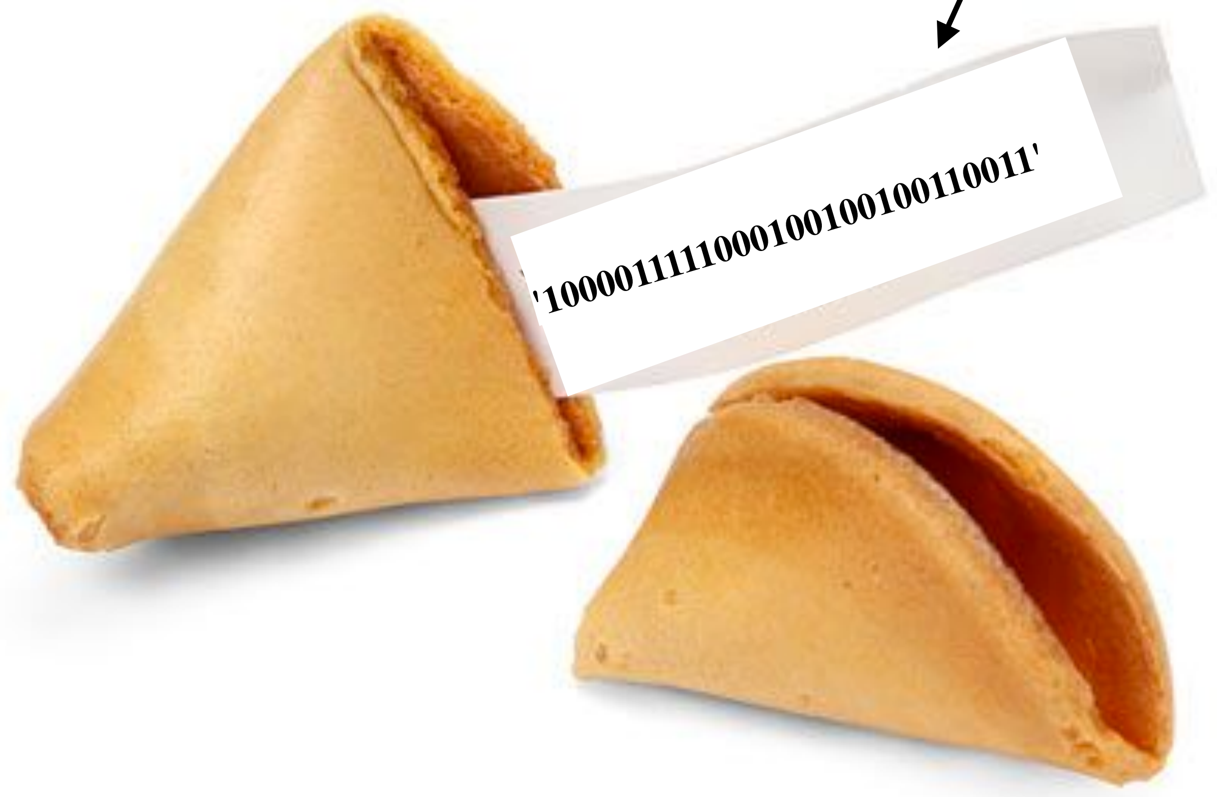
One Time Pad in Python:
A Story in Three Acts

Act 1: The exchange of the key

Alice

Here’s the secret key, don’t lose this.

random binary digits



Sure thing Alice !

Act 2: Alice encodes her message

Message : “ Boo ”

def encode_to_decimal( ):

ASCII: [98, 111, 111]

def decimal_to_binary_string( ):

binary_message: ' 01100010 01101111 01101111'

You already have these functions!

*Notice there’s extra key? that’s ok! We can just toss it

def secret_key_generator(4):

key : ‘01000011 01110001 00010010 00110011'

def one_time_encrypt( ):

Encrypted binary string: ‘00100001 00011110 01111101’

Act 3: Bob receives encrypted message from Alice and decodes

Encrypted binary string: ‘00100001 00011110 01111101’

↓ def apply_key( ):

(After key) decrypted binary string: ' 01100010 01101111 01101111'

↓

Binary to decimal : [98, 111, 111]

↓ decimal to ASCII

Message : ‘Boo’

*you already have this function!

def decode( ):

So what are you supposed to do?

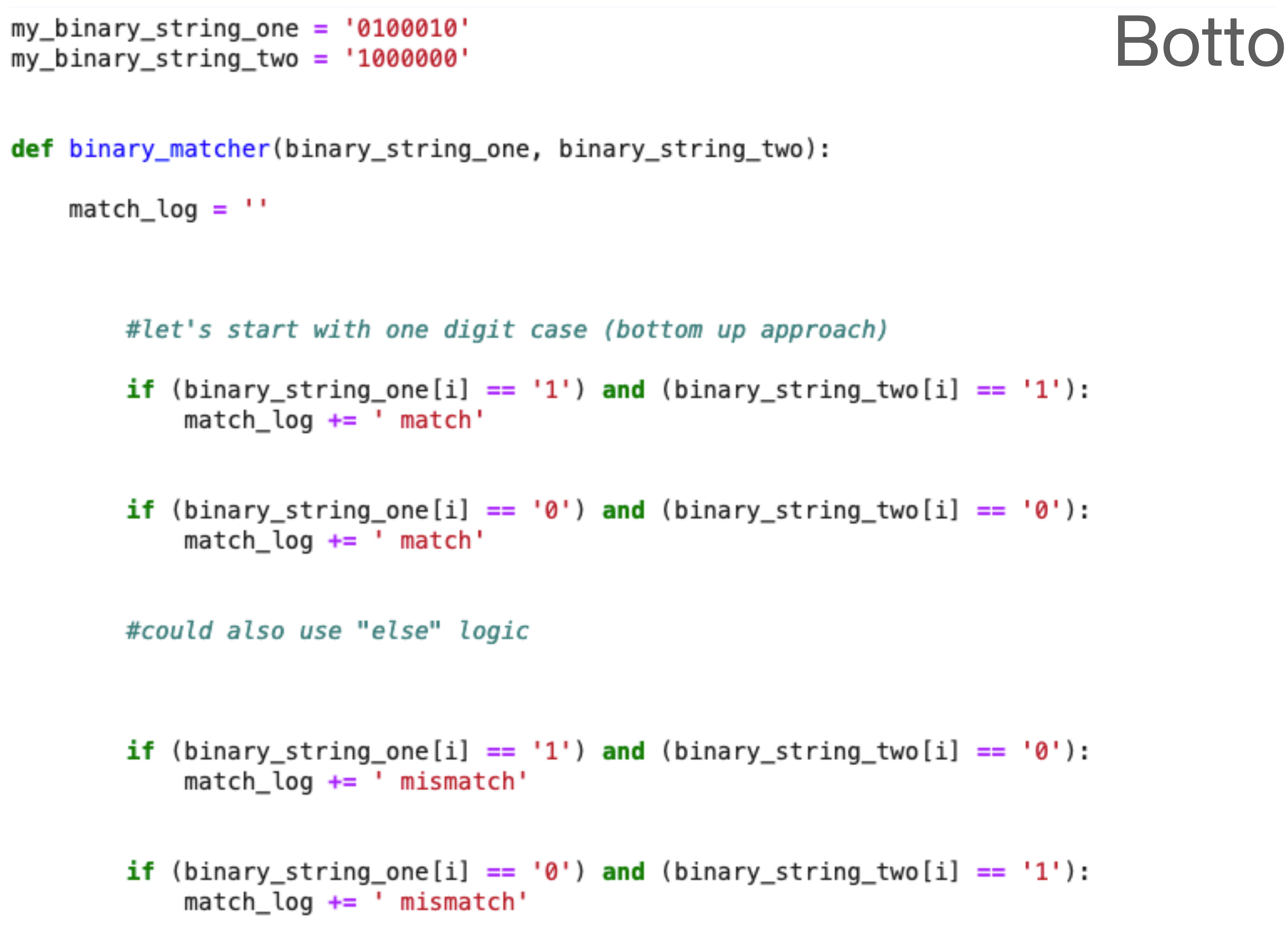
-Take the necessary functions from Symmetric Key encryption assignment
-Add them to new functions in one_time_pad.iynb
- Add comments to the new functions in one_time_pad.iynb describing what they do
- Implement the mod 2 sum logic (see Canvas) in the one_time_encrypt( ) and apply_key( )
functions
- Create a message, and use the functions in the order given in
this video to enact the one time pad protocol. Make sure that your secret key is as long or longer than the message.
- Save your notebook, making sure that all of your outputs at each step are visible
- Answer the questions in Canvas
- Turn in .ipynb

```python
my_binary_string_one = '0100010'
my_binary_string_two = '1000000'


def binary_matcher(binary_string_one, binary_string_two):
```

Goal output:

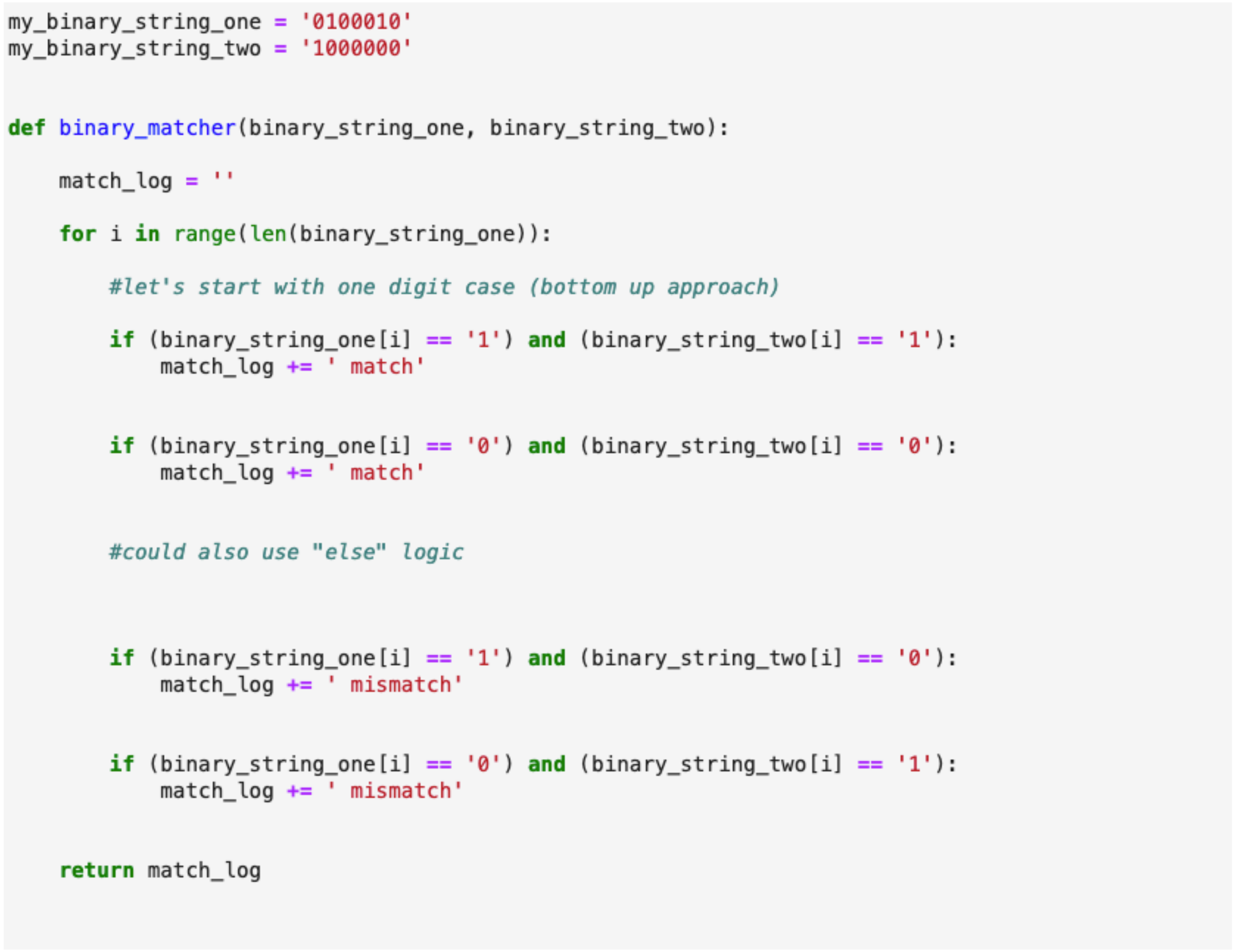
```
' mismatch mismatch match match match mismatch match'
```

Bottom up approach



```python
my_binary_string_one = '0100010'
my_binary_string_two = '1000000'


def binary_matcher(binary_string_one, binary_string_two):

    match_log = ''



        #let's start with one digit case (bottom up approach)

        if (binary_string_one[i] == '1') and (binary_string_two[i] == '1'):
            match_log += ' match'


        if (binary_string_one[i] == '0') and (binary_string_two[i] == '0'):
            match_log += ' match'


        #could also use "else" logic



        if (binary_string_one[i] == '1') and (binary_string_two[i] == '0'):
            match_log += ' mismatch'


        if (binary_string_one[i] == '0') and (binary_string_two[i] == '1'):
            match_log += ' mismatch'
```

Now iterate!



```python
my_binary_string_one = '0100010'
my_binary_string_two = '1000000'


def binary_matcher(binary_string_one, binary_string_two):

    match_log = ''

    for i in range(len(binary_string_one)):

        #let's start with one digit case (bottom up approach)

        if (binary_string_one[i] == '1') and (binary_string_two[i] == '1'):
            match_log += ' match'


        if (binary_string_one[i] == '0') and (binary_string_two[i] == '0'):
            match_log += ' match'


        #could also use "else" logic



        if (binary_string_one[i] == '1') and (binary_string_two[i] == '0'):
            match_log += ' mismatch'


        if (binary_string_one[i] == '0') and (binary_string_two[i] == '1'):
            match_log += ' mismatch'


    return match_log
```

```python
my_match_log = binary_matcher(my_binary_string_one, my_binary_string_two)
```

```python
my_match_log
```