license_re.py

Write a Python program called `license_re.py` that will create a regular expression for a license plate that accounts for characters and numbers which might be confused according to the following list:

- 5 S
- X K
- 1 I
- 3 E
- 0 O Q
- M N
- U V W
- 2 8

Print the plate, the regular expression that would match that plate with all possible ambiguities, and then print all possible combinations of plates that includes the options along with the result of comparing the regular expression you created to the generated plate.

```
$ ./license_re.py
usage: license_re.py [-h] PLATE
license_re.py: error: the following arguments are required: PLATE
$ ./license_re.py ABC1234
plate = "ABC1234"
regex = "^ABC[1I][27][3E]4$"
ABC1234 OK
ABC12E4 OK
ABC1734 OK
ABC17E4 OK
ABCI234 OK
ABCI2E4 OK
ABCI734 OK
ABCI7E4 OK
$ ./license_re.py 123456
plate = "123456"
regex = "^[1I][27][3E]4[5S]6$"
123456 OK
1234S6 OK
12E456 OK
12E4S6 OK
173456 OK
1734S6 OK
17E456 OK
17E4S6 OK
I23456 OK
I234S6 OK
```

```
I2E456 OK
I2E4S6 OK
I73456 OK
I734S6 OK
I7E456 OK
```

Discussion

Owing to the vagaries of the typefaces chosen by different states as well as the wear of the plates themselves, it would seem to me that people might easily confuse certain letters and numbers on plates. In the above example, `ABC1234`, the number `1` might look like the letter `I`, so the plate could be `ABD1234` or `ABCI234`. Granted, most license plates follow a pattern of using only letters in some spots and numbers in others, e.g., 3 letters plus 4 numbers, but I want to focus on all possibilities in this problem both because it makes the problem a bit easier and also because it doesn't have to worry about how each state formats their plates. Additionally, I want to account for customized plates that do not follow any pattern and might use any combination of characters.

I represented the above confusion table as a list of tuples. At first I though I might use a dictionary, but there is a problem when three characters are involved, e.g., `0`, `O`, and `Q`. I iterate through each character in the provided plate and decide if the character exists in any of the tuples. If so, I represent that position in the regular expression as a choice; if not, it is just the character.

If you think about a regular expression as a graph, it starts with the first character, e.g., `A` which must be followed by `B` which must be followed by `C` which must be followed by either a `1` or an `I` which must be followed by a `2` or a `7`, etc.

```
                  1        2        3
A -> B -> C -> <   > -> <   > -> <   > -> 4
                  I        7        E
```

In creating all the possible plates from your regular expression, you are making concrete what the regular expression is, well, ... expressing. I find `itertools.product` to be just the ticket for creating all those possibilites, which must be sorted for the sake of the test.

Test Suite

```
$ make test
pytest -v test.py
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.3.1, py-1.8.0, pluggy-0.9.0 -- /Users/kyclark/anac
```

```
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/python/practical_python_for_data_science/ch08-regular-expressio
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.3.0, arraydiff-0.3
collected 3 items

test.py::test_usage PASSED                                               [ 33%]
test.py::test_accept_01 PASSED                                           [ 66%]
test.py::test_accept_02 PASSED                                           [100%]

========================== 3 passed in 0.22 seconds ==========================
```