Programming Language Learning Series
Mastery of Python Language
(Interview Questions/Assignment-Functional Style)

Q1: Write a function to compute 1/2+2/3+3/4+...+n/n+1 with a given n (n>0).

Q2: Write a function to find the sum of all the multiples of 3 or 5 below 1000.

Q3: A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009 = 91 × 99. Write a function to find the largest palindrome made from the product of two 3-digit numbers.

Q4: We count 35 heads and 94 legs among the chickens and rabbits in a farm. Write a python function that returns how many rabbits and how many chickens do we have.

Q5: Given a text file as input, we are interested to computing the following text analytics on that input:

- Compute the number of words in the given file
- Find the 10 most frequent words in the given file
- Find the number of times a given word appears in the file

Assuming that we want to develop a solution for the required text analytics using procedural abstractions. Which abstraction do you prefer and why?

Starting with `range(1, 20)`, make a list of the squares of each odd number in the following ways

- With a for loop
- Using a list comprehension
- Using map and filter

Rewrite the following code into functional form using lambdas, map, filter and reduce.

- `In [43]:`
- `n = 10`
- `s = 10`
- `for i in range(n):`

- `    if i % 2:`
- `        s |= i**2`
- `s`

How many numbers in `range(100, 1000)` are divisible by 17 after you square them and add 1? Find this out using only **lambda** functions, **map**, **filter** and **reduce** on `xs`, where `xs = range(100, 10000)`.

In pseudo-code, you want to achieve

```
xs = range(100, 10000)
count(y for y in (x**2 + 1 for x in xs) if y % 17 == 0)
```

Write a `flatmap` function that works like `map` except that the function given takes a list and returns a list of lists that is then flattened (4 points).

In other words, `flatmap` takes two arguments, a function and a list (or other iterable), just like `map`. However the function given as the first argument takes a single argument and returns a list (or other iterable). In order to get a simple list back, we need to unravel the resulting list of lists, hence the flatten part.

For example,

```
flatmap(lambda x: x.split(), ["hello world", "the quick dog"])
```

should return

```
["hello", "world", "the", "quick", "dog"]
```