## Comparaisons

- Par défaut, deux objets *Point* distincts seront toujours considérés comme différents (l'égalité par défaut compare les références...).
- En réalité, les opérateurs de comparaison (et les autres aussi, d'ailleurs) appellent des méthodes spéciales d'*object* que l'on peut redéfinir pour modifier leurs comportements.
- Dans le cas des opérateurs de comparaison, il s'agit des méthodes *__eq__*, *__ne__*, *__lt__*, *__le__*, *__gt__* et *__ge__*.
- Si l'on redéfinit *__eq__*, Python redéfinit automatiquement *__ne__* si on ne le fait pas (mais autant le faire explicitement).
- Pour les comparaisons, une bonne pratique consiste à redéfinir tous les opérateurs afin qu'il n'y ait pas d'incohérences. Afin de faciliter, cette tâche, on dispose de *functools.total_ordering* qui les génère tous à partir de *__eq__* et de l'un des opérateurs *__lt__*, *__le__*, *__gt__* et *__ge__*.
- L'instruction *obj1 == obj2* appelle en réalité *obj1.__eq__(obj2)* (idem pour les autres fonctions de comparaison).

# Exemple

```
import math
from functools import total_ordering

@total_ordering
class Point:
  def __init__(self, x, y):
    self.__x, self.__y = x, y

  def __eq__(self, autre):
    if not isinstance(autre, Point):
      return NotImplemented           # Python essaiera de trouver autre.__eq__(self)
    else:
      return self.__x == autre.x and self.__y == autre.y

  def __lt__(self, autre):
    if not isinstance(autre, Point):
      return NotImplemented           # Python essaiera de trouver autre.__lt__(self)
    else:
      return self.distance < autre.distance

  @property
  def x(self): return self.__x
  @x.setter
  def x(self, new_x): self.__x = new_x

  @property
  def y(self): return self.__y
  @y.setter
  def y(self, new_y): self.__y = new_y

  @property
  def distance(self): return math.hypot(self.__x, self.__y)
```

# Exemple (suite)

```
@total_ordering
class PointNomme(Point):
  """Classe définissant un point étiqueté"""

  def __init__(self, x, y, nom):
     Point.__init__(self, x, y)         # Ou super().__init__(x, y)
     self.__nom = nom

  def __eq__(self, autre):
    return Point.__eq__(self, autre) and self.__nom == autre.nom
    # ou return super().__eq__(autre) and self.__nom == autre.nom

  def __lt__(self, autre):
    return Point.__lt__(self, autre) and self.__nom < autre.nom
    # ou return super().__lt__(autre) or self.__nom < autre.nom

  @property
  def nom(self): return self.__nom
```

Tests :

```
p = point3.Point(3, 2)
q = point3.Point(3, 2)
r = point3.Point(4, 6)
p == q                          # True
p != r                          # True
p < r                           # True
p > r                           # False
p <= q                          # True
```

## La méthode spéciale ___hash___

- On rappelle que pour pouvoir servir de clé, un objet doit être immutable et disposer d'une méthode *___hash___* (qui est appelée automatiquement par la fonction *hash(obj)*).
- Les objets *Point* ne sont pas immutables (pour qu'ils le soient, il suffit de supprimer les deux propriétés setters qui permettent de modifier les attributs du point). Voir l'exemple suivant.
- Si l'on redéfinit la méthode spéciale *___eq___*, Python ne fournit plus de méthode *___hash___* par défaut : les objets de notre classe ne peuvent donc plus servir de clé de dictionnaire.
- La création d'une méthode de hachage est un problème épineux car il faut s'assurer qu'elle fournira une valeur différente pour chaque objet différent au sens de *___eq___*.
- Dans le cas d'un point, notamment, il faut que le hachage d'un *Point(3, 2)* soit différent du hachage d'un *Point(2, 3)* et que *tous* les *Point(3, 2)* renverront la même valeur de hachage.
- Pour écrire une nouvelle fonction de hachage, le plus simple (et le plus sûr) consiste à utiliser une fonction de hachage existante.

# Exemple

```
# Module point5.py

import math
from functools import total_ordering

@total_ordering
class Point:
  """Classe définissant un point du plan"""
  (...)
  @property
  def x(self): return self.__x      # On a supprimé le setter

  @property
  def y(self): return self.__y      # On a supprimé le setter

  @property
  def distance(self): return math.hypot(self.__x, self.__y)

  def __hash__(self):
   return hash((self.__x, self.__y))  # le hachage du tuple (x,y) sera différent de celui de (y, x)

@total_ordering
class PointNomme(Point):
  """Classe définissant un point étiqueté"""
  (...)
  @property
  def nom(self): return self.__nom

  def __hash__(self):
    return hash((self.__x, self.__y, self.__nom))
```

## Exemple

La fonction prédéfinie *hash(obj)* appelle la méthode *obj.__hash__()* :

```
from point5 import Point

p = Point(3, 2)
q = Point(3, 2)
r = Point(2, 3)

id(p)                       # 139751287446864
id(q)                       # 139751287446672 (donc différent alors que p == q)
hash(p)                     # 3713083796995235906
hash(q)                     # 3713083796995235906 (donc égal)
hash(r)                     # 3713082714463740756

d = {p: "toto", r: "titi"}  # {Point(3, 2): 'toto', Point(2, 3): 'titi'}
d[q] = "normal"
d                           # {Point(3, 2): 'normal', Point(2, 3): 'titi'}
p = Point(1, 1)
d                           # {Point(3, 2): 'normal...', Point(2, 3): 'titi'}
d[p] = "autre"
d                           # {Point(1, 2): 'autre', Point(3, 2): 'normal...', Point(2, 3): 'titi'}
```

**Remarque :**
La fonction *id(obj)* renvoie l'adresse mémoire de l'objet concerné. C'est donc un moyen d'identifier de façon unique deux objets différents mais elle considérera aussi comme différents deux points qui ont pourtant les mêmes coordonnées et qui, du point de vue d'un hachage, devraient être considérés comme égaux.

## Création d'une classe collection

- Le but, ici, est de créer une classe « collection » qui se comporte comme les classes collections prédéfinies (*list*, *dict*, *tuple*)...
- On veut notamment pouvoir accéder en lecture (et en écriture, si notre classe n'est pas immutable) à un élément en utilisant la notation des crochets et on veut pouvoir parcourir tous les éléments de notre collection à l'aide d'une boucle *for*.
- Les méthodes spéciales *__getitem__*, *__setitem__* et *__delitem__* permettent de bénéficier de la notation entre crochets.
- La méthode *__contains__* permettent d'utiliser les opérateurs *in* et *not in* pour tester la présence d'un élément dans une collection.
- Les méthodes spéciales *__add__*, *__mul__* et *__len__* permettent d'implémenter, respectivement, la concaténation, la multiplication et la longueur d'une collection.
- La méthode spéciale *__iter__* permet de parcourir les éléments d'une collection. Elle est appelée automatiquement par *for*.

# Premier exemple : un type ensemble

```
class Ensemble:
  """ Implémentation d'un ensemble d'entiers à l'aide d'un hachage
      dont les clés sont les entiers et les valeurs des booléens """

  def __init__(self, *liste):
     self.__corps = {}
     for e in liste: self.__corps[e] = True

  def contient(self, elt):
    return self.__corps.get(elt, False)   # ou : return elt in self.__corps.keys()

  def ajoute(self, elt):
    self.__corps[elt] = True

  def supprime(self, elt):
    if self.__corps.get(elt): del self.__corps[elt]

  def __str__(self):
    s = ""
    for e in self.__corps.keys(): s += str(e) + ", "
    return s[:-2]

  def union(self, autre):
    if not isinstance(autre, Ensemble): return self
    res = Ensemble()
    for e in self.__corps.keys(): res.ajoute(e)
    for e in autre.__corps.keys(): res.ajoute(e)
    return res

  (...)
```

# Premier exemple : un type ensemble

```
  def intersect(self, autre):
    if not isinstance(autre, Ensemble): return None
    res = Ensemble()
    for e in self.__corps.keys():
       if e in autre.__corps.keys():
         res.ajoute(e)
    return res

  def diff(self, autre):
    if not isinstance(autre, Ensemble): return self
    res = Ensemble()
    for e in self.__corps.keys():
       if e not in autre.__corps.keys():
         res.ajoute(e)
    return res

if __name__ == '__main__':
  ens = Ensemble(12, 2, 1, 3)
  print(ens)                          # 1, 2, 3, 12
  ens.ajoute(30)
  print(ens)                          # 1, 2, 3, 12, 30
  ens.supprime(100)
  ens.supprime(30)
  print(ens)                          # 1, 2, 3, 12
  assert ens.contient(3)
  assert not ens.contient(42)
  ens2 = Ensemble(3, 2, 100, 34)
  print(ens2)                         # 2, 3, 100, 34
  print(ens.union(ens2))              # 1, 2, 3, 100, 12, 34
  print(ens.intersect(ens2))          # 2, 3
  print(ens.diff(ens2))               # 1, 12
```

## Premier exemple : un type ensemble

Ça « marche », mais c'est assez « Java-esque »... Voici une nouvelle version, plus « pythonique » :

```
class Ensemble:

  def __init__(self, *liste):
     self.__corps = {}
     for e in liste: self.__corps[e] = True

  def __iadd__(self, elt):
     self.__corps[elt] = True
     return self

  def __str__(self):
    s = ""
    for e in sorted(self.__corps.keys()): s += str(e) + ", "
    return s[:-2]

  def __or__(self, autre):
    if not isinstance(autre, Ensemble): return self
    res = Ensemble()
    for e in self: res += e   # Utilisation de __contains__ et de __iadd__
    for e in autre: res += e  # idem
    return res

(...)
```

# Premier exemple : un type ensemble

```
def __and__(self, autre):
  if not isinstance(autre, Ensemble): return None
  res = Ensemble()
  for e in self:            # Utilisation de __contains__
     if e in autre:         # Utilisation de __contains__
       res += e             # Utilisation de __iadd__
  return res

def __sub__(self, autre):
  if not isinstance(autre, Ensemble): return self
  res = Ensemble()
  for e in self:
     if e not in autre:
       res += e
  return res

def __iter__(self):
  for e in sorted(self.__corps.keys()): yield e

def __getitem__(self, e):
  return self.__corps.get(e, False)

def __contains__(self, e):
  return self[e]

def __delitem__(self, e):
  self.__corps.pop(e, None)

def __len__(self):
  return len(self.__corps)
```

# Premier exemple : un type ensemble

Et l'on peut maintenant écrire :

```
if __name__ == '__main__':
  ens = Ensemble(12, 2, 1, 3)
  print(ens)                                  # 1, 2, 3, 12
  ens += 30
  print(ens)                                  # 1, 2, 3, 12, 30
  del ens[100]
  del ens[30]
  print(ens)                                  # 1, 2, 3, 12
  assert ens[3]
  assert (3 in ens)
  assert not ens[42]
  assert (42 not in ens)
  ens2 = Ensemble(3, 2, 100, 34)
  print(ens2)                                 # 2, 3, 34, 100
  print(ens | ens2)                           # 1, 2, 3, 12, 34, 100
  print(ens & ens2)                           # 2, 3
  print(ens - ens2)                           # 1, 12
  for e in ens: print(e, end=" ")             # 1 2 3 12
  print()
```

## Le module abc

- Le module *abc* (*Abstract Base Classes*) permet de définir des classes abstraites.
- Ce module fournit notamment la méta-classe *ABCMeta* qui permet de définir des classes abstraites.
- Il fournit également le décorateur *abstractmethod* pour définir des méthodes abstraites dans une classe dont la méta-classe est *ABCMeta* : la classe ne pourra être instanciée que si elle redéfinit toutes ses méthodes abstraites.
- Ce décorateur permet également de définir des méthodes de classes abstraites (on le combine avec *classmethod*) et des propriétés abstraites (on le combine avec *property*).
- Attention, en Python, le terme de *classe abstraite* n'est donc pas le même qu'en C++/Java/C# puisqu'une classe abstraite peut très bien ne pas avoir de méthode abstraites... (voir le premier exemple).

## Le module abc

- Python fournit également les modules *numbers* et *collections* qui fournissent des classes abstraites pour les nombres et les collections (une classe qui hérite de *collections.Sequence* doit implémenter les méthodes *__getitem__* et *__len__*, par exemple, mais disposera automatiquement d'autres méthodes "mixins").

# Exemple d'utilisation du module abc

```
from abc import ABCMeta, abstractmethod

class Figure(metaclass=ABCMeta):
  def __init__(self, nom):
    self._nom = nom

class FigureFermee(Figure):
  def __init__(self, nom):
    Figure.__init__(self, nom)

  @abstractmethod
  def surface(self):
    ...

class Rectangle(FigureFermee):
  def __init__(self, nom, coords, largeur, hauteur):
    FigureFermee.__init__(self, nom)
    self._coin_inf_gauche = coords
    self._largeur, self._hauteur = largeur, hauteur

  def surface(self):
    return self._largeur * self._hauteur

fig = Figure("abstraite")                          # Ok car pas de abstractmethod
fig_fermee = FigureFermee("toujours abstraite")    # TypeError: Can't instantiate abstract
                                                   # class FigureFermee with abstract methods surface
rect = Rectangle("carré", (1, 1), 10, 10)
rect.surface()                                     # 100
```

# Exemple amélioré

```
class Figure(metaclass=ABCMeta):
  @abstractmethod
  def __init__(self, nom):
    self._nom = nom

  @property
  def nom(self):
    return self._nom

class FigureFermee(Figure):
  def __init__(self, nom):
    Figure.__init__(self, nom)

  @property
  @abstractmethod
  def surface(self):            # Toute figure fermée a une surface...
    ...

class Rectangle(FigureFermee):
  def __init__(self, nom, coords, largeur, hauteur):
    FigureFermee.__init__(self, nom)
    self._coin_inf_gauche = coords
    self._largeur, self._hauteur = largeur, hauteur

  @property
  def surface(self):          # Redéfinition de la propriété abstraite
    return self._largeur * self._hauteur

fig = Figure("abstraite")                          # TypeError: Can't instantiate abstract class...
rect = Rectangle("carré", (1, 1), 10, 10)
rect.surface                                       # 100
rect.nom                                           # 'carré'
```