MAC CODE DOCUMENTATION

A Class named **MAC** is created incorporating all required variables and functions

```
class MAC:
    def __init__(self,message,length):
        self.length = length
        self.func_key = ""
        self.message  = message
        if length%4 or len(message) >= 2**(length/4):
            raise Exception("Please Check value of N or message length")
        self.generate_key()
```

In init all the parameters passed to the class PRF are initialized,the parameters are as follows:

length : This variable stores the value of the length of the message(in bits representation)
func_key: This variable stores the value of the key generated.
message: This variable stores the value of the message given in input.
For variable-length classifier length of the MAC, key should be a multiple of 4 and the length of the message should be smaller than $2^{(length/4)}$. Error handling has been done in this part.

```
def generate_key(self):
    for i in range(self.length):
        self.func_key+=str(random.randint(0,1))
```

generate_key :
This function is used to generate an n bit uniform random string, it is used to generate the key.

```python
def generate_message_identifier(self,length):
    message_identifier = ""
    for i in range(length):
        message_identifier+=str(random.randint(0,1))
    return message_identifier
```

generate_message_identifier :
This function is used to generate an n bit uniform random string, it is used as a message identifier.

```python
def pad_sequence(self):
    append_val = 0
    message_length = bin(len(self.message))[2:].zfill(int(self.length/4))
    print(message_length)
    print(len(self.message))
    if (len(self.message))%(int(self.length/4)):
        append_val = int(self.length/4)-(len(self.message)%int(self.length/4))
    for i in range(append_val):
        self.message+='0'
    message_identifier = self.generate_message_identifier(int(self.length/4))
    slice_message = []
    start = 0
    # print("append_val = ",append_val)
    while start<len(self.message):
        # print("Start = ",start)
        slice_message.append(self.message[start:int(start+(self.length/4))])
        start = start + int(self.length/4)
    print(slice_message)
    tag_set = []
    for i in range(len(slice_message)):
        prf = PRF(int(self.func_key,2),len(message_identifier+message_length+bin(
        prf.find_function()
        tag_set.append(prf.output)
    # print("tag_set = ",tag_set)
    final_output = []
    final_output.append((self.func_key,message_identifier))
    for i in tag_set:
        final_output.append(i)
    # print(final_output)
    return final_output
```

pad_sequence function is used in case of variable length MAC , first length of the input message is made a multiple of 4 by appending 0s at the end of it.

After this, the message is sliced into segments of length n/4 each.It is stored in **sliced_message** array.

After this for each part of length n/4 a tag is generated through the PRF, it is shown as follows.

$$t_i \leftarrow \mathsf{Mac}'_k(r\|\ell\|i\|m_i)$$

This concatenation is passed to the input of the given prf in the code and the output tag is appended in the **tag_set** array.
Now final list **final_output** contains all the tags as well a tuple consisting of the function key and message identifier.

```
def verify_variable_length_message(self,encrypt,message):
    # print("insert")
    key_used = encrypt[0][0]
    message_identifier = encrypt[0][1]
    unpadded_length = bin(len(message))[2:].zfill(int(self.length/4))
    append_val = 0
    if len(message)%int(self.length/4):
        append_val = int(self.length/4)-(len(message)%int(self.length/4))
    for i in range(append_val):
        message+='0'
    # flag = 0
    if (len(message)/int(self.length/4))!=(len(encrypt)-1):
        return 0
    start = 0
    slice_message = []
    while start<len(self.message):
        # print("Start = ",start)
        slice_message.append(self.message[start:start+int(self.length/4)])
        start = start + int(self.length/4)
    print(slice_message)
    # print("Here")
    verify_tags = []
    for i in range(len(slice_message)):
        # prf = PRF(int(key_used,2),len(i),i)
        prf = PRF(int(key_used,2),len(message_identifier+unpadded_length+bin(i+1)[2:].zfill(int(s
        prf.find_function()
        verify_tags.append(prf.output)
    # print("verify tags = ",verify_tags)
    for i in range(1,len(encrypt)):
        if (encrypt[i] != verify_tags[i-1]):
            return 0
    return 1
```

The function **verify_variable_length** message takes input tags in addition to the message identifier and function key.

Its working is exactly similar to the **pad_sequence** function the only difference being here it takes the given key, and message identifier and generated tags are verified against the input tags, if all of them are equal then 1 is returned otherwise 0 is returned.

```
def encrypt(self):
    prf = PRF(int(self.func_key,2),self.length,self.message)
    prf.find_function()
    tag = prf.output
    encrypted_output = (tag,self.message,self.func_key)
    return encrypted_output
```

This (**encrypt**) function generates tag in the case of fixed length MACs, the message is sent as an input to the prf, and the tag is generated, then a tuple consisting of the message key and tag are generated and returned from the function.

```
def verify(self,encrypted):
    given_tag = encrypted[0]
    prf = PRF(int(encrypted[2],2),self.length,encrypted[1])
    prf.find_function()
    generated_tag = prf.output
    # print("given tag = ",given_tag,"generated tag = ",genera
    if given_tag==generated_tag:
        return 1
    else:
        return 0
```

This (**verify**) function is used in case of fixed length MAC, here it takes tags, message, and function key as an input and it generates the tag using the same method it generates the tag again and it is verified against the

previously given tags, if they are equal then 1 is returned otherwise 0 is returned.

```
if __name__=="__main__":
    # message = int(generate_key(17),2)
    message = generate_key(55)
    print("message = ",message)
    print("converted = ",int(message,2))
    mac = MAC(message,40)
    # cipher_value = mac.encrypt() #Encrypting the message
    # print(cipher_value)
    # print(mac.verify(cipher_value))
    key = mac.pad_sequence()
    print("key = ",key)
    print(mac.verify_variable_length_message(key,message))
    # decrypt_value = cpa.decrypt(cipher_value) #Decrypting
    # print("decrypt_valye = ",decrypt_value)
```

The main function takes all the required inputs of mac and required messages and keys.