

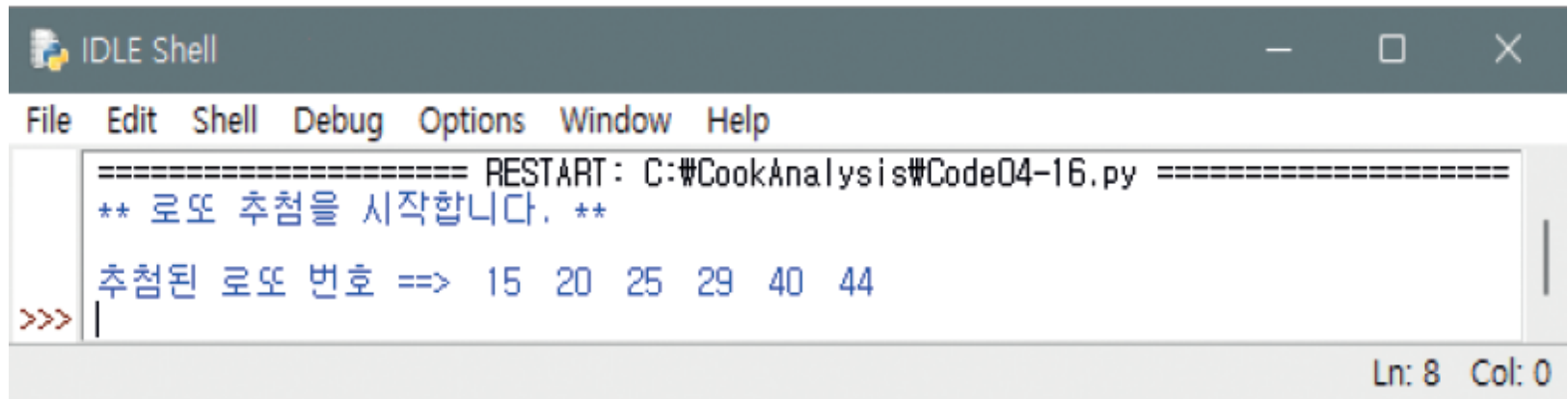
학습 목표

- 리스트 개념을 학습하고 활용한다.
- 튜플 및 딕셔너리를 익힌다.
- 문자열과 문자열을 조작하는 함수를 익힌다.
- 함수의 개념을 익힌다.
- 함수의 작성법과 사용법을 익힌다.

Section 01 이 장에서 만들 프로그램

■ [프로그램] 로또 번호 추첨

- 로또 추첨처럼 1부터 45까지의 숫자 중에서 6개를 뽑는 프로그램



```
===== RESTART: C:\CookAnalysis\Code04-16.py =====  
** 로또 추첨을 시작합니다. **  
추첨된 로또 번호 ==> 15 20 25 29 40 44  
>>> |
```

Ln: 8 Col: 0

Section 02 리스트

■ 리스트의 개념과 필요성

- 리스트는 [그림 4-1]과 같이 하나씩 사용하던 박스(변수)를 한 줄로 붙여 놓은 것
- 리스트는 박스(변수)를 한 줄로 붙인 후 전체에 이름(aa)을 지정
- 각각은 aa[0], aa[1], aa[2], aa[3]처럼 번호(첨자)를 붙여 사용

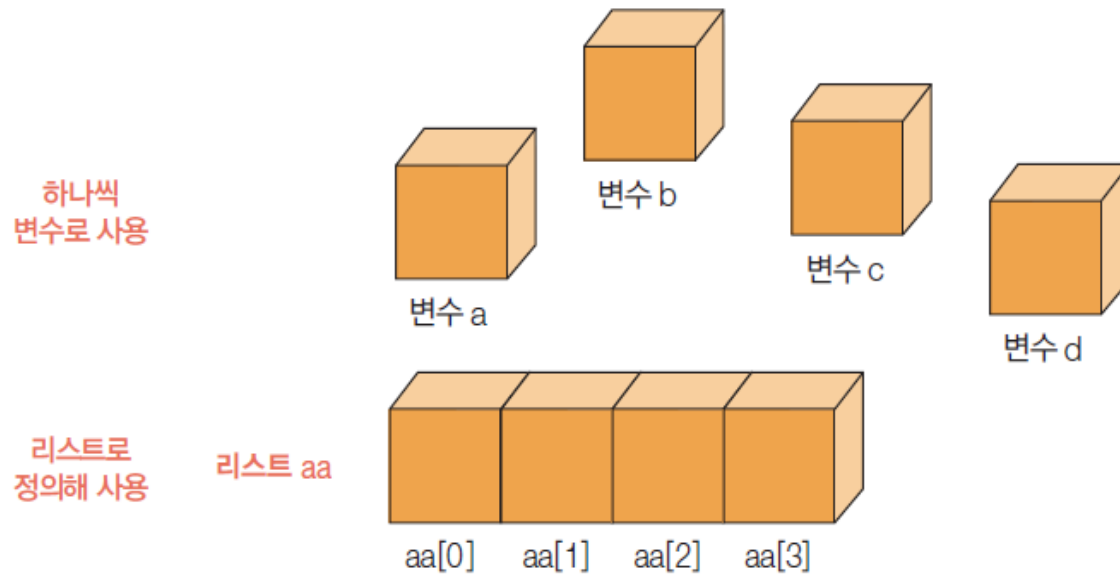


그림 4-1 리스트의 개념

Section 02 리스트

■ 리스트의 개념과 필요성

- 정수형 변수 4개를 선언한 후 이 변수에 값을 입력받고 합계를 출력하는 프로그램

Code04-01.py

```
01 a, b, c, d = 0, 0, 0, 0
02 hap = 0
03
04 a = int(input("1번째 숫자 : "))
05 b = int(input("2번째 숫자 : "))
06 c = int(input("3번째 숫자 : "))
07 d = int(input("4번째 숫자 : "))
08
09 hap = a + b + c + d
10
11 print("합계 ==> %d" % hap)
```

실행 결과

1번째 숫자 : 10
2번째 숫자 : 20
3번째 숫자 : 30
4번째 숫자 : 40
합계 ==> 100

Section 02 리스트

■ 리스트의 개념과 필요성

- 입력을 100개 이상과 같이 많이 받게 되면 변수를 선언하고 할당하는 것이 굉장히 힘들
- 이때 필요한 것이 바로 리스트

- 리스트 선언 방법 : 대괄호 안에 값을 선언함

```
리스트명 = [값1, 값2, 값3, ...]
```

- 다음은 값 4개를 담은 정수형 리스트를 생성함

```
aa = [10, 20, 30, 40]
```

Section 02 리스트

■ 리스트의 개념과 필요성

- 리스트는 인덱스를 사용해 ❷처럼 각 변수를 aa[0], aa[1], aa[2], aa[3]으로 사용
- 리스트의 첨자는 0부터 시작한다는 점을 반드시 유의

❶ 각 변수 사용

a, b, c, d = 10, 20, 30, 40

a 사용

b 사용

c 사용

d 사용

❷ 리스트 사용

aa = [10, 20, 30, 40]

aa[0] 사용

aa[1] 사용

aa[2] 사용

aa[3] 사용

Section 02 리스트

■ 리스트의 개념과 필요성

- 리스트는 인덱스를 사용해 ❷처럼 각 변수를 aa[0], aa[1], aa[2], aa[3]으로 사용
- 리스트의 인덱스는 0부터 시작한다는 점을 반드시 유의

Code04-02.py

```
01 aa = [0,0,0,0]
02 hap = 0
03
04 aa[0] = int(input("1번째 숫자 : "))
05 aa[1] = int(input("2번째 숫자 : "))
06 aa[2] = int(input("3번째 숫자 : "))
07 aa[3] = int(input("4번째 숫자 : "))
08
09 hap = aa[0] + aa[1] + aa[2] + aa[3]
10
11 print("합계 ==> %d" % hap)
```

항목 4개인 리스트 생성

리스트 사용

실행 결과

1번째 숫자 : 10
2번째 숫자 : 20
3번째 숫자 : 30
4번째 숫자 : 40
합계 ==> 100

Section 02 리스트

■ 리스트 활용

- 먼저 빈 리스트를 만들고 리스트명.append(값) 함수로 리스트에 항목을 하나씩 추가할 수 있음

```
aa = []  
aa.append(0)  
aa.append(0)  
aa.append(0)  
aa.append(0)  
aa
```

실행 결과

```
[0, 0, 0, 0]
```


Section 02 리스트

■ 리스트 활용

- 항목이 100개와 같이 많은 리스트를 작성할 경우, append() 함수와 함께 for 문을 활용하면 간단히 해결할 수 있음

```
aa = []  
for i in range(0, 100) :  
    aa.append(0)  
len(aa)
```

실행 결과

100

Section 02 리스트

■ 리스트 활용

- for 문을 이용해서 리스트의 인덱스가 순서대로 변할 수 있도록 할 수 있음

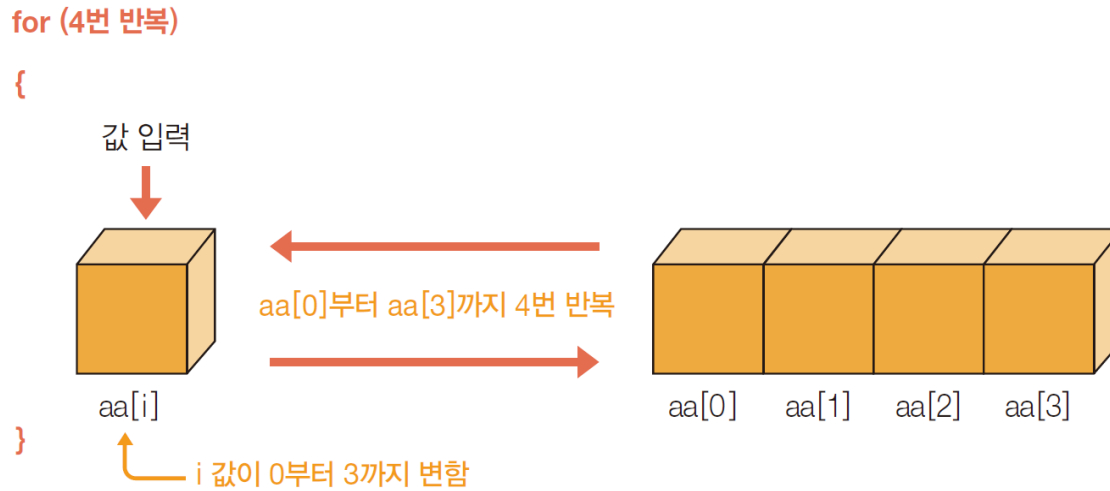


그림 4-2 for 문으로 리스트값 입력

Section 02 리스트

■ 리스트 활용

- for 문을 이용해서 리스트의 첨자가 순서대로 변할 수 있도록 할 수 있음

Code04-03.py

```
01 aa = []  
02 for i in range(0, 4) :  
03     aa.append(0)  
04 hap = 0  
05  
06 for i in range(0, 4) :  
07     aa[i] = int(input(str(i + 1) + "번째 숫자 : " ))  
08  
09 hap = aa[0] + aa[1] + aa[2] + aa[3]  
10 print("합계 ==> %d" % hap)  
11
```

The diagram shows three red arrows pointing from the code to callout boxes:

- From line 02 to line 03, a bracket and an arrow point to a box labeled "항목 4개인 리스트 생성" (Create a list with 4 items).
- From line 06 to line 07, an arrow points to a box labeled "4번 반복" (Repeat 4 times).
- From line 09, an arrow points to a box labeled "변수 4개를 더함" (Add 4 variables).

실행 결과

1번째 숫자 : 10
2번째 숫자 : 20
3번째 숫자 : 30
4번째 숫자 : 40
합계 ==> 100

SELF STUDY 4-1

값을 4개가 아닌 10개를 입력받아 합계를 출력하도록 Code04-03.py를 수정해 보자. 또 합계를 구하기 위해 for 문 대신 while 문을 사용해 보자.

Section 02 리스트

■ 리스트 값 접근

- 음수 값 인덱스를 이용한 리스트 접근
- 리스트에 접근할 때 콜론(:)을 사용해 범위를 지정할 수도 있음
- '리스트명[시작값:끝값+1]'은 리스트의 시작 위치부터 끝 위치까지 모든 값을 의미

```
aa = [10, 20, 30, 40]  
print("aa[-1]은 %d, aa[-2]는 %d" % (aa[-1], aa[-2]))
```

실행 결과

```
aa[-1]은 40, aa[-2]는 30
```

Section 02 리스트

■ 리스트 값 접근

- 콜론의 앞이나 뒤 숫자를 생략할 수 있음
- `aa[2:]`는 `aa[2]`부터 끝까지를 의미하며, `aa[:2]`는 처음부터 `aa[1]`까지를 의미
- `aa[2]`는 포함되지 않는 점에 주의

```
aa = [10, 20, 30, 40]  
aa[0:3]  
aa[2:4]
```

실행 결과

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

```
aa = [10, 20, 30, 40]  
aa[2:]  
aa[:2]
```

실행 결과

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

Section 02 리스트

■ 리스트 값 접근

- 리스트끼리 덧셈 및 곱셈 연산도 가능
- 리스트끼리 더하니 요소들이 합쳐져 결과로 리스트 하나가 되었음
- 리스트끼리 곱하니 항목들이 횟수만큼 반복해서 출력되었음

```
aa = [10, 20, 30]  
bb = [40, 50, 60]  
aa + bb  
aa * 3
```

실행 결과

```
[10, 20, 30, 40, 50, 60]  
[10, 20, 30, 10, 20, 30, 10, 20, 30]
```

Section 02 리스트

■ 리스트 값 변경

- 두 번째에 위치한 값을 변경하는 방법은 아래와 같음

```
aa = [10, 20, 30]  
aa[1] = 200  
aa
```

실행 결과

```
[10, 200, 30]
```

- 연속된 범위의 값을 변경하는 방법은 아래와 같음
- aa[1:2]는 리스트 aa의 첫 번째부터 다음 첫 번째($2-1=1$)를 의미
- 즉 두 번째인 aa[1]의 위치를 [200, 201]로 교체하라는 의미

```
aa = [10, 20, 30]  
aa[1:2] = [200, 201]  
aa
```

실행 결과

```
[10, 200, 201, 30]
```

Section 02 리스트

■ 리스트 값 변경

- aa[1:2] 대신 그냥 aa[1]을 사용하면 리스트 안에 또 다른 리스트가 추가되어 있음
- 결과가 틀린 것은 아니지만, 이렇게 사용하는 경우는 많지 않으니 주의할 필요가 있음

```
aa = [10, 20, 30]  
aa[1] = [200, 201]  
aa
```

실행 결과

```
[10, [200, 201], 30]
```

- 리스트의 항목을 삭제하려면 del() 함수를 사용

```
aa = [10, 20, 30]  
del(aa[1])  
aa
```

실행 결과

```
[10, 30]
```


Section 02 리스트

■ 리스트 값 변경

- 항목을 여러 개 삭제하려면 `aa[시작값:끝값+1]=[]`으로 설정
- 다음은 두 번째인 `aa[1]`에서 네 번째인 `aa[3]`까지 삭제함

```
aa = [10, 20, 30, 40, 50]
aa[1:4] = []
aa
```

실행 결과

```
[10, 50]
```

- 리스트 자체를 삭제하는 방법은 다음과 같이 다양함
- ❶은 리스트 내용을 모두 삭제해 빈 리스트로 만들고, ❷은 리스트에 `None`값을 넣어 `aa`를 빈 변수로 만들고, ❸은 `aa` 변수를 삭제함

```
❶ aa = [10, 20, 30]; aa = []; aa
❷ aa = [10, 20, 30]; aa = None; aa
❸ aa = [10, 20, 30]; del(aa); aa
```

실행 결과

```
[]
아무것도 안 나옴
오류 발생
```

Section 02 리스트

■ 리스트 조작 함수

■ 다양한 함수로 리스트를 조작할 수 있음

함수	설명	사용법
append()	리스트 맨 뒤에 항목을 추가한다.	리스트명.append(값)
pop()	리스트 맨 뒤의 항목을 빼낸다(리스트에서 해당 항목이 삭제된다).	리스트명.pop()
sort()	리스트의 항목을 정렬한다.	리스트명.sort()
reverse()	리스트 항목의 순서를 역순으로 만든다.	리스트명.reverse()
index()	지정한 값을 찾아 해당 위치를 반환한다.	리스트명.index(찾을값)
insert()	지정된 위치에 값을 삽입한다.	리스트명.insert(위치, 값)
remove()	리스트에서 지정한 값을 삭제한다. 단 지정한 값이 여러 개면 첫 번째 값만 지운다.	리스트명.remove(지울값)
extend()	리스트 뒤에 리스트를 추가한다. 리스트의 더하기(+) 연산과 기능이 동일하다.	리스트명.extend(추가할리스트)
count()	리스트에서 해당 값의 개수를 센다.	리스트명.count(찾을값)
clear()	리스트의 내용을 모두 지운다.	리스트명.clear()
del()	리스트에서 해당 위치의 항목을 삭제한다.	del(리스트명[위치])
len()	리스트에 포함된 전체 항목의 개수를 센다.	len(리스트명)
copy()	리스트의 내용을 새로운 리스트에 복사한다.	새리스트=리스트명.copy()
sorted()	리스트의 항목을 정렬해서 새로운 리스트에 대입한다.	새리스트=sorted(리스트)

Section 02 리스트

■ 리스트 조작 함수

Code04-04.py

```
01  myList = [30, 10, 20]
02  print("현재 리스트 : %s" % myList)
03
04  myList.append(40)
05  print("append(40) 후의 리스트 : %s" % myList)
06
07  print("pop()으로 추출한 값 : %s" % myList.pop())
08  print("pop() 후의 리스트 : %s" % myList)
09
10  myList.sort()
11  print("sort() 후의 리스트 : %s" % myList)
12
13  myList.reverse()
14  print("reverse() 후의 리스트 : %s" % myList)
15
16  print("20값의 위치 : %d" % myList.index(20))
17  myList.insert(2, 222)
18
19  print("insert(2, 222) 후의 리스트 : %s" % myList)
```

Section 02 리스트

■ 리스트 조작 함수

Code04-04.py

```
20
21 myList.remove(222)
22 print("remove(222) 후의 리스트 : %s" % myList)
23
24 myList.extend([77, 88, 77])
25 print("extend([77, 88, 77]) 후의 리스트 : %s" % myList)
26
27 print("77값의 개수 : %d" % myList.count(77))
```

실행 결과

현재 리스트 : [30, 10, 20]
append(40) 후의 리스트 : [30, 10, 20, 40]
pop()으로 추출한 값 : 40
pop() 후의 리스트 : [30, 10, 20]
sort() 후의 리스트 : [10, 20, 30]
reverse() 후의 리스트 : [30, 20, 10]
20값의 위치 : 1
insert(2, 222) 후의 리스트 : [30, 20, 222, 10]
remove(222) 후의 리스트 : [30, 20, 10]
extend([77, 88, 77]) 후의 리스트 : [30, 20, 10, 77, 88, 77]
77값의 개수 : 2

Section 02 리스트

■ 리스트 조작 함수

- 기존 리스트는 변경하지 않고 정렬된 새로운 리스트를 생성하고 싶다면 다음과 같이 사용

```
myList = [30, 10, 20]  
newList = sorted(myList)  
print("sorted() 후의 myList : %s" % myList)  
print("sorted() 후의 newList : %s" % newList)
```

실행 결과

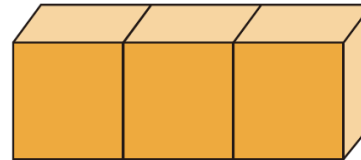
```
sorted() 후의 myList : [30, 10, 20]  
sorted() 후의 newList : [10, 20, 30]
```

Section 02 리스트

■ 2차원 리스트 개념

- 2차원 리스트는 1차원 리스트를 여러 개 연결한 것으로 인덱스를 2개 사용
- 앞서 1차원 리스트는 박스를 나란히 세워 놓은 것이라고 했음
- 따라서 다음과 같이 리스트를 생성하면 aa[0], aa[1], aa[2]라는 항목 3개가 생성됨

aa = [10, 20, 30]



aa[0] aa[1] aa[2]

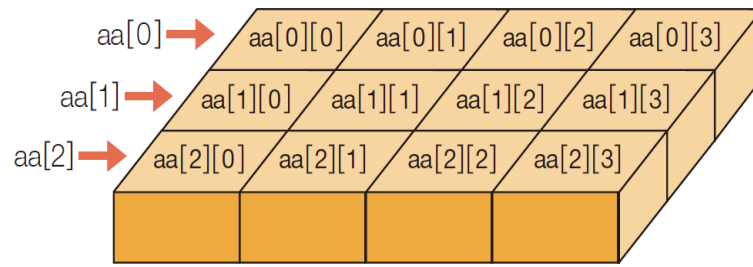
그림 4-3 1차원 리스트의 개념

Section 02 리스트

■ 2차원 리스트 개념

- 1차원 리스트를 확장해 다음과 같이 2차원 리스트를 정의할 수 있음
- 3행 4열 리스트가 생성되어 총 항목의 개수는 12개(3×4)가 됨

```
aa = [[1, 2, 3, 4],  
      [5, 6, 7, 8],  
      [9, 10, 11, 12]]
```



전체 리스트명 : `aa`

그림 4-4 2차원 리스트의 개념

Section 02 리스트

■ 2차원 리스트 개념

- 2차원 리스트에서 각 항목에 접근하려면 `aa[0][0]`처럼 인덱스를 2개 사용
- 1차원 리스트와 같이 첨자가 가로는 0~2, 세로는 0~3으로 변한다는 점에 주의
- 즉 `aa[3][4]` 항목은 존재하지 않음

Section 02 리스트

■ 2차원 리스트 개념

- index가 2개이므로 중첩 for 문을 사용해서 3행 4열 리스트를 생성
- 항목 1부터 12까지 입력하고 출력하는 코드

Code04-05.py

```
01 list1 = []
02 list2 = []
03 value = 1
04 for i in range(0, 3) :
05     for k in range(0, 4) :
06         list1.append(value)
07         value += 1
08     list2.append(list1)
09     list1 = []
10
11 for i in range(0, 3) :
12     for k in range(0, 4) :
13         print("%3d" % list2[i][k], end = " ")
14     print("")
```

실행 결과

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

Section 02 리스트

■ 2차원 리스트 개념

SELF STUDY 4-2

4행 5열의 2차원 리스트를 만들고, 0부터 3의 배수를 입력하고 출력하도록 Code04-05.py를 수정해 보자. 실행 결과는 다음과 같다.

실행 결과

```
0 3 6 9 12
15 18 21 24 27
30 33 36 39 42
45 48 51 54 57
```

Section 03 튜플과 딕셔너리

■ 튜플

- 튜플은 소괄호(())로 생성, 소괄호 생략도 가능
- 값을 수정할 수 없으며, 읽기만 가능해 읽기 전용 자료를 저장할 때 사용

```
tt1 = (10, 20, 30); tt1  
tt2 = 10, 20, 30; tt2
```

실행 결과

```
(10, 20, 30)  
(10, 20, 30)
```

- 튜플은 읽기 전용이므로 다음 코드는 모두 오류 발생

```
tt1.append(40)  
tt1[0] = 40  
del(tt1[0])
```

Section 03 튜플과 딕셔너리

■ 튜플

- 튜플 자체는 del() 함수로 삭제할 수 있음

```
del(tt1)  
del(tt2)
```

- 튜플의 항목에 접근할 때는 리스트처럼 '튜플명[위치]'를 사용함

```
tt1 = (10, 20, 30, 40)  
tt1[0]  
tt1[0] + tt1[1] + tt1[2]
```

실행 결과

```
10  
60
```

Section 03 튜플과 딕셔너리

■ 튜플

- 튜플 범위에 접근하려면 리스트와 마찬가지로 '(시작값:끝값+1)'을 사용

```
tt1[1:3]  
tt1[1:]  
tt1[:3]
```

실행 결과

```
(20, 30)  
(20, 30, 40)  
(10, 20, 30)
```

- 튜플의 덧셈 및 곱셈 연산도 가능

```
tt2 = ('A', 'B')  
tt1 + tt2  
tt2 * 3
```

실행 결과

```
(10, 20, 30, 40, 'A', 'B')  
( 'A', 'B', 'A', 'B', 'A', 'B')
```

Section 03 튜플과 딕셔너리

■ 튜플

SELF STUDY 4-3

다음과 같이 2차원 튜플을 생성한 후 모든 값을 출력해 보자.

```
tt = ((1, 2, 3),  
      (4, 5, 6),  
      (7, 8, 9))
```

실행 결과

```
1 2 3  
4 5 6  
7 8 9
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- 간단한 딕셔너리를 만드는 방법은 다음과 같음
- 키를 1, 2, 3으로 하고, 값을 'a', 'b', 'c'로 만듦

```
dic1 = {1 : 'a', 2 : 'b', 3 : 'c'}  
dic1
```

실행 결과

```
{1 : 'a', 2 : 'b', 3 : 'c'}
```

- 다음과 같이 키와 값을 반대로 생성해도 됨

```
dic2 = {'a' : 1, 'b' : 2, 'c' : 3}  
dic2
```

실행 결과

```
{'a' : 1, 'b' : 2, 'c' : 3}
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- 딕셔너리는 여러 정보를 하나의 변수로 표현할 때 유용
- 예를 들어 홍길동이라는 학생에게는 [표 4-2]와 같은 정보가 있다고 가정

키	값
학번	1000
이름	홍길동
학과	컴퓨터학과

Section 03 튜플과 딕셔너리

■ 딕셔너리

- [표 4-2]를 딕셔너리로 표현한다면 student1 변수에 홍길동 학생의 모든 정보가 저장되어 있는 것

```
student1 = {'학번' : 1000, '이름' : '홍길동', '학과' : '컴퓨터학과'}  
student1
```

실행 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '컴퓨터학과'}
```

- 딕셔너리에는 '딕셔너리명[키]=값' 형식으로 쌍을 추가할 수 있음
- 앞서 생성한 student1에 연락처를 추가하려면 다음 코드를 사용

```
student1['연락처'] = '010-1111-2222'  
student1
```

실행 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '컴퓨터학과', '연락처': '010-1111-2222'}
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- 이미 존재하는 키를 사용하면 새로운 쌍이 추가되는 것이 아니라 기존 값이 변경됨
- 딕셔너리의 특성상 키는 유일해야 하기 때문임
- 학과를 수정하는 코드는 다음과 같음

```
student1['학과'] = '파이썬학과'  
student1
```

실행 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '파이썬학과', '연락처': '010-1111-2222'}
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- 딕셔너리의 쌍은 'del(딕셔너리명[키])' 형식으로 삭제할 수 있음
- 다음 코드는 student1의 학과를 삭제함

```
del(student1['학과'])  
student1
```

실행 결과

```
{'학번': 1000, '이름': '홍길동', '연락처': '010-1111-2222'}
```

- 딕셔너리의 키는 유일해야 하므로 다음 코드를 작성하면 동일한 키를 갖는 딕셔너리를 생성하는 것이 아니라 마지막에 있는 키가 적용됨

```
student1 = {'학번': 1000, '이름': '홍길동', '학과': '파이썬학과', '학번': 2000}  
student1
```

실행 결과

```
{'학번': 2000, '이름': '홍길동', '학과': '파이썬학과'}
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- 키로 값에 접근하는 코드

```
student1['학번']  
student1['이름']  
student1['학과']
```

실행 결과

```
2000  
'홍길동'  
'파이썬학과'
```

- 딕셔너리명.get(키) 함수를 사용해 키로 값에 접근할 수 있음

```
student1.get('이름')
```

실행 결과

```
'홍길동'
```

- 딕셔너리와 관련된 함수 중 딕셔너리명.keys()는 딕셔너리의 모든 키를 반환

```
student1.keys()
```

실행 결과

```
dict_keys(['학번', '이름', '학과'])
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- 실행 결과의 dict_keys가 보기 싫으면 list(딕셔너리명.keys()) 함수를 사용하면 됨
- 딕셔너리명.items() 함수를 사용하면 키와 값의 쌍을 튜플 형태로도 구할 수 있음

```
list(student1.keys())
```

실행 결과

```
['학번', '이름', '학과']
```

- 딕셔너리명.values() 함수는 딕셔너리의 모든 값을 리스트로 만들어 반환함

```
student1.values()
```

실행 결과

```
dict_values([2000, '홍길동', '파이썬학과'])
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- 딕셔너리명.items() 함수를 사용하면 튜플 형태로도 구할 수 있음

```
student1.items()
```

실행 결과

```
dict_items([('학번', 2000), ('이름', '홍길  
동'), ('학과', '파이썬학과')])
```

- 딕셔너리 안에 해당 키가 있는지 없는지는 in을 사용해 확인할 수 있음
- 다음 코드는 딕셔너리에 키가 있다면 True를 반환하고, 없다면 False를 반환함

```
'이름' in student1  
'주소' in student1
```

실행 결과

```
True  
False
```

Section 03 튜플과 딕셔너리

■ 딕셔너리

- for 문을 활용해 딕셔너리의 모든 값을 출력하는 코드

Code04-07.py

```
01 singer = {}  
02  
03 singer['이름'] = '트와이스'  
04 singer['구성원 수'] = 9  
05 singer['데뷔'] = '서바이벌 식스틴'  
06 singer['대표곡'] = 'SIGNAL'  
07  
08 for k in singer.keys():  
09     print('%s --> %s' % (k, singer[k]))
```

실행 결과

이름 --> 트와이스
구성원 수 --> 9
데뷔 --> 서바이벌 식스틴
대표곡 --> SIGNAL

Section 04 문자열

■ 문자열의 개념

- 파이썬에서는 문자열을 큰따옴표(" ")나 작은따옴표(' ')로 묶어 표현함
- 큰따옴표를 사용하더라도 print()로 출력하면 작은따옴표로 표시함
- 문자열은 리스트와 비슷한 부분이 많은데, 리스트는 대괄호([])로 묶고 문자열은 작은따옴표(' ')로 묶어 출력된다는 것만 다름

```
aa = [10, 20, 30, 40, 50]  
aa[0]  
aa[1:3]  
aa[3:]
```

실행 결과

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

```
ss = "파이썬최고"  
ss[0]  
ss[1:3]  
ss[3:]
```

실행 결과

```
'파'  
'이썬'  
'최고'
```


Section 04 문자열

■ 문자열 개념

- 문자열도 리스트와 마찬가지로 덧셈(+) 기호를 사용해 연결함
- 또 곱셈(*) 기호를 사용해 문자열을 반복할 수도 있음

```
ss = '파이썬' + '최고'  
ss  
ss = '파이썬' * 3  
ss
```

실행 결과

```
'파이썬최고'  
'파이썬파이썬파이썬'
```

- 문자열 길이를 파악할 때도 리스트처럼 len() 함수를 사용

```
ss = '파이썬abcd'  
len(ss)
```

실행 결과

```
7
```

Section 04 문자열

■ 문자열 개념

- 문자열도 len() 함수로 개수를 파악할 수 있기 때문에 리스트처럼 for 문을 사용해 처리할 수 있음

Code04-08.py

```
01 ss = '파이썬짱!'
02
03 sslen = len(ss)
04 for i in range(0, sslen) :
05     print(ss[i] + '$', end = '')
```

실행 결과

파\$이\$썬\$짱\$!\$

SELF STUDY 4-4

Code04-08.py를 수정해서 '파이썬은완전재미있어요'에서 '파#썬#완#재#있#요'를 출력해 보자. 즉 짝수 번째 글자는 그대로 출력되고, 홀수 번째는 글자 대신 #이 표시되도록 하면 된다.

힌트 if 문을 사용해 i가 짝수일 때와 홀수일 때를 다르게 처리한다. 짝수는 2로 나누어서 나머지 값이 0이면 짝수이다.

Section 04 문자열

■ 문자열 함수

- 문자열 데이터형 자체에서 제공하는 다양한 문자열 함수로 문자열을 처리 가능

■ 대·소문자 변환

표 4-3 대·소문자 변환 함수

함수	기능
upper()	소문자를 대문자로 변환한다.
lower()	대문자를 소문자로 변환한다.
swapcase()	대·소문자를 상호 변환한다.
title()	각 단어의 앞 글자만 대문자로 변환한다. 한글 또는 기호에는 영향을 주지 않는다.

Section 04 문자열

■ 대·소문자 변환

```
ss = 'Python is Easy. 그래서 programming이 재미있습니다. ^^'  
ss.upper()  
ss.lower()  
ss.swapcase()  
ss.title()
```

실행 결과

```
'PYTHON IS EASY. 그래서 PROGRAMMING이 재미있습니다. ^^'  
'python is easy. 그래서 programming이 재미있습니다. ^^'  
'pYTHON IS eASY. 그래서 PROGRAMMING이 재미있습니다. ^^'  
'Python Is Easy. 그래서 Programming이 재미있습니다. ^^'
```

Section 04 문자열

■ 대·소문자 변환



여기서 잠깐

함수와 메서드는 상당히 비슷하지만 차이점이 약간 있다. 우선 함수는 단독으로 사용된다. 예로 리스트나 문자열의 길이를 알아내는 `len()` 함수는 다음과 같이 사용된다.

```
ss = "abcd"
len(ss)
```

함수
(Function)와
메서드
(Method)

하지만 메서드는 문자열 자료형에 그 기능이 들어 있기 때문에 '변수명.메서드()' 형식으로 사용된다. 예로 문자열을 대문자로 바꾸는 `upper()` 메서드는 다음과 같이 사용된다.

```
ss = "abcd"
ss.upper()
```

객체지향에서는 함수와 메서드를 정확히 구분해야 하지만 지금은 '둘 다 뒤에 괄호가 붙는다' 정도만 알면 되므로 모두 함수라고 칭한다.

Section 04 문자열

■ 문자열 찾기

```
ss = '파이썬 공부는 즐겁습니다. 물론 모든 공부가 다 재미있지는 않죠. ^^'
ss.count('공부')
print(ss.find('공부'), ss.rfind('공부'), ss.find('공부', 5), ss.find('없다'))
print(ss.index('공부'), ss.rindex('공부'), ss.index('공부', 5))
print(ss.startswith('파이썬'), ss.startswith('파이썬', 10), ss.endswith('^^'))
```

실행 결과

```
2
4 21 21 -1
4 21 21
True False True
```

표 4-4 문자열을 찾는 함수

함수	기능
count('찾을문자열')	'찾을문자열'이 몇 개 들어 있는지 개수를 센다.
find('찾을문자열')	'찾을문자열'이 왼쪽 끝(0번)부터 시작해서 몇 번째에 위치하는지 찾는다.
rfind('찾을문자열')	find()와 반대로 오른쪽부터 찾는다. 앞의 예에서 '공부'라는 글자를 찾으려고 find('공부') 함수를 사용하면 왼쪽 끝(0번)부터 찾기 시작하므로 4번 위치를 반환한다.
find('공부', 5)	찾기 시작할 위치를 왼쪽에서 5번째('부' 글자)부터 시작하므로 21번 위치를 반환한다. find() 함수는 찾을 문자열이 없으면 -1을 반환한다.
index()	find() 함수와 동일하지만 index('없다')처럼 찾을 문자열이 없으면 오류가 발생한다.
startswith('찾을문자열')	'찾을문자열'로 시작하면 True를 반환하고, 그렇지 않으면 False를 반환한다.
endswith('찾을문자열')	'찾을문자열'로 끝나면 True를 반환하고, 그렇지 않으면 False를 반환한다.
startswith('찾을문자열', 위치)	'위치'에서 '찾을문자열'로 시작하면 True를 반환한다.

Section 04 문자열

■ 문자열 공백 삭제·변경

- 문자열의 앞뒤 공백이나 특정 문자를 삭제하려면 strip(), rstrip(), lstrip() 함수를 사용
- 단, 문자열中间的 공백은 삭제되지 않음

```
ss = ' 파 이 션 '  
ss.strip()  
ss.rstrip()  
ss.lstrip()
```

실행 결과

```
'파 이 션'  
' 파 이 션'  
'파 이 션 '
```

```
ss = '----파---이---션----'  
print(ss.strip('-'))  
ss = '<<<파 << 이 >> 션>>>'  
print(ss.strip('<>'))
```

실행 결과

```
파---이---션  
파 << 이 >> 션
```

Section 04 문자열

■ 문자열 함수

- 문자열 변경은 `replace('기존문자열', '새문자열')` 함수를 사용

```
ss = '열심히 파이썬 공부 중~~'  
ss.replace('파이썬', 'Python')
```

실행 결과

```
'열심히 Python 공부 중~~'
```


Section 04 문자열

■ 문자열 분리·결합

- split() 함수는 문자열을 공백이나 다른 문자로 분리해 리스트를 반환
- join() 함수는 문자열을 합침

```
ss = 'Python을 열심히 공부 중'
ss.split()
ss = '하나:둘:셋'
ss.split(':')
ss = '하나\n둘\n셋'
ss.splitlines()
ss = '%'
ss.join('파이썬')
```

실행 결과

```
['Python을', '열심히', '공부', '중']
['하나', '둘', '셋']
['하나', '둘', '셋']
'파%i%썬'
```

Section 04 문자열

■ 함수명에 대입

- `map(함수명, 리스트명)` 함수는 리스트값 하나하나를 함수명에 대입
- 따라서 다음 코드와 같은 방법으로 문자열로 구성된 리스트를 숫자로 변환할 수 있음

```
before = ['2019', '12', '31']  
after = list(map(int, before))  
after
```

실행 결과

```
[2019, 12, 31]
```

- `int()` 함수의 이름을 `map()` 함수의 매개변수로 사용함
- 즉 `int('2019')`, `int('12')`, `int('31')`의 연속적인 효과를 가져옴
- 그리고 결과 값을 다시 `list()` 함수를 사용해 리스트 형태로 변환함

Section 04 문자열

■ 문자열 구성 파악

- 문자열이 무엇으로 구성되어 있는지 함수로 파악해서 True와 False로 알려 줌
- 다음 코드는 결과가 모두 True

```
'1234'.isdigit()  
'abcd'.isalpha()  
'abc123'.isalnum()  
'abcd'.islower()  
'ABCD'.isupper()  
' '.isspace()
```

표 4-5 문자열 구성 파악 함수

함수	기능
isdigit()	숫자로만 구성되어 있는지 확인한다.
isalpha()	글자(한글, 영어)로만 구성되어 있는지 확인한다.
isalnum()	글자와 숫자가 섞여 있는지 확인한다.
islower()	전체가 소문자로만 구성되어 있는지 확인한다.
isupper()	대문자로만 구성되어 있는지 확인한다.
isspace()	공백 문자로만 구성되어 있는지 확인한다. 이때 문자열을 ss='1234' 방식으로 변수에 넣은 후 ss.isdigit()처럼 사용해도 되고, '1234'.isdigit() 방식으로 해당 문자열에 함수를 직접 사용해도 된다.

Section 04 문자열

■ 문자열 구성 파악

SELF STUDY 4-5

입력한 값이 영어나 한글이면 '글자입니다', 숫자이면 '숫자입니다', 섞여 있으면 '글자+숫자입니다', 특수문자 등이면 '모르겠습니다'가 출력되는 프로그램을 작성해 보자.

실행 결과

문자열 입력 : **abcd123**

글자+숫자입니다.

Section 05 함수

■ 함수의 개념

- 함수(Function)는 '무엇'을 넣으면 '어떤 것'을 돌려주거나 '어떤 일'이 일어나게 함
- 파이썬에서 제공하는 함수는 다음 형식으로 사용함

```
함수명()
```

- 프린트 함수를 통한 예시

```
print("CookBook-파이썬")
```

Section 05 함수

■ 함수의 형식

- 반복적으로 코딩해야 할 내용을 함수로 만들어 두면 필요할 때 바로 사용할 수 있음

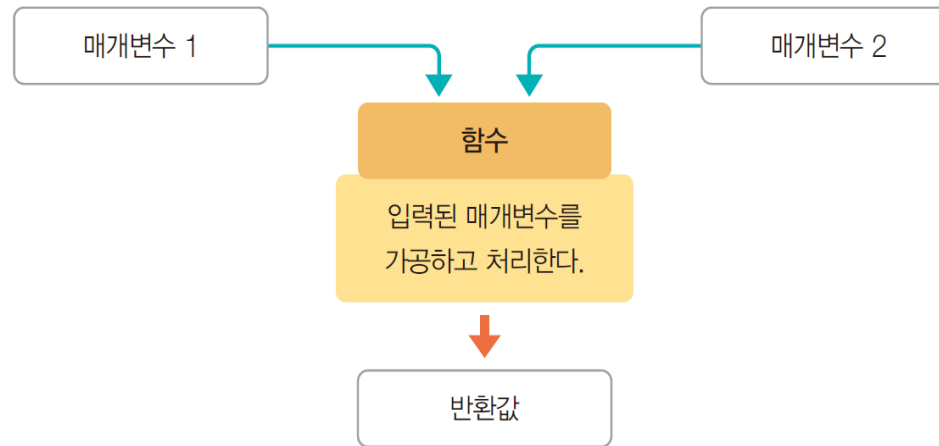


그림 4-5 함수의 기본 형식

Section 05 함수

■ 함수의 형식

- 두 정수를 입력받아 합계를 반환하는 plus() 함수

Code04-09.py

```
01  ## 함수 선언 부분 ##
02  def plus(v1, v2) :
03      result = 0
04      result = v1 + v2
05      return result
06
07  ## 변수 선언 부분 ##
08  hap = 0
09
10  ## 메인 코드 부분 ##
11  hap = plus(100, 200)
12  print("100과 200의 plus() 함수 결과는 %d" % hap)
```

실행 결과

100과 200의 plus() 함수 결과는 300

Section 05 함수

■ 함수의 형식

- [그림 4-6]은 plus() 함수를 정의하고 호출하는 과정을 보여줌

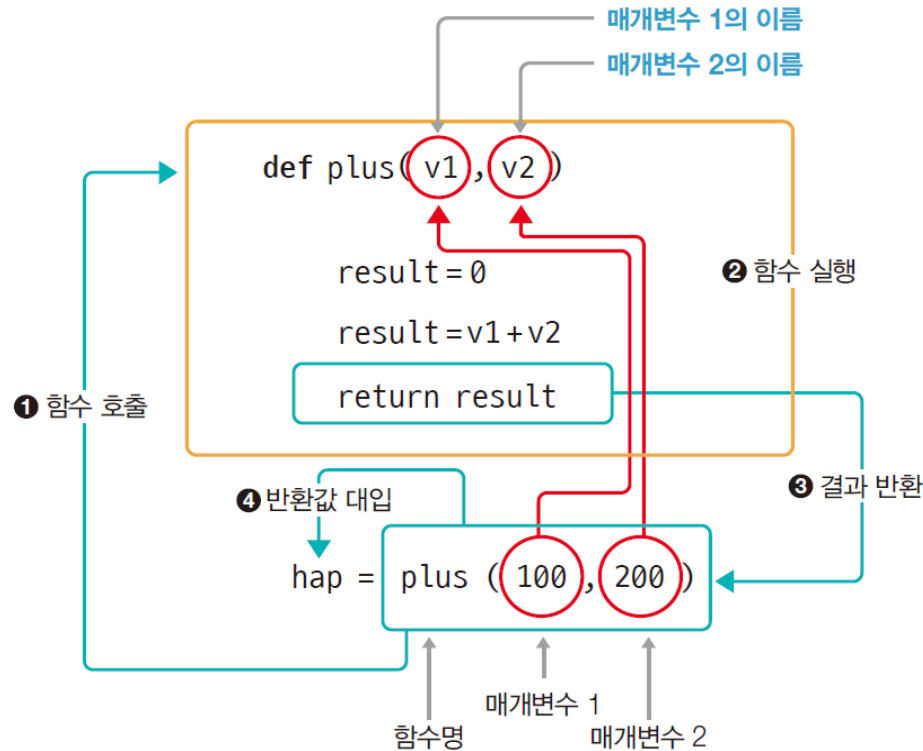


그림 4-6 plus() 함수의 형식과 호출 순서

Section 05 함수

■ 함수의 형식

- [그림 4-6]보다 더욱 간단하게 다음과 같이 설명할 수도 있음

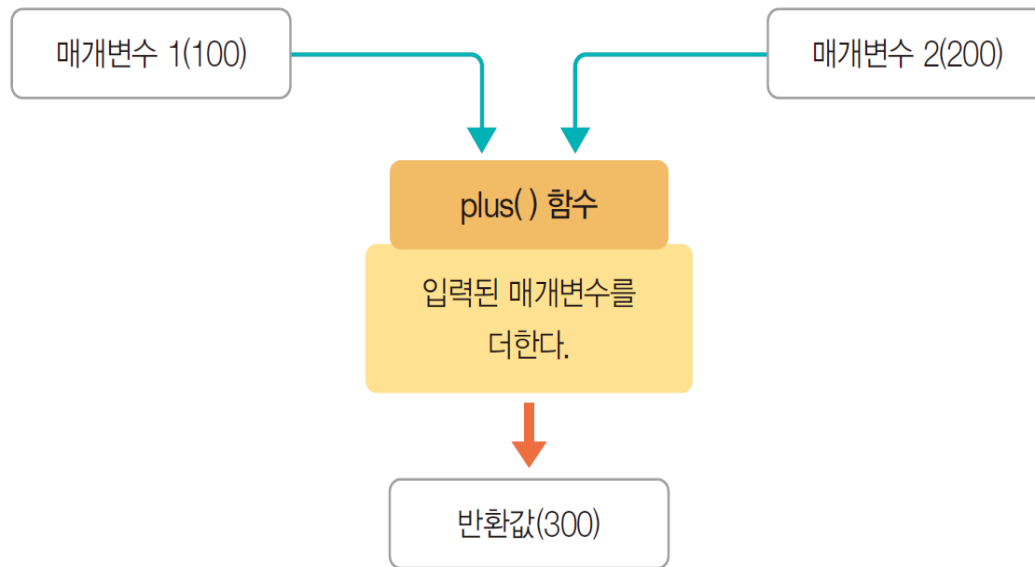


그림 4-7 plus() 함수의 호출 간략 표현

Section 05 함수

■ 함수의 형식

- 사용자가 입력한 두 숫자로 덧셈, 뺄셈, 곱셈, 나눗셈을 하는 계산기 함수

Code04-10.py

```
01  ## 함수 선언 부분 ##
02  def calc(v1, v2, op) :
03      result = 0
04      if op == '+' :
05          result = v1 + v2
06      elif op == '-' :
07          result = v1 - v2
08      elif op == '*' :
09          result = v1 * v2
10      elif op == '/' :
11          result = v1 / v2
12
13      return result
14
```

Section 05 함수

■ 함수의 형식

- 사용자가 입력한 두 숫자로 덧셈, 뺄셈, 곱셈, 나눗셈을 하는 계산기 함수

Code04-10.py

```
15  ## 변수 선언 부분 ##
16  res = 0
17  var1, var2, oper = 0, 0, ""
18
19  ## 메인 코드 부분 ##
20  oper = input("계산을 입력하세요(+, -, *, /) : ")
21  var1 = int(input("첫 번째 수를 입력하세요 : "))
22  var2 = int(input("두 번째 수를 입력하세요 : "))
23
24  res = calc(var1, var2, oper)
25
26  print("## 계산기 : %d %s %d = %d" % (var1, oper, var2, res))
```

실행 결과

계산을 입력하세요(+, -, *, /) : *
첫 번째 수를 입력하세요 : 7
두 번째 수를 입력하세요 : 8
계산기 : 7 * 8 = 56

Section 05 함수

■ 함수 형식

SELF STUDY 4-6

Code04-10.py에 다음 기능을 추가해 보자.

- ① 숫자1, 연산자, 숫자2 순서로 입력받는다.
- ② 제곱(**) 연산자를 추가한다.
- ③ 0으로 나누려고 하면 메시지를 출력하고 계산되지 않도록 한다.

힌트 메인 코드 부분에 if~else 문을 활용한다.

실행 결과

첫 번째 수를 입력하세요 : 2
계산을 입력하세요(+, -, *, /, **) : **
두 번째 수를 입력하세요 : 4
계산기 : 2 ** 4 = 16

실행 결과

첫 번째 수를 입력하세요 : 8
계산을 입력하세요(+, -, *, /, **) : /
두 번째 수를 입력하세요 : 0
0으로는 나누면 안 됩니다.ㅠㅠ

Section 05 함수

■ 지역 변수와 전역 변수

- 지역 변수는 말 그대로 한정된 지역에서만 사용되는 변수
- 전역 변수는 프로그램 전체에서 사용되는 변수

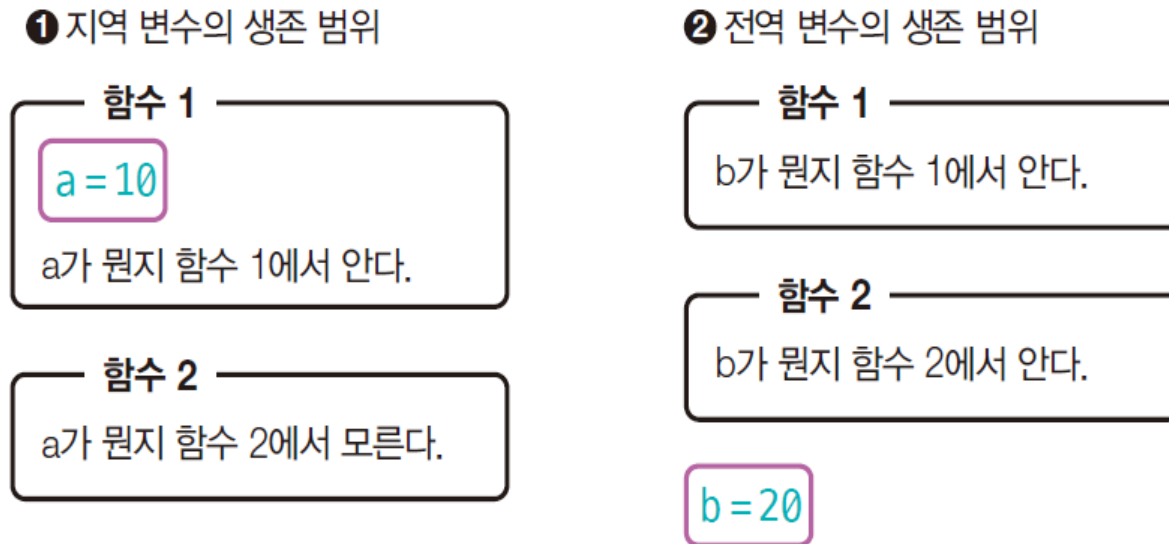


그림 4-8 지역 변수와 전역 변수의 생존 범위

Section 05 함수

■ 지역 변수와 전역 변수

- 변수명이 같다면 지역 변수가 우선됨

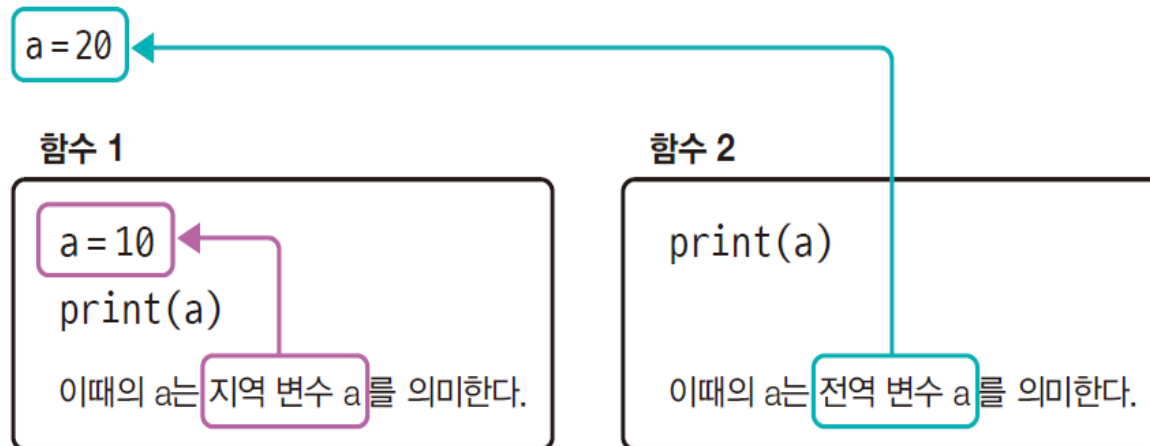


그림 4-9 지역 변수와 전역 변수의 공존

Section 05 함수

■ 지역 변수와 전역 변수

Code04-11.py

```
01  ## 함수 선언 부분 ##
02  def func1() :
03      a = 10 # 지역 변수
04      print("func1()에서 a값 %d" % a)
05
06  def func2() :
07      print("func2()에서 a값 %d" % a)
08
09  ## 변수 선언 부분 ##
10  a = 20      # 전역 변수
11
12  ## 메인 코드 부분 ##
13  func1()
14  func2()
```

실행 결과

func1()에서 a값 10
func2()에서 a값 20

Section 05 함수

■ 지역 변수와 전역 변수

- 10행을 지우고 다시 실행하면, 다음과 같은 오류가 발생할 것
- func1() 함수의 a는 지역 변수가 있으므로 잘 출력되지만, func2() 함수에는 a가 없으므로 오류가 발생함

실행 결과

```
func1()에서 a의 값 10
Traceback (most recent call last):
File "C:/CookAnalysis/04-11.py", line 14, in <module>
    func2()
File "C:/CookAnalysis/04-11.py", line 7, in func2
    print("func2()에서 a값 %d" % a)
NameError: name 'a' is not defined
```


Section 05 함수

■ global 예약어

- 함수 안에서 사용되는 변수를 지역 변수 대신 전역 변수로 사용하고 싶을 수 있음
- 지역 변수가 아닌 전역 변수로 사용하려면 함수 안에서 global이라는 예약어로 전역 변수라는 것을 명시해야 함

Code04-12.py

```
01  ## 함수 선언 부분 ##
02  def func1() :
03      global a # 이 함수 안에서 a는 전역 변수
04      a = 10
05      print("func1()에서 a값 %d" % a)
06
07  def func2() :
08      print("func2()에서 a값 %d" % a)
09
10  ## 변수 선언 부분 ##
11  a = 20 # 전역 변수
12
13  ## 메인 코드 부분 ##
14  func1()
15  func2()
```

실행 결과

func1()에서 a값 10
func2()에서 a값 10

Section 05 함수

■ 함수 반환값

- 함수를 사용하다 보면 실행 후에 값을 돌려줄 때도 있고 돌려주지 않을 때도 있음

■ 반환값이 있는 함수

- 함수에서 어떤 계산이나 작동을 한 후 반환할 값이 있으면 'return 반환값' 형식으로 표현

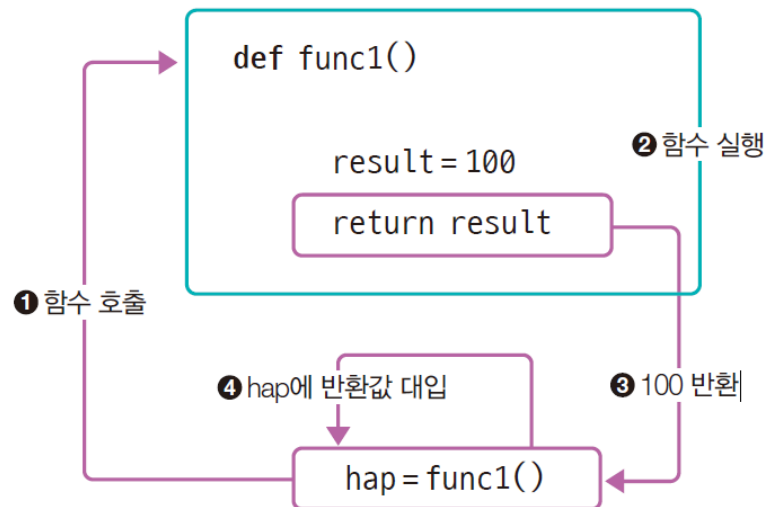


그림 4-10 값의 반환

Section 05 함수

■ 반환값이 없는 함수

- 함수의 실행 결과로 돌려줄 것이 없을 때는 return 문을 생략하면 됨
- 또는 반환값 없이 return만 입력해도 됨

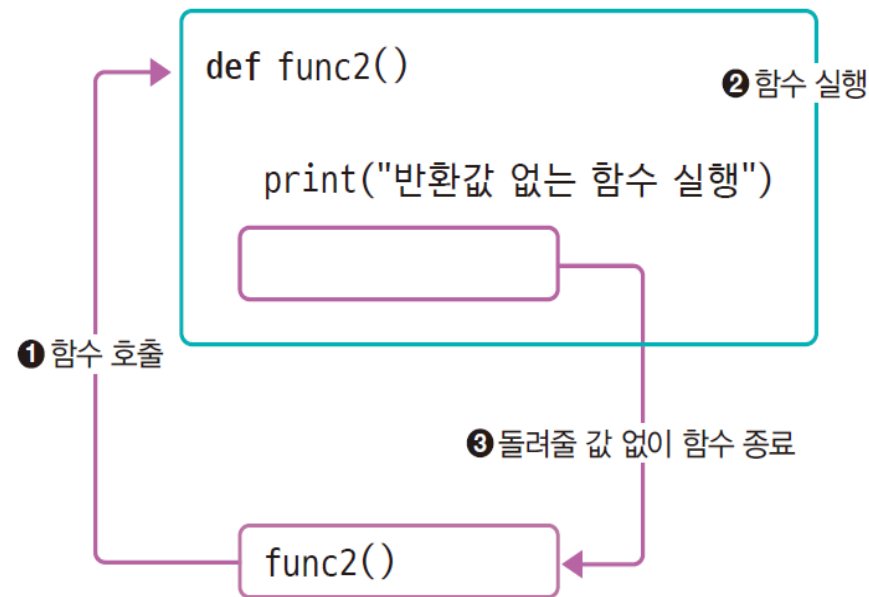


그림 4-11 반환값이 없는 함수의 작동

Section 05 함수

■ 함수 반환값

Code04-13.py

```
01  ## 함수 선언 부분 ##
02  def func1() :
03      result = 100
04      return result
05
06  def func2() :
07      print("반환값이 없는 함수 실행")
08
09  ## 변수 선언 부분 ##
10  hap = 0
11
12  ## 메인 코드 부분 ##
13  hap = func1()
14  print("func1()에서 돌려준 값 ==> %d" % hap)
15  func2()
```

실행 결과

func1()에서 돌려준 값 ==> 100
반환값이 없는 함수 실행

Section 05 함수

■ 반환값이 여러 개인 함수

- 함수는 문법상 반환값이 2개 이상일 수 없음에도 코드를 작성하다 보면 종종 값을 2개 이상 반환해야 할 때가 있음
- 리스트에 반환할 값을 넣은 후 리스트를 반환하면 됨
- 리스트에 아무리 많은 데이터를 넣더라도 결국 리스트 1개를 반환하는 것이므로 문법상 문제가 없음

Section 05 함수

■ 반환값이 여러 개인 함수

Code04-14.py

```
01  ## 함수 선언 부분 ##
02  def multi(v1, v2) :
03      retList = [] # 반환할 리스트
04      res1 = v1 + v2
05      res2 = v1 - v2
06      retList.append(res1)
07      retList.append(res2)
08      return retList
09
10  ## 변수 선언 부분 ##
11  myList = []
12  hap, sub = 0, 0
13
14  ## 메인 코드 부분 ##
15  myList = multi(100, 200)
16  hap = myList[0]
17  sub = myList[1]
18  print("multi()에서 돌려준 값 ==> %d, %d" % (hap, sub))
```

실행 결과

multi()에서 돌려준 값 ==> 300, -100

Section 05 함수

■ pass 예약어

- 함수를 실행한 결과로 돌려줄 것이 없을 때는 return 문을 생략함
- 또한, 함수를 구현할 때 일단 이름만 만들어 놓고 그 내용은 pass 예약어를 사용해 비워 놓을 수 있음

```
def myFunc() :  
    pass
```

- 다음과 같이 True 이후의 구문을 비워둘 경우 오류 발생

```
if True :  
  
else :  
    print('거짓이네요')
```

Section 05 함수

■ pass 예약어

- 다음과 같이 pass를 넣어 간편하게 해결할 수 있음

```
if True :  
    pass  
else :  
    print('거짓이네요')
```


Section 05 함수

■ 매개변수 개수를 지정해 전달하는 방법

Code04-15.py

```
01  ## 함수 선언 부분 ##
02  def para2_func( v1, v2 ) :
03      result = 0
04      result = v1 + v2
05      return result
06
07  def para3_func( v1, v2, v3 ) :
08      result = 0
09      result = v1 + v2 + v3
10      return result
11
12  ## 변수 선언 부분 ##
13  hap = 0
14
15  ## 메인 코드 부분 ##
16  hap = para2_func(10, 20)
17  print("매개변수가 2개인 함수를 호출한 결과 ==> %d" % hap)
18  hap = para3_func(10, 20, 30)
19  print("매개변수가 3개인 함수를 호출한 결과 ==> %d" % hap)
```

실행 결과

매개변수가 2개인 함수를 호출한 결과 ==> 30

매개변수가 3개인 함수를 호출한 결과 ==> 60

Section 05 함수

■ [프로그램] 완성

Code04-16.py

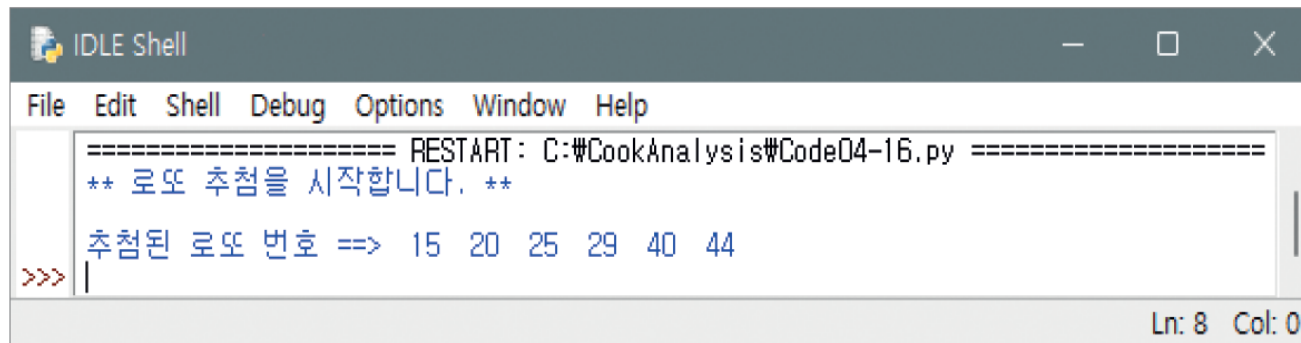
```
01  import random
02
03  ## 함수 선언 부분 ##
04  def getNumber() :
05      return random.randrange(1, 46)
06
07  ## 변수 선언 부분 ##
08  lotto = []
09  num = 0
10
11  ## 메인 코드 부분 ##
12  print("** 로또 추첨을 시작합니다. ** \n");
13
14  while True :
15      num = getNumber()
16
17      if lotto.count(num) == 0 :
18          lotto.append(num)
```

Section 05 함수

■ [프로그램]의 완성

Code04-16.py

```
19     if len(lotto) >= 6 :  
20         break  
21  
22     print("추첨된 로또 번호 ==> ", end = '')  
23     lotto.sort()  
24     for i in range(0, 6) :  
25         print("%d " % lotto[i], end = '')
```



```
===== RESTART: C:\CookAnalysis\Code04-16.py =====  
** 로또 추첨을 시작합니다. **  
추첨된 로또 번호 ==> 15 20 25 29 40 44  
>>> |
```

Ln: 8 Col: 0