05. 리스트

5.1 리스트의 연산

리스트는 순서를 가지는 객체들의 집합으로 파이썬 자료형들 중에서 가장 유용하게 활용된다. 리스트는 시퀀스 자료형이면서 변경 가능 자료형이다. 따라서 시퀀스 자료형의 특성(인덱싱, 슬라이싱, 연결, 반복, 멤버 검사 등)들을 지원하며, 데이터의 크기를 동적으로 그리고 임의로 조절하거나, 내용을 치환하여 변경할수 있다. 리스트는 대괄로 []로 표현한다.

```
>>> a = []              # 빈 리스트
>>> a = [1, 2, "Great"]
>>> print(a[0], a[-1])  # 인덱싱
1 Great
>>> print(a[1:3], a[:]) # 슬라이싱
[2, 'Great'] [1, 2, 'Great']
>>> L = list(range(10))
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[::2]      # 확장 슬라이싱
[0, 2, 4, 6, 8]
>>> a * 2       # 반복하기
[1, 2, 'Great', 1, 2, 'Great']
>>> a + [3, 4, 5]       # 연결하기
[1, 2, 'Great', 3, 4, 5]
>>> len(a)      # 리스트의 길이 정보
3
>>> 4 in L      # 멤버 검사
True
```

다음은 리스트의 일부 값을 변경하는 예이다.

```
>>> a = ['spam','egg',100, 1234] ]]]
>>> a[2] = a[2] + 23
>>> a
['spam', 'egg', 123, 1234]
```

다음은 리스트의 일부 값을 치환하는 예이다.

```
>>> a = ['spam', 'eggs', 123, 1234] ]]]
>>> a[0:2] = [1, 12]    #  항목 두 개를 교체한다.
>>> a
[1, 12, 123, 1234]
>>> a[0:2] = [1]        # 크기가 달라도 된다.
>>> a
```

```
[1, 123, 1234]
```

다음은 리스트의 일부 값을 삭제하는 예이다.

```
>>> a = [1, 12, 123, 1234]
>>> a[0:2] = []# 항목 두 개를 삭제한다.
>>> a
[123, 1234]
```

다음은 리스트에서 일부 값을 추가하는 예이다.

```
>>> a = [123, 1234]
>>> a[1:1] = ['spam','ham']            # 항목을 추가한다.
>>> a
[123, 'spam', 'ham', 1234]
```

확장 슬라이싱이 오른쪽에 오는 경우 오른쪽과 왼쪽 요소의 개수가 맞아야 한다.

```
>>> a = list(range(4))
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
>>> a[::2] = list(range(-10, -12, -1))  # 왼쪽 두 개 오른쪽 두 개다.
>>> a
[-10, 1, -11, 3]
>>> a [::2] = range(3)  # 오른쪽과 왼쪽 요소의 개수가 맞지 않는다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 3 to extended slice of size 2
```

다음은 del 문을 이용해서 값을 삭제하는 예이다.

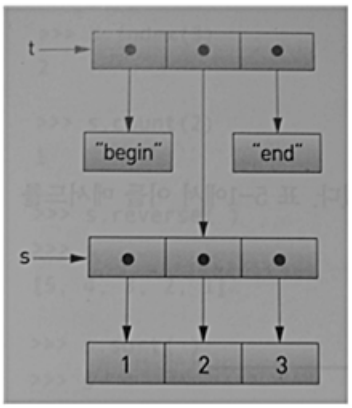
```
>>> a = [1, 2, 3, 4]
>>> del a[0]    # 값을 삭제한다.
>>> a
[2, 3, 4]
>>> del a[1:]   # 일부를 삭제한다.
>>> a
[2]
```

5.2 중첩 리스트

리스트 안에 또 다른 리스트가 포함되어 있는 경우 중첩 리스트라고 한다.

```
>>> s = [1, 2, 3]
>>> t = ['begin', s, 'end']   # 중첩 리스트
>>> t
['begin', [1, 2, 3], 'end']
>>> t[1][1]
2
```

리스트는 다른 객체를 직접 저장하지 않고, 객체들의 참조만을 저장한다. 여기서 참조란 객체의 주소를 말한다.



s에 의해서 참조되고 있는 리스트는 t[1]로부터도 참조되고 있다. 따라서 s[1]의 내용을 고친다면, t를 통해서도 변경된 내용을 보게 된다.

```
>>> s[1] = 100
>>> t
['begin', [1, 100, 3], 'end']
```

좀더 복잡한 예를 하나 더 다루어 보자. 다음 리스트는 삼중의 리스트이다.

```
>>> L = [1, ['a',['x','y'],'b'],3]
>>> L[0]
1
>>> L[1]
['a', ['x', 'y'], 'b']
>>> L[1][1]
['x', 'y']
>>> L[1][1][1]
'y'
```

5.3 리스트 메서드

- 리스트 메서드

리스트 객체는 내장 함수로 유용한 메서드를 여러개 가지고 있다.

메서드	설명
append	데이터를 리스트 끝에 추가(혹은 스택의 Push)한다.
insert	데이터를 지정한 위치에 삽입한다.
index	요소를 검색(Search)한다.
count	요소의 개수를 알아낸다.
sort	리스트를 정렬한다.
reverse	리스트의 순서를 바꾼다.
remove	리스트의 지정한 값 하나를 삭제한다.
pop	리스트의 지정한 값 하나를 읽어 내고 삭제(스택의 Pop)한다.
extend	리스트를 추가한다.

다음은 리스트 메서드를 사용한 예이다.

```
>>> s = [1, 2, 3]
>>> s.append(5)          # 리스트 마지막에 추가한다.
>>> s
[1, 2, 3, 5]
>>> s.insert(3, 4)       # 3위치에 4를 삽입한다.
>>> s
[1, 2, 3, 4, 5]
>>> s.index(3)           # 값 3의 위치는?
2
>>> s.count(2)           # 값2의 개수는?
1
>>> s.reverse()          # 리스트이 순서를 뒤집는다. 반환 값이 없다.
>>> s
[5, 4, 3, 2, 1]
>>> s.sort()             # 리스트를 정렬한다. 반환 값이 없다.
>>> s
[1, 2, 3, 4, 5]
>>> s = [10, 20, 30, 40, 50]
>>> s.remove(10)         # 값 10을 삭제한다. 여러 개이면 처음 것만 삭ㅈ한다.
>>> s
[20, 30, 40, 50]
>>> s.extend([60, 70])   # 리스트를 추가한다.
>>> s
[20, 30, 40, 50, 60, 70]
```

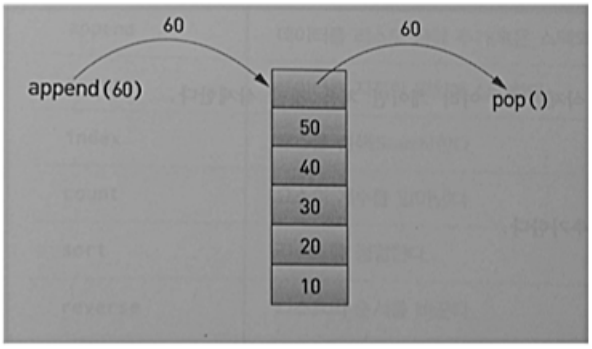
- 리스트를 스택으로 사용하기

스택이란 나중에 넣은 데이터를 먼저 꺼내도록 되어 있는 메모리 구조를 말한다. 혹시 택시를 탔을 때 동전을 끼워넣는 동전꽂이를보았는지 모르겠다. 동전은 위에서 끼워넣고 꺼낼때도 위에서부터 꺼낸다. 이와 같은 LIFO 형태의 메모리 구조를 스택이라고 한다. 따라서, 스택에 데이터를 넣을때는 어디에 넣으라고 지시하지 않는다. 데이터를 꺼낼때도 어디에서 꺼내라고 지시하지 않는다. 즉, 위치 정보를 지정하지 않는다. 여기서 넣는 연산을 push라고, 꺼내는 연산을 pop이라고 한다.

리스트는 그 자체를 스택으로 사용할수 있게 설계되었다. 스택에서 push 연산은 append() 메서드를, pop 연산은 pop() 메서드를 사용한다.

```
>>> s = [10, 20, 30, 40, 50]
>>> s.append(60)        # ①
>>> s
[10, 20, 30, 40, 50, 60]
>>> s.pop()             # ②
60
>>> s
[10, 20, 30, 40, 50]
```

① 스택의 push 연산이며, 리스트의 마지막에 데이터를 추가한다.
② 스택의 pop 연산이며, 마지막 데이터를 반환하고, 그 데이터를 리스트에서 제거한다.



리스트는 다음과 같이 일반화된 스택 연산을 수행한다. 즉, 원하는 위치에서 데이터를 마음대로 꺼낼수도 있다.

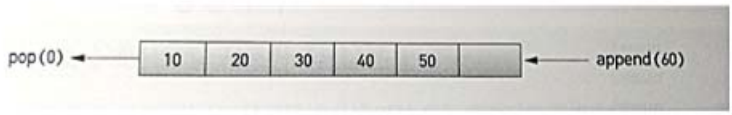
```
>>> s.pop(0)            # 맨 앞에 있는 데이터를 꺼낸다.
10
>>> s
[20, 30, 40, 50]
>>> s.pop(1)            # 두 번째 데이터를 꺼낸다.
30
>>> s
[20, 40, 50]
```

- 리스트를 큐로 사용하기

큐란 먼저 넣은 데이터를 먼저 꺼내도록 되어있는 메모리 구조를 말한다. 예를 들어, 은행에 갔을 때 대기표를 먼저 받은 사람이 먼저 서비스를 받는것과 같다. 이와 같은 FIFO 형태의 메모리 구조를 큐라고 한다.

큐에도 두 개의 연산이 필요하다. 데이터를 넣을 때 사용하는 연산으로 append() 메서드를 사용하며, 데이터를 꺼낼 때 사용하는 연산으로 pop(0) 메서드를 사용한다.

```
>>> s = [10, 20, 30, 40, 50]
>>> s.append(60)          # 큐의 맨 뒤에 값을 추가한다.
>>> s.pop(0)              # 큐의 맨 앞에서 값을 꺼낸다.
10
>>> s
[20, 30, 40, 50, 60]
```



5.4 리스트에 튜플이나 리스트가 있을 때 반복 참조하기

리스트에 저장된 데이터가 리스트나 튜플과 같이 복합 자료형인 경우 데이터를 다루는 예를 보자. 먼저 리스트의 데이터가 튜플이고, 이들을 출력한다면 다음과 같이 할수 있다.

```
>>> lt = [('one',1),('two',2),('three',3)]
>>> for t in lt:
...     print('name={:7}  num={}'.format(t[0],t[1]))
...
name=one      num=1
name=two      num=2
name=three    num=3
```

아니면 다음과 같이 사용할 수도 있다.

```
>>> lt = [('one',1),('two',2),('three',3)]
>>> for t in lt:
...     print('name={0[0]:7}num={0[1]}'.format(t))
...
name=one    num=1
name=two    num=2
name=three  num=3
```

하지만 튜플의 각각의 값을 for문에서 이용한다면 아예 시작 부분에서 다른 이름올 받는 것이 더 명확하다.

```
>>> for name, num in lt:
...     print('name={:7}num{}'.format(name, num))
...
name=one    num1
name=two    num2
name=three  num3
```

다음으로 리스트의 데이터가 리스트이고 이를 각각의 내부 값을 참조하는 것도 같은 방식으로 가능하다.

```
>>> LL = [ ['one',1], ['two',2], ['three',3] ]
>>> for name, num in LL:
...     print(name, num)
...
one 1
two 2
three 3
```

5.5 리스트 정렬하기

- sort() 메서드를 사용한 정렬

리스트의 sort() 메서드는 내장된 순서에 따라 모든 데이터를 정렬한다.

```
>>> L= [1, 5, 3, 9, 8, 4, 2]
>>> L.sort()
>>> L
[1, 2, 3, 4, 5, 8, 9]
```

만일 역순으로 정렬가기를 원하면 다음과 같이 sort() 메서드에 reverse 키워드를 사용한다.

```
>>> L.sort(reverse = True)
>>> L
[9, 8, 5, 4, 3, 2, 1]
```

다음예와 같이 문자열을 대소문자 구분없이 정렬하기를 원하면 key 키워드를 사용하여 정렬기준을 바꿀수 있다.

```
>>> L = 'Python is a Programming Language'.split()      # 문자열 리스트이다.
>>> L.sort()                                            # 일단은 그냥 정렬해 보자.
>>> L                                                   # 대문자가 소문자보다 앞에 나온다.
['Language', 'Programming', 'Python', 'a', 'is']
>>> L.sort(key = str.lower)                             # str.lower(' ABC' )이면 ' abc' 가 된다.
>>> L                                                   # 대소문자 무시하고 정렬한다.
['a', 'is', 'Language', 'Programming', 'Python']
>>> L.sort(key = str.lower, reverse = True)             # 역순으로 정렬이 된다.
>>> L
['Python', 'Programming', 'Language', 'is', 'a']
```

내부에서 정렬하려면 두 값의 크기를 비교해야 하는데, 비교할 값은 key 키워드로 전달된 함수를 적용한 결과값이 된다. 예를 들어, a와 Python 두 문자열을 비교할 때 실제로는 str.lower() 함수를 적용한 a와 pytho 두 문자열을 비교한다. 하지만, 두 값을 비교할 때마다 str.lower() 함수를 적용하는 것은 아니다. 초기에 모든 값을 str.lower() 함수에 적용하여 한번에 변환하고 이변환된 값들을 비교하기 때문에 정렬이 빠르다.

다음은 문자열로 표현된 숫자들을 숫자의 크기대로 정렬하는 예이다.

```
>>> L = ['123','34','56','2345'] ]]]]]]]
>>>
>>>
>>> L = ['123','34','56','2345']
>>> L.sort()                    # 문자열 비교 정렬을 하면 숫자 크기와는 관계없이 정렬된다.
```

```
>>> L
['123', '2345', '34', '56']
>>> L.sort(key = int)          # 각요소는 key 키워드로 전달된 함수를 통과한 값과 비교된다.
>>> L
['34', '56', '123', '2345']
```

다음예는 정렬 기준을 어떤수를 3으로 나누었을때의 나머지로 한다. 즉, 나머지가 0인 9가 나머지가 1인 4보다 앞에 나오도록 정렬한다.

```
>>> def mykey(a):
...     return a % 3
...
>>> L = [1, 5, 3, 9, 8, 4, 2]
>>> L.sort(key = mykey)
>>> L
[3, 9, 1, 4, 5, 8, 2]
```

앞의 예에서 key 키워드로 mykey() 함수를 전달했다. 이 함수는 어떤 수를 3으로 나눈 나머지를 반환한다. 모든 숫자는 mykey() 함수를 적용한 결괏값에 따라서 비교하고 정렬한다. 만일예에서 같은 그룹내의 숫자들(5, 8, 2)이 크기에 따라 정렬되기를 원하면 mykey() 함수를 다음과 같이 약간 수정한다.

```
>>> def mykey(a):
...     return (a % 3, a)       # 튜플로 넘긴다. 같은 나머지라면 숫자의 크기로 비교한다.
...
>>> L = [1, 5, 3, 9, 8, 4, 2]
>>> L.sort(key = mykey)
>>> L
[3, 9, 1, 4, 2, 5, 8]
```

lambda 함수는 한 줄로 함수 객체를 생성한다.

```
>>> f = lambda 입력: 입력*2     # 출력
>>> f(2)
4
```

lambda 함수를 사용하여 별도로 함수를 정의하지 않고 sort() 메서드를 호출할수도 있다.

```
>>> L = [1, 5, 3, 9, 8, 4, 2]
>>> L.sort(key = lambda a: (a % 3, a))          # 직접 함수를 튜플을 반환한다.
>>> L
[3, 9, 1, 4, 2, 5, 8]
```

여러개의 데이터가 튜플과 같은 자료형에 있을 경우, 첫 번째 값이 아닌 다른 값을 기준으로 정렬할수도 있다. 다음은 (이름, 경력, 나이)로 구성된 튜플의 리스트를 정렬하는 예이다. 나이를 기준으로 정렬하고 있다.

```
>>> L = [ ('lee',5, 38), ('kim', 3, 28), ('jung',10, 36)]
>>> L.sort(key = lambda a: a[2])
>>> L
[('kim', 3, 28), ('jung', 10, 36), ('lee', 5, 38)]
```

- sorted() 함수를 사용한 정렬

만일 리스트 내부의 순서는 변경하지 않고 그대로 두고 새로 정렬된 리스트만 원할 때 sorted() 내장 함수를 사용할수 있다. 리스트 정렬에 있어서 초보자가 흔히 범하는 실수는 다음과 같이 정렬된 결과를 반환 값으로 얻으려 하는 것이다.

```
>>> L = [1, 6, 3, 8, 6, 2, 9]
>>> newList = L.sort()                  #  내부 정렬을 수행한다. 반환 값은 없다.
>>> print(newList)
None
```

sorted() 내장함수는 다음과 같이 새로 정렬된 리스트를 반환 값으로 돌려준다.

```
>>> L= [1, 6, 3, 8, 6, 2, 9]
>>> newList = sorted(L)         # 리스트 L은 변경되지 않고, 새로운 리스트가 반환된다.
>>> newList
[1, 2, 3, 6, 6, 8, 9]
```

따라서 다음과 같이 for 문에 직접 사용하는 것이 가능하다.

```
>>> for ele in sorted(L):
...     print(ele, end = '')
...
1236689
```

sorted() 내장함수는 리스트의 sort() 메서드와 같이 key와 reverse 키워드를 동일하게 지원한다.

```
>>> L = [1, 6, 3, 8, 6, 2, 9]
>>> sorted(L, reverse = True)
[9, 8, 6, 6, 3, 2, 1]
>>> L = ['123','34','56','2345']
>>> sorted(L, key = int)
['34', '56', '123', '2345']
```

sorted()내장함수는 리스트뿐 아니라 사전에도 사용할수 있다.

```
>>> d = {'one':1, 'two':2, 'three':3,'four':4}
>>> sorted(d)
['four', 'one', 'three', 'two']
>>> for key in sorted(d):
...     print(key)
...
four
one
three
two
```

만일 값을 중심으로 정렬하려면 다음과 같이 할수 있다.

```
>>> d = {'one':1, 'two':2, 'three':3, 'four':4}
>>> for key in sorted(d, key = lambda a:d[a]):
...     print(key)
...
one
two
three
four
```

다음과 같은 코딩도 가능하다.

```
>>> sorted(d.items(), key = lambda a:a[1])
[('one', 1), ('two', 2), ('three', 3), ('four', 4)]
```

- reversed() 함수를 사용한 정렬
매우 큰 리스트를 역순으로 참조하려 한다면 다음과 같이 할수 있을 것이다.

```
>>> L
['123', '34', '56', '2345']
>>> L.reverse()         # 순서 뒤집기
>>> L
['2345', '56', '34', '123']
>>> L.reverse()         # 다시 원래 상태로
```

하지만, 리스트의 순서를 직접 변경시키는 이와 같은 처리 방식은 매우 비효율적이다. 시퀀스 자료형에서 직접 역순으로 참조하는 내장 함수 reversed() 가 있다.

```
>>> L
['123', '34', '56', '2345']
>>> for ele in reversed(L):
...     print(ele)
...
2345
56
34
123
```

reversed() 내장함수가 반환하는 객체는 반복자로서 리스트를 재구성하거나 복사하지 않기 때문에 메모리나 처리 시간의 낭비가 없다.

5.6 리스트 내장

- 리스트 내장

리스트 내장은 시퀀스 자료형으로부터 리스트를 쉽게 만드는 데 유용하다. 리스트 내장을 이용하여 0부터 9까지 수의 제곱의 리스트를 만들어 보자.

```
>>> L = [k * k for  k in range(10)]
>>> print(L)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

for 문에 의해 리스트로 만들어진 결과를 출력 수식이 반환한다.

```
[k * k for  k in range(10)]
출력서식          입력시퀀시
```

앞의 코드는 다음과 같이 for 문을 이용하여 리스트를 만드는 것과 같다.

```
>>> L = []
>>> for k in range(10):
...     L.append(k * k)
...
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

리스트 내장은 다음의 형식을 취한다.

```
[<식> for <타깃1> in <객체1>
        for <타깃2> in <객체2>
        ...
        for <타깃N> in <객체N>
        (if <조건식>) ]
```

for ~ in 절은 <객체>의 항목을 반복한다. 여기서 <객체>는 시퀀스형 데이터여야 한다. for 문으로 취해지는 각각의 값은 <식>에서 사용한다. 마지막 if절은 선택 요소이다. 만일 if절이 있으면, <식>은 <조건식> 이 참일때만 값을 계산하고 결과에 추가한다.

앞서 설명한 리스트 내장은 다음 파이썬 코드와 동등하다.

```
for <타깃1> in <객체1>:
        for <타깃2> in <객체2>:
          ...
          for <타깃N> in <객체N>:
          if <조건식>:
               # 식의 값을 결과 리스트에 추가한다.
```

다음 예는 10보다 작은 정수 중 홀수의 제곱만을 반환한다.

```
>>> L = [k * k for k in range(10) if k % 2 == 1]
>>> L
[1, 9, 25, 49, 81]
```

다음예는 2의 배수와 3의 배수 중 두수의 합이 7의 배수가 되는 두 수와 곱의 리스트를 만들어 낸다.

```
>>> [(i, j, i * j) for i in range(2, 100, 2)
...     for j in range(3, 100, 3)
...             if (i + j) % 7 == 0]
[(2, 12, 24), (2, 33, 66), (2, 54, 108), (2, 75, 150), (2, 96, 192), (4, 3, 12), (4, 24, 96), (4, 45,
180), (4, 66, 264), (4, 87, 348), (6, 15, 90), (6, 36, 216), (6, 57, 342), (6, 78, 468), (6, 99, 594),
(8, 6, 48), (8, 27, 216), (8, 48, 384), (8, 69, 552), (8, 90, 720), (10, 18, 180), (10, 39, 390), (10,
60, 600), (10, 81, 810), (12, 9, 108), (12, 30, 360), …생략…
```

다음예는 두 시퀀스 자료형의 모든 데이터의 조합을 만들어 낸다.

```
>>> seq1 = 'abc'
>>> seq2 = (1, 2, 3)
>>> [(x, y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3), ('b', 1), ('b', 2), ('b', 3), ('c', 1), ('c', 2), ('c', 3)]
```

출력 수식 부분이 큐플이라면, 소괄호()로 감싸야 한다. 다음은 문법상 에러이다
[x, y for x in seq1 for y in seq2]
다음은 문법상 올바르다.

- 중첩 리스트 내장
리스트 내장은 또 다른 리스트 내장에 포함될수 있다. 즉, 중첩될수 있다.

```
>>> [[row + (i * 3) for row in [10, 11, 12]] for i in [0, 1, 2]]
[[10, 11, 12], [13, 14, 15], [16, 17, 18]]
```

다음 코드는 단위 행렬 형태의 리스트를 만드는 코드이다.

```
>>> [[1 if  col_idx == row_idx else 0 for col_idx in range(0, 3)]
...     for row_idx in range(0, 3)]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

- 발생자 내장
리스트 내장의 대괄호[]를 소괄호 ()로 바꾸면 리스트가 아닌 발생자 객체가 생성된다.

```
>>> (k * k for k in range(10))
<generator object <genexpr> at 0x000001C09B83B0A0>
```

발생자는 반복자의 일종으로 데이터를 순차적으로 제공할 준비가 되어 있는 객체이다. 즉, 아직 실제로 제곱의 계산이 이루어지지 않은 상태이며, 데이터를 필요할 때 하나씩 제곱을 해서 넘겨준다.
따가서 새로운 리스트를 만들지 안흥면서 다른 연산에 사용할 수 있어서 메모리 사용량이나 연산량에서 발생자 내장이 리스트 내장보다 효과적일 수 있다.

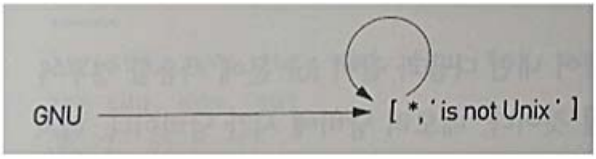
```
>>> sum([x * x for x in range(10)])            # 리스트를 만들고 sum()을 계산한다.
285
>>> sum(k * k for k in range(10))              # 리스트를 만들지 않고 sum()을 계산한다.
285
>>> list(k * k for k in range(10))             # 리스트로 변환한다.
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

5.7 순환 참조 리스트

순환 참조란 어떤 객체가 자기 자신을 직접 혹은 간접적으로 참조하는 경우를 말한다. 어떠한 복합 객체도 이러한 구조를 가질수 있지만, 리스트로 쉽게 표현할수 있으므로 이절에서 설명한다. 순환 참조의 예를 보자. 다음은 GNU의 뜻인 'GNU의 뜻인 ' GNU is not Unix' 란 재귀적인 구문을 표현한 것이다.

```
>>> GNU = ['is not Unix']
>>> GNU.insert(0, GNU)
>>> GNU
[[...], 'is not Unix']
>>> GNU[0][0]
[[...], 'is not Unix']
>>> GNU[0][0][0]
[[...], 'is not Unix']
```

예에서 파이썬은 GNU 객체를 무한히 출력하기보다는 [...]으로 대신한다.



이러한 순환 참조는 조심스럽게 사용해야 한다. 순환 참조는 쓰레기 수집을 방해하는 주요 요인이다. 쓰레기 수집은 어떤 객체의 참조 횟수가 0이 될 때 메모리에서 자동으로 객체를 제거하는 기능으로, 순환 참조의 참조 횟수는 0이 될수 없기 때문에 객체의 제거가 불가능하다.

만일 순환 참조를 사용해야 한다면 약한 참조를 이용할수 있다. 약한 참조란 참조횟수에 포함되지 않는 참조이다. 즉, 쓰레기 수집 문제를 방해하지 않는 참조이다. 즉, 쓰레기 수집 문제를 방해하지 않는 참조이다. 약한 참조에 대해서는 18장에서 별도로 다루고 있다.

5.8 순차적인 정수 리스트 만들기

이미 앞서 살펴보았지만 순차적인 정수 리스트를 만들때는 손으로 직접 입력하기보다는 range() 함수를 사용하면 편리하다. 이 함수는 자주 for 문과 같이 사용한다. range() 함수는 한 개에서 세 개까지의 인수를 받을수 있다. range(x)와 같이 인수가 주어져 있을 경우, 0부터 x보다 작은 수까지의 리스트를 1 간격으로 만들어 내며, range(x, y)인 경우 x부터 y보다 작은수까지의 리스트를 1간격으로 만들며, range(x, y, s)인 경우 range(x, y) 와 같지만 간격을 s로 한다. 파이썬 3에서는 range() 함수가 리스트를 직접 만들어 내지 않고 range 객체를 만들어 낸다.

```
>>> range(10)
range(0, 10)
>>> type(range(10))
<class 'range'>
```

이유는 만일 range() 함수가 리스트를 만들어 내면 다음과 같이 for문에 사용할 경우에 10만개 크기의 리스트가 만들어지고 버려지게 될 것이다. 메모리 낭비에 시간 낭비이다. 하지만, 파이썬 3에서는 이런 일이 일어나지 않는다. range 객체는 데이터를 순차적으로 넘겨줄 준비만 하고 있다가 데이터가 요구되는 경우에 하나씩 넘겨주는 반복자로 동작한다.

```
>>> for k in range(100000):
...     a = k * k
...     if a > 1000000:
...             break
...
>>> k
1001
```

리스트나 튜플을 얻으려면 명시적인 형변환을 요구할수 있다.

```
>>> list(range(10))             # 0부터 10보다 작은 수의 정수 리스트
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> list(range(5, 15))          # 5부터
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range(5, 15, 2))       # 5부터 2간격으로
[5, 7, 9, 11, 13]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

range객체는 순차적인 값을 할당하는 데도 사용할수 있다.

```
>>> sun, mon, tue, wed, thu, fir, sat = range(7)
```

```
>>> sun, mon, sat
(0, 1, 6)
```

5.9 지역적으로 사용 가능한 이름 리스트 얻기

인수없이 dir() 함수를 사용하면 현재 지역적으로 사용 가능한 심볼 테이블(사용 가능한 이름 목록)의 내용을 알수 있다.

```
>>> dir()
['GNU', 'L', 'LL', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'd', 'ele', 'f', 'fir', 'k', 'key', 'lt', 'mon', 'mykey', 'name',
'newList', 'num', 's', 'sat', 'se12', 'seq1', 'seq2', 'sun', 't', 'thu', 'tue', 'wed']
>>> a = 100
>>> dir()
['GNU', 'L', 'LL', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'd', 'ele', 'f', 'fir', 'k', 'key', 'lt', 'mon', 'mykey', 'name',
'newList', 'num', 's', 'sat', 'se12', 'seq1', 'seq2', 'sun', 't', 'thu', 'tue', 'wed']
>>> _name_
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_name_' is not defined
>>> __name__
'__main__'
>>> __doc__
>>> __builtins__
<module 'builtins' (built-in)>
```

dir() 함수의 인수에 임의의 객체를 전달하면, 해당 객체에서 사용할수 있는 속성(함수, 변수등의 이름들)의 리스트를 반환한다. 모듈에 어떤 함수나 변수가 정의되어 있는지 클래스에 어떤 메서드를 사용할수 있는지 등의 정보를 간단히 얻을수 있다.

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__', '__name__',
'__package__', '__spec__', '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
'_current_frames', '_debugmallocstats', '_enablelegacywindowsfsencoding', '_getframe', '_git',
'_home', '_xoptions', 'api_version', 'argv', 'base_exec_prefix', 'base_prefix',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook',
'dllhandle', 'dont_write_bytecode', 'exc_info', …생략…
```