
Python II



7. Python 클래스
8. Python 예외처리
9. Python 파일처리
10. Python 데이터베이스 프로그래밍
11. Python 네트워크 프로그래밍
12. Python GUI 프로그래밍

7. Python 클래스

Python 클래스에 대한 이해

1. Python 클래스란
2. 클래스 정의와 인스턴스 객체 생성
3. 메서드 정의와 호출
4. 클래스 멤버와 인스턴스 멤버
5. 연산자 중복
6. 장식자
7. 상속

1. Python 클래스란

- 클래스는 새로운 이름 공간을 지원하는 단위이다. 이 이름 공간에는 함수와 변수가 포함될 수 있는데 이 점은 모듈과 유사하다.
- 모듈은 파일 단위로 이름 공간을 구성하고, 클래스는 클래스 이름 공간과 클래스가 생성하는 인스턴스 이름 공간을 각각 갖는다.
- 클래스 이름 공간과 인스턴스 이름 공간은 유기적인 관계로 연결되어 있으며 상속 관계에 있는 클래스 간의 이름 공간도 유기적으로 연결되어 있다.
- 클래스를 정의하는 것은 새로운 자료형을 하나 만드는 것이고, 인스턴스는 이 자료형의 객체를 생성하는 것이다.

1. Python 클래스란

➤ 클래스는 하나의 이름 공간이다.

```
>>> class S1:
    a = 1
>>> S1.a
>>> S1.b = 2          # 클래스 이름 공간에 새로운 이름을 만든다.
>>> S1.b
>>> dir( S1 )         # 속성
>>> del S1.b          # 이름 공간 S1에서 이름 b를 삭제한다.
```

1. Python 클래스란

- 클래스 인스턴스(Class Instance)는 클래스의 실제 객체이다. 인스턴스 객체도 독자적인 이름 공간을 갖는다. 클래스는 하나 이상의 인스턴스 객체를 생성하는 자료형과 같다.

```
>>> x = S1()           # S1 클래스의 인스턴스 객체 x를 생성
>>> x.a = 10           # 클래스 인스턴스 x의 이름 공간에 이름 a 생성
>>> x.a
>>> S1.a               # 클래스 이름 공간과 인스턴스 이름 공간은 다르다.
>>> y = S1()           # 또 다른 클래스 인스턴스 생성
>>> y.a = 300          # 인스턴스 객체 y의 이름 공간에 이름 a 생성
>>> y.a
>>> x.a                # x 이름 공간의 이름 a를 확인한다.
>>> S1.a
```

1. Python 클래스란

- 클래스는 상속(Inheritance)이 가능하다. 상속받은 클래스(Subclass, 하위클래스)는 상속해 준 클래스(Superclass, 상위클래스)의 모든 속성을 자동으로 물려받는다.
- 따라서 상속받은 클래스는 물려받은 속성 이외에 추가로 필요한 개별적인 속성만을 정의하면 된다.

```
>>> class A:
    def f( self ):
        print( 'base' )

>>> class B( A ):
    pass
# 클래스 A의 속성을 모두 상속받는다.

>>> b = B()

>>> b.f()
# 클래스 A의 메서드 f가 호출된다.
```


1. Python 클래스란

- 클래스는 연산자 중복을 지원한다. 파이썬이 사용하는 모든 연산자(산술/논리 연산자, 슬라이싱, 인덱싱등)의 동작을 직접 재정의할 수 있다.
- 연산자를 중복하면 내장 자료형과 비슷한 방식으로 동작하는 클래스를 설계할 수 있다.

```
>>> class MyClass:
    def __add__( self, x ):    # __add__는 + 연산자를 중복한다.
        print( 'add {} called'.format( x ) )
        return x

>>> a = MyClass()
>>> a + 3
```

1. Python 클래스란

➤ 파이썬에서 클래스 관련 용어

- 클래스(Class) : class문으로 정의하며, 멤버와 메서드를 가지는 객체이다.
- 클래스 객체(Class Object) : 클래스와 같은 의미로 사용한다. 클래스는 특정 대상을 가리키지 않고 일반적으로 언급하기 위해서 사용하는 반면에, 클래스 객체는 어떤 클래스를 구체적으로 지정하기 위해 사용하기도 한다.
- 클래스 인스턴스(Class Instance) : 클래스를 호출하여 생성된 객체이다.
- 클래스 인스턴스 객체(Class Instance Object) : 클래스 인스턴스와 같은 의미이다. 인스턴스 객체라고 부르기도 한다.
- 멤버(Member) : 클래스 혹은 클래스 인스턴스 공간에 정의된 변수이다.
- 메서드(Method) : 클래스 공간에 정의된 함수이다.
- 속성(Attribute) : 멤버와 메서드 전체를 가리킨다. 즉, 이름 공간의 이름 전체를 의미한다.
- 상속(Inheritance) : 상위 클래스의 속성과 행동을 그대로 받아들이고 추가로 필요한 기능을 클래스에 덧붙이는 것이다. 소프트웨어의 재사용 관점에서 상속은 대단히 중요한 역할을 하며, 프로그램의 개발 시간을 단축해 준다. 다른 프로그래밍 기법과 객체지향 프로그래밍을 구분하는 중요한 특징이다. A 클래스를 상위 클래스로 하는 클래스 B를 만들면 B 'is-a' A 관계가 성립한다.

1. Python 클래스란

- 상위 클래스(Superclass) : 기반 클래스(Base class)라고 하기도 한다. 어떤 클래스의 상위에 있으며 각종 속성을 하위 클래스로 상속해 준다.
- 하위 클래스(Subclass) : 파생 클래스(Derived class)라고 하기도 한다. 상위 클래스로부터 상속받는 하위의 클래스를 말한다. 상위 클래스로부터 각종 속성을 상속받으므로 코드와 변수를 공유한다.
- 다형성(Polymorphism) : 상속 관계 내의 다른 클래스의 인스턴스들이 같은 메서드 호출에 대해 각각 다르게 반응하도록 하는 기능이다.
- 정보 은닉(Encapsulation) : 메서드와 멤버를 클래스 내에 포함하고 외부에서 접근할 수 있도록 인터페이스만을 공개한다. 그리고 다른 속성은 내부에 숨기는 것이다.
- 다중 상속(Multiple Inheritance) : 두 개 이상의 상위 클래스로부터 상속받는 것을 말한다.

2. 클래스 정의와 인스턴스 객체 생성

➤ 클래스 구조

```
class 클래스 이름( 부모 클래스명 ):
```

```
    <클래스 변수들>
```

```
    def __init__( self, [, 인수1, 인수2, ... ] ):           # 생성자
```

```
        <수행할 문장들>
```

```
    def __del__( self ):                                     # 소멸자
```

```
        <수행할 문장들>
```

```
    def 클래스 함수1( self, [, 인수1, 인수2, ... ] ):      # 클래스 메서드
```

```
        <수행할 문장들>
```

- ❖ 클래스 내부에서 선언할 수 있는 클래스 변수와 클래스 함수의 개수는 제한되어 있지 않다.

2. 클래스 정의와 인스턴스 객체 생성

➤ 클래스 정의

```
>>> class Simple:           # Header, 실행문이다. 따라서 아무 곳에서나 기술할 수 있다.
    pass                     # Body, 들여쓰기가 된 상태로 기술. pass 키워드는 아무 일도
                             # 하지 않는, 자리를 채우는 문이다.
```

```
>>> Simple
```

❖ 파이썬 클래스는 object 클래스를 기반(Base) 클래스로 한다.

```
>>> Simple.__bases__
>>> object.__dict__
```

```
>>> s1 = Simple()           # 인스턴스 객체 생성
>>> s2 = Simple()
>>> s1
```

- ❖ 인스턴스 객체를 생성하는 방법은 클래스를 호출하면 된다. 클래스를 호출하면 인스턴스 객체를 생성한 후 해당하는 참조를 반환한다.
- ❖ 각각의 인스턴스 객체도 독립적인 이름 공간을 갖는다.

2. 클래스 정의와 인스턴스 객체 생성

```
>>> s1.stack = []                # 인스턴스 s1에 stack 생성
>>> s1.stack.append( 1 )         # 값 추가
>>> s1.stack.append( 2 )
>>> s1.stack.append( 3 )
>>> s1.stack                     # 인스턴스 s1.stack 값 출력
>>> s1.stack.pop()              # 값을 읽어내고 삭제
>>> s1.stack.pop()
>>> s1.stack
>>> s2.stack                     # Error, s2에는 stack을 정의한 적이 없다.
>>> del s1.stack
```

- ❖ 동적으로 외부에서 멤버를 생성할 수 있다. 클래스는 하나의 이름 공간에 불과하다. 클래스 인스턴스 객체 s1은 클래스 Simple안에 내포된 독립적인 이름 공간을 가지며, 이 이름 공간에서는 동적으로 이름을 설정하는 것이 가능하다.

3. 메서드 정의와 호출

➤ 일반 메서드 정의와 호출

- 클래스 내부에서 메서드를 정의하는 방법은 일반 함수를 정의하는 방법과 동일하다. 다른 점은 메서드의 첫 번째 인수는 반드시 해당 클래스의 인스턴스 객체이어야 한다. 관례로 `self`란 이름으로 첫 번째 인수를 선언한다.

```
>>> class MyClass:
    def set( self, v ):          # 메서드의 첫 인수 self는 반드시 인스턴스 객체이어야 한다.
        self.value = v
    def get( self ):
        return self.value
```

- ❖ 메서드 호출하는 방법 첫 번째는 클래스를 이용하여 호출하는 것으로 언바운드 메서드 호출(Unbound Method Call)이라 부른다.

```
>>> c = MyClass()              # 인스턴스 객체 생성
>>> MyClass.set                # 메서드 확인
>>> MyClass.set( c, 'egg' )    # 언바운드 메서드 호출
>>> MyClass.get( c )
```

3. 메서드 정의와 호출

- ❖ 메서드 호출 방법 두 번째는 인스턴스 객체를 이용하여 호출하는 것으로 바운드 메서드 호출(Bound Method Call)이라 부른다. 두 번째 방법으로 메서드를 호출하면 첫 인수(self)로 인스턴스 객체가 자동으로 전달된다.

```
>>> c = MyClass()
```

```
>>> c
```

```
>>> c.set                                     # set() 메서드는 c에 바운드되어 있다.
```

- ❖ MyClass.set 메서드는 이미 인스턴스 객체 c와 연결되어 있다는 것을 알 수 있다. 즉, c.set의 set() 메서드는 이미 인스턴스 객체로 c와 연결되어 있다는 의미이다.

```
>>> c.set( 'spam' )                          # 바운드 메서드 호출
```

- ❖ 이미 인스턴스 객체로 c와 연결되어 있으므로 self를 명시적으로 전달할 필요가 없다. c.set()의 호출에서 self는 인스턴스 객체 c가 암시적으로 전달한다.

```
>>> c.get()
```


3. 메서드 정의와 호출

➤ 클래스 내부에서 메서드 호출

- 클래스 내부에서 인스턴스 멤버나 클래스 메서드를 호출할 때는 인수 self를 이용해야 한다.

```
>>> class MyClass2:
    def set( self, v ):
        self.value = v
    def get( self ):
        return self.value
    def increase( self ):
        self.value += 1          # 인스턴스 멤버 참조
        return self.get()       # 메서드 호출

>>> c2 = MyClass2()
>>> c2.set( 1 )
>>> c2.get()
>>> c2.increase()
```

- ❖ 클래스의 멤버나 메서드를 참조하려면 언제나 self를 이용하는 것을 잊지 말자!!!

3. 메서드 정의와 호출

➤ 생성자/소멸자

- 파이썬은 이름 공간을 관리하는 측면에서 동적인 특성이 강하다. 따라서 클래스의 인스턴스 객체를 생성할 때 인스턴스 멤버가 자동으로 생성되지는 않는다.

```
>>> c2 = MyClass2()          # 빈 이름 공간이 만들어진다.
>>> c2.set( 1 )              # 이름 공간에 value가 만들어 진다.
>>> c2.get()
```

❖ 클래스에서 사용될 멤버들은 인스턴스 객체를 생성하면서 먼저 정의하고 초기화되는 것이 일반적이다.

- 클래스는 생성자(Constructor)와 소멸자(Destructor)라 불리는 메서드를 정의할 수 있다. 생성자는 인스턴스 객체가 생성될 때 초기화를 위해 자동으로 호출되는 초기화 메서드이고, 소멸자는 인스턴스 객체를 사용하고서 메모리에서 제거할 때 자동으로 호출되는 메서드이다.

- 파이썬 클래스에서 생성자/소멸자를 위해 특별한 이름을 정해놓았다.

- ❖ 생성자 함수 이름 `__init__`
- ❖ 소멸자 함수 이름 `__del__`

3. 메서드 정의와 호출

```
>>> class MyClass3:
    def __init__( self ):          # 인스턴스 객체가 생성될 때 자동 호출
        self.value = 0
    def set( self, v ):
        self.value = v
    def get( self ):
        return self.value

>>> c3 = MyClass3()
>>> c3.get()                     # 멤버 value가 0으로 초기화되어 있다.

>>> from time import time, ctime, sleep
>>> class Life:
    def __init__( self ):          # 생성자
        self.birthday = ctime(); print( 'Birthday', self.birthday )
    def __del__( self ):           # 소멸자
        print( 'Deathday', ctime() )

>>> life = Life()                # 인스턴스 객체가 생성되면 자동으로 생성자 호출
>>> del life                     # 참조 횟수가 0이 되면 소멸자가 호출되고 인스턴스 객체가
                                # 제거된다.
```

3. 메서드 정의와 호출

- ❖ 소멸자(__del__)는 자주 정의되지는 않는다. 대부분의 메모리나 자원 관리가 자동으로 이루어지기 때문에 특별한 조치를 취하지 않아도 인스턴스 객체가 제거되면서 자원이 원상 복귀되기 때문이다.

```
>>> class Member:
    def __init__( self, name, nick, birthday ):
        self.name = name
        self.nick = nick
        self.birthday = birthday
    def getName( self ):
        return self.name
    def getNick( self ):
        return self.nick
    def getBirthday( self ):
        return self.birthday
```

```
>>> m1 = Member( 'hong', 'h', '2018-01-01' )
```

```
>>> m1.getName()
```

```
>>> m2 = Member( name = 'kim', nick = 'k', birthday = '2017-01-01' )
```

- ❖ 인스턴스 객체를 생성할 때 열거된 인수들은 생성자 인수에 전달된다. self는 결과적으로 m1과 같은 참조가 되며, 나머지는 순서대로 전달된다. 함수에서와 같은 사용 방법이 적용된다.

4. 클래스 멤버와 인스턴스 멤버

- 멤버에는 클래스 멤버(Class Member)와 인스턴스 멤버(Instance Member) 두 가지가 있다
 - 클래스 멤버는 클래스의 이름 공간에 생성된다.
 - 인스턴스 멤버는 인스턴스 객체의 이름 공간에 생성된다.
 - 클래스 멤버는 모든 인스턴스 객체에 의해서 공유된다.
 - 인스턴스 멤버는 각각의 인스턴스 객체 내에서만 참조된다.

```
>>> class Var:
    c_mem = 100                # 클래스 멤버
    def f( self ):
        self.i_mem = 200      # 인스턴스 멤버
    def g( self ):
        return self.i_mem, self.c_mem
```

- 클래스 멤버는 메서드 바깥에 정의한다. 인스턴스 멤버는 메서드 내부에서 self를 이용하여 정의한다.
- 클래스 내부에서 멤버를 참조할 때는 'self.멤버명'과 같은 형식을 사용한다.

4. 클래스 멤버와 인스턴스 멤버

- 클래스 외부에서 참조할 때는 다음과 같은 형식으로 호출할 수 있다.
 - 클래스 멤버 클래스.멤버(혹은 인스턴스 멤버)
 - 인스턴스 멤버 인스턴스.멤버
 - ❖ 인스턴스.멤버 형식을 사용할 때 인스턴스 객체의 이름 공간에 멤버가 없으면 클래스의 멤버로 인식한다.

```
>>> Var.c_mem                    # 클래스 객체를 통하여
>>> v1 = Var()
>>> v1.c_mem                    # 인스턴스 객체를 통하여
>>> v1.f()                      # 인스턴스 멤버 i_mem을 생성
>>> v2 = Var()
```

- ❖ self.c_mem 형식이나 인스턴스.멤버 형식으로 멤버를 참조할 때, 검색 순서
 - ① 먼저 인스턴스 객체의 이름 공간에서 멤버를 참조한다.
 - ② 만일 인스턴스 객체의 이름 공간에 멤버가 없으면 클래스의 멤버를 참조한다.
- ❖ 클래스 멤버는 클래스의 모든 인스턴스 객체가 공유하는 멤버이고, 인스턴스 멤버는 각각의 인스턴스 객체가 별도로 가지고 있는 멤버이다. 즉, 각 인스턴스 객체의 특성을 나타낸다고 할 수 있다.

4. 클래스 멤버와 인스턴스 멤버

```
>>> v1.c_mem          # 클래스 멤버 참조
>>> v2.c_mem          # 클래스 멤버 참조
>>> v1.c_mem = 50      # 인스턴스 객체의 이름 공간에 c_mem을 생성
>>> v1.c_mem          # 인스턴스 멤버 참조
>>> v2.c_mem          # 인스턴스 멤버가 없으므로 클래스 멤버 참조
>>> Var.c_mem         # 클래스 멤버 참조
```

- 클래스에 정의한 이름 이외의 속성을 만들 수 없게 하는 것이 필요하다면 `__slots__` 속성을 이용한다. 클래스 멤버 `__slots__`는 리스트로 설정 가능한 속성의 이름을 갖는다. `__slots__` 속성에 등록되지 않은 이름은 사용할 수 없게 된다. 이 속성이 사용되는 경우에는 이름 공간을 표현하는 `__dict__`는 사용되지 않는다.

```
>>> class Person:
    __slots__ = [ 'name', 'tel' ]
>>> m1 = Person()
>>> m1.name = 'hong'          # __slots__ 속성에 등록된 속성
>>> m1.tel = '1234'          # __slots__ 속성에 등록된 속성
>>> m1.address = 'seoul'     # Error, __slots__ 속성에 등록되지 않은 속성
```

5. 연산자 중복

- 연산자 중복(Operator Overloading)은 프로그램 언어에서 지원하는 연산자에 대해 클래스가 새로운 동작을 정의하는 것이다. 파이썬에서는 내장 자료형에 사용하는 모든 연산자를 클래스 내에서 새롭게 정의할 수 있다.
- 수치 연산자 중복

1. 이항 연산자

```
>>> class MyStr:
    def __init__( self, s ):
        self.s = s
    def __truediv__( self, b ):
        return self.s.split( b )
    def __add__( self, b ):
        return self.s + b

>>> s1 = MyStr( 'a:b:c:' )
>>> s1 / ':'
>>> s1 + ':d'
```

나누기(/) 연산자 중복

더하기(+) 연산자 중복

나누기(/) 연산이 가능해졌다.

더하기(+) 연산이 가능해졌다.

- ❖ 파이썬은 모든 연산자에 대응하는 적절한 이름의 메서드가 정해져 있어서 연산자가 사용 될 때 해당 메서드로 확장된다.

5. 연산자 중복

➤ 수치 연산자 메서드

메서드	연산자
<code>__add__(self, other)</code>	<code>+</code>
<code>__sub__(self, other)</code>	<code>-</code>
<code>__mul__(self, other)</code>	<code>*</code>
<code>__truediv__(self, other)</code>	<code>/</code>
<code>__floordiv__(self, other)</code>	<code>//</code>
<code>__mod__(self, other)</code>	<code>%</code>
<code>__divmod__(self, other)</code>	<code>divmod()</code>
<code>__pow__(self, other[, modulo])</code>	<code>pow()</code> <code>**</code>
<code>__lshift__(self, other)</code>	<code><<</code>
<code>__rshift__(self, other)</code>	<code>>></code>
<code>__and__(self, other)</code>	<code>&</code>
<code>__xor__(self, other)</code>	<code>^</code>
<code>__or__(self, other)</code>	<code> </code>

5. 연산자 중복

2. 역 이항 연산자 : 피연산자의 순서가 바뀐 경우의 연산자 중복

```
>>> class MyStr:
    def __init__( self, s ):
        self.s = s
    def __truediv__( self, b ):
        return self.s.split( b )
    def __add__( self, b ):
        return self.s + b
    def __radd__( self, b ):
        return b + self.s
```

```
>>> s1 = MyStr( 'a:b:c:' )
```

```
>>> 'z:' + s1
```

❖ a + b 연산에서 a.__add__(b)를 우선 시도하고, 이것이 구현되어 있지 않으면 b.__radd__(a)를 시도한다.

5. 연산자 중복

➤ 피연산자가 바뀐 경우의 수치 연산자 메서드

메서드	연산자
<code>__radd__(self, other)</code>	<code>+</code>
<code>__rsub__(self, other)</code>	<code>-</code>
<code>__rmul__(self, other)</code>	<code>*</code>
<code>__rtruediv__(self, other)</code>	<code>/</code>
<code>__rfloordiv__(self, other)</code>	<code>//</code>
<code>__rmod__(self, other)</code>	<code>%</code>
<code>__rdivmod__(self, other)</code>	<code>divmod()</code>
<code>__rpow__(self, other[, modulo])</code>	<code>pow()</code> <code>**</code>
<code>__rlshift__(self, other)</code>	<code><<</code>
<code>__rrshift__(self, other)</code>	<code>>></code>
<code>__rand__(self, other)</code>	<code>&</code>
<code>__rxor__(self, other)</code>	<code>^</code>
<code>__ror__(self, other)</code>	<code> </code>

5. 연산자 중복

3. 확장 산술 연산자 메서드 : `s += b -> a.__iadd__(b)`

메서드	연산자
<code>__iadd__(self, other)</code>	<code>+=</code>
<code>__isub__(self, other)</code>	<code>-=</code>
<code>__imul__(self, other)</code>	<code>*=</code>
<code>__itruediv__(self, other)</code>	<code>/=</code>
<code>__ifloordiv__(self, other)</code>	<code>//=</code>
<code>__imod__(self, other)</code>	<code>%=</code>
<code>__ipow__(self, other[, modulo])</code>	<code>**=</code>
<code>__ilshift__(self, other)</code>	<code><<=</code>
<code>__irshift__(self, other)</code>	<code>>>=</code>
<code>__iand__(self, other)</code>	<code>&=</code>
<code>__ixor__(self, other)</code>	<code>^=</code>
<code>__ior__(self, other)</code>	<code> =</code>

5. 연산자 중복

4. 단항 연산자와 형변환 연산자 메서드

`-s` \rightarrow `s.__neg__()`

`+s` \rightarrow `s.__pos__()`

`abs(s)` \rightarrow `s.__abs__()`

메서드	연산자
<code>__neg__(self)</code>	-
<code>__pos__(self)</code>	+
<code>__abs__(self)</code>	<code>abs()</code>
<code>__invert__(self)</code>	~(비트 반전)

5. 연산자 중복

5. 기타 형변환 메서드

메서드	연산자
<code>__complex__(self)</code>	<code>complex()</code>
<code>__int__(self)</code>	<code>int()</code>
<code>__float__(self)</code>	<code>float()</code>
<code>__round__(self)</code>	<code>round()</code>
<code>__index__(self)</code>	<code>operator.index()</code>

❖ 객체를 인수로 하여 함수 형태로 호출되며, 이에 따라 적절한 값을 반환해 주어야 한다.

`complex(s)` \rightarrow `s.__complex__()`

`int(s)` \rightarrow `s.__int__()`

`float(s)` \rightarrow `s.__float__()`

`round(s)` \rightarrow `s.__round__()`

`operator.index(s)` \rightarrow `s.__index__()`

5. 연산자 중복

- ❖ `a.__index()` 메서드는 `operator.index(a)`에 의해 호출되며, `a`를 정수로 변환할 수 있는지 확인해서 정수 값을 반환한다. 만일 `a`가 정수가 아니면 `TypeError`가 발행한다. 인덱스로 사용할 자료형은 정수로만 표현해야 하는데 이를 검사하기 위해서 `__index__()` 메서드가 존재한다.

```
>>> a = 5
```

```
>>> a.__index__()          # 정수는 인덱스로 사용할 수 있다. operator.index( 5 )
```

```
>>> a = 5.5
```

```
>>> a.__index__()          # float는 인덱스로 사용할 수 없다. operator.index( 5.5 )
```

- ❖ `bin()`, `hex()`, `oct()`와 같은 함수들은 정수 인수가 필요하며 내부적으로 `__index__()` 메서드를 사용한다.

<code>operator.index(s)</code>	<code>-> s.__index__()</code>
<code>bin(s)</code>	<code>-> bin(s.__index__())</code>
<code>hex(s)</code>	<code>-> hex(s.__index__())</code>
<code>oct(s)</code>	<code>-> oct(s.__index__())</code>

5. 연산자 중복

```
>>> class Index:
    def __index__( self ):
        print( '__index__ called' )
        return 3

>>> l = [ 1, 2, 3, 4, 5 ]
>>> i = Index()
>>> l[ i: ]
>>> bin( i )
>>> oct( i )
>>> hex( i )
```


5. 연산자 중복

- 컨테이너 자료형(시퀀스 자료형 + 매핑 자료형)의 연산자 중복
 - 기본적으로 `__len__()`, `__contains__()`, `__getitem__()`, `__setitem__()`, `__delitem__()` 메서드를 구현해야 한다.
 - 시퀀스형이면서 변경 가능한 자료형을 만들겠다면 `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()`, `sort()` 메서드 등을 구현하면 된다.
 - 사전과 같은 매핑 자료형은 `keys()`, `values()`, `items()`, `get()`, `clear()`, `copy()`, `setdefault()`, `pop()`, `popitem()`, `update()` 같은 메서드를 구현하면 된다.
 - 산술 연산으로는 `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()`, `__imul__()` 메서드 등을 구현하면 된다.
 - 반복자를 지원하려면 `__iter__()` 메서드를 구현한다.

5. 연산자 중복

➤ 시퀀스 자료형 메서드

메서드	연산자
<code>object.__len__(self)</code>	<code>len(object)</code>
<code>object.__contains__(self, item)</code>	<code>item in object</code>
<code>object.__getitem__(self, key)</code>	<code>object[key]</code>
<code>object.__setitem__(self, key, value)</code>	<code>object[key] = value</code>
<code>object.__delitem__(self, key)</code>	<code>del object[key]</code>

5. 연산자 중복

1. 인덱싱 : 인덱싱은 시퀀스 자료형에서 순서에 의해서 데이터에 접근하기 위한 방법을 제공한다. 기본적으로는 `__getitem__()` 메서드를 정의해야 한다.

```
>>> class Square:
    def __init__( self, end ):
        self.end = end
    def __len__( self ):
        return self.end
    def __getitem__( self, k ):
        if type( k ) != int:
            raise TypeError( '...' )
        if k < 0 or self.end <= k:
            raise IndexError( 'index {} out of range'.format( k ) )
        return k * k

>>> s1 = Square( 10 )
>>> len( s1 )           # s1.__len__()
>>> s1[ 4 ]             # s1.__getitem__( 4 )
>>> s1.__getitem__( 4 ) # s1[ 4 ]
>>> s1[ 20 ]            # Error
>>> s1[ 'a' ]           # Error
```

5. 연산자 중복

```
>>> for x in s1:
```

```
    print( x, end = ' ' )
```

- ❖ for 문은 인스턴스 객체 s1의 `__getitem__()` 메서드를 0부터 호출하기 시작한다. 인스턴스 객체 s1은 제한된 범위 내에서 시퀀스형 객체로서의 역할을 충실히 수행한다.

```
>>> list( s1 )
```

```
>>> tuple( s1 )
```

- ❖ `__getitem__()` 메서드만 정의되어 있으면, 다른 시퀀스 자료형으로 변환하는 것이 가능하다.
- ❖ 이와 같은 방식으로 연산을 지원하면 내장 자료형과 사용법이 같은 일관된 연산을 적용할 수 있고, 이것은 통일성과 편리함을 가져다준다.
- ❖ 사용자가 정의한 클래스가 기존 자료형의 코딩 스타일을 그대로 따르므로 일관된 코딩 스타일을 유지할 수가 있다.
- ❖ `__getitem__()` 메서드가 치환 연산자 오른쪽에서 인덱싱에 의해서 호출되는 메서드라면 `__setitem__()` 메서드는 치환 연산자 왼쪽에서 호출되는 메서드이다. `__delitem__()` 메서드는 `del` 을 사용시 호출된다.

5. 연산자 중복

2. 슬라이싱 : 슬라이싱은 인덱싱에서와 같이 `__getitem__()`, `__setitem__()`, `__delitem__()` 메서드를 사용하지만 인수가 아닌 `slice` 객체를 전달한다.

- `slice` 객체는 `start`와 `step`, `stop` 세 개의 멤버를 가지는 단순한 객체로 이해하면 된다.

```
slice( [start, ] stop [, step] )
```

```
>>> s = slice( 1, 10, 2 )
```

```
>>> s
```

```
>>> type( s )
```

```
>>> s.start, s.stop, s.step
```

```
>>> slice( 10 )
```

인수가 생략되면 `None` 객체를 기본값으로 가진다.

```
>>> slice( 1, 10 )
```

```
>>> slice( 1, 10, 3 )
```

- ❖ 슬라이싱 `m[1:5]`는 `m.__getitem__(slice(1, 5))`를 호출한다. 즉, 인덱싱의 정수 인덱스 대신에 `slice` 객체가 범위를 나타내는데 사용된다. 확장 슬라이싱 `m[1:10:2]`는 `m.__getitem__(slice(1, 10, 2))`를 호출한다.

5. 연산자 중복

```
>>> class Square:
    def __init__( self, end ):
        self.end = end
    def __len__( self ):
        return self.end
    def __getitem__( self, k ):
        if type( k ) == slice:           # slice 자료형인가?
            start = k.start or 0
            stop = k.stop or self.end
            step = k.step or 1
            return map( self.__getitem__, range( start, stop, step ) )
        elif type( k ) == int:           # 인덱싱
            if k < 0 or self.end <= k:
                raise IndexError( 'index {} out of range'.format( k ) )
            return k * k
        else:
            raise TypeError( '...' )
```

5. 연산자 중복

```
>>> s = Square( 10 )
>>> s[ 4 ]                # 인덱싱
>>> list( s[ 1:5 ] )      # 슬라이싱
>>> list( s[ 1:10:2 ] )   # 간격은 2로
>>> list( s[ : ] )        # 전체 범위
```

- ❖ `__getitem__()` 메서드 정의에서 `k`의 자료형이 `slice` 형인지 검사해서 참이면 슬라이싱을, 아니면 인덱싱을 적용한다. 슬라이싱 부분에서 `start`, `stop`, `step`을 별도의 지역 변수에 치환한 이유는 `range()` 함수가 정수 인수만을 요구하기 때문이다. 최종적으로 `map()` 함수에 의해서 인덱스 값의 제공에 대한 리스트를 반환한다.

5. 연산자 중복

3. 매핑 자료형 : 매핑 자료형에서 `object.__getitem__(self, key)` 등의 메서드의 `key`는 사전의 키로 사용할 수 있는 임의의 객체가 될 수 있다. 만일 `key`에 대응하는 값을 찾을 수 없으면 `KeyError` 를 발생시킨다.

```
>>> class MyDict:
    def __init__( self ):
        self.d = {}
    def __getitem__( self, k ):
        return self.d[ k ]
    def __setitem__( self, k, v ):
        self.d[ k ] = v
    def __len__( self ):
        return len( self.d )

>>> m = MyDict()                    # __init__()
>>> m[ 'day' ] = 'light'            # m.__setitem__( 'day', 'light' )
>>> m.__setitem__( 'night', 'darkness' ) # m[ 'night' ] = 'darkness'
>>> m[ 'day' ]; m[ 'night' ]        # m.__getitem__( 'night' )
>>> len( m )                        # __len__()
```

5. 연산자 중복

➤ 문자열 변환 연산

- 인스턴스 객체를 `print()` 함수로 출력할 때 내가 원하는 형식으로 출력하거나, 인스턴스 객체를 사람이 읽기 좋은 형태로 변환하려면 문자열로 변환하는 기능이 필요하다.

1. 문자열로의 변환 : `__str__()`과 `__repr__()` 메서드

인스턴스 객체를 문자열로 변환하는 메서드는 `__str__()`과 `__repr__()` 두 가지이다. 이 두 메서드는 호출되는 시점이 다르다.

```
>>> class StringRepr:
    def __repr__( self ):
        return 'repr called'
    def __str__( self ):
        return 'str called'

>>> s = StringRepr()
>>> print( s )
>>> str( s )
>>> repr( s )
```

5. 연산자 중복

- ❖ `print()` 함수와 `str()` 함수에 의해서 `__str__()` 메서드가 호출되며, `repr()` 함수에 의해서 `__repr__()` 메서드가 호출된다. `__repr__()` 메서드의 목적은 객체를 대표해서 유일하게 표현할 수 있는 문자열을 만들어 내는 것이다. 즉, 다른 객체의 출력과 혼동되지 않는 모양으로 표현해야 한다는 의미이다.

```
>>> repr( 2 )
```

```
>>> repr( '2' )
```

```
>>> repr( 'abc' )           # 문자열 'abc'에 대한 repr 문자열
```

```
>>> repr( [ 1, 2, 3 ] )     # 리스트 [ 1, 2, 3 ]에 대한 repr 문자열
```

- ❖ `__str__()` 메서드의 목적은 사용자가 읽기 편한 형태의 표현으로 출력한다.

```
>>> str( 2 )
```

```
>>> str( '2' )
```

- ❖ 컨테이너 자료형(리스트와 사전 등)의 `__str__()` 메서드는 내부 객체의 `__repr__()` 메서드를 사용한다.

```
>>> l = [ 2, '2' ]
```

```
>>> str( l )
```

```
>>> repr( l )
```

```
>>> str( l ) == repr( l )
```

5. 연산자 중복

- ❖ 만일 `__str__()` 메서드를 호출할 상황에서 `__str__()` 메서드가 정의되어 있지 않으면 `__repr__()` 메서드가 대신 호출된다.

```
>>> class StringRepr:
    def __repr__( self ):
        return 'repr called'

>>> s = StringRepr()
>>> str( s )
>>> repr( s )
```

- ❖ 그러나 `__repr__()` 메서드가 정의되어 있지 않은 경우에 `__str__()` 메서드가 `__repr__()` 메서드를 대신하지 않는다.

```
>>> class StringRepr:
    def __str__( self ):
        return 'str called'

>>> s = StringRepr()
>>> str( s )
>>> repr( s )
```

5. 연산자 중복

2. 바이트로의 변환 : `__bytes__()` 메서드

문자열이 아닌 바이트 자료형으로 변환하려면 `__bytes__()` 메서드를 사용한다. `b.__bytes__()` 메서드는 `bytes(b)` 함수에 의해 호출된다.

```
>>> class BytesRepr:
    def __bytes__( self ):
        return 'bytes called'.encode( 'utf-8' )

>>> b = BytesRepr()
>>> bytes( b )
```

3. 서식 기호 새로 지정하기 : `__format__()` 메서드

`__format__()` 메서드는 `format()` 함수나 문자열의 `format()` 메서드에 의해서 호출된다.

```
>>> x = 10
>>> format( x, "0" )           # x.__format__( "0" )
>>> "x:{:0}".format( x )      # x.__format__( "0" )
```

5. 연산자 중복

- ❖ `__format__()` 메서드는 `format()` 함수나 문자열의 `format()` 메서드에 의해서 호출된다. 변환 기호가 요구될 때는 `__format__()` 메서드가 호출된다.

```
>>> class MyStr:
```

```
    def __init__( self, s ):
```

```
        self.s = s
```

```
    def __format__( self, fmt ):
```

```
        print( fmt )
```

서식 문자열을 확인한다.

```
        if fmt[ 0 ] == 'u':
```

u이면 대문자로 변환한다.

```
            s = self.s.upper()
```

```
            fmt = fmt[ 1: ]
```

```
        elif fmt[ 0 ] == 'l':
```

l이면 소문자로 변환한다.

```
            s = self.s.lower()
```

```
            fmt = fmt[ 1: ]
```

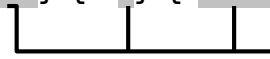
```
        else:
```

```
            s = str( self.s )
```

```
        return s.__format__( fmt )
```

```
>>> s = MyStr( 'Hello' )
```

```
>>> print( '{0:u^20} {0:l} {0:*^20}'.format( s ) )
```

 `__format__` 메서드의 인수로 입력되며, 반환 값으로 치환된다.

5. 연산자 중복

➤ 진리값과 비교 연산

1. `__bool__()` 메서드

클래스 인스턴스의 진리값은 `__bool__()` 메서드의 반환 값으로 결정된다. 만일 이 메서드가 정의되어 있지 않으면 `__len__()` 메서드를 호출한 결과가 0 이면 False로 간주하고 아니면 True로 간주한다. 만일, `__len__()` 과 `__bool__()` 메서드 모두가 정의되어 있지 않으면 모든 인스턴스는 True가 된다.

```
>>> class Truth:
    def __init__( self, num ):
        self.num = num
    def __bool__( self ):
        return self.num != 0

>>> bool( Truth( 0 ) )
>>> bool( Truth( 3 ) )
```

5. 연산자 중복

1. 비교 연산 : 모든 비교 연산은 중복이 가능하도록 메서드 이름이 준비됨

연산자	메서드
<	object.__lt__(self, other)
<=	object.__le__(self, other)
>	object.__gt__(self, other)
>=	object.__ge__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)

- ❖ $x < y$ 는 $x.__lt__(y)$ 메서드로 확장되며, $x \leq y$ 는 $x.__le__(y)$ 메서드로 확장된다. 다른 연산자도 같은 방식으로 적용된다. `__eq__()`와 `__ne__()` 메서드는 각자의 논리로 적용될 수 있다. 즉, $o == other$ 이 참이라고 해서 $o != other$ 가 거짓이 아닐 수도 있다는 것이다. 하지만, `__ne__()` 메서드가 정의되어 있지 않고, `__eq__()` 메서드만 정의되어 있을 경우 $o != other$ 는 `not(o == other)`의 논리가 적용된다.

5. 연산자 중복

```
>>> class Compare:
    def __init__( self, n ):
        self.n = n
    def __eq__( self, o ):
        print( '__eq__ called' )
        return self.n == o
    def __lt__( self, o ):
        print( '__lt__ called' )
        return self.n < 0
    def __le__( self, o ):
        print( '__le__ called' )
        return self.n <= 0

>>> c = Compare( 10 )
>>> c < 10                # __lt__() 메서드
>>> c <= 10               # __le__() 메서드
>>> c > 10                # Error, __gt__가 정의되어 있지 않다.
>>> c == 10               # __eq__() 메서드
>>> c != 10               # __ne__() 메서드나 not __eq__() 메서드 결과
```


5. 연산자 중복

➤ 해시 값에 접근하기 : `__hash__()` 메서드

- 해시 값을 돌려주는 내장 함수 `hash(m)`가 호출될 때 `m.__hash__()` 메서드가 호출된다. `hash()` 함수를 사용한 예로, 사전은 (키, 값)쌍을 저장할 때 키에 대한 `hash()` 함수의 호출 결과를 값을 저장하기 위한 해시 키로 사용한다. `__hash__()` 메서드는 정수를 반환해야 한다. 이 메서드를 정의한 클래스는 `__eq__()` 메서드로 함께 정의해야 해시 가능한 객체로 취급한다.

```
>>> class Obj:
    def __init__( self, a, b ):
        self.a = a
        self.b = b
    def _key( self ):
        return ( self.a, self.b )
    def __eq__( self, o ):
        return self._key() == o._key()
    def __hash__( self ):
        return hash( self._key() )
```

5. 연산자 중복

```
>>> o1 = Obj( 1, 2 )
>>> o2 = Obj( 3, 4 )
>>> hash( o1 )
>>> hash( o2 )
>>> d = { o1:1 o2:2 }
```

- ❖ 해시 키는 변경이 가능해서는 안 된다. 만일 변경 가능한 자료형으로 클래스를 정의하면 hash() 함수를 호출할 때 TypeError 예러를 반환해서 해시 키로 사용할 수 없도록 해야 한다.

```
>>> class Obj2:
    def __init__( self, a, b ):
        self.a = a
        self.b = b
    def __hash__( self ):
        raise TypeError( 'not proper type' )

>>> o1 = Obj2( 1, 2 )
>>> d = { o1:1 }                # 키로 사용할 수 없다.
```

5. 연산자 중복

➤ 속성 값에 접근하기

인스턴스 객체의 속성을 다루는 메서드

메서드	설명
<code>__getattr__(self, name)</code>	정의되어 있지 않은 속성을 참조할 때, 이 메서드가 호출된다. 속성 이름 name은 문자열이다.
<code>__getattribute__(self, name)</code>	<code>__getattr__()</code> 메서드와 같으나 속성이 정의되어 있어도 호출된다.
<code>__setattr__(self, name, value)</code>	<code>self.name = value</code> 와 같이 속성에 치환(대입)이 일어날 때 호출된다.
<code>__delattr__(self, name)</code>	<code>del self.name</code> 에 의해서 호출된다.

1. `__getattr__()`와 `__getattribute__()` 메서드

인스턴스 객체에 대한 일반적인 접근 방법인 `obj.attr()` 메서드는 `getattr(obj, 'attr')`로 수행한다.

- `__getattr__()` 메서드는 이름 공간에 정의되지 않는 이름에 접근할 때 호출되며 이에 대해서 처리를 할 수 있다.

5. 연산자 중복

```
>>> class GetAttr1( object ):
    def __getattr__( self, x ):
        print( '__getattr__', x )
        if x == 'test':
            return 10
        raise AttributeError

>>> g1 = GetAttr1()
>>> g1.c = 10
>>> g1.c          # 정의된 이름을 호출
>>> g1.a          # 정의되지 않은 이름을 호출
>>> g1.test       # 정의되지 않았지만 준비된 이름
```

- `__getattr__()` 메서드는 이름 정의 여부에 관계없이 모든 속성에 접근하면 호출된다. 따라서 `__getattr__()` 메서드는 호출되는 이름 전체에 대한 제어권을 얻어낸다.

5. 연산자 중복

```
>>> class GetAttr2( object ):
    def __getattr__( self, x ):
        print( '__getattr__ called..', x )
        return object.__getattr__( self, x )

>>> g2 = GetAttr2()
>>> g2.c = 10
>>> g2.c          # 정의된 이름 호출
>>> g2.a          # 정의되지 않은 이름 호출
```

5. 연산자 중복

```
>>> class GetAttr3( object ):
    def __getattr__( self, x ):
        print( '__getattr__', x )
        raise AttributeError
    def __getattribute__( self, x ):
        print( '__getattribute__ called..', x )
        return object.__getattribute__( self, x )

>>> g3 = GetAttr3()
>>> g3.c = 10
>>> g3.c                # 정의된 속성에 접근
>>> g3.a                # 정의되지 않은 속성에 접근
```

- ❖ 주의할 점은 `self.__getattribute__(x)`와 같이 호출해서는 안 된다. 재귀적으로 자기 자신을 무한히 호출하게 되므로 상위 클래스를 통해서 `object.__getattribute__(self, x)`와 같은 식으로 접근해야 한다.

5. 연산자 중복

2. `__setattr__()`와 `__delattr__()` 메서드

인스턴스 객체에서 속성을 설정할 때는 `__setattr__()` 메서드를, 속성을 삭제할 때는 `__delattr__()` 메서드를 사용한다.

`obj.x = o`는 `setattr(obj, 'x', o)`로 수행되며 `obj.__setattr__('x', o)`를 호출하고, `del obj.x`는 `delattr(obj, 'x')`로 수행되며 `obj.__delattr__('x')`를 호출한다.

```
>>> class Attr:
    def __setattr__( self, name, value ):
        print( '__setattr__((s)=%s called' % ( name, value ) )
        object.__setattr__( self, name, value )
    def __delattr__( self, name ):
        print( '__delattr__((s) called' % name )
        object.__delattr__( self, name )

>>> a = Attr()
>>> a.x = 10
>>> del a.x
```

5. 연산자 중복

➤ 인스턴스 객체를 호출하기

1. `__call__()` 메서드

어떤 클래스 인스턴스가 `__call__()` 메서드를 가지고 있으면, 해당 인스턴스 객체는 함수와 같은 모양으로 호출할 수 있다. 인스턴스 객체 `x`에 대해 다음과 같이 확장된다.

`x(a1, a2, a3)` \rightarrow `x.__call__(a1, a2, a3)`

- ❖ 클래스 `Factorial`은 고속 처리를 위하여 기억 기법(Memorization Technique)을 사용한다. 한 번 계산된 팩토리얼 값은 인스턴스 객체의 `cache` 멤버에 저장되어 있다가 필요할 때 다시 사용한다. 팩토리얼 계산은 인스턴스 객체의 `__call__()` 메서드를 호출하여 이루어진다.

5. 연산자 중복

```
>>> class Factorial:
    def __init__( self ):
        self.cache = {}
    def __call__( self, n ):
        if n not in self.cache:
            if n == 0:
                self.cache[ n ] = 1
            else:
                self.cache[ n ] = n * self.__call__( n - 1 )
        return self.cache[ n ]

>>> fact = Factorial()
>>> for i in range( 10 ):
    print( '{} != {}'.format( i, fact( i ) ) )
```

5. 연산자 중복

2. 호출 가능한지 확인하기

어떤 객체가 호출한지 알아보려면 `collections.Callable`의 인스턴스 객체인지 확인한다.

```
>>> import collections
>>> def f():
    pass
>>> isinstance( f, collections.Callable )# 함수 객체를 확인한다.
>>> fact = Factorial()
>>> isinstance( fact, collections.Callable )
```

5. 연산자 중복

➤ 인스턴스 객체 생성하기 : `__new__()` 메서드

클래스의 `__init__()` 메서드는 객체가 생성된 이후에 객체를 초기화하기 위해 호출되는 메서드이다. 반면 `__new__()`는 객체의 생성을 담당하는 메서드로 `__new__()` 메서드에 의해서 생성된 객체가 `__init__()` 메서드에 의해서 초기화된다. `__new__()` 메서드는 object 클래스의 `__new__()` 메서드를 통해서 인스턴스 객체를 생성해야 한다.

```
>>> class NewTest:
```

```
    def __new__( cls, *args, **kw ):                # cls는 NewTest
        print( "__new__ called", cls )
        instance = object.__new__( cls )          # 인스턴스 객체를 생성한다.
        return instance
    def __init__( self, *args, **kw ):              # self는 생성된 인스턴스 객체이다.
        print( "__init__ called", self )
```

```
>>> t = NewTest()
```

5. 연산자 중복

- ❖ 만일 `__new__()` 메서드가 인스턴스 객체를 반환하면 `__init__()` 메서드가 호출되지만, 그렇지 않으면 `__init__()` 메서드는 호출되지 않는다.

```
>>> class Super:
    def __new__( cls, *args, **kw ):
        obj = object.__new__( cls )
        obj.data = []
        return obj
```

```
>>> class Sub( Super ):
    def __init__( self, name ):
        self.name = name
```

```
>>> s = Sub( 'hong' )
```

```
>>> s.name
```

```
>>> s.data
```

- ❖ 위 예는 `__new__()` 메서드를 사용하여 멤버 값을 초기화하는 예로, 멤버값의 초기화는 일반적으로 `__init__()` 메서드에서 이루어지지만 상위 클래스의 `__init__()` 메서드를 명시적으로 호출하지 않으면 상위 클래스의 `__init__()` 메서드는 실행되지 않는다. `__new__()` 메서드를 사용하여 상위 클래스의 `__init__()` 메서드 호출 여부와 관계없이 멤버 값의 초기화를 수행 예이다.

5. 연산자 중복

```
>>> class Singleton:
    __instance = None          # 유일한 객체를 저장하기 위한 클래스 변수
    def __new__( cls, *args, **kw ):
        if cls.__instance is None:
            cls.__instance = object.__new__( cls )
        return cls.__instance
>>> class Sub( Singleton ):
    pass
>>> s1 = Sub(); s2 = Sub()
>>> s1 is s2
```

- ❖ 싱글톤(Singleton)이란 인스턴스 객체를 오직 하나만 생성해 내는 클래스를 의미한다. 유일하게 하나만 시스템에 존재해야 하는 객체를 정의할 때 유용하다.

6. 장식자

- 장식자(Decorator)는 함수를 인수로 받는 함수 클로저(Function Closure)이다.
- 함수 클로저를 간단히 말하면 함수 코드와 그 함수의 변수 참조 영역을 묶은 객체라고 할 수 있다. 함수를 인스턴스화하는데 유용하다. 따라서 장식자에 유용하게 활용된다.

```
>>> def wrapper( func ):                # 장식자는 함수 객체를 인수로 받는다.
    def wrapped_func():                  # 내부에서는 wrapper() 함수를 정의한다.
        print( 'before..' ) # 원래 함수 이전에 실행되어야 할 코드이다.
        func()               # 원래 함수이다.
        print( 'after..' ) # 원래 함수 이후에 실행되어야 할 코드이다.
    return wrapped_func              # 원래 함수를 감싼 함수 클로저를 반환한다.

>>> def myfunc():                      # 원래 함수를 정의한다.
    print( 'I am here' )

>>> myfunc()                          # 원래 함수 실행

>>> myfunc = wrapper( myfunc )        # 장식자에 함수를 전달한다.

>>> myfunc()                          # 장식된 함수를 실행한다.
```

6. 장식자

- ❖ 앞의 예에서 `wrapper()` 함수는 장식자이다. 함수를 인수로 받으며 함수 클로저를 반환한다. `myfunc = wrapper(myfunc)`로 반환된(장식된) 함수는 실행할 때 마다 실제로는 `wrapped_func()` 함수가 실행된다. 이것이 장식자이다.
- ❖ 장식자를 좀 더 간단히 표현하는 방법이 있는데 `@wrapper` 형식의 선언을 함수 앞에 하는 것이다.

<code>@wrapper</code>		<code>def f():</code>
<code>def f():</code>	<code>-></code>	<code>~생략</code>
<code>~생략</code>		<code>f = wrapper(f)</code>

```
>>> @wrapper           # 장식자이다.
def myfunc2():         # myfunc2 = wrapper( myfunc2 )
    print( 'I am here 2..' )
>>> myfunc2()          # 장식된 함수를 실행한다.
```

6. 장식자

❖ 장식자를 사용하는 대표적인 예는 정적 메서드와 클래스 메서드이다.

```
>>> class D:
    @staticmethod                # add를 static method로
    def add( x, y ):
        return x + y

>>> class D:
    def add( x, y ):
        return x + y
    add = staticmethod( add )

>>> D.add( 2, 4 )
```

- ❖ 장식자는 다양한 경우에 유용하게 활용된다. 예를 들어, 외부에서 제공되는 수정할 수 없는 라이브러리의 행동 특성을 변경하기 위해서, 소스를 건드리지 않고 디버깅 정보를 출력하기 위해서, 같은 방식으로 여러 함수를 확장하기 위해서 등등이 될 수 있다.
- ❖ 파이썬에서 표준으로 제공되는 장식자들은 거의 없다. 파이썬에서는 단지 기능만을 제공할 뿐 무한한 사용 가능성에 대해서는 사용자에게 일임하고 있다. 마치 클래스나 함수를 만드는 기능을 제공하는 것과 마찬가지로이다. 직접 만들어 보거나 여러 유용한 장식자가 공개되고 있으므로 필요에 따라 검색해 보면 좋을 것이다.

6. 장식자

- 정적 메서드(Static Method)란 인스턴스 객체를 생성하지 않고도, 혹은 인스턴스 객체를 이용하지 않고도 클래스를 이용하여 직접 호출할 수 있는 메서드 이다.
- 일반 메서드는 첫 번째 인수로 인스턴스 객체를 반드시 전달해야 하지만 정적 메서드는 일반 함수와 동일한 방식으로 호출한다.
- 정적 메서드는 일반 메서드와 달리 첫 인수로 self를 받지 않는다. 필요한 만큼의 인수를 선언하면 된다.
- @staticmethod를 메서드 앞에 장식하면 정적 메서드가 된다.

```
>>> class D:
    @staticmethod                # add를 정적 메서드로
    def add( x, y ):
        return x + y

>>> D.add( 2, 3 )                # 인스턴스 객체 없이 클래스에서 직접 호출한다.
>>> d = D()
>>> d.add( 2, 3 )                # 인스턴스 객체를 통해서도 호출할 수 있다.
```

6. 장식자

- ❖ 정적메서드는 인스턴스 객체와 관계없이 실행되어야 하므로 인스턴스 변수를 참조할 수 없다. 대신 클래스 멤버는 참조할 수 있다.

```
>>> class E:
    acc = 0
    @staticmethod
    def accumulate( v ):
        E.acc += v
        return E.acc                                # 클래스 멤버

>>> E.accumulate( 10 )
>>> E.accumulate( 20 )
>>> E.acc
```

6. 장식자

- 클래스 메서드(Class Method)는 첫 인수로 클래스 객체를 전달받는다. 정적 메서드와 같이 클래스를 통하여 직접 호출하는 것이 일반적이지만 첫 인수로 클래스 객체가 전달되는 것이 다르다.
- 클래스 메서드는 @classmethod 장식자에 의해서 선언된다.

```
>>> class CM:
    acc = 0
    @classmethod
    def accumulate( cls, v ):          # 클래스 메서드
        cls.acc += v
        return cls.acc

>>> CM.accumulate( 10 )
>>> CM.accumulate( 20 )
>>> CM.acc

>>> c = CM()
>>> c.accumulate( 5 )                # 인스턴스 객체를 통한 호출이 가능하다.
>>> CM.acc
>>> c.__class__ is CM                 # c.__class__와 CM이 같은 객체인가?
```

6. 장식자

- Property 속성이란 멤버 변수와 같은 접근 방식을 사용하지만 실제로는 메서드의 호출로 처리되는 속성을 말한다. 즉, 메서드로 정의되어 있지만 호출은 멤버 변수를 사용하는 것처럼 한다.
- property 함수를 통해서 속성을 정의할 수 있는데, 이 함수는 변수에 값을 저장하는 메서드(fset), 읽는 메서드(fget), 삭제하는 메서드(fdel)를 지정하고 관련 연산이 이들 메서드를 통해서 이루어지는 객체를 생성한다.

```
property( fget = None, fset = None, fdel = None, doc = None )
```

- ❖ fget은 값을 읽을 때, fset은 값을 쓸 때, fdel은 값을 삭제할 때 자동으로 호출되는 메서드이다.

6. 장식자

```
>>> class PropertyClass:
    def get_deg( self ):
        return self.__deg
    def set_deg( self, d ):
        self.__deg = d % 360; return self.__deg
    deg = property( get_deg, set_deg )
```

```
>>> p = PropertyClass()
```

```
>>> p.deg = 390; p.deg
```

```
>>> p.deg = -370; p.deg
```

❖ 장식자를 이용한 방법

```
>>> class PropertyClass:
    @property                                # getter 메서드를 등록한다.
    def get_deg( self ):
        return self.__deg
    @property                                # deg의 setter 메서드를 등록한다.
    def set_deg( self, d ):
        return self.__deg = d % 360
```

```
>>> p = PropertyClass()
```

```
>>> p.deg = 390; p.deg
```

```
>>> p.deg = -370; p.deg
```

7. 상속

- 상속(Inheritance)은 클래스가 갖는 중요한 특징이다. 상속이 중요한 이유는 재사용성에 있다. 상속받은 클래스는 상속해 준 클래스의 속성을 사용할 수 있으므로, 추가로 필요한 기능만을 정의 하거나, 기존의 기능을 변경해서 새로운 클래스를 만들면 된다.
- 클래스 A에서 상속된 클래스 B가 있다고 할 때, 클래스 A를 기반(Base) 클래스, 부모(Parent) 클래스 또는 상위(Super) 클래스라고 하며, 클래스 B를 파생(Derived) 클래스, 자식(Child) 클래스 또는 하위(Sub) 클래스라 한다.
- 상속 관계는 is-a 관계를 갖는다.
 - ❖ '사람은 포유류이고 포유류는 동물이다.'에서 사람->포유류->동물의 관계가 성립한다. 동물은 포유류의 상위 클래스이고 포유류는 사람의 상위 클래스이다.
 - ❖ is-a 관계는 두 개의 클래스의 계층적인 관계를 따질 때 사용한다.

7. 상속

- ❖ 하위 클래스는 상위 클래스의 모든 속성을 그대로 상속받으므로 하위 클래스에는 상위 클래스에 없는 새로운 기능이나 수정하고 싶은 기능만을 재정의하면 된다.

```
>>> class Person:
    def __init__( self, name, phone = None ):
        self.name = name
        self.phone = phone
    def __repr__( self ):
        return '<Person {} {}>'.format( self.name, self.phone )

>>> Person.__bases__          # base 클래스 확인
```

- ❖ 파이썬의 모든 클래스의 base 클래스는 object 이다.

7. 상속

❖ 상속 표현 I

```
>>> class Employee( Person ):                # base 클래스는 괄호 안에 표현한다.
    def __init__( self, name, phone, position, salary ):
        Person.__init__( self, name, phone ) # Person 클래스 생성자 호출
        self.position = position
        self.salary = salary
```

❖ 상속 표현 II(선호하는 방식)

```
>>> class Employee( Person ):                # base 클래스는 괄호 안에 표현한다.
    def __init__( self, name, phone, position, salary ):
        super().__init__( name, phone )      # 상위 클래스 생성자 호출
        self.position = position
        self.salary = salary
```


7. 상속

```
>>> m1 = employee( 'hong', 5564, '대리', 200 )
>>> m2 = employee( 'kim', 8546, '과장', 300 )
>>> print( m1.name, m1.position )
>>> print( m2.name, m2.postion )
>>> print( m1 )                # Person.__repr__() 메서드 호출
>>> print( m2 )
```

- ❖ 상위 클래스와 하위 클래스는 별도의 이름 공간을 가지며 계층적인 관계를 가진다. 클래스 객체와 인스턴스 객체도 역시 모두 별도의 이름 공간과 계층적인 관계를 가진다.

7. 상속

- 메서드 대치(Override)는 상위 클래스의 같은 메서드를 재정의한 경우로, 기능을 대치하는 효과가 얻는다. 하위 클래스와 상위 클래스에 같은 메서드가 있을 때 하위 클래스의 메서드를 먼저 취하기 때문이다. 즉, 메서드의 검색 우선순위는 하위 클래스이다.

```
>>> class Employee( Person ):
    def __init__( self, name, phone, position, salary ):
        super().__init__( self, name, phone )
        self.position = position
        self.salary = salary
    def __repr__( self ):
        return '<Employee {} {} {} {}>'.format( self.name,
                                                self.phone, self.position, self.salary )

>>> m1 = Employee( 'son', 5564, '대리', 200 )
>>> print( m1 )
```

7. 상속

- 메서드 확장은 하위 클래스에서 그 속성을 변화시키기 위해서 상위 클래스의 메서드를 호출하고, 그 결과를 활용하는 것을 말한다.
- 메서드의 확장과 치환은 하위 클래스에서 상위 클래스 메서드를 호출하느냐 하지 않느냐에 따라 구분된다.

```
>>> class Person:
    def __init__( self, name, phone = None ):
        self.name = name
        self.phone = phone
    def __repr__( self ):
        return '<Person {} {}>'.format( self.name, self.phone )
```

7. 상속

```
>>> class Employee( Person ):
    def __init__( self, name, phone, position, salary ):
        super().__init__( self, name, phone )
        self.position = position
        self.salary = salary
    def __repr__( self ):
        s = super().__repr__()
        return s + '<Employee {} {}>'.format( self.position, self.salary )

>>> p1 = Person( 'lee', 5284 )
>>> print( p1 )
>>> m1 = Employee( 'park', 5564, '대리', 200 )
>>> print( m1 )
```

7. 상속

❖ 상속 클래스 예

```
# inheritance_ex.py
```

```
import math
```

```
class Point:
```

```
    def __init__( self, x, y ):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def area( self ):                # 점의 면적은 0
```

```
        return 0;
```

```
    def move( self, dx, dy ):
```

```
        self.x += dx
```

```
        self.y += dy
```

```
    def __repr__( self ):
```

```
        return 'x = {} y = {}'.format( self.x, self.y )
```

7. 상속

```
class Circle( Point ):
    def __init__( self, x, y, r ):
        super().__init__( x, y )
        self.radius = r
    def area( self ):
        return math.pi * self.radius * self.radius;
    def __repr__( self ):
        return '{} radius = {}'.format( super().__repr__(), self.radius )
```

7. 상속

```
class Cylinder( Circle ):
    def __init__( self, x, y, r, h ):
        super().__init__( x, y, r )
        self.height = h
    def area( self ):    # 원주의 표면적 = 위아래 원의 면적 + 기둥의 표면적
        return 2 * Circle.area( self ) + 2 * math.pi * self.radius * self.height
    def volume( self ): # 체적
        return Circle.area( self ) * self.height
    def __repr__( self ):
        return '{} height = {}'.format( super().__repr__(), self.height )
```

7. 상속

```
if __name__ == '__main__':  
    p1 = Point( 3, 5 )  
    c1 = Circle( 3, 4, 5 )  
    c2 = Cylinder( 3, 4, 5, 6 )  
  
    print( p1 )  
    print( c1 )  
    print( c2 )  
  
    print( c2.area(), c2.volume() )  
    print( c1.area() )  
  
    c1.move( 10, 10 )  
    print( c1 )
```


7. 상속

- 파이썬 클래스의 모든 메서드는 가상 함수이다. 가상 함수란 메서드의 호출이 참조되는 클래스 인스턴스에 따라서 동적으로 결정되는(Dynamic Binding) 함수를 말한다.

```
>>> class Base:
    def f( self ):
        self.g()                # g()를 호출
    def g( self ):
        print( 'Base' )
>>> class Derived( Base ):
    def g( self ):
        print( 'Derived' )
>>> b = Base()
>>> b.f()
>>> a = Derived()              # 객체에 따라서 해당 객체에 연관된 함수 호출되는 것을
>>> a.f()                      # 가상함수라 한다.
```

7. 상속

- 두 개 이상의 클래스로부터 상속받는 것을 다중 상속(Multiple Inheritance)이라고 한다. 다중 상속 클래스 정의는

```
class Employee( Person, Job ):
```

와 같이 기반 클래스 이름들을 나열하면 된다.

7. 상속

❖ 다중 상속 예

```
# multiple_inheritance_ex.py
```

```
class Person:
```

```
    def __init__( self, name, phone = None ):
```

```
        self.name = name
```

```
        self.phone = phone
```

```
    def __repr__( self ):
```

```
        return 'name = {} tel = {}'.format( self.name, self.phone )
```

```
class Job:
```

```
    def __init__( self, position, salary ):
```

```
        self.position = position
```

```
        self.salary = salary
```

```
    def __repr__( self ):
```

```
        return 'position = {} salary = {}'.format( self.position, self.salary )
```

7. 상속

```
class Employee( Person, Job ):  
    def __init__( self, name, phone, position, salary ):  
        Person.__init__( self, name, phone )           # 언바운드 메서드 호출  
        Job.__init__( self, position, salary )          # 언바운드 메서드 호출  
    def raisesalary( self, rate ):  
        self.salary = self.salary * rate  
    def __repr__( self ):  
        # 언바운드 메서드 호출  
        return Person.__repr__( self ) + ' ' + Job.__repr__( self )  
  
if __name__ == '__main__':  
    e = Employee( 'hong', 5244, 'prof', 300 )  
    e.raisesalary( 1.5 )  
    print( e )
```

7. 상속

- ❖ 다중 상속과 단일 상속은 이름 공간이 두 개 이상 연결되어 있다는 점을 제외하고는 다르지 않다. 만일 같은 이름의 두 상위 클래스 모두에 정의되어 있으면 이름을 찾는 순서가 의미가 있게 된다.
- ❖ Employee 클래스는 두 상위 클래스(Person, Job) 중에서 왼쪽에 먼저 기술된 Person 클래스의 이름 공간을 먼저 찾는다.

7. 상속

➤ 메서드 처리 순서

```
>>> class A:
    def __init__( self ):
        pass
```

```
>>> class B:
    def __init__( self ):
        pass
```

```
>>> class AA( A ): pass
```

```
>>> class BB( B ): pass
```

```
>>> class C( AA, BB ): pass
```

```
>>> c = C()
```

❖ `__init__()` 메서드의 검색 순서는 다음과 같다.

C -> AA -> A -> BB -> B -> Object

```
>>> C.__mro__          # 검색 순서 확인
```

```
>>> C.mro()           # 검색 순서 확인
```

7. 상속

- 상위 클래스를 동적으로 얻어내는 `super()` 함수는 다중 상속으로 가면 클래스 상속 관계에 따라서 다른 결과를 내기도 한다.
- `super()` 함수는 `mro()` 함수가 출력하는 클래스 순서에 따라 `super()` 함수를 호출하는 현재 클래스의 다음 클래스를 결과로 반환한다.
- 예를 들어 `mro()` 함수가 [A, B, C]이고 `super()` 함수를 호출하는 클래스가 B이면 `super()` 함수의 출력은 C가 된다.
- 인스턴스 객체의 클래스를 알아내려면 `__class__` 속성을 이용하고, 인스턴스 객체와 클래스와의 관계를 파악하려면 `isinstance(instance, class)` 메서드를 사용한다.

```
>>> class A: pass
```

```
>>> class B: pass
```

```
>>> class C( B ): pass
```

```
>>> c = C(); c.__class__
```

```
# 인스턴스 객체의 클래스
```

```
>>> isinstance( c, A )
```

```
# c는 A의 인스턴스 객체가 아님
```

```
>>> isinstance( c, C )
```

```
# c는 C의 인스턴스 객체
```

```
>>> isinstance( c, B )
```

```
# c는 B의 인스턴스 객체
```

7. 상속

- 두 클래스 간의 상속 관계를 알아내려면 `issubclass()` 함수 사용

```
>>> issubclass( C, B )
```

```
>>> issubclass( C, A )
```

- 어떤 클래스의 상위 클래스를 알아보려면 `__bases__` 멤버를 이용한다.
`__bases__`는 어떠한 클래스로부터 직접 상속받았는지를 튜플로 알려주는 변수이다.

```
>>> class A: pass
```

```
>>> class B( A ): pass
```

```
>>> class C( B ): pass
```

```
>>> C.__bases__          # 바로 위의 상위 클래스 목록
```

- 전체적으로 상위 클래스의 목록을 얻으려면 `inspect` 모듈의 `getmro()` 함수를 사용한다.

```
>>> import inspect
```

```
>>> inspect.getmro( C )
```


7. 상속

- 전체적으로 내포된 클래스 구조를 알고 싶으면 inspect 모듈의 getclasstree() 함수를 사용한다.

```
>>> class A: pass
>>> class AA( A ): pass
>>> class B: pass
>>> class BB( B ): pass
>>> class C( AA, BB ): pass

>>> import inspect
>>> import pprint
>>> pp = pprint.PrettyPrinter( indent = 4 )
>>> pp.pprint( inspect.getclasstree( [C] ) )
```

7. 상속

- 다형성(Polymorphism)이란 '여러 형태를 가진다.'는 의미의 그리스어에서 유래된 말로, 상속 관계에서 다른 클래스의 인스턴스 객체들이 같은 멤버 함수의 호출에 대해 각각 다르게 반응하도록 하는 기능이다.
- 예를 들어, $a + b$ 라는 연산을 수행할 때 $+$ 연산은 객체 a 와 b 에 따라 동적으로 결정된다. $a.__add__(b)$ 가 호출되는 것인데, 객체 a 와 b 가 정수이면 정수형 객체의 $__add__()$ 메서드를, 문자열이면 문자열 객체의 $__add__()$ 가 호출된다. 이처럼 동일한 이름의 연산자라 해도 객체에 따라 다른 메서드가 호출되는 것이 다형성이라 한다.
- 다형성은 객체의 종류에 관계없이 하나의 이름으로 원하는 유사한 작업을 수행시킬 수 있으므로 프로그램의 작성과 코드의 이해를 쉽게 해준다.

7. 상속

- 캡슐화(Encapsulation)라 필요한 메서드와 멤버를 하나의 단위로 묶어 외부에서 접근 가능하도록 인터페이스를 공개하는 것을 의미한다.
- 파이썬에서 캡슐화는 코드를 묶는 것(패키지화하는 것)을 의미하며, 반드시 정보를 숨기는 것이 아님을 유의해야 한다.
- 캡슐화는 완전히 내부 정보가 숨겨지는 방식(Black Box)으로 구현될 수도 있고, 외부에서 접근 가능하도록 공개된 방식(White Box)으로 구현될 수도 있다. 정보를 숨기는 것을 정보 은닉(Information Hiding)이라는 용어를 사용한다.
- 파이썬은 주로 공개 방식의 캡슐화를 주로 사용한다. 파이썬의 모든 정보는 기본적으로 공개되어 있다. 관례로 내부적으로만 사용하거나 차기 버전에서 변경 가능성 있는 이름은 _(밑줄)로 시작한다. 단지 변수가 내부용이라는 것이지 완전히 숨기는 것은 아니다.

7. 상속

- 위임(Delegation)은 상속 체계 대신에 사용되는 기법으로, 어떤 객체가 자신이 처리할 수 없는 메시지(메서드 호출)를 받으면, 해당 메시지를 처리할 수 있는 다른 객체에 전달하는 것이다.
- 또는 다른 객체의 메서드 호출을 중간에 있는 클래스가 대신 위임받아 처리하는 것이다.
- 위임은 상속 체계보다 융통성이 있고 일반적이다.
- 파이썬에서 위임은 `__getattr__()` 메서드로 구현한다. 이 메서드는 정의되지 않는 속성을 참조하려고 했을 때 호출된다.

`__getattr__(self, name)`

- ❖ 참조하는 속성 이름이 `name`을 통해 전달된다. 이 메서드는 구해진 속성값을 전달하거나, 속성 값이 없다는 것을 나타내기 위해 `AttributeError` 예외를 발생시켜야 한다.

7. 상속

```
# delegation.py
class Delegation:
    def __init__( self, data ):
        self.stack = data

    def __getattr__( self, name ):          # 정의되지 않은 속성을 참조할 때 호출
        print( 'Delegation {} '.format( name ), end = ' ' )
        return getattr( self.stack, name)  # self.stack의 속성을 대신 이용

a = Delegation( [ 1, 2, 3, 1, 5 ] )
print( a.pop() )
print( a.count( 1 ) )
```

- ❖ `__getattr__()` 메서드가 모든 메서드를 대신 호출해 주는 것은 아니다. `__getattr__()` 메서드는 `__getitem__()`과 `__repr__()`, `__len__()` 처럼 `__`로 시작하는 메서드를 필요로 하는 이름들은 잡아내지 못한다.
- ❖ `__getattr__()`를 요구하는 `a[0]`을 수행하면 `__getattr__()` 메서드가 수행되지 않고 `AttributeError` 에러가 발생한다.

8. Python 예외 처리

Python 예외 처리에 대한 이해

1. 예외 처리란
2. try~except~else문
3. try문에서 finally절 사용하기
4. raise 문으로 예외 발생시키기
5. assert 문으로 예외 발생시키기

1. 예외 처리란

- 프로그램을 수행하다 보면 문법은 맞으나 실행 중 더 이상 진행할 수 없는 상황이 발생한다. 이것을 예외(Exception)라고 한다.
- 예외 상황에는 여러 가지 경우가 있을 수 있다. 예를 들면, 0으로 숫자 나누기, 문자열과 숫자 더하기, 참조 범위를 넘어서 인덱스 참조하기 등이다.

```
>>> a, b = 5, 0
>>> c = a / b                # ZeroDivisionError 발생
>>> 4 + spam * 3            # NameError 발생
>>> '2' + 2                 # TypeError 발생
```

- 예외가 발생하면 에러 메시지가 나온다. 스택 추적 형태로 상황을 알려주며, 마지막 부분에 최종적으로 에러가 발행한 정보를 표시해 준다.
- 모든 예외는 클래스로 표현된다. 최상위에 있는 클래스는 BaseException 클래스이다.

1. 예외 처리란

<https://docs.python.org/3/library/exceptions.html?highlight=baseexception#exception-hierarchy>

BaseException

--- SystemExit

--- KeyboardInterrupt

--- GeneratorExit

--- Exception

 --- StopIteration

 --- StopAsyncIteration

 --- ArithmeticError

 | --- FloatingPointError

 | --- OverflowError

 | --- ZeroDivisionError

 --- EOFError

 --- ImportError

 --- NameError

 --- RuntimeError

 --- SyntaxError

 --- TypeError

 --- ValueError

 | --- UnicodeError

 --- Warning

 --- SyntaxWarning

2. try~except~else문

- 예외가 발생하면 프로그램에서 try ~ except 문을 사용하여 예외를 잡아낼 수 있다. 사용하는 구문의 형식은 다음과 같다.

try:

 <문1>

except <예외 종류1>:

 <문2>

<문1>을 수행하는 중 <예외 종류1>이 발생하면 수행

else:

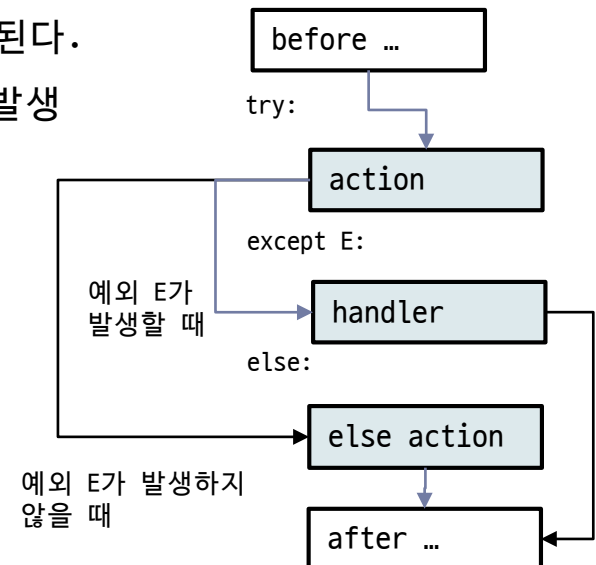
else이하는 생략할 수 있다.

 <문3>

예외가 발생하지 않으면 <문3>가 수행된다.

- 처리 순서는, 우선 try 절(try와 except 사이)이 실행된다.

예외가 발생하지 않으면 else 절을 수행한다. 예외가 발생하면 except 절을 수행하고 try ~ except 문을 마친다. else 절은 생략할 수 있다.



2. try~except~else문

```
>>> x = 0
>>> try:
    print( 1.0 / x )
except ZeroDivisionError:                # 0으로 나누기 exception
    print( 'has no inverse' )

>>> name = 'notexistingfile'
>>> try:
    f = open( name, 'r' )
except IOError:                          # IO exception
    print( 'cannot open', name )
else:
    print( name, 'has', len( f.readlines() ), 'lines' )
    f.close()
```

2. try~except~else문

```
>>> try:
    spam()
except NameError as x:      # NameError 객체를 x로 받는다.
    print( x )
```

```
>>> def this_fails():
    x = 1 / 0
```

```
>>> try:
    this_fails()
except ZeroDivisionError as err:
    print( 'Runtime error : ', err )
```

- try:는 try 절에서 발생하는 예외뿐 아니라, 간접적으로 호출한 함수의 내부 예외도 처리한다.

2. *try~except~else*문

```
>>> try:
    some_job()
except ZeroDivisionError:
    ~ 생략 ~
except NameError:
    ~ 생략 ~
except ( TypeError, IOError ):           # 두 예외는 같은 루틴을 공유한다.
    ~ 생략 ~
else:
    ~ 생략 ~
```

- 여러 개의 예외가 발생할 수 있고, 각 경우마다 별도의 처리를 원한다면 except 절을 여러 번 사용할 수 있다.

```
>>> try:
    some_job()
except:
    ~ 생략 ~
```

- except 절에 아무것도 기술하지 않으면 모든 예외를 처리한다.

2. *try~except~else*문

- 예외 클래스는 같은 종류의 예외 간에 계층적인 관계를 가지고 있으므로 이를 이용하면 여러 가지의 예외를 한꺼번에 검출해 낼 수 있다. `except` 절에 기술된 예외 클래스는 하위의 파생 클래스의 예외까지 함께 처리할 수 있다.

```
>>> def dosomething():  
    a = 1 / 0  
  
>>> try:  
    dosomething()  
except ArithmeticError:  
    print( 'Exception occurred' )
```

2. *try~except~else*문

- 좀 더 구체적으로 어떤 예외가 발생하는지 알고 싶다면 sys 모듈의 exc_info() 함수를 사용할 수 있다. exc_info() 함수는 예외 클래스, 예외 인스턴스 객체, traceback 객체를 반환한다. 예외 정보를 출력하려면 traceback 모듈의 print_exception()나 print_exc() 함수를 사용할 수 있다.

```
>>> import sys
>>> import traceback
>>> x = 0
>>> try:
    1 / x
except:
    etype, evalue, tb = sys.exc_info()
    print( 'Exception class = ', etype )
    print( 'value = ', evalue )
    print( 'traceback object = ', tb )
    traceback.print_exception( etype, evalue, tb )
    traceback.print_exc()
```

3. try 문에서 finally절 사용하기

- try 문에서 finally 절을 사용할 수 있는데, finally 절에 포함된 문들은 예외 발생 여부에 관계없이 모두 수행된다.

```
>>> f = open( filename, 'w' )
>>> try:
    do_something_with( f )
finally:
    f.close()
```

- do_something_with(f) 문에서 예외가 발행하지 않으면 finally 절의 f.close()를 수행하고, 예외가 발생해도 f.close()를 수행한다. 이 상황은 어떤 경우에도 파일을 close 할 경우에 유용하게 사용된다.

```
>>> x = 0
>>> try:
    1 / x
except:
    print( sys.exc_info() )
finally:
    print( 'All the time' )
```


4. *raise* 문으로 예외 발생시키기

- 이미 시스템에 내장 되어있는 예외를 *raise* 문을 이용하여 발생시킬 수 있다. *raise* 문은 예외 클래스의 인스턴스 객체를 매개 변수로 받아들인다.

```
>>> raise IndexError( "range error" )
```

- 만일 *raise* 문이 인수 없이 사용된다면 가장 최근에 발생했던 예외가 다시 발행한다. 만일 최근에 발생한 예외가 없다면 *RuntimeError* 가 발생한다.

```
>>> try:
```

```
    raise IndexError( 'a' )
```

```
except:
```

```
    print( 'in except...' )
```

```
    raise
```

- 이름 공간을 이용하면 예외 객체에 정보를 넘기는 것도 가능하다.

```
>>> ie = IndexError( 'a' ); ie.value = 10; ie.count = 3
```

```
>>> try:      raise ie
```

```
except IndexError as x:
```

```
    print( x, x.value, x.count )
```

4. *raise* 문으로 예외 발생시키기

- 사용자 예외를 발생시키는 표준 방법은 `BaseException` 클래스의 하위 클래스를 이용하는 것이다. 상속받을 수 있는 클래스는 내장 예외의 어떤 것일 수도 있다.

```
# user_defined_exception.py
import sys

class Big( Exception ):
    pass

class Small( Big ):
    pass

def dosomething():
    raise Big( 'big exception' )

def dosomething2():
    raise Small( 'small exception' )

for f in (dosomething, dosomething2 ):
    try:  f()
    except Big: print( sys.exc_info() )
```

5. assert 문으로 예외 발생시키기

- 예외를 발생시키는 특수한 경우로 assert 문에 의한 AssertionError 에러가 있다. assert 문은 주로 디버깅할 때 많이 사용한다. 프로그램이 바르게 진행되는지를 시험할 때 사용한다.

assert <시험 코드>, <데이터>

- <시험 코드>가 거짓이면 'raise AssertionError, 데이터 ' 예외를 발생시킨다. <시험 코드>가 참이면 그냥 통과한다. <데이터>는 사용하지 않아도 된다.

```
# asserttest.py
```

```
a = 30
```

```
margin = 2 * 0.2
```

```
assert margin > 10, 'not enough margin'
```

- ❖ 파이썬을 -O 옵션으로 실행하면 assert 문은 컴파일된 바이트 코드로부터 자동으로 삭제된다. 또한, -O 옵션은 __debug__ 플래그가 0이 되어 디버깅 상태가 아님을 알린다.

9. Python 파일 처리

Python 파일 처리에 대한 이해

1. 파일 처리 구조
2. 파일 처리 함수
3. 파일 처리 예제

1. 파일 처리 구조

- 표준 입/출력 함수는 표준 장치에 대한 입/출력을 담당하였다. 하지만 때로는 출력 결과를 파일에 보관하거나, 파일로 부터 읽어서 처리를 수행해야 하는 경우를 파일 처리라고 한다.

- 파일 처리 기본 절차
 1. 파일 open
 2. 파일에 데이터 쓰기 or 파일에서 데이터 읽기
 3. 파일 close

2. 파일 처리 함수

➤ 파일을 open할 때 사용하는 파이썬 함수는 open()이다.

파일 객체 = open("파일이름", "파일 open mode")

파일 open mode	설명
r	읽기 모드(Default), 스트림은 파일의 시작 부분에 위치
w	쓰기 모드, 파일이 있으면 모든 내용을 삭제, 없으면 생성, 스트림은 파일의 시작 부분에 위치
x	쓰기 모드, 파일이 있으면 오류 발생, 스트림은 파일의 시작 부분에 위치
a	쓰기 모드, 파일이 있으면 내용을 끝에 추가, 파일이 없으면 생성, 스트림은 파일의 끝에 위치
+	읽기/쓰기 모드, r/w/a와 조합하여 사용한다.
t	텍스트 모드, 텍스트 문자 기록에 사용(Default), r/w/a와 조합하여 사용한다.
b	바이너리 모드, 바이트 단위 데이터 기록에 사용, r/w/a와 조합하여 사용한다.

2. 파일 처리 함수

➤ 파일 open() 사용 예

`f = open('file.txt', 'rt')` # Default, 텍스트 읽기, rt는 생략 가능

`f = open('file.txt', 'wb')` # 바이너리 쓰기

`f = open('file.txt', 'r+t')` # 텍스트 읽기/쓰기, 없으면 에러, 맨 앞부터 덮어쓴다

`f = open('file.txt', 'w+t')` # 텍스트 읽기/쓰기, 파일 내용 지우고 처음부터 다시 쓴다

`f = open('file.txt', 'a+t')` # 텍스트 읽기/쓰기, 파일의 맨 뒤에서 부터 쓴다

❖ '+'가 포함되면 파일모드는 모두 읽기쓰기 모드이지만 기존 파일 내용을 처리한 방식에 차이가 있다.

➤ 파일을 close할 때 사용하는 파이썬 함수는 close()이다.

파일 객체.close()

❖ 열려 있는 파일 객체를 닫아 주는 역할을 한다. 파일 객체를 닫지 않아도 프로그램이 종료할 때 파이썬 인터프리터가 열려 있는 파일 객체를 자동으로 닫아준다. 하지만 close()를 사용해서 열려 있는 파일을 직접 닫아 주는 것이 데이터 손실을 방지하는 좋은 방법이다.

2. 파일 처리 함수

- 파일에서 데이터를 읽는 방법은 다음 함수를 사용한다.

파일 객체.readline() # 한 줄 읽기

파일 객체.readlines() # 모든 줄을 읽어 한 줄 단위로 구분된 리스트로 반환

파일 객체.read() # 모든 문자열 읽기

- 파일에 데이터를 기록 하는 방법은 다음 함수를 사용한다.

파일 객체.write('데이터') # 한 개의 데이터 기록

파일 객체.writelines([데이터 집합]) # 여러 개의 데이터 기록

❖ 줄바꿈이 필요하면 직접 '\n'을 데이터에 추가해야 한다.

- 파일 포인터 관련 함수

파일 객체.tell() # 현재 파일 포인터의 위치 반환

파일 객체.seek(p) # 파일 포인터의 위치를 p로 이동

3. 파일 처리 예제

❖ 파일 생성

```
>>> f = open( 'c:\\workspace\\file.txt', 'w' )  
>>> f.close()
```

❖ 텍스트 파일에 기록

```
#writedata.py  
f = open( 'c:\\Workspace\\file.txt', 'w' )  
for x in range( 1, 11 ):  
    data = '{:>2} 번째 줄입니다.\n'.format( x )  
    f.write( data )  
f.close()
```

3. 파일 처리 예제

❖ 파일로부터 하나의 데이터 읽기

```
# readline.py
f = open( 'c:\\Workspacefile.txt', 'r' )
line = f.readline()
print( line )
f.close()
```

❖ 파일로부터 여러 데이터 읽기

```
# readline_all.py
f = open( 'c:\\Workspace\\file.txt', 'r' )
while True:
    line = f.readline()
    if not line:
        break
    print( line )
f.close()
```

3. 파일 처리 예제

❖ 파일로부터 여러 데이터 읽기

```
# readlines.py
f = open( 'c:\\Workspace\\file.txt', 'r' )
lines = f.readlines()
for line in lines:
    print( line )
f.close()
```

```
# read.py
f = open( 'c:\\Workspace\\file.txt', 'r' )
data = f.read()
print( data )
f.close()
```

3. 파일 처리 예제

❖ 파일에 여러 데이터 추가

```
# adddata.py
```

```
f = open( 'c:\\Workspace\\file.txt','a' )
```

```
for x in range( 11, 20 ):
```

```
    data = '{:>2} 번째 줄입니다.\n'.format( x )
```

```
    f.write( data )
```

```
f.close()
```

3. 파일 처리 예제

❖ 파일 포인터 함수 사용

```
# filepointer.py
f = open( 'c:\\Workspace\\file.txt', 'r' )
print( '파일 현재 위치:', f.tell() )
print( f.read() )
print( '파일 현재 위치:', f.tell() )
print()
f.seek( 0 )                      # 처음으로
print( '파일 현재 위치:', f.tell() )
print( f.readline(), end='' )
print( '파일 현재 위치:', f.tell() )
f.close()
```

3. 파일 처리 예제

❖ with문을 이용한 파일 처리

```
# file.py
```

```
f = open( 'c:\\Workspace\\foo.txt', 'w')
```

```
f.write( "Life is too short, you need python" )
```

```
f.close()
```

```
# file_with.py
```

```
with open( 'k:\\Workspace\\foo.txt', 'w' ) as f:
```

```
    f.write( "Life is too short, you need python" )
```

❖ with문을 이용하면 with 블록을 벗어나는 순간 열린 파일 객체 f가 자동으로 close되어 편리하다.

3. 파일 처리 예제

❖ 예외 처리를 통한 파일 close

```
# file_exception.py
```

```
try:
```

```
    f = open( 'c:\\Workspace\\file.txt' )
```

```
    for line in f:
```

```
        print( line )
```

```
finally:
```

```
    f.close()
```

```
# file_exception_with.py
```

```
with open( 'c:\\Workspace\\file.txt' ) as f:
```

```
    for line in f:
```

```
        print( line )
```


3. 파일 처리 예제

❖ Binary 파일 쓰기/읽기

```
# binary_write.py
data = [1, 2, 3, 4, 5]
with open( 'c:\\Workspace\\test.bin', 'wb' ) as f:
    f.write( bytes( data ) )
```

❖ binary 파일로 부터 읽고 쓸 때는 기본적으로 bytes type을 사용하게 된다. 리스트를 파일에 쓰기 전에 bytes()를 이용하여 bytes type으로 형 변환후에 wirte한다.

```
# binary_read.py
with open( 'c:\\Workspace\\test.bin', 'rb' ) as f:
    content = f.read()          # 모두 읽음
    print( type( content ) )    # bytes type
    for b in content:
        print( b )
```

❖ read()가 리턴하는 객체의 type은 bytes type이다.

3. 파일 처리 예제

❖ png 파일 내용 읽기

binary_read_png.py

```
with open( 'c:\\Workspace\\python.png', 'rb' ) as f:
```

```
    byte = f.read( 1 )
```

```
    while byte != b"":
```

```
        print( byte )
```

```
        byte = f.read( 1 )
```

3. 파일 처리 예제

❖ pickle 모듈을 이용한 binary 파일 처리

```
import pickle
f = open( 'c:\\Workspace\\file.dat', 'wb' )
pickle.dump( 'hello world', f )           # string 덤프
pickle.dump( 12345, f )                   # int 덤프
pickle.dump( 3.14, f )                     # float 덤프
pickle.dump( [ 'one', 'two', 'three', 'four' ], f ) # list 덤프
pickle.dump( { 1:'first', 2:'second' }, f ) # dict 덤프
f.close()
```

```
f = open( 'c:\\Workspace\\file.dat', 'rb' )
d = pickle.load( f ); print( type( d ) ); print( d ) # string 로드
d = pickle.load( f ); print( type( d ) ); print( d ) # int 로드
d = pickle.load( f ); print( type( d ) ); print( d ) # float 로드
d = pickle.load( f ); print( type( d ) ); print( d ) # list 로드
d = pickle.load( f ); print( type( d ) ); print( d ) # dict 로드
f.close()
```

3. 파일 처리 예제

- ❖ binary 파일을 다루기 위해 pickle 모듈을 사용할 수 있다.
- ❖ pickle 모듈을 사용하여 파이썬에서 사용되는 객체를 binary 파일로 저장할 수 있다.
- ❖ binary 파일에 쓰기 위해 dump(), 읽기 위해 load()를 사용한다.
- ❖ 보안에 문제가 있을 수 있기 때문에 모르는 파일을 로드해서는 안된다.

10. Python 데이터베이스 프로그래밍

Python 데이터베이스 프로그래밍에 대한 이해

1. Python 데이터베이스 개요
2. Python 데이터베이스 API
3. SQLite

1. Python 데이터베이스 개요

- 정보화 시대에 광범위한 데이터를 관리하기 위해서 DBMS를 이용한다.
- DBMS(Database Management System)는 데이터베이스의 데이터에 대한 생성, 저장, 수정, 조회 등을 처리할 수 있는 소프트웨어 시스템이다.
- DBMS의 규모는 개인용 컴퓨터에서 실행되는 소형 시스템에서부터 메인프레임에서 실행되는 대형 시스템에 이르기까지 다양하다.
- 파이썬은 관계형 데이터베이스에 대한 표준 인터페이스를 제공한다.
Python Database API 인터페이스V2.0(<https://www.python.org/dev/peps/pep-0249/>)
- Python Database API 인터페이스는 데이터베이스의 종류와는 관계없이 최소한 지원해야 하는 기능을 정의하고 있다.
- 파이썬은 SQLite3 데이터베이스를 자체 지원하지만, 다른 데이터베이스에 연결해서 사용하려면 해당 데이터베이스의 파이썬 인터페이스 모듈을 설치해야 하고, 이들 모듈은 Python Database API 인터페이스 규정을 따른다.

2. Python 데이터베이스 API

- 파이썬은 데이터베이스와 상호작용할 수 있다.
- 파이썬은 데이터베이스와 인터페이스하기 위해 Python Database API를 사용한다. 이 API를 이용하면 다른 DBMS를 이용한 프로그래밍을 할 수 있다.
- DBMS를 다루기 위한 처리 절차
 1. 선택한 데이터베이스에 대한 연결(connection)을 설정
 2. 데이터 전달을 위한 커서(cursor)를 생성
 3. SQL을 이용해 데이터를 조작(상호작용)
 4. SQL 조작을 데이터에 적용한 후 이를 영구적으로 반영하거나(commit) 그러한 조작을 중단시켜(callback) 상호작용이 발생하기 전의 상태로 데이터를 되돌리도록 연결에 지시
 5. 데이터베이스에 대한 연결을 닫음(close)

3. SQLite

- SQLite(<https://www.sqlite.org/>)는 오픈소스로서 모든 기능을 갖춘 독립형 (외부 라이브러리를 거의 필요로 하지 않음), 서버리스(서버가 데이터베이스 엔진을 실행하지 않아도 되고 로컬에 저장된 데이터베이스), 제로 설정(설치하거나 설정할 것이 아무것도 없음), SQL 기반 경량 데이터베이스 관리 시스템(SQL 쿼리가 SQLite 테이블을 대상으로 실행 될 수 있음)으로서 하나의 데이터 파일을 사용해 데이터를 저장한다.
- SQLite는 작은 파일 하나에 모든 것을 다 포함하도록 C 언어로 구현하였다. 또한, 여러 프로세스에서 동시에 데이터베이스에 접근할 수 있도록 lock도 지원한다.
- SQLite는 google, apple, Microsoft 등의 대기업에서 사용되므로 매우 안정적이다.
- 파이썬의 SQLite3 모듈은 Python Database API를 지원한다.

3. SQLite

➤ 파이썬에서 SQLite를 이용한 작업은 5가지 주요 단계로 이루어진다.

1. 선택한 데이터베이스에 대한 연결 설정

```
>>> import sqlite3
```

```
>>> conn = sqlite3.connect( 'company.db' )
```

```
>>> conn
```

❖ 데이터베이스에 연결하는 객체 생성은 `connect()` 사용하고, 주어진 이름의 데이터베이스가 이미 있으면 기존의 데이터베이스에 연결한다. 하드디스크의 파일이 아닌 메모리에 임시 데이터베이스를 만들려면 `':memory'`라는 이름을 사용한다.

2. 데이터 전달을 위한 커서를 생성

```
>>> curs = conn.cursor()
```

```
>>> curs
```

```
>>> sqlite3.sqlite_version_info
```

```
>>> curs.execute( 'SELECT sqlite_version()' ).fetchone()[0]
```

❖ `sqlite3` 모듈의 버전 정보 확인

3. SQLite

3. SQL을 이용해 데이터를 조작

- ❖ 데이터베이스와 연결하고 커서를 생성하고 나면 이제 데이터를 작업(상호작용)할 준비가 끝났다. 즉, 이제 company.db 데이터 베이스에 대해 SQL 명령을 실행할 수 있다.

```
>>> curs.execute( 'drop table if exists employee' )           # 테이블삭제
>>> curs.execute( 'create table employee( name, age )' )       # 테이블생성
>>> curs.execute( "insert into employee value( 'hong', 28 )" ) # 데이터삽입
>>> values = [ ( 'kim', 54 ), ( 'park', 34 ), ( 'yun', 28 ) ]
>>> curs.execute( 'insert into employee values( ?,? )', values )
```

4. 변경사항 commit

```
>>> conn.commit()
```

- ❖ 변경 내용을 적용(commit)하는 동작

5. 데이터베이스에 대한 연결 닫기

```
>>> conn.close()
```

3. SQLite

```
# dbtest.py
import sqlite3
conn = sqlite3.connect('company.db')
curs = conn.cursor()
curs.execute('create table employee (name, age)')
curs.execute("insert into employee values ('hong', 28)")
values = [('kim',54), ('park', 34), ('nam', 28), ('yun', 44)]
curs.executemany('insert into employee values(?,?)', values)
conn.commit()
conn.close()
```

- ❖ 스크립트를 실행하면 현재 디렉터리에 company.db라는 파일이 만들어진다. 이 파일을 다운로드하면 별도의 DB 분석툴을 이용하여 DB 내용을 확인할 수 있다.
- ❖ SQLite 데이터베이스 분석용 도구로 DB Browser for SQLite를 사용할 수 있다. (<http://sqlitebrowser.org/>)

3. SQLite

```
#sqlitetest.py
import sqlite3

conn = sqlite3.connect("pets.db")    # 테이블 생성/오픈
cursor = conn.cursor()

# 버전 확인
print(sqlite3.sqlite_version_info)
version = cursor.execute('SELECT sqlite_version()').fetchone()[0]
print(version)

# 테이블 삭제
cursor.execute("""DROP TABLE IF EXISTS pets""")
conn.commit()
```

3. SQLite

테이블 생성

```
cursor.execute('''CREATE TABLE pets (  
                name VARCHAR(20),  
                owner VARCHAR(20),  
                species VARCHAR(20),  
                sex CHAR(1),  
                birth DATE,  
                death DATE)''')
```

데이터 추가

```
cursor.execute("INSERT INTO pets VALUES('Fluffy', 'Harold', 'cat', 'f', '1993-  
02-04', NULL)")
```

3. *SQLite*

```
cursor.executescript("""
INSERT INTO pets VALUES('Claws', 'Gwen', 'cat', 'm', '1994-03-17', NULL);
INSERT INTO pets VALUES('Buffy', 'Harold', 'dog', 'f', '1989-05-13', NULL);
INSERT INTO pets VALUES('Fang', 'Benny', 'dog', 'm', '1990-08-27', NULL);
INSERT INTO pets VALUES('Bowser', 'Diane', 'dog', 'm', '1998-08-31', '1995-07-
29');
INSERT INTO pets VALUES('Chirpy', 'Gwen', 'bird', 'f', '1998-09-11', NULL);
INSERT INTO pets VALUES('Whistler', 'Gwen', 'bird', 'f', '1997-12-09', NULL);
INSERT INTO pets VALUES('Slim', 'Benny', 'snake', 'm', '1996-04-29', NULL);
""")

conn.commit()
```

3. SQLite

데이터 검색

```
cursor.execute('select * from pets')
```

```
print(cursor.fetchone())
```

```
print(cursor.fetchmany(3))
```

```
print(cursor.fetchall())
```

데이터 검색

```
cursor.execute('select * from pets')
```

```
for row in cursor:
```

```
    print(row)
```

```
for row in cursor.execute('select * from pets'):
```

```
    print(row)
```

테이블 삭제

```
#cursor.execute('DROP TABLE pets')
```

```
#cursor.close()
```


11. Python 네트워크 프로그래밍

Python 네트워크 프로그래밍에 대한 이해

1. 소켓(Socket) 이해
2. TCP 소켓 프로그래밍
3. UDP 소켓 프로그래밍

1. 소켓(Socket) 이해

- 소켓(Socket)은 TCP/IP 프로토콜의 프로그래머 인터페이스이다.
- 1982년 BSD UNIX 4.1에서 처음 소개되었으며, 현재 1986년 BSD UNIX 4.3에서 개정한 소켓이 널리 사용되고 있다.
- 소켓은 존재하는 프로세스 간의 대화가 가능하도록 하는 프로세스 간의 통신 방식이다. 이들 프로세스는 동일한 컴퓨터 내에 있거나 다른 컴퓨터에 있어도 된다.
- 소켓이 유용한 이유는 네트워크를 통한 통신 능력 때문이다.
- 소켓을 경유한 프로세스 간의 통신은 클라이언트/서버 모델에 기초한다.
- 서버 프로세스는 해당 컴퓨터에서 유일하게 할당된 번호의 소켓을 만든다. 연결에 성공하면 서버와 클라이언트 모두에게 각각 하나의 소켓 기술자가 반환되는데 이것으로 읽기와 쓰기를 한다.
- 소켓은 양방향 통신을 지원한다.

1. 소켓(Socket) 이해

➤ 포트 번호

- 물리적인 전송선은 하나지만 이것을 여러 응용 프로그램이 나누어 쓰기 위해서 포트라는 것을 사용한다.
- 한 컴퓨터 내의(소켓을 사용하는) 모든 서버 프로세스는 별도의 포트 번호가 부여된 소켓을 가지고 있다. 이것은 TCP/IP가 지원하는 상위 계층의 응용 프로그램을 구분하기 위한 번호이다.
- 포트 번호로 16bit를 사용하며 범위는 0 ~ 65535이다. 일반적으로 잘 알려진 인터넷 서비스는 포트 번호 1 ~ 255를 사용하고, 그 밖의 서비스는 포트번호 256 ~ 1023에 예약되어 있다. 포트 번호 1024 ~ 4999는 임시로 시스템에서 활용 가능한 포트이므로 사용자는 5000 ~ 65535 사이의 포크번호를 사용하면 된다.
- 파이썬으로 서비스의 포트 번호를 확인하려면 다음 함수를 사용한다.

```
socket.getservbyname( servicename, protocolname )
```

```
>>> import socket
```

```
>>> socket.getservbyname( 'http', 'tcp' ); socket.getservbyname( 'ftp', 'tcp' )
```

1. 소켓(Socket) 이해

➤ 프로토콜과 표준 포트 번호

프로토콜	명령어	포트번호
Echo	echo	7
Daytime	daytime	13
File Transfer	ftp	21/20
Telnet Terminal	telnet	23
Simple Mail Transfer	smtp	25
Trivial File Transfer	tftp	69
Finger	finger	79
Domain Name Service	domain	53
HyperText Transfer	http	80/84/8000
NetNews	nnntp	119

1. 소켓(Socket) 이해

➤ 소켓의 종류

- 소켓은 도메인(Domain)과 유형(Type)으로 분류할 수 있다.
- 도메인은 서버와 클라이언트의 소켓이 있는 장소를 가리킨다. AF_INET이 가장 일반적으로 사용된다. AF는 Address Family의 약자이다.

AF_INET	IPv4 소켓, 주소 표현을 위해 (host, port) 튜플이 사용된다.
AF_INET6	IPv6 소켓, 주소 표현을 위해 (host, port, flowinfo, scopeid) 튜플이 사용된다.
AF_UNIX	클라이언트와 서버는 동일한 기계에 있다. 유닉스 도메인 소켓이다.
AF_TIPC	리눅스 전용으로 Transparent IPC 프로토콜이다.
AF_NETLINK	Netlink 프로세스간 통신이다.
AF_BLUETOOTH	블루투스 프로토콜이다.
AF_PACKET	링크 수준 패킷이다.

1. 소켓(Socket) 이해

- 소켓 유형은 서버와 클라이언트 사이에 있을 수 있는 통신 유형이다.
SOCK_STREAM과 SOCK_DGRAM이 가장 일반적으로 사용된다.

SOCK_STREAM TCP 통신 소켓 유형이다. 신뢰성 있는 스트림 방식으로 소켓을 만든다. TCP 일련번호가 붙으며, 양방향 연결에 기초한 바이트 가변 길이의 스트림이다.

SOCK_DGRAM UDP 통신 소켓 유형이다. 데이터그램 방식의 소켓을 만든다. 비연결, 비신뢰성인 고정 길이의 메시지를 사용한다.

SOCK_RAW 무가공 소켓이다.

SOCK_RDM 신뢰성 있는 데이터그램이다.

SOCK_SEQPACKET 순서를 갖는 연결 모드로 레코드 전송에 사용된다.

2. TCP 소켓 프로그램

➤ TCP 절차

1. 서버

- ① TCP 소켓 객체를 생성한다.
- ② 소켓을 Host 컴퓨터의 Port에 연결한다.
- ③ 한 번에 처리할 수 있는 연결 수를 결정한다. 5까지 설정이 가능하다.
- ④ 클라이언트로부터 소켓 연결이 올 때까지 대기한다.
- ⑤ 클라이언트의 connect 함수로부터 소켓이 send()/recv()를 이용하여 데이터를 주고받는다. 데이터는 바이트 열로 표현한다.
- ⑥ 작업을 마쳤으면 클라이언트와의 연결을 종료한다.
- ⑦ 소켓 서비스를 끝낸다.

2. TCP 소켓 프로그램

```
① >>> from socket import *  
② >>> svrsock = socket( AF_INET, SOCK_STREAM )  
③ >>> svrsock.bind( ( 'localhost', 8000 ) )  
④ >>> svrsock.listen( 1 )  
⑤ >>> conn, addr = svrsock.accept()  
    >>> addr  
⑥ >>> conn.recv( 1024 )  
⑦ >>> conn.close()
```

2. TCP 소켓 프로그램

2. 클라이언트

- ① TCP 소켓 객체를 생성한다.
- ② 소켓을 서버 컴퓨터의 특정 코트에 연결한다.
- ③ 연결되었으면 데이터를 수신하고 전송하다. 데이터는 바이트 열로 해야 한다.
- ④ 작업을 마쳤으면 소켓을 종료한다.

```
>>> from socket import *
```

- ① >>> clientsock = socket(AF_INET, SOCK_STREAM)
- ② >>> clientsock.connect(('localhost', 8000))
- ③ >>> clientsock.send('Hello'.encode())
- ④ >>> clientsock.close()

2. TCP 소켓 프로그램

- ❖ 서버에서 ⑤번까지 수행하면 하위 프로세스는 블록(Block) 상태(멈춰 있는 상태)로 들어간다. 클라이언트에서 ① ~ ③ 까지 입력한다. 같은 서버가 아니라면 ④에서 connect() 메서드의 첫 인수에 서버 주소를 입력한다. 서버는 블록 상태가 풀리고 >>>가 나타난다. 서버에서 클라이언트의 주소를 확인할 수 있다.(>>> addr). 클라이언트에서 'Hello'를 보내는데 바이트열로 보내야 한다. 서버에서 메시지를 받는다. 클라이언트가 종료한다. 서버도 종료한다.
- ❖ 이와 같은 구조로 TCP 기반 클라이언트/서버 프로그램을 작성 할 수 있으나 여러 클라이언트가 동시 접속 요청에 대한 처리 능력은 없다.

2. TCP 소켓 프로그램

- `asyncore` 모듈을 이용한 TCP 클라이언트/서버 프로그램
 - `asyncore` 모듈은 비동기 소켓 핸들러(Asynchronous Socket Handler)
 - 내부에서 운영 체제의 `select()` 시스템 호출을 사용한다. `select()` 시스템 호출을 한 개 이상의 소켓들이 입출력 연산을 수행할 수 있는 상태가 되는 지를 감시하다가 연산을 수행할 수 있는 상태가 되면 즉시 그 정보를 알려준다.
 - `select()`를 이용하면 스레드나 프로세스를 사용하지 않고도 비동기적으로 여러 입력과 출력 스트림에 걸쳐서 다중 처리를 구현할 수 있다. 따라서 여러 클라이언트로부터의 동시 접속 요청을 수용할 수 있다.
 - `asyncore` 모듈은 소켓을 만들고 일일이 작업을 지시하지 않아도 상황에 따라 실행되어야 하는 메서드만 설정해 놓으면 알아서 호출되어 실행되도록 할 수 있더 간편하게 클라이언트/서버 기능을 구현할 수 있다.
 - 비동기 소켓 핸들러는 `dispatcher` 클래스를 상속받아서 구현한다.

2. TCP 소켓 프로그램

➤ 비동기 소켓 핸들러가 필요에 따라 구현해야하는 메서드

메서드	설명
writable()	데이터를 쓸 수 있으면 True, 아니면 False, 기본 메서드는 True
readable()	데이터를 읽을 수 있는 상태이면 True, 기본 메서드는 True
handle_connect()	클라이언트에서 연결 시도가 성공하면 호출
handle_accepted(sock, addr)	서버에서 새로운 연결이 설정되면 호출, sock은 data 송수신용, addr은 클라이언트의 주소
handle_expt()	소켓에 사용하는 것 외의 연결 상에서 데이터가 발생했을 때 호출, 거의 발생안함
handle_read()	새로운 데이터가 소켓에 있을 때 호출, readable()가 True 반환 상태에서 실행
handle_write()	쓰기 소켓에 쓰는 것이 가능해지면 호출, writable()가 True 반환 상태에서 실행
handle_close()	소켓이 닫힐 때 호출
handler_error()	처리되지 않은 예외가 발생하면 호출된다.
loop([timeout[, use_poll[, map[, count]]]])	이벤트를 polling한다. 모든 인수는 옵션이다.

❖ asyncore 모듈 이용 클라이언트/서버 적용 예제는 별도 제공

3. UDP 소켓 프로그램

- UDP 소켓 프로그래밍을 TCP 소켓 프로그래밍과 절차에서 약간의 차이가 있다. 소켓 객체는 인수 AF_INET과 SOCK_DGRAM로 생성한다.
- UDP는 비연결형 프로토콜이므로 서버와 클라이언트에서 연결 요청 (connect)과 연결 함수(accept)를 수행할 필요가 없다. 서버는 사용하는 포트 번호로만 소켓을 묶으면(bind) 되고, 클라이언트는 자신이 사용하려는 포트 번호를 소켓에 묶으면 된다. 서버와 클라이언트의 포트 번호가 같을 필요는 없다.
- 데이터 송수신은 sendto()와 recvfrom() 을 이용한다.
- sendto()에서 소켓은 연결된 상태가 아니어야 하고 인수 address형식은 (host, port) 튜플이다.
`sendto(bytes [, flags], address)`
- bind 소켓으로 들어오는 데이터를 읽어내는 recvfrom()는 다음과 같다.
`recvfrom(bufsize [, flags])`

2. UDP 소켓 프로그램

➤ UDP 절차

1. 서버

- ① `>>> from socket import *`
- ② `>>> svrsock = socket(AF_INET, SOCK_DGRAM)`
- ③ `>>> svrsock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)`
- ④ `>>> svrsock.bind(('localhost', 5001))`
데이터와 발신 주소를 함께 얻는다. 1024는 버퍼 크기이다.
- ⑤ `>>> s, addr = svrsock.recvfrom(1024)`
`>>> s`
`>>> addr`
메시지와 수신 주소와 포트를 입력한다.
- ⑥ `>>> svrsock.sendto('Hello'.encode(), addr)`

2. UDP 소켓 프로그램

2. 클라이언트

```
① >>> from socket import *  
② >>> csock = socket( AF_INET, SOCK_DGRAM )  
③ >>> csock.sendto( 'HI'.encode(), ( 'localhost', 5001 ) )  
④ >>> s, addr = svrsock.recvfrom( 1024 )  
  
>>> s  
  
>>> addr
```


12. Python GUI 프로그래밍

Python GUI 프로그래밍에 대한 이해

1. Python GUI 및 Tkinter 개요
2. 프로그램 구조
3. TKinter 위젯

1. Python GUI 및 Tkinter 개요

- 파이썬에서는 GUI 프로그램을 만들기 위해서는 여러 GUI Framework(또는 Toolkit)들을 사용할 수 있다.
- 파이썬의 GUI Framework/Toolkit에는 기본적으로 제공되는 표준 GUI 라이브러리인 Tkinter, Qt Framework를 파이썬에서 사용하도록 한 PyQt, GTK Toolkit을 파이썬에서 사용하도록 한 PyGTK등 다양한 툴들이 있다.
- Tkinter는 Tcl/Tk에 대한 파이썬 Wrapper로서 Tcl/Tk를 파이썬에서 사용할 수 있도록 한 Lightweight GUI 모듈이다.
- Tcl은 Tool Command Language의 약자로서 일종의 프로그래밍 언어이며, Tk는 크로스 플랫폼에 사용되는 일종의 GUI Toolkit이다.
- Tkinter는 타 GUI Framework나 Toolkit에 비해 지원되는 위젯들이 부족하고 UI도 그렇게 예쁘지 않다는 단점이 있지만, 파이썬 설치 시 기본적으로 내장되어 있는 파이썬 표준 라이브러리이기 때문에 쉽고 간단한 GUI 프로그램을 만들 때 활용될 수 있다.

2. 프로그램 구조

➤ Tkinter를 이용한 GUI 프로그램은 다음과 같은 방식으로 작성한다.

1. 메인 윈도우(TK 객체) 생성
2. 위젯 생성
3. 위젯을 윈도우에 배치
4. 메인 루프 실행

- Tkinter 라이브러리를 사용하려면 우선 메인 윈도우인 `Tk` 객체의 인스턴스를 생성해야 한다.
- 위젯은 컨트롤이라고도 하며 윈도우 내용을 구성하는 용도로 사용되는 객체로서 위젯을 생성한 후 윈도우에 배치한다.
- 메인 루프를 실행하면 윈도우가 보여진다. 메인 루프는 이벤트 메시지 루프로서 발생한 이벤트로부터 메시지를 수신하여 해당 이벤트를 수신한 위젯에 전달하여 메시지 핸들러가 동작되게 하는 역할을 담당한다.

2. 프로그램 구조

- Tkinter 라이브러리를 사용하면 미리 준비된 위젯을 윈도우에 배치하는 것으로도 간단한 GUI 프로그램을 완성할 수 있다.
- 위젯의 초기값을 변경하지 않고 그대로 사용해도 동작하므로 소스 코드의 길이가 짧는데 이것이 Tkinter 라이브러리의 장점이다.

```
# firstGUI.py
```

```
from tkinter import *
```

```
root = Tk() # 메인 윈도우 생성
```

```
label = Label( root, text = 'Python is great!!!' ) # 위젯 생성
```

```
label.pack() # 위젯 배치
```

```
root.mainloop() # 메인 루프 실행
```

3. Tkinter 위젯

➤ 위젯은 윈도우를 구성하는데 사용되는 객체로서 개개의 위젯이 제공하는 옵션을 조정하여 윈도우에 표시되거나 동작을 제어 할 수 있다.

❖ 위젯 공통 옵션

옵션 분류	옵션명	설정값	설명
색	activebackground	색 이름 또는 #RGB 형식의 16진수 지정하지 않으면 투명, 색 이름에는 red, blue, green, white, black, magenta, cyan, yellow등이 있음, 예를 들어 빨강을 지정하려면 red 또는 #ff0000	활성 위젯 배경색
	activeforeground		활성 위젯 전경색
	background(bg)		위젯의 배경색
	disabledforeground		비활성 위젯의 전경색
	foreground(fg)		위젯의 전경색
	highlightbackground		포커스된 위젯의 하이라이트 영역 배경색
	highlightcolor		포커스된 위젯의 하이라이트 영역 전경색
	selectbackground		선택 항목의 배경색
	selectforeground		선택 항목의 전경색

3. Tkinter 위젯

❖ 위젯 공통 옵션

옵션 분류	옵션명	설정값	설명
길이	borderwidth(bd)	숫자	경계선 너비
	height		위젯 높이
	highlightthickness		포커스된 위젯의 하이라이트 영역 너비
	padx		x 방향 추가 여백
	pady		y 방향 추가 여백
	selectborderwidth		선택 항목의 주변 경계선 너비
	wraplength		줄 바꿈할 길이의 최대폭
	width		위젯 너비
문자	underline	숫자(단위는 문자 개수)	글자 아래에 선을 그을 때 첫 번째 글자를 0이라 하고 몇 번째 문자까지 밑줄을 그을지 지정함

3. Tkinter 위젯

❖ 위젯 공통 옵션

옵션 분류	옵션명	설정값	설명
기타	anchor	n, ne, e, se, s, sw, w, nw, center중 하나	위젯 안 표시위치를 방위각을 표시
	command	함수명 또는 메서드명	이벤트 발생시 호출할 함수
	font	(폰트명, 크기, 옵션)의 튜플로 지정	위젯의 텍스트 폰트
	image	PhotoImage 객체/BitmapImage 객체	위젯에 출력된 이미지
	justify	left, center, right	텍스트를 여러 줄로 출력할 때 정렬 방법
	relief	raised, sunken, flat, ridge, solid, groove	위젯을 3D로 표시하는 방법
	state	normal, active, disabled	위젯 상태
	takefocus	True/False	True로 지정하면 Tab키 또는 Shift + Tab키로 포커스 이동
	text	문자열	출력할 텍스트
	textvariable	Variable 객체	위젯에 지정할 Variable 객체

3. Tkinter 위젯

➤ Label 위젯 : 텍스트를 출력 할 때 사용하는 위젯

```
label = Label( parent, 옵션... )
```

❖ Label 위젯 옵션

옵션명	설정값	설명
background	색 이름 또는 #RGB 형식의 16진수	레이블의 배경색
cursor	커서명	레이블에 마우스 커서를 올렸을 때 마우스 모양
font	(폰트명, 크기, 옵션)의 튜플을 지정	레이블 텍스트 폰트
height	정수	레이블 높이(줄 수를 지정)
text	문자열	출력할 텍스트
textvariable	Variable 객체	레이블에 지정할 Variable 객체
width	정수	위젯 너비(문자 개수를 지정)

3. Tkinter 위젯

➤ pack()은 지오메트리 매니저라고 부르며 생성한 위젯을 윈도우에 배치하는 기능이 있다.

❖ pack 메서드 옵션

옵션명	설정값	설명
anchor	n, ne, e, se, s, sw, w, nw, center	위젯을 어느 영역에 배치할지 화면 위쪽을 북쪽으로 본 방향값으로 지정
expand	True/False	True를 설정하면 위젯을 배치한 영역 크기가 변경되면 위젯 크기도 변함, 기본값은 False
fill	none, x, y, both	위젯을 배치한 영역에 맞게 위젯을 확대할지 여부, 기본값은 none
ipadx	숫자	수평 방향으로 내부 여백, 기본값은 0
ipady	숫자	수직 방향으로 내부 여백, 기본값은 0
padx	숫자	수평 방향으로 외부 여백, 기본값은 0
pady	숫자	수직 방향으로 외부 여백, 기본값은 0
side	top, bottom, left, right	위젯 배치 순서, 기본값은 top

3. Tkinter 위젯

➤ Button 위젯 : 사용자로 부터 선택을 받을때 사용하는 위젯

button = Button(parent, 옵션...)

❖ Button 위젯 옵션

옵션명	설정값	설명
bitmap	비트맵 이름	지정한 비트맵 파일을 표시
command	함수명 또는 메서드명	사용자가 버튼을 눌렀을 때 호출되는 함수
cursor	커서명	위젯 위에 마우스 커서를 올렸을 때 마우스 커서
default	normal, active, disabled	버튼의 기본 상태를 지정함. 기본값은 disabled
height	숫자	버튼 높이
image	PhotoImage 객체/BitmapImage 객체	위젯에 표시되는 이미지
text	문자열	버튼에 표시되는 텍스트
textvariable	Variable 객체	버튼에 연결된 Variable 객체
width	숫자	버튼 너비

3. Tkinter 위젯

```
# button1.py
from tkinter import *

root = Tk()

def btnEvent():                                # 이벤트 발생시 호출되는 callback 함수
    print( 'Button Pushed' )

button = Button( root, text = 'Push!', command = btnEvent )
button.pack()

root.mainloop()
```

3. Tkinter 위젯

```
# button2.py
from tkinter import *

root = Tk()

def btnEvent():
    label.config(text = 'Pushed')          # 레이블 변경에 config()

label = Label(root, text = 'Push Button')
label.pack()

button = Button( root, text = 'Push!', command = btnEvent )
button.pack()

root.mainloop()
```

3. Tkinter 위젯

❖ 위젯 공통 메서드

메서드	설명
cget(option)	위젯에 설정한 옵션값을 반환
config(option = value)	위젯의 option 값을 value로 변경
focus_set()	포커스를 둘 위젯을 설정
grab_set()/grab_release()	지정한 위젯에 모든 이벤트 보내기/grab_set() 해제
mainloop()	이벤트 루프
quit()	이벤트 루프 종료
update()	미처리 이벤트 처리
update_idletasks()	이벤트 루프가 아이들 상태일 때 미처리 이벤트 처리
wait_variable(v)	인수로 지정된 Variable 객체가 변하는 것을 대기
wait_visibility(w)	인수로 지정된 위젯이 가시 상태가 될 때까지 대기
wait_window(w)	인수로 지정한 위젯이 없어질 때까지 대기
wininfo_height()	지정한 위젯의 높이를 반환
wininfo_width()	지정한 위젯의 너비를 반환

3. Tkinter 위젯

```
# button3.py
from tkinter import *

root = Tk()

def btnEvent():
    label.config( text = 'Pushed' )
def event( ev ):
    label.config( text = 'Push Button' )

label = Label( root, text = 'Push Button' )
label.pack()
button = Button( root, text = 'Push!', command = btnEvent )
button.pack()
button.bind( '<Leave>', event)      # 마우스 커서가 버튼을 벗어났을때 이벤트 추가

root.mainloop()
```

3. Tkinter 위젯

❖ 이벤트 시퀀스

종류	이벤트명	설명
키보드 이벤트	특수키	ctrl/shift 같은 키를 눌렀을 때 발생하는 이벤트, 키 명칭을 <>로 감싸서 표시
	일반키	영문이나 숫자, 기호 키를 눌렀을 때 발생하는 이벤트, 작은 따옴표를 사용해서 지정
마우스 이벤트	Button-1, Button-2, Button-3	마우스 버튼을 눌렀을 때 발생하는 이벤트
	B1-Motion, B2-Motion, B3-Motion	마우스 버튼을 누르고 커서를 이동했을 때 발생하는 이벤트
	ButtonRelease-1, ButtonRelease-2, ButtonRelease-3	마우스 버튼을 뗐을 때 발생하는 이벤트
	Double-Button-1, Double-Button-2, Double-Button-3	마우스 버튼을 더블 클릭했을 때 발생하는 이벤트
	Enter	마우스 커서가 위젯 안에 들어갔을 때 발생하는 이벤트
	Leave	마우스 커서가 위젯 밖으로 나왔을 때 발생하는 이벤트

3. Tkinter 위젯

❖ 이벤트 관련 메서드

메서드	설명
<code>bind(event_name, callable[, '+'])</code>	특정 이벤트가 발생했을 때 지정 함수가 호출됨, '+'는 이벤트 추가
<code>bind_all(event_name, callable[, '+'])</code>	모든 위젯에 특정 이벤트가 발생 했을 때 지정한 함수가 호출되도록 지정
<code>unbind(event_name)</code>	지정한 이벤트에 관련된 콜백 함수를 제거
<code>unbind_all(event_name</code>	모든 위젯에서 지정한 이벤트에 관련된 콜백 함수를 제거
<code>after(ms, callable, *arg)</code>	타이머로 동작해서 지정한 시간뒤에 콜백 함수를 호출
<code>after_cancel(id)</code>	타이머 제거
<code>after_idle(callable, *arg)</code>	이벤트 루프가 아이들 상태일 때(모든 이벤트가 처리됐을 때)지정 함수가 호출됨

3. Tkinter 위젯

```
# entry.py
from tkinter import *

root = Tk()

def event( ev ):
    label.config( text = entry.get() )

label = Label( root, text = 'Input Text' )
label.pack()

entry = Entry( root )
entry.pack()

entry.bind( '<Return>', event )

root.mainloop()
```

3. Tkinter 위젯

```
# radio.py
from tkinter import *
root = Tk()
def func1():
    label.config( text = 'Button 1' )
def func2():
    label.config( text = 'Button 2' )
sel = IntVar()
sel.set(1)
label = Label( root, text = 'Select Button' )
label.pack()
rb1 = Radiobutton(root, text = 'Button 1', variable = sel, value = 1, command = func1)
rb1.pack()
rb2 = Radiobutton(root, text = 'Button 2', variable = sel, value = 2, command = func2)
rb2.pack()

root.mainloop()
```

참고

- 파이썬3 바이블 - 이강성(프리렉)
- 점프 투 파이썬 - 박응용(이지스퍼블리싱)
- 모두의 라즈베리파이 with 파이썬 - 서수환오피스(길벗)
- Wikipedia.org를 통한 검색