Other Fun Stuff in the itertools Module

```
import itertools
print (list(itertools.product('ABC', '123')))   #①
#[('A', '1'), ('A', '2'), ('A', '3'),
# ('B', '1'), ('B', '2'), ('B', '3'),
# ('C', '1'), ('C', '2'), ('C', '3')]

print (list(itertools.combinations('ABC', 2)))  #②
#[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

① The `itertools.product()` function returns an iterator containing the Cartesian product of two sequences.

② The `itertools.combinations()` function returns an iterator containing all the possible combinations of the given sequence of the given length. This is like the `itertools.permutations()` function, except combinations don't include items that are duplicates of other items in a different order. So `itertools.permutations('ABC', 2)` will return both `('A', 'B')` and `('B', 'A')` (among others), but `itertools.combinations('ABC', 2)` will not return `('B', 'A')` because it is a duplicate of `('A', 'B')` in a different order.

```
names = list(open('favorite-people.txt', encoding='utf-8'))          #①
print (names)
#['Dora\n', 'Ethan\n', 'Wesley\n', 'John\n', 'Anne\n','Mike\n', 'Chris\n', 'Sarah\n', 'Alex\r

names = [name.rstrip() for name in names]                            #②
print (names)
#['Dora', 'Ethan', 'Wesley', 'John', 'Anne','Mike', 'Chris', 'Sarah', 'Alex', 'Lizzie']

names = sorted(names)                                                #③
print (names)
#['Alex', 'Anne', 'Chris', 'Dora', 'Ethan','John', 'Lizzie', 'Mike', 'Sarah', 'Wesley']

names = sorted(names, key=len)                                       #④
print (names)
#['Alex', 'Anne', 'Dora', 'John', 'Mike','Chris', 'Ethan', 'Sarah', 'Lizzie', 'Wesley']
```

① This idiom returns a list of the lines in a text file.

② Unfortunately (for this example), the `list(open(filename))` idiom also includes the carriage returns at the end of each line. This list comprehension uses the `rstrip()` string method to strip trailing whitespace from each line. (Strings also have an `lstrip()` method to strip leading whitespace, and a `strip()` method which strips both.)

③ The `sorted()` function takes a list and returns it sorted. By default, it sorts alphabetically.

④ But the `sorted()` function can also take a function as the `key` parameter, and it sorts by that key. In this case, the sort function is `len()`, so it sorts by `len(each item)`. Shorter names come first, then longer, then longest.

What does this have to do with the `itertools` module? I'm glad you asked.

```
import itertools
names = list(open('favorite-people.txt', encoding='utf-8'))

groups = itertools.groupby(names, len)  #①
print (groups)
#<itertools.groupby object at 0x7f45b71409a8>

print (list(groups))
#[(5, <itertools._grouper object at 0x7f45b7143c18>),
# (6, <itertools._grouper object at 0x7f45b714dc18>),
# (7, <itertools._grouper object at 0x7f45b714dbe0>),
# (5, <itertools._grouper object at 0x7f45b714de80>),
# (6, <itertools._grouper object at 0x7f45b714deb8>),
# (5, <itertools._grouper object at 0x7f45b714def0>),
# (6, <itertools._grouper object at 0x7f45b714df28>)]

groups = itertools.groupby(names, len)   #②
for name_length, name_iter in groups:    #③
     print('Names with {0:d} letters:'.format(name_length))
     for name in name_iter:
         print(name)

#Names with 4 letters:
#Alex
#Anne
#Dora
#John
#Mike
#Names with 5 letters:
#Chris
#Ethan
#Sarah
#Names with 6 letters:
```

```
#Lizzie
#Wesley
```

① The `itertools.groupby(`) function takes a sequence and a key function, and returns an iterator that generates pairs. Each pair contains the result of `key_function(each item)` and another iterator containing all the items that shared that key result.

② Calling the `list(`) function "exhausted" the iterator, i.e. you've already generated every item in the iterator to make the list. There's no "reset" button on an iterator; you can't just start over once you've exhausted it. If you want to loop through it again (say, in the upcoming `for` loop), you need to call `itertools.groupby()` again to create a new iterator.

③ In this example, given a list of names *already sorted by length*, `itertools.groupby(names, len)` will put all the 4-letter names in one iterator, all the 5-letter names in another iterator, and so on. The `groupby()` function is completely generic; it could group strings by first letter, numbers by their number of factors, or any other key function you can think of.

> *The **itertools.groupby()** function only works if the input sequence is already sorted by the grouping function. In the example above, you grouped a list of names by the **len()** function. That only worked because the input list was already sorted by length.*

Are you watching closely?

```
import itertools

print (list(range(0, 3)))
#[0, 1, 2]

print (list(range(10, 13)))
#[10, 11, 12]

print (list(itertools.chain(range(0, 3), range(10, 13))) )      #①
#[0, 1, 2, 10, 11, 12]

print (list(zip(range(0, 3), range(10, 13))) )                  #②
#[(0, 10), (1, 11), (2, 12)]
```

```
print (list(zip(range(0, 3), range(10, 14))))                    #③
#[(0, 10), (1, 11), (2, 12)]

print (list(itertools.zip_longest(range(0, 3), range(10, 14))))  #④
#[(0, 10), (1, 11), (2, 12), (None, 13)]
```

① The `itertools.chain()` function takes two iterators and returns an iterator that contains all the items from the first iterator, followed by all the items from the second iterator. (Actually, it can take any number of iterators, and it chains them all in the order they were passed to the function.)

② The `zip()` function does something prosaic that turns out to be extremely useful: it takes any number of sequences and returns an iterator which returns tuples of the first items of each sequence, then the second items of each, then the third, and so on.

③ The `zip()` function stops at the end of the shortest sequence. `range(10, 14)` has 4 items (10, 11, 12, and 13), but `range(0, 3)` only has 3, so the `zip()` function returns an iterator of 3 items.

④ On the other hand, the `itertools.zip_longest()` function stops at the end of the *longest* sequence, inserting `None` values for items past the end of the shorter sequences.

OK, that was all very interesting, but how does it relate to the alphametics solver? Here's how:

```
characters = ('S', 'M', 'E', 'D', 'O', 'N', 'R', 'Y')
guess = ('1', '2', '0', '3', '4', '5', '6', '7')
print (tuple(zip(characters, guess)))  #①
#(('S', '1'), ('M', '2'), ('E', '0'), ('D', '3'), ('O', '4'), ('N', '5'), ('R', '6'), ('Y', '

print (dict(zip(characters, guess)))   #②
#{'O': '4', 'N': '5', 'S': '1', 'R': '6', 'M': '2', 'Y': '7', 'E': '0', 'D': '3'}
```

① Given a list of letters and a list of digits (each represented here as 1-character strings), the `zip` function will create a pairing of letters and digits, in order.

② Why is that cool? Because that data structure happens to be exactly the right structure to pass to the `dict()` function to create a dictionary that uses letters as keys and their associated digits as values. (This isn't the only way to do it, of course. You could use a dictionary comprehension to create the dictionary directly.) Although the printed representation of the dictionary lists the pairs in a different order (dictionaries have no "order" per se), you can see that each letter is associated with the digit, based on the ordering of the original `characters` and `guess` sequences.

The alphametics solver uses this technique to create a dictionary that maps letters in the puzzle to digits in the solution, for each possible solution.

```
characters = tuple(ord(c) for c in sorted_characters)
digits = tuple(ord(c) for c in '0123456789')
#...
for guess in itertools.permutations(digits, len(characters)):
    #...
    equation = puzzle.translate(dict(zip(characters, guess)))
```

But what is this `translate()` method? Ah, now you're getting to the really fun part.