16. 메타 클래스

16.1 클래스 동적 생성과 type() 함수

메타클래스를 잘 이해하기 위해서 파이썬 클래스를 다시한번 짚어보자. 파이썬의 클래스는 그자체로 객체이다. 따라서 변수에 치환될 수도 있고, 복사도 가능하고, 속성값을 동적으로 추가할수도 있고, 함수의 매개 변수로 전달할수도 있다.

```
>>> class Klass:                        # 클래스를 정의한다.
...    pass
>>> Klass                               # Klass 객체값을 출력한다.
__main__.Klass
>>> def echo(o):
...     print(o)
>>> echo(Klass)                      # 함수로 전달할수 있다.
<class '__main__.Klass'>
>>> Klass.new_attr = 'sori'             # 속성을 추가한다.
>>> Klass.new_attr                      # 추가한 속성을 확인한다.
'sori'
>>> Mirror = Klass                      # 변수에 치환한다.
>>> Mirror.new_attr
'sori'
>>> Mirror()
<__main__.Klass at 0x19b4560da08>
```

클래스가 객체이기 때문에 다른 객체와 같이 코드 블록 안에서 즉시 생성하는 것이 가능하다.

```
>>> def make_class(name):
...     if name == 'sori':
...             class Sori: pass
...             return Sori     # 생성된 클래스를 반환한다.
...     else:
...             class Nori: pass
...             return Nori
...
>>> NewClass = make_class('sori')
>>> NewClass                    # 클래스 객체임을 확인한다.
__main__.make_class.<locals>.Sori
>>> NewClass()                  # 인스턴스 객체의 생성 가능을 확인한다.
<__main__.make_class.<locals>.Sori at 0x19b4561f708>
```

하지만, 이 방법은 동적이지 않다. 이제 동적으로 클래스를 생성하는 법을 살펴보자. type() 함수가 사용한다. 이함수는 원래 객체의 자료형을 확인하는 함수였다.

```
>>> type(1)
<class 'int' >
>>> type("123")
<class 'str' >
>>> k = Klass()
>>> type(k)
<class '_main_.Klass;>
>>> type(Klass)
<class 'type' >
```

하지만, type() 함수의 또 다른 목적은 새로운 클래스를 만드는 메타클래스이다. 클래스가 인스턴스 객체를 생성하듯이, type() 함수는 클래스 객체를 생성한다. 사용하는 방법은 다음과 같다.

```
type(name, bases, dict)
```

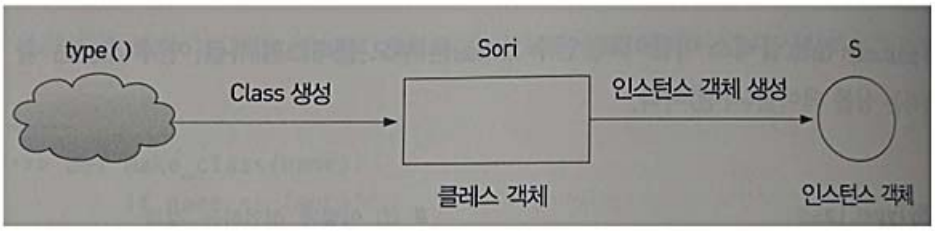
여기서 인수 name은 만들 클래스 이름이고, 인수 bases는 부모 클래스의 튜플, 인수 dict는 속성값을 정의하는 심볼 테이블(사전)이다.

```
>>> class MyTypeClass:                    # ① 이렇게 선언하는 것과
...     pass
...
>>> MyTypeClass = type('MyTypeClass', (), {}) # ② 이렇게 선언하는 것은 같다.
>>> MyTypeClass
__main__.MyTypeClass
>>> c = MyTypeClass()
>>> c
<__main__.MyTypeClass at 0x19b45629ac8>
```

이번에는 클래스 멤버를 지정하는 심볼 테이블을 넘겨주어 보자.

```
>>> class Sori:                           # ① 이렇게 하는 것과
...     a = 10
...     b = 20
...
>>> Sori = type('sori', ( ), {'a':10, 'b':20})  # ② 이렇게 하는 것은 같다.
>>> Sori
<class '_main_.Sori' >
>>> Sori.a
10
>>> s = Sori()
>>> s.b
20
```

```
>>> type(s)
<class '_main_.Sori' >
>>> type(Sori)
type
```




메타클래스는 클래스 객체를 생성하는 클래스이다. 따라서 type이 메타클래스이다. 여기서 생성된 클래스 객체 Sori 는 당연하겠지만 다음과 같이 상속도 할수 있다.

```
>>> class Sound(Sori):
...     pass
```

상속은 다음과 같이 type() 함수로도 가능하다.

```
>>> Sound = type('Sound', (Sori,), {})
>>> Sound
<class '_main_.Sound' >
>>> Sound.__bases__    # 베이스 클래스 확인
(<class '_main_.Sori' >,)
```

이번에는 메서드를 포함하는 클래스를 동적으로 만들어 보자.

```
>>> def display(self):
...     print(self.a)
...
>>> SoundChild = type('SoundChild', (Sound,), {'display':display})
>>> s = SoundChild()
>>> s.display()
10
>>> SoundChild.mro()    # mro 확인
[<class '__main__.SoundChild' >, <class '__main__.Sound' >, <class' __main__.Sori' >,<class
 'object' >]
```

예들을 통해서 type() 함수를 사용하여 멤버뿐 아니라 메서드까지도 동적으로 설정된 클래스를 만드는 것이 가능하다는 것을 알았다.

## 16.2 메타클래스 만들기

메타클래스 type을 상속한 클래스는 역시 메타클래스이다.

```
>>> class SubType(type):                    # 메타클래스
...     def __init__(cls, name, bases, dct):  # 첫 인수는 클래스 객체이다.
...         print('__init__', name)
...         super().__init__(name, bases, dct)
...
>>> S1 = SubType('SubMetaClass', ( ), {})       # 클래스 만들기
__init__ SubMetaClass
>>> S1
<class '_main_.SubMetaClass' >

#  메서드가 있는 클래스 만들기
>>> S2 = SubType('SubMetaClass2', (), {'foo':lambda self:'bar'})
_init_ SubMetaClass2
>>> si = S2()
>>> si.foo()
'bar'
```

메타클래스는 클래스 객체를 생성하므로 _init_() 메서드에 전달되는 첫인수 cls는 생성된 클래스 객체이다. S1에는 아무 속성값을 정의하지 않았지만 S2에는 메서들 foo를 정의했다.

메타클래스의 _new_() 메서드는 클래스 객체를 생성하는 일을 하며 다음과 같은 기본 기능을 수행한다.

```
>>> class MetaClass(type):
...     def __new__(meta_cls, cls_name, based, dct):
...         return type.__new__(meta_cls, cls_name, bases, dct)
```

__new__() 메서드를 사용하면 메서드를 자동으로 추가하는 것이 가능하다.

```
>>> class SubType2(type):
...      def __new__(meta_cls, cls_name, bases, dct):
...              dct['foo'] = lambda self: 'bar'  # 메서드를 추가한다.
...              return type.__new__(meta_cls, cls_name, bases, dct)
...
>>> S3 = SubType2('SubMetaClass3', (), {})
>>> s = S3()                              # 메서드를호출한다
>>> s.foo()
'bar'
```

__new__() 메서드로 전달되는 cls와 _init_()  메서드로 전달되는 첫 인수는 다른 객체이다.

```
>>> class SubType3(type):
...     def __new__(meta_cls, cls_name, bases, dct):
...             print('__new__',meta_cls.__name__, cls_name)
...             return type.__new__(meta_cls, cls_name, bases, dct)
...     def __init__(cls, name, bases, dct):
...             print('__init__',cls.__name__, name)
...             type.__init__(cls, name, bases, dct)
...
>>> S3 = SubType3('SubMetaClass3', (), {})
__new__ SubType3 SubMetaClass3
__init__ SubMetaClass3 SubMetaClass3
```

__new__() 메서드의 인수 meta_cls는 메타클래스 SubType3이고 _init_()  메서드의 인수 cls 는 생성된 클래스 객체인 SubMetaClass3이다. 이제 __new__() 와 _init_() 메서드의 구분이 명확해진 것 같다.

메타 클래스에 정의된 모든 클래스는 첫 인수로 생성된 클래스 객체를 받는다.  메타클래스의 메서드는 생성된 객체(클래스)를 인수로 받기 때문이다.  따라서 첫인수의 이름을 self가 아닌 cls로 했다.

```
>>> class MyName(type):                # 메타클래스
...     def whoami(cls):               # 첫 인수는 생성된 클래스 객체이다.
...             print("I am", cls.__name__)
...
>>> Foo = MyName('foo', (), {}) # 클래스 생성
>>> Foo.whoami
<bound method MyName.whoami of <class '__main__.foo'>>
>>> Foo.whoami()                       # 바운드 메서드를 호출한다.
I am foo
>>> MyName.whoami(Foo)                 # 언바운드 메서드를 호출한다.
I am foo
```

## 16.3 메타클래스 선택하기

type이 아닌 메타클래스 MyName을 클래스 생성에 사용하는 방법은 기본적으로 다음과 같다.

```
Foo = MyName( 'Foo' , (), {})
```

메타클래스 MyName을 이용하여 클래스 Foo를 만들고 있다. 하지만, 다음과 같은 방법으로 metaclass 키워드 인수를 이용하여 메타클래스를 지정할수 있다.

```
>>> class Foo(metaclass = MyName):              # 키워드 metaclass 메타클래스를 지정한다.
...     pass
...
>>> Foo.whoami()
I am Foo
```

두가지 방법은 사실은 동일하다. 다음 코드를 보자.

```
>>> class PrintType(type):              # 메타클래스
...     def __new__(cls, name, bases, namespace):
...             print(name, bases, namespace)
...             return type.__new__(cls, name, bases, namespace)
...
>>> class FOO(object, metaclass = PrintType):
...    bar = 10
...
FOO (<class 'object'>,) {'__module__': '__main__', '__qualname__': 'FOO', 'bar': 10}
```

앞에서 class 문에 의해서 클래스 Foo를 만드는 것은 다음과 같이 PrintType을 호출한 것과 사실은 동일하다.

```
>>> Foo = PrintType('Foo', (object,),{'bar':10, '_module_':'_main_'})
Foo (<class  'object' >,){ '_module_' :' _main_ ',' bar' :10}
```

## 16.4 메타클래스의 __call__() 메서드

메타클래스의 또 다른 유용한 메서드는 _call_()이다. 메타클래스의 _new_()나 _init_() 메서드는 클래스 인스턴스를 생성할 때 호출되지만, _call_() 메서드는 생성된 클래스 인스턴스를 이용하여 객체를 생성할 때 호출된다.

```
>>> class CallType(type):
...     def __call__(cls, *args, **kwargs):
...             print("__call__ called", cls.__name__)
...             obj = type.__call__(cls, *args)  # 인스턴스 객체를 생성한다.
...             return obj
...
>>> class CallClass(metaclass = CallType):
...     pass
...
>>> c = CallClass()
__call__ called CallClass
>>> c
<__main__.CallClass at 0x19b45636c48>
```

클래스 인스턴스 객체를 생성할 때 호출되는 메서드로는 _new_()와 _init_() 등이 있는데 이들의 순서를 확인해 보자.

```
>>> class CallType(type):
...     def __call__(cls, *args, **kwargs):
...             print("__call__ called 1",cls)
...             obj = type.__call__(cls, *args) # 인스턴스 객체를 생성한다.
...             print(obj)
...             return obj
...
>>> class CallClass(metaclass = CallType):
...     def __new__(cls, *args, **kw):
...             print('_new_ called')
...             return super().__new__(cls, *args, **kw)
...     def __init__(self):
...             print('__init__ called 2')
...
>>> c = CallClass()
__call__ called 1 <class '__main__.CallClass'>
_new_ called
__init__ called 2
<__main__.CallClass object at 0x0000019B45640DC8>
>>> c
```

<__main__.CallClass at 0x19b45640dc8>

확인한 것처럼 클래스 CallClass()에 의해서 수행되는 순서는 클래스 CallType의 _call_() 메서드, 클래스 CallClass의 _new_()와 _init_() 메서드 순서이다. 사실 클래스 CallType의 _call_() 메서드에서 호출되는 type._call_(cls, *args)에 의해서 클래스 CallClass의 _new_()와 _init_() 메서드를 호출해 주기 때문에 당연할 결과이다. 메타클래스의 _call_() 메서드는 인스턴스 객체를 반환해야 한다. 앞서의 예에서 _call_() 메서드에서 반환하는 객체는 c=CallClass()의 c와 동일한 객체이다.

16.5 메타클래스의 예

- 추상 클래스

추상클래스란 클래스 내에서 한 개 이상의 구현이 되어 있지 않은 추상 메서드가 있어서 인스턴스 객체를 생성할 수 없는 클래스이다. 파이썬에서는 하위 클래스가 반드시 구현해야 할 메서드를 가지고 있는 클래스라고 이해할수 있다. 추상 클래스는 abc 모듈의 ABCMeta 클래스를 메타클래스로 지정하면 되고 추상 메서드는 @abstractmethod로 장식한다.

```
>>> from abc import *
>>> class C(metaclass = ABCMeta):       # C는 추상클래스이다.
...     @abstractmethod
...     def absMethod(self):
...             pass
...
>>> c = C()             # 인스턴스 객체를 생성할 수 없다.
TypeError                                 Traceback (most recent call last)
<ipython-input-117-449f2ccb3492> in <module>
      5             pass
      6
----> 7 c = C()         # 인스턴스 객체를 생성할 수 없다.

TypeError: Can't instantiate abstract class C with abstract methods absMethod
>>> class D(C):         # 자식 클래스에서 메서드를 구현한다.
...     def absMethod(self):
...             print('absMethod implemented')
...
>>> d = D()             # 이제 인스턴스 객체를 생성하는 것이 가능하다.
>>> d.absMethod()
absMethod implemented
```

- 자동으로 멤버를 설정하는 메타클래스

일반적으로 생성자에 선언된 변수는 멤버 변수로 그대로 사용하는 경우가 많다. 이 경우 생성자에서 일일이 멤버 변수들을 생성한다.

```
class Panel:
        def _init_(self, width, height):
                self.width = widty      # 일일이 반복해야 한다.
                self.height = height    # ...
```

하지만, 멤버 변수 생성이 자동으로 된다면 편리할 것이다. 다음은 인수로 사용된 변수들을 자동으로 멤버 변수로 설정되게 하는 메타클래스의 예이다.

```
# automember.py

class AutoMemberSetType(type):
    def __call__(cls, *args, **kwargs):
        obj = type.__call__(cls, *args, **kwargs)
        arg_names = obj.__init__.__func__.__code__.co_varnames[1:]
        defaults = obj.__init__.__func__.__defaults__
        for name, value in zip(arg_names, args+defaults):
            setattr(obj, name, value)
        for name, value in kwargs.items():
            setattr(obj, name, value)
        return obj

class Panel(metaclass=AutoMemberSetType):
    def __init__(self, width, height=400):
        pass                    # 별로 할 일이 없다.
```

코드를 실행한 예는 다음과 같다.

```
>>> p = Panel(60, 40)
>>> print (p.width, p.height) # 자동으로 멤버들이 설정되어 있다.
60 40
>>> p = Panel(60, 40)
>>> p.width, p.height          # 자동으로 멤버들이 설정되어 있다.
(60, 40)
>>> p = Panel(600)
>>> p.width, p.height
(600, 400)
>>> p = Panel(width = 600, height = 300)
>>> p.width, p.height
(600, 300)
```

- 싱글톤

인스턴스 객체를 오직 하나만 생성해 내는 클래스인 싱글톤을 구현하기 위하여 메타클래스를 사용한 예를 보자. 다음과 같이 싱글톤 메타클래스를 정의한다.

```
# singleton_meta.py

class Singleton(type):
    __instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls.__instances:
```

```
            cls.__instances[cls] = super().__call__(*args, **kwargs)
        return cls.__instances[cls]
```

코드를 실행한 결과는 다음과 같다.

```
>>> class MyClass(metaclass=Singleton):
...     pass

>>> m1 = MyClass()
>>> m2 = MyClass()
>>> print(m1 is m2)             # True이다. 동일한 객체가 맞다.
true
```

- final 메타클래스

더 이상의 클래스 상속이 가능하지 않도록 하는 final 메타클래스의 예이다. 이 메타클래스로부터 만들어진 클래스는 더 이상 상속이 가능하지 않다.

```
# final_meta.py

class final(type):
    def __init__(cls, name, bases, namespace):
        super().__init__(name, bases, namespace)
        for klass in bases:     # 기반 클래스에 final이 있으면 에러를 발생시킨다
            if isinstance(klass, final):
                raise TypeError(klass.__name__ + ' is final')
```

사용하는 예를 보자. B클래스가 final을 메타클로스로 설정했다. 더 이상 B클래스에서 상속받을수는 없다.

```
>>> class A: pass
>>> class B(A, metaclass=final): pass
>>> class C(B): pass
...
TypeError: B is final
```

- 디버깅

메타클래스를 이요하면 디버깅 정보를 메서드마다 일일이 추가하지 않고도 자동으로 디버깅 정보를 모든 메서드에 일괄적으로 적용할수 있다. 디버깅 정보는 환경 변수 'DEBUG' 의 값이 'TRUE' 이면 디버깅 모드로 동작을 하고 그렇지 않으면 정상적인 수행을 하게 된다.

```
# rectangle.py
```

```
import os

def debugged(func, cls_name):   # 장식자
    # 디버깅 모드가 아니면 수정 없이 func을 반환한다.
    if os.environ.get('DEBUG','FALSE') != 'TRUE':
        return func

    # 디버깅 래퍼(wrapper) 만들기
    def call(*args,**kwargs):
        print("* {}.{} {} {}".format(cls_name, func.__name__, args[1:], kwargs))
        result = func(*args,**kwargs)
        print("    returning {}".format(result))
        return result
    return call

class DebugMeta (type): # 메타클래스
    def __new__(cls,name,bases,dict):
        if os.environ.get('DEBUG','FALSE') == 'TRUE':
            # 호출 가능한 모든 멤버를 찾아서
            # 디버깅 래퍼에 적용한다.
            for key,member in dict.items():
                if hasattr(member,'__call__'):
                    dict[key] = debugged(member, name)
        return type.__new__(cls,name,bases,dict)
```

다음 클래스 Rectangle은 DebugMeta를 메타클래스로 설정한다.

```
class Rectangle(metaclass=DebugMeta):
    def __init__(self,width,height):
        self.width = width
        self.height = height
    def area(self):
        return self.height*self.width

r = Rectangle(3, 4)
print(r.area())
```

환경 변수 설정에 따른 실행 결과는 다음과 같다.

```
C:₩Program Files₩Python36>set DEBUG=FALSE

C:₩Program Files₩Python36>python rectangle.py
```

```
12

C:¥Program Files¥Python36>set DEBUG=TRUE

C:¥Program Files¥Python36>python rectangle.py
* Rectangle.__init__ (3, 4) {}
    returning None
* Rectangle.area () {}
    returning 12
12
```