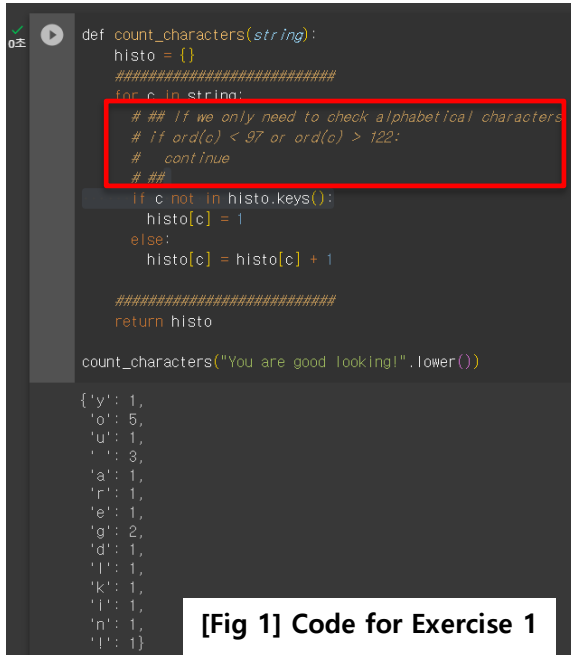


## Summary Report [Week 2]. Python Basics

20190106 KimByungjun (김병준)

### ● Exercises

1) Count the number of each character in the given sentences.



```
def count_characters(string):
    histo = {}
    #####
    for c in string:
        ## If we only need to check alphabetical characters
        # if ord(c) < 97 or ord(c) > 122:
        #     continue
        ##
        if c not in histo.keys():
            histo[c] = 1
        else:
            histo[c] = histo[c] + 1

    #####
    return histo

count_characters("You are good looking!".lower())
```

```
{'y': 1,
'o': 5,
'u': 1,
'a': 3,
'e': 1,
'r': 1,
'g': 1,
'd': 2,
'l': 1,
'k': 1,
'i': 1,
'n': 1,
' ': 1}
```

[Fig 1] Code for Exercise 1

#### < Explanation >

해당 Python code 에서 def keyword 를 이용해 **string** 변수를 받고 해당 string 에 각 문자가 몇 개 있는 지를 나타내는 **histo** 라는 **dictionary** 변수를 반환하는 count\_characters 라는 함수를 정의하였다.

기본 structure code 로 주어진 dictionary 변수 histo 를 이용하여야 하므로 dictionary 형 변수가 가지는 장점이자 큰 특징인 key-value pair 성질을 이용하고자 한다.

이 때, key 값은 각 문자(character)가 되며, key 에 해당하는 value 는 input string 내 해당 문자가 나온 총 횟수를 저장한다. (단, 해당 Exercise 에서 character 는 알파벳 뿐만 아니라 ASCII Code 에 저장된 공백, non-alphabetical character 등을 모두 지칭한다.)

Code flow 는 아래와 같이 진행된다.

**Step 1.** 우선 empty dictionary 인 histo 를 선언한다.

**Step 2.** for statement 를 이용하여 string 내 모든 문자를 c 라는 변수에 한 번에 하나씩 차례대로 받는다.

**Step 3.** if statement 에서 만약 histo 딕셔너리 내에 c 를 key 값으로 가지는 pair 가 없다면, histo[c] = 1 로 초기값을 넣어준다.

**Step 4.** 만약, 이미 histo 딕셔너리 내에 c key 를 가지는 key-value pair 가 존재한다면 해당 pair 의 value 값을 1 증가(increment) 시켜준다. (histo[c] += 1 로도 표현 가능하다.)

**Step 5.** for loop 종료 후, count\_characters 함수는 완성된 histo 딕셔너리를 반환한다.

**Result.** "You are good looking!".lower() 라는 소문자 처리가 된 string (이 경우, 대문자 Y 과 소문자 y 로 변환된다.) 를 input 으로 받은 count\_characters 함수가 [Fig 1]에서 보이는 dictionary 를 반환한다. 이 때, key 의 order 는 string 의 맨 앞에서부터 받은 문자 순서대로 나타나며 정렬되어 있지 않다. 예로, g 다음 d 가 나타난 이유는 you 에서 이미 o 에 대한 key-value pair 가 형성되어 있기 때문이다.

**Note.** Input string 이 소문자만 가지고 있다는 전제하에 만일 alphabetic 문자의 개수만 체크하고 싶다면 위 [Fig 1] 에서 빨간색 박스처리 된 code 부분을 활성화시키면 된다. a 에서 z 까지의 ASCII 코드가 97~122 에 해당하며, 이 외의 문자(공백, non-alphabetical character 등)는 continue 문을 통해 다음 loop 로 skip 된다.

## 2) Check if the given digits are prime numbers.



```
def check_prime(number):
    counter = 0
    #####

    if number <= 1 :
        print(False)
        return

    for i in range(2, number):
        if number % i == 0:
            counter = 1
            break

    if counter == 1:
        print(False)
    else:
        print(True)

    #####

check_prime(2)
check_prime(5)
check_prime(12)
```

True  
True  
False

[Fig 2] Code for Exercise 2

### < Explanation >

counter 변수는 나중에 소수 판별 결과를 나타내 줄 flag 변수로 사용한다. 해당 문제에서 받는 input 인 number 는 정수형이라 가정하도록 하겠다.

**Step 1.** counter 변수를 0 으로 선언 및 initialize 한다.

**Step 2.** 만일 number 가 1 이하 인 경우, 2 이상부터 존재하는 소수가 나올 수 없는 범위이므로 False 를 출력한다.

**Step 3.** 기본적인 소수 판별의 원리는 숫자 number 본인을 본인보다 작은 2 이상의 자연수들로 나눴을 때, 모든 경우에서 나누어 떨어지지 않으면 소수, 한 번이라도 나누어 떨어지면 소수가 아닌 합성수로 판별하는 것이다.

2 부터 number-1 까지의 자연수를 차례로 i 라는 변수에 저장하여 number 를 i 로 나누는 과정을 나타내기 위해 for statement 를 이용할 것이다.

**Step 4.** if number % i == 0 를 통해 나누어 떨어짐이 발생하는 경우 counter 를 0 에서 1 로 바꾸며 break 를 통해 for loop 를 나간다.

**Step 5.** counter 의 값이 1 이면 합성수(False), 0 이면 소수(True)를 출력한다.

**Result.** [Fig 2]의 결과처럼 2, 5 는 소수라 True 를 반환하며, 12 는  $12 = 2^2 \times 3$  로 소인수분해가 되는 합성수이므로 False 를 반환한다. 만약, check\_prime 에 정수형 input 을 넣지 않은 경우(e.g. float), for 문에서 TypeError 가 발생하며 code 는 정상적으로 실행되지 않는다.

**Note.** for 문을 이용할 때 number 를 나눌 숫자 i 가 number-1 까지 할 필요없이 number 의 제곱근( $\sqrt{\text{number}}$ ) 까지만 해도 소수 판별이 가능하다. 다만, python 에서 제곱근 사용 시, math 라이브러리를 함수 밖에서 import 하여 math.sqrt() 함수를 사용해야 하므로, skeleton code 가 지정된 해당 exercise 에선 이용하지 않았다.

## 3) Compute i'th element of Fibonacci sequence. ( $F_0 = 0, F_1 = 1$ )

피보나치 수열을 계산할 때, return fibo(n-1) + fibo(n-2) 문을 통해 brute-force 하게 일반적인 recursive 한 programming 을 하게 되는 경우 input n 에 대해 exponential 한 횟수(time complexity :  $O(2^n)$ )로 연산을 하게 되고, 이에 따라 large n 에 대해 매우 오랜 시간 연산을 해야 한다.

이를 해결하기 위해, 한 번 계산한 피보나치 수열의 값을 저장해 뒀 재연산을 방지하는 dynamic programming 을 제시하였다. 이 경우, input n 에 대해 linear 한 횟수로 연산하여(time complexity :  $O(n)$ ) 상대적으로 매우 빠르게 연산할 수 있으나, linear 한 크기의 memory 를 가진 list 를 사용한다는 단점이 있다.

```

def fibo(n):
    #####
    if n < 0:
        return "Invalid Input"
    elif n == 0:
        return 0
    elif n == 1:
        return 1

    # Dynamic Programming : Store values into list (n>=2)
    arr = [-1] * (n+1)
    arr[0] = 0
    arr[1] = 1

    for i in range(2,n+1):
        arr[i] = arr[i-1] + arr[i-2]
    return arr[n]

    # Brute Force
    # if n == 0:
    #     return 0
    # elif n == 1:
    #     return 1
    # else:
    #     return fibo(n-1) + fibo(n-2)

    #####
print(fibo(10))

```

[Fig 3-1] Code for Exercise 3

### < Explanation >

**Step 1.** Recursive function 에서 첫번째로 중요한 것은 초기값의 설정이다. fibo(n)이란 함수를 정의하며,  $n < 0$  일 때는 오류 문구("Invalid Input")를, 0 일 때는  $0(F_0 = 0)$ , 1 일 때는  $1(F_1 = 1)$ 을 반환한다.

**Step 2.** 배열의 길이가  $n+1$  인 list 형 변수 arr 을 만들고, 모든 값을 -1 로 초기화한다. 그 뒤,  $F_0 = 0, F_1 = 1$  을 표현하기 위해  $arr[0] = 0, arr[1] = 1$  로 값을 바꾼다. 목표는  $F_n$  을 나타내는  $arr[n]$  을 구하는 것으로, 이를 위해  $n+1$  길이의 리스트를 만들었다.

**Step 3.** for 문을 이용하여  $arr[2]$  부터  $arr[n]$  까지 피보나치 수열의 정의에 따라  $arr[i] = arr[i-1] + arr[i-2]$  ( $i = 2, 3, \dots, n$ ) 로 연산하였다. 마지막으로,  $arr[n]$  을 반환한다.

**Result.** [Fig 3-1]의 결과처럼 10 번째 피보나치 수열의 값은 0,1,1,2,3,5,8,13,21,34,55...로 55 의 값을 출력한다.

**Note.** 아래 [Fig 3-2]와 같이 fibo(n)은 위에서 설명한 대로 list 에 값을 저장하는 Dynamic Programming 을 한 함수, fibo2(n)는 brute-force 한 오래 걸리는 피보나치 수열을 계산하는 함수로 정의하였다. 그리고, time 라이브러리의 time 모듈을 **from time import time** 을 선두에 선언해 모듈 호출을 하였고, 각 함수가 실행되는 시간을 아래 빨간색 박스처럼 계산해보았다. 이 때, input 크기가 30 일 때, 두 함수의 실행시간이 매우 큰 차이가 남을 알 수 있으며, input 크기가 40 정도만 되어도 fibo2 함수의 연산이 매우 오래 걸림을 알 수 있었다.

```

def fibo2(n):
    ##Brute Force
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return (fibo2(n-1) + fibo2(n-2))

    #####
t0 = time()
print(fibo(30))
t1 = time()
print("Dynamic Programming : ", t1-t0)

t2 = time()
print(fibo2(30))
t3 = time()
print("Brute Force : ", t3-t2)

```

832040  
Dynamic Programming : 0.0023276805877685547  
832040  
Brute Force : 0.4087456512451172

[Fig 3-2] Time consumption during different Fibonacci functions