

5장. 파이썬 날개 달기, 객체지향 프로그래밍

I. 클래스

II. 모듈

III. 패키지

IV. 예외 처리

V. 내장 함수

VI. 외장 함수

클래스(class)는 객체 지향 프로그래밍(OOP)에서 특정 객체를 생성하기 위해 변수와 메소드(함수)를 정의하는 일종의 틀(template)이다.

❖ 클래스는 도대체 왜 필요한가?

- 함수를 이용해 계산기의 '더하기' 기능을 구현한 예

```
#add1.py
result = 0

def add(num):
    global result
    result += num
    return result

print(add(3))
print(add(4))
```

- ✓ 계산된 결과값을 유지하기 위해서 result라는 전역 변수(global)를 사용했음
- ✓ 실행하면 예상한 대로 다음과 같은 결과값이 출력됨


```
3
7
```

- 한 프로그램에서 2대의 계산기가 필요한 상황이 발생하면 어떻게 해야 할까?
각 계산기는 각각의 결과값을 유지하기 위해 add 함수를 각각 따로 만들어야 함

```
#add2.py
result1 = 0
result2 = 0

def add1(num):
    global result1
    result1 += num
    return result1
def add2(num):
    global result2
    result2 += num
    return result2

print(add1(3))
print(add1(4))
print(add2(3))
print(add2(7))
```



계산기 1

계산기 2

- 똑같은 일을 하는 add1과 add2 함수를 만들었고, 결과값을 유지하면서 저장하는 전역 변수 result1, result2가 필요하게 되었음
- 결과값은 다음과 같이 출력됨

```
3
7
3
10
```

- ✓ 계산기 1의 결과값이 계산기 2에 영향을 끼치지 않음을 확인함
- ✓ 하지만 계산기가 3개, 5개, 10개로 더 많이 필요해진다면 어떻게 해야 할까?
- ✓ 그때마다 전역 변수와 함수를 추가할 것인가?
- ✓ 여기에 빼기나 곱하기 등의 기능을 추가해야 한다면 상황은 더 어려워질 것임

- 위와 같은 경우 클래스를 이용하면 간단하게 해결할 수 있음

```
#add3.py
class Calculator:
    def __init__(self):
        self.result = 0

    def add(self, num):
        self.result += num
        return self.result

cal1 = Calculator()
cal2 = Calculator()

print(cal1.add(3))
print(cal1.add(4))
print(cal2.add(3))
print(cal2.add(7))
```

- 실행하면 함수 2개를 사용했을 때와 동일한 결과가 출력됨

```
3
7
3
10
```

- ✓ Calculator 클래스로 만든 별개의 계산기 cal1, cal2(객체)는 각각의 역할을 수행함
 - ✓ 계산기(cal1, cal2)의 결과값도 각각 독립적인 값을 유지함
 - ✓ 클래스를 사용하면 계산기 대수가 늘어나더라도 객체를 생성하기만 하면 되기 때문에 함수를 사용하는 경우와 달리 매우 간단해짐
- Calculator 클래스에 빼기 기능 함수를 추가한 예

```
def sub(self, num)
    self.result -= num
    return self.result
```

❖ 클래스와 객체

- 클래스는 과자 틀과 비슷함
- 클래스(class)란 똑같은 무엇인가를 계속 만들어낼 수 있는 설계 도면이고(과자 틀), 객체(object)란 클래스로 만든 피조물(과자 틀을 사용해 만든 과자)을 뜻함

- 과자 틀 → 클래스(class)
- 과자 틀을 사용해 만든 과자 → 객체(object)



- 클래스의 간단한 예

```
>>> class Cookie:  
...     pass  
...  
>>>
```

- ✓ 위의 클래스는 아무런 기능도 갖고 있지 않은 껍질뿐인 클래스임
- ✓ 하지만 껍질뿐인 클래스도 객체를 생성하는 기능이 있음
- ✓ 객체는 클래스로 만들며, 1개의 클래스는 무수히 많은 객체를 만들어낼 수 있음

- 앞에서 만든 Cookie 클래스의 객체를 만드는 방법

```
>>> a = Cookie()
>>> b = Cookie()
```

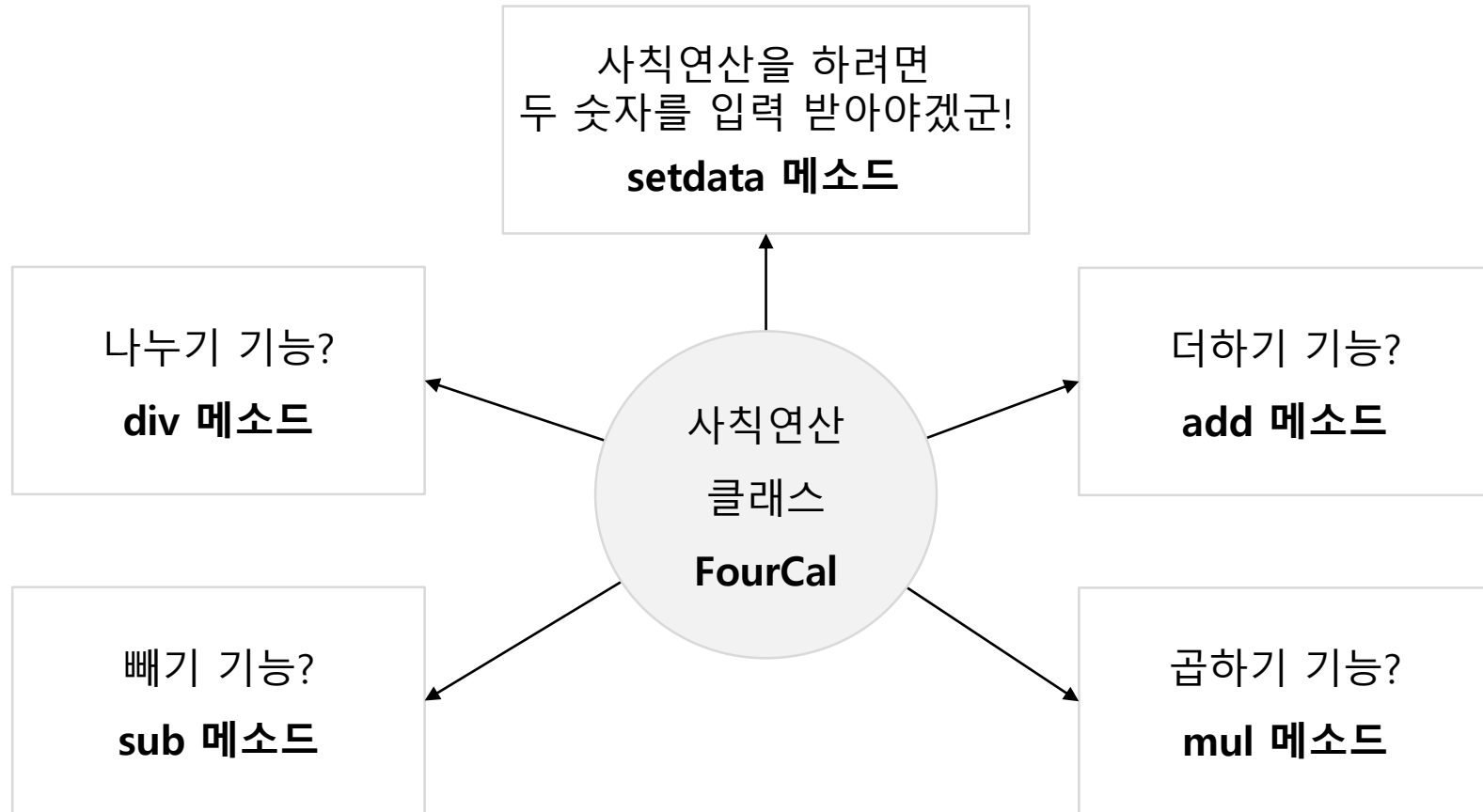
- ✓ Cookie()의 결과값을 돌려받은 a와 b가 객체임
- ✓ 함수를 사용해서 그 결과값을 돌려받는 모습과 비슷함

[객체와 인스턴스의 차이]

- ✓ 클래스로 만든 객체를 인스턴스라고도 함
- ✓ a = Cookie() 이렇게 만든 a는 객체임
- ✓ a 객체는 Cookie의 인스턴스임
- ✓ 즉 인스턴스라는 말은 특정 객체(a)가 어떤 클래스(Cookie)의 객체인지를 관계 위주로 설명할 때 사용됨
- ✓ a는 '인스턴스' 보다는 a는 '객체'라는 표현이 어울림
- ✓ a는 'Cookie의 객체' 보다는 a는 'Cookie의 인스턴스'라는 표현이 어울림

❖ 사칙연산 클래스 만들기

- 클래스를 어떻게 만들지 먼저 구상하기



- 사칙연산을 가능하게 하는 FourCal 클래스가 다음처럼 동작한다고 가정함

>>> a = FourCal()	← a = FourCal처럼 입력해서 a라는 객체를 만들
>>> a.setdata(4, 2)	← a.setdata(4, 2)처럼 입력해서 4와 2라는 숫자를 a에 지정해 줌
>>> print(a.add())	← a.add()를 수행하면 두 수를 합한 결과(4 + 2)를 돌려줌
6	
>>> print(a.mul())	← a.mul()를 수행하면 두 수를 곱한 결과(4 * 2)를 돌려줌
8	
>>> print(a.sub())	← a.sub()를 수행하면 두 수를 뺀 결과(4 - 2)를 돌려줌
2	
>>> print(a.div())	← a.div()를 수행하면 두 수를 나눈 결과(4 / 2)를 돌려줌
2	

✓ 이렇게 동작하는 FourCal 클래스를 만드는 것이 목표임

■ 클래스 구조 만들기

```
>>> class FourCal:
...     pass          ← 아무것도 수행하지 않는 문법으로 임시로 코드를 작성할 때 사용
...
>>>
```

- ✓ `a = FourCal()` 처럼 객체를 아무 기능이 없이 만듦
- ✓ `FourCal` 클래스는 아무런 변수나 함수도 포함하지 않지만, 원하는 객체 `a`를 만들 수 있는 기능은 가지고 있음

```
>>> a = FourCal()
>>> type(a)          ← 객체의 타입을 출력하는 내장 함수
<class '__main__.FourCal'>    ← 객체 a의 타입은 FourCal 클래스임
```

- ✓ `a = FourCal()`로 `a` 객체를 먼저 만들고, 그 다음에 `type(a)`로 `a`라는 객체가 어떤 타입인지 알아보았음
- ✓ 역시 객체 `a`가 `FourCal` 클래스의 객체임을 알 수 있음

■ 객체에 숫자 지정할 수 있게 만들기

```
>>> a.setdata(4, 2)
```

- ✓ a 객체에 사칙연산을 할 때 사용할 2개의 숫자를 먼저 알려주기 위해 연산을 수행할 대상(4, 2)을 객체에 지정할 수 있게 만듦
- ✓ 위 문장이 수행되려면 다음과 같이 소스 코드를 작성해야 함

```
>>> class FourCal:  
...     def setdata(self, first, second):  
...         self.first = first  
...         self.second = second  
...  
>>>
```

- ✓ 이전에 만들었던 pass 문장을 삭제하고, setdata 함수를 만듦
- ✓ 클래스 안에 함수는 **메소드(Method)**라고 부름

```
class FourCal:
```

```
    def setdata(self, first, second):
```

```
        self.first = first
```

```
        self.second = second
```

← ① 메소드의 매개변수

} ② 메소드의 수행문

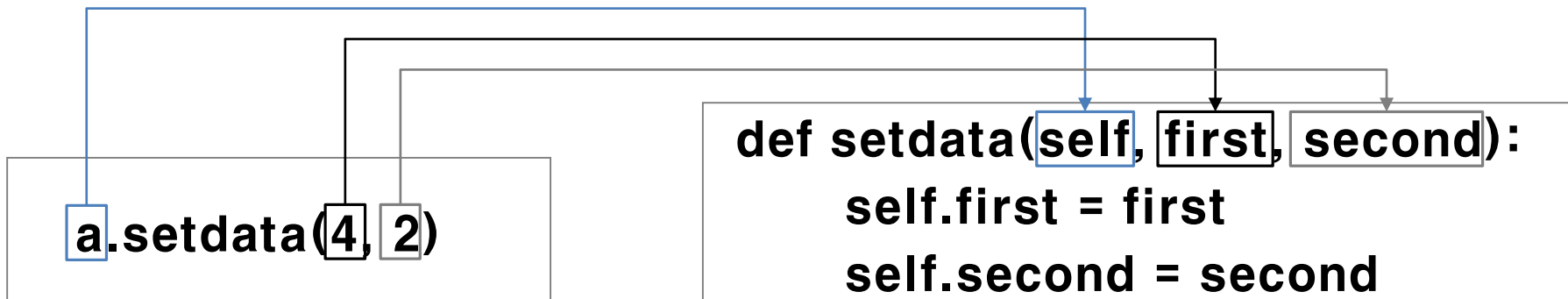
① setdata 메소드의 매개변수

- ✓ 매개변수로 self, first, second라는 3개의 입력값을 받음
- ✓ 메소드의 첫 번째 매개변수 self는 특별한 의미를 가짐

```
>>> a = FourCal()
```

```
>>> a.setdata(4, 2) ← 객체를 통해 클래스의 메소드를 호출하려면 도트(.) 연산자를 사용함
```

- ✓ a.setdata(4, 2)처럼 호출하면 setdata 메소드의 첫 번째 매개변수 self에는 setdata 메소드를 호출한 객체 a가 자동으로 전달됨



[메소드의 또 다른 호출 방법]

```
>>> a = FourCal()
>>> FourCal.setdata(a, 4, 2)
```

- ✓ '클래스명.메소드' 형태로 호출할 때는 객체 a를 첫 번째 매개변수 self에 전달해 주어야 함
- ✓ 반면에 앞에서 보았듯이 '객체.메소드' 형태로 호출할 때는 self를 반드시 생략해서 호출해야 함

```
class FourCal:
```

```
    def setdata(self, first, second):
```

```
        self.first = first
```

```
        self.second = second
```

← ① 메소드의 매개변수

} ② 메소드의 수행문

② setdata 메소드의 수행문

- ✓ a.setdata(4, 2)처럼 호출하면 setdata 메소드의 매개변수 first, second에는 각각 값 4와 2가 전달되어 setdata 메소드의 수행문은 다음과 같이 해석됨

```
self.first = 4
```

```
self.second = 2
```

- ✓ self는 전달된 객체 a이므로 다시 다음과 같이 해석됨

```
a.first = 4
```

```
a.second = 2
```

- ✓ a 객체에 객체변수 first가 생성되고 값 4가 저장됨
- ✓ 객체에 생성되는 객체만의 변수를 객체변수라고 부름


```
>>> a = FourCal()
>>> a.setdata(4, 2)
>>> print(a.first)
4
>>> print(a.second)
2
```

✓ a 객체에 객체변수 first와 second가 생성되었음을 확인할 수 있음

```
>>> a = FourCal()
>>> b = FourCal()
>>> a.setdata(4, 2)
>>> print(a.first)
4
>>> b.setdata(3, 7)
>>> print(b.first)
3
```

✓ a와 b 객체는 first라는 객체변수를 가지고 있지만 그 변수의 값은 각기 다름

✓ 객체변수는 다른 객체에 영향을 받지 않고 독립적으로 그 값을 유지함

■ 더하기 기능 만들기

```
>>> a = FourCal()
>>> a.setdata(4, 2)
>>> print(a.add())
6
```

- ✓ 2개의 숫자를 더하는 기능을 갖춘 클래스를 만들어야 함
- ✓ 이 연산이 가능하도록 FourCal 클래스를 만들어 봄

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...     def add(self):
...         result = self.first + self.second
...         return result
...
>>>
```

```
>>> a = FourCal()
>>> a.setdata(4, 2)
>>> print(a.add())
6
```

- ✓ a.add()라고 호출하면 새로 추가된 add 메소드가 호출되어 값 6이 출력됨
- ✓ 어떤 과정을 거쳐 출력되는지 add 메소드를 자세히 살펴봄

```
def add(self):
    result = self.first + self.second
    return result
```

- ✓ add 메소드의 매개변수는 self이고, 반환 값은 result임

result = self.first + self.second

result = a.first + a.second ← a.add()와 같이 a 객체에 의해 add 메소드가 수행되면, self에는 객체 a가 자동으로 입력됨

result = 4 + 2 ← a.setdata(4, 2)가 먼저 호출되어 a.first=4, a.second=2이 설정되었기에 다시 해석됨

■ 곱하기, 빼기, 나누기 기능 만들기

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...     def add(self):
...         result = self.first + self.second
...         return result
...     def mul(self):
...         result = self.first * self.second
...         return result
...     def sub(self):
...         result = self.first - self.second
...         return result
...     def div(self):
...         result = self.first / self.second
...         return result
...
>>>
```

```
>>> a = FourCal()
>>> b = FourCal()
>>> a.setdata(4, 2)
>>> b.setdata(3, 8)
>>> a.add()
6
>>> a.mul()
8
>>> a.sub()
2
>>> a.div()
2
>>> b.add()
11
>>> b.mul()
24
>>> b.sub()
-5
>>> b.div()
0.375
```

✓ 모든 기능이 정상적으로 동작하는 사칙연산 클래스를 만들어 봄

❖ 생성자(Constructor)

- 위에서 만든 FourCal 클래스를 다음과 같이 실행하면 어떻게 될까?

```
>>> a= FourCal()
>>> a.add()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in add
AttributeError: 'FourCal' object has no attribute 'first'
```

- ✓ FourCal 클래스의 인스턴스 a에 setdata 메소드를 수행하지 않고, add 메소드를 수행하면 오류가 발생함.
- ✓ 왜냐하면 setdata 메소드를 수행해야 객체 a의 first와 second가 생성되는 때문임
- ✓ 객체에 초기값을 설정해야 할 필요가 있을 때는 생성자를 구현하는 것이 안전함
- ✓ 생성자(Constructor)란 객체가 생성될 때 자동으로 호출되는 메소드를 의미함

- 메소트 이름으로 `__init__`를 사용하면 이 메소드는 생성자가 됨

```
>>> class FourCal:
...     def __init__(self, first, second):
...         self.first = first
...         self.second = second
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...     def add(self):
...         result = self.first + self.second
...         return result
...     ...생략...
...     def div(self):
...         result = self.first / self.second
...         return result
...
>>>
```

- 생성자 `__init__` 메소드

```
def __init__(self, first, second):  
    self.first = first  
    self.second = second
```

- ✓ `setdata` 메소드와의 차이점은 이름을 `__init__`으로 했기 때문에 생성자로 인식되어 객체가 생성되는 시점에 자동으로 호출됨

```
>>> a = FourCall()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: __init__() missing 2 required positional arguments: 'first' and 'second'
```

- ✓ `a = FourCall()`을 수행할 때, 생성자 `__init__`의 매개변수 `first`, `second`에 해당하는 값이 전달되지 않았기 때문에 오류가 발생함
- ✓ 오류를 해결하려면, `first`와 `second`에 해당하는 값을 전달하여 객체를 생성해야 함

```
>>> a = FourCal(4, 2)  
>>>
```


- `__init__` 메소드가 호출되면 객체변수가 생성되는 것을 확인할 수 있음

```
>>> a = FourCal(4, 2)
>>> print(a.first)
4
>>> print(a.second)
2
>>> a.add()
6
>>> a.dev()
2.0
```

❖ 클래스의 상속

상속(Inheritance)이란 '물려받다'라는 뜻으로, 어떤 클래스를 만들 때 다른 클래스의 기능을 물려받을 수 있게 만드는 것임

- FourCal 클래스를 상속하는 MoreFourCal 클래스 만들기

```
>>> class MoreFourCal(FourCal):  
...     pass  
...  
>>> a = MoreFourCal(4, 2)  
>>> a.add()  
6  
>>> a.mul()  
8
```

✓ MoreFourCal 클래스는 FourCal 클래스의 모든 기능을 사용할 수 있음

- 클래스의 상속 사용법

```
class 클래스 이름(상속할 클래스 이름)
```

- a의 b제곱(a^b)을 계산하는 MoreFourcal 클래스 만들기

```
>>> class MoreFourCal(FourCal):
...     def pow(self):
...         result = self.first ** self.second
...         return result
...
>>> a = MoreFourCal(4, 2)
>>> a.pow()
16
```

[왜 상속을 해야 할까?]

- ✓ 상속은 기존 클래스를 변경하지 않고 기능을 추가하거나 기존 기능을 변경하려고 할 때 사용한다.
- ✓ 기존 클래스가 라이브러리 형태로 제공되거나 수정이 허용되지 않는 상황이라면 상속을 사용해야 한다.

▪ 매소드 오버라이딩(Overriding)

```
>>> a = FourCal(4, 0)
>>> a.div()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    result = self.first / self.second
ZeroDivisionError: division by zero
```

- ✓ div 메소드를 호출하면 4를 0으로 나누려고 하기 때문에 오류가 발생함
- ✓ 0으로 나눌 때 오류가 아닌 0을 돌려주도록 만들고 싶다면 어떻게 해야 할까?

```
>>> class SafeFourCal(FourCal):
...     def div(self):
...         if self.second == 0:  ← 나누는 값이 0인 경우, 숫자 0을 돌려주도록 수정
...             return 0
...         else:
...             return self.first / self.second
...
>>>
```

- ✓ SafeFourCal 클래스는 FourCal 클래스에 있는 div 메소드를 동일한 이름으로 다시 작성하였음
- ✓ 부모 클래스에 있는 메소드를 동일한 이름으로 다시 만드는 것을 **메소드 오버라이딩**(Overriding, 덮어쓰기) 이라고 함
- ✓ 메소드를 오버라이딩하면 부모 클래스의 메소드 대신 오버라이딩한 메소드가 호출됨

```
>>> a = SafeFourCal(4, 0)
>>> a.div()
0
```

- ✓ SafeFourCal 클래스를 사용하여 0을 돌려주는 것을 확인함

❖ 클래스 변수

■ 클래스 변수 선언

```
>>> class Family:
...     lastname = "김"
...
```

- ✓ Family 클래스에 선언한 lastname이 클래스 변수임
- ✓ 클래스 변수는 클래스 안에 메소드(함수)와 마찬가지로 변수를 선언하여 생성함

■ 클래스 변수 사용

```
>>> print(Family.lastname)    ← ① 클래스 이름.클래스 변수로 사용
김
>>> a = Family()
>>> b = Family()
>>> print(a.lastname)         ← ② 클래스로 만든 객체를 통해서 사용
김
>>> print(b.lastname)
김
```

- Family 클래스의 lastname을 "박"이라는 문자열로 바꾸면 어떻게 될까?

```
>>> Family.lastname = "박"
>>> print(a.lastname)
박
>>> print(b.lastname)
박
```

- ✓ 클래스 변수 값을 변경했더니 클래스로 만든 객체의 lastname 값도 모두 변경됨
- ✓ 즉, 클래스 변수는 클래스로 만든 모든 객체에 공유된다는 특징이 있음

❖ 모듈이란?

- 함수나 변수 또는 클래스를 모아 놓은 파일
- 다른 파이썬 프로그램에서 불러와 사용할 수 있게끔 만든 파이썬 파일이라고도 함

❖ 모듈 만들기

```
# mod1.py
def add(a, b):
    return a + b

def sub(a, b):
    return a - b
```

- 위와 같이 add와 sub 함수만 있는 파일 mod1.py를 만들고, c:\wdoit 디렉터리에 저장
- 확장자 .py로 만든 파이썬 파일은 모두 모듈임

❖ 모듈 불러오기

```
>>> import mod1
>>> print(mod1.add(3, 4))
7
>>> print(mod1.sub(4, 2))
2
```

- ✓ mod1.py를 불러오기 위해 import mod1이라고 입력함
- ✓ import는 이미 만들어진 파이썬 모듈을 사용할 수 있게 해주는 명령어임
- ✓ mod1.py 파일에 있는 add 함수를 사용하기 위해서는 mod1.add 처럼 모듈 이름 뒤에 '.'(도트 연산자)를 붙이고 함수 이름을 쓰면 됨

▪ import의 사용 방법

import 모듈 이름

- ✓ 모듈 이름은 mod1.py에서 .py라는 확장자를 제거한 mod1만을 가리킴
- ✓ **모듈 이름.함수 이름** 처럼 사용함

- 모듈 이름 없이 함수 이름만 쓰는 방법

```
from 모듈 이름 import 모듈 함수
```

- ✓ 모듈 이름을 붙이지 않고 모듈의 함수를 쓸 수 있음

```
>>> from mod1 import add
>>> add(3,4)
7
```

- ✓ mod1.py 파일의 add 함수만 사용할 수 있음
- ✓ add 함수와 sub 함수를 모두 사용하는 방법은 2가지가 있음

- ① from mod1 import add, sub
- ② from mod1 import *

- ✓ ①번 방법은 콤마로 구분하여 필요한 함수를 불러올 수 있음
- ✓ ②번 방법은 * 문자를 사용하여 모든 함수를 불러올 수 있음

❖ if __name__ == "__main__": 의 의미

```
# mod1.py
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

print(add(1, 4))
print(sub(4, 2))
```

✓ add(1, 4)와 sub(4, 2)의 결과를 출력하는 문장을 추가함

- 작성한 mod1.py 파일을 다음과 같이 실행함

```
C:\wdoit>python mod1.py
5
2
```

- mod1 모듈을 import할 때는 문제가 생김

```
C:\wdoit>python
>>> import mod1
5
2
```

- ✓ mod1.py 파일의 add와 sub 함수만 사용하려고 했는데, import mod1을 수행하는 순간 mod1.py가 실행이 되어 결과값을 출력함

- 이러한 문제를 방지하려면, mod1.py 파일의 마지막 부분을 변경해야 함

```
if __name__ == "__main__":
    print(add(1, 4))
    print(sub(4, 2))
```

- ✓ if __name__ == "__main__":을 사용하면 C:\wdoit>python mod1.py처럼 직접 이 파일을 실행시켰을 때는 __name__ == "__main__"이 참이 되어 if문 다음 문장이 수행됨
 - ✓ 반대로 대화형 인터프리터나 다른 파일에서 이 모듈을 불러서 사용할 때는 __name__ == "__main__"이 거짓이 되어 if문 다음 문장이 수행되지 않음

❖ 클래스나 변수 등을 포함한 모듈

```
# mod2.py
PI = 3.141592          ← 원주율 값에 해당하는 PI 변수

class Math:            ← 원의 넓이를 계산하는 Math 클래스
    def solv(self, r):
        return PI * (r ** 2)

def add(a, b):          ← 두 값을 더하는 add함수
    return a + b

if __name__ == "__main__":
    print(PI)
    a = Math()
    print(a.solv(2))
    print(add(PI, 4.4))
```

} 파일을 직접 실행시켰을 때 수행됨

✓ 변수, 클래스, 함수 등을 모두 포함하고 있는 모듈임

▪ mod2.py 모듈을 대화형 인터프리터에서 수행

```
C:\Wdoit>python
>>> import mod2
>>> print(mod2.PI)           ← mod2.PI 처럼 입력해서 PI 변수 값을 사용
3.141592
>>> a = mod2.Math()         ← Math 클래스를 사용하려면 '.' 앞에 모듈 이름을 먼저 입력
>>> print(a.solv(2))
12.566368
>>> print(mod2.add(mod2.PI, 4.4)) ← add 함수 사용
7.541592
```

▪ mod2.py 모듈을 다른 파일에서 불러오기

```
# modtest.py
import mod2
result = mod2.add(3, 4)
print(result)
```

✓ 다른 파이썬 파일에서도 import mod2로 mod2 모듈을 불러와서 사용하면 됨

❖ 패키지란 무엇인가?

- 패키지(Package)는 도트(.)를 사용하여 파이썬 모듈을 계층적(디렉터리 구조)으로 관리할 수 있게 해줌
- 모듈 이름이 A.B인 경우에 A는 패키지이름이 되고, B는 A 패키지의 B 모듈이 됨
- 파이썬 패키지는 디렉터리와 파이썬 모듈로 이루어진 구조임
- game, sound, graphic, play는 디렉터리 이름이고, 확장자가 .py인 파일이 파이썬 모듈임
- game 디렉터리가 이 패키지의 루트 디렉터리이고, sound, graphic, play는 서브 디렉터리임
- 패키지 구조로 파이썬 프로그램을 만드는 것이 공동 작업이나 유지 보수 등 여러 면에서 유리함

```
game/  
  __init__.py  
  sound/  
    __init__.py  
    echo.py  
    wav.py  
  graphic/  
    __init__.py  
    screen.py  
    render.py  
  play/  
    __init__.py  
    run.py  
    test.py
```

❖ 패키지 만들기

▪ 패키지 기본 구성 요소 준비하기

1. C:\doit 디렉터리 밑에 game 및 서브 디렉터리를 생성하고 .py 파일들을 만듦

```
C:/doit/game/__init__.py  
C:/doit/game/sound/__init__.py  
C:/doit/game/sound/echo.py  
C:/doit/game/graphic/__init__.py  
C:/doit/game/graphic/render.py
```

2. 각 디렉터리에 __init__.py 파일을 만들어 놓기만 하고 내용은 일단 비워 둠
3. echo.py 파일은 다음과 같이 만듦

```
# echo.py  
def echo_test():  
    print("echo")
```


4. render 파일을 다음과 같이 만듦

```
# render.py
def render_test():
    print("render")
```

5. game 패키지를 참조할 수 있도록 명령 프롬프트창에서 set 명령어로 PYTHONPATH 환경 변수에 C:\wdoit 디렉터리를 추가함

```
C:\W>set PYTHONPATH=C:\wdoit
C:\W>python
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:54:25) [MSC v.1900 64 bit
(AM...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

▪ 패키지 안의 함수 실행하기

패키지를 사용하여 echo.py 파일의 echo_test 함수를 실행해 봄 (3가지 방법)

첫 번째, echo 모듈을 import하여 실행하는 방법

```
>>> import game.sound.echo
>>> game.sound.echo.echo_test()
echo
```

두 번째, echo 모듈이 있는 디렉터리까지를 from ... import하여 실행하는 방법

```
>>> from game.sound import echo
>>> echo.echo_test()
echo
```

세 번째, echo 모듈의 echo_test 함수를 직접 import하여 실행하는 방법

```
>>> from game.sound.echo import echo_test
>>> echo_test()
echo
```

▪ 패키지 안의 함수 사용할 때 주의사항

첫 번째, 다음과 같이 echo_test 함수를 사용하는 것은 불가능함

```
>>> import game
>>> game.sound.echo.echo_test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'sound'
```

- ✓ import game을 수행하면 game 디렉터리의 모듈 또는 game 디렉터리의 __init__.py에 정의된 것만 참조할 수 있음

두 번째, import 마지막 항목에 함수를 사용하는 것도 불가능함

```
>>> import game.sound.echo.echo_test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named echo_test
```

- ✓ 도트 연산자(.)를 사용해서 import a.b.c처럼 import할 때 가장 마지막 항목인 c는 반드시 모듈 또는 패키지여야만 함

❖ __init__.py의 용도

- __init__.py 파일은 해당 디렉터리가 패키지의 일부임을 알려주는 역할을 함
- game, sound, graphic 등 패키지에 포함된 디렉터리에 __init__.py 파일이 없다면 패키지로 인식되지 않음
- sound 디렉터리의 __init__.py를 제거하고 다음을 수행해 봄

```
>>> import game.sound.echo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named sound.echo
```

- ✓ sound 디렉터리에 __init__.py 파일이 없어서 ImportError가 발생하게 됨
- ✓ python3.3 버전부터는 __init__.py 파일이 없어도 패키지로 인식함. 하위 버전 호환을 위해 생성하는 것이 안전함

▪ __all__의 용도

```
>>> from game.sound import *
>>> echo.echo_test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'echo' is not defined
```

- ✓ echo 모듈을 사용할 수 있어야 할 것 같은데, echo라는 이름이 정의되지 않았다는 이름 오류(NameError)가 발생함
- ✓ 오류 이유는, 특정 디렉터리의 모듈을 *를 사용하여 import 할 때에는 해당 디렉터리의 __init__.py 파일에 __all__ 변수를 설정해야 함

```
# C:\wdoit\game\sound\__init__.py
__all__ = ['echo']
```

- ✓ 여기서 __all__이 의미하는 것은 sound 디렉터리에서 *기호를 이용하여 import할 경우 이곳에 정의된 echo 모듈만 import 된다는 의미임

```
>>> from game.sound import *
>>> echo.echo_test()
echo
```

❖ relative 패키지

- graphic 디렉터리의 render.py 모듈이 sound 디렉터리의 echo.py 모듈을 사용하려면, render.py를 수정하면 가능함

```
# render.py
from game.sound.echo import echo_test

def render_test():
    print ("render")
    echo_test()
```

- from game.sound.echo import echo_test라는 문장을 추가하여 echo_test()함수를 사용할 수 있도록 수정함
- 수정 후 다음과 같이 수행함

```
>>> from game.graphic.render import render_test
>>> render_test()
render
echo
```

- 전체 경로를 이용하여 import할 수 있지만, relative하게 import하는 것도 가능함

```
# render.py
from ..sound.echo import echo_test

def render_test():
    print ("render")
    echo_test()
```

- ✓ ..은 부모 디렉터리, .은 현재 디렉터리를 의미함
- ✓ graphic과 sound 디렉터리는 동일한 깊이(depth)이므로 부모 디렉터리(..)를 사용하여 위와 같은 import가 가능한 것임
- ✓ ..과 같은 relativ한 접근자는 render.py와 같이 모듈 안에서만 사용해야 함
- ✓ 파이썬 인터프리터에서 relative한 접근자를 사용하면 'SystemError: cannot perform relative import'와 같은 오류가 발생함

- ❖ 프로그램을 만들다 보면 수없이 많은 오류를 만나게 됨
 - 오류가 발생하는 이유는 프로그램이 잘못 동작하는 것을 막기 위한 배려임
 - 하지만 이러한 오류를 무시하고 싶을 때가 있는데 try, except를 이용해서 예외적으로 오류를 처리할 수 있게 해줌

❖ 오류는 어떨 때 발생하는가?

- 오타를 입력했을 때 발생하는 구문 오류
- 디렉터리 안에 없는 파일을 열려고 시도할 때 발생하는 오류

```
>>> f = open("나없는파일", 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    f = open("나없는파일", 'r')
FileNotFoundError: [Errno 2] No such file or directory: '나없는파일'
```

- ✓ 없는 파일을 열려고 시도하면 'FileNotFoundError' 오류가 발생함

- 0으로 다른 숫자를 나누는 경우 발생하는 오류

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

✓ 4를 0으로 나누려니까 'ZeroDivisionError' 오류가 발생함

- 리스트에 존재하지 않는 요소를 얻으려고 할 때 발생하는 오류

```
>>> a = [1, 2, 3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

✓ a[4]는 a리스트에서 얻을 수 없는 값이므로 'IndexError'오류가 발생함

❖ 오류 예외 처리 기법

▪ try, except문

✓ 기본 구조

```
try:
    ...
except [발생오류[as 오류 메시지 변수]]:
    ...
```

- try 블록 수행 중 오류가 발생하면 except 블록이 수행됨
- 하지만 try 블록에서 오류가 발생하지 않는다면, except 블록은 수행되지 않음
- except 구문의 []기호는 괄호 안의 내용을 생략할 수 있다는 관례 표기법임

1. try, except만 쓰는 방법

```
try:
    ...
except:
    ...
```

- 오류 종류에 상관없이 오류가 발생하면 except블록을 수행함

2. 발생 오류만 포함한 except문

```
try:  
    ...  
except 발생 오류:  
    ...
```

- 오류가 발생했을 때 except문에 미리 정해 놓은 오류 이름과 일치할 때만 except 블록을 수행한다는 의미임

3. 발생 오류와 오류 메시지 변수까지 포함한 except문

```
try:  
    ...  
except 발생 오류 as 오류 메시지 변수:  
    ...
```

- 두 번째 경우에서 오류 메시지의 내용까지 알고 싶을 때 사용하는 방법임

```
try:  
    4 / 0  
except ZeroDivisionError as e:  
    print(e)
```

- 위처럼 4를 0으로 나누려고 하면 ZeroDivisionError가 발생하여 except 블록이 실행되고 e라는 오류 메시지를 다음과 같이 출력함

결과값: division by zero

▪ try .. else

- ✓ try문은 else절을 지원함
- ✓ else절은 예외가 발생하지 않은 경우에 실행되며 반드시 except절 바로 다음에 위치함

```
try:
    f = open('foo.txt', 'r')
except FileNotFoundError as e:
    print(str(e))
else:
    data = f.read()
    f.close()
```

- 만약 foo.txt라는 파일이 없다면 except절이 수행되고, 있다면 else절이 수행됨

▪ try .. finally

- ✓ try문에는 finally절을 사용할 수 있음
- ✓ finally절은 try문 수행 도중 예외 발생 여부에 상관없이 항상 수행됨
- ✓ 보통 finally절은 사용한 리소스를 close해야 할 때에 많이 사용함

```
f = open('foo.txt', 'w')
try:
    #무언가를 수행함
finally:
    f.close()
```

- foo.txt라는 파일을 쓰기 모드로 연 후에 try문이 수행된 후 예외 발생 여부에 상관없이 finally절에서 f.close()로 열린 파일을 닫을 수 있음

■ 여러 개의 오류 처리하기

```
try:  
    ...  
except 발생 오류 1:  
    ...  
except 발생 오류 2:  
    ...
```

✓ 0으로 나누는 오류와 인덱싱 오류를 같이 처리한 예

```
try:  
    a = [1, 2]  
    print(a[3])  
    4/0  
except ZeroDivisionError:  
    print("0으로 나눌 수 없습니다.")  
except IndexError:  
    print("인덱싱할 수 없습니다.")
```

- 먼저 발생한 오류의 문자열이 출력됨

✓ 오류 메시지 변수를 사용한 예

```
try:
    a = [1, 2]
    print(a[3])
    4/0
except ZeroDivisionError as e:
    print(e)
except IndexError as e:
    print(e)
```

- list index out of range 오류 메시지가 출력될 것임

✓ ZeroDivisionError와 IndexError를 함께 처리한 예

```
try:
    a = [1, 2]
    print(a[3])
    4/0
except (ZeroDivisionError, IndexError) as e:
    print(e)
```

- 2개 이상의 오류를 동시에 처리하기 위해 괄호로 묶어 처리함

❖ 오류 회피하기

- 프로그래밍을 하다 보면 특정 오류가 발생할 경우 그냥 통과시켜야 할 때가 있음

```
try:  
    f = open('나없는파일', 'r')  
except FileNotFoundError: ← 파일이 없더라도 오류를 발생시키지 않고 통과함  
    pass
```

- ✓ try문 안에서 FileNotFoundError가 발생할 경우에 pass를 사용해 오류를 회피함

❖ 오류 일부러 발생시키기

- 프로그래밍을 하다 보면 오류를 일부러 발생시켜야 할 경우도 생김
- raise라는 명령어를 이용해 오류를 강제로 발생시킬 수 있음
- 강제로 오류를 발생시키는 예

```
class Bird:
    def fly(self):
        raise NotImplementedError
```

- ✓ Bird 클래스를 상속받는 자식 클래스는 반드시 fly라는 함수를 구현해야 한다는 의지를 보여 줌
- 자식 클래스가 fly 함수를 구현하지 않은 상태로 fly 함수를 호출한다면 어떻게 될까?

```
class Eagle(Bird):
    pass

eagle = Eagle()
eagle.fly()
```

← Eagle 클래스는 Bird 클래스를 상속 받음

- ✓ Eagle 클래스에서 fly 함수를 구현하지 않아서, Bird 클래스의 fly함수가 호출됨

- raise문에 의해 NotImplemented Error가 발생할 것임

```
Traceback (most recent call last):  
  File "...", line 33, in <module>  
    eagle.fly()  
  File "...", line 26, in <module>  
    raise NotImplementedError  
NotImplementedError
```

- NotImplementedError가 발생되지 않게 하려면 Eagle 클래스에 fly 함수 구현해야 함

```
class Eagle(Bird):  
    def fly(self):  
        print("very fast")  
  
eagle = Eagle()  
eagle.fly()
```

- 자식 클래스에 fly함수를 구현한 후 프로그램을 실행하면 오류 없이 출력됨

```
very fast
```

❖ 예외 만들기

- 프로그램 수행 도중 특수한 경우에는 예외를 직접 만들어서 사용함
- 별명을 출력해 주는 예

```
class MyError(Exception):  ← 파이썬 내장 클래스인 Exception 클래스를 상속받음
    pass
```

```
def say_nick(nick):
    if nick == '바보':
        raise MyError()
    print(nick)
```

```
say_nick('천사')
say_nick('바보')
```

천사

Traceback (most recent call last):

File "...", line 11, in <module>

say_nick('바보')

File "...", line 7, in say_nick

raise MyError()

MyError

- ✓ 프로그램 실행해 보면 '천사'가 한 번 출력된 후, MyError가 발생함

- 예외 처리 기법을 사용하여 MyError 발생을 예외 처리해 봄

```
class MyError(Exception):  
    pass  
  
def say_nick(nick):  
    if nick == '바보':  
        raise MyError()  
    print(nick)  
  
try:  
    say_nick('천사')  
    say_nick('바보')  
except MyError:  
    print("허용되지 않는 별명입니다.")
```

천사
허용되지 않는 별명입니다.

- 오류 메시지를 사용하고 싶다면 예외 처리해 봄

```
class MyError(Exception):  
    pass
```

```
def say_nick(nick):  
    if nick == '바보':  
        raise MyError()  
    print(nick)
```

```
try:  
    say_nick('천사')  
    say_nick('바보')  
except MyError as e:  
    print(e)
```

천사

- ✓ 프로그램 실행해 보면 print(e)로 오류 메시지가 출력되지 않는 것을 확인함

- 오류 클래스에 `__str__` 메소드 구현

```
class MyError(Exception):  
    def __str__(self):  
        return "허용되지 않는 별명입니다."  
  
def say_nick(nick):  
    if nick == '바보':  
        raise MyError()  
    print(nick)  
  
try:  
    say_nick('천사')  
    say_nick('바보')  
except MyError as e:  
    print(e)
```

천사
허용되지 않는 별명입니다.

- ✓ `__str__` 메소드는 `print(e)`처럼 오류 메시지를 `print`문으로 출력하는 메소드임

Don't Reinvent The Wheel, 이미 있는 것을 다시 만드느라 쓸데없이 시간을 낭비하지 말라.
파이썬 내장 함수들은 import 없이 사용할 수 있음

❖ abs

- `abs(x)`는 어떤 숫자를 입력을 받았을 때, 그 숫자의 절대값을 돌려주는 함수

```
>>> abs(3)
3
>>> abs(-3)
3
>>> abs(-1.2)
1.2
```

❖ all

- `all(x)`는 반복 가능한 자료형 `x`를 입력 인수로 받으며 이 `x`가 모두 참이면 `True`, 거짓이 하나라도 있으면 `False`를 돌려줌

```
>>> all([1, 2, 3])
True
```

- 리스트 자료형 `[1, 2, 3]`은 모든 요소가 참이므로 `True`를 돌려줌


```
>>> all([1, 2, 3, 0])  
False
```

- 리스트 자료형[1, 2, 3, 0]중에서 요소 0은 거짓이므로 False를 돌려줌

❖ any

- any(x)는 x중 하나라도 참이 있으면 True를 돌려주고, x가 모두 거짓일 때에만 False를 돌려줌. all(x)의 반대임.

```
>>> any([1, 2, 3, 0])  
True
```

- ✓ 리스트 자료형[1, 2, 3, 0]중에서 1, 2, 3이 참이므로 True를 돌려줌

```
>>> any([0, ""])  
False
```

- ✓ 리스트 자료형[0, ""]의 요소 0과 ""은 모두 거짓이므로 False를 돌려줌

❖ chr

- chr(i)는 아스키 코드 값을 입력으로 받아 그 코드에 해당하는 문자를 출력하는 함수

```
>>> chr(97)
'a'           ← 아스키 코드 97은 소문자 a
>>> chr(48)
'0'           ← 아스키 코드 48은 숫자 0
```

❖ dir

- dir은 객체가 자체적으로 가지고 있는 변수나 함수를 보여줌

```
>>> dir([1, 2, 3])
['append', 'count', 'extend', 'index', 'insert', 'pop',...]
>>> dir({'1':'a'})
['clear', 'copy', 'get', 'has_key', 'items', 'keys',...]
```

❖ divmod

- `divmod(a, b)`는 2개 숫자를 입력으로 받아서 `a`를 `b`로 나눈 몫과 나머지를 튜플 형태로 돌려주는 함수

```
>>> divmod(7, 3)
(2, 1)      ← 7 나누기 3의 몫은 2, 나머지는 1
```

```
>>> 7 // 3
2
```

```
>>> 7 % 3
1
```

❖ enumerate

- 순서가 있는 자료형(리스트, 튜플, 문자열)을 입력으로 받아 인덱스 값을 포함하는 enumerate 객체를 돌려주는 함수

```
>>> for i, name in enumerate(['body', 'foo', 'bar']):  
...     print(i, name)  
...  
0 body  
1 foo  
2 bar
```

- ✓ 순서 값과 함께 body, foo, bar가 순서대로 출력됨
- ✓ enumerate를 for문과 함께 사용하면 자료형의 현재 순서(index)와 그 값을 쉽게 알 수 있음
- ✓ for문처럼 반복되는 구간에서 객체가 현재 어느 위치에 있는지 알려주는 인덱스 값이 필요할 때 enumerate함수를 사용하면 유용함

❖ eval

- `eval(expression)`은 실행 가능한 문자열(`1+2`, `'hi' + 'a'` 등)을 입력으로 받아 문자열을 실행한 결과값을 돌려주는 함수

```
>>> eval('1+2')
3
>>> eval("'hi' + 'a'")
'hia'
>>> eval('divmod(4, 3)')
(1, 1)
```

- ✓ 보통 `eval`은 입력 받은 문자열로 파이썬 함수나 클래스를 동적으로 실행하고 싶은 경우에 사용됨

❖ filter

- 첫 번째 인수로 함수 이름을, 두 번째 인수로 그 함수에 차례로 들어갈 자료형을 받음
- 두 번째 인수인 반복 가능한 자료형 요소가 첫 번째 인수인 함수에 입력되었을 때 반환 값이 참인 것만 걸러내서 돌려줌
- positive 함수는 리스트를 입력받아 양수 값을 돌려주는 함수

```
#positive.py
def positive(l):
    result = []      ← 리턴값이 참인 것만 걸러내서 저장할 변수
    for i in l:
        if i > 0:    ← i가 0보다 클 때
            result.append(i)  ← 리스트에 i 추가
    return result

print(positive([1, -3, 2, 0, -5, 6]))
```

결과값: [1, 2, 6]

- filter 함수를 사용하면 앞의 내용을 간략하게 작성할 수 있음

```
#filter1.py
def positive(x):
    return x > 0  ←———— 비교연산자로 참이면 True, 거짓이면 False를 반환

print(list(filter(positive, [1, -3, 2, 0, -5, 6])))
```

결과값: [1, 2, 6]

- ✓ 두 번째 인수인 리스트의 요소들이 첫 번째 인수인 positive 함수에 입력되었을 때 리턴값이 참인 것만 묶어서 돌려 줌
- lambda를 사용하면 더 간략하게 작성할 수 있음

```
>>> list(filter(lambda x: x > 0, [1, -3, 2, 0, -5, 6]))
```

❖ hex

- hex(x)는 정수 값을 입력받아 16진수(hexadecimal)로 변환하여 돌려주는 함수

```
>>> hex(234)
```

```
'0xea'
```

```
>>> hex(3)
```

```
'0x3'
```


❖ id

- `id(object)`는 객체를 입력 받아 객체의 고유 주소 값(레퍼런스)을 돌려주는 함수

```
>>> a = 3
>>> id(3)
135072304
>>> id(a)
135072304
>>> b = a
>>> id(b)
135072304
```

3, a, b는 모두 같은 객체를 가리킴

- 3, a, b는 고유 주소 값이 모두 135072304임
- `id(4)`라고 입력하면 4는 3, a, b와 다른 객체이므로 다른 고유 주소 값이 출력됨

```
>>> id(4)
135072292
```

❖ input

- `input([prompt])`은 사용자 입력을 받는 함수
- 매개변수로 문자열을 주면, 그 문자열은 프롬프트가 됨

```
>>> a = input() ← 사용자가 입력한 정보를 변수 a에 저장
```

```
hi
```

```
>>> a
```

```
'hi'
```

```
>>> b = input("Enter: ") ← Enter: 라는 프롬프트를 띄우고 사용자 입력을 받음
```

```
Enter: hi
```

```
>>> b
```

```
'hi' ← 사용자 입력으로 받은 'hi' 출력
```

❖ int

- int(x)는 문자열 형태의 숫자나 소수점이 있는 숫자 등을 정수 형태로 돌려주는 함수

```
>>> int('3')  ← 문자열 형태 '3'
3
>>> int(3.4)  ← 소수점이 있는 숫자 3.4
3
```

- int(x, radix)는 radix 진수로 표현된 문자열 x를 10진수로 변환하여 돌려줌

```
>>> int('11', 2)  ← 2진수로 표현된 '11'의 10진수 값을 구함
3
>>> int('1A', 16) ← 16진수로 표현된 '1A'의 10진수 값을 구함
26
```

❖ isinstance

- `isinstance(object, class)`는 첫 번째 인수로 인스턴스, 두 번째 인수로 클래스를 받음
- 인스턴스가 그 클래스의 인스턴스인지 판단해 참이면 `True`, 거짓이면 `False`를 돌려줌

```
>>> class Person: pass  ← 아무 기능이 없는 Person 클래스 생성
...
>>> a = Person()  ← Person 클래스의 인스턴스 a 생성
>>> isinstance(a, Person)  ← a가 Person 클래스의 인스턴스인지 확인
True
```

✓ a가 Person 클래스가 만든 인스턴스이므로 `True`를 돌려줌

```
>>> b = 3
>>> isinstance(b, Person)  ← b가 Person 클래스의 인스턴스인지 확인
False
```

✓ b는 Person 클래스가 만든 인스턴스가 아니므로 `False`를 돌려줌

❖ lambda

- lambda는 함수를 생성할 때 사용하는 예약어로 def와 동일한 역할을 함
- 함수를 한 줄로 간결하게 만들 때 사용함
- def를 사용해야 할 정도로 복잡하지 않거나 def를 사용할 수 없는 곳에 쓰임

[함수명 =] lambda 매개변수1, 매개변수2, ... : 매개변수를 사용한 표현식

```
>>> add = lambda a, b: a+b
>>> result = add(3, 4)
>>> print(result)
7
```

def 함수와 동일

```
>>> def add(a, b)
...     return a + b
...
```

- ✓ add는 두 개의 인수를 받아 서로 더한 값을 돌려주는 lambda 함수임
- ✓ lambda 예약어로 만든 함수는 return 명령어가 없어도 결과값을 돌려줌

- 리스트 내에 lambda 함수 만들기

```
>>> myList = [lambda a, b:a+b, lambda a, b:a*b]
>>> myList
[at 0x811eb2c>, at 0x811eb64>] ← 리스트 myList에 람다 함수가 2개 추가됨
```

- ✓ 리스트 각각의 요소에 lambda 함수를 만들어 바로 사용할 수 있음

```
>>> myList[0]
at 0x811eb2c>
>>> myList[0](3, 4)
7
```

- ✓ 리스트의 첫 번째 요소 myList[0]은 2개의 입력값을 받아 두 값의 합을 돌려주는 lambda 함수임

```
>>> myList[1](3, 4)
12
```

- ✓ 리스트의 두 번째 요소 myList[1]은 2개의 입력값을 받아 두 값의 곱을 돌려주는 lambda 함수임

❖ len

- len(s)은 입력값 s의 길이(요소의 전체 개수)를 돌려주는 함수

```
>>> len("python")
```

```
6
```

```
>>> len([1,2,3])
```

```
3
```

```
>>> len((1, 'a'))
```

```
2
```

❖ list

- `list(s)`는 반복 가능한 자료형 `s`를 입력받아 리스트를 돌려주는 함수

```
>>> list("python")  
['p', 'y', 't', 'h', 'o', 'n']  
>>> list((1,2,3))  
[1, 2, 3]
```

- `list` 함수에 리스트를 입력으로 주면 똑같은 리스트를 복사하여 돌려 줌

```
>>> a = [1, 2, 3]  
>>> b = list(a)  
>>> b  
[1, 2, 3]
```


❖ map

- `map(f, iterable)`은 함수(`f`)와 반복 가능한(`iterable`) 자료형을 입력으로 받음
- `map`은 입력받은 자료형의 각 요소를 함수 `f`가 수행한 결과를 묶어서 돌려주는 함수
- `two_times` 함수는 리스트 요소를 입력받아 각 요소에 2를 곱한 결과값을 돌려 줌

```
#two_times.py
def two_times(numberList):
    result = [ ]
    for number in numberList:
        result.append(number*2)
    return result

result = two_times([1, 2, 3, 4])
print(result)
```

} 실행 부분

결과값: [2, 4, 6, 8]

- map 함수를 사용하면 앞의 내용을 간략하게 작성할 수 있음

```
>>> def two_times(x): return x*2
...
>>> list(map(two_times, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

- ✓ 리스트의 첫 번째 요소인 1이 two_times 함수의 입력값으로 들어가고 $1 * 2$ 의 과정을 거쳐서 2가 됨
- ✓ 총 4개의 요소값이 모두 수행되면 마지막으로 [2, 4, 6, 8]을 돌려줌

- lambda를 사용하면 더 간략하게 작성할 수 있음

```
>>> list(map(lambda a: a*2, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

❖ max

- max(iterable)는 인수로 반복 가능한 자료형을 입력받아 최대값을 돌려주는 함수

```
>>> max([1, 2, 3])
3
>>> max("python")
'y'
```

❖ min

- min(iterable)는 인수로 반복 가능한 자료형을 입력받아 최소값을 돌려주는 함수

```
>>> min([1, 2, 3])
1
>>> min("python")
'h'
```

❖ oct

- oct(x)는 정수 형태의 숫자를 8진수 문자열로 바꾸어 돌려주는 함수

```
>>> oct(34)
'0o42'
>>> oct(12345)
'0o30071'
```

❖ open

- `open(filename, [mode])`은 '파일 이름'과 '읽기 방법'을 입력받아 파일 객체를 돌려주는 함수
- 읽기 방법(mode)을 생략하면 기본값인 읽기 전용 모드(r)로 파일 객체를 만들어 돌려줌

모드	설명
w	쓰기 모드로 파일 열기
r	읽기 모드로 파일 열기
a	추가 모드로 파일 열기
b	바이너리 모드로 파일 열기

✓ b는 w, r, a와 함께 사용됨

```
>>> f = open("binary_file", "rb")
```

✓ rb는 '바이너리 읽기 모드'를 의미함

```
>>> fread = open("read_mode.txt", "r")  
>>> fread2 = open("read_mode.txt")
```

- ✓ fread와 fread2는 동일한 방법임
- ✓ 즉, 모드 부분이 생략되면 기본값으로 읽기 모드인 r을 갖게 됨

```
>>> fappend = open("append_mode.txt", "a")
```

- ✓ 추가 모드(a)로 파일을 여는 예제임

❖ ord

- ord(c)는 문자를 입력으로 받아 아스키 코드 값을 리턴하는 함수

```
>>> ord('a')
97
>>> ord('0')
48
```

❖ pow

- pow(x, y)는 x의 y제곱한 결과값을 리턴하는 함수

```
>>> pow(2, 4)
16
>>> pow(3, 3)
27
```

❖ range

- `range([start,]stop[,step])`는 for문과 함께 자주 사용되는 함수
- 입력받은 숫자에 해당하는 범위 값을 반복 가능한 객체로 만들어 돌려줌

▪ 인수가 하나일 경우

```
>>> list(range(5))  
[0, 1, 2, 3, 4]
```

- ✓ 시작 숫자를 지정해 주지 않으면 range 함수는 0부터 시작함

▪ 인수가 2개일 경우

```
>>> list(range(5, 10))  
[5, 6, 7, 8, 9] ← 끝 숫자 10은 포함되지 않는다는 것에 주의
```

- ✓ 입력으로 주어지는 2개의 인수는 시작 숫자와 끝 숫자를 나타냄

▪ 인수가 3개일 경우

```
>>> list(range(1, 10, 2))  
[1, 3, 5, 7, 9] ← 1부터 9까지, 숫자 사이의 거리는 2  
>>> list(range(0, -10, -1))  
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9] ← 0부터 -9까지, 숫자 사이의 거리는 -1
```

- ✓ 세 번째 인수는 숫자 사이의 거리를 의미함

❖ round

- `round(number[,ndigits])` 함수는 숫자를 입력받아 반올림해 주는 함수

```
>>> round(4.6)
```

```
5
```

```
>>> round(4.2)
```

```
4
```

```
>>> round(5.678, 2)  ← 실수 5.678을 소수점 2자리까지만 반올림하여 표시  
5.68
```


❖ sorted

- sorted(iterable) 함수는 입력값을 정렬한 후 그 결과를 리스트로 돌려주는 함수

```
>>> sorted([3, 1, 2])
[1, 2, 3]
>>> sorted(['a', 'c', 'b'])
['a', 'b', 'c']
>>> sorted("zero")
['e', 'o', 'r', 'z']

>>> a = [3, 1, 2]
>>> sorted(a)
[1, 2, 3]
>>> a
[3, 1, 2]
```

✓ 객체 자체를 정렬하지 않음

❖ sort

- `sort(iterable)` 함수는 리스트 객체 그 자체를 정렬할 뿐 정렬된 결과를 돌려주지 않음

```
>>> a = [3, 1, 2]
>>> result = a.sort()  ← sort 함수로 a 리스트 정렬
>>> print(result)
None  ← 반환값이 없기 때문에 None이 출력됨
>>> a
[1, 2, 3]
```

- ✓ `sort` 함수는 반환값이 없기 때문에 `result` 변수에 저장되는 값이 없음
- ✓ 따라서 `print(result)`를 하면 `None`이 출력됨
- ✓ `sort` 함수를 수행한 후 반환값은 없지만, 리스트 객체 `a`를 확인하면 `[3, 1, 2]`가 `[1, 2, 3]`으로 정렬됨

❖ str

- `str(object)`은 문자열 형태로 객체를 반환하여 돌려주는 함수

```
>>> str(3)
'3'
>>> str('hi')
'hi'
>>> str('hi'.upper())
'HI'
```

❖ sum

- `sum(iterable)`은 입력받은 리스트나 튜플의 모든 요소의 합을 돌려주는 함수

```
>>> sum([1, 2, 3])
6
>>> sum([4, 5, 6])
15
```

❖ tuple

- tuple(iterable)은 반복 가능한 자료형을 입력 받아 튜플 형태로 바꾸어 돌려주는 함수

```
>>> tuple("abc")
('a', 'b', 'c')
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple((1, 2, 3))  ← 튜플이 입력으로 들어오면 그대로 돌려준다
(1, 2, 3)
```

❖ type

- type(object)은 입력값의 자료형이 무엇인지 알려 주는 함수

```
>>> type("abc")
<class 'str'>  ← "abc"는 문자열 자료형
>>> type([ ])
<class 'list'>  ← [ ]는 리스트 자료형
>>> type(open("test", 'w'))
<class '_io.TextIOWrapper'>  ← 파일 자료형
```

❖ zip

- zip(*iterable)은 동일한 개수로 이루어진 자료형을 묶어 주는 역할을 하는 함수

```
>>> list(zip([1, 2, 3], [4, 5, 6]))  
[(1, 4), (2, 5), (3, 6)]
```

```
>>> list(zip([1, 2, 3], [4, 5, 6], [7, 8, 9]))  
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

```
>>> list(zip("abc", "def"))  
[('a', 'd'), ('b', 'e'), ('c', 'f')]
```

전 세계의 파이썬 사용자들이 만든 유용한 프로그램을 모아 놓은 것이 라이브러리임
파이썬 라이브러리는 설치할 때 자동으로 컴퓨터에 설치함

❖ sys

- sys 모듈은 파이썬 인터프리터가 제공하는 변수와 함수를 제어할 수 있게 해주는 모듈
- 명령 행에서 인수 전달하기 - sys.argv

```
C:\wdoit>python test.py abc pey guido
```

- ✓ 명령 프롬프트 창에서 test.py 뒤에 또 다른 값들을 함께 넣어 주면 sys.argv 리스트에 그 값이 추가됨

```
# C:/doit/Mymod/argv_test.py
import sys
print(sys.argv)
```

```
C:\wdoit\WMyMod>python argv_test.py you need python  
['argv_test.py', 'you', 'need', 'python']
```

- ✓ python 명령어 뒤의 모든 것들이 공백을 기준으로 나뉘어서 sys.argv 리스트의 요소가 됨

▪ 강제로 스크립트 종료하기 - sys.exit

```
>>> sys.exit()
```

- ✓ sys.exit는 [Ctrl+Z]나 [Ctrl+D]를 눌러서 대화형 인터프리터를 종료하는 것과 같음
- ✓ 프로그램 파일 내에서 사용하면 프로그램을 중단함

▪ 자신이 만든 모듈 불러와 사용하기 - sys.path

```
>>> import sys
>>> sys.path
[' ', 'C:\\Windows\\SYSTEM32\\python35.zip', 'C:\\Python35\\lib',
'C:\\Python35', 'C:\\Python35\\lib\\site-packages']
>>>
```

- ✓ sys.path는 파이썬 모듈들이 저장되어 있는 위치를 나타냄
- ✓ 즉, 이 위치에 있는 파이썬 모듈들은 경로에 상관없이 어디에서나 불러올 수 있음
- ✓ ' '는 현재 디렉터리를 의미함

```
# C:/doit/Mymod/path_append.py
import sys
sys.path.append("C:/doit/Mymod")
```

- 위와 같이 파이썬 프로그램 파일에서 sys.path.append를 이용해 경로명을 추가함
- 이렇게 하고 난 후에는 C:\\doit\\Mymod 디렉터리에 있는 모듈을 불러와서 사용함

❖ pickle

- pickle은 객체의 형태를 그대로 유지하면서 파일에 저장하고 불러올 수 있게 하는 모듈

```
>>> import pickle
>>> f = open("test.txt", 'wb')
>>> data = {1: 'python', 2: 'you need'}
>>> pickle.dump(data, f)
>>> f.close()
```

- ✓ pickle 모듈의 dump 함수를 이용하여 딕셔너리 객체인 data를 파일에 저장함

```
>>> import pickle
>>> f = open("test.txt", 'rb')
>>> data = pickle.load(f)
>>> print(data)
{1: 'python', 2: 'you need'}
```

- ✓ pickle.dump로 저장한 파일을 pickle.load를 이용해서 원래 있던 딕셔너리 객체 (data) 상태 그대로 불러옴

❖ OS

- OS 모듈은 환경 변수나 디렉터리, 파일 등의 OS 자원을 제어할 수 있게 해주는 모듈
- **내 시스템의 환경 변수 값을 알고 싶을 때 - os.environ**

```
>>> import os
>>> os.environ
environ({'PROGRAMFILES': 'C:\Program Files', 'APPDATA': ... 생략 ...})
>>>
```

- ✓ os.environ은 현재 시스템의 환경 변수에 대한 정보들을 딕셔너리 객체로 돌려줌
- ✓ 객체가 딕셔너리이기 때문에 다음과 같이 호출할 수 있음

```
>>> os.environ['PATH']
'C:\ProgramData\Oracle\Java\javapath; ... 생략 ...'
```

- ✓ 시스템의 PATH 환경 변수에 대한 내용임

▪ 디렉터리 위치 변경하기 - `os.chdir`

```
>>> os.chdir("C:\\WINDOWS")
```

- ✓ `os.chdir`를 사용하면 현재 디렉터리의 위치를 변경할 수 있음

▪ 디렉터리 위치 돌려받기 - `os.getcwd`

```
>>> os.getcwd()  
'C:\\WINDOWS'
```

- ✓ `os.getcwd`는 현재 자신의 디렉터리 위치를 돌려줌

▪ 시스템 명령어 호출하기 - `os.system`

```
>>> os.system("dir")
```

- ✓ 시스템 자체의 프로그램이나 기타 명령어를 파이썬에서 호출할 수 있음
- ✓ `os.system("명령어")`처럼 사용함
- ✓ 현재 디렉터리에서 시스템 명령어인 `dir`을 실행하는 예제임

■ 실행한 시스템 명령어의 결과값 리턴 받기 - os.popen

```
>>> f = os.popen("dir")
```

- ✓ os.popen은 시스템 명령어를 실행시킨 결과값을 읽기 모드 형태의 파일 객체로 돌려줌

```
>>> print(f.read())
```

- ✓ 읽어 들인 파일 객체의 내용을 보기 위해서는 read()를 사용함

■ 기타 유용한 os관련 함수

함수	설명
os.mkdir(디렉터리)	디렉터리를 생성함
os.rmdir(디렉터리)	디렉터리를 삭제함(단, 디렉터리가 비어 있어야 삭제 가능)
os.unlink(파일 이름)	파일을 지움
os.rename(src, dst)	src라는 이름의 파일을 dst라는 이름으로 바꿈

❖ shutil

- shutil은 파일을 복사해 주는 파이썬 모듈
- 파일 복사하기 - `shutil.copy(src, dst)`

```
>>> import shutil  
>>> shutil.copy("src.txt", "dst.txt")
```

- ✓ src.txt 파일과 동일한 내용의 파일이 dst.txt로 복사되는 것을 확인함

❖ glob

- 특정 디렉터리에 있는 파일 이름을 모두 알아야 할 때 사용하는 모듈
- 디렉터리에 있는 파일들을 리스트로 만들기 - glob(pathname)

```
>>> import glob
>>> glob.glob("C:\\doit\\q*")
['C:\\doit\\quiz.py', 'C:\\doit\\quiz.py.bak']
>>>
```

- ✓ glob 모듈은 디렉터리 내의 파일들을 읽어서 돌려줌
- ✓ *, ? 등 메타 문자를 써서 원하는 파일만 읽어 들일 수 있음

❖ tempfile

- 파일을 임시로 만들어서 사용할 때 유용한 모듈
- **tempfile.mktemp()**는 중복되지 않는 임시 파일의 이름을 무작위로 만들어서 돌려줌

```
>>> import tempfile
>>> filename = tempfile.mktemp()
>>> filename
'C:\WINDOWS\TEMP\~275151-0'
```

- **tempfile.TemporaryFile()**은 임시 저장 공간으로 사용할 파일 객체를 돌려줌

```
>>> import tempfile
>>> f = tempfile.TemporaryFile()
>>> f.close() ← 생성한 임시 파일이 자동으로 삭제됨
```

- ✓ 이 파일은 기본적으로 바이너리 쓰기 모드(wb)임
- ✓ f.close()가 호출되면 이 파일 객체는 자동으로 사라짐

❖ time

▪ time.time

```
>>> import time
>>> time.time()
988458015.73417199
```

- ✓ time.time()은 UTC(Universal Time Coordinated, 협정 세계 표준시)를 사용하여 현재 시간을 실수 형태로 돌려주는 함수
- ✓ 1970년 1월 1일 0시 0분 0초를 기준으로 지난 시간을 초 단위로 돌려줌

▪ time.localtime

```
>>> time.localtime(time.time())
time.struct_time(tm_year=2013, tm_mon=5, tm_mday=21, tm_hour=16,
tm_min=48, tm_sec=42, tm_wday=1, tm_yday=141, tm_isdst=0)
```

- ✓ time.localtime은 time.time()에 의해서 반환된 실수값을 이용해서 연도, 월, 일, 시, 분, 초, ...의 형태로 바꾸어 주는 함수

▪ **time.asctime**

```
>>> time.asctime(time.localtime(time.time()))  
'Sat Apr 28 20:50:20 2001'
```

- ✓ time.localtime에 의해서 반환된 튜플 형태의 값을 인수로 받아서 날짜와 시간을 알아보기 쉬운 형태로 리턴하는 함수

▪ **time.ctime**

```
>>> time.ctime()  
'Sat Apr 28 20:56:31 2001'
```

- ✓ time.asctime(time.localtime(time.time()))은 time.ctime()을 사용해 간편하게 표시
- ✓ asctime과 다른 점은 ctime은 항상 현재 시간만을 리턴한다는 점임

▪ **time.strftime**

- ✓ strftime 함수는 시간에 관계된 것을 세밀하게 표현할 수 있는 포맷 코드를 제공

```
time.strftime('출력할 형식 포맷 코드', time.localtime(time.time()))
```

✓ 시간에 관계된 것을 표현하는 포맷 코드

코드	설명	예
%a	요일 줄임말	Mon
%A	요일	Monday
%b	달 줄임말	Jan
%B	달	January
%c	날짜와 시간을 출력함	06/01/01 17:22:21
%d	날(day)	[00,31]
%H	시간(hour)-24시간 출력 형태	[00,23]
%I	시간(hour)-12시간 출력 형태	[01,12]
%j	1년 중 누적 날짜	[001,366]
%m	달	[01,12]
%M	분 ¹⁰⁵	[01,59]

코드	설명	예
%p	AM or PM	AM
%S	초	[00,61]
%U	1년 중 누적 주-일요일을 시작으로	[00,53]
%w	숫자로 된 요일	[0(일요일),6]
%W	1년 중 누적 주-월요일을 시작으로	[00,53]
%x	현재 설정된 Locale에 기반한 날짜 출력	06/01/01
%X	현재 설정된 Locale에 기반한 시간 출력	17:22:21
%Y	연도 출력	2001
%Z	시간대 출력	대한민국 표준시
%%	문자	%
%y	세기 부분을 제외한 연도 출력 ¹⁰⁶	01

```
>>> import time
>>> time.strftime('%x', time.localtime(time.time()))
'05/01/01'
>>> time.strftime('%c', time.localtime(time.time()))
'05/01/01 17:22:21'
```

✓ time.strftime을 사용한 예제임

▪ time.sleep

```
#sleep1.py
import time
for i in range(10):
    print(i)
    time.sleep(1)
```

- ✓ time.sleep 함수는 주로 루프 안에서 많이 사용됨
- ✓ 이 함수를 사용하면 일정한 시간 간격을 두고 루프를 실행할 수 있음
- ✓ 위의 예는 1초 간격으로 0부터 9까지의 숫자를 출력함
- ✓ time.sleep 함수의 인수는 실수 형태로 쓸 수 있음

❖ calendar

- calendar는 파이썬에서 달력을 볼 수 있게 해주는 모듈임
- **calendar.calendar(연도)**로 사용하면 그 해 전체 달력을 볼 수 있음

```
>>> import calendar  
>>> print(calendar.calendar(2015))
```

- **calendar.prcal(연도)**를 사용해도 위와 같은 결과를 얻을 수 있음

```
>>> calendar.prcal(2015)
```

- **calendar.prmonth(년도,월)**로 사용하면 그 달의 달력만 보여줌

```
>>> calendar.prmonth(2015, 12)
```

- **calendar.weekday(연도, 월, 일)** 함수는 그 날짜에 해당하는 요일 정보를 돌려줌

```
>>> calendar.weekday(2015, 12, 31)
3      ← 목요일
```

- ✓ 월요일은 0, 화요일은 1, 수요일은 2, 목요일은 3, 금요일은 4, 토요일은 5, 일요일은 6이라는 값을 돌려줌

- **calendar.monthrange(연도, 월)** 함수는 입력받은 달의 1일이 무슨 요일인지와 그 달이 며칠까지 있는지를 튜플 형태로 돌려줌

```
>>> calendar.monthrange(2015, 12)
(1, 31)
```

- ✓ 2015년 12월 1일은 화요일이고, 이 달은 31일까지 있다는 것을 보여 줌

❖ random

- random은 난수(규칙이 없는 임의의 수)를 발생시키는 모듈
- **random.random()**는 0.0에서 1.0사이의 실수 중에서 난수 값을 돌려줌

```
>>> import random
>>> random.random()
0.53840103305098674
```

- **random.randint(x, y)**는 x에서 y사이의 정수 중에서 난수 값을 돌려줌

```
>>> random.randint(1, 10)
6
>>> random.randint(1, 55)
43
```

- **random.randrange(x, y[,z])**는 x에서 y사이의 정수 중에서 난수 값을 돌려줌

```
>>> random.randrange(0, 20)
5
```

✓ 추가적으로 z 입력 시 z의 배수를 돌려줌

- random 모듈을 이용해서 재미있는 함수를 만들어 봄

```
#random_pop.py
import random
def random_pop(data):
    number = random.randint(0, len(data)-1)
    return data.pop(number)

if __name__ == "__main__":
    data = [1, 2, 3, 4, 5]
    while data:
        print(random_pop(data))
```

← 리스트의 요소 중에서 무작위로 하나를 선택하여 꺼내서 돌려줌

← 꺼내진 요소는 pop 메소드에 의해 사라짐

결과값:

2
3
1
5
4

- random_pop 함수는 random.choice 함수를 사용하면 직관적으로 만들 수 있음

```
def random_pop(data):  
    number = random.choice(data)  
    data.remove(number)  
    return number
```

✓ random.choice 함수는 입력받은 리스트에서 무작위로 하나를 선택하여 돌려줌

- **random.shuffle**는 리스트의 항목을 무작위로 섞고 싶을 때는 사용하는 함수

```
>>> import random  
>>> data = [1, 2, 3, 4, 5]  
>>> random.shuffle(data)  
>>> data  
[5, 1, 3, 4, 2]
```

❖ webbrowser

- webbrowser는 시스템에서 사용하는 기본 웹 브라우저를 자동으로 실행하는 모듈

```
>>> import webbrowser  
>>> webbrowser.open("http://google.com")
```

- ✓ 웹 브라우저를 자동으로 실행시키고, <http://google.com>으로 가게 해줌
- ✓ webbrowser의 open 함수는 웹 브라우저가 실행된 상태이면 입력주소로 이동함

```
>>> webbrowser.open_new("http://google.com")
```

- ✓ open_new 함수는 이미 웹 브라우저가 실행된 상태이더라도 새로운 창으로 해당 주소가 열리도록 함