■インスタンスのアトリビュート

・クラスからインスタンスを作る

ただクラスからインスタンスを生成した状態では、インスタンスは何のデータもメソッドも持っていない

Python インスタンスには「**アトリビュート**」と呼ばれる変数のようなものが備わっている
したがって、オブジェクトに代入することでデータを定義できる。

オブジェクト abc に value を定義する

```
abc.value = 5
```

アトリビュートは実際に代入を行ったインスタンスにしか定義されない。

■初期化メソッド　_ _init_ _

たとえば、未定義のアトリビュートを参照するとエラーになる。そこで、インスタンスに数値データを持たせるためには、毎回アトリビュートへの代入をする必要がある。これを毎回するのは大変。

メソッドには「def」を使う。初期化メソッドとして引数には必ず「self」を使う。この self を使うとインスタンス自体を操作できる。

◆Java、C＋との違い

よく使われる言語では、クラスを作るときにインスタンスにどのようなデータ（メンバ）を持たせるか定義ができる。しかし、Python にはその機能がない。
Python では、メンバを定義する際にはインスタンスのアトリビュートに代入する必要がある。[1]

[1] ただし、メタクラスを使えばそれができるが・・・

■メソッドと第 1 引数 self

第 1 引数は必ず [2]self

幅（width）、高さ（height）、奥行（depth）から体積（Prism）を求めるクラス

```
#1
class Volume:
      def _ _init_ _(self, width, height, depth):
            self.width=width
            self.height=height
            self.depth=depth

  def content(self):
       return self.width*self.height*self.depth
```

＃1 のクラスからインスタンス abc1 を生成して、実行するプログラム

```
#2
  abc1=Volume(10, 20, 30)
  print(abc1.content())
```

別なインスタンス abc2 の異なった数値からは、違った結果になる

```
#3
  abc2=Volume(50, 60, 70)
  print(abc2.content())
```

◆インスタンスから個別のアトリビュートを取り出す

```
#4
  print(abc1.height)
  print(abc2.height)
```

このプログラムを正しく動かすためには、＃1 のプログラムをもう一度動かす必要がある。

---

[2] 実際には self は慣習であって、ほかの言語と同じく this などを使うこともできる

◆アトリビュートを書き換える

```
#5
  abc1.height=50
  print(abc1.content())
```

◆サンプル問題

もう少し単純な例で、幅と高さから面積を求めるクラス
「ア」には何が入るか？

```
#6
 class Square:
       def _ _init_ _(self, width, height):
              self.width = width
              self.height = height
       def area(ア):
              return self.width*self.height
```

■アトリビュートを隠蔽

アトリビュートにはさまざまなものが代入できる
「数値」や「文字列」だけではなく「リスト」なども扱える。しかし、「型」の間違った使い方などをするととんでもない結果を返してくる。

そこで、外部からの利用をできなくするために「**カプセル化**」（encapsulation）という方法がオブジェクト指向にはある[3]。

Python では、クラスの外からアクセスをできなくする方法には 2 つある

・アトリビュート名やメソッド名の先頭にアンダースコア「_」を 1 つ付ける

例：　_size

・アトリビュート名やメソッド名の先頭にアンダースコアを 2 つ付ける

より厳しいアクセスの制限をする。

例：　_ _size

◆クラス外からの操作を制限する

```
class Food:
   def __init__(self, name, price):
      self._name = name
      self._price = price

   def show(self):
      print(self._name, self._price)

x = Food('milk', 150)
x._price //= 2
x.show()
```

[3] C++や java 言語では「public」や「private」などのキーワードでデータへのアクセスを制限できます

さらに念入りに属性の操作を制限したい場合には「__ __」(2 個のアンダースコア) を属性名の前に付けます。こうした働きを「**マングリング**」(mangling) とよびます。この機能は、データ名だけではなくメソッド名にも使えます。

しかし、この場合、次の記述では**エラー**になります。(単に 2 つのアンダースコアを付けただけ)

```
class Food:
   def __init__(self, name, price):
      self._ _name = name
      self._ _price = price

   def show(self):
      print(self._ _name, self._ _price)

x = Food('milk', 150)
x._ _price //= 2
x.show()
```

このエラーを解消するには、次のように記述する必要があります。[4]

クラス名 _属性名

```
x = Food('milk', 150)
x._Food_ _price //= 2
x.show()
```

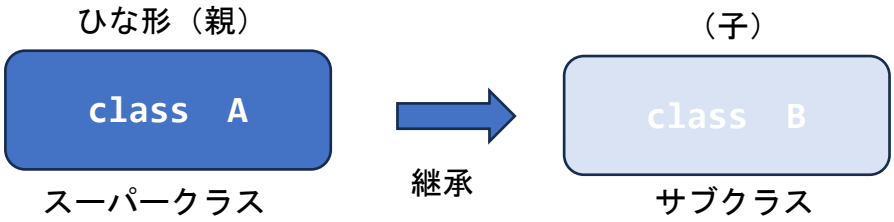
__init__だけは例外なので、先頭に 2 つのアンダースコアがついても外部からアクセスすることができます。

[4] 実は、アンダースコアを 2 つ付けても Python は完全にはアトリビュートやメソッドを隠すことができない。Python のオブジェクト指向は非常にシンプルですが、その反面、こうした不確実な部分もある。だからと言って、「Python のオブジェクト指向は劣っている」という声は聞きません。その辺は、うまくやっていくことです！

■クラスの継承

継承（inheritance）とは、あるクラスを雛形にして別のクラスを作ること。
雛形になる「親」のクラスを「スーパークラス」。スーパークラスを雛形にして作られたクラスを「サブクラス」と呼びます。

サブクラスは、スーパークラスのメソッドなどを引き継ぎ、必要であれば修正もできます。また、新たに機能を加えることもできます。



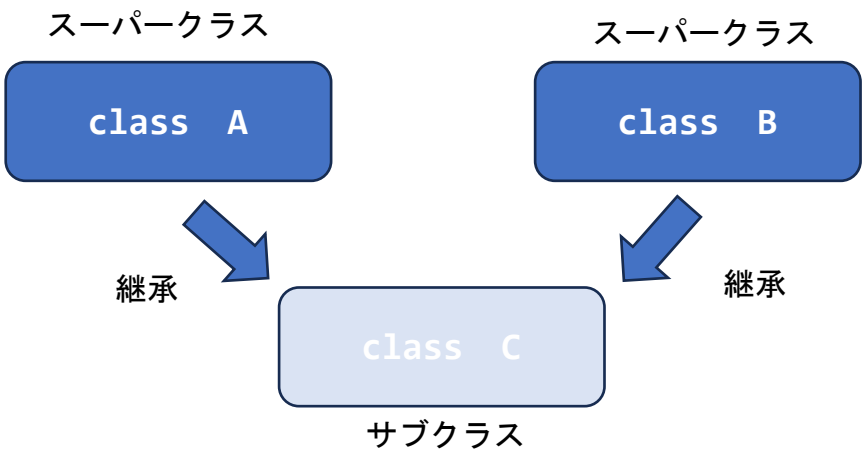
サブクラスの定義には、親クラス（スーパークラス）を（）で指定します。

```
スーパークラスの定義：
  class サブクラス名（スーパークラス名 1、・・・）

  class Cube(Volume):
```

◆多重継承

Python は、複数のクラスを親にして新しいクラスを作る「多重継承」（multiple inheritance）ができます。
複数の親クラスがある場合には「,」（コンマ）で区切って表記します。



◆メソッドのオーバーライド

スーパークラスのメソッドは、サブクラスにそのまま引き継がれる。
機能を変更したい場合は、サブクラスのメソッドを定義しなおせば良い。こうした上書きをメソッドの「オーバーライド」という。

```
 #7
～#1：スーパークラスの記述
class Cube(Volume):  //Volume クラスを継承
      def _ _init_ _(self, length):  //メソッドを再定義
            self.width = self.height = self.depth =length
                                    // アトリビュートを length で初期化

 c= Cube(20)  //length に 20 を渡す
 print(c.content())  //Volume メソッドを呼び出す
```