第Ⅲ編　クラスと継承

1　クラス・変数・メソッド

(1) クラス・メソッド・変数の宣言

クラスの一例を示す。

```
class クラス名:

    def _ _init_ _( self ):# コンストラクタ
            self.インスタンス変数 = ""

    def _ _str_ _( self ):  #文字列化
            return "クラスの説明など"

    def _ _del_ _( self ):  #デストラクタ
            print("クラスの終了")

    def メソッド名( self ):          # メソッド
            return self.carname

    def メソッド名( self, 引数名 ): # ソッド
            self.インスタンス変数 = 引数名
```

Java言語などと比較：

<同じ>

(1) 構造は、同じように、コンストラクタ、デストラクタ、メソッドがある。
　　呼び名もクラスの場合は関数ではなくメソッドと呼ぶ。

<相違>

(1) 文字列化メソッドがある。これにはクラスの説明を記述する。
(2) コンストラクタ、デストラクタの書き方は**_ _int_ _(), _ _del_ _()**になる。
(3) メソッドなどには　修飾子(指定子)の所に、関数の　**def**　をつける。
(4) 修飾子がない。
(5) 第１引数には、必ず**self**(呼び出し元のオブジェクト(インスタンス))を用いる。
　　呼び出すときには、**第１引数は無視**する。
(6) 引数は関数と同じで初期値データを設定することができる。
(7) メソッドの中で**self.変数名とインスタンス変数**になる。
　　つまり、self.インスタンス変数として用いる。すべてのメソッドで用いることが可能である。

上のクラスを参考に、クラス・メソッド・変数の宣言および呼び出しなどを示す。
例は、クラスはMyCarClass，オブジェクトはcar　である。

(A) クラス定義とオブジェクト生成

クラスの定義		
書式：		
	class クラス名:　# クラス名は任意、一般的には大文字で始める **#ブロック開始** **#ブロック終わり**	
オブジェクト(インスタンス)の生成		
	オブジェクト変数名=クラス名() または **オブジェクト変数名=クラス名(引数1,引数2,..)**	
クラスの定義	**class MyCarClass:** **#メソッド**	
オブジェクト生成 （インスタンス）	**car = MyCarClass()**	# オブジェクトを生成

(B) コンストラクタ　# __ は2個の_であるが、記述は便宜上スペース _ _ を入れた。

コンストラクタ		
書式(標準型)：（インスタンス変数の初期化のために使う）		
	def _ _init_ _(self):# コンストラクタ **# ブロック（初期化などを記述）**	
呼び出し：		
	上記のオブジェクト生成時に自動的に呼び出される。	
コンストラクタ	def _ _init_ _(**self**): 　self.carname = ""	# コンストラクタ
呼び出し (オブジェクトの生成)	car = MyCarClass()	
コンストラクタ	def _ _init_ _(**self, carName="Crown"**):	# 変数に初期値がある場合
	self.carname = carName	
呼び出し (オブジェクトの生成)	car = MyCarClass(**"Prius"**)	
コンストラクタ	def _ _init_ _(**self, carName**):	# 引数がある場合
	self.carname = carName	
呼び出し (オブジェクトの生成)	car = MyCarClass(**"Prius"**)	

(C) オブジェクトの文字列化と呼び出し

オブジェクトの文字列化		
書式：（オブジェクトと変数に文字列を組み込む） def _ _str_ _(self):# 文字列化 return "（オブジェクトの文字列化）"		
呼び出し： print(オブジェクト)　で呼び出され表示される。		
文字列化 呼び出	def _ _str_ _(self): return "車クラス" car　or　print(car)	#文字列化

(D) デストラクトと自動呼び出し

デストラクタ		
書式：（オブジェクトが消滅するときに呼び出される） def _ _del_ _(self):# デストラクタ # ブロック（消滅するときにの処理を記述）		
呼び出し： 上記のオブジェクト消滅時に自動的に呼び出される。		
デストラクタ 呼び出	def _ _del_ _(self): print("車クラスの終了") (なし)	#デストラクタ # 他の言語のように close()とかDestroy()関数を呼び出さなくてもプログラムの終了時に自動で呼ば

(E) メソッドと呼び出し

メソッド(関数)		
書式:(#ブロック内に記述) **def メソッド名(self, 引数1, 引数2, ..)** **return 戻り値**		
呼び出し： **オブジェクト変数名.メソッド名(引数1の値, 引数2の値, ...)**		
メソッド	def getCarName(self): return self.carname	# getCarName()メソッド
メソッド	def setCarName(self, carname): self.carname = carname	# setCarName()メソッド
呼び出し	car.setCarName("TOYOTAクラウン") car.getCarName()	

(F)　インスタンス(フィールド)変数

インスタンス(フィールド)変数の参照		
定義：　クラスのメソッド中で定義 **self.インスタンス変数名**		
参照 クラスメソッド内 **self.インスタンス変数名** クラスの外では： **オブジェクト変数名.インスタンス変数名** **# 注意：インスタンス変数がない場合には、クラス変数を参照する。**		
参照	self.carname	# クラスメソッド内
	car.carname	# クラス外

下記のプログラムをサクラエディットで作成し、
文字コードUTF-8で[C:¥pythonPG¥sample3010.py]として保存しなさい。

```
# 例題ファイル：C:¥pythonPG¥sample3010.py

print( "==========クラス定義==================" )
class MyCarClass:
        def __init__(self):  # コンストラクタ
                self.carname = "" # インスタンス変数 フィールド変数）

        def __str__(self):   #文字列化
                return "車クラス"

        def __del__(self): #デストラクタ
                print("車クラスの終了")

        def getCarName(self):          # getCarName()メソッド
                return self.carname

        def setCarName(self, carname):  # setCarName()メソッド
                self.carname = carname

print( "==========オブジェクト生成============" )
car = MyCarClass()    # インスタンスを生成
print(car)            # オブジェクトの文字列化表示

car.setCarName("TOYOTAクラウン")              # メソッドの呼び出し

print ("メソッドアクセス==", car.getCarName() ) # メソッドの呼び出し

print("オブジェクトアクセス==", car.carname )  # オブジェクト.インスタンス変数の参照
```

(2) クラス変数とクラスメソッド/スタティックメソッド

変数：	インスタンス変数(前項に記述) **クラス変数**
メソッド：	インスタンスメソッド(前項に記述) スタティックメソッド **クラスメソッド**

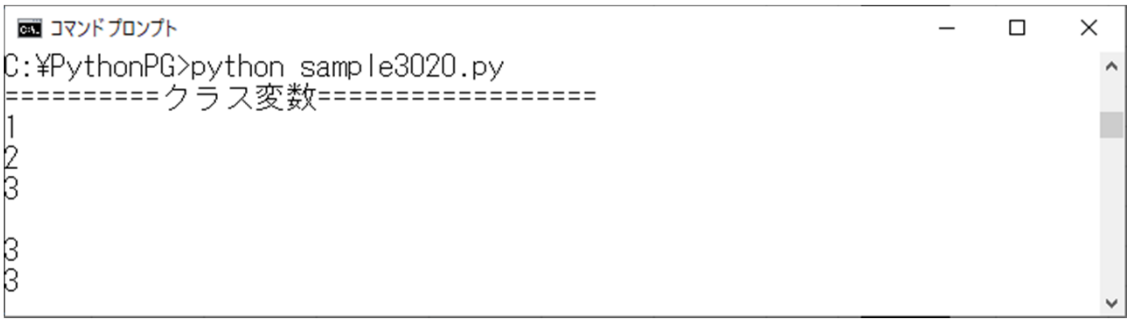
A　クラス変数

クラス変数はオブジェクトに無関係であるので、オブジェクト間で共通な値を持つ。

定義	class MyClass: 　　**クラス変数の定義**　　# この場所に定義した変数(static不要) 　　コンストラクタ関数 　　　　**インスタンス変数の定義** 　　メソッド 　　　　**インスタンス変数の定義**
アクセス	**クラス名.クラス変数**　　#クラス変数へのアクセス 　　# オブジェクトを生成する必要なし (オブジェクトを生成してから下記を実行) **オブジェクト.クラス変数**　　#オブジェクトからもアクセス オブジェクト.インスタンス変数　　#インスタンス変数へのアクセス

例題	#　例題ファイル：C:¥pythonPG¥sample3020.py
	print("==========クラス変数==========")
	
	class MyClass:
	　　access_count = 0　　# クラス変数の初期化
	
	　　def __init__(self):
	　　　　MyClass.access_count += 1　　# クラス変数へのアクセス
	　　　　print(MyClass.access_count)
	a1 = MyClass()
	a2 = MyClass()
	a3 = MyClass()
	print()
	print(MyClass.access_count)　　# クラス変数へのアクセス
	print(a1.access_count)　　# オブジェクトからクラス変数へのアクセス

上記のプログラムを[C:¥pythonPG¥sample3020.py]を作成し、実行しなさい。



B クラスメソッド

クラスメソッド	
書　式	**@classmethod** def クラスメソッド名(**cls**, 変数1, 変数2, ...) #ブロック # ブロック
呼び出し	オブジェクト.クラスメソッド名(変数1の値, 変数2の値, ...) または クラス.クラスメソッド名(変数1の値, 変数2の値, ...)
	clsには呼び出し元のオブジェクトのクラスが挿入される。 故に、　object1 = cls()　**# 新規のオブジェクト生成**　　**変数＝初期値** とすると、クラスのオブジェクトを生成したことになる。

つぎのプログラムを作成し、[C:¥pythonPG¥sample3030.py]として保存しなさい。

```
1  #  例題ファイル：C:¥pythonPG¥sample3030.py
2
3  class ClassMethods:
4      vari = 0    # クラス変数
5
6      def __init__(self):
7          self.vari = 0
8          ClassMethods.vari = ClassMethods.vari + 1
9
10     @classmethod
11     def class_method( cls ):
12         """(クラス)メソッド呼び出し"""
13         print("クラス変数vari=", cls.vari)
14
15 object = ClassMethods()
16 print( "クラスメソッドへの２通りアクセス" )
17 object.class_method( )
18 ClassMethods.class_method( )
19
```

実行結果：

```
コマンド プロンプト
C:¥PythonPG>python sample3030.py
クラスメソッドへの２通りアクセス
クラス変数vari= 1
クラス変数vari= 1
```

(3) アクセス制限（カプセル化：外部から勝手にアクセスできない事）

インスタンス変数やメソッドについて、他言語のようにアクセス制限に関する
修飾子(指定子(public, protected, private))などがなく、
すべてどこからもアクセスが可能(public)である。

ただし、**(_)と (_ _)**について
(_)で始まる変数名やメソッド名はクラス外からアクセスしないという慣習的なルールがある。
また、コンストラクトのように(_ _)で始まるものは**クラス外からアクセスが不可能**である。
アクセス不可能：(AttributeError例外)
いわゆる、Java言語などのprivateメソッドである。

なお、　オブジェクト名._クラス名__変数名
オブジェクト名._クラス名__メソッド名（）でアクセス可能です。
また、この代表的な例は、コンストラクタ：_ _init(self)_ _である。

下記のプログラムを[C:¥pythonPG¥sample3045.py]に作成し、実行しなさい。

```
#  例題ファイル：C:¥pythonPG¥sample3045.py
↵
class CapsuleClass:↵
    def __init__( self ):↵
        print("+++ アクセスコンストラクタに：__ini__() +++")↵
        self._AAA = "AAA"↵
        self.__BBB = "BBB"↵
    ↵
    def __funcA( self ):↵
        print("=== アクセス：__funcA() ===")↵
↵
print()↵
myclass = CapsuleClass()        # オブジェクトの生成↵
print( "アクセス変数に:_AAA  ==>", myclass._AAA )↵
print( "アクセス変数に:_ _BBB==>", myclass._CapsuleClass__BBB )↵
myclass._CapsuleClass__funcA() # メソッドにアクセス↵
↵
print()↵
myclass.__init__( )           # コンストラクタにアクセス↵
↵
```

実行結果：

```
コマンド プロンプト
C:¥PythonPG>python sample3045.py

+++ アクセスコンストラクタに：__ini__() +++
アクセス変数に:_AAA  ==> AAA
アクセス変数に:_ _BBB==> BBB
=== アクセス：__funcA() ===

+++ アクセスコンストラクタに：__ini__() +++
```

2　クラスの継承

(1) クラスの継承

つぎのような既存クラスがあるとする。

既存クラス=CarClass :

```
class CarClass:
        def __init__(self, num="1234"):
                self.__number = num
        def setnumber(self, num):
                self.__number = num
        def getnumber(self):
                return self.__number
```

このとき、つぎのような新しいクラスを考える。

新しいクラス=CopanyCar :

```
class CompanyCar:
        def __init__(self, num="1234"):
                self.__number = num
        def setnumber(self, num):
                self.__number = num
        def getnumber(self):
                return self.__number
        def setcompany( self, company ):
                self.company = company
        def getcompany(self):
                print(self.company)
```

既存クラスと新しいクラスの灰色部分は**重複**していることが分かる。
このCompanyCarクラスに既存のCarClassをつぎのように用いることができる。
これが、**クラスの継承**である。

書式	**class クラス名(継承クラス名):**

CarClass :　　スパークラス
CompanyCar :　サブクラス

```
class CompanyCar(CarClass):
        def setcompany( self, company ):
                self.company = company
        def getcompany(self):
                print(self.company)
```

(2) オーバーライド(メソッド)

継承クラスにおいてスーパークラスと**同じメソッド名やコンストラクタ**を**定義可能**である。

A　メソッドつぎのようなスーパークラスに対して

スーパークラス=CarClass :

```
class CarClass:
        def __init__(self, num="1234"):
                self.number = num
        def setnumber(self, num):
                self.number = num
        def getnumber(self):
                print(self.number)
```

上記のスーパークラスと同じ名前のメソッドgetnumber()をもつ次のサブクラスを考える。

CarClass :　　スパークラス

CompanyCar :　サブクラス

```
class CompanyCar(CarClass):
        def getnumber( self ):
                super().getnumber()
                print( "オーバーライドメソッド",self.number)
```

赤いい部分がスーパークラスに付加された部分になっている。

スーパークラスのメソッドの呼び出	super().メソッド名()

B　コンストラクタのオーバーライドと親クラスのメソッド呼び出し

書 式	super( ).メソッド名(引数)

つぎのようなスーパークラスに対して

スーパークラス=CarClass :

```
class CarClass:
        def __init__(self, num="1234"):
                print("スーパークラスのコンストラクタ", num)
                self.number = num
        def setnumber(self, num):
                self.number = num
        def getnumber(self):
                return self.number
```

上記のスーパークラスに対してコンストラクタをもつ次のサブクラスを考える。

CarClass :　　スパークラス

CompanyCar :　サブクラス

```
class CompanyCar(CarClass):
        def __init__(self, num):
                super( ).__init__( num )
                print( "サブクラスのコンストラクタ",self.number)
```

(3) 多重継承

Javaなどでは継承は**スーパークラス1個**であるが、Pythonは**複数個可能**である。
そのためにJava等では、interface がある。

書式	**class　クラス名(スーパークラス１, スーパークラス2, …)**

スーパークラス：Super11, SuperAA

```
class Super11:
        def __init__(self):
                print("11111")
        def func11(self):
                print ( "Super11:func11" )
class SuperAA:
        def __init__(self):
                print("AAAAAA")
        def funcAA(self):
                print ( "SuperAA:funcAA" )
```

サブクラス：

```
class SubClass( SuperAA, Super11 ):
        pass
```

スーパークラス間にコンストラクタや同じ名前のメソッドが存在する場合には
継承を指定する（スーパークラス１, スーパークラス2, …）**内の先に書いたクラスが優先**する。

(4) import文 (モジュール化)

前の例で

スーパークラス：Super11, SuperAA

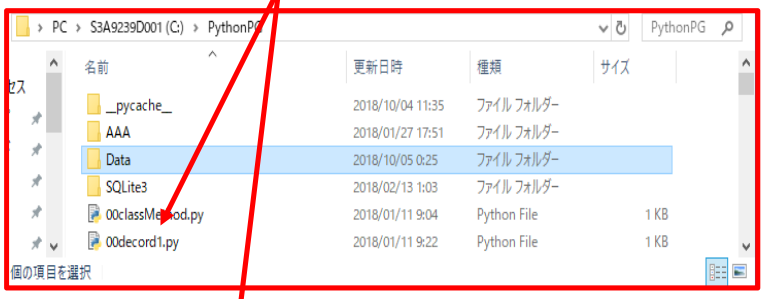
```
class Super11:
        def __init__(self):
                print("11111")
        def func11(self):
                print ( "Super11:func11" )
```

```
class SuperAA:
        def __init__(self):
                print("AAAAAA")
        def funcAA(self):
                print ( "SuperAA:funcAA" )
```

が、同じディレクトリ（パッケージ）の別ファイルとか、
別ディレクトリにあるなどモジュール化してある場合を考える。
これを使う場合はつぎのような書式である。

書　式	# 組み込み
	import パス.ファイル名 import パス.ファイル名 as 別名 from パス.ファイル名 import クラス名（または、関数名や変数名）
	# 使い方
	パス.ファイル名.クラス名（または、関数名・変数名） 別名.クラス名（または、関数名・変数名） クラス名（または、関数名・変数名）

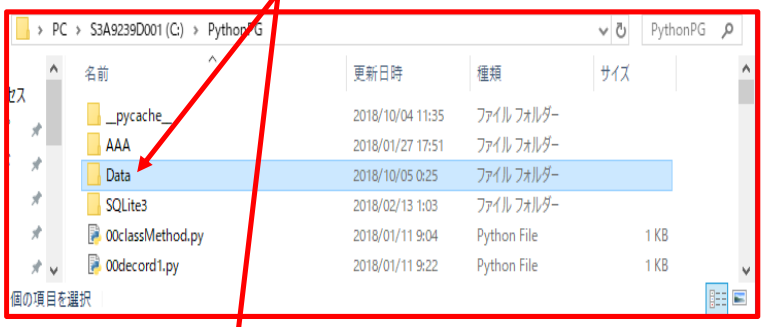
同じディレクトリ (C:¥PythonPG) に super11aa.py の名前で保存されている場合を考える。



この場合、サブクラスはつぎのように作成する。

書式	**import super11aa** class SubClass(**super11aa.**SuperAA, **super11aa.**Super11): 　　pass

別ディレクトリ (C:¥PythonPG¥Data) に super11aa.py の名前で保存されている場合を考える。



この場合、サブクラスはつぎのように作成する。

書式	**import Data.super11aa** class SubClass(**Data.super11aa.**SuperAA, **Data.super11aa.**Super11): 　　pass

つぎのプログラムを作成し、[C:¥pythonPG¥super11aa.py], [C:¥pythonPG¥sample3090.py] として同じフォルダに保存し、実行しなさい。

```
# 例題ファイル：C:¥pythonPG¥super11aa.py

print( "===== 2個のスーパークラス ==========" )
class Super11:
        def __init__(self):
                print("11111")
        def func11(self):
                print ( "Super11:func11" )
class SuperAA:
        def __init__(self):
                print("AAAAAA")
        def funcAA(self):
                print ( "SuperAA:funcAA" )
```

```
# 例題ファイル：C:¥pythonPG¥sample3090.py
#import Data.super11aa as Dsuper
import super11aa

print( "====== サブクラス ========" )

#class SubClass( Dsuper.SuperAA, Dsuper.Super11 ):
class SubClass( super11aa.SuperAA, super11aa.Super11 ):
        pass

object = SubClass()
object.func11()
object.funcAA()
```

実行結果：