Comparaisons d'objets

La différence entre la notion ***d'objets identiques*** et celle **d'objets égaux** est cruciale pour comprendre la différence entre les opérateurs ***is*** et **==** de Python.

- Une expression == renvoie Vrai si les objets sont égaux (même contenu)
- Une expression ***is*** renvoie Vrai si les objets ont la même identité

Exemple :

```
>>> obj1 = [1, 2, 3]

>>> obj2 = obj1
```

Comparaisons d'objets

obj1 et obj2 pointent vers deux listes d'apparences identiques

```
>>> obj1
[1, 2, 3]

>>> obj2
[1, 2, 3]

>>> obj1 == obj2
True
```

Et obj1 et obj2 pointent vers le même objet

```
>>> obj1 is obj2
True
```

Comparaisons d'objets

Créons une copie identique de notre liste obj1

```
>>> c = list (obj1)
>>> c
[1, 2, 3]

>>> c == obj1
True                    c et obj1 ont le même contenu (égaux)

>>> obj1 is c
False                   c et obj1 ne pointent pas vers le même objet
                        (non identiques)
```

Conversion en String

```
>>>  class Voiture ( object ):
        def  __init__ ( self , couleur, kilometrage) :
            self.couleur = couleur
            self.kilometrage = kilometrage

>>> maVoiture = Voiture( 'rouge', 30000)

>>> maVoiture
<__console.Voiture__  object  at  0x1…>

>>> print ( maVoiture )
<__console.Voiture__  object  at  0x1…>
```

Conversion en String

```
>>> print ( maVoiture.couleur,  maVoiture.kilometrage )
rouge  30000
```

Conversion en String

```
>>> print ( maVoiture.couleur,  maVoiture.kilometrage )
rouge  30000
```

Dans Python toute classe nécessite d'inclure les méthodes __str__ et __repr__ , en particulier cette dernière.

Conversion en String

```
>>> print ( maVoiture.couleur,  maVoiture.kilometrage )
rouge  30000
```

Dans Python toute classe nécessite d'inclure les méthodes __str__ et __repr__ , en particulier cette dernière.

```
>>> class Voiture ( object ):
        def  __init__ ( self , couleur, kilometrage) :
            self.couleur = couleur
            self.kilometrage = kilometrage

        def  __str__ ( self ) :
            return f 'une  voiture  {self.couleur}'
```

Conversion en String

```
>>> maVoiture = Voiture ( 'rouge' , 30000 )
>>> maVoiture
<__console.Voiture__  object  at  0x1…>
```

La méthode __str__ est appelée lorsqu'on essaie de convertir un objet en une chaîne de caractères :

```
>>> print ( maVoiture )
'une voiture rouge'
```

```
>>> str( maVoiture )
'une voiture rouge'
```

```
>>> '{ }'. format ( maVoiture )
'une voiture rouge'
```

Conversion en String

Une autre méthode contrôle comment les objets sont convertis en String dans Python 3 : ***__repr__.*** Elle est utilisée dans des situations différentes.

```
>>> class Voiture ( object ):
        def __init__ ( self , couleur, kilometrage) :
            self.couleur = couleur
            self.kilometrage = kilometrage

        def __repr__ ( self ) :
            return '__repr__ pour Voiture'

        def __str__ ( self ) :
            return '__str__ pour Voiture '
```

Conversion en String

```
>>> maVoiture = Voiture ( 'rouge' , 30000 )
>>> print ( maVoiture )
'__str__ pour Voiture '

>>> '{ }'. format ( maVoiture )
'__str__ pour Voiture '

>>> maVoiture
'__repr__ pour Voiture '
```

=> Inspecter un objet en utilisant l'interpréteur Python affiche le résultat de la méthode *__repr__* de l'objet.

Conversion en String

Les conteneurs comme *lists* et *dicts* utilisent toujours le résultat de la méthode ***__repr__*** pour représenter les objets qu'ils contiennent.

```
>>> str ( [ maVoiture ] )
'[ __repr__  pour  Voiture ] '
```

Toujours préférer utiliser directement les fonctions **str( )** et **repr( )** au lieu d'appeler les méthodes **__str__** et **__repr__** des objets.

```
>>> str ( maVoiture )
' __str__  pour  Voiture '
```

```
>>> repr ( maVoiture )
' __repr__  pour  Voiture '
```

Conversion en String

Pourquoi cette différence entre les méthodes **__str__** et **__repr__** ?

```
>>> import  datetime
>>> today = datetime.date.today ( )

>>> str ( today )
'2023 - 03 - 22'
```

```
>>> repr( today )
' datetime.date ( 2023,  3,  22 ) '
```

Conversion en String

- **__str__ doit fournir :**
  - Une information principalement lisible.
  - Une représentation textuelle concise pour l'utilisateur

- **__repr__ doit fournir :**
  - Un résultat non ambiguë (c'est plus important) et plus élaboré (d'où les noms du module et de la classe)
  - Une aide précieuse pour le développeur (debugging)

Conversion en String

Remarque :

Les méthodes commençant et se terminant par des underscores (« dunder » methods) sont parfois appelées «méthodes magiques », mais elles ne sont pas supposées être magiques.

- **Ce n'est qu'une convention** pour les identifier comme des caractéristiques de base de Python.

- **Cela évite les collisions de noms** avec les méthodes et attributs utilisateurs.

Conversion en String

Pourquoi chaque classe nécessite une __repr__ ?

Si une classe ne contient pas de méthode **__str__**, Python recherche automatiquement la méthode **__repr__**.

Toujours inclure une méthode __repr__ !

Exemple (1):

```
>>> class Voiture ( object ):
        def __init__ ( self , couleur, kilometrage) :
            self.couleur = couleur
            self.kilometrage = kilometrage

>>> def __repr__ ( self ) :
            return f' Voiture( { self.couleur ! r }, { self.kilometrage ! r } )'
```

Conversion en String

Exemple (2): ou pour ne pas utiliser explicitement le nom de la classe

```
>>> class Voiture ( object ):
        def __init__ ( self , couleur, kilometrage) :
            self.couleur = couleur
            self.kilometrage = kilometrage

>>> def __repr__ ( self ) :
            return ( f' { self.__class__.name__} (
                    f' {  self.couleur ! r }, { self.kilometrage ! r } )' )
```

Conversion en String

La bonne approche :

- **On peut contrôler la conversion en String** dans les classes users en utilisant les méthodes ***__str__*** et ***__repr__***.
- Le résultat de ***__str__*** doit être **lisible,** le résultat de ***__repr__*** doit être **non ambiguë**.
- Utiliser la méthode ***__unicode__*** à la place de ***__str__*** dans Python 2 : ***__str__*** retourne des *bytes* et ***__unicode__*** retourne des *caractères*.

Conversion en String

Exemple : Python 2.x

```
>>> class Voiture ( object ):
        def __init__ ( self , couleur, kilometrage) :
            self.couleur = couleur
            self.kilometrage = kilometrage

>>> def __repr__ ( self ) :
            return ( ' { } ( { !r }, { !r } )' . Format (
                self.__class__.__name__, self.couleur, self.kilometrage )
```

Conversion en String

Exemple : Python 2.x (suite)

```
def __unicode__ ( self ) :
    return u 'a {self.couleur } Voiture' . Format ( self = self )

def __str__ ( self ) :
    return unicode ( self ) .encode ( 'utf-8' )
```

Clonage des objets

L'affectation *ne crée pas de copie* d'un objet, elle lie juste le nom à l'objet

- **Pour les objets immuables** cela ne fait aucune différence.
- **Pour les objets mutables ou les collections d'objets**, on peut vouloir créer de « *réelles copies* » ou des « *clones* » de ces objets.
- Parfois, on souhaite des copies que l'on peut modifier, sans automatiquement modifier l'original en même temps.

Clonage des objets

Collections modifiables de Python

Appel des fonctions « d'usine » pour créer des copies de ces collections

```
>>> nouvelle_liste      = list ( liste_originale )
>>> nouveau_dico        = dict ( dico_original )
>>> nouvel_ensemble = set ( ensemble_original )
```

Pour ces collections, il y a une importante différence entre des copies *superficielles et* des copies *réelles.*

Clonage des objets

1) Copie superficielle

Création d'une nouvelle collection contenant des *références aux objets* de la collection originale → *un seul niveau de profondeur*

```
>>> ma_liste  =  [ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]
>>> ma_copie  = list( ma_liste )
```

ma_copie est un nouvel objet indépendant et de même contenu que *ma_liste.*

Clonage des objets

Vérification 1

```
>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                # Même contenu

>>> ma_liste.append ( [ 'nouvelle sous-liste' ] )

>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]
```

Clonage des objets

Vérification 1

```
>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                # Même contenu

>>> ma_liste.append ( [ 'nouvelle sous-liste' ] )

>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                # Copie indépendante
```

Clonage des objets

Vérification 2

```
>>> ma_liste [ 1 ][ 0 ] = 'X'
>>> ma_liste
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
```

Clonage des objets

Vérification 2

```
>>> ma_liste [ 1 ][ 0 ] = 'X'
>>> ma_liste
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
```

```
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ] ]          # Même modification
```

Clonage des objets

Vérification 2

```
>>> ma_liste [ 1 ][ 0 ] = 'X'
>>> ma_liste
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ] ]          # Même modification

>>> del ma_liste[ 0 ]

>>> ma_liste
[ ['X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]
```


Clonage des objets

Vérification 2

```
>>> ma_liste [ 1 ][ 0 ] = 'X'
>>> ma_liste
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ] ]          # Même modification

>>> del ma_liste[ 0 ]

>>> ma_liste
[ ['X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
```

Clonage des objets

Vérification 2

```
>>> ma_liste [ 1 ][ 0 ] = 'X'
>>> ma_liste
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ] ]          # Même modification

>>> del ma_liste[ 0 ]

>>> ma_liste
[ ['X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , ['X', 5, 6 ] , [ 7, 8, 9 ] ]          # Copie indépendante
```

Clonage des objets

Remarque :

- On peut également créer une copie superficielle en utilisant la fonction ***copy( )***.
- Cependant, l'utilisation des fonctions ***list( )***, ***dict( )*** et ***set( )*** pour créer des copies superficielles d'objets est considérée comme très «Pythonique ».

Clonage des objets

2) Copie réelle (*deep copy*)

Création d'une nouvelle collection, puis son remplissage récursivement de copies *des objets* de la collection originale.

➡ *processus récursif à travers tout l'arbre des objets*

```
>>> import copy
>>> ma_liste = [ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]
>>> ma_copie = copy.deepcopy( ma_liste )
```

L'objet *ma_copie* est un clone indépendant de l'objet *ma_liste.*

Clonage des objets

Vérification 1

```
>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]
>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                    # Même contenu
>>> ma_liste.append ( [ 'nouvelle sous-liste' ] )
>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]
>>> ma_copie
```

Clonage des objets

Vérification 1

```
>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                    # Même contenu

>>> ma_liste.append ( [ 'nouvelle sous-liste' ] )

>>> ma_liste
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                    # Copie indépendante
```

Clonage des objets

Vérification 2

```
>>> ma_liste [ 1 ][ 0 ] = 'X'
>>> ma_liste
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                    # Pas de modification

>>> del ma_liste[ 0 ]

>>> ma_liste
[ ['X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
```

Clonage des objets

Vérification 2

```
>>> ma_liste [ 1 ][ 0 ] = 'X'
>>> ma_liste
[ [ 1, 2, 3 ] , [ 'X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                    # Pas de modification

>>> del ma_liste[ 0 ]

>>> ma_liste
[ ['X', 5, 6 ] , [ 7, 8, 9 ], [ 'nouvelle sous-liste' ] ]

>>> ma_copie
[ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]                    # Clone indépendant
```

Clonage des objets

Les objets arbitraires

Comment créer des copies superficielles ou réelles d'objets arbitraires ?

```
>>> class Point(object):
        def __init__( self, x, y ):
            self.x = x
            self.y = y

        def __repr__( self ):
            return f'Point ( { self.x ! r }, { self.y ! r } )'
```

Clonage des objets

```
>>> pta  = Point ( 23,  42 )
>>> ptb  = copy.copy( pta )              # copie superficielle de pta

>>> pta
Point ( 23,  42 )

>>> ptb
Point ( 23, 42 )                         # Même contenu

>>> pta  is  ptb
False                                    # pta n'est pas ptb
```

NB : Comme les objets Point utilisent un **type immuable (int)** pour leurs coordonnées il n'y a *pas de différence entre une copie superficielle et une copie réelle.*

Clonage des objets

Exemple 2 :

```
>>> class Rectangle (object):

        def __init__( self, hautG, basD ):
            self.pointHG = hautG
            self.pointBD  = basD

        def __repr__( self ):
            return  ( f'Rectangle ( { self. pointHG ! r }, '
                      f'{ self. pointBD ! r } )'  )
```

Clonage des objets

```
>>> rect   = Rectangle ( Point ( 0, 1 ), Point ( 5, 6 ) )
>>> rectSup  = copy.copy( rect )          # copie superficielle de rect

>>> rect
Rectangle ( Point ( 0, 1 ), Point ( 5, 6 ) )     # retour de __repr__()

>>> rectSup
Rectangle ( Point ( 0, 1 ), Point ( 5, 6 ) )     # Même contenu

>>> rect  is  rectSup
False                                            # rect n'est pas rectSup
```

Clonage des objets

```
>>> rect.pointHG.x = 999
>>> rect
Rectangle ( Point ( 999, 1 ), Point ( 5, 6 ) )

>>> rectSup
Rectangle ( Point ( 999, 1 ), Point ( 5, 6 ) )        # Même modification

>>> rectRee = copy.deepcopy ( rectSup )
>>> rectRee.pointHG.x = 222
>>> rectRee
Rectangle ( Point ( 222, 1 ), Point ( 5, 6 ) )

>>> rect                                    >>> rectSup
```

Clonage des objets

```
>>> rect.pointHG.x = 999
>>> rect
Rectangle ( Point ( 999, 1 ), Point ( 5, 6 ) )

>>> rectSup
Rectangle ( Point ( 999, 1 ), Point ( 5, 6 ) )        # Même modification

>>> rectRee = copy.deepcopy ( rectSup )
>>> rectRee.pointHG.x = 222
>>> rectRee
Rectangle ( Point ( 222, 1 ), Point ( 5, 6 ) )

>>> rect                                              >>> rectSup
                Rectangle ( Point ( 999, 1 ), Point ( 5, 6 ) )
```

Clonage des objets

Conclusions :

- **Faire une copie superficielle** d'un objet *ne clone pas* ses objets fils.
  - ➡ La copie *n'est pas totalement indépendante* de l'originale.
- **Faire une copie profonde (réelle)** d'un objet *clone* ses objets fils.
  - ➡ La copie est *totalement indépendante* de l'originale, mais une telle copie prend du temps.
- On peut **faire une copie profonde** de n'importe quel type d'objet (classes personnalisées inclues).

Namedtuples

Les tuples de Python sont une simple structure de données (immuable) pour regrouper des objets arbitraires.

```
>>> tup = ( 'hello',  object( ),  42 )
>>> tup
( 'hello',  < object object at 0x105e76b70 >,  42 )

>>> tup [ 2 ]
42

>>> tup [ 2 ] = 23
TypeError:
'' 'tuple'  object does not support item assignement ''
```

Namedtuples

Inconvénients des tuples

- **Accès aux données en utilisant les indices (entiers).** On ne peut pas leur donner de noms.
- **Un tuple est toujours une structure ad-hoc.** Il est difficile de garantir que deux tuples ont le même nombre de champs et contiennent les mêmes données.

➡ Introduction des **namedtuples**

Namedtuples

Python fournit un type de conteneurs spécialisés : ***namedtuple.***

- **Namedtuple** peut être vu comme une extension du type de données *tuple*. C'est un conteneur immuable.
- Tous les attributs d'un **namedtuple** suivent le principe « write once, read many »
- On peut accéder à tout objet dans un **namedtuple** en utilisant un identifiant unique.
- Il peut constituer une bonne alternative pour définir manuellement une classe.

Namedtuples

Exemple :

```
>>> from collections import namedtuple
>>> Voiture = namedtuple ( 'Voiture' , ' couleur kilometrage' )
```

« typename »

2 champs

- Le « typename » Voiture est le nom d'une nouvelle classe créée par la fonction *namedtuple( )*

- Voiture est donc un type de données caractérisé par deux champs : *couleur*, *kilometrage*.

Namedtuples

Exemple (v2) :

```
>>> from collections import namedtuple

>>> ' couleur   kilometrage ' . split( )
[ ' couleur ' ,  ' kilometrage ' ]

>>> Voiture  =  namedtuple  ( 'Voiture'  , [ ' couleur ' ,  ' kilometrage ' ] )
```

Namedtuples

On peut maintenant créer des objets de la classe *Voiture()*

```
>>>  ma_voiture   = Voiture ( ' rouge ',  38132 )

>>>  ma_voiture. couleur                  # accès en utilisant le nom
'rouge'
>>>  ma_voiture. kilometrage             # accès en utilisant le nom
38132

>>>ma_voiture [ 0 ]                      # accès en utilisant l'indice
'rouge'
>>>  ma_voiture [ 1 ]                    # accès en utilisant l'indice
38132
```

Namedtuples

../..

```
>>> tuple ( ma_voiture )
( 'rouge' , 38132 )

>>> couleur,  kilometrage = ma_voiture      # dépaquetage
>>> print ( couleur,  kilometrage )
'rouge' ,  38132

>>> print ( *ma_voiture )                   # *-opérateur de dépaquetage
'rouge' ,  38132                            # pour les fonctions
>>> ma_voiture
Voiture ( couleur = 'rouge' ,  kilometrage =  38132 )
```

Namedtuples

- Comme les tuples, namedtuples sont immuables

```
>>> ma_voiture .couleur = ' bleue '
AttributeError: ''can't  set  attribute''
```

- **Les objets namedtuples** sont implémentés comme des classes Python régulières.
- En terme d'utilisation de la mémoire, les namedtuples sont ''meilleurs'' que les classes régulières, et tout aussi efficaces que les tuples réguliers.

Namedtuples

Dérivation des Namedtuples

Comme toute classe régulière dans Python, un namedtuple peut être étendu en rajoutant des méthodes et des propriétés.

Exemple :

```
>>> Voiture = namedtuple ( 'Voiture' , ' couleur   kilometrage ' )
>>> class MaVoitureAvecMethods ( Voiture )
        def hexacouleur ( self ):
            if self.couleur == ' rouge '
                return ' #ff0000 '
            else :
                return ' #0000 '
```

Namedtuples

On peut maintenant créer des objets MaVoitureAvecMethodes( )

```
>>> ma_VAM   = MaVoitureAvecMethodes ( ' rouge ',  1234 )
>>> ma_VAM. hexcouleur( )
'#ff0000'
```

Namedtuples

On peut maintenant créer des objets MaVoitureAvecMethodes( )

```
>>> ma_VAM   = MaVoitureAvecMethodes ( ‘ rouge ‘,  1234 )
>>> ma_VAM. hexcouleur( )
‘#ff0000’
```

A moins de vouloir une classe avec des propriétés immuables, cette approche est un peu maladroite.

Namedtuples

On peut maintenant créer des objets MaVoitureAvecMethodes( )

```
>>> ma_VAM   = MaVoitureAvecMethodes ( ‘ rouge ‘,  1234 )
>>> ma_VAM. hexcouleur( )
‘#ff0000’
```

A moins de vouloir une classe avec des propriétés immuables, cette approche est un peu maladroite.

Ajouter un champs immuable est difficile à cause de la manière dont les *namedtuples* sont structurés en interne.

Namedtuples

On peut maintenant créer des objets MaVoitureAvecMethodes( )

```
>>> ma_VAM   = MaVoitureAvecMethodes ( ‘ rouge ‘,  1234 )
>>> ma_VAM. hexcouleur( )
‘#ff0000’
```

A moins de vouloir une classe avec des propriétés immuables, cette approche est un peu maladroite.

Ajouter un champs immuable est difficile à cause de la manière dont les *namedtuples* sont structurés en interne.

➡ Utiliser la propriété *_fields* des tuples pour créer une hiérarchie de *namedtuples*

Namedtuples

Exemple : Ajout d’un champs immuable

```
>>> Voiture  =  namedtuple  ( ‘Voiture’  ,  ‘ couleur   kilometrage ‘ )
>>> VoitureElectrique = namedtuple (
                        ‘VoitureElectrique ‘ , Voiture._fields + ( ‘ charge ‘ ) )

>>> VoitureElectrique (  ‘ rouge ‘,  1234,  45.0 )
VoitureElectrique (couleur= ‘ rouge ‘, kilometrage=1234, charge=45.0 )
```

Namedtuples

Méthodes d'assistance intégrées

- Toute instance *namedtuple* fournit des méthodes d'assistance très pratiques pour le programmeur.
- Ces méthodes sont privées (*_nom*) et comme les propriétés privées, elles font partie de l'interface publique des *namedtuples*

Namedtuples

Exemple 1 : La méthode **_asdict( )**

```
>>> Voiture = namedtuple ( 'Voiture' , ' couleur  kilometrage ' )
>>> ma_voiture = Voiture( 'rouge' , 3812 )

>>> ma_voiture. _asdict ( )
OrderedDict( [ ( 'couleur',  ' rouge ' ), ( 'kilometrage' , 3812 ) ] )

>>> json. dumps ( ma_voiture. _asdict ( ) )
'{ ''couleur'':  ''rouge '' ,  ''kilometrage''  : 3812 }'
```

Namedtuples

Exemple 2 : La méthode **_replace( )**

```
>>> Voiture = namedtuple ( 'Voiture' , ' couleur  kilometrage ' )
>>> ma_voiture = Voiture( 'rouge' , 3812 )


>>> ma_voiture. _replace ( couleur = 'bleue' )
```

Voiture(couleur= 'bleue' , kilometrage= 3812)

Attention : _replace ( ) crée une copie *superficielle* du tuple et permet de remplacer certains de ses champs.

Namedtuples

Exemple 3 : La méthode de classe **_make( )**

```
>>> Voiture = namedtuple ( 'Voiture' , ' couleur  kilometrage ' )
>>> ma_voiture = Voiture( 'rouge' , 3812 )


>>> Voiture. _make ( [ 'rouge' , 999 ] )
```

Voiture(couleur= 'rouge' , kilometrage= 999)

Attention : _make ( ) est une méthode de classe qui crée de nouvelles instances d'un namedtuple à partir d'une *séquence* ou d'un *itérable*.

Namedtuples

Conclusions :

- **Les namedtuples sont un raccourci** pour définir manuellement des classes immuables en Python avec une certaine efficacité en terme de mémoire.
- **Les nametuples fournissent des méthodes d'assistance** dont le nom commence par un *underscore*, mais font partie de l'interface publique. On peut donc les utiliser.

Les Classes Exceptions

Définir ses propres types d'erreurs peut être un atout pour un programmeur Python.

- Cas d'erreurs clairement identifiés.
- *Meilleure maintenabilité des fonctions, des méthodes et des modules*

Exemple :

```
>>> def valider( nom ):
        if len ( nom ) < 10 :
            raise  ValueError          # classe générique d'exceptions
                                       # de ''haut niveau''
```

Les Classes Exceptions

```
>>> valider ( 'Joe' )
Traceback ( most recent call last ) :
  File ''< input >'', line 1, in < module >
      valider( 'Joe' )
  File ''< input >'', line 3, in valider
      raise ValueError
  ValueError
```

Message pas très utile pour une personne qui n'a pas codé la fonction.

Les Classes Exceptions

Solution : un type d'exception personnalisé

Exemple :

```
>>> class ErreurNomTropCourt ( ValueError ):
        pass

    def valider ( nom ):
     if len ( nom ) < 10 :
        raise  ErreurNomTropCourt ( nom )
```

Type d'exceptions ErreurNomTropCourt 'auto-documenté' qui étend la classe native ValueError

Les Classes Exceptions

```
>>> valider ( 'Jane' )
Traceback ( most recent call last ) :
  File ''< input >'', line 1, in < module >
        valider( 'Jane' )
  File ''< input >'', line 3, in valider
        raise ErreurNomTropCourt (nom)
  ErreurNomTropCourt : Jane
```

Message plus informatif, donc plus utile aux utilisateurs de la fonction.

Les Classes Exceptions

Hiérarchie d'exceptions personnalisées pour un module/package

Exemple :

```
>>> class ErreurValidationDeBase ( ValueError ):
        pass

    class ErreurNomTropCourt ( ErreurValidationDeBase ):
        pass

    class  ErreurNomTropLong ( ErreurValidationDeBase ):
        pass
```

Les Classes Exceptions

../..

```
>>> def valider ( nom ):
        if len ( nom ) < 10 :
            raise ErreurNomTropCourt ( nom )
        elif len ( nom ) > 10 :
            raise ErreurNomTropLong ( nom )

>>> try:
        valider ( nom ):
    except ErreurValidationDeBase as err :
        handle_validation_error ( err )
```

Les Classes Exceptions

Remarques :

- On peut grouper logiquement les exceptions en sous hiérarchies en utilisant l'héritage.

  Attention à ne pas introduire une complexité inutile.

- Définir des classes d'exceptions personnalisées facilite aux utilisateurs de vos modules l'adoption du style de codage *EAFP* (*Easier to Ask for Forgiveness than Permission*) considéré comme très 'Pythonique'