08. 클래스

08-01. 클래스 개요와 사용법

- 변수, 함수를 묶어서 코드 작성 실행 문법
- 객체지향 구현 : 실제세계를 모델링하여 개발하는 개발방법론
- 사용법 (메서드:클래스에서 선언된 함수)
- 클래스선언(코드작성) > 객체생성(메모리사용) > 메서드호출(코드실행)
- 클래스선언(설계도작성) > 객체생성(제품생산) > 메서드호출(제품사용)
- 클래스 식별자 컨벤션 : PascalCase
- class, self, spacial_methods(`__init__()`)
- 클래스는 사용자정의 데이터타입이다.

08-02. 클래스의 사용

- 클래스선언 : 코드작성
- 계좌 : Account : balance, deposit(), withdraw()

In [1]:
```
class Account:

    balance = 10000

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

- 객체생성 : 메모리사용

In [2]:
```
account1 = Account()
account2 = Account()
```

In [3]:
```
account1.balance, account2.balance
```

Out[3]:
```
(10000, 10000)
```

- 메서드호출 : 코드실행

In [4]:
```
account1.deposit(3000)
account2.withdraw(2000)
```

In [5]:
```
account1.balance, account2.balance
```

Out[5]:
```
(13000, 8000)
```

- dir() : 객체에 저장되어 있는 변수, 함수 출력

In [6]:
```
dir(account1)[-3:], dir(account2)[-3:]
```

Out[6]:
```
(['balance', 'deposit', 'withdraw'], ['balance', 'deposit', 'withdraw'])
```

08-03. self 명령어의 의미

- self : 객체자신
- Account.deposit(amount) : self.balance += amount
- account1.deposit(3000) : self.balance += amount
  - self == account1 : account1.balance += 3000

- Account.withdraw(amount) : self.balance -= amount
- account2.withdraw(2000) : self.balance -= amount
  - self == account2 : account2.balance -= 2000

In [7]:
```
print(account1.balance)
account1.balance = 15000
account1.balance
```

```
13000
```

Out[7]:
```
15000
```

In [8]:
```
account1.balance += 3000
account1.balance
```

Out[8]:
```
18000
```

08-04. 클래스를 사용하여 스타크래프트 마린 유닛 만드는 문제

- 클래스선언 : 코드작성
- 스타크래프트 : Marine : health=40, ap=5, attack(target)

In [ ]:
```
# CODE
```

In [9]:
```
class Marine:
    health, ap = 40, 5
    def attack(self, target):
        target.health -= self.ap
```

- 객체생성 : 메모리사용 : m1, m2

In [ ]:
```
# CODE
```

In [10]:
```
m1 = Marine()
m2 = Marine()
```

- 메서드호출 : 코드실행 : m1.attack(m2)

In [ ]:
```
# CODE
```

In [11]:
```
m1.attack(m2)
```

In [12]:
```
m1.health, m1.ap, m2.health, m2.ap
```

Out[12]:
```
(40, 5, 35, 5)
```

In [13]:
```
m1.health += 20
m1.ap += 2
```

In [14]:
```
m1.health, m1.ap, m2.health, m2.ap
```

Out[14]:
```
(60, 7, 35, 5)
```

In [15]:
```
m1.attack(m2)
```

In [16]:
```
m1.health, m1.ap, m2.health, m2.ap
```

Out[16]:
```
(60, 7, 28, 5)
```

08-05. 생성자 스페셜 메서드의 사용

- spacial methods
  - 앞뒤로 __가 붙은 메서드
  - 특별한 기능을 하는 메서드
  - 생성자 메서드 : `__init__()` : 객체를 생성할때 실행되는 메서드
    - 객체에서 변수의 초기값 설정
    - 변수가 선언되지 않아 다른 메서드를 실행할때 에러가 발생하는것을 방지

In [17]:
```
# 클래스선언 : 코드작성
class Account:
    # 객체에서 사용되는 변수의 초기값을 설정할때 사용
    def __init__(self, balance=10000):
        self.balance = balance
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
```

In [18]:
```
# 객체생성 : 메모리사용
account = Account()
```

In [19]:
```
# 메서드호출 : 코드실행
account.deposit(2000)
account.balance
```

Out[19]:
```
12000
```

08-06. 다양한 스페셜 메서드의 사용

- spacial methods
- `__add__()` : + 연산자 정의
- `__eq__()` : == 연산자 정의
- `__repr__()` : 객체의 속성 출력 정의
- `__str__()` : print() 함수의 출력 정의

In [20]:
```
# 데이터 타입에 따라서 수행되는 연산이 다르다
d1, d2, d3, d4 = 1, 2, '3', '4'
d1 + d2, d3 + d4
```

Out[20]:
```
(3, '34')
```

In [21]:
```
# __add__() == +
dir(d1)[:2], dir(d3)[:2]
```

Out[21]:
```
(['__abs__', '__add__'], ['__add__', '__class__'])
```

In [22]:
```
# d1.__add__(d2) == d1 + d2
# d1.__add__(d2) : 클래스 int : int 클래스에 정의되어 있는 __add__() 호출
# d3.__add__(d4) : 클래스 str : str 클래스에 정의되어 있는 __add__() 호출
d1.__add__(d2), d1 + d2, d3.__add__(d4), d3 + d4
```

Out[22]:
```
(3, 3, '34', '34')
```

In [23]:
```
class Number:
    def __init__(self, data):
        self.data = data
    def __add__(self, obj): # + 연산자 정의
        return self.data - obj.data
    def __repr__(self): # 객체의 속성값 출력 : 개발용
        print('repr')
        return f'<Number data:{self.data}>'
    def __str__(self): # 객체의 데이터 출력 : print() 함수실행
        print('str')
        return str(self.data)
```

In [24]:
```
n1 = Number(10)
n2 = Number(4)
```

In [25]:
```
n1
```

```
repr
```

Out[25]:
```
<Number data:10>
```

In [26]:
```
print(n1)
```

```
str
10
```

In [27]:
```
n1.__add__(n2), n1 + n2
```

Out[27]:
```
(6, 6)
```

08-07. 클래스와 데이터 타입의 관계

- 클래스는 사용자정의 데이터타입이다.

In [28]:
```
# 클래스선언 : 코드작성
class Account:
    # 객체에서 사용되는 변수의 초기값을 설정할때 사용
    def __init__(self, balance=10000):
        self.balance = balance
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
```

- acc 객체의 클래스 : Account
- acc 객체의 데이터타입 : Account
- 클래스는 데이터타입이다.
- Account 클래스는 우리가 직접 만듦 : 사용자정의
- 클래스는 사용자정의 데이터타입이다.

In [29]:
```
acc = Account()
type(acc)
```

Out[29]:
```
__main__.Account
```

In [30]:
```
# data 객체의 클래스는 : str
# data 객체의 데이터타입 : str
# str 클래스는 우리가 만들지 X
data = 'python'
type(data)
```

Out[30]:
```
str
```

In [31]:
```
# acc 객체의 클래스(Account:balance, deposit, withdraw)
# data 객체의 클래스(str:lower, split, count, upper ...)
dir(acc)[-3:], dir(data)[-3:]
```

Out[31]:
```
(['balance', 'deposit', 'withdraw'], ['translate', 'upper', 'zfill'])
```

In [32]:
```
# 객체는 데이터타입(클래스)에 따라서 사용가능한 변수, 메서드가 다르다.
data.upper()
```

Out[32]:
```
'PYTHON'
```

In [33]:
```
ls = [1, 2, 3]
# dir(ls)
help(ls.sort)
```

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

08-08. 클래스의 상속

- 클래스 상속
  - 다른 클래스의 변수와 메서드를 가져와서 사용하는 방법
  - 다중 상속 가능
    - IPhone1 : call()
    - IPhone2 : call(), sms()
    - IPhone3 : call(), sms(), internet()

In [34]:
```
class IPhone1():
    def call(self):
        print('calling')

class IPhone2(IPhone1):
    def sms(self):
        print('send sms')

# 다중상속 : IPhone1 > IPhone2 > IPhone3 : IPhone3(IPhone2, IPhone1)
class IPhone3(IPhone2):
    def call(self): # 메서드 오버라이딩
        print('calling face')
    def internet(self):
        print('use internet')
```

08-09. is a, has a 개념

- 클래스를 설계할때 사용하는 개념
- is a : A is a B : A는 B이다. : A와 B는 동일하다.
- has a : A has a B : A는 B를 포함한다.

In [35]:
```
# User 클래스 : name, age 정보
```

In [36]:
```
# is a : 상속
```

In [37]:
```
class Info:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

In [38]:
```
class User(Info):
    def disp(self):
        print(self.name, self.age)
```

In [39]:
```
user = User('peter', 20)
user.disp()
```

```
peter 20
```

In [40]:
```
# has a : 객체 > 생성자( 아규먼트, 파라미터 : 객체)
```

In [41]:
```
class Name:
    def __init__(self, name_str):
        self.name_str = name_str

class Age:
    def __init__(self, age_str):
        self.age_str = age_str
```

In [42]:
```
name = Name('peter')
age = Age(20)
```

In [43]:
```
name.name_str
```

Out[43]:
```
'peter'
```

In [44]:
```
class User:
    def __init__(self, name_obj, age_obj):
        self.name_obj = name_obj
        self.age_obj = age_obj
    def disp(self):
        print(self.name_obj.name_str, self.age_obj.age_str)
```

In [45]:
```
user = User(name, age)
user.disp()
```

```
peter 20
```

08-10. 클래스를 사용하여 스타크래프트 메딕 유닛 만드는 문제

- 문제. 아래의 코드에서 아래의 조건을 추가하여 클래스의 코드를 수정 또는 추가하세요.
  - 생성자 메서드 사용
  - max_health 추가
  - 체력이 0이하이면 공격해도 체력이 감소하지 않음

In [46]:
```
class Marine:

    health, ap = 40, 5

    def attack(self, target):
        target.health -= self.ap
```

In [47]:
```
# CODE
```

In [3]:
```
class Marine:

    def __init__(self, health=40, ap=5):
        self.max_health = self.health = health
        self.ap = ap

    def attack(self, target):
        target.health -= self.ap
        if target.health <= 0:
            target.health = 0
```

- 문제. 메딕 클래스를 생성하세요.
  - 메딕 클래스 생성 : Medic : health=60, hp=6
  - 체력이 최대 체력이면 치료해도 체력이 증가하지 않음

In [48]:
```
# CODE
```

In [4]:
```
class Medic:
    def __init__(self, health=60, hp=6):
        self.max_health = self.health = health
        self.hp = hp
    def heal(self, target):
        if target.health == 0:
            return
        target.health += self.hp
        if target.health > target.max_health:
            target.health = target.max_health
```

In [7]:
```
m1 = Marine()
m2 = Marine(70, 8)
m1.health, m1.ap, m2.health, m2.ap
```

Out[7]:
```
(40, 5, 70, 8)
```

In [8]:
```
m2.attack(m1)
m1.health, m1.ap, m2.health, m2.ap
```

Out[8]:
```
(32, 5, 70, 8)
```

In [9]:
```
medic = Medic()
medic.health, medic.hp
```

Out[9]:
```
(60, 6)
```

In [10]:
```
medic.heal(m1)
medic.heal(m1)
m1.health, m1.ap, m2.health, m2.ap
```

Out[10]:
```
(40, 5, 70, 8)
```

08-11. 클래스 메서드 종류

- Instance Method
  - self 사용
  - 객체를 통해서 사용
- Class Method
  - cls 사용
  - 클래스를 통해서 사용
- Static Method
  - 클래스 안에 있는 일반 함수

In [49]:
```
# 이자율이 적용된 계좌 클래스
```

In [50]:
```
class Account:

    interest = 0.01

    def __init__(self, balance=0):
        self.balance = balance

    def change_interest_instance(self, interest):
        self.interest = interest

    @classmethod
    def change_interest_class(cls, interest):
        cls.interest = interest

    @staticmethod
    def change_interest_static(interest, obj):
        obj.interest = interest
```

In [51]:
```
acc1 = Account(1000)
acc2 = Account(2000)
acc3 = Account(3000)
```

In [52]:
```
acc1.interest, acc2.interest, acc3.interest
```

Out[52]:
```
(0.01, 0.01, 0.01)
```

In [53]:
```
# instance method
acc1.change_interest_instance(0.03)
acc2.change_interest_instance(0.03)
```

In [54]:
```
acc1.interest, acc2.interest, acc3.interest
```

Out[54]:
```
(0.03, 0.03, 0.01)
```

In [55]:
```
# class method
Account.change_interest_class(0.04)
```

In [56]:
```
acc1.interest, acc2.interest, acc3.interest
```

Out[56]:
```
(0.03, 0.03, 0.04)
```

In [57]:
```
# static method
Account.change_interest_static(0.05, acc2)
```

In [58]:
```
acc1.interest, acc2.interest, acc3.interest
```

Out[58]:
```
(0.03, 0.05, 0.04)
```