DETAILED EXPLANATION OF THE ALGORITHM/OUR CODE:

AES is a block cipher with a block length of 128 bits, that allows for key lengths of 128 or 256 bits. Encryption with 128-bit keys requires 10 rounds of processing and 14 rounds for 256-bit keys. Each of these rounds involves performing a single-byte based substitution step, a row-shifting step, a column-wise mixed matrix multiplication step, and the addition of the round key, but the order of the steps differs between encryption and decryption (the steps are reversed in decryption).

File I/O (Reading Plaintext and Ciphertext):

The very first thing we do is read the passed-in input file. Un-encrypted input files go through *read_and_pad()*, where they are read into a 2D array, *state*, until the EOF is reached, at which point they are padded with 0-bytes until they are at a length of 16 bytes, with the last byte indicating how many padded bytes were added. If the initial file was already at a multiple of 16, which we keep track of with our *real_byte_count*, we simply append 15 0-byte blocks and a 0x10 byte at the end to show that this entire block was padded and not part of the initial file. Encrypted files do not need to be padded, so they are simply read in using the *read()* function.

Key Expansion:

Once the input file has been read, we perform a key expansion on the key given through the key file. The bit size indicated by the first argument determines whether we expand the key into either 11 or 15 16-byte round keys, for 128 and 256 bits respectively. During key expansion, we initially append the original key and then build out from there, looping until we have reached our desired key length. Since each round key consists of 4 4-byte blocks, we calculate everything in terms of 16 bytes. We get the expanded key column for a given round from the corresponding 4 columns of the round key for the previous round, which is why use *'(num_bytes_generated % 16) // 4) - 1'* to get the column.

On every 4th column (or 16 bytes later) we perform the core key expansion operation using that key column. The *key_expansion_core()* function takes in this column and the number of iterations the expansion has undergone. It first rotates the column one element to the left. Then the iteration number determines which byte from the Rcon lookup table to XOR the first byte of the given column with, once it has been replaced with its corresponding value from the AES S-Box table. The S-Box table indices are calculated in *get_sbox_indices()* based on the int value of the current byte. Finally, every round key is used to generate the new column by being XORed with the corresponding column in the previous 4-column grouping before being added to the final *expanded_key*, which becomes an array of 4-byte blocks. Every time the next 4-byte block has been appended, which we keep track of with our *ek_index* variable, we append an empty list to ensure we maintain a 4-column grouping in our expanded key.

Add Round Key:

The first method that gets called in encryption *add_round_key()*, which takes in the block to be encrypted and the round key to be used, which is simply one of the arrays in *expanded_keys*. The function simply loops through the 4 columns in the box and returns it after it has been XORed with the corresponding element in the passed in *round_key*.

Sub Bytes:

sub_bytes(), or *inv_sub_bytes()* for decryption*,* takes in the block to be encrypted/decrypted as a parameter. This function replaces each value in the block with the corresponding values in the AES S-box. The indices within the S-box are found by getting the block's least significant nibble for the column index and the most significant nibble for the row index. Both of these values are found by ANDing the current value with masks defined in *utils.py*. Decryption uses the *inv_sub_bytes()* function, which is essentially the same but uses the inverse S-box table instead.

Shift Rows:

Since columns are represented as rows in our implementation, we needed to transpose the block before and after calling *shift_rows()* and *inv_shift_rows()*. These functions shift the bytes in the first row by 1 spot, bytes in the second row by 2 spots, and so on. If the message is being encrypted, the shift is to the left, and if it is being decrypted, the shift is to the right.

Mix Columns:

The final function is *mix_columns()* or *inv_mix_columns()*. The mix column step replaces each byte of a column by a function of all the bytes in the same column. So for encryption, each byte in a column is replaced by 2 times that byte, XORed with 3 times the next byte, XORed with the 2 bytes that follow. The values have to be XORed instead of added because this step uses Galois arithmetic. For decryption, each byte in a column is replaced by 14 times that byte, XORed with 11 times the next byte, XORed with 13 times the next byte, XORed with the following byte. We use lookup tables to perform the multiplication.

HOW TO RUN:

Our program can be run in the following fashion:is step

./aes.py <key size> <key file> <input file name> <output file name> <mode>

EX: ./aes.py 128 test/key test/input test/output encrypt

EX: ./aes.py 128 test/key test/output test/plaintext decrypt

For the key size, we have set the options to include 128 and 256 bits to select between, and for the mode, we set the options to include ‘encrypt, ‘e’, ‘decrypt’, and ‘d’ to indicate whether they want the message in the input file to be encrypted or decrypted.