7장 함수

- 코딩하다 보면 특정 기능을 반복해서 수행해야 할 때가 있음
- 그때마다 같은 기능을 수행하는 코드 를 반복해서 작성한다면 비효율적
- 사용할 수 있는 것이 함수(function)
- 함수는 특정 기능을 수행하는 코드의 묶음
- 함수를 이용하면 같은 기능을 수행하는 코드를 반복해서 작성할 필요가 없음

- 출력을 위한 print() 함수, 데이터 타입을 알기 위한 type() 함수 등이 바로 내장 함수

7.1 함수 정의와 호출

함수의 기본 구조

- 함수는 특정 기능을 수행하는 코드의 묶음
- 수학 함수에서 입력값을 프로그래밍 함수에서는 인자
- 프로그래밍에서는 이 인자를 통해 함수에 값을 전달할 수 있음
- 수학 함수에서 계산된 결과값을 프로그래밍 함수에 서는 반환 값
- 프로그래밍의 함수는 수학 함수와 달리 인자와 반환 값이 없을 수도 있음

- 함수를 이용하려면 먼저 함수를 정의(함수 만들기)해야 함
- 함수를 정의한 후에야 함수를 호출(정의된 함수 부르기)할 수 있음

- def 함수명([인자1, 인자2, • • , 인자n]): 〈코드 블록〉 [return <반환 값>]
- 함수는 def 키워드로 시작
- 프로그래머가 정의한 함수명(함수 이름)을 입력
- 함수 명은 주로 영문 알파벳 소문자로 구성되며 가독성을 높이기 위해 밑줄(_)을 이용
- 함수 명을 입력한 후에는 소괄호와 콜론을 입력
- 대괄호 안의 내용은 필요에 따라 쓸 수 도 있고 쓰지 않을 수도 있음

- 함수에서 사용할 인자가 있으면 소괄호 안에 인자를 입력
- 인자를 여러 개 입력할 수 있는데 인자와 인자 사이는 콤마로 구분
- 함수에서 사용할 인자가 없으면 소괄호만 입력
- 소괄호 다음에는 콜론(:)을 입력
- 그 이후 줄을 바꾸고 들여쓰기를 한 후에〈코드 블록〉에 코드를 입력
- 반환 값이 있으면 마지막 줄에 return <반환 값>을 입력하고 없으면 아무것도 입력하지 않음

- 인자도 반환 값도 없는 함수
- 인자는 있으나 반환 값이 없는 함수
- 인자도 있고 반환 값도 있는 함수

- 함수를 정의한 후 정의된 함수를 호출해야 함수가 실행

- 함수를 호출할 때는 함수를 정의할 때 지정했던 인자도 함께 써야 함
- 이때 인자의 개수와 순서는 같아야 함
- 함수를 정의할 때 인자가 없었으면 함수를 호출할 때 소괄호 안에 아무것도 입력하지 않음
- 정의한 함수를 호출하는 코드 구조
- 함수명([인자1, 인자2, • • • , 인자n])

인자도 반환 값도 없는 함수

- 함수는 전달할 인자와 반환 값의 유무에 따라 다양한 구조로 만들 수 있음
- 인자도 없고 반환 값도 없는 가장 간단한 구조의 함수

In [ ]:

```
def my_func():
    print("My first function!")
    print("This is a function.")
```

- 함수를 정의하고 실행하면 아무 일도 일어나지 않음
- 오류가 발생하지 않고 아무 일도 일어나지 않는다면 함수가 잘 정의된 것
- 만약 오류가 발생한다면 잘못된 곳을 수정한 후 다시 실행

- 함수를 정의한 후에 호출해야 비로소 함수가 실행
- my_func() 함수에는 인자가 없기 때문에 함수명과 소괄호만 입력해 함수를 호출

In [ ]:

```
my_func()
```

인자는 있으나 반환 값이 없는 함수

- 인자는 있지만 반환 값이 없는 함수를 정의하고 호출
- 인자{여기서는 문자열) 하나를 넘겨받아 출력하는 함수를 정의

In [ ]:

```
def my_friend(friendName):
    print("{}는 나의 친구입니다.".format(friendName))
```

- 정의한 함수에 인자를 각각 입력해서 호출

In [ ]:

```
my_friend("철수")
my_friend("영미")
```

- 함수를 두 번 호출했는데 각각 인자를 다르게 입력해 출력 결과가 다르게 나왔음
- 함수에 인자를 이용하면 인자에 따라 다른 결과를 얻을 수 있음

- 함수에서 인자는 하나일 수도 있고 여러 개일 수도 있음
- 인자가 세 개인 함수

In [ ]:

```
def my_student_info(name, school_ID, phoneNumber):
    print("-----------------------")
    print("- 학생이름:", name)
    print("- 학급번호:", school_ID)
    print("- 전화번호:", phoneNumber)
```

- 함수를 정의할 때 세 개의 인자를 넘겨받도록 만들었기 때문에 호출할 때도 세 개의 인자를 입력해서 함수를 호출

In [ ]:

```
my_student_info("현아", "01", "01-235-6789")
my_student_info("진수", "02", "01-987-6543")
```

- 정의한 my_student_info() 함수를 수정하면 이 함수를 호출하는 모든 코드에 적용
- my_student_info() 함수의 출력 양식을 수정한 함수

In [ ]:

```
def my_student_info(name, school_ID,  phoneNumber):
    print("***************************")
    print("* 학생이름:", name)
    print("* 학급번호:", school_ID)
    print("* 전화번호:", phoneNumber)
```

- 입력한 인자를 변경 없이 그대로 입력해서 my_student_info() 함수를 다시 호출

In [ ]:

```
my_student_info("현아", "01", "01-235-6789")
my_student_info("진수", "02", "01-987-6543")
```

인자도 있고 반환 값도 있는 함수

- 인자를 받아서 처리한 후에 값을 반환하는 함수

- 가로와 세로 길이 인자 두 개를 받아서 넓이를 계산한 후 그 결과를 반환하는 함수를 정의

In [ ]:

```
def my_calc(x,y):
    z = x*y
    return z
```

- 인자를 입력해서 함수를 호출하면 결과값(넓이)을 출력

In [ ]:

```
my_calc(3,4)
```

- 파이썬에서는 함수의 인자로 리스트, 세트, 튜플, 딕셔너리도 사용 있음

- 리스트를 인자로 갖는 함수

In [ ]:

```
def my_student_info_list(student_info):
    print("****************************")
    print("* 학생이름:", student_info[0])
    print("* 학급번호:", student_info[1])
    print("* 전화번호:", student_info[2])
    print("****************************")
```

- 함수의 인자는 리스트이므로 인자를 리스트로 입력해서 함수를 호출

In [ ]:

```
student1_info = ["현아", "01", "01-235-6789"]
my_student_info_list(student1_info)
```

- 리스트를 변수에 넣은 후에 이것을 함수의 인자로 넣지 않고 리스트를 바로 인자로 지정할 수도 있음

In [ ]:

```
my_student_info_list(["진수", "02", "01-987-6543"])
```

7.2 변수의 유효 범위

- 함수 안에서 정의한{혹은 생성한) 변수는 함수 안에서만 사용
- 함수 안에서 생성한 변수는 함수를 호출해 실행되는 동안만 사용할 수 있고 함수 실행이 끝나면 더는 사용할 수 없음

- 함수 안에서 정의한 변수는 함수 영역 안에서만 동작하는 변수이므로 지역 변수(local variable)
- 지역 변수는 함수가 호출될 때 임시 저장 공간(메모리)에 할당되고 함수 실행이 끝나면 사라짐
- 다른 함수에서 같은 이름으로 변수를 사용해도 각각 다른 임시 저장 공간에 할당되므로 독립적으로 동작

- 지역 변수의 상대적인 개념으로 함수 밖에서 생성한 변수인 전역 변수(global variable)
- 지역 변수는 함수 안에서만 사용할 수 있지만 전역 변수는 코드 내 어디서나 사용할 수 있음

- 변수를 정의할 때 변수가 저장되는 공간을 이름 공간이라고 하는데
- 변수를 함수 안에서 정의했느냐 , 함수 밖에서 정의했느냐에 따라 코드 내에서 영향을 미치는 유효 범위(scope)가 달라짐

- 지역 변수를 저장하는 이름 공간을 지역 영역(local scope)이라고 하고
- 전역 변수를 저장하는 이름 공간을 전역 영역(global scope)이라고 함

- 파이썬 자체에서 정의한 이름 공간을 내장 영역(built-in scope)이라고 함
- 함수에서 어떤 변수를 호출하면 지역 영역, 전역 영역, 내장 영역 순서대로 변수가 있는지 확인
- 스코핑 룰(Scoping rule) 혹은 LGB 룰 (Local/Global/Built_in rule) 이라고 함

- 함수를 작성할 때 지역 변수와 전역 변수의 이름이 다르다면 문제가 되지 않지만
- 만약 동일한 변수명을 지역 변수와 전역 변수에 모두 이용했다면 이것을 이용할 때 스코핑 룰에 따라 변수가 선택

- 하나의 코드에서 같은 이름의 변수를 지역 변수와 전역 변수로 모두 사용

In [ ]:

```
a = 5 # 전역 변수

def func1():
    a = 1 # 지역 변수. func1()에서만 사용
    print("[func1] 지역 변수 a =", a)

def func2():
    a = 2 # 지역 변수. func2()에서만 사용
    print("[func2] 지역 변수 a =", a)

def func3():
    print("[func3] 전역 변수 a =", a)

def func4():
    global a  # 함수 내에서 전역 변수 변경 위해 선언
    a = 4     # 전역 변수의 값 변경
    print("[func4] 전역 변수 a =",a)
```

- 함수 밖에서 정의된 전역 변수 a에는 5를 할당
- 함수 func1()과 func2() 안에서 정 의된 지역 변수 a에는 각각 1과 2를 할당하고 출력
- 함수 func3()에서는 지역 변수 a를 정의하지 않고 a를 출력
- 전역 변수 a를 가져와서 출력
- 전역 변수는 코드 내 어디서나 사용할 수 있으므로 함수안과 함수 밖에서 모두 이용할 수 있음
- 함수 func4() 에서도 지역 변수 a를 정의하지 않고 a를 이용
- 함수 func4()에서는 전역 변수 a의 값을 변경
- 함수 안에서 전역 변수의 내용을 변경할 때는 'global 전역 변수명'을 먼저 선언해야 전역 변수의 내용을 변경할 수 있음

- 정의한 각각 함수를 호출해 각 함수에서 어떤 a가 사용

In [ ]:

```
func1() #함수 func1() 호출
func2() #함수 func2() 호출
print("전역 변수 a =", a) # 전역 변수 출력
```

- 함수 안에서 정의된 지역 변수는 같은 이름의 전역 변수보다 먼저 참조되고 함수가 끝나면 그 효력이 즉시 사라지는 것을 알 수 있음
- 함수를 호출하지 않고 변수를 이용하면 전 역 변수를 참조하는 것을 확인할 수 있음

- 함수 안에서 전역 변수 이용하는 함수 func3()과 func4()를 호출

In [ ]:

```
func3() #함수 func3() 호출
func4() #함수 func4() 호출
func3() #함수 func3() 호출
```

- 함수 호출 결과를 비교해 보면 함수 func4() 호출로 전역 변수 a의 값이 변경됐음을 알 수 있음

7.3 람다(lambda) 함수

- 파이썬에는 한 줄로 함수를 표현하는 람다(lambda) 함수가 있음
- 람다 함수는 구성이 단순해 간단한 연산을 하는 데 종종 사용됨

- 람다 함수의 기본 구조
- lambda〈인자〉: <인자 활용 수행 코드>
- 람다 함수는 <인자>를 전달하면 <인자 활용 수행 코드>를 수행한 후 결과를 바로 반환
- <인자>는 콤마(,)로 구분해 여러 개를 사용할 수 있음

- 람다를 사용할 때는 람다 함수 전체를 소괄호로 감싸고 그 다음에 별도의 소괄호에 인자를 씀
- (lambda〈인자〉 : 〈인자 활용 수행 코드〉) (<인자>)

- 람다 함수를 다른 변수에 할당하고 이 변수를 함수명처럼 이용해 람다 함수를 호출

- 이때 정의한〈인자〉도 함께 입력
- 람다 함수를 변수에 할당할 때는 람다 함수 전체를 소괄호로 감싸지 않아도 됨
- lambda_function = lambda <인자> : 〈인자 활용 수행 코드〉 lambda_function (<인자〉)

- 람다 함수를 이용해 입력된 수의 제곱을 반환하는 코드
- 입력 인자와 연산 코드로 구성된 람다 함수를 정의하고 인자로 3을 입력

In [ ]:

```
(lambda x : x**2) (3)
```

- 제곱을 구하는 람다 함수에 숫자 3을 입력해 원하는 결과를 얻었지만
- 제곱을 계속 수행해야 한다면 람다 함수를 반복해서 써야 해서 번거로움

- 람다 함수를 변수에 할당한 후에 인자를 입력해서 호출

In [ ]:

```
mySquare = lambda x : x**2
mySquare(2)
```

- 람다 함수를 변수로 할당했으므로 이제 다른 인자의 제곱 값을 얻고 싶다면 인자만 바꿔서 람다 함수를 호출

In [ ]:

```
mySquare(5)
```

- 여러 개의 인자를 입력받아 연산 결과를 반환하는 람다 함수

- x, y, z라는 세 개의 인자를 입력받아 '2*x* + 3y + z' 연산의 결과를 반환
- 정의한 람다 함수를 변수에 할당하고 세 개의 인자를 입력해서 람다 함수를 호출

In [ ]:

```
mySimpleFunc = lambda x,y,z : 2*x + 3*y + z
mySimpleFunc(1,2,3)
```

7.4 유용한 내장 함수

- 파이썬의 다양한 내장 함수를 활용하면 함수를 새로 만들지 않고 원하는 기능을 수행할 수 있음

형 변환 함수

- 코드를 작성하다 보면 데이터의 형태(타입)를 변환해야 하는 경우가 발생
- 파이씬의 내장 함수를 이용해 데이터의 형태를 변환하는 방법

정수형으로 변환

- 내장 함수 int()는 실수나 문자열(정수 표시) 데이터를 정수로 변환
- 실수 데이터를 정수로 변환
- 실수의 경우 소수점 이하는 버리는 방식으로 정수로 변환

In [ ]:

```
[int(0.123), int(3.5123456), int(-1.312367)]
```

- 문자열의 경우는 문자열이 정수를 표시할 때만 정수로 변환
- 문자열이 실수를 표시하거나 숫자 외의 문자가 포함돼 있는 경우 int()로 변환을 시도하면 오류가 발생

In [ ]:

```
[int('1234'), int('5678'), int('-9012')]
```

실수형으로 변환

- 내장 함수 float()는 정수나 문자열(정수 및 실수 표시) 데이터를 실수로 변환
- 정수 데이터를 실수로 변환

In [ ]:

```
[float(0), float(123), float(-567)]
```

- 실수를 표시한 문자열을 실수로 변환
- 문자열에 정수나 실수를 표시하는 문자 외 의 다른 문자가 있으면 float()로 변환할 때 오류가 발생

In [ ]:

```
[float('10'), float('0.123'), float('-567.89')]
```

문자형으로 변환

- 내장 함수 str()은 정수나 실수 데이터를 문자열로 변환
- 정수 데이터를 문자열 데이터로 변환

In [ ]:

```
[str(123), str(459678), str(-987)]
```

- 실수 데이터를 문자열 데이터로 변환

In [ ]:

```
[str(0.123), str(345.678), str(-5.987)]
```

리스트, 튜플, 세트형으로 변환

- 리스트, 튜플, 세트 데이터는특성이 비슷해서 list(), tuple(), set() 함수처럼 서로 변환할수 있음

- 리스트, 튜플, 세트 데이터끼리 변환
- 리스트, 튜플, 세트 데이터를 만듬

In [ ]:

```
list_data = ['abc', 1, 2, 'def']
tuple_data = ('abc', 1, 2, 'def')
set_data = {'abc', 1, 2, 'def'}
```

- 각 데이터 타입은 type() 함수로 확인

In [ ]:

```
[type(list_data), type(tuple_data), type(set_data)]
```

- list() 함수를 이용해 튜플 데이터와 세트 데이터를 리스트로 변환

In [ ]:

```
print("리스트로 변환: ", list(tuple_data), list(set_data))
```

- tuple() 함수로 리스트 데이터와 세트 데이터를 튜플로 변환

In [ ]:

```
print("튜플로 변환:", tuple(list_data), tuple(set_data))
```

- set() 함수를 이용해 리스트 데이터와 튜플 데이터를 세트로 변환

In [ ]:

```
print("세트로 변환:", set(list_data), set(tuple_data))
```

bool 함수

내장 함수 bool()은 True 혹은 False의 결과를 반환

숫자를 인자로 bool 함수 호출

- bool 함수의 인자가 숫자인 경우
- 숫자 0이면 False, 0 이외의 숫자 <양의 정수 , 음의 정수, 양의 실수, 음의 실수> 이면 True를 반환

- 숫자 0을 인자로 삼아 bool() 함수를 호출하면 False를 반환

In [ ]:

```
bool(0) # 인자: 숫자 0
```

0 이외의 숫자를 인자로 삼아 bool()함수를 호출하면 모두 True를 반환

In [ ]:

```
bool(1) # 인자: 양의 정수
```

In [ ]:

```
bool(-10) # 인자: 음의 정수
```

In [ ]:

```
bool(5.12) # 인자: 양의 실수
```

In [ ]:

```
bool(-3.26) # 인자: 음의 실수
```

문자열을 인자로 bool 함수 호출

- bool() 함수의 인자가 문자열인 경우 문자열이 있으면 True를 반환하고 없으면 False를 반환
- bool()함수를 어떤 문자열이 빈 문자열인지 아닌지를 검사하는 데 이용할 수 있음
- 문자열에서 공백은 공백 문자열이 있는 것이고 빈 문자열('')은 문자열이 없는 것
- 파이썬에서 None 은 아무것도 없는 것으로 간주

- 함수 bool () 의 인자로 문자열을 사용

In [ ]:

```
bool('a') # 인자: 문자열 'a'
```

In [ ]:

```
bool(' ') # 인자: 빈 문자열(공백)
```

In [ ]:

```
bool('') # 인자: 문자열 없음
```

In [ ]:

```
bool(None) #인자: None
```

리스트, 튜플, 세트를 인자로 bool 함수 호출

- bool()함수는 리스트, 튜플, 세트를 인자로 호출할 수도 있음
- 문자열과 마찬가지로 항목이 있으면 True, 없으면 False를 반환
- bool() 함수는 리스트 , 튜플, 세트 데이터에서 항목이 있는지 없는지 검사할 때 유용하게 이용할수 있음

- 데이터 타입이 리스트이지만 리스트에 항목은 아무것도 없는 빈 리스트를 인자로 bool() 함수를 호출

In [ ]:

```
myFriends = []
bool(myFriends) # 인자: 항목이 없는 빈 리스트
```

- 항목이 있는 리스트를 인자로 bool() 함수를 호출하면 True를 반환

In [ ]:

```
myFriends = ['James', 'Robert', 'Lisa', 'Mary']
bool(myFriends) # 인자: 항목이 있는 리스트
```

- 튜플에 대해서도 항목이 없는 튜플과 있는 튜플을 인자로 bool()함수를 호출하면
- 각각 False 와 True 를 반환

In [ ]:

```
myNum = ()
bool(myNum) # 인자: 항목이 없는 빈 튜플
```

In [ ]:

```
myNum = (1,2,3)
bool(myNum) # 인자: 항목이 있는 튜플
```

- 세트에 대해서도 마찬가지로 항목 유무에 따라 각각 False와 True를 반환

In [ ]:

```
mySetA = {}
bool(mySetA) # 인자: 항목이 없는 빈 세트
```

In [ ]:

```
mySetA = {10,20,30}
bool(mySetA) # 인자: 항목이 있는 세트
```

bool 함수의 활용

- print_name() 함수 는 name 인자에 문자열이 있으면 이름을 출력하고, 없으면 입력된 문자열이 없다고 출력

In [ ]:

```
def print_name(name):
    if bool(name):
        print("입력된 이름:", name)
    else:
        print("입력된 이름이 없습니다.")
```

- print_name() 함수를 호출할 때 인자로 문자열이 있는 경우와 없는 경우에 각각 출력 결과

In [ ]:

```
print_name("James")
```

In [ ]:

```
print_name("")
```

최소값과 최대값을 구하는 함수

- 데이터에서 최소값 혹은 최대값을 구할 때 내장 함수 min()과 max()를 이용
- 내장 함수 min()과 max()는 리스트, 튜플, 세트의 항목이나 문자열 중에서 각각 최소값과 최대값을 반환

- 숫자를 포함한 리스트에서 최소값과 최대값을 구하기

In [ ]:

```
myNum = [10, 5, 12, 0, 3.5, 99.5, 42]
[min(myNum), max(myNum)]
```

- 숫자뿐만 아니라 문자열에 대해서도 최소값과 최대값을 구할 수 있음
- 문자열을 입력해 문자열에서 최소값과 최대값을 구하기

In [ ]:

```
myStr = 'zxyabc'
[min(myStr), max(myStr)]
```

- 각각 튜플과 세트에서 최소값과 최대값을 구하기

In [ ]:

```
myNum = (10, 5, 12, 0, 3.5, 99.5, 42)
[min(myNum), max(myNum)]
```

In [ ]:

```
myNum = {"Abc", "abc", "bcd", "efg"}
[min(myNum), max(myNum)]
```

- 크기를 비교하고자 하는 항목이 문자열일 경우에는 첫 문자 먼저 비교하고
- 첫 문자가 같다면 두 번째 문자를 비교하는 식으로 전체를 비교함
- 로마자 알파벳의 경우 A ~ Z, a ~ z 순서대로 크기가 큼
- 문자열로 된 숫자와 로마자 알파벳을 비교했을 때 숫자가 더 작음

절대값과 전체 합을 구하는 함수

- 절대값은 숫자의 부호와 상관없이 숫자의 크기만을 나타냄
- 내장 함수 abs()로 숫자의 절대값을 구 할 수 있음

- abs()를 이용해 숫자의 절대값을 구하기

In [ ]:

```
[abs(10), abs(-10)]
```

In [ ]:

```
[abs(2.45), abs(-2.45)]
```

- 내장 함수 sum()은 리스트, 튜플, 세트 데이터의 모든 항목을 더해 전체 합을 결과값으로 반환

- sum()을 이용해 리스트 데이터의 모든 항목을 더하기
- 튜플과 세트 데이터에 대해서도 sum() 함수를 적용해 모든 항목의 합을 구할 수 있음

In [ ]:

```
sumList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum(sumList)
```

항목의 개수를 구하는 함수

- 문자열, 리스트, 튜플, 딕셔너리 안에 있는 항목의 개수를 알아야 할 때
- 내장 함수 len()은 항목의 개수(데이터의 길이)를 반환

- 내장 함수 len()으로 문자열, 리스트, 튜플, 딕셔너리 데이터의 길이를 구하기

In [ ]:

```
len("ab cd") # 문자열
```

In [ ]:

```
len([1, 2, 3, 4, 5, 6, 7, 8]) # 리스트
```

In [ ]:

```
len((1, 2, 3, 4, 5)) # 튜플
```

In [ ]:

```
len({'a', 'b', 'c', 'd'}) # 세트
```

In [ ]:

```
len({1:"Thomas", 2:"Edward", 3:"Henry"}) # 딕셔너리
```

- len() 함수를 이용하면 문자열은 공백을 포함한 문자 길이(개수)를 구할 수 있고 리스트, 튜플, 딕셔너리는 항목의 길이(개수)를 구할 수 있음

내장 함수의 활용

- 시험 점수가 입력된 리스트가 있다고 할 때 sum()과 len()을 이용해 데이터 항목의 총합과 길이를 쉽게 구할 수 있음
- 이를 이용하면 평균값도 쉽게 구할 수 있음
- sum()과 len()을 이용하지 않고 총점과 평균값을 구하기

In [ ]:

```
scores = [90, 80, 95, 85] # 과목별 시험 점수

score_sum = 0                    # 총점 계산을 위한 초깃값 설정
subject_num = 0                  # 과목수 계산을 위한 초깃값 설정
for score in scores:
    score_sum = score_sum + score # 과목별 점수 모두 더하기
    subject_num = subject_num + 1 # 과목수 계산

average = score_sum / subject_num # 평균(총점 / 과목수) 구하기

print("총점:{0}, 평균:{1}".format(score_sum,average))
```

- 총점과 과목 수를 구하려고 for 문을 이용했고 평균을 구하려고 총점을 과목 수로 나웠음

- 같은 작업을 sum()과 len()을 이용해 수행

In [ ]:

```
scores = [90, 80, 95, 85] # 과목별 시험 점수

print("총점:{0}, 평균:{1}".format(sum(scores), sum(scores)/len(scores)))
```

- sum()과 len()을 이용하면 for 문을 사용해 총점과 과목 수를 구하지 않아도 되고
- 총점과 과목 수 계산을 위한 변수도 필요 없어서 편리하게 총점과 평균값을 구할 수 있음

- 내장 함수 min()과 max()를 이용하함 시험 점수 중 최고점과 최하점도 손쉽게 구함

In [ ]:

```
print("최하 점수:{0}, 최고 점수:{1}".format(min(scores), max(scores)))
```

7.5 정리

- 함수를 정의하고 호출하는 방법
- 인자나 반환 값의 유무에 따라 함수를 작성하는 방법과 호출하는 방법
- 지역 변수와 전역 변수의 유효 범위에 대한 개념과 한 줄짜리 함수인 람다 함수
- 유용한 내장 함 수에 대해 알아보고 활용