تابع درهم سازی sha1 دارای بلوک های ۵۱۲ بیت ، تعداد دور ۸۰ و خروجی ۱۶۰ بیت می باشد.

برای اینکه قالب ناقص آخر نداشته باشیم یک عملیات padding برروی متنی که قصد داریم آن را هش کنیم انجام می دهیم تا نهایتا به این نحو طول متن اصلی ما مضربی از ۵۱۲ بیت شود و بتوان آن را به قالب های کامل ۵۱۲ بیتی شکست.

این متن ۵۱۲ * n بیتی شامل L بیت متن پیام اصلی ، ۶۴ بیت طول پیام (همان L که در قالب یک عدد ۶۴ بیتی نوشته می شود) و K + 1 بیت در فرمت خاص برای تکمیل 512 * n بیت است.(فرمت خاص بدین شکل خواهد بود : 100...00 که تعداد k بیت صفر است).

$K = 512 - L - 1 - 64 \bmod 512 \rightarrow k = 448 - (L + 1) \bmod 512$

توضیحات کد :

تابع hexTobin یک رشته از اعداد که در قالب یک عدد هگزادسیمال است را به رشته اعداد در قالب باینری تبدیل می کند.

به این صورت که اعداد 0 تا f به شکل باینری نوشته شده و از 0 تا 9 در آرایه های bin_num و از a تا f در آرایه bin_num2 نگهداری می شود. تابع ord(i) کد اسکی کاراکتر i را باز می گرداند و برای رسیدن به خانه ای از ارایه که باینری شده عدد i را برگرداند کافی است ord('0') – ord(i) را محاسبه کنیم فرضا اگر I کاراکتر ۰ باشد آنگاه حاصل صفر خواهد شد و واضح است که باینری کاراکتر ۰ در خانه صفر ام آرایه قرار دارد به همین ترتیب اختلاف کد اسکی ۰ و ۱ برابر یک واحد است پس باینری شده عدد ۱ در خانه ی ۱ ام ارایه قرار می گیرد و... برای اعداد a تا f نیز به همین منوال به سبب اینکه کدهای اسکی کاراکتر های متوالی یک دنباله از اعداد متوالی با اختلاف یک واحد است.

```
bin_num = ['0000','0001','0010','0011','0100','0101','0110','0111','1000','1001']
bin_num_2 = ['1010','1011','1100','1101','1110','1111']

def hexTobin(hex_msg):
    res = ''
    for i in hex_msg:
        if(i <= '9') and (i >= '0'):
            res = res + bin_num[ord(i) - ord('0')]
        else:
            res = res + bin_num_2[ord(i) - ord('a')]
    return res
```

تابع pad تکه پیام را در قالب یک رشته باینری دریافت می کند (البته این رشته در انتهای خود یک 1 اضافی دارد که همان سازنده فرمت 0...10 است) و قصد دارد تعداد k بیت صفر را به انتهای قالب اضافه کند تا فرمت 0..100 (با k بیت صفر) کامل شود.

```
def pad(bin_d):
    while len(bin_d) % 512 != 448:
        bin_d = bin_d + '0'
    return bin_d
```

تابع chunks یک پیام L را دریافت کرده و آن را به بخش های n بیتی تقسیم می کند. این تابع در قالب یک for نوشته شده که شمارنده آن از 0 شروع شده و هربار n به آن اضافه می شود و پیام L[I : I + n] همان قالب های n بیتی را می سازد فرضا زمانی که شمارنده ۰ است L[0 : n] اولین بخش را می سازند به همین ترتیب بخش های بعدی L[n : 2n] و ...

```
def chunks(l, n):
    return [l[i:i+n] for i in range(0, len(l), n)]
```

تابع circular_left_shift یک رشته باینری را دریافت کرده و b بیت به سمت چپ شیفت گردشی می دهد.

```
def circular_left_shift(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff
```

تابع sha1 یک متن را دریافت کرده و قصد دارد هش آن را محاسبه کند

مقادیر h0 تا h4 مقادیر اولیه و ثابت هستند به شکل زیر:

```
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0
```

بخش زیر متن دریافتی را در قالب کد اسکی تبدیل و سپس به عدد هگزا دسیمال تبدیل می کند حال باینری این متن را نیاز داریم به همین سبب تابع hexTobin را برای آن فراخوانی می کنیم به علاوه عدد 1 که در انتها به پیام اضافه شده همان 1 مربوط به فرمت 10...0 (k بیت صفر) است.

```
bin_data = hexTobin((data.encode('ascii')).hex()) + '1'
```

سپس این bin_data وارد تابع pad می شود تا تعداد k بیت صفر مورد نیاز نیز در انتهای قالب قرار بگیرد.

```
bin_d = pad(bin_data)
```

سپس لازم است طول پیام اصلی (بدون 1 + k بیت اضافه شده) یعنی l را در قالب یک عدد ۶۴ بیتی به انتهای پیام حاصل از بخش های قبلی اضافه کنیم برای این منظور از قطعه کد زیر استفاده می کنیم : (عدد ۱ کسر شده از len(bin_data) به سبب این است که در انتهای bin_data عدد ۱ مربوط به فرمت 100...0(k بیت صفر) اضافه شده بود و طول متن پیام اصلی یک بیت کمتر از طول bin_data است.)

```
bin_d = bin_d + '{0:064b}'.format(len(bin_data)-1)
```

لازم است پیامی که آن را به صورت مضربی از ۵۱۲ تبدیل کردیم را در قالب های ۵۱۲ بیتی تقسیم کنیم و سپس هر یک ازین ۵۱۲ بیت را به ۱۶ بخش ۳۲ بیتی بشکنیم و xi ها را برای هر مرحله تشکیل دهیم سپس یک لیست متشکل از w79 ... w0 تشکیل می دهیم که مقادیر w0 تا w15 به ترتیب همان xi های بدست آمده است یعنی w0 تا w15 همان قالب ۵۱۲ بیت شکسته شده به ۱۶ قسمت ۳۲ بیتی است یعنی ۳۲ بیت اول w0 ، ۳۲ بیت دوم w1 و ... ۳۲ بیت آخر (۱۶ ام) w15 را می سازند.

و برای w16 تا w79 مقدار wi به صورت تابعی از wi های قبلی از رابطه زیر محاسبه می شود(یک بیت شیفت گردشی به چپ نیز در انتهای این تابع داریم).

w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16] <<< 1

```
for c in chunks(bin_d, 512):
    x = chunks(c, 32)
    w = [0] * 80

    for i in range(0, 16):
        w[i] = int(x[i], 2)

    for i in range(16, 80):
        w[i] = circular_left_shift((w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16]), 1)
```

حال ۸۰ دور درهم سازی را پیاده می کنیم :

در دور های ۰ ام تا ۱۹ ام تابع f به صورت (b & c) | ((~b) & d) است, مقدار k آن برابر 0x5A827999 = k می باشد.

در دور های ۲۰ ام تا ۳۹ ام تابع f به صورت b ^ c ^ d است, مقدار k آن برابر 0x6ED9EBA1 = k می باشد.

در دور های ۴۰ ام تا ۵۹ ام تابع f به صورت (b & c) | (b & d) | (c & d) است, مقدار k آن برابر

0x8F1BBCDC = k می باشد.

در دور های ۶۰ ام تا ۷۹ ام تابع f به صورت b ^ c ^ d است, مقدار k آن برابر 0xCA62C1D6 = k می باشد.

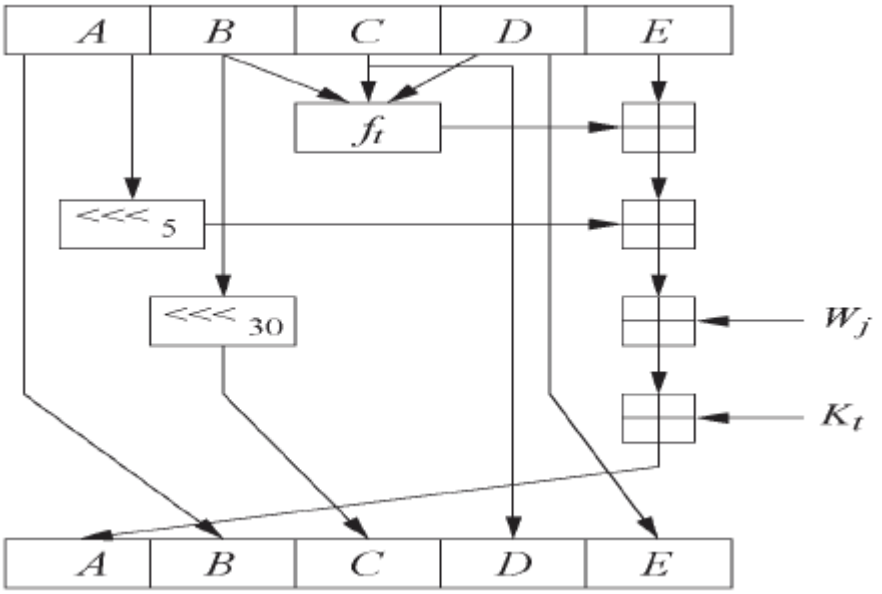
```
for i in range(0, 80):
    if 0 <= i <= 19:
        f = (b & c) | ((~b) & d)
        k = 0x5A827999
    elif 20 <= i <= 39:
        f = b ^ c ^ d
        k = 0x6ED9EBA1
    elif 40 <= i <= 59:
        f = (b & c) | (b & d) | (c & d)
        k = 0x8F1BBCDC
    elif 60 <= i <= 79:
        f = b ^ c ^ d
        k = 0xCA62C1D6
```

مطابق جدول زیر که در اسلاید های درس آمده است:

Stage t	Round j	Constant K_t	Function f_t
1	00…19	K=5A827999	f(B,C,D)=(B∧C)∨(¯B∧D)
2	20…39	K=6ED9EBA1	f(B,C,D)=B⊕C⊕D
3	40…59	K=8F1BBCDC	f(B,C,D)=(B⊕C)∨(B⊕D)∨(C⊕D)
4	60…79	K=CA62C1D6	f(B,C,D)=B⊕C⊕D

مرحله بعدی دقیقا مراحل پیاده سازی شکل زیر است :

```
    last_a = a
    a = circular_left_shift(a, 5) + f + e + k + w[i] & 0xffffffff
    e = d
    d = c
    c = circular_left_shift(b, 30)
    b = last_a
```



و نهایتا پس از دور آخر مقدار h0 با a جمع و حاصل در پیمانه 2^{32} بدست می آید و h0 نهایی را می سازد به همین ترتیب h1 با b جمع شده حاصل در پیمانه 2^{32} محاسبه می شود و h1 نهایی را می سازد و .... در حقیقت عمل &0x ffffffff همان محاسبه در پیمانه 2^{32} می باشد.

```
h0 = h0 + a & 0xffffffff
h1 = h1 + b & 0xffffffff
h2 = h2 + c & 0xffffffff
h3 = h3 + d & 0xffffffff
h4 = h4 + e & 0xffffffff
```

از کنار هم قرار گرفتن h0 تا h4 هش نهایی بدست می آید.

توضیحات کد hmac :

ابتدا کلید و متن دریافتی را به قالب اسکی تبدیل می کنیم:

```
k = k.encode('ascii')
text = text.encode('ascii')
```

می دانیم در حالت کلی تعداد ۲۵۶ کد اسکی داریم که از ۰ تا ۲۵۵ هر کد به یک کاراکتر اختصاص دارد حال بخش زیر هر یک ازین کد های اسکی را با عدد 0x5C و 0x36 ، xor می کند و حاصل را در یک byte array قرار می دهد.

```
t36 = bytes((x ^ 0x36) for x in range(256))
t5C = bytes((x ^ 0x5C) for x in range(256))
```

به علاوه می دانیم در hmac نیاز به کلید با طول ۵۱۲ بیت داریم زیرا از تابع sha256 استفاده می کنیم که طول ورودی ۵۱۲ بیت دارد و در هش اول قالب اول ورودی این تابع حاصل xor شده کلید ۵۱۲ بیتی با ۵۱۲ بیت تکرار 0x36 است. و در هش دوم قالب اول حاصل xor شده کلید ۵۱۲ بیتی با ۵۱۲ بیت تکرار 0x5C است. و در ادامه قالب های بعدی بخش های ۵۱۲ بیت از پیام اصلی هستند.

تابع pad با ورودی k , l طول کلید دریافتی را به 8 * l بیت گسترش می دهد و این عمل را با اضافه کردن بایت های 0 به کلید انجام می دهد به عبارتی ابتدا کلید را به قالب اسکی تبدیل می کنیم و سپس آن را به تابع pad پاس می دهیم در قالب اسکی هر کاراکتر ۸ بیت خواهد بود لذا اگر کلید متشکل از x تا کاراکتر باشد عملا 8 * x بیت دارد و 8 * x – 512 بیت صفر باید به انتهای آن اضافه شود که این معادل اضافه شدن x – 64 بایت صفر به انتهای پیام است که تابع زیر این کار را انجام می دهد.

```
def pad(a, l):
   while len(a) != l:
      a = a + b'\0'
   return a


  k = pad(k, 64)
```

سپس تابع translate را برای کلید pad شده فراخوانی می کنیم این تابع هر یک از کاراکتر های موجود در کلید را به حاصل xor شده آن کاراکتر با 0x36 یا 0x5C تبدیل می کند به عبارتی یک مپینگ بین کاراکتر های ورودی تابع translate و کلید اتفاق می افتد و هر کاراکتر کلید با کاراکتر نظیرش جایگزین می شود.

```
k_xor_ipad = k.translate(t36)
k_xor_opad = k.translate(t5C)
```

به این ترتیب با این روش یک بار حاصل xor هر کاراکتر با 0x36 یا 0x5C را محاسبه و برای بار های بعدی تماما استفاده میکنیم که بار پردازشی را در کلید های طولانی می تواند به خوبی کاهش دهد.

حال طبق الگوریتم hamc حاصل k_xor_ipad را با متن اصلی که آن را به فرم اسکی تبدیل کرده بودیم کانکت کرده و به تابع هش پاس می دهیم

```
msg = k_xor_ipad + text

h_msg = hashlib.sha256(msg).digest()
```

سپس k_xor_opad را با متن هش بدست آمده در مرحله قبل کانکت کرده و آن را به تابع هش مجددا پاس می دهیم :

```
msg = k_xor_opad + bytes(h_msg)

h_msg = hashlib.sha256(msg).hexdigest()

return h_msg
```

نتیجه بدست آمده همان حاصل hamc می باشد.