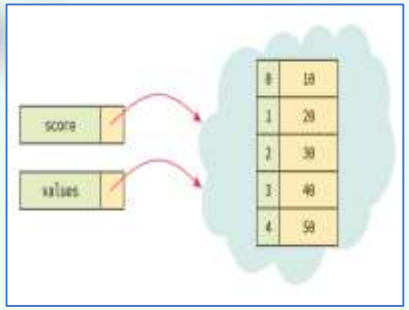
제9장
리스트-2

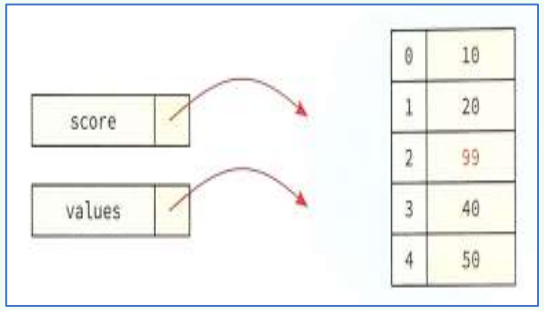
1. 리스트 복사하기

1) 얕은 복사(shallow copy)

- 파이썬에서 리스트 변수는 리스트 객체를 직접 저장하고 있지 않다. 리스트 자체는 다른 곳에 저장되고 리스트의 참조값(reference)만 변수에 저장된다. 참조값이란 메모리에서 리스트 객체의 주소값이다. 평상시에는 이런 사소한 것에 신경 쓸 필요는 없다.
  하지만, 리스트를 복사하려고 할 때는 약간의 신경을 써야 한다.
- 만약 리스트를 복사하기 위하여 아래와 같은 문장을 실행하였다고 하자. 어떤 일이 벌어질까?
- 결론부터 말하자면 리스트는 복사되지 않는다. scores와 values는 모두 동일한 리스트 객체를 가리키고 있다.
  values는 scores 리스트의 별칭이나 마찬가지이다. 이것을 얕은 복사(shallow copy)라고 한다.

```
scores = [ 10, 20, 30, 40, 50 ]
values = scores
```





* 만약 values를 통하여 리스트 요소의 값을 변경한다면 scores 리스트도 변경된다. 이것을 확인해보자.

```
scores = [ 10, 20, 30, 40, 50 ]
values = scores
values[2] = 99
for element in scores:
        print(element, end=" ")
```

```
10 20 99 40 50
```

위의 결과와 마찬가지로 얕은 복사는 원본 리스트의 값까지 변경시킨다. 그 이유는 같은 주소값을 가지고 있기 때문이다.

1. 리스트 복사하기

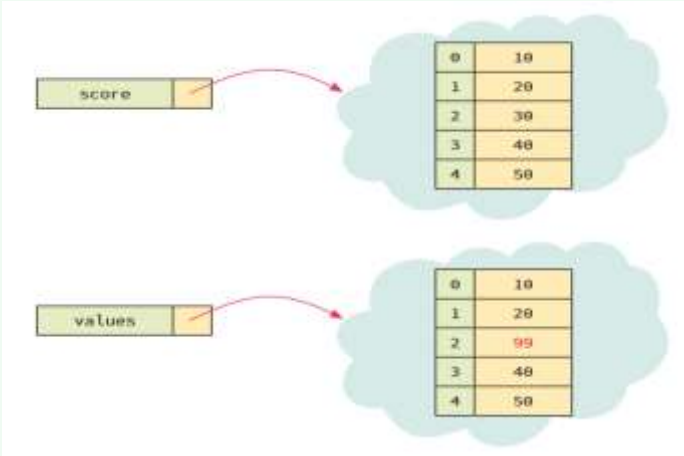
2) 깊은 복사(deep copy)

- 그렇다면 리스트를 올바르게 복사하는 방법은 무엇일까? 이것을 깊은 복사(deep copy)라고 한다. 몇 가지 방법이 있다.

- 첫 번째 방법은 list() 메소드를 사용하는 것이다. list() 내장 함수는 리스트를 받아서 복사본을 생성하여 반환한다.

```
scores = [ 10, 20, 30, 40, 50 ]
values = list(scores)
values [2]=99
print(scores)
print(values)
```

```
[10, 20, 30, 40, 50]
[10, 20, 99, 40, 50]
```



- 두 번째 방법은 deepcopy()나 copy()메소드를 사용하는 방법도 있다. deepcopy()나 copy() 내장 함수도 리스트를 받아서 복사본을 생성하여 반환한다. 깊은 복사를 하게 되면 다른 주소값을 지니게 되어 원본 리스트에는 영향을 끼치지 아니한다.

```
from copy import deepcopy
scores = [ 10, 20, 30, 40, 150 ]
values = deepcopy(scores)
```

* 참고로 비교 연산자였던 == 은 리스트 요소의 값을 비교하여 같다면 True를 리턴 하였지만, 주소값을 비교하는 키워드는 바로 is 이다.
얕은 복사를 하면 주소값이 같고, 깊은 복사를 하면 주소값이 틀리다.

1. 리스트 복사하기

2) 깊은 복사(deep copy)

- 마지막으로 세 번째 방법은 [:]인덱스를 이용하는 것이다.

```
scores = [ 10, 20, 30, 40, 50 ]
values = scores[:]
values[2]=99
print(scores)
print(values)
```

```
[10, 20, 30, 40, 50]
[10, 20, 99, 40, 50]
```

2. 리스트와 함수

1) “값으로 호출하기"와 참조로 호출하기"

- 이미 우리는 앞서서 함수로 인수를 전달하는 방식에는 다음과 같은 2가지의 방법이 있다는 것을 알고 있다.

1. “값으로 호출하기”(Call-by-Value) : 함수로 변수를 전달할 때, 변수의 값이 복사되는 방식으로 가장 많이 사용되는 방법이다.
2. “참조로 호출하기”(Call-by-Reference) : 함수로 변수를 전달할 때, 변수의 참조가 전달되는 방법이다. 함수에서 매개 변수를 통하여 원본 변수를 변경할 수 있다.

- 그렇다면 파이썬에서는 어떤 방법을 사용하는가? 파이썬에서는 정수나 문자열처럼 변경이 불가능한 객체들은 “값으로 호출하기” 방법으로 전달된다. 객체의 참조값이 함수의 매개변수로 전달되지만 함수 안에서 객체의 값을 변경하면 새로운 객체가 생성되기 때문이다.

```
def func1(x) :
        print( "x=", x," id=",id(x))
        x=42  # 새로운 객체 생성
        print(  " x= " , x, "  id= " ,id(x))
y = 10
print( "y=",y," id=",id(y))
func1(y)
print( "y=",y," id=",id(y))
```

```
y= 10     id= 1640249248
x= 10     id= 1640249248
X= 42     id= 1640249760
y= 10     id= 1640249248
```

2. 리스트와 함수

1) “값으로 호출하기"와 참조로 호출하기"

- 변경가능한 객체인 리스트를 함수에 전달하면 어떻게 될까? 상황은 달라진다. 리스트는 참조값으로 전달된다. 리스트는 함수 안에서 변경할 수 있다. 즉 리스트의 요소들은 변경될 수 있는 것이다.

```
def func2(list):
    list[0] = 99

values = [0, 1, 1, 2, 3, 5, 8]
print(values)
func2(values)
print(values)
```

```
[0, 1, 1, 2, 3, 5, 8]
[99, 1, 1, 2, 3, 5, 8]
```

3. 리스트 함축

1) 리스트 함축(list comprehensions)

- 파이썬은 "리스트 함축(list comprehensions)" 또는 "리스트 컴프리헨션"이라는 개념을 지원한다. comprehension은 함축, 포함, 내포라는 의미이다.

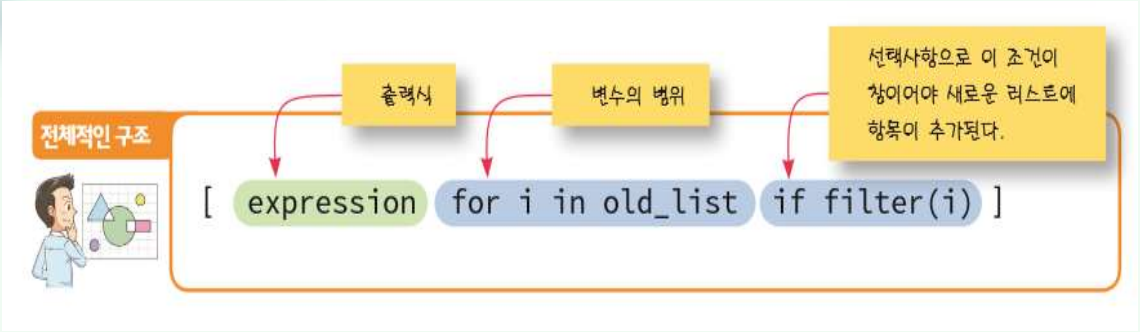
  리스트 함축은 수학자들이 집합을 정의하는 것과 유사하다. 수학에서 제곱값의 집합은 { x^2 | x ∈ N }와 같이 정의된다. 즉 자연수에 속하는 x에 대하여 x^2 값들이 모여서 집합을 생성한다. 파이썬에서 리스트를 수학과 유사하게 정의할 수 있다. 파이썬에는 다음과 같이 0부터 9까지의 자연수에 대하여 제곱값의 리스트를 정의할 수 있다.

```
s = [ x**2 for x in range(10) ]
```

- 위의 문장을 해석해보면 다음과 같다. range(10)에 속하는 모든 정수에 대하여 x^2을 계산하여서 리스트를 생성한다. 결과적으로 아래와 같은 리스트가 된다.

```
s = [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

- 리스트 함축은 다음과 같은 형식을 가진다.



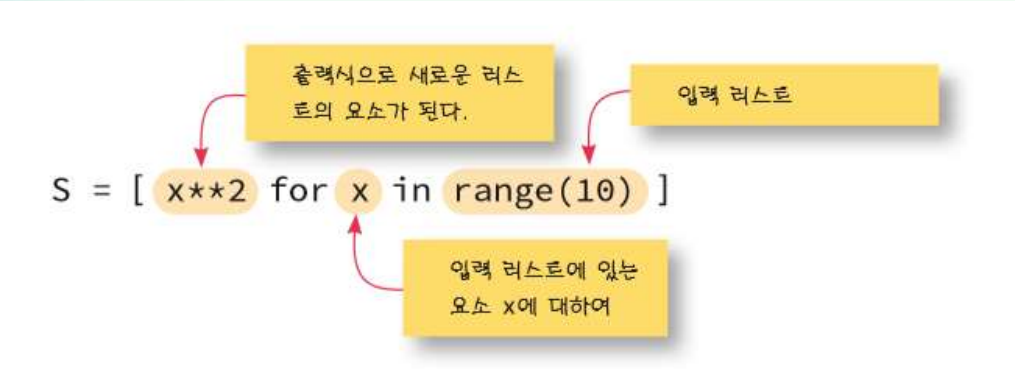
좌측 그림을 코드로 풀어쓴다면 아래와 같다.

```
new_list = []
for i in old list:
        if filter(i):
                new_list.append(i)
```

3. 리스트 함축

1) 리스트 함축(list comprehensions)

- 리스트 함축을 사용하면 아주 간결하게 리스트를 생성할 수 있다는 큰 장점이 존재한다. 앞 슬라이드에서 보았던 제곱의 집합을 리스트 함축으로 표현한 예를 다시 한번 분석하여 보자.



- x는 입력 리스트의 요소를 나타내는 변수이다.
- x**2는 출력식으로서 새로운 리스트의 요소를 생성한다.
- range(10)은 입력 리스트를 나타낸다.
- [...]은 결과가 새로운 리스트라는 것을 의미한다.

만약 좌측과 같이 리스트 함축을 사용하지 않는다면 아래와 같이 반복문을 사용해야 할 것이다.

```
squares = []
for x in range(10):
        squares.append(x**2)
```

리스트 [3, 4, 5]의 모든 항목에 2를 곱해서 새로운 리스트를 생성하는 문장은 아래와 같다.

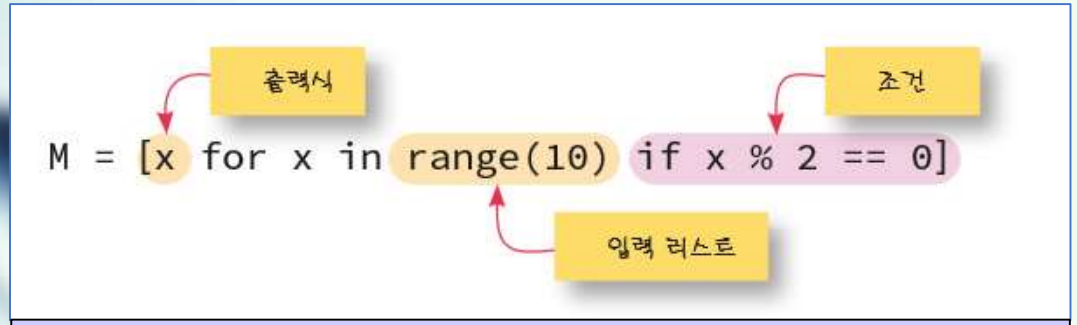
```
list1 = [3, 4, 5]
list2 = [x*2 for x in list1]
print(list2)
```

```
[6, 8, 10]
```

3. 리스트 함축

2) 조건이 붙는 리스트 함축

- 리스트 함축에는 if를 사용하여 조건이 추가될 수 있다. 예를 들어서 0부터 9사이의 정수 중에서 짝수의 집합을 리스트 함축으로 표현하면 다음과 같다.




```
[0, 2, 4, 6, 8]
```

3) 다양한 자료형에 대한 리스트 함축

- 리스트 함축은 숫자에 대해서만 적용되는 것은 아니다. 어떤 자료형에 대해서도 리스트 함축을 적용할 수 있다. 단어를 저장하는 리스트를 가정하자. 단어의 첫 글자만을 추출하여 리스트로 만드는 문장을 작성해보자.

```
list1 = ["All", "good", "things", "must", "come", "to", "an", "end."]
items = [word[0] for word in list1]
print (items)
```

```
['A', 'g', 't', 'm', 'c', 't', 'a', 'e']
```

3. 리스트 함축

3) 다양한 자료형에 대한 리스트 함축

- 추가적인 예로 문자열을 구성하는 단어를 추출하여 단어의 길이를 계산하고 이것을 모아서 새로운 리스트로 생성해보자.

```
word_list = "Doncount your chickens before they are hatched".split()
result_list = [ len(w) for w in word_list]
print(result_list)
```

- 첫 번째 문장에서 문자열을 분리하여 리스트를 만든다. 두 번째 문장에서 word_list 리스트에 있는 각각의 단어에 대하여 단어의 길이를 계산하여 리스트에 추가한다. 세 번째 문장에서 result_list를 출력한다. 실행 결과는 다음과 같다.

```
[8, 4, 8, 6, 4, 3, 7]
```

4) 상호곱(Cross product) 형태의 집합

- 리스트 함축은 2개의 집합의 상호곱 형태로도 표현할 수 있다. 예를 들어서 색상의 집합과 자동차의 집합을 상호 곱하여 새로운 리스트를 생성할 수 있다.
- 다음 슬라이드에 이어서 설명하겠다.

3. 리스트 함축

4) 상호곱(Cross product) 형태의 집합

- 아래 코드는 colors 리스트의 원소와 cars 리스트의 원소가 하나씩 짝지어져서 튜플이 되고 이 튜플이 모여서 리스트가 된다.

```
colors = [ "white", "silver", "black" ]
cars = [ "bmw5", "sonata", "malibu", "sm6" ]
colored_cars = [ (x, y) for x in colors for y in cars ]
print(colored_cars)
```

```
[('white', 'bmw5'), ('white', 'sonata'), ('white', 'malibu'), ('white', 'sm6'), ('silver', 'bmw5'), ('silver', 'sonata'), ('silver', 'malibu'), ('silver', 'sm6'), ('black', 'bmw5'), ('black', 'sonata'), ('black', 'malibu'), ('black', 'sm6')]
```

4. 일반적인 리스트 연산들

1) 최소값과 최대값 구하기

- 앞에서 min()과 max()를 이용하여 숫자들이 들어 있는 리스트에서 최소값과 최대값을 구할 수 있었다. 하지만 리스트에 숫자가 아닌 다른 종류의 데이터가 들어 있다면 일반적인 알고리즘을 알고 있어야 한다.

  리스트에 저장된 값들의 최대값이나 최소값을 어떻게 계산하는지를 생각해보자. 이것은 실제 프로그래밍에서도 상당히 많이 등장하는 문제이므로 정확하게 알고 있어야 한다.

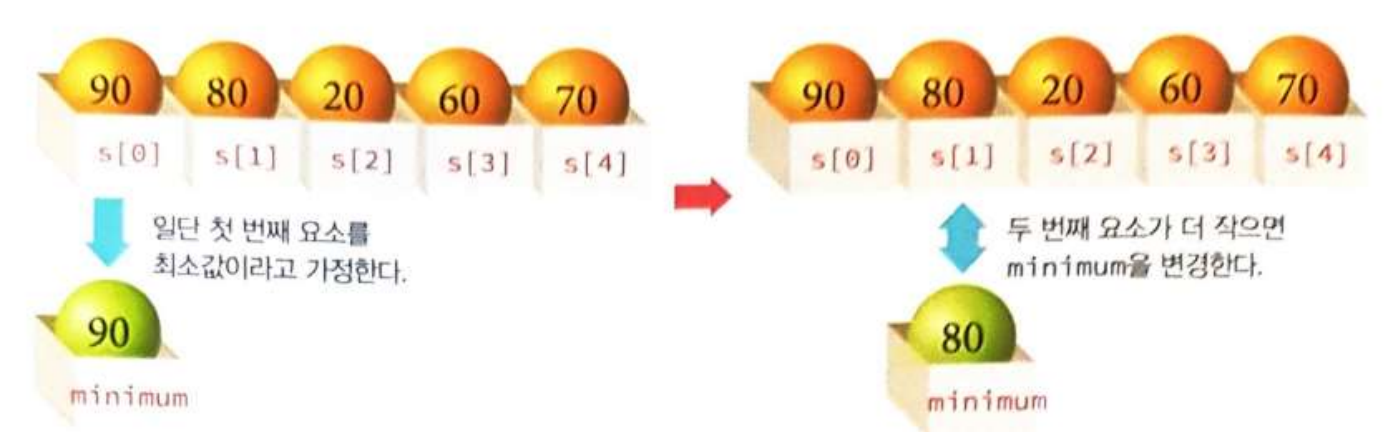
- 예를 들어 보면, 우리는 인터넷에서 특정한 상품(예를 들어서 TV)을 구입하고자 한다. 여러 인터넷 사이트에서 판매되는 가격이 1차원 리스트 prices[]에 저장되어 있다고 가정하자. 어떻게 하면 최소 가격으로 상품을 구입할 수 있을까? 리스트 요소 중 에서 최소값을 구하면 된다. 최소값을 구하는 알고리즘을 생각해보자.

- 최소값을 구할 때는 일단 리스트의 첫 번째 요소를 최소값으로 가정한다. 리스트의 두 번째 요소부터 마지막 요소까지 이 최소값과 비교한다. 만약 어떤 요소가 현재의 최소값보다 작다면 새로운 최소값을 이것으로 변경하면 된다. 모든 요소들의 검사가 종료되면 최소값을 찾을 수 있다.

```
# 첫 번째 요소를 최소값, minimum이라고 가정한다.
for i in range(1, 리스트의 크기) :
        if s[i] < minimum :
                minimum = s[i]
# 반복이 종료되면 minimum에 최소값이 저장된다.
```

4. 일반적인 리스트 연산들

1) 최소값과 최대값 구하기



- 최소값과 최대값을 계산하는 일반적인 알고리즘은 다음과 같다. 최소값을 계산한다고 가정하자.

```
lowPrice = prices[0]
for x in range(1, len(prices)):
        if prices[i] < lowPrice :
                lowPrice = prices[i]
print("가장 싼 가격은 ", lowPrice)
```

4. 일반적인 리스트 연산들

1) 최소값과 최대값 구하기

- 문자열이 저장된 리스트를 가정하자. 가장 길이가 짧은 문자열을 찾으려고 한다. min()을 사용하면 알파벳 순서로 가장 앞에 있는 문자열을 반환한다. 따라서 우리는 나름대로의 코드를 작성해야 한다.

```
words = ["cat", "mouse", "tiger", "lion" ]
shortest = words[0]
for i in range(1, len(words)):
        if len(words[i]) < len(shortest) :
                shortest = words[i]
print("가장 짧은 단어는", shortest)
```

```
가장 짧은 단어는 cat
```

2) 탐색하기

- 사람들은 항상 무엇인가를 찾아 헤맨다. 예를 들면 출근할 때 입을 옷을 찾는다거나 서랍속의 서류를 찾기도 한다. 컴퓨터에서도 마찬가지이다. 탐색(search)은 컴퓨터가 가장 많이 하는 작업 중의 하나이다. 단순하게 여러분이 하루에 인터넷에서 필요한 자료들을 얼마나 많이 탐색하는지를 생각하면 된다. 탐색은 많은 시간이 요구되는 작업이므로 효율적으로 수행하는 것은 매우 중요하다.

4. 일반적인 리스트 연산들

2) 탐색하기

- 탐색의 대상이 되는 데이터는 보통 리스트에 지장이 있다고 가정한다. 탐색이란 리스트에서 특정한 값을 찾는 것이다. 리스트에서 탐색은 앞서 살펴보았던 index() 메소드를 사용할 수 있다. index()는 리스트에서 항목을 찾아서 항목의 인덱스를 반환한다.

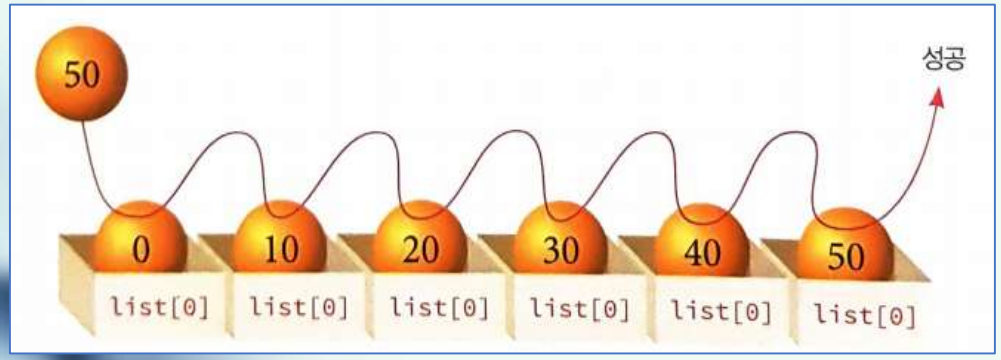
```
list1 = [ "white", "silver", "blue", "red", "black" ]
print(list1.index ("red"))
```

```
3
```

- 하지만 경우에 따라서는 프로그래머가 탐색을 구현하여야 하는 경우도 있다. 따라서 우리는 탐색의 기본적인 방법을 알아두자. 여기서는 가장 간단한 알고리즘인 순차 탐색만 살펴보자. 순차 탐색(sequential search)은 탐색 방법 중에서 가장 간단하고 직접적인 탐색 방법이다. 순차 탐색은 리스트의 요소를 순서대로 하나씩 꺼내서 탐색키와 비교하여 원하는 값을 찾아가는 방법이다. 순차 탐색은 일치하는 항목을 찾을 때까지 비교를 계속한다. 순차 탐색은 첫 번째 요소에서 성공할 수도 있고 마지막 요소까지 가야 되는 경우도 있다. 평균적으로는 절반 정도의 리스트 요소와 비교하여야 할 것이다.

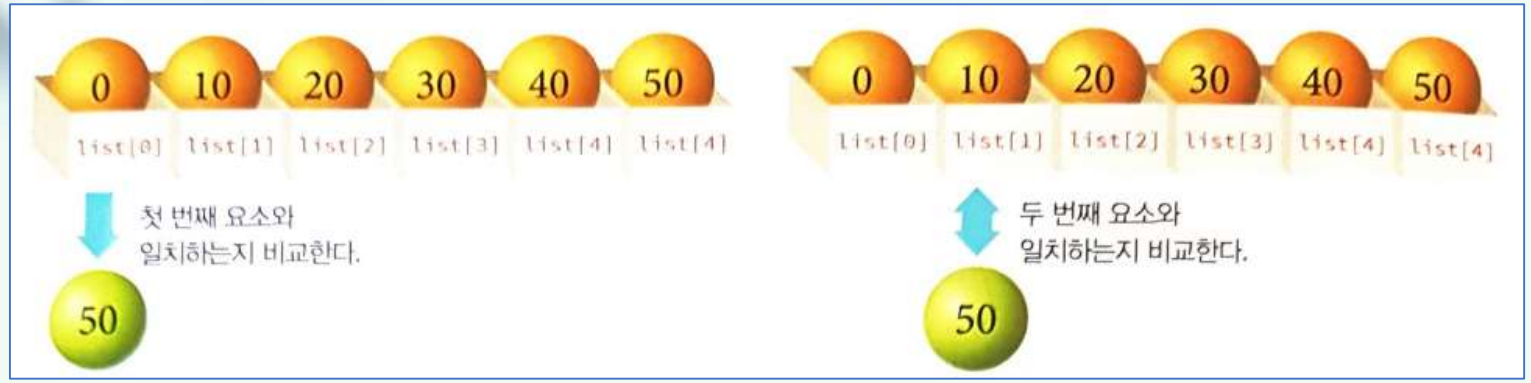
4. 일반적인 리스트 연산들

2) 탐색하기



- 순차 탐색 알고리즘을 파이썬 함수로 표현하면 다음과 같다.

```
def linear_Search(aList, key):
        for I in range(len(aList)):
                if key == aList[i]:
                        탐색 성공이고 복귀한다.
        복귀하지 않고 반복루프가 종료되었으면 탐색 실패이다.
```



4. 일반적인 리스트 연산들

2) 탐색하기

- 순차 탐색 알고리즘을 파이썬으로 구현하고 테스트하는 코드는 다음과 같다.

```
def linear_Search(aList, key):
        for i in range(len(aList)):        # 리스트의 길이만큼 반복한다.
                if key == alist[i]:        # 키가 발견되면 i를 반환하고 종료한다.
                        return i
        return -1          # 키가 발견되지 않고 반복이 종료되었으면 -1을 반환한다.

numbers = [ 10, 20, 30, 40, 50, 60, 70, 80, 90 ]
position = linear_Search(numbers, 80)

if position != -1:
        print("탐색 성공 위치 = ", position)
else:
        print("탐색 실패 ")
```

```
탐색 성공 위치 = 7
```

4. 일반적인 리스트 연산들

3) 조건을 만족하는 항목 모두 찾기

- 앞에서는 조건을 만족하는 첫 번째 요소의 위치만을 찾았다. 만약 조건을 만족하는 모든 요소를 찾으려면 어떻게 해야 할까? 공백 리스트를 만들고 요소가 발견될 때마다 여기에 추가하면 쉽다. 예를 들어서 리스트에서 50이 넘는 숫자들을 모두 찾는 코드는 다음과 같다.

```
numbers = [ 10, 20, 30, 40, 50, 60, 70, 80, 90 ]
result = []
for value in numbers:
        if value > 50:
                result.append(value))
print(result)
```

```
[60, 70, 80, 90]
```

4. 일반적인 리스트 연산들

4) 파일을 읽어서 리스트에 저장하기

- 파일에서 데이터를 읽어서 리스트에 저장하는 작업은 아주 많이 등장한다. 기본적인 방법을 알아두자.

```
data = []
f = open("C:₩₩temp₩₩data.txt", "r")
for line in f.readlines():      # 파일에 저장된 모든 줄을 읽는다.
        data.append(line.strip()) # 줄바꿈 문자를 삭제한 후에 리스트에 추가한다.
print(data)
```

5) 정렬하기

- 정렬이란 리스트 안에 저장된 값을 순서에 따라서 나열하는 것이다. 리스트에서 정렬은 sort() 메소드를 이용하여 쉽게 수행할 수 있다.

```
a = [ 3, 2, 1, 5, 4 ]
a.sort()
print(a)
```

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

4. 일반적인 리스트 연산들

5) 정렬하기

- 하지만 경우에 따라서는 우리가 정렬을 구현해야 하는 경우도 있다. 따라서 기본적인 방법은 알아두자. 선택 정렬은 가장 이해하기가 쉬운 정렬 방법이다. 먼저 두 개의 리스트가 있다고 가정하자. 왼쪽에는 정렬된 리스트가 있고 오른쪽에는 정렬이 되지 않은 리스트가 있다. 초기 상태에서 정렬되어야 할 숫자들은 모두 정렬되지 않은 오른쪽 리스트에 들어 있다. 선택 정렬은 오른쪽 리스트를 탐색하여 가장 작은 숫자를 찾고 이 숫자를 왼쪽 리스트로 옮긴다. 다음에 다시 오른쪽 리스트에 남아있는 숫자 중에서 다시 가장 작은 숫자를 찾아서 왼쪽 리스트로 옮긴다. 이 과정을 오른쪽 리스트가 공백 상태가 될 때까지 계속한다.

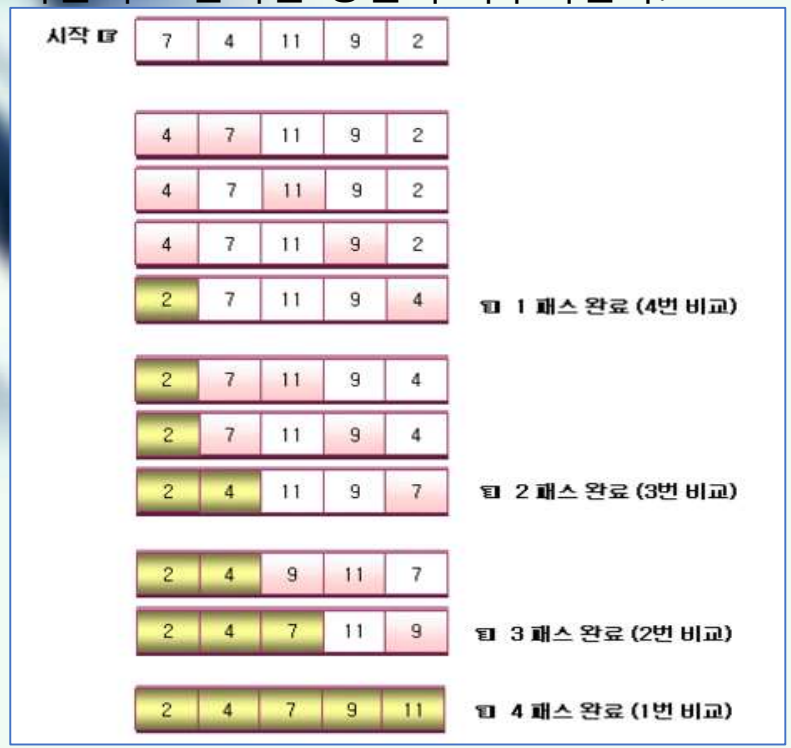
왼쪽 리스트	오른쪽 리스트	설명
( )	(5,3,8,1,2,7)	초기상태
(1)	(5,3,8,2,7)	1선택
(1,2)	(5,3,8,7)	2선택
(1,2,3)	(5,8,7)	3선택
(1,2,3,5)	(8,7)	5선택
(1,2,3,5,7)	(8)	7선택
(1,2,3,5,7,8)	()	8선택

- 다음 슬라이드에서 계속 진행하겠다.

4. 일반적인 리스트 연산들

5) 정렬하기

- 앞의 슬라이드에 표의 방법을 구현하기 위해서는 메모리를 절약하기 위하여 입력 리스트 외에 추가적인 공간을 사용하지 않는 선택 정렬 알고리즘을 생각해 보자. 이렇게 입력 리스트 이외에는 다른 추가 메모리를 요구하지 않는 정렬 방법을 제자리 정렬(in-place sort)이라고 한다. 아래 그림에서 보면 하나의 값을 변수에 담고 리스트의 루핑을 돌면서 계속 비교를 한다. 비교를 하다 보면 최소값이 바뀔 수도 있고 처음 값이 최소값이 될 수도 있다.리스트에서 최소값이 발견되면 0번째 인덱스와 최소값의 인덱스를 가지고 서로 교환을 한다. 이런 작업을 리스트의 끝까지 하게 되면 비로소 전체 숫자들이 오름차순 정렬이 이루어진다.




```
def selectionSort(aList):
        l, least, leastValue = 0
        for i in range(len(aList)):                 # 리스트의 모든 요소에 대하여 반복
                least = i                           # i요소를 최소값이라고 가정
                leastValue = aList[i]
                for k in range(i+1, len(aList )):
                        if aList[k] < leastValue:   #k번째 요소가 현재의 최소값보다 작으면..
                                least = k                   # k번째 요소를 최소값으로 한다.
                                leastValue = aList[k]
                tmp = aList[i]                              #i번째 요소와 최소값을 교환한다.
                aList[i] = aList[least]
                aList[least] = tmp
list1 = [ 7, 9, 5, 1, 8 ]
selectionSort(list1)
print(list1)
```

선택정렬 말고도 버블정렬, 삽입정렬, 병합정렬, 퀵정렬, 힙정렬이 존재한다.

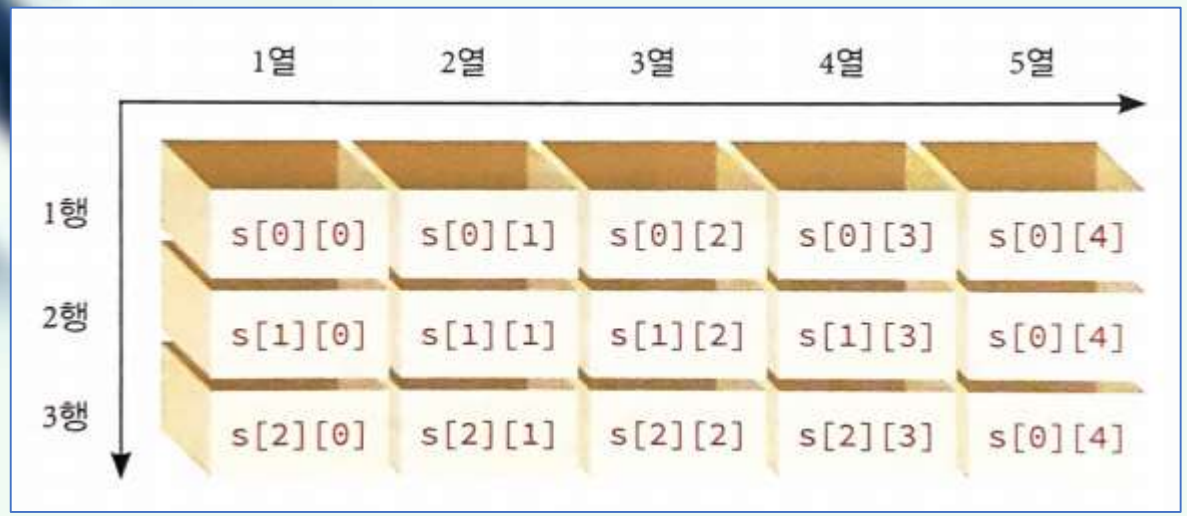
5. 2차원 리스트란?

1) 2차원 리스트

- 2차원 테이블을 이용하여 많은 일들을 처리한다. 예를 들어서 학생들의 과목별 성적도 2차원 형태로 나타낼 수 있다.

학생	국어	영어	수학	과학	사회
김철수	1	2	3	4	5
김영희	6	7	8	9	10
최자영	11	12	13	14	15

- 파이썬에서는 리스트를 2차원으로 만들 수 있다. 행렬과 같은 개념으로 이해를 하면 편리할 것이다.



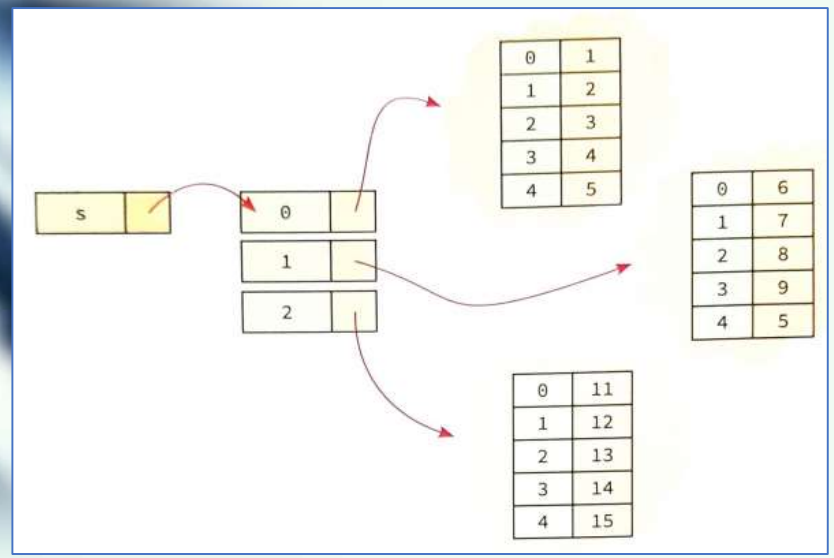
5. 2차원 리스트란?

2) 2차원 리스트 예제와 연결 구조

- 2차원 리스트의 예제는 아래와 같다.

```
s = [ [ 1, 2, 3, 4, 5], [ 6, 7, 8, 9, 10 ], [11, 12, 13, 14, 15 ] ]
print(s)
```

```
[ [1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15] ]
```



- 위의 코드에서 리스트 [1, 2, 3, 4, 5]가 첫 번째 행을 나타내고 리스트 [6, 7, 8, 9, 10]이 두 번째 행을 나타낸다.
  파이썬 튜터를 이용하여 그림의 개념을 확인해보자.

5. 2차원 리스트란?

3) 2차원 리스트 동적 생성

- 위의 2차원 리스트는 초기값이 미리 결정되어 있어서 정적으로 생성되었다. 실제로는 동적으로 2차원 리스트를 생성하는 경우가 더 많다. 리스트의 크기가 매우 큰 경우에도 동적으로 생성하여야 한다. 많은 방법이 있다. 가장 많이 사용되는 방법부터 살펴보자.

```
# 동적으로 2차원 리스트를 생성한다.
rows = 3
cols = 5
s = []

for row in range(rows):
        s += [ [0]*cols ]
print("s =", s)
```

```
s = [ [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0] ]
```

- 위의 코드는 리스트 함축을 사용하여도 된다.

```
rows = 3
cols = 5
s = [ ([0] * cols) for row in range(rows) ]      # 리스트 함축
print("s =", s)
```

5. 2차원 리스트란?

4) 2차원 리스트 요소 접근

- 2차원 리스트에서 요소에 접근하려면 2개의 인덱스 번호를 지정하여야 한다. 첫 번째 번호가 행 번호이고 두 번째 번호가 열 번호가 된다. 예를 들어서 2차원 리스트 s에서 2번째 행의 1번째 열에 있는 요소는 s[2][1]가 된다.

```
score = s[2][1]
```

- 2차원 리스트에 저장된 모든 값을 출력하려면 아래와 같이 더블 루프를 사용하여야 한다.

```
s = [ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10 ], [11, 12, 13, 14, 15] ]
# 행과 열의 개수를 구한다.
rows = len(s)
cols = len(s[0])
for r in range (rows):
        for c in range(cols):
                print(s[r][c], end="₩t")
        print()
```

```
1    2    3    4    5
6    7    8    9    10
11   12   13   14   15
```

-여기서 len(s)는 행의 개수이고 len(s[0])은 첫 번째 행에 들어 있는 열의 개수이다. 리스트 안에 다른 리스트를 내장하는 것도 가능하다. 이것은 실제 프로그래밍에서 많이 사용된다.

```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]      #리스트 x 안에 리스트 a와 n이 들어 있다.
```

6. 2차원 리스트 연산

1) 행 합계 계산하기

- 아마 2차원 리스트에서 가장 기본적인 연산은 행의 합계나 열의 합계일 것이다. 엑셀에서의 작업을 떠올리면 될 것이다. 0번째 행의 합계를 계산해보면 다음과 같다.

```
s = [ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10 ], [11, 12, 13, 14, 15] ]
cols = len(s[0])
sum = 0

for c in range(cols):          # 0번째 행의 합을 계산한다.
        sum = sum + s[0][c]
print(sum)
```

```
15
```

2) 요소 접근하기

- 2차원 리스트는 이미지를 처리할 때도 사용된다. 이미지도 본질적으로 각 화소(픽셀)의 값을 2차원적으로 나열한 것이기 때문이다. 하나의 화소에서 이웃 화소들의 인덱스를 찾아보자. 즉 화소의 위치가 (r, c)라고 했을 때, 상하좌우에 있는 화소들의 인덱스는 얼마가 될까?

(r−1, c−1)	(r−1, c)	(r−1, c+1)
(r, c−1)	(r, c)	(r, c+1)
(r+1, c−1)	(r+1, c)	(r+1, c+1)

6. 2차원 리스트 연산

3) 2차원 리스트와 함수

- 2차원 리스트도 객체이므로 함수로 전달할 수 있다. 만약 함수에서 리스트를 변경하면 원본 리스트가 변경된다. 하나의 예로 다음과 같이 1과 0이 반복되는 체커보드 형태의 10×10 크기의 2차원 리스트를 초기화하는 함수 init()를 작성하고 테스트 해보자.

```
table = []
# 2차원 리스트를 화면에 출력한다.
def printList(mylist):
        for row in range(len(mylist)):
                for col in range(len(mylist[0])):
                        print(mylist[row][col], end=" ")
        print()
# 2차원 리스트를 체커보드 형태로 초기화한다.
def init(mylist):
        for row in range(len(mylist)):
                for col in range(len(mylist[0])):
                        if (row+col)%2 == 0:     # (row+col)이 짝수이면 1을 저장
                                table[row][col] =1
if __name__ == "__main__":
        for row in range(10):    # 0으로 초기화된 2차원 리스트를 생성한다.
                table += [ [0] * 10]
        init(table)
        printList(table)
```

출력 결과

```
1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1
```


감사합니다.