Classes abstraites

- **Elle permet de décrire un modèle abstrait**, regroupant un certain nombre de caractéristiques communes.
- **Les caractéristiques de la classe abstraite serviront de base** à la création de nouvelles classes.
- **Une classe abstraite est une classe dont il ne peut exister aucune instance**

Classes abstraites

Python ne propose pas de **mécanisme natif** pour déclarer des classes abstraites étant donné le *duck typing.*

Oui, mais non…

➔ Introduction de la bibliothèque ***abc (Abstract Base Classes : classes de base abstraites)***, qui permet de déclarer une classe abstraite et des méthodes abstraites.

Classes abstraites

Exemple (1) : définition d'une classe Forme abstraite

```
>>> from  abc  import  ABCMeta, abstractmethod

class Forme ( metaclass = ABCMeta ):

    @abstractmethod
    def  perimetre( self ):
        print( '' Calcul du périmètre…'' )
```

Forme spécifie sa métaclasse

Décorateur : la méthode est abstraite

Le décorateur indique que la méthode doit être redéfinie dans les classes dérivées de Forme

Classes abstraites

Exemple (3) : définition d'une classe Forme abstraite

```
>>>  c = Cercle( 5 )

>>>  c.perimetre()
```

Calcul du périmètre …
31.400000000000002

```
>>> class Etoile ( Forme ) :
        pass

>>>  e = Etoile()
```

Erreur car Etoile() ne redéfinit pas perimetre()

TypeError: can't instantiate abstract class Etoile with abstract methods perimetre

Classes abstraites

Il est toujours possible d'affilier manuellement une classe concrète à une classe abstraite grâce au module *abc*.

- L'affiliation est réalisée grâce à la méthode de classe **register()** récupérée par la classe abstraite en déclarant ABCMeta en tant que métaclasse.

- **La classe concrète devient classe fille de la classe abstraite**

  ➔ elle est instanciable, même si elle n'implémente pas les méthodes abstraites de sa classe mère

La classe concrète fille n'hérite cependant pas des membres (attributs et méthodes) de la classe abstraite.

Classes abstraites

Exemple (1) : définition d'une classe Forme abstraite

```
>>> class Etoile (object) :        <-------- classe de base
        pass

>>>  Forme.register( Etoile )
<class '__main__.Etoile'>

>>> e = Etoile ()
>>> isinstance( e, Forme )
True
```

La classe de base Etoile () est bien affiliée à la classe abstraite Forme ()

Classes abstraites

Exemple (2) : définition d'une classe Forme abstraite

```
>>> e = Etoile()
>>>  print(e)
<__main.Etoile object at 0x…>

>>> e.perimetre( )
AttributeError: 'Etoile'  object has no attribute 'perimetre'
```

L'objet *e* n'a pas hérité de la méthode abstraite *perimetre*()

Visibilité des membres

En programmation OO, on distingue trois niveaux de visibilité des membres (attributs, méthodes) d'une classe.

- **Les membres privés** que l'on ne peut accéder que depuis la classe elle-même.
- **Les membres protégés** que l'on ne peut accéder que depuis la classe elle-même et depuis les classes dérivées de cette classe.
- **Les membres publics** auxquels on peut accéder de partout.

Visibilité des membres

Exemple (1) :

```
>>> class Vecteur2d ( object ) :
        def __init__( self, x, y ) :
            self.x, self.y  = x, y
        def getX( self ) :
            return self.x
        def getX(self):
            return self.y

        def setX(self, valeur ):
            self.x = valeur

        def setY(self, valeur ):
            self.y = valeur
```

Visibilité des membres

Exemple (2) :

```
>>> v =Vecteur2d ( 5, 7 )
>>>  v.getX( )
5
>>>  y.getY( )
7
>>>  v.setX(15)
>>>  v.setY(17)
>>> print ( v.getX( ), v.getY( ) )
15  17
```

Visibilité des membres

Exemple (3):

```
>>> v.x = 'A'
>>> v.y = 5.9
>>> v.getX( )
A
>>> y.getY( )
5.9
>>> v.setX(8)
>>> v.setY(100)
>>> print ( v.getX( ), v.getY( ) )
9  100
```

Visibilité des membres

Dans Python, par défaut, tous les membres d'une classe sont publics.

- Tous les membres de la classe sont accessibles depuis l'extérieur de la classe.
- La définition et l'utilisation des *getters* et *setters* ne changent pas ce fait.

➔ Les attributs x et y sont des *attributs publics*.

Visibilité des membres

Python possède un mécanisme appelé *name mangling* qui permet d'accéder aux membres d'une classe de l'extérieur de la classe.

Il remplace automatiquement tout membre d'une classe de la forme :

__nomDuMembre

par :

_nomDeClasse__ nomDuMembre

Visibilité des membres

➔ ***Ce mécanisme est proche de la notion de private que l'on trouve dans des langages OO tels que C++ ou Java.***

Attention : ce n'est pas exactement private !

Visibilité des membres

Par convention :

- Les noms des membres ''*privés*'' d'une classe sont préfixés avec deux (2) underscores :

  __nomDuMembre

- Les noms des membres ''*protégés*'' d'une classe sont préfixés avec un (1) underscore

  _nomDuMembre

Attributs privés

Un attribut privé est un membre de la classe qui n'est accessible qu'en lecture de l'extérieur de la classe.

- Le nom de l'attribut ''*privé*'' est préfixé avec deux (2) *underscores*

- Le décorateur **@property** est utilisé pour marquer la méthode qui permet d'accéder à tout attribut ''*privé*'' (getter)

- Le nom de l'attribut est aussi le nom de la méthode d'accès.

Attributs privés

Exemple (1) :

```
>>> class Vecteur2d ( object ) :
        def __init__( self, x, y ) :
            self.__x,  self.__y  = x,  y

        @property
        def x ( self ) :
            return self.__x
        @property
        def y (self):
            return self.__y
```

Attributs privés

Exemple (2) :

```
        def setX(self, valeur ):
            self.__x = valeur

        def setY(self, valeur ):
            self.__y = valeur

        def __iter__( self ):
            retrun ( i  for  i   in  ( self.x,   self.y ) )
```

Attributs privés

Exemple (3) :

```
>>> vect  = Vecteur2d ( 7, 14 )
>>> vect.x
7
>>> vect.y
14
>>> vect.x = 17
AttributeError : can't set attribute
>>> vect.setX (17)
```

Modification possible via le *setter*

Attributs privés

Exemple (4) :

```
>>> vect.x
17
>>> vect.__dict__
{ '_x' :  17,  '_y' :  14 }
>>> vect.setX ( 'A' )
>>> vect.__dict__
{ '_x' :  'A',  '_y' :  14 }
```

str au lieu de *int*

Attributs privés

Les vecteurs créés deviennent *immuables* vis à vis de l'extérieur de la classe.

- Il est nécessaires de passer par les setters pour les modifier
- L'accès au contenu d'un attribut se fait implicitement via les getters

  ➔ le code est plus clair, concis et modulable

Généricité

La généricité est une forme de polymorphisme qui permet de :

- écrire *des fonctions, des méthodes et des classes paramétriques* tout en maintenant les relations entre les éléments (arguments, valeurs de retour, ..)
- mieux définir et contrôler comment les types peuvent être mixés

➔ utilisation de *types génériques* au lieu de types concrets.

Généricité

En principe, Python n'a pas besoin de la généricité.

- Il n'est pas explicitement typé
- On n'a pas besoin d'une syntaxe spéciale pour faire de la généricité
- *Duck typing*

Généricité

Introduction d'une solution intégrée à partir de Python 3.5

- Utilisation d'un moteur de vérification de type statique
  ➔ ***mypy*** ou ***Pyre***
- Combinaison des bénéfices du typage dynamique (*duck typing*) et du typage statique

Généricité

Syntaxe

- **Typage d'une variable**

  nomVariable: type = *unType*

- **Types primitifs :** *int, str, …*

- **Types complexes du langage (List, Dict, …)** doivent être importés du module ***typing***

Généricité

Exemple (1) :

```
>>> def premier ( conteneur ):
        return conteneur[0]

>>> liste1 = [ 'a', 'b', 'c' ]
>>> premier ( liste1 )
'a'
>>> liste2 = [ 1, 2, 3 ]
>>> premier ( liste2 )
1
>>> liste3 = [ (2, 3), 'b', 3, 5 ]
>>> premier ( liste3 )
(2, 3)
```

Généricité

Exemple (2) :

```
>> from typing import Any, List
>>> def premier ( conteneur : List[Any] ) -> Any :
        return container [0]
>>> liste1 = List[ str ] = [ 'a', 'b', 'c' ]
>>> premier ( liste1 )
'a'
>>> liste2 = List[ int ] = [ 1, 2, 3 ]
>>> premier ( liste2 )
1
```

Généricité

Exemple (3) :

```
>>> from typing import List, TypeVar
>>> T = TypeVar( 'T' )
>>> def premier ( conteneur : List[ T ] ) → T :
        return container [0]
>>> liste1 = List[ str ] = [ 'a', 'b', 'c' ]
>>> premier ( liste1 )
'a'
>>> liste2 = List[ int ] = [ 1, 2, 3 ]
>>> premier ( liste2 )
1
```

Généricité

Exemple (4) :

```
>>> from typing import List, TypeVar
>>> T = TypeVar( 'T' )
>>> def premier ( conteneur : List[ T ] ) → T :
        return 'a'

>>> liste1 = List[ str ] = [ 'a', 'b', 'c' ]
>>> premier ( liste1 )
```

Incompatible return value Type (got ''str'', expected ''T'')

NB : Message d'erreur du vérificateur statique *mypy*

Généricité

Les types génériques peuvent représenter n'importe quel type.

On peut limiter les types pouvant être représentés par un type générique à une liste de types.

- **T** = **TypeVar**(''T'', **str**, **int**)
  - ➔ **T** peut représenter uniquement les types ***str*** et **int**
- T = TypeVar (''T'', bound = **int**)
  - ➔ **T** peut représenter uniquement le type ***int*** et son sous-type ***bool***

Généricité

Types utilisateurs

```
>>> from typing import TypeVar, Generic, List
>>> T = TypeVar ( 'T' )
>>> class Pile ( Generic [ T ] ):
        def __init__ ( self ) -> None :
            self.elements : List [ T ] = [ ]
        def push ( self, elem: T ) -> None :
            self.elements.append ( elem )
        def pop ( self ) -> T :
            return self.elements.pop ( )
        def empty ( self ) -> bool :
            return not self.elements
```

Type des paramètres et des retours spécifié

Généricité

```
>>> pileEntiers = Pile [ int ]( )
>>> pileEntiers.push ( 5 )
>>> pileEntiers.push ( 3 )
>>> elem = pileEntiers.pop ( )
>>> print(elem)
3
>>> pileEntiers.empty ( )
False
>>> elem = pileEntiers.pop ( )
>>> print(elem)
5
>>> pileEntiers.empty ( )
True
```

Création d'une pile d'entiers

```
>>> pileEntiers.push ( 'a' )
TypeError
```

Généricité

```
>>> pileChars = Pile [ str ]( )
>>> pileChars.push ( 'ABC' )
>>> pileChars.push ( 'BAC' )

>>> elem = pileChars.pop (  )
>>> print(elem)
BAC
>>> pileChars.empty ( )
False
>>> elem = pileChars.pop (  )
>>> print(elem)
ABC
>>> pileChars.empty ( )
True
```

Création d'une pile de chaînes de caractères

```
>>> pileChars.push ( 3.5 )
TypeError
```

Généricité

```
>>> pileReels = Pile [ float ]( )
>>> pileReels.push ( 0.5 )
>>> pileReels.push ( 89.6 )

>>> elem = pileReel.pop (  )
>>> print(elem)
89.6
>>> pileReels.empty ( )
False
>>> elem = pileReels.pop (  )
>>> print(elem)
0.5
>>> pileReels.empty ( )
True
```

Création d'une pile de nombres réels