기초빅데이터프로그래밍

Python Identifier
Deep copy
Scoping rule

서강대학교
SOGANG UNIVERSITY

식별자(Identifier)와 변수명(Variable Name)

- 식별자는 변수, 함수, 클래스, 객체, 모듈 등을 구분 혹은 식별하기 위해 프로그래머가 사용하는 이름이다.
- 파이썬 식별자 생성 규칙
    ① 식별자는 문자나 밑줄(_)로 시작한다. (대소문자를 구분하며, 한글을 사용해도 된다.)
    ② 식별자는 숫자로 시작해서는 안 된다.
    ③ 예약어를 식별자로 사용할 수 없다.
    ④ 다음과 같은 특수문자, !, @, #, $, %나 공백 문자는 식별자에 사용할 수 없다.
    ⑤ 식별자의 길이 제한은 없다.

- **식별자 생성 규칙을 따르는 경우**
  - abc, hello, st1, _world, 저장소1, 저장소2

- **잘못된 식별자의 예**

  1abc (식별자는 숫자로 시작할 수 없다.)
  if (예약어를 식별자로 사용해서는 안 된다.)
  msg@book (식별자에 @라는 특수 문자가 들어 갈 수 없다.)
  st 1 (식별자 중간에 공백 문자를 사용할 수 없다.)

- 잘못된 식별자를 사용하는 경우 SyntaxError 발생

```
>>> 1abc = 1
File "<stdin>", line 1
1abc = 1
^
SyntaxError: invalid syntax
>>> if = 1
File "<stdin>", line 1
if = 1
^
SyntaxError: invalid syntax
>>> msg@book = "Hello World"
File "<stdin>", line 1
msg@book = "Hello World"
^
SyntaxError: invalid syntax
>>> st 1 = 1
File "<stdin>", line 1
st 1 = 1
^
SyntaxError: invalid syntax
```

접근지정(Access Modifiers)

- 파이썬에서는 객체의 속성이나 메서드의 접근을 제어하는 접근 지정자(public, private, protected)가 없이, 이름규칙을 통해 처리된다.
- 접근지정을 구분하기 위해 밑줄(_) 사용한다.

- **밑줄이 없는 경우**: 공개 모드로 객체외부에서 접근가능
- **밑줄이 한 개인 경우 _** : 보호 모드로 객체외부에서 접근불가능
  - 비공개 method, 비공개 instance 변수 이름
- **밑줄이 두 개인 경우 __** : 비공개 모드로 객체외부뿐만 아니라 상속받은 객체에서도 접근불가능
  - 상속된 클래스의 method, instance 변수 이름과의 충돌 방지하기 위해 사용

__name__ 변수

- 현재 실행되는 모듈의 이름을 담고 있는 내장 변수
- 파이썬 인터프리터가 파이썬 프로그램을 입력 받아서 실행하면 __name__ 을 "__main__"으로 설정한다.

```
# hello_mge.py

def hello_message( ):
        print("Hello World")
        print("서강대학교")

if __name__ == "__main__":
        hello_message( )
else:
        print("Bye")
```

```
%%writefile hello_mge.py

def hello_message( ):
    print("Hello World")
    print("서강대학교")

if __name__ == "__main__":
    hello_message( )
else:
    print("Bye")
```

```
Writing hello_mge.py
```

```
%run hello_mge.py
```

```
Hello World
서강대학교
```

```
import hello_mge
```

```
Bye
```

command line 모듈 실행

- 직접 모듈 호출할 경우 __name__의 값이 모듈명이 아닌 __main__으로 처리
- 실제 모듈에서 호출된 것과 import되어 활용하는 부분을 별도로 구현이 가능

```
if __name__ == "__main__":    # 직접 모듈 호출시 실행되는 영역
    print(safe_sum('a', 1))
    print(safe_sum(1, 4))
    print(sum(10, 10.4))
else :                        # import 시 실행되는 영역
```

NameA.py - C:/CT/소스코드/NameA.py (3.6.2)

File Edit Format Run Options Window Help

```
def func():
    print("function A.py야!")

print("top-level A.py")

if __name__ == "__main__":
    print("A.py 직접 실행")
else:
    print("A.py가 임포트되어 사용됨")
```

2017_NameB.py - C:/CT/소스코드/2017_NameB.py (3.6.2)

File Edit Format Run Options Window Help

```
import NameA

print("top-level in B.py")
NameA.func()

if __name__ == "__main__":
    print("B.py가 직접 실행")
else:
    print("B.py가 임포트되어 사용됨")
```

```
========== RESTART: C:/CT/소스코드/NameA.py ===
top-level A.py
A.py 직접 실행
>>>
>>>
========== RESTART: C:/CT/소스코드/2017_NameB.py =
top-level A.py
A.py가 임포트되어 사용됨
top-level in B.py
function A.py야!
B.py가 직접 실행
>>>
```

In [1]:
```
%%writefile usingNameTest.py

if __name__ == '__main__':
    print ("This program is being run by itself")
else:
    print ("I am being imported from another module")
```

```
Writing usingNameTest.py
```

In [2]:
```
%run usingNameTest.py
```

```
This program is being run by itself
```

In [3]:
```
import usingNameTest
```

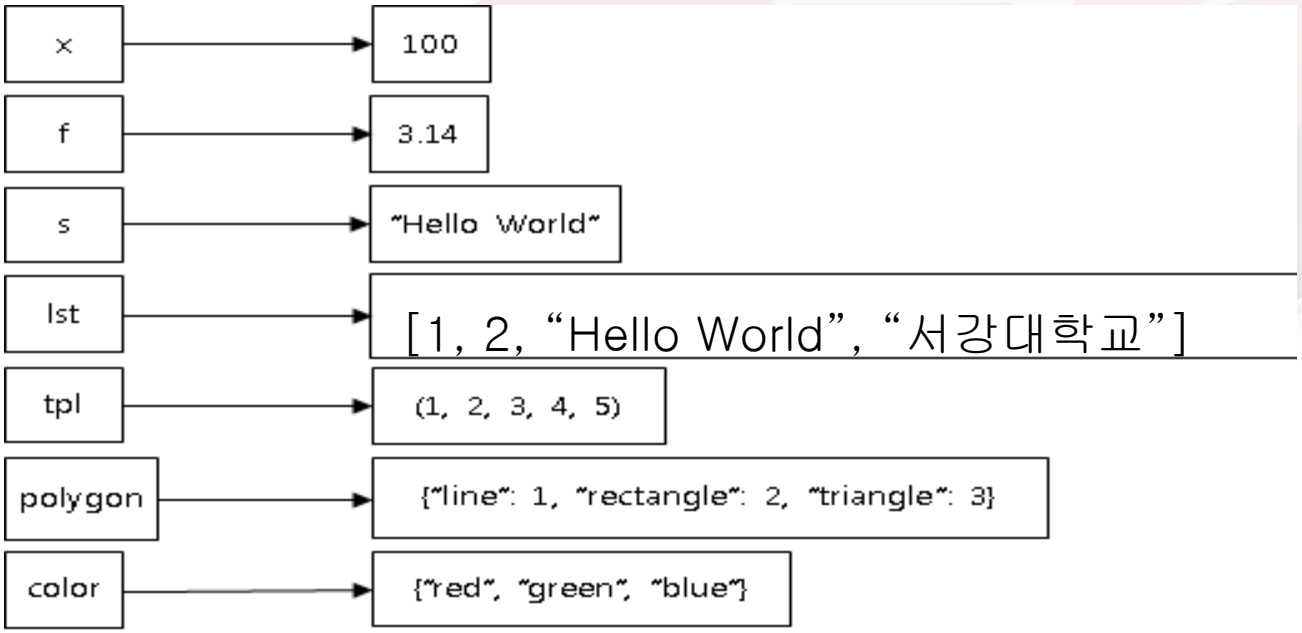
```
I am being imported from another module
```

magic command에는 line(%)과 cell(%%)로 지정해서 처리할 수 있음

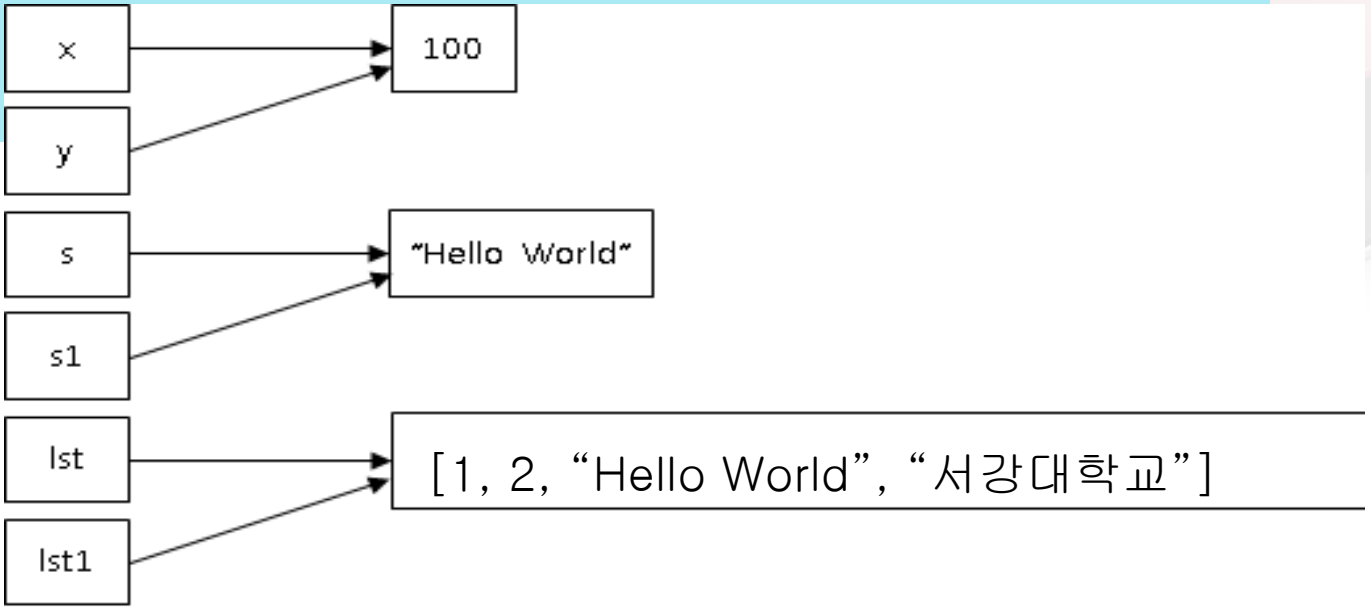
변수명과 데이터

- **= 연산자를 이용해서 데이터에 이름을 붙인다.**

```
>>> x = 100
>>> f = 3.14
>>> s = "Hello World"
>>> lst = [1, 2, "Hello World", "서강대학교"]
>>> tpl = (1, 2, 3, 4, 5)
>>> polygon = {"line": 1, "rectangle": 2, "triangle": 3}
>>> color = {"red", "green", "blue"}
```



```
>>> x = 100
>>> y = x  # 얕은 복사(shallow copy)
>>> y
100
>>> s = "Hello World"
>>> s1 = s
>>> s1
'Hello World'
>>> lst = [1, 2, "Hello World", "서강대학교"]
>>> lst1 = lst
>>> lst1
[1, 2, 'Hello World', '서강대학교']
```



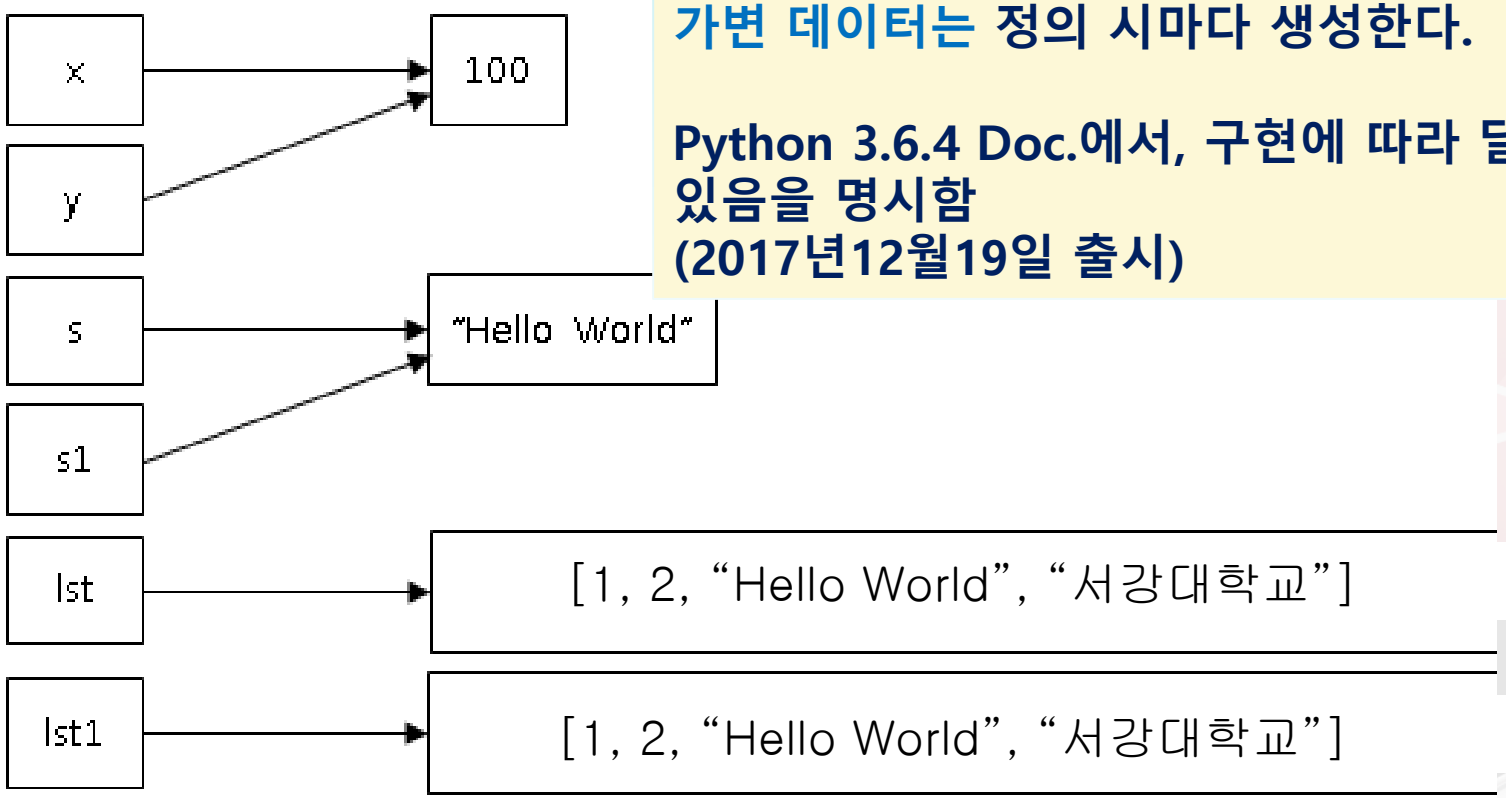
서강대학교 SOGANG UNIVERSITY

- # 불변 데이터 타입과 가변 데이터 타입의 변수명

```
>>> x = 100    #정수형
>>> y = 100
>>> id(x) == id(y)
True
>>> s = "Hello World"  #문자형
>>> s1 = "Hello World"
>>> id(s) == id(s1)
True
>>> lst = [1, 2, "Hello World", "서강대학교"]  #리스트 유형
>>> lst1 = [1, 2, "Hello World", "서강대학교"]
>>> id(lst) == id(lst1)
False
```

불변데이터는 객체를 하나만 생성
가변 데이터는 정의 시마다 생성한다

id(object) : 객체의 고유값을 반환하는 함수



불변데이터는 객체를 하나만 생성하지만,
가변 데이터는 정의 시마다 생성한다.

Python 3.6.4 Doc.에서, 구현에 따라 달라질 수 있음을 명시함
(2017년12월19일 출시)

```
In [1]: x=257
        y=257
        id(x)==id(y)
```

```
Out[1]: False
```

```
In [3]: x1="hello world"
        y1="hello world"
        id(x1)==id(y1)
```

```
Out[3]: False
```

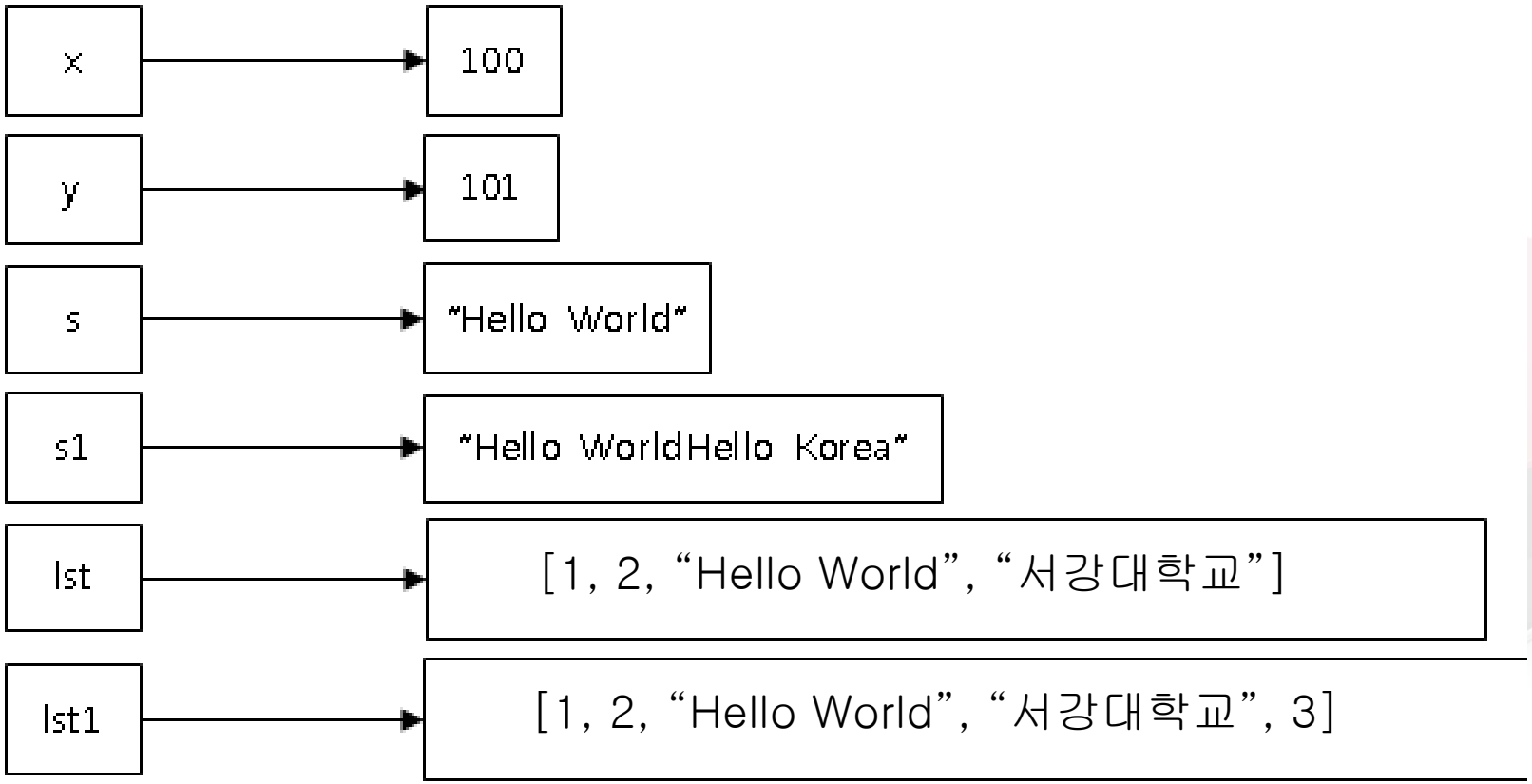
```
In [4]: x="hello"
        y="hello"
        id(x)==id(y)
```

```
Out[4]: True
```

서강대학교 SOGANG UNIVERSITY

- **불변 데이터 타입과 가변 데이터 타입의 연산**

```
>>> x = 100
>>> y = 100
>>> y = y + 1
>>> x
100
>>> y
101
>>> s = "Hello World"
>>> s1 = "Hello World"
>>> s1 = s1 + "Hello Korea"
>>> s
"Hello World"
>>> s1
"Hello WorldHello Korea"
>>> lst = [1, 2, "Hello World", "서강대학교"]
>>> lst1 = [1, 2, "Hello World", "서강대학교"]
>>> lst1.append(3)
>>> lst
[1, 2, "Hello World", "서강대학교"]
>>> lst1
[1, 2, "Hello World", "서강대학교", 3]
```

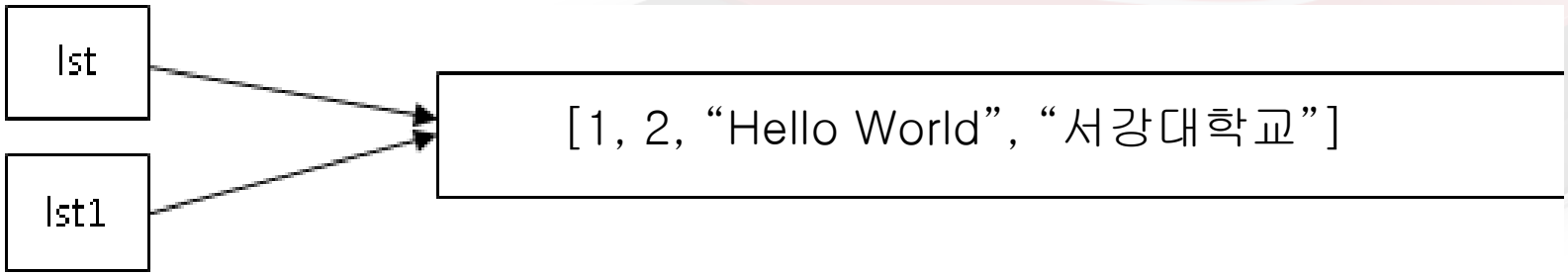

x
100
y
101
s
"Hello World"
s1
"Hello WorldHello Korea"
lst
[1, 2, "Hello World", "서강대학교"]
lst1
[1, 2, "Hello World", "서강대학교", 3]

서강대학교 SOGANG UNIVERSITY

얕은 복사와 깊은 복사

- 가변 데이터 타입을 가리키는 변수명을 다른 변수명에 대입한 경우 얕은 복사가 일어난다.

```
>>> lst = [1, 2, "Hello World", "서강대학교"]
>>> lst1 = lst
```



얕은 복사를 이용한 데이터 조작

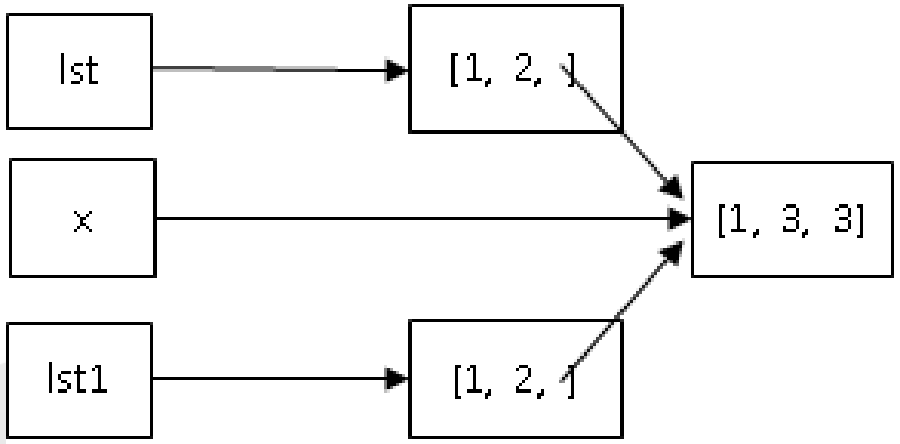
lst1 = lst

```
>>> lst1
[1, 2, "Hello World", "서강대학교"]
>>> lst1.append(3)
>>> lst1
[1, 2, "Hello World", "서강대학교", 3]
>>> lst
[1, 2, "Hello World", "서강대학교", 3]
```

- **같은 내용을 가지지만 다른 객체를 가리키게 할 방법**
  1) 매번 가변 데이터를 생성해서 사용
  2) **deepcopy** 함수를 사용

가변 데이터 내에 가변 데이터를 참조하는 경우

```
>>> x = [1, 2, 3]
>>> lst = [1, 2, x]
>>> lst1 = [1, 2, x]
>>> id(lst) == id(lst1)
False
>>> lst
[1, 2, [1, 2, 3]]
>>> lst1
[1, 2, [1, 2, 3]]
>>> lst1[2][1] = 3
>>> lst1
[1, 2, [1, 3, 3]]
>>> lst
[1, 2, [1, 3, 3]]
```



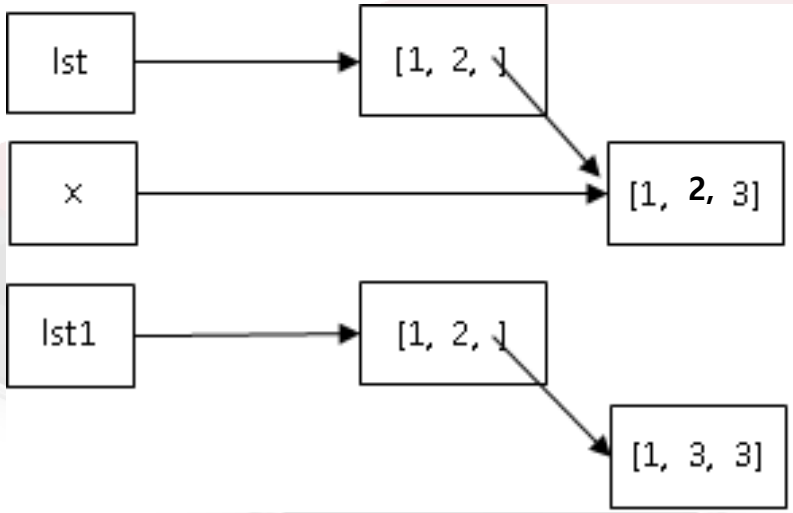
가변데이터를 따로 만들어 사용해도 가변 데이터 내에 가변 데이터를 참조하는 경우는 같이 수정된다.

서강대학교 SOGANG UNIVERSITY

deepcopy를 이용한 복사

– 원본 데이터와 복사본 데이터를 확실히 분리 관리하기 위해

```
>>> from copy import deepcopy
>>> x = [1, 2, 3]
>>> lst = [1, 2, x]
>>> lst1 = deepcopy(lst)
>>> id(lst) == id(lst1)
False
>>> lst
[1, 2, [1, 2, 3]]
>>> lst1
[1, 2, [1, 2, 3]]
>>> lst1[2][1] = 3
>>> lst1
[1, 2, [1, 3, 3]]
>>> lst
[1, 2, [1, 2, 3]]
```



서강대학교 SOGANG UNIVERSITY

```
In [1]: x=[1,2,3]
        lst=[1,2,x]
        lst1=[1,2,x]
        id(lst) == id(lst1)

Out[1]: False

In [2]: lst

Out[2]: [1, 2, [1, 2, 3]]

In [3]: lst1

Out[3]: [1, 2, [1, 2, 3]]

In [4]: lst1[2][1]=3
        lst1

Out[4]: [1, 2, [1, 3, 3]]

In [5]: lst

Out[5]: [1, 2, [1, 3, 3]]
```

```
In [1]: from copy import deepcopy
        x= [1,2,3]
        lst = [1,2,x]
        lst1 = deepcopy(lst)
        id(lst) == id(lst1)

Out[1]: False

In [2]: lst

Out[2]: [1, 2, [1, 2, 3]]

In [3]: lst1

Out[3]: [1, 2, [1, 2, 3]]

In [4]: lst1[2][1]=3
        lst1

Out[4]: [1, 2, [1, 3, 3]]

In [5]: lst

Out[5]: [1, 2, [1, 2, 3]]
```

Scoping Rule

- 변수가 사용되는 범위 (함수 또는 메인프로그램)
- 지역변수(local variable) : 함수 내에서만 사용
  전역변수(Global variable) : 프로그램전체에서 사용
- 전역변수는 함수에서 사용가능하나,
  함수내에서 전역변수에 새로운 값을 할당했을 경우,
  새로운 지역변수가 생김
- 함수내에 전역변수 사용시 global 키워드 사용

Scoping Rule

```
def calculate(x, y):
    total = x + y  # 새로운 값이 할당되어 함수 내 total은 지역변수가 됨
    print( "In Function")
    print( "a:", str(a), "b:", str(b), "a+b:", str(a+b), "total :", str(total))
    return total
a = 5              # a와 b는 전역변수
b = 7
total = 0          # 전역변수 total
print ("In Program ")
print ("a:", str(a), "b:", str(b), "a+b:", str(a+b))

sum = calculate (a,b)
print ("After Calculation")
print ("Total :", str(total), " Sum:", str(sum) )# 지역변수는 전역변수에 영향 X
```

```
def calculate(x,y):
    total = x + y
    print ("in Function")
    print ("a:",str(a), "b:", str(b), "a+b:", str(a+b), "total:", str(total))
    return total
a=5
b=7
total =0
print("In program")
print("a:",str(a), "b:", str(b), "a+b:", str(a+b), )

sum=calculate(a,b)
print ("After calculation")
print ("total:", str(total), "Sum:", str(sum))
```

```
In program
a: 5 b: 7 a+b: 12
in Function
a: 5 b: 7 a+b: 12 total: 12
After calculation
total: 0 Sum: 12
```

global keyword

In [3]:
```
total = 0
def calculate(x,y):
    global total
    total = x + y
    print ("in Function")
    print ("a:",str(a), "b:", str(b), "a+b:", str(a+b), "total:", str(total))
    return total
a=5
b=7
print("In program")
print("a:",str(a), "b:", str(b), "a+b:", str(a+b), )

sum=calculate(a,b)
print ("After calculation")
print ("total:", str(total), "Sum:", str(sum))
```

```
In program
a: 5 b: 7 a+b: 12
in Function
a: 5 b: 7 a+b: 12 total: 12
After calculation
total: 12 Sum: 12
```

SWAP

swap_offset: a 리스트의 전역 변수 값을 직접 변경

swap_reference: a 리스트 객체의 주소값을 받아 값을 변경

In [3]:
```
def swap_value(x, y):
    temp = x
    x = y
    y = temp

def swap_offset(offset_x, offset_y):
    temp = a[offset_x]
    a[offset_x]= a[offset_y]
    a[offset_y]= temp

def swap_reference(list, offset_x, offset_y):
    temp = list[offset_x]
    list[offset_x]= list[offset_y]
    list[offset_y]= temp

a =[1,2,3,4,5]
swap_value(a[1], a[2])
print (a)

swap_offset(1,2)
print(a)

swap_reference(a,1,2)
print(a)
```

```
[1, 2, 3, 4, 5]
[1, 3, 2, 4, 5]
[1, 2, 3, 4, 5]
```

실습: Fibonacci number 구하기

- **피보나치 수** F_{n}는 다음과 같은 초기값 및 점화식으로 정의되는 수열이다.

  F_{1}=F_{2}=1

  F_{n}=F_{n-1}+F_{n-2}, n in {3,4,...}

- 0번째 항부터 시작할 경우 다음과 같이 정의된다.

  F_{0}=0

  F_{1}=1

  F_{n}=F_{n-1}+F_{n-2}, n in {2,3,4,...}

- 피보나치 수의 처음 몇 항은 (0번째 항부터 시작할 경우) 다음과 같다
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,...

실습

- 입력 받은 수의 피보나치 수를 구하는 파이썬 프로그램 fibonacci.py를 magic command를 이용해서 만들어 실행시키시오.

```
%run fibonacci.py
```

```
input:
```

```
%run fibonacci.py
```

```
input: 7
13
```

- 스스로 작성한 피보나치 수를 구하는 파이썬 프로그램을 import하여 해결하시오. 예를 들어, 파일이름 fibona.py에 함수 fibo()를 작성해서 해결한 경우

```
import fibona
fibona.fibo(7)
```

```
13
```