

강의교안 이용 안내

- 본 강의교안의 저작권은 도서출판 홍릉에 있습니다.
- 불법한 PDF 파일이 사용되지 않도록 지도 바랍니다.
- 이 자료를 무단으로 전제하거나 배포할 경우 저작권법 136조에 의거하여 벌금에 처할 수 있고 이를 병과(併科)할 수도 있습니다.

단원 06

사용자 정의 함수

인공지능소프트웨어학과

강환수 교수



Section 1. 함수 정의와 활용

-



함수 개요

- 함수(function)
 - 특정 기능을 수행하는 코드 블록
 - 스마트폰, 믹서기, 커피머신처럼
 - 특정한 기능을 독립적으로 수행할 수 있도록 코드를 묶어 놓은 단위
- 내장 함수(built-in functions)
 - 다양한 내장 함수
 - `print()`, `input()`, `str()`, `int()`, `float()`, `len()`
 - 내장 함수 확인
 - `dir(__builtins__)`
 - `eval()` 함수
 - 문자열 형태의 Python 표현식을 실행
 - `eval('3.14 * 4.78**2')`
- 사용자 정의 함수(user-defined functions)
 - 우리(개발자)가 직접 만든 함수

함수 호출과 실행 순서

- 함수는 함수 정의만으로는 실행되지 않음
 - 함수 정의는 함수를 사용하기 위해 메모리에 저장하는 방법
 - 함수 호출에 의해 비로소 실행
- 실행 순서
 - 함수를 호출하면 함수 정의 부분으로 이동하여 코드를 실행
 - 함수 실행 후에는 다시 호출한 부분으로 돌아와 다음 코드를 실행
 - 함수 호출 시 함수 내부의 코드가 실행되고 결과 값이 반환될 수 있음

```

1 def print_message():
2     print("함수 내에서 메시지 출력")
3
4 print("함수 호출 전")
5 print_message()
6 print("함수 호출 후")

```

결과

.....

함수 호출 전

함수 내에서 메시지 출력

함수 호출 후

.....

함수 이름과 변수

- 함수 이름은 함수를 호출할 때 사용하는 식별자
- 함수 이름은 변수에 할당하여 동일한 기능을 하는 함수처럼 사용 가능
 - id() 함수를 사용하면 변수 또는 함수 객체의 메모리 주소를 확인
 - is 연산자를 사용하면 두 변수가 동일한 객체를 참조하는지 확인

```
>>> def add(x, y): # x와 y라는 매개 변수
...     return x + y # x와 y의 합을 반환
...
>>> add(3, 5)
8
>>> f = add
>>> f(3, 5)
8
>>> id(add)
2591698276640
>>> id(f)
2591698276640
>>> f, add
(<function add at 0x0000025B6D415120>, <function add at 0x0000025B6D415120>)
>>> add is f
True
```

지역과 전역 변수

- 지역 변수(local var)
 - 함수 내부에서 정의되고 함수 내부에서만 사용 가능한 변수
 - 함수가 실행될 때 메모리에 생성되며, 함수 실행이 끝나면 소멸됨
- 전역 변수(global var)
 - 함수 외부에서 정의되고 프로그램 전체에서 사용 가능한 변수
 - 프로그램 시작 시 생성되어 프로그램 종료 시까지 유지
- 함수 내부에서 지역 변수와 전역 변수 이름이 같을 경우, 지역 변수가 우선됨

```
# %% 지역 변수와 전역 변수
## 지역 변수와 전역 변수의 예시
def func():
    b = 2 # 지역 변수 b에 2를 할당
    print(b) # 지역 변수 b의 값을 출력, 결과: 2

print(b) # 지역 변수 b의 값을 출력, 오류: NameError: name 'b' is not defined
```

내장함수 locals() globals()

- **locals() 함수**
 - 현재 스코프의 지역 변수 정보를 확인할 수 있음
- **globals() 함수**
 - 전역 변수 정보를 확인할 수 있음

```
>>> def double(x): # double이라는 이름의 함수를 정의, x라는 매개변수를 받음.
...     a = 2
...     print(locals())
...     return x * a # x의 두 배를 반환
...
>>> b = 10
>>> double(b)
{'x': 10, 'a': 2}
20
```


지역 변수와 전역 변수의 이름이 같은 경우

- 함수 내부에서 대입이 있는 변수는 지역 변수로 인식

```
# %% 코딩 예제 06-02localvar.py
def addone():
    '''함수에서 대입에 사용되는 변수는 지역변수'''
    i = 30 # 지역 변수 생성
    i += 1 # 지역 변수 수정
    print(f'\t지역 변수 i: {i}') # 지역 변수 참조

i = 20 # 전역 변수, 이름 i가 위의 지역 변수와 같으나 다른 변수
print(f'전역 변수 i: {i}') # 전역 변수 i 출력
addone() # 함수 호출
print(f'전역 변수 i: {i}') # 전역 변수 i 출력
```

- 함수 내부에서 대입이 없는 변수는 전역 변수로 인식

함수에서 global 문장으로 변수를 전역 변수로 지정

- 키워드 global로 변수 선언

- 함수 내부에서 전역 변수를 직접 수정

```
# %% 코딩 예제 06-04keywordglobal.py 키워드 global
def addone():
    '''명시적으로 변수를 전역변수로 지정'''
    global i # i를 명시적으로 전역 변수로 인식
    print(f'\t함수 addone() 내부, 전역 변수 i 읽기: i + 1: {i + 1}')
    i += 1 # 전역 변수 i 수정

i = 20 # 전역 변수
print(f'함수 addone() 호출 이전: i = {i}')
addone()
print(f'함수 addone() 호출 이후: i = {i}')
```

함수의 기본값 사용

- 함수의 매개변수에 기본값(default value)을 지정
 - 기본값이 있는 매개변수는 함수 호출 시 생략 가능
 - 기본값이 없는 매개변수는 반드시 값을 전달해야 함
 - 기본값이 있는 매개변수는 기본값이 없는 매개변수 뒤에 위치해야 함
- 함수 호출 시 함수의 인자 개수를 줄일 수 있음

```
>>> # message 매개변수에는 "Hello"라는 기본값을 지정
>>> def greet(name, message="안녕!"):
...     print(name, message) # name과 message를 출력
...

>>> greet("가희") # message 인자는 생략, 기본값인 '안녕!'으로 출력
가희 안녕!
```

기본 인자가 가변 객체인 경우, 기본값은 오직 한 번만 사용

- 인자의 기본값이 가변 객체인 경우, 첫 함수 호출 시 오직 한 번만 지정 (아래 왼쪽 코드)

- 기본 인자로 수정가능한(가변) 객체(리스트, 사전, 집합 등)를 사용하면,
 - 이후 호출에서 같은 객체가 재사용됨

```
# %%
def default_arg(a, obj=[]):
    |   obj.append(a)
    |   return obj

print(default_arg(1))
print(default_arg(2))
print(default_arg(3))

✓ 0.0s
```

```
[1]
[1, 2]
[1, 2, 3]
```

```
>>> def default_none(a, obj=None):
...     if obj is None:
...         obj = []
...     obj.append(a)
...     return obj
...
>>> default_none(1)
[1]
>>> default_none(2)
[2]
>>> default_none(3)
[3]
>>> default_none(3, [10, 20])
[10, 20, 3]
>>>
```

- 연속된 호출 간에 기본값이 공유되지 않기를 원한다면, (위 오른쪽 코드)

- 함수 헤더에서 obj=None을 지정해 매번 새로운 목록을 만들어 사용

매개변수로 함수 사용

- 매개변수로 함수를 정의하면 필요한 함수를 인자로 호출 가능

```
>>> def my_apply(func, x, y):  
...     return func(x, y)  
...  
>>> def my_add(a, b):  
...     return a + b  
...  
>>> def my_mult(a, b):  
...     return a * b  
...  
>>> my_apply(my_add, 3, 5)  
8  
>>> my_apply(my_mult, 3, 5)  
15  
>>> my_apply(lambda a, b: a - b, 10, 5)  
5  
>>> my_apply(lambda a, b: a * b, 10, 5)  
50
```

함수 내부에서 함수를 만들어 반환

심화과정

- 함수 `inc(n)`
 - 함수 `add(x)`를 만들어 반환
 - 내부 함수 `add`는 외부 함수 `inc`의 매개변수 `n`을 참조 가능
- 클로저(closure)
 - 함수 내부에서 외부 변수 `n`을 사용하는 함수
 - 다음 내부 함수 `add()`와 람다 함수는 클로저

```
>>> def inc(n):
...     def add(x):
...         return x + n
...     return add
...
>>> incbase100 = inc(100)
>>> incbase100(10)
110
>>> def my_inc(n):
...     return lambda x: x + n
...
>>> incbase50 = my_inc(50)
>>> incbase50(20)
70
>>>
```

내장 함수 vars()와 globals()

• 함수 vars()

- 인자가 없으면, 지역변수를 딕셔너리 형식으로 반환
 - `__main__`에서의 지역변수는 전역변수가 되므로 전역변수도 확인 가능

• 함수 globals()

- 전역 변수 딕셔너리 형식으로 반환

```
>>> vars()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000001679F66B950>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'sys': <module 'sys' (built-in)>, 'original_ps1': '>>> ', 'REPLHooks': <class '__main__.REPLHooks'>, 'get_last_command': <function get_last_command at 0x000001679FA372E0>, 'PS1': <class '__main__.PS1'>}
```

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000001679F66B950>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'sys': <module 'sys' (built-in)>, 'original_ps1': '>>> ', 'REPLHooks': <class '__main__.REPLHooks'>, 'get_last_command': <function get_last_command at 0x000001679FA372E0>, 'PS1': <class '__main__.PS1'>}
```

```
>>> def calc(x, y, z):
...     return x + y - z
...
>>> calc(3, 4, 2)
5
```

```
>>> vars()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000001679F66B950>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'sys': <module 'sys' (built-in)>, 'original_ps1': '>>> ', 'REPLHooks': <class '__main__.REPLHooks'>, 'get_last_command': <function get_last_command at 0x000001679FA372E0>, 'PS1': <class '__main__.PS1'>, 'calc': <function calc at 0x000001679FA37560>}
```

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000001679F66B950>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'sys': <module 'sys' (built-in)>, 'original_ps1': '>>> ', 'REPLHooks': <class '__main__.REPLHooks'>, 'get_last_command': <function get_last_command at 0x000001679FA372E0>, 'PS1': <class '__main__.PS1'>, 'calc': <function calc at 0x000001679FA37560>}
```

문서화 문자열(docstring)

• 함수의 설명과 사용법을 작성

- 함수의 문서화 내용인 문자열
- 쌍따옴표 세 개(“”” 또는 ‘’)로 둘러싸인 문자열
 - 함수의 첫 번째 문장에 위치
 - 주로 큰따옴표 세 개를 사용
- 함수의 목적, 매개변수, 반환값, 예외, 예제 등을 포함
 - 문서 이해에 필요하다면 중간 중간에 빈 줄을 삽입

```
>>> # 함수 add() 구현, docstring 예시
>>> def add(x, y):
...     """두 수의 합을 반환하는 함수입니다.
...
...     매개변수:
...         x (int): 첫 번째 숫자입니다.
...         y (int): 두 번째 숫자입니다.
...
...     반환 값:
...         int: x와 y의 합입니다.
...
...     예외:
...         TypeError: x나 y가 정수가 아닌 경우에 발생합니다.
...
...     예제:
...         >>> add(3, 5)
...         8
...     """
...     if not isinstance(x, int) or not isinstance(y, int):
...         raise TypeError("x와 y는 정수여야 합니다.")
...     return x + y
... 
```

docstring

문서화 문자열 참조 방법

`help(fun_name)` 또는 `func_name.__doc__`

- 문서화 문자열은 파이썬이 제공하는 `help()` 함수나 `doc` 속성을 통해 접근 가능

```
>>> help(add) # add 함수에 대한 도움말을 출력
Help on function add in module __main__:

add(x, y)
    두 수의 합을 반환하는 함수입니다.
    매개변수:
        x (int): 첫 번째 숫자입니다.
        y (int): 두 번째 숫자입니다.
    반환 값:
        int: x와 y의 합입니다.
    예외:
        TypeError: x나 y가 정수가 아닌 경우에 발생합니다.
    예제:
        >>> add(3, 5)
        8
```

```
>>> print(add.__doc__)
두 수의 합을 반환하는 함수입니다.
매개변수:
    x (int): 첫 번째 숫자입니다.
    y (int): 두 번째 숫자입니다.
반환 값:
    int: x와 y의 합입니다.
예외:
    TypeError: x나 y가 정수가 아닌 경우에 발생합니다.
예제:
    >>> add(3, 5)
    8
```

Section 2. 위치 인자와 키워드 인자

-

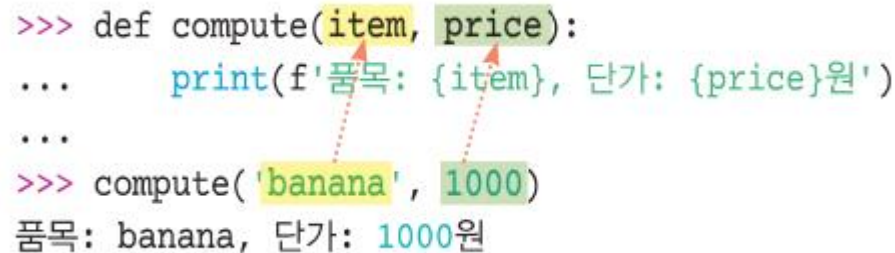


함수 호출 시 인자를 전달하는 두 가지 방법

• 위치 인자 (Positional Argument)

- 함수 정의 시 매개변수 순서대로 인자를 전달하는 방식
- 함수 호출 시 괄호 안의 인자 순서가 매개변수 순서와 일치해야 함

```
>>> def compute(item, price):  
...     print(f'품목: {item}, 단가: {price}원')  
...  
>>> compute('banana', 1000)  
품목: banana, 단가: 1000원
```



• 키워드 인자 (Keyword Argument)

- 매개변수 이름을 지정하여 인자를 전달하는 방식
- 매개변수이름=값 형태로 인자를 전달하며, 순서에 상관없이 인자를 지정할 수 있음

```
>>> compute(item='포도', price=2000)  
품목: 포도, 단가: 2000원  
  
>>> compute(price=3000, item='배')  
품목: 배, 단가: 3000원
```

키워드 인자와 위치 인자 혼합 사용

- 위치 인자 우선
 - 함수 호출 시 위치 인자는 항상 먼저 사용해야 함
- 위치 인자가 키워드 인자 뒤에 오면 오류 발생
 - `compute(item='귤', 500)` (오류!)
 - 해결: 위치 인자를 먼저 쓰고 키워드 인자를 나중에 사용해야 함
 - `>>> compute('귤', price=500)`
 - 품목: 귤, 단가: 500원
- 중복 저장 오류
 - 인자로 item에 '사과'를 저장한 후, 다시 키워드 인자로 중복되게 item에 '귤'을 저장
 - 키워드 인자를 사용하면 함수 인수를 지정하는 순서에 유연성을 주지만 인수 개수는 여전히 일치

```
>>> compute('사과', item='귤')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: compute() got multiple values for argument 'item'
```

가변 인자 *args

함수에 임의의 개수의 인자를 전달할 수 있게 해주는 방법

• *args

- 함수 정의 시 매개변수 이름 앞에 *를 붙여 사용
 - 튜플 형태로 인자를 받음
- 이름: args
 - 관례적으로 사용하는 이름이며, 다른 이름을 사용해도 무방
- print() 함수
 - print() 함수가 가변 인자를 사용하는 대표적인 예
- 활용
 - 여러 개의 인자를 묶어서 함수에 전달하거나, 미리 정의되지 않은 개수의 인자를 처리할 때 유용

```
>>> help(print)
```

```
Help on built-in function print in module builtins:
```

```
print(*args, sep=' ', end='\n', file=None, flush=False)
```

```
Prints the values to a stream, or to sys.stdout by default.
```

```
sep
```

```
string inserted between values, default a space.
```

```
end
```

```
string appended after the last value, default a newline.
```

네 개의 매개변수는 기본값이 지정되어 함수 호출 시 인자가 생략될 수 있다.

가변 인자 *args 활용

- **sum_numbers() 함수 정의:**

```
>>> def sum_numbers(*args):  
...     result = 0  
...     for num in args:  
...         result += num  
...     return result  
...
```

- 호출

- **sum_numbers(1, 2, 3, 4, 5)**
- **결과: 15**

- **튜플 언패킹 ***

- 함수 호출 시 튜플이나 리스트 앞에 * 연산자를 붙여서 각각의 원소를 개별 인자로 전달 가능

```
>>> lst = [3, 4, 5]  
>>> sum_numbers(*[10, 20, 30], *lst)  
72
```

가변 키워드 인자 **kwargs

함수에 임의의 개수의 키워드 인자를 전달할 수 있게 해주는 방법

- ****kwargs**

- 함수 정의 시 매개변수 이름 앞에 **를 붙여 사용
 - 딕셔너리 형태로 인자를 받음
 - 키워드 인자를 딕셔너리 형태로 묶어서 전달하는 역할
- **kwargs
 - 관례적으로 사용

- **활용**

- 함수에 미리 정의되지 않은 키워드 인자를 처리해야 할 경우 유용

```
>>> def personal_info(**kwargs):
...     for kw, arg in kwargs.items(): # kwargs 딕셔너리를 순회하며
...         print(kw, ': ', arg, sep='') # 각 키와 값 쌍을 출력한다.
...
>>> personal_info(name='홍길동', age=30, address='서울시 용산구 이촌동')
name: 홍길동
age: 30
address: 서울시 용산구 이촌동
```

딕셔너리 언패킹

- ** 연산자를 사용

- 딕셔너리의 키-값 쌍을 키워드 인자로 전달 가능

```
>>> info = dict(phone='010-1234-5678', email='hong@naver.com')
>>> personal_info(**info, **dict(name='김한길', age=30))
phone: 010-1234-5678
email: hong@naver.com
name: 김한길
age: 30
```


일반 인자와 가변 인자 *args를 함께 사용

- 함수 `display_info(name, *args)`

- 일반 위치 인자 `name` 한 개와 가변 인자 `*args` 한 개를 갖는 함수

```
>>> def display_info(name, *args):  
...     print("이름:", name)  
...     print("추가 정보:")  
...     for info in args:  
...         print(info)  
...
```

```
>>> # 함수 호출: 일반 인자와 다수의 추가 정보 전달
```

```
>>> display_info("Alice", "나이: 30세", "도시: 뉴욕", "직업: 개발자")
```

```
이름: Alice
```

```
추가 정보:
```

가변 인자 *args 와 가변 키워드 인자 **kwargs 함께 사용

- 함수 정의 시 *args는 **kwargs보다 먼저 정의되어야 함

6-7 코딩

06-07-argskwargs.py | 가변 인자와 키워드 가변 인자 함수 구현

난이도 응용

```
1      # 주문 정보를 저장하는 사전
2      orders = []
3
4      # 함수를 사용하여 주문을 추가하는 기능
5      def add_order(item, *options, **special_requests):
6          global orders
```

Section 3. 함수 인자의 다양한 활용

-



Unpacking(압축 풀기) 연산자 *

- 함수 정의 시, 인자의 개수와 호출 시 전달되는 인자의 개수가 일치해야 함

- 튜플, 리스트, 셋 등의 여러 항목을 함수 인자로 전달할 때 사용

```
>>> def f(x, y, z):
...     print(f'x = {x}, y = {y}, z = {z}')
...
>>> f(*(1, 2, 3))
x = 1, y = 2, z = 3
```

- 함수 호출 시 *를 사용하여 튜플이나 리스트의 각 항목을 개별 인자로 전달

- *를 사용하면 여러 개의 값을 묶어서 하나의 인자로 전달 가능
- 결국 `f(*(1, 2, 3))`
 - `f(1, 2, 3)`과 동일

- 리스트 병합

- `merged = [*lst1, *lst2, *lst3]`

```
>>> lst1= [1, 2]
>>> lst2= [3, 4]
>>> lst3= [5, 6]
>>> merged = lst1 + lst2 + lst3
>>> merged
[1, 2, 3, 4, 5, 6]
```

- 문자열 분해

- `s1 = ['Python']` 결과는 ['P', 'y', 't', 'h', 'o', 'n']

대입 연산자 좌변에 *를 사용할 경우

- 대입 연산에서의 언패킹(unpacking)에 의해 = 왼쪽의 변수 수만큼 분할이 가능

- `*a, b, c = lst`

```
>>> lst = [1, 2, 3, 4, 5, 6]
>>> a, *b, c = lst
>>> a, b, c
(1, [2, 3, 4, 5], 6)
>>> b
[2, 3, 4, 5]
>>> *a, b = 10, 20, 30, 40
>>> a
[10, 20, 30]
>>> b
40
```

- `*a = 10, 20, 30, 40`

- 오류 발생

- `*a, = 10, 20, 30, 40`

```
>>> *a, = 10, 20, 30, 40
>>> a
[10, 20, 30, 40]
>>>
```

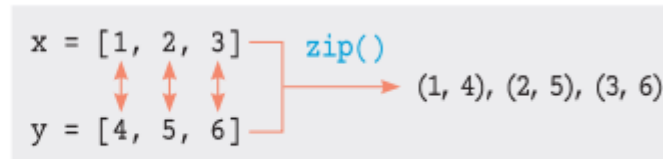
내장 함수 zip()과 압축 풀기 연산자 * 활용

• 내장 함수 zip()

- 여러 개의 반복가능한(iterable) 객체를 인자로 받고,
 - 각 객체가 담고 있는 원소를 튜플의 형태로 차례로 접근할 수 있는 반복자(iterator)인 zip 객체를 반환
 - 옷의 지퍼를 올리는 것처럼 인자인 두 객체에 있는 항목을 하나씩 차례로 짝을 지어 튜플인 항목을 생성

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> z = zip(x, y)
>>> z
<zip object at 0x0000021ABE53CDC0>
```

```
>>> lst = list(z)
>>> lst
[(1, 4), (2, 5), (3, 6)]
```



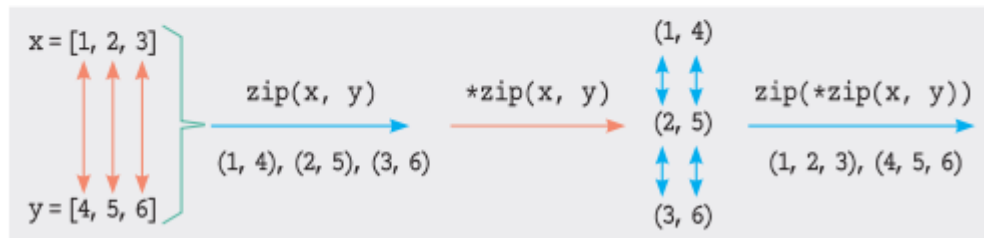
• 다음으로 x와 y의 항목이 들어있는 튜플을 얻음

```
>>> x2, y2 = zip(*lst)
>>> x2
(1, 2, 3)
>>> y2
(4, 5, 6)
```

문장 zip(*zip(x, y))

- 원래 2개의 x, y와 같은 항목을 갖는 튜플을 바로 얻을 수 있음

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> x2, y2 = zip(*zip(x, y))
>>> x2
(1, 2, 3)
>>> y2
(4, 5, 6)
```



▲ 그림 2 zip()과 압축 풀기 연산자 *

6-8 코딩

수정: 교재에 “표준입력 부분”이 없으니 다음 표준입력이 필요합니다!

- 실행 시, 다음 표준 입력하면 실행 됨
 - 품목을 콤마로 구분해서 입력한 후 enter
 - 다음, 품목에 대응되는 가격을 콤마로 구분해서 입력한 후 enter

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "d:/2025 0116목
구글드라이브 공유)/10. 개인집필/2025 [최신 파이썬 완전정복] 강의자료/py3 code/ch06/06-08unpacking.py"
```

구매한 상품 목록 (콤마로 구분): 우유, 과자, 물
 각 상품의 가격 (콤마로 구분): 2600, 3400, 1800

상품 목록: ['우유', '과자', '물']
 가격 목록: [2600, 3400, 1800]
 총 금액: 5200 원

2줄의 사용자 표준입력이 필요

사전 압축 풀기 연산자 **

• 함수 f() 정의

- 인자 C나 C#이 없으므로 오류가 발생

```
>>>> f(**{'C': 1971, 'C#': 2002}) # 오류, 인자의 이름이 다름
```

```
>>> def f(java, python):
...     print(f'java = {java}, y = {python}')
...
>>> f(1995, 1991)
java = 1995, y = 1991
>>> f(java=1995, python=1991)
java = 1995, y = 1991
>>> f(**{'java': 1995, 'python': 1991})
java = 1995, y = 1991
>>> f(**{'C': 1971, 'C#': 2002}) # 오류, 인자의 이름이 다름
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got an unexpected keyword argument 'C'
```

특수 매개변수 / *

- 가독성과 성능을 위해 인자 전달 방식을 제한하려면 특수 문자 /와 *을 사용
 - 함수 헤더에서 / 앞 부분 인자 [pos]
 - 위치 전용(position only) 매개변수
 - 다음으로 / 뒤와 * 앞의 중간 부분 인자 [pos_or_kwds]
 - 위치 또는 키워드 인자 방식의 일반 매개변수
 - 마지막으로 * 뒤의 잔여 부분 인자 [kwds]
 - 키워드 전용(keyword only) 매개변수

특수 매개변수 /, *

```
def fun_name( [pos], /, [pos_or_kwds], * , [kwds] )
```

- /와 *는 위치 전용(position only) 인자나 키워드 전용(keyword only) 인자를 함수에 전달할 수 있는지 여부를 정의
- /를 사용하여 위치 전용 인자와 일반(위치나 키워드로 전달) 인자로 전달할 수 있는 인자 사이의 경계를 정의
- *를 사용하여 일반(위치나 키워드로 전달) 인자와 키워드 전용 인자로 전달해야 하는 인수 사이의 경계를 정의
- /는 *는 모두 옵션으로 /는 *보다 앞에 위치
- /는 맨 앞에 올 수 없으며 *는 맨 뒤에 올 수 없음.

기호 /와 *의 구분

• 다음의 인자 방식으로 값을 전달

- 특수 매개변수 /는 맨 앞에 올 수 없음
- 반대로, 특수 매개변수 *는 맨 뒤에 올 수 없음

```
>>> def pos_kwd_function(pos1, pos2, /, arg, *, kwd1, kwd2):
...     print(f'위치 인자: pos1 = {pos1}, pos2 = {pos2}')
...     print(f'일반 인자: arg = {arg}')
...     print(f'키워드 인자: kwd1 = {kwd1}, kwd2 = {kwd2}')
... 
```

구분	위치 전용 인자		구분자	위치-키워드(일반) 인자	구분자	키워드 전용 인자	
함수 헤더	pos1,	pos2,	/,	arg,	*	kwd1,	kwd2
설명	인자의 순서대로 전달 하며 키워드로 매개 변수를 전달 불가능		호출 시 불필요	위치나 키워드 인자 전달 방식 모두 가능	호출 시 불필요	반드시 키워드 인자로 전달해야 하며, 인자의 순서인 위치로는 전달 불가능	
사용 가능	10	'java'		3.14 arg='ja'		kwd1=3	kwd2=10
사용 불가능	pos1=3	pos2=3,4		두 가지 방식 모두 가능		3	'py'

****kwargs와 /의 사용**

- 키워드 가변인자 ****kwargs** 뒤에는 더 이상 인자가 올 수 없으므로 *****도 사용할 수 없음

```
>>> def fun(n, **kwargs, *, kwd):  
File "<stdin>", line 1  
    def fun(n, **kwargs, *, kwd):  
                        ^  
SyntaxError: arguments cannot follow var-keyword argument
```

일반 인자와 키워드 인자 이름 충돌

• 주의

- 일반 인자 name과 name을 키로 가지는 **kwargs 사이에 잠재적인 충돌이 발생할 수 있기 때문

```
>>> def fun1(name, **kwargs):
...     return 'name' in kwargs
...

>>> fun1(30000, **{'name': '미수', 'age': 35})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun1() got multiple values for argument 'name'
```

• 해결 방법

- 앞의 name매개변수를 위치 전용 인자로 제한하면 해결
- name을 위치 인자로, 동시에 'name'을 키워드 인자의 키로 사용
 - 즉, 위치 전용 매개변수의 이름을 **kwargs에서 충돌 없이 사용

```
>>> def fun2(name, /, **kwargs):
...     return 'name' in kwargs
...

>>> fun2(30000, **{'name': '미희', 'age': 21})
True
```