

2022

01 26

수요일

〈Python Operator〉

- 논리비트 연산자

: Bool 데이터를 가지고 연산을 해서 결과를 bool로 리턴하는 연산자
python 은

True =1

False = 0 으로 간주해서 산술 연산에 사용 가능

혹은

0 이 아닌 숫자를 True

0 인 숫자를 False 로 간주해서 논리 연산에 사용 가능

- 데이터가 존재하는 Vector 자료형도 True 로 간주
데이터가 존재하지 않는 Vector 자료형은 False 로 간주

데이터 분석 Library 에서는 ...

논리 비트 연산자 (and , or , not) 을 사용하면 전체를 하나의

True , False 로 간주하고 연산을 수행해서

bool 로 값을 return

산술 비트 연산자 (& , | , ^) 를 사용하면

각 요소 단위로 연산을 해서 Vector 자료형을 Return

= 일반 응용 프로그램이나

Web Programming을 할 때는

논리 비트연산자가

데이터 분석을 할 때는

산술 비트 연산자가

조금 더 중요하다

- 1. and : 둘 다 True 인 경우만 True , 그 이외는 False

- 2. or : 둘 다 False 인 경우만 False , 나머지는 True

- 3. not : True 이면 False , False 이면 True

: and 와 or 는 피연산자의 위치를 변경해서

결과는 변화가 없지만

연산의 횟수가 달라 질 수 있다

: and 는 앞의 데이터가 False 이면
뒤의 데이터를 확인하지 않으며

or 는 앞의 데이터가 True 이면
뒤의 데이터를 확인하지 않음

```
print(True and False)
→ False
```

```
# 정수 데이터도 논리연산이 가능하다
print(1 and 1)
→ 1
```

<복합 할당 연산자>

: 다른 연산자와 할당 연산자가 같이 사용되는 것을 말함

변수 연산자 = 데이터(변수, Literal, 표현식 - 연산식 이나 함수 호출 구문)

변수가 가리키는 데이터의
오른쪽 데이터를 연산자를 이용해서 연산을 수행
왼쪽의 변수가 가리키도록 한다

```
# 복합 할당 연산자
x = 10

x += 20

print(x)

→ 30
```

<자료형 확인과 참조하고 있는 데이터의 위치 확인>

case 1. type(데이터)

: 데이터의 자료형을 문자열로 리턴 —> 주기적으로 사용
(어떤 자료형으로 결과를 주는지 확인해봐야한다)

case 2. id(데이터)

: 데이터가 참조하고 있는

데이터의 id(메모리 영역을 구분하기 위한 구분자) return

```
# <자료형 확인>
```

```
x = "Hello"
```

```
print(type(x))
```

```
# <class 'str'> : 문자열
```

```
y = (10, 30)
```

```
print(type(y))
```

```
# <class 'tuple'> : 튜플형식
```

```
# < 데이터의 저장 구분자 확인 >
```

```
print(id(x))
```

```
# 140533613072944
```

< 자료형 변환 >

Case

정수 변환

`int`(숫자 데이터 또는 정수 변환이 가능한 문자열, `bool` 데이터)
: 실수의 경우 소수는 버림 한다.

실수 변환

`float`(숫자 데이터 또는 숫자로 변환이 가능한 문자열, `bool` 데이터)

bool 변환

`bool`(숫자 데이터 나 `bool`로 변환이 가능한 문자열)

문자열 변환

`str`(데이터)

< 콘솔 입출력 >

Console

: windows에서는 Command window
Linux/Mac에서는 Terminal window
→ IDE에서는 Console로 가는 것을 가로채서 자신의 콘솔을 사용

Console 출력

`print` 함수 이용

: 출력하고자 하는 데이터를 `print` 함수의 parameter로 나열
출력이 끝나면 다음 줄로 넘어간다.

```
help(print)
```

```
→ result
```

```
print(...)  
  print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)  
  
Prints the values to a stream, or to sys.stdout by default.  
Optional keyword arguments:  
file: a file-like object (stream); defaults to the current sys.stdout.  
sep: string inserted between values, default a space.  
end: string appended after the last value, default a newline.  
flush: whether to forcibly flush the stream.
```

sep 라는 parameter에 값을 설정하면
여러개의 데이터를 출력할 때 각 데이터의 구분자가 된다.

end 라는 parameter 값을 설정
출력을 하고 난 후 줄바꿈 대신 설정한 값이 출력

```
print("Hello")
# 출력을 한 후 줄 바꿈
print("hi")

# 여러 개의 데이터 출력

print("Hi", "-mickey")

# 여러 개의 데이터 사이에 구분자를 설정
print("hi", "my Name is", "Angel", sep='-')

# 출력한 후 줄바꿈 대신 다른 내용을 출력
print('python is', end='--')
print("is an amazing Program Language");
```

— Result

Hello

hi

Hi -mickey

hi-my Name is-Angel

python is--is an amazing Program Language

<서식 설정>

: '-' 이 와 동일한 형태를 이용해서
서식을 적용한 문자열도 생성 가능

- 서식을 설정할 때는 % 를 사용
- 문자열 리터럴 안에 서식을 설정하고
이어서 %(데이터 나열) 을 추가하면
서식에 데이터가 순서대로 하나씩 매칭되서 출력

%s : 문자열

%c : 문자 1 개

%d : 정수

%f : 실수

%o : 8진수

%x : 16진수

% % : % 출력

— % 다음에 자릿수를 기재해서 전체 자릿수를 확보한 후 출력 가능

— 실수의 경우 전체자릿수 . 소수자릿수
or
. 소수 자릿수를
이용해서 소수 자릿수를 설정하는 것이 가능

```
print("파이는 %.2f"%(3.141592))  
# 파이는 3.14
```

< '{ }'.format >

: % formatting 과 결과는 동일
python 에서는 이 방법을 권장

'문자열 {index 설정 가능}'.format(데이터 나열)
- index 가 설정되어 있으면 index 번째 데이터로 치환
index 가 없으면 순서대로 치환

```
print('{0} is a {1} '.format('mickey', 'mouse'))  
# mickey is a mouse
```

< f-strings >

: 문자열 리터럴 안에 f"{데이터이름} ..."의 형태로 생성하는 것이 가능

```
first = "Hamburger"  
second = "food"  
  
print(f"{first} is a {second}")
```

< Console 입력 >

-> input('parameter')

: 문자열을 입력 받아서 Return 하는 함수
Enter 누를 때 까지 입력 받음

< Control Statement >

Expression(표현식)

한 번에 수행되는 문장으로 변수에 값이나 연산식
또는 함수의 실행 결과를 저장하거나 함수를 호출하는 문장

- python 에서 하나의 코드 Block 을 만드는 방법
: 블럭 내 코드는 일정한 들여쓰기 간격을 가져야 한다 .

- Block 은 제어문 , 함수 , 클래스 등이 생성 가능
하나의 파일 내에 있는 코드는
전체를 하나의 블럭으로 간주(Module)

제어문 (Control Statement)

- 기본적인 프로그램의 흐름은 순차적
(위에서 아래로 순서대로)
이러한 흐름을 변경하는 명령어

(if , for , while → 3. 0 version 부터 switch 가 제공)

(if)

1. 단순 if

— if 리턴이 있는 표현식
표현식의 값이 True 일 때 수행할 내용

```
# if
var = True
if var:
    print("bool 데이터를 가지고 수행 ")

var = 11
if var:
    print("숫자를 사용해서 수행 ") # 0 이 아닌 숫자는 True

var = [10, 20, 3]
if var:
    print("데이터 목록을 가지고 수행") # 데이터가 존재하면 True

var = []
if var:
    print("데이터 목록을 가지고 수행")
```

Result

bool 데이터를 가지고 수행
숫자를 사용해서 수행
데이터 목록을 가지고 수행

2. if -else

if 리턴이 있는 표현식:

표현식의 값이 True 일 때 수행할 내용

else:

표현식의 값이 False 일 때 수행할 내용

```
time = random.randint(1, 20)
if time > 10:
    print(time)
    print("A")
else:
    print(time)
    print("B")
```

RESULT_____

18

A

Python 은 변수에 값을 할당하는 부분에 제어문 사용가능

variable = 기본값 if 표현식 else 표현식이 거짓을 때 대입할 값

```
# 조건문을 이용한 변수의 값 할당
result = "success"
score = random.randint(1, 100)

if score >= 51:
    print(score)
    print(result)
else:
    print(score)
    result = "Try again"
    print(result)

result = "success" if score >= 51 else "Try Again"
print(score)
print(result)
```

result _____

72

success

72

success

3. if - elif -else

if 리턴이 있는 표현식 1 :

표현식의 값이 True 일 때 수행할 내용

elif 리턴이 있는 표현식 2 :

표현식1의 값이 False 이고 표현식 2의 값이 True 일때 수행 할. 내용

else:

표현식의 값이 False 일 때 수행할 내용

— elif 는 여러개 작성이 가능하고
else 는 생략이 가능

```
# 하나의 숫자를 입력 받아서
score = random.randint(1,100)

# 90-100 --> A
# 70-90 --> B
# 50-70 --> C
# 나머지 ----> D

if 90 <= score:
    print(score)
    print("A")
elif 70 <= score < 90:
    print(score)
    print("B")
elif 50 <= score < 70:
    print(score)
    print("C")
else:
    print(score)
    print("D")
```

RESULT

98

A

반복문

while

while 표현식:

표현식이 True 일 때 수행할 내용

```
#while 1 부터 10 까지 출력
i = 1
while i <= 10:
    print(i)
    i = i+1
```

RESULT ———

1
2
3
4
5
6
7
8
9
10

컴퓨터 프로그램으로 어떤 문제 해결
——> computational Thinking

- 분해 : 복잡한 문제를 작은 문제로 나눔
- 패턴 인식
- 추상화 : 문제의 핵심에 집중하고 부가적인 것은 제외
- 알고리즘 : 이렇게 정의한 문제를 해결

```
# https://www.donga.com/news/search?p=데이터의 시작번호&query= 검색어

# 1 page = 1 번에서 시작
# 2 page = 16 번에서 시작
# 3 page = 31 번에서 시작
# 검색어를 입력받아서 1 page 부터 10 page 까지 크롤링 할 수 있는 URL 출력
query = input("insert what you want ");
idx = 1
while idx < 11:
    print(f"https://www.donga.com/news/search?p={1+15*(idx-1)}"+f"&query={query}")
    idx += 1
```

```
insert what you want ice
https://www.donga.com/news/search?p=1&query=ice
https://www.donga.com/news/search?p=16&query=ice
https://www.donga.com/news/search?p=31&query=ice
https://www.donga.com/news/search?p=46&query=ice
https://www.donga.com/news/search?p=61&query=ice
https://www.donga.com/news/search?p=76&query=ice
https://www.donga.com/news/search?p=91&query=ice
https://www.donga.com/news/search?p=106&query=ice
https://www.donga.com/news/search?p=121&query=ice
https://www.donga.com/news/search?p=136&query=ice
```

— 무한 반복

while True:
반복할 내용

```
i = 0
while i < 5:
    print(i)
    i = i + 1

else:
    print("반복문 모두 실행 후 종료")
```

RESULT

0

1

2

3

4

반복문 모두 실행 후 종료

```
i = 0
while i < 5:
    print(i)
    i = i + 1
    if i > 3:
        break
# while 다음의 else 는 반복문이 전부 수행된 경우
# --> 중간에 종료되지 않는 경우에만 수행
else:
    print("반복문 모두 실행 후 종료")
```

RESULT _____

0

1

2

3

For

for 임시변수 in 순서열(순회 가능한 데이터의 모임):
 순서열의 데이터를 임시변수에 순서대로 하나씩 대입 하면서 수행 될 문장

—— 순서열의 데이터는 `__iter__` 가 구현된 자료형의 데이터만 가능

```
s = "Test String"

for ch in s:
    print(ch)

pl = ["java", "python", "Sql"]
print((dir(pl)))
for language in pl:
    print(language)

dic = {"1": "one", "2": "two", "3": "three"}
print(dir(dic))
for imsi in dic:
    print(imsi)
```

——> dict 는 key를 순회한다

Result

```
T
e
s
t

S
t
r
i
n
g
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
java
python
Sql
['__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__ior__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__',
__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__ror__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
1
2
3
```

range

: 일정한 간격을 갖는 순서열을 만드는데 사용하는 함수

range(시작 값, 종료 값, 간격) 의 형태로 생성

- 종료 값은 생략 불가능
- 값을 하나만 설정하면 종료값으로 설정, 시작값 0, 간격 1 로 설정
- 값을 2개 설정하면 시작값과 종료값이 되고 간격은 1.
- 종료 값은 포함되지 않음

```
#Range
print(type(range(0, 0, 1)))
print(dir(range))
```

```
<class 'range'>
['__bool__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', 'count', 'index', 'start', 'step', 'stop']
```

```
for idx in range(0, 10, 2):
    print(idx)
```

0
2
4
6
8

< 제어문 중첩 >

제어문은 중첩해서 사용하는 것이 가능

시작은 바깥쪽 제어문에서
안쪽 제어문을 전부 수행하고 바깥쪽 제어문의 다음 제어를 수행

제어문 중첩

```
i = 0
j = 0
k = "*"
for i in range(1, 6, 1):# 5줄
    for j in range(1, i+1, 1): # 각줄은 1 번부터 줄 번호까지
        print('*',end='') # * 을 출력하고 줄바꿈을 하지 않도록
    print()
```

```
*
**
***
****
*****
```

```
for i in range(1, 6, 1):
    if i <= 3:
        for j in range(1, i + 1, 1):
            print("*", end='')
    else:
        for j in range(1, 7-i, 1):
            print('*', end='')
    print()
```

```
for i in range(1, 6, 1):
    j = i+1 if i <= 3 else 7 - i
    for j in range(1, j, 1):
        print('*', end='')
    print()
```

```
*
**
***
**
*
```

```
*
**
***
**
*
```


< Continue 와 Break >

: 반복문에서만 사용 가능

- continue 는 다음 반복문으로 이동

- break 는 반복문을 중단

: for - break - else

while - break - else

for 나 while 다음에 else 가 있으면

for 나 while 이 break 를 만나지 않으면

else 블록을 생성

```
# Continue 와 Break

# 3 까지 출력하고 멈춤
for i in range(1, 10):
    if i % 3 == 0:
        print(i)
        break
print()

# 3의 배수를 건너뛰고 출력
for i in range(1, 10):
    if i % 3 == 0:
        print(i)
        continue
print()
```

3

3
6
9

```
# 피보나치수열
# 1, 1, 2, 3, 5, 8, 13, 21 ,34, 55 ,89
# 첫번째와 두번째는 1
# 세번째 부터 앞의 2개의 항의 합
# 15항의 피보나치 수열의값을 재귀를 쓰지않고 출력
```

```
n1 = 1
n2 = 1
fibonacci = 1
for i in range(3, 16, 1):
    fibonacci = n1 + n2
    # n2 = n1
    # n1 = fibonacci
    # 위처럼 해도 되지만 파이썬은 동시에 값을 넣는 것도 가능함
    n2, n1 = n1, fibonacci
    # 위와 같이 파이썬은 여러개의 변수에 여러값을 한꺼번에 대입하는 것이 가능
print(fibonacci)
```

610

< Function >

함수 자체의 의미

: 한번에 수행되어야 하는 코드의 블록을

하나의 이름으로 묶어서 이름만으로 수행하기 위해 만든것을 말함

Method는 호출하는 대상 (Class 나 Instance) 이 있는 경우에만 사용이 가능하다.

— 함수를 사용하는 경우 장점 —

1. 모듈화를 통해서 코드의 가독성을 높일 수 있음
: 즉 읽기 쉽게 만들 수 있다.
2. 코드의 중복을 제거
: 유지 보수하기 쉬워진다.
3. 3rd Party Function
: 다른 개발자가 만든 함수를 사용할 수 있다.

— 함수의 종류 —

1. Maker Function : 프로그래밍 언어에서 제공
2. User Define Function : 개발자가 만든 함수

— 함수는 일급 객체 —

함수도 하나의 자료형이다.

: 이 말은 함수를 변수에 대입할 수 있고

Parameter로 사용할 수 있고

return 타입으로도 사용이 가능하다.

함수의 이름은 함수에 대한 참조가 되고

다른 함수 호출은 함수이름() 으로 작성 해야한다.

— 내장 함수 —

파이썬이 제공하는 함수(Maker Function , Builtin Function)

확인 : `dir(__builtins__)`

```
print(dir(__builtins__))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',  
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',  
'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError',  
'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',  
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',  
'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',  
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError',  
'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',  
'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',  
'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning',  
'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',  
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',  
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__',  
'__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool',  
'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',  
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',  
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',  
'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',  
'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',  
'type', 'vars', 'zip']
```

```
# 최대값을 구해주는 함수  
print(max([100, 200, 3000]))
```

3000

```
print(help(max))
```

리턴 되는 값

max(...)

max(iterable, *[, default=obj, key=func]) -> value

max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The default keyword-only argument specifies an object to return if the provided iterable is empty.

With two or more arguments, return the largest argument.

None

< 사용자 정의 함수 >

정의

```
def 함수이름( [Parameter 나열] ) :  
    함수의 내용  
    [return 데이터]
```

— Parameter 와 Return 도 생략이 가능

[함수 호출]

함수 이름 (parameter에 대입할 데이터)

: 함수를 정의 하면 static 영역에 함수가 저장

함수를 호출하면 함수의 정의

구문을 가지고 별도의 메모리 영역을 생성 후 작업 수행

더 이상 수행할 문장이 없으면 호출한 곳으로 제어권이 다시 넘어옴

——> 함수가 실행되기 위해서 할당 받았던 메모리 영역은 소멸

[Return]

return 은 함수의 수행을 종료하고

호출한 곳으로 돌아가는 명령어

return 뒤에 데이터를 기재할 수 있는데

데이터를 기재하면 호출한 곳으로 돌아갈 때

이 데이터를 가지고 가서 함수의 수행 결과가 된다

기본적으로 return 은 1개만 가능

python에서는 . 를 사용해서 여러개의 데이터를 리턴 할 수 있다.

(실제로는 하나의 Tuple 로 묶어서 Return)

여러개의 데이터를 리턴하고자 할때는

별도의 클래스를 만들어서 Return 하거나

vector 자료형으로 묶어서 리턴

함수가 return 을 하면

그 데이터를 가지고 계속 작업을 수행하는 것이 가능

```
# 함수의 정의와 호출  
def func():  
    ment = 'No Parameter No Return Value'  
    print(ment)
```

```
func()  
func()
```

None

No Parameter No Return Value

No Parameter No Return Value

[Argument(Parameter , 매개변수 또는 인자 , 인수)]

함수를 호출할 때

호출하는 곳에서 함수에게 넘겨주는 데이터

함수에게 데이터를 넘길 때

scala 자료형

→ 직접 접근하는 데이터 타입 : 값이 전달되는 형태

scala 데이터 넘김 : 원본데이터 수정 x

vector 자료형

→ 간접 접근하는 형태 : 참조가 전달되는 형태

vector 데이터를 넘김 : 원본 데이터 수정이 가능

```
def callByValue(n):  
  n = n+1  
  print(f"함수 내부 : {n}")  
  
def callByReference(ar):  
  ar[0] = ar[0] + 1  
  print(f"함수 내부 : {ar}")
```

```
x = 100
```

— Scala 데이터를 넘겼음으로 x의 데이터는 변하지 않음

```
callByValue(x)  
print(f"x : {x}")
```

```
xr = [100, 200, 300]
```

— Vector 데이터를 넘겼음으로 xr의 데이터는 변경 될 수 있음

```
callByReference(xr)  
print(f"xr : {xr}")
```

함수 내부 : 101

x : 100

함수 내부 : [101, 200, 300]

xr : [101, 200, 300]

〈 Pure Function 〉 (순수함수)

함수의 실행이 외부 상태에
영향을 끼치지 않는 함수

parameter 의 데이터를 변경해서
parameter로 대입된 원본데이터를 변경하지 않는 함수

- 결과를 return 하는 구조여야 한다
- 동일한 parameter가 오면 동일한 결과를 만들어야 한다 .

원본 데이터에 변형을 가하게 되면
다른 알고리즘을 적용하고자 할때
원본 데이터를 다시 불러와야 하는 불편함이 발생

- 데이터 분석을 할 때는
중간에 변형을 가해야 할 상황이 발생하면
새로운 이름을 부여하고 원본은 그대로 유지

: 일반적인 Programming 언어들은
memory 효율 때문에 원본을 변경하는 경우가 흔함

〈—〉 데이터 분석에 사용되는 것들은 대부분 복사 작업을 한 후 작업을 수행
(R 이 대표적 Python 의 numpy 와 Pandas 의 함수들이 이런식으로 동작)

함수 외부에서 넘겨준 데이터를 변경해서
외부에 영향을 주는 함수를 **modifier function** 이라 한다 .

: 되도록이면 pure function을 만들어서 사용하는 것이 좋다

〈String 과 String build 의 차이〉

String : 원본데이터 변경 불가

String Builder : 원본데이터 변경 가능

〈 Parameter 의 기본값 설정 〉

- 〉 함수에 Parameter 가 있는데
 함수를 호출할 때 parameter 에 대응되는 값을 넘겨주지 않으면 Error

함수 정의 시 Parameter = 기본값
: 사용하면 parameter 에
 대응되는 값이 넘어오지 않을 때
 기본값을 사용한다 .

기본 값을 설정한 parameter 뒤에
 기본 값이 없는 parameter 는 올 수 없다

〈 Parameter 의 이름과 함께 Parameter를 대입해서 함수 호출 〉

- 함수를 만들 때 사용한 Parameter 와
 다른 순서로 parameter 를 대입하는 것이 가능 (가독성이 좋아짐)

첫번째 parameter 는 이름과 함께 대입하지 않는 것이 관례
 일반적으로 첫번째 parameter 는 함수의 기능에서 반드시 필요한 Data

그래프를 그릴 때. Data 는 필수 요소
 이런 경우 데이터는 첫번째 parameter 만들고
 굳이 이름을 입력하지 않아도 되도록 합니다 .

- 〉 첫번째 데이터는 는 이름 안씀
- 〉 두번째 부터 이름을 입력하는 것이 좋음

```
print(help(max))  
# ----> 업로드 할꺼면 반드시 해줘야함  
  
# 2개의 데이터를 대입하면 덧셈을 해서 결과를 리턴하는 함수  
def func1(x, y=0):  
    return x + y  
  
print(func1(1, 3)) # 2개의 데이터를 모두 대입했으므로 1+3  
print(func1(1)) # 하나의 데이터만 대입했으므로 두번째 데이터는 기본값 0 -> 1+0  
print(func1(y=2, x=30))  
# Parmater의 이름과 함께 대입,  
# 대입순서를 변경해서 대입해도 출력이 된다.
```

Result —————
None
4
1
32

```
# 합계 구해주는 함수는 sum
help(sum)

print(sum([100, 300, 200], start=400))
```

Default 값. 설정안하면 0

Help on built-in function sum in module builtins:

sum(iterable, /, start=0)

Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value.

This function is intended specifically for use with numeric values and may reject non-numeric types.

1000

< 매개 변수의 UnPacking >

— 매개변수를 대입할 때.

vector 자료형을 넘겨 줄 때.

앞에 * 붙이면 순서대로 분해해서 대입

dict 의 경우

1. * 순서대로 대입

2. ** key의 이름과 일치하는 parameter에 대입

```
def func3(first1, second2):
    print(f"first number parameter : {first1}")
    print(f"Second Number Parameter : {second2}")
```

원래 사용하는 방식

```
func3("하나", "둘")
```

```
func3("아메", "리카노")
```

순서를 바꾸고 싶다면

```
func3(second2="바꾸자", first1="순서를")
```

first number parameter : 하나
 Second Number Parameter : 둘
 first number parameter : 아메
 Second Number Parameter : 리카노
 first number parameter : 순서를
 Second Number Parameter : 바꾸자

< 매개 변수의 UnPacking >

- 매개변수를 대입할 때.
vector 자료형을 넘겨 줄 때.
앞에 * 붙이면 순서대로 분해해서 대입

dict 의 경우

1. * 순서대로 대입
2. ** key의 이름과 일치하는 parameter에 대입

```
def func3(first1, second2):  
    print(f"first number parameter : {first1}")  
    print(f"Second Number Parameter : {second2}")  
  
# 원래 사용하는 방식  
func3("하나", "둘")  
func3("아메", "리카노")  
  
# 순서를 바꾸고 싶다면  
func3(second2="바꾸자", first1="순서를")
```

first number parameter : 하나
Second Number Parameter : 둘
first number parameter : 아메
Second Number Parameter : 리카노
first number parameter : 순서를
Second Number Parameter : 바꾸자

```
# 매개변수의 unpacking  
func3(*["Coffee", "Bean"])  
  
# ERROR  
func3(["Coffee", "Bean"])
```

first number parameter : Coffee
Second Number Parameter : Bean
Traceback (most recent call last):
File "/Users/mac/PycharmProjects/pythonProject/0126_2nd.py", line 314, in
(module)
 func3(["Coffee", "Bean"])
TypeError: func3() missing 1 required positional argument: 'second2'

```
#Dict 의 경우  
——> Key 값이 호출  
func3(*{"second2": "hi", "first1": "bye"})  
  
——> 키 이름과 일치하는 parameter 대입  
func3(**{"second2": "hi", "first1": "bye"})
```

first number parameter : second2
Second Number Parameter : first1
first number parameter : bye
Second Number Parameter : hi

——> vector 자료형의 데이터를 분해해서 parameter로 대입하는것을
Unpacking 이라고 한다.