하나님은 모든 사람이 구원을 받으며 진리를 아는 데에 이르기를 원하시느니라 (딤전2:4)

NOTE: The following materials have been compiled and adapted from the numerous sources including my own. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Send any comments or criticisms to `idebtor@gmail.com` Your assistances and comments will be appreciated.

Strings

학습 목표

- 문자열에 대해 이해한다.
- 파이썬에서 제공하는 문자열 메소드를 살펴본다.

학습 내용

1. 슬라이싱 문법
2. 다양한 문자열 메소드
3. 문자열 포멧
4. 데이터 보관 방법

A string can be read from the standard input using the function `input()` or defined in single or double quotes. Two strings can be concatenated, and we can also repeat a string n times multiplying it by integer.

Let's look at how you can create simple strings with a pair of single, double or triple quote.

In [ ]:
```
x = "Sola WnSola"
print(x)
```

You can also use **single quotes** or **triple quotes**: 'Fide'

In [ ]:
```
y = None
print(y)
```

There's no difference between using single and double quotes, but you need to be consistent in which one you choose for a given string.

You can add two strings **using** `+` , concatenating them together: x + y

In [ ]:
```
z = None
print(z)
```

You can also **multiply strings by an integer**, which duplicates the string that number of times:

Complete the code such that it prints **`Halleluja Halleluja Halleluja`**

In [ ]:
```
s = None
print(s)
```

You can get the length of a string with the generic function `len()` which operates on any Python sequence, but works with strings in particular:

In [ ]:
```
s = "In the beginning God created the heavens and the earth"
None
```

Example 1:

Complete the code such that it counts the number of digits in `x = 2**1000`.

In [1]:
```
x = 2 ** 1000
None
```

Example 2: Count the number of digits in `x = 2**1000` without using the type conversion function `str()`, but using a `while` loop and `//`.

In [ ]:
```
x = 2 ** 1000
count = 0
None
```

Example 3:

Which one is faster between Example 1 and 2 shown above?

- Use %%timeit cell magic to get an average elapsed time of a cell execution.

Slice a single character

A **slice** gives from the given string one character or some fragment: substring or subsequence.

There are three forms of slices. The simplest form of the slice: a single character slice `S[i]` gives ith character of the string. We count characters starting from 0. That is, if `S = 'Hello'`, `S[0] == 'H'`, `S[1] == 'e'`, `S[2] == 'l'`, `S[3] == 'l'`, `S[4] == 'o'`. Note that in Python there is no separate type for characters of the string. `S[i]` also has the type `str`, just as the source string.

Number `i` in `S[i]` is called an index.

If you specify a negative index, then it is counted from the end, starting with the number `-1`. That is, `S[-1] == 'o'`, `S[-2] == 'l'`, `S[-3] == 'l'`, `S[-4] == 'e'`, `S[-5] == 'H'`.

Example: double_char()

Given a string, this function returns a string where for every char in the original, there are two chars.

For example:

```
double_char('The') ---> 'TThhee'
double_char('AAbb') ---> 'AAAAbbbb'
double_char('Hi-There') ---> 'HHii--TThheerree'
```

Solution: Using slicing of each character in `str` object

In [ ]:
```
def double_char(st):
    None
```

Exptected Output:

```
ok ---> ookk
abba ---> aabbbbaa
Word! ---> WWoorrdd!!
Hi ---> HHii
```

In [ ]:
```
list = ['ok', 'abba', 'Word!', 'Hi']
for s in list:
    print(s, '--->', double_char(s))
```

Another solution: Using the fact that a `str` object is iterable.

In [ ]:
```
def double_char(st):
    None
```

In [ ]:
```
list = ['ok', 'abba', 'Word!', 'Hi']
for s in list:
    print(s, '--->', double_char(s))
```

Slicing substring

Slicing is used to extract a subsequence out of your sequence(e.g. list, string):

```
var[lower:upper:step]
```

The element which has index equal to the lower bound is included in the slice, but the element which has index equal to the upper bound is excluded, so mathematically, the ie slice is `[lower, upper)`. If you think of the indices as being between the elements, then this mentally works very nicely, as we'll see.

The `step` argument is optional, and indicates the strides between elements in the subsequence, so a step of 2 takes every second element.

List slicing also has very similar semantics to slicing on strings.

```
- a[start:end]  items start through end-1
- a[start:]     items start through the rest of the array
- a[:end]       items from the beginning through end - 1
- a[:]          a copy of the whole list
- a[start:end:step]  start through end - 1, by step
```

Let's see the examples: Why don't you write your answers first and then run the code to compare?

In [ ]:
```
s = 'gratia'
print(s[1])
print(s[-1])
print(s[1:3])
print(s[1:-1])
print(s[:3])
print(s[2:])
print(s[:-1])
print(s[::2])
print(s[1::2])
print(s[::-1])
print(s[:])
```

Slice - immutability of strings

Any slice of a string creates a new string and never modifies the original one. In Python strings are **immutable**, i.e they can not be changed as the objects. You can only assign the variable to the new string, but the old one stays in memory.

In fact in Python there is no variables. There are only the names that are associated with any objects. You can first associate a name with one object, and then — with another. Can several names be associated with one and the same object.

Example 1: change `hello1` to `jello` by slicing

In [ ]:
```
s = 'hello'
print(None)
s = 'jello'
print(None)
```

In [ ]:
```
s = 'hello'
s[0] = 'j'     # TypeError
```

Then how do you make a new string `'jello'` out of `'hello'` string?

Example 2: change `hello` to `jello`

- Using string method called `replace()` .

In [ ]:
```
# method 1
s = 'hello'
print(s, id(s))            # print s, and its id
t = None    # replace h with j
print(t, id(t))            # print s and its id
print(s, id(s))
```

Example 3: change `hello` to `jello`

- Using `+` concatenation and slicing

In [ ]:
```
# method 2
```

```
s = 'hello'
print(s, id(s))          # print s, and its id
s = None                 # j + slicing
print(s, id(s))          # print s, and its id
```

String methods

Strings in Python are objects of `str` class, so they have many methods. Only some of the commonly used string methods are shown below. Since strings are immutable, none of these methods mutates string s. Methods `count()` and `find()` return an integer, method `split()` returns a list, and the remaining methods return a (usually) modified copy of string `s`.

> `s.capitalize()` A copy of string s with the first character capitalized if it is a letter in the alphabet
> `s.count(target)` The number of occurences of substring `target` in string s
> `s.find(target)` The index of the first occurrence of substring `target` in string s.
> `s.lower()` A copy of string s converted to lowercase
> `s.upper()` A copy of string s converted to uppercase
> `s.replace(old, new)` A copy of string s in which every occurrence of substing `old`, when string s is scanned from left to right, is replaced by substring `new`.
> `s.translate(table)` A copy of string s in which characters have been replaced using the mapping described by `table`
> `s.split(sep)` A list of substrings of strings s, obtained using delimiter string `sep`; the default delimiter is the blank space
> `s.strip()` A copy of string s with leading and trailing blank spaces removed

`count()` method

This method counts the number of occurrences of one string within another string. The simplest form is this one: `s.count(substring)`. Only non-overlapping occurrences are taken into account:

Example 1:

1. Count `a` in a string `abraham`.
2. Count `by` in a string `Walk by faith, not by sight`

In [ ]:
```
# count 'a' in a string
s = 'abraham'
print(None)
```

In [ ]:
```
# count 'by' in a string
s = 'Walk by faith, not by sight'
print(None)
```

`find()` and `rfind()` methods

A method is a function that is bound to the object. When the method is called, the method is applied to the object and does some computations related to it. Methods are invoked as `object_name.method_name(arguments)`. For example, in `s.find("e")` the string method `find()` is applied to the string `s` with one argument `"e"`.

The `find()` method searches a substring, passed as an argument, inside the string on which it's called. The function returns the index of the first occurrence of the substring. If the substring is not found, the method returns `-1`.

The `rfind()` method returns the highest index of the substring (if found). If not found, it returns -1.

Example 2:

Find indices of `'i'`, `'Deo'` and `'F'` in `s = 'Soli Deo Gloria'` which are 3, 5 and -1, respectively.

In [ ]:
```
s = 'Soli Deo Gloria'
# find indice of 'i', 'Deo' and 'F'
print(None)
print(None)
```

If you call `find()` with three arguments `s.find(substring, left, right)`, the search is performed inside the slice `s[left:right]`. If you specify only two arguments, like `s.find(substring, left)`, the search is performed in the slice `s[left:]`, that is, starting with the character at index `left` to the end of the string. Method `s.find(substring, left, right)` returns the absolute index, relatively to the whole string `s`, and not to the slice.

Example 3:

- Find three indices of the first, last and second occurrences of `'by'` in `s = 'Walk by faith, not by sight. bye now!'`.
- Use two indices and the phrase "faith, not by sight" between them.

In [1]:
```
# find the indices of the first, last, and second occurrences of 'by'
s = 'Walk by faith, not by sight. bye now!'
first = None
last = None
second = None

# slice out "faith, not by sight" and print
print(None)
```

```
None
```

`replace()` method

Strings have a nice convenience method for replacing text in the string. The method `replace()` replaces all occurrences of a given substring with another one.

Syntax: `s.replace(old, new)` takes the string `S` and replaces all occurrences of substring old with the substring new.

Example 4:

Replace the `"Fide"` in `s = 'Sola Fide'` with `"Gratia"` :

In [ ]:
```
s = "Sola Fide Fide Fide"
t = None
print(t)
```

One can pass the third argument count, like this: `s.replace(old, new, count)` . It makes `replace()` to replace only first `count` occurrences and then stop. Without `count` , by default, it replaces all occurrences.

In [ ]:
```
s = 'Sola Fide Fide Fide'
#t = s.replace("Fide", "Gratia", 2)
s.replace(None)
print(t)
```

In [ ]:
```
help(str)
```

Lecture Note: What is `self` argument in method definition?

```
replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old replaced by
new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count
occurrences are replaced.
```

dir() method

There are many other methods on strings, and Python's `dir()` function will list all the methods on an object:

In [ ]:
```
dir(str)
```

IPython makes it even easier: if you have a string `s` you can just type `s.` and then hit the "tab" key and get a list of all the methods.

In [ ]:
```
s = "Scriptura"
s
s.upper()            # s<tab> to see the list available - try it yourself
```

Looking through the methods you will see the ones we just talked about, but also things like `lstrip()` and `rstrip()` which strip from the left and right respectively, and a whole host of others.

strip() method

The `strip()` method removes excess characters from the ends of a string, which is useful as often in the real world strings come with extra garbage characters at the front and back that you

don't care about. It is the same as `trim()` in Java.

By defualt, `strip()` removes whitespace:

In [ ]:
```
s = "Wt  Gratia  Wn"
s.strip()
```

split() method

You can split a string wherever it has white space (a sequence of spaces, tabs or newlines) using the `split()` method:

Example 7:

Split the following phrase into a list of words.

In [ ]:
```
s = "Walk by faith, not by sight"
word_list = s.split()
print(word_list)
```

The result is a list, which we'll learn about later, containing the strings `'Walk' 'by' 'faith,' 'not' 'by'` and `'sight'`.

Example 8:

Print each word per line in `s = "Walk by faith, not by sight"`.

Expected Output:

```
Walk
by
faith,
not
by
sight
```

In [1]:
```
s = 'Walk by faith, not by sight'
for w in s.split():
    print(w)
```

```
Walk
by
faith,
not
by
sight
```

join() method

You can put the words back together by starting with a single space and then using its `join()` method with the word list to join the strings together with a space between each item:

```
join(self, iterable)
    Concatenate any number of strings.
```

```
    The string whose method is called is inserted in between each
given string.
    The result is returned as a new string.
```

Example:

```
', '.join([1, 2, 3]) -> '1, 2, 3'
```

Example 9:

Review the following example and recover the original phrase: `'Walk by faith, not by sight'` .

Expected Output:

```
Walk by faith, not by sight
```

In [ ]:
```
word_list = ['Walk', 'by', 'faith,', 'not', 'by', 'sight']
t = None
print(t)
```

upper() & lower() methods

The `upper()` or `lower()` method converts all of the string's characters to upper case or lower case, respectively:

Expected Output:

```
SCRIPTURE ALONE
scripture alone
```

In [2]:
```
s = "Scripture Alone"
print(None)
print(None)
```

```
SCRIPTURE ALONE
scripture alone
```

string.format() basics

Here are a couple of example of basic string substitution, **the {} is the placeholder** for the substituted variables. If no format is specified, it will insert and format as a string.

```
(1)   '{0}:{1}:{2}'.format( hour, min, sec )
      11:45:33

(2)   '{2}:{0}:{1}'.format( hour, min, sec )
      33:11:45

(3)   '{ }:{ }:{ }'.format( hour, min, sec )
      11:45:33
```

Figure 1. String Method f or $mat()$

As you see the statement (1) shown above, the string invoking the format() function - that is, the string `'{0}:{1}:{2}'` - is the **format string**: It describes the output format. All the characters outside the curly braces - that is, the two colons - are going to printed as is. The curly braces {0}, {1}, and {2} are **placeholders** where the objects will be printed. The numbers 0, 1, and 2 explicitly indicate that the placeholders are for the first, second, and third arguments of the `format()` function call, respectively.

As you see the statement (2), it shows what happens when we move the indexes 0, 1, and 2 in the previous example.

As you see the statement (3), when no explict number is given inside the curly braces, the default is to assign the first placeholder to the first argument of the `format()` function, the second placeholder to the second argument, and so on.

In [ ]:
```
s1 = "God is good {}".format("all the time")
s2 = "{} and {} that God is good.".format("Taste", "see")

print(s1)
print(s2)
```

In [ ]:
```
a = 'Soli'
b = 'Deo'
c = 'Gloria'
# Basic formatting
print('{} {} {}'.format(a, b, c))
```

Example:

Replace the `None` with your code such that it produces the following line:

```
`Gloria!, Soli Soli Deo Gloria!`
```

In [ ]:
```
# Numbered fields refer to the position of the arguments.
a = 'Soli'
b = 'Deo'
c = 'Gloria'
print("{2}!, {0} {0} {1} {2}".format(a, b, c))
```

Lining up data in columns

To illustrate the issues, let us consider the problem of properly lining up values of functions i^2, i^3 and 2^i for $i = 1, 2, 3, \ldots$ Lining up the values properly is useful because it illustrates the very different growth rates of these functions:

In [9]:
```
print('i       i**2       i**3       2**i')
for i in range (1, 13):
    print(i, i**2, i**3, 2**i, sep = '          ')
```

```
i       i**2       i**3       2**i
1          1          1          2
2          4          8          4
3          9          27          8
4          16          64          16
```

```
5          25          125          32
6          36          216          64
7          49          343          128
8          64          512          256
9          81          729          512
10          100          1000          1024
11          121          1331          2048
12          144          1728          4096
```

While the first few rows look OK, we can see that the entries in the same columns are not properly lined up. To fix this problem, what we need is a way to fix the width of each column of numbers and print vlaues **right-justified** within these fixed-width columns.

Inside the curly braces of a format string, we can specify how the value mapped to the curly brace placeholder should be presented; we can specify its **field width, alignment, decimal precision, type,** and so on.

For example,

In [20]:
```
print('1st case: {0:5} {1:10}'.format(12, 345))
print('2nd case: {:5} {:10}'.format(12, 345))
```

```
1st case:    12        345
2nd case:    12        345
```

As you observe two cases above, the statements are identical. The first number argument of the placeholder must be the order of argument of `format()` function. The 0 in `0:5` refers to 12 in `.format(12, 345)`. The second argument 5 in `{0:5}` indicates the width of the placeholders for 12 in `format(12, 345)`. The second argument 10 in `{1:10}` indicates the width of the placeholders for 345 in `format(12, 345)`. Some extra spaces are added in front.

The precision is a decimal number that specifies how many digits should be displayed before and after the decimal point of a floating-point value. It follows the field width and a period separates them.

In the following example, the field width is 10 but only five digits of floating value are displayed in the first case, while the default displays a lot more:

In [24]:
```
print('1st case: {:10.5}'.format(1000/3))
print('2nd case:', 1000/3)
```

```
1st case:     333.33
2nd case: 333.3333333333333
```

Integer presentation types:

> b outputs the number in binary
> c outputs the Unicode character corresponding to the integer value
> d outputs the number of decimal notation (defualt)
> o outputs the number in base 8
> x outputs the number in base 16, using lowercase
> X outputs the number in base 16, using uppercase

Floating point number presentation types:

> f outputs the number in a fixed-point number (i.e., with a decimal point and fractional part) e outputs the number in scientific notation in which the exponent is shown after the character e

In [31]:
```
n = 10
print('{:b}'.format(n))
print('{:c}'.format(n))   # new line
print('{:d}'.format(n))
print('{:o}'.format(n))
print('{:x}'.format(n))
print('{:X}'.format(n))

print('{:10.2f}'.format(5/3))
print('{:e}'.format(5/3))
```

```
1010


10
12
a
A
      1.67
1.666667e+00
```

Now let us go back to our original problem of presenting the values of functions i^2, i^3 and 2^i for $i = 1, 2, 3, \ldots$ Lining up the values properly is useful because it illustrates the very different growth rates of these functions:

In [42]:
```
def growth_rates(n):
    print(' i     i**2     i**3     2**i')
    format_str = '{:2d} {:8d} {:8d} {:8d}'
    for i in range (1, n):
        print(format_str.format(i, i**2, i**3, 2**i))

growth_rates(15)
```

```
 i     i**2     i**3     2**i
 1        1        1        2
 2        4        8        4
 3        9       27        8
 4       16       64       16
 5       25      125       32
 6       36      216       64
 7       49      343      128
 8       64      512      256
 9       81      729      512
10      100     1000     1024
11      121     1331     2048
12      144     1728     4096
13      169     2197     8192
14      196     2744    16384
```

In [47]:
```
def growth_rates(n):
    print('{:>2} {:>8} {:>8} {:>8}'.format('i', 'i**2', 'i**3','2**i'))
    format_str = '{:2d} {:8d} {:8d} {:8d}'
    for i in range (1, n):
        print(format_str.format(i, i**2, i**3, 2**i))

growth_rates(15)
```

```
 i      i**2      i**3      2**i
 1         1         1         2
 2         4         8         4
 3         9        27         8
 4        16        64        16
 5        25       125        32
 6        36       216        64
 7        49       343       128
 8        64       512       256
 9        81       729       512
10       100      1000      1024
11       121      1331      2048
12       144      1728      4096
13       169      2197      8192
14       196      2744     16384
```

Exercises

Using string methods

Assuming that variable `forcast` has been assigned string

```
It will be a sunny day today
```

Write Python statements corresponding to these assignments:

1. To variable `count`, the number of occurrences of string `day` in string `forcast`.
2. To variable `weather`, the index where substing `sunny` begins.
3. To variable `change`, a copy of `forcast` in which every occurrence of substing `sunny` is replaced by `cloudy`.

In [ ]:

```
```

Using string.format()

In the following code, change the first line, `print()` function, such that it also uses the right-justified format of the string. Replace `title_str = None` with your format string answer.

```
def growth_rates(n):
    print(' i     i**2     i**3     2**i')
    format_str = '{:2d} {:8d} {:8d} {:8d}'
    for i in range (1, n):
        print(format_str.format(i, i**2, i**3, 2**i))

growth_rates(15)
```

Expected Output:

```
 i     i**2     i**3     2**i
 1        1        1        2
 2        4        8        4
```

```
 3        9       27        8
 4       16       64       16
 5       25      125       32
 6       36      216       64
 7       49      343      128
 8       64      512      256
 9       81      729      512
10      100     1000     1024
11      121     1331     2048
12      144     1728     4096
13      169     2197     8192
14      196     2744    16384
```

Solution:

In [49]:

```
def growth_rates(n):
    #title_str = None
    #print(title_str.format('i', 'i**2', 'i**3','2**i'))

    format_str = '{:2d} {:8d} {:8d} {:8d}'
    for i in range (1, n):
        print(format_str.format(i, i**2, i**3, 2**i))

growth_rates(15)
```

```
1        1        1        2
2        4        8        4
3        9       27        8
4       16       64       16
5       25      125       32
6       36      216       64
7       49      343      128
8       64      512      256
9       81      729      512
10      100     1000     1024
11      121     1331     2048
12      144     1728     4096
13      169     2197     8192
14      196     2744    16384
```

def growth_rates(n): print('{:>2} {:>8} {:>8} {:>8}'.format('i', 'i**2', 'i**3','2**i')) **format_str = '{:2d} {:8d} {:8d} {:8d}' for i in range (1, n): print(format_str.format(i, i**2, i**3, 2**i))**

growth_rates(15)

`swap_two()` : To swap the two words

Given a string consisting of **exactly two words separated by a space**. Print a new string with the first and second word positions swapped (the second word is printed first). This task should **not use loops**, but use other built-in functions.

Hint: Python funtion can have multiple assignments and returns as shown below. It produces `4, 2` :

```
def swap(x, y):
    return y, x
```

```
a, b = 2, 4
a, b = swap(a, b)
print(a, b)
```

Solution:

In [ ]:
```
def swap_two(str):
    return None
```

Expected Output:

```
Hello, World! ---> World! Hello,
car race ---> race car
Sola Gratia ---> Gratia Sola
Han Dong ---> Dong Han
H D ---> D H
```

In [ ]:
```
list = ['Hello, World!', 'car race', 'Sola Gratia', 'Han Dong', 'H D']

for s in list:
    one, two = swap_two(s)
    print(s, '--->', one, two)
```

swap_cases

Write the function to swap cases of a given string.

Hint: You may use some of the following functions: `isupper()`, `islower()`, `lower()`, `upper()`

Sample run:

```
print(swap_cases("Blessing of HANDONG"))
print(swap_cases("Hello World!"))
print(swap_cases("NumPy"))

bLESSING OF handong
hELLO wORLD!
nUMpY
```

Solution:

In [ ]:
```
def swap_cases(st)
    return None
```

빈도 수에 따라 정렬하기

단어에 나오는 알파벳의 빈돗수에 따라 정렬하십시오. sorted(), join(), count() 등과 같은 내장함수나 메소드를 이용하면 도움이 됩니다. 물론, 람다(익명)함수도 필요할 것입니다.

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in
ascending order.

    A custom key function can be supplied to customize the sort order,
and the
    reverse flag can be set to request the result in descending order.
```

Sample Run:

```
msg = 'handong-global'
...
-bdhaaggllnnoo

msg = 'tennessee'
...
tnnsseeee

msg = 'tallahassee'
...
hteellssaaa
```

In [1]:
```
msg = 'tennessee'
# msg = 'mississippi'
# msg = 'tallahassee'
# msg = 'handong-global'

None

print(final_msg)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~₩AppData₩Local₩Temp/ipykernel_26452/3379998399.py in <module>
      6 None
      7
----> 8 print(final_msg)

NameError: name 'final_msg' is not defined
```

count_hi() : Count the number of occurences

The function, `count_hi()` returns the number of times that the string "hi" appears anywhere in the given string. **Do not use** the str's methods such as `count()` .

Hint: Use a for-loop and slicing properly.

Sample Run:

```
count_hi('abc hi ho') ---> 1
count_hi('ABChi hi') ---> 2
count_hi('hihi') ---> 2
count_hi('hohihi hih') ---> 3
```

Solution:

In [ ]:
```
def count_hi(str):
    return None
```

Expected Output:

```
hihi ---> 2
hike ---> 1
 ---> 0
h ---> 0
hi ---> 1
Aim high ---> 1
hi, hike ---> 2
highlight hi ---> 2
hohihi hih ---> 3
```

In [ ]:
```
# 테스트를 위한 아래 코드를 수정하지 말고 사용하십시오. Don't modifty the following co

list = ['hihi', 'hike', '', 'h', 'hi', 'Aim high', 'hi, hike', 'highlight hi', 'hohihi
for s in list:
    count = count_hi(s)
    print(s, '--->', count)
```

`remove_between()` : A Challenge Problem

Given a string in which a letter **x** occurs at least twice. Remove from that string the first and the last occurrence of the letter **x**, as well as all the characters between them. The **x** can be any letters that you may type from a keyboard.

Hint: If you use `find()`, `rfind()` and slicing properly, you may code this function in even two or three lines.

Solution:

In [ ]:
```
def remove_between(str, ch):
    return None
```

Expected Output:

Case 1. with `l`

```
hello ---> heo
long life ---> ife
living, life, love ---> ove
Be joyful, joyful ---> Be joyfu
```

Case 2: with `g`

```
gg --->
gag --->
```

```
garage ---> e
garbage in garbage out, ---> e out,
```

Case 3: with y

```
yoyo ---> o
bye bye ---> be
Walk by faith, not by sight ---> Walk b sight
```

In [ ]:
```
# 테스트를 위한 아래 코드를 수정하지 말고 사용하십시오. Don't modifty the following co

list = ['hello', 'long life', 'living, life, love',  'Be joyful, joyful']
for s in list:
    answer = remove_between(s, 'l')
    print(s, '--->', answer)
```

In [ ]:
```
# 테스트를 위한 아래 코드를 수정하지 말고 사용하십시오. Don't modifty the following co

list = ['gg', 'gag', 'garage', 'garbage in garbage out,']
for s in list:
    answer = remove_between(s, 'g')
    print(s, '--->', answer)
```

In [ ]:
```
# 테스트를 위한 아래 코드를 수정하지 말고 사용하십시오. Don't modifty the following co

list = ['yoyo', 'bye bye', 'Walk by faith, not by sight']
for s in list:
    answer = remove_between(s, 'y')
    print(s, '--->', answer)
```

학습 정리

1. 슬라이싱 문법
2. 다양한 문자열 메소드
3. 문자열 포멧
4. 데이터 보관 방법

참고자료

- Python Strings
- Python Strings-Formatting

"Everything is permissible" - but not everything is beneficial. "Everything is permissible" - but not everything is constructive. 1 Corinthians 10:23