case abc is sattisfied and the hash is:
ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a219299
2a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f

test case  is sattisfied and the hash is:
cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d1
3c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327af927da3e

test case abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq is
sattisfied and the hash is:
204a8fc6dda82f0a0ced7beb8e08a41657c16ef468b228a8279be331a703c33596fd15
c13b1b07f9aa1d3bea57789ca031ad85c7a71dd70354ec631238ca3445

test case
abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklmn
opjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu is sattisfied and the hash
is:
8e959b75dae313da8cf4f72814fc143f8f7779c6eb9f7fa17299aeadb6889018501d28
9e4900f7e4331b99dec4b5433ac7d329eeb6dd26545e96e55b874be909
```