The python program sha512.py implements the full SHA512 hashing algorithm. Given a file containing some plaintext, sha512.py will write a file containing the correct hashed result. When command-line below is invoked, sha512.py will perform SHA512 hashing on input.txt write the result to a file called hashed.txt.

```
python sha512.py input.txt hashed.txt
```

Figure I. Example of Command-Line Syntax to Invoke SHA512 hashing using sha512.py

An example of this is when a message.txt contains the following text:

```
Boku no Kokoro no Yabai Yatsu is the greatest romance, slice of life manga I've ever read. It is a series of constant progress that respects the reader's time and trusts them to read between the lines - Characters make mistakes and learn from them. Misunderstandings are never used to pad out the story, and never feel cheap. Progress is never undone. It is one of the most fully realized depictions of the liminal space between two young people as they begin to fall in love - The roller coaster between bubbly feelings and crippling cringe that is first love is so difficult to portray. I've never encountered another manga that has managed to capture this specific feeling so accurately and with so much detail.
```

Figure II. Example of the content in message.txt

When the above command-line is invoked, sha512.py will output the following to hashed.txt:

```
84f353348a552229554fba7ba822005edcb6bca2fac8cf1735d53ae9e2915aa2e625f6d3cfa0106c8707ff0004d3ce952
81b47b851b380ef91c86d2fb0e58b28
```

Figure III. Contents of the written output of to hashed.txt using SHA512 Algorithm in sha512.py

It is important to note that the output file of the SHA512 hashing process of sha512.py is saved as a single-line hex file. The overall hashing process is in accordance with the standard of SHA512 hash algorithm, where it will generate a fixed-sized message digest from input data as large as 2^{128} bits. The algorithm would first pad the input so that its length is an integral multiple of 1024 bits, in which the last block also contains a value of the length of the message. The program will then generate the message schedule that is required to process the 1024-bit block of the input message. The message schedule consists of 80 64-bit words. The first 16 of these words are obtained from the 1024-bit message block and the rest of the words are obtained by applying permutation and mixing operations to some of the previously generated words. Each of the blocks will then go through 80 rounds of round-based processing.

For further reference, below is the SHA512 algorithm implemented in sha512.py:

```
def sha512hash(self):
    h0 = BitVector(hexstring='6a09e667f3bcc908')
    h1 = BitVector(hexstring='bb67ae8584caa73b')
    h2 = BitVector(hexstring='3c6ef372fe94f82b')
    h3 = BitVector(hexstring='a54ff53a5f1d36f1')
    h4 = BitVector(hexstring='510e527fade682d1')
    h5 = BitVector(hexstring='9b05688c2b3e6c1f')
    h6 = BitVector(hexstring='1f83d9abfb41bd6b')
    h7 = BitVector(hexstring='5be0cd19137e2179')
    length_0 = self.bv.length()
    bv_1 = self.bv + BitVector(bitstring="1")
    length1 = bv_1.length()
    numZeroes = (896 - length1) % 1024
    listZero = [0] * numZeroes
    bv_2 = bv_1 + BitVector(bitlist=listZero)
    bv_3 = BitVector(intVal=length_0, size=128)
    constructed_bv = bv_2 + bv_3
    words = [None]*80
    for n in range(0, len(constructed_bv), 1024):
        bvBlock = constructed_bv[n:n+1024]
        words[0:16] = [bvBlock[i:i + 64] for i in range(0, 1024, 64)]
        for i in range(16, 80):
            word_i_minus_2 = words[i - 2]
            word_i_minus_15 = words[i - 15]
            sigma0 = (word_i_minus_15.deep_copy() >> 1) ^ (word_i_minus_15.deep_copy() >> 8) ^ (word_i_minus_15.deep_copy().shift_right(7))
            sigma1 = (word_i_minus_2.deep_copy() >> 19) ^ (word_i_minus_2.deep_copy() >> 61) ^ (word_i_minus_2.deep_copy().shift_right(6))
            words[i] = BitVector(intVal=(int(words[i-16]) + int(sigma1) + int(words[i-7]) + int(sigma0)) & 0xFFFFFFFFFFFFFFFF, size=64)
        a, b, c, d, e, f, g, h = h0, h1, h2, h3, h4, h5, h6, h7
        for i in range(80):
            ch = (e & f) ^ ((~e) & g)
            maj = (a & b) ^ (a & c) ^ (b & c)
            sum_a = ((a.deep_copy()) >> 28) ^ ((a.deep_copy()) >> 34) ^ ((a.deep_copy()) >> 39)
            sum_e = ((e.deep_copy()) >> 14) ^ ((e.deep_copy()) >> 18) ^ ((e.deep_copy()) >> 41)
            t1 = BitVector(intVal=(int(h) + int(ch) + int(sum_e) + int(words[i]) + int(self.k_bv[i])) % (2**64), size=64)
            t2 = BitVector(intVal=(int(sum_a) + int(maj)) % (2**64), size=64)
            h = g
            g = f
            f = e
            e = BitVector(intVal=(int(d) + int(t1)) % (2**64), size=64)
            d = c
            c = b
            b = a
            a = BitVector(intVal=(int(t1) + int(t2)) % (2**64), size=64)
        h0 = BitVector(intVal=(int(h0) + int(a)) % (2**64), size=64)
        h1 = BitVector(intVal=(int(h1) + int(b)) % (2**64), size=64)
        h2 = BitVector(intVal=(int(h2) + int(c)) % (2**64), size=64)
        h3 = BitVector(intVal=(int(h3) + int(d)) % (2**64), size=64)
        h4 = BitVector(intVal=(int(h4) + int(e)) % (2**64), size=64)
        h5 = BitVector(intVal=(int(h5) + int(f)) % (2**64), size=64)
        h6 = BitVector(intVal=(int(h6) + int(g)) % (2**64), size=64)
        h7 = BitVector(intVal=(int(h7) + int(h)) % (2**64), size=64)
    hashed = h0 + h1 + h2 + h3 + h4 + h5 + h6 + h7
    return hashed
```

Figure IV. SHA512 algorithm of sha512.py

The process of initializing the K constants and input bit vector is done in the __init__ method of the sha512 class (class constructor when creating a sha512 object), which is the following:

```
def __init__(self, input_val):
    self.bv = BitVector(textstring=input_val)
    self.k = ['428a2f98d728ae22', '7137449123ef65cd', 'b5c0fbcfec4d3b2f', 'e9b5dba58189dbbc',
            '3956c25bf348b538', '59f111f1b605d019', '923f82a4af194f9b', 'ab1c5ed5da6d8118',
            'd807aa98a3030242', '12835b0145706fbe', '243185be4ee4b28c', '550c7dc3d5ffb4e2',
            '72be5d74f27b896f', '80deb1fe3b1696b1', '9bdc06a725c71235', 'c19bf174cf692694',
            'e49b69c19ef14ad2', 'efbe4786384f25e3', '0fc19dc68b8cd5b5', '240ca1cc77ac9c65',
            '2de92c6f592b0275', '4a7484aa6ea6e483', '5cb0a9dcbd41fbd4', '76f988da831153b5',
            '983e5152ee66dfab', 'a831c66d2db43210', 'b00327c898fb213f', 'bf597fc7beef0ee4',
            'c6e00bf33da88fc2', 'd5a79147930aa725', '06ca6351e003826f', '142929670a0e6e70',
            '27b70a8546d22ffc', '2e1b21385c26c926', '4d2c6dfc5ac42aed', '53380d139d95b3df',
            '650a73548baf63de', '766a0abb3c77b2a8', '81c2c92e47edaee6', '92722c851482353b',
            'a2bfe8a14cf10364', 'a81a664bbc423001', 'c24b8b70d0f89791', 'c76c51a30654be30',
            'd192e819d6ef5218', 'd69906245565a910', 'f40e35855771202a', '106aa07032bbd1b8',
            '19a4c116b8d2d0c8', '1e376c085141ab53', '2748774cdf8eeb99', '34b0bcb5e19b48a8',
            '391c0cb3c5c95a63', '4ed8aa4ae3418acb', '5b9cca4f7763e373', '682e6ff3d6b2b8a3',
            '748f82ee5defb2fc', '78a5636f43172f60', '84c87814a1f0ab72', '8cc702081a6439ec',
            '90befffa23631e28', 'a4506cebde82bde9', 'bef9a3f7b2c67915', 'c67178f2e372532b',
            'ca273eceea26619c', 'd186b8c721c0c207', 'eada7dd6cde0eb1e', 'f57d4f7fee6ed178',
            '06f067aa72176fba', '0a637dc5a2c898a6', '113f9804bef90dae', '1b710b35131c471b',
            '28db77f523047d84', '32caab7b40c72493', '3c9ebe0a15c9bebc', '431d67c49c100d4c',
            '4cc5d4becb3e42b6', '597f299cfc657e2a', '5fcb6fab3ad6faec', '6c44198c4a475817']
    self.k_bv = [BitVector(hexstring=string) for string in self.k]
```

Figure V. __init__ method of the sha512 class