オブジェクト指向(再)

別な角度（用語）から Python のオブジェクト指向を再度見直してみます。

■クラスを定義する

```
class クラス名：

  def メソッド名（self, 引数リスト）：
      self.変数＝・・・
      ・・・
      return 式
```

「**クラス**」には、「**データ属性**」と「**メソッド**」を記述します。

◆データ属性（data attribute）

Python の場合、「データ属性」と「メソッド」をまとめて、「**属性**」（attribute）と呼ぶことがある。

```
class Person:  //クラスの定義
   def getName(self):  //メソッドの定義
      return self.name  //データ属性
   def getAge(self):   //メソッドの定義
      return self.age
```

■インスタンスの作成

クラスを利用するには「**インスタンス**」（instance）を作成し、それを利用します。

```
文法：インスタンスの作成

      インスタンス名 ＝ クラス名（）
```

```
#1
class Person:
     def getName(self):  //メソッドを定義
            return self.name  //データー属性を表すには self をつける

     def getAge(self):
            return self.age

pr=Person()  //インスタンスの作成
pr.name='鈴木'  //データ属性に値を代入
pr.age=23

n=pr.getName()  //メソッドを呼び出します
a=pr.getAge()

print(n,"さんは", a, "才です。")
```

■データ属性の利用

文法：
インスタンス名.データ属性名
インスタンス名.メソッド名（引数リスト）

```
#1 に続けて～
pr1=Person
pr1.name='鈴木'
pr1.age=23
n1=pr1.getName()
a1=pr1.getAge()

pr2=Person
pr2.name='田中'
pr2.age=38
n1=pr2.getName()
a1=pr2.getAge()

print(n1, "さんは", a1, "才です")
print(n2, "さんは", a2, "才です")
```

■コレクションで多くのインスタンスを管理できる

people=[Person(), Person()]

■コンストラクタ

コンストラクタ（constructor）
インスタンスが作成される時に必ず処理されるメソッド
コンストラクタは「__init__」という名前のメソッドとして扱われる。

文法：コンストラクタ

```
Def __init__(self,引数リスト)：
```

```
#3  コンストラクタ
class Person:
  def __init__(self, name, age):
      self.name = name
      self.age = age

  def getName(self):
     return self.name

  def getAge(self):
     return self.age

pr = Person("鈴木", 23)
n = pr.getName()
a = pr.getAge()

print(n,"さん", a, "才です")
```

インスタンスが作成されるときにコンストラクタ（__init__）が呼び出される

■クラス変数･クラスメソッド

クラス全体で共有して値をもつ必要がある。共有できるデータ属性やメソッドを定義することができる。

データ属性やメソッドは、クラス全体に関連づけられる

```
#4
class Person:
     count = 0

     def geName(self):
           return self.name
```

クラスで定義されたデータ属性はクラスに一つだけの値が存在する。
このデータ属性を「**クラス変数**」とよぶ。
インスタンスごとに値の存在するデータ属性は「**インスタンス変数**」と呼ぶ。

◆クラスメソッドの仕組み

クラスの関連づけられたメソッドを「クラスメソッド」と呼ぶ。
クラスメソッドは、クラスの定義内に記述した@classmethod という指定の下に定義をする。

```
#5
@classmethod
def  getCount(cls):
     return cls.count
```

「cls」はクラス名を受け取るための引数。
クラスメソッドの 1 番目にはクラス名が渡される。

これまでの self を引数として持つメソッドは「**インスタンスメソッド**」と呼ばれる。

＜まとめ＞

・ラス全体で管理されるデータ属性（クラス変数）は、クラスの下で定義をする
・クラス全体で管理されるメソッド（クラスメソッド）は、@classmethod の下で定義される。

■クラスメソッド・クラス変数を利用する

文法：
クラス変数
Person.count
クラスメソッド
Person.getCount()

```
#6
class Person:
    count = 0

    def __init__(self, name, age):
        Person.count = Person.count + 1

        self.name = name
        self.age = age

    def getName(self):
        return self.name

    def getAge(self):
        return self.age

    @classmethod
    def getCount(cls):
        return cls.count

pr1 = Person("鈴木", 23)
pr2 = Person("佐藤", 38)

print(pr1.getName(), "さんは", pr1.getAge(), "才です")
print(pr2.getName(), "さんは", pr2.getAge(), "才です")
print("合計人数は", Person.getCount(), "です")
```

コンストラクタで count を 1 ずつ増やしている。

■カプセル化

オブジェクト指向では、データの属性を勝手に変更できないようにしておくことが重要。
したがって、クラスの外から「abc.age=」のようなアクセスをできないようにする必要がある。
このように、データを保護し、外部から勝手にアクセス出来ないようにすることを「カプセル化」という。

◆属性へのアクセス制限

Python では、言語構造のシンプルさから、クラスの外部からデータ属性を利用することができる。
そこで、Python のクラスを制限して定義する際には、アクセスを制限したい属性の名前を、アンダースコアではじめる

```
#7
class Person:
    def __init__(self, name, age):
        self._age = age
```

また、それを利用する際にもアンダースコアをつける

```
pr._age = 23
```

◆属性へのアクセスの禁止

書き換えられないように強制する方法もある。
この場合、属性名の前にアンダースコアを 2 つつける

```
#8
  class Person:
      def __init__(self, name, age):
          self._ _age = age
```

この場合、外部からアクセスは一切できなくなります。
こうした仕組みを「マングリング」と呼ぶ

◆セッターとゲッター

　データを設定するメソッド「**セッター**」(setter) と、データを取得するメソッド「**ゲッター**」(getter) がある。　組み込み関数 property()によって指定できます
ただし、あまり使われない。

文法：

```
class Person:
  def データを設定するメソッド名（self, ・・・）
  def データを設定するメソッド名（self, ・・・）
age = property(データを設定するメソッド名、データを取得するメソッド名)

pr = Person()
pr.age = 値
変数 = pr.age
```

■継承

元となるクラス(**基底クラス** base class)を拡張(extends)して新しいクラス(**派生クラス** derived class)を作ることができます。これを継承（inheritance）と呼びます。

```
文法：
  class 派生クラス名（基底クラス名）：
      def 派生クラスに追加するメソッド（self, 引数のリスト）：
          self.派生クラスに追加するデータ属性　＝　値
```

＃2 の Person クラスをもとに顧客クラス Customer を定義する

```
#9
class Customer(Person):  ・・・派生クラスを定義
    def __init__(self, nm, ag, ad, tl):  ・・派生クラスのコンストラクタ
        super().__init__(nm, ag)  ・・基底クラスを初期化するためコンストラクタ
        self.adr = ad  ・・追加するデータ属性
        self.tel = tl

    def getName(self):  ・・・基底クラスのメソッドを上書きする
        self.name = "顧客：" + self.name
        retuen self.name

    def getAdr(self):
        retuen self.ad  ・・・追加するメソッド

    def getTel(self):
        return self.tel  ・・・追加するメソッド
```

＜まとめ＞

・基底クラスから派生クラスを拡張できる

・基底クラスは派生クラスの属性を継承する

■オーバーライド

基底クラスと派生クラスには同じ名前のメソッドを持つことができる。
たとえば、Person クラスでの getName () メソッドと同じものを、Customer クラスでも持つことができる。

このように、派生クラスのメソッドが、基底クラスのメソッドに変わって機能することを「**オーバー**」[1]と呼ぶ。

```
#10
  class Person:
      ・・・
      def getName(self):   ・・・基底クラスの getName メソッド
          return self.name


  class Customer(Person):
      ・・・
      def getName(self):   ・・・派生クラスの getName メソッド
          self.name = "顧客："+self.name
      ・・・
      return self.name
```

[1] Python には標準でオーバーロードがありません。これは Python には型がないことに起因しています。ただ、バージョン 3.4 からは特殊な方法で実現ができます。お勧めではありません。

■派生クラスの利用

```
#11
  class Person:  ・・・基底クラス
      def __init__(self, name, age):
          self.name = name
          self.age = age

      def getName(self):
          return self.name

      def getAge(self):
          return self.age

  class Customer(Person):  ・・・派生クラス
      def __init__(self, nm, ag, ad, tl):
          super().__init__(nm, ag)
 ・・・基底クラスのデータ属性を初期化するため、基底クラスのコンストラクタを呼び出す

          self.adr = ad  ・・・追加するデータ属性
          self.tel = tl

      def getName(self):  ・・・基底クラスメソッドの上書き
          self.name = "顧客：" + self.name
          return self.name

      def getAdr(self):  ・・・追加するメソッド
          return self.adr

      def getTel(self):  ・・・追加するメソッド
          return self.tel

  pr=Customer("鈴木", 23, "mmm@nnn.nn.jp", "xx-xxx-xxxx")

  nm=pr.getName()
  ag=pr.getAge()
  ad=pr.getAdr()
  tl=pr.getTel()

  print(nm, "さんは", ag, "歳です")
  print("アドレスは", ad, "電話番号は", tl, "です")
```