Cours_Python_SRI_advanced

September 25, 2020

1 Concepts avancés en python

Nous allons maintenant voir quelques facilités fournies par le langage python

- introspection des objets
- itérateurs
- générateurs
- gestion des classes et objets : classes abstraites, attributs/méthodes de classes
- décorateurs
- héritage multiple

1.1 Introspection

En python on peut avoir accès à certaines définitions internes des objets que l'on manipule.

Par exemple:

[1]:
```
class Test:
    """une classe de test"""
    def __init__(self,x,y):
        self.one = x
        self.two = y

    def add(self,z):
        """l'addition"""
        self.one += z
        self.rwo +=z

# n'est pas éxécuté si le script est importé par un autre script
if __name__=="__main__":
    a = Test(1,2)
    print(a.__dict__)
    print(a.__module__)
    print(a.__doc__)
    print(a.__class__)
    help(a)
```

```
{'two': 2, 'one': 1}
__main__
une classe de test
<class '__main__.Test'>
Help on Test in module __main__ object:

class Test(builtins.object)
 |  une classe de test
 |
 |  Methods defined here:
 |
 |  __init__(self, x, y)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  add(self, z)
 |      l'addition
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

[2]:
```
print(__name__)
```

```
__main__
```

[3]:
```
print(dir(a))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'add', 'one', 'two']
```

1.2 Itérateurs

On a vu que l'on pouvait itérer directement sur des contenants comme des listes

```
for x in [1,3,5]:
    print(x)
```

Supposons que l'on définisse un objet qui contient lui aussi un ensemble de valeurs quelconque

[4]:
```
class Equipe:
    def __init__(self,nom,membres):
        self.nom = nom
        self.membres = membres

team = Equipe("Preum",["Adam","Eve"])
# on  peut itérer sur les membres
for x in team.membres:
    print(x)
```

```
Adam
Eve
```

mais on voudrait directement itérer sur l'objet, qui est une collection de membres le seul champ sur lequel on veut pouvoir itérer

[5]:
```
for x in team:
    print(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-5-e81d53a88246> in <module>()
----> 1 for x in team:
      2     print(x)


TypeError: 'Equipe' object is not iterable
```

[1]:
```
class Equipe:

    def __init__(self,nom,membres):
        self.nom = nom
        self.membres = membres

    # on peut définir un itérateur sur la classe.
    def __iter__(self):
        return iter(self.membres)

team = Equipe("Preum",["Adam","Eve"])
for x in team:
    print(x)
```

```
Adam
Eve
```

Mais comment est défini l'itérateur sur la liste ?

Il suffit que l'objet ait une méthode *_next_*(), qui est déjà défini sur les listes.

*_next_*() doit soulever une exception StopIteration quand il est censé s'arrêter (pas obligatoire...).

On peut le définir soi-même sur un objet:

Exercice: définir un itérateur qui parcours une liste à l'envers.

[13]:
```
class Reverse:
    """itérateur pour parcourir un conteneur à l'envers"""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

    def reset(self):
        self.index = len(self.data)


for x in Reverse([1,2,3]):
    print(x,end=" ")


a = Reverse([1,2,3])
for x in Reverse(range(10)):
    print(x,end=" ")
```

```
3 2 1 9 8 7 6 5 4 3 2 1 0
```

[2]:
```
a = iter([1,2,3])
print(next(a))
print(a.__next__())
print(a.__next__())
print(a.__next__())
print(a.__next__())
```

```
1
2
3
```

```
---------------------------------------------------------------------------

StopIteration                             Traceback (most recent call last)

<ipython-input-2-2719eafce35d> in <module>
      3 print(a.__next__())
      4 print(a.__next__())
----> 5 print(a.__next__())
      6 print(a.__next__())


StopIteration:
```

[ ]:

2 Générateurs

un générateur est un moyen plus simple de définir des itérateurs.

le mot clef “yield” dans une fonction suffit à la définir la fonction comme générateur

[3]:
```
def natural(n):
    i = 0
    while i<n:
        yield i
        i = i + 1

#gen = natural(3)
for x in natural(3):
    print(x,end=" ")
```

```
0 1 2
```

Un générateur permet de générer au fur et à mesure des besoins des valeurs à énumérer, sans tout stocker explicitement.

La syntaxe cache en fait les choses suivantes:

- la définition de l’initialisation de l’itérateur
- la définition de ce qu’il renvoie à chaque étape (next)
- l’exception quand on sort de ce qu’il doit énumérer

La fin de l’itération est optionnelle !

[10]:
```
def natural0():
    i = 1
```

```
    while True:
        yield i
        i = i + 1

gen = natural0()
print(next(gen), gen.__next__(), gen.__next__(), gen.__next__())
```

```
1 2 3 4 5
```

Un générateur définit implicitement un itérateur, à condition de s'arrêter.

[11]:
```
for i in natural(100):
    print(i,end=" ")
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

En fait, natural fait la même chose que la fonction range

On peut définir un parcours d'un conteneur "à l'avance"

Exemple :

[18]:
```
from sys import getsizeof
# là on crée une liste à chaque fois
a1 = [2*x for x in range(10000)]
print(getsizeof(a1))
a1 = [x-3 for x in a1]
print(getsizeof(a1))

l = [x for x in range(100) if x>50]
```

```
87624
87624
```

[13]:
```
# ou bien ...
a2 = (2*x for x  in range(10000))
print(a2)
print(getsizeof(a2))
a2 = (x-3 for   x in a2)
print(a2)
print(getsizeof(a2))

# plus simple
a2 = (y-3 for y in (2*x for x in range(100)))

for x in a2:
    print(x,end=" ")
```

```
a2.__next__()
```

```
<generator object <genexpr> at 0x7f177cf090a0>
88
<generator object <genexpr> at 0x7f177cf09360>
88
-3 -1 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99 101 103
105 107 109 111 113 115 117 119 121 123 125 127 129 131 133 135 137 139 141 143
145 147 149 151 153 155 157 159 161 163 165 167 169 171 173 175 177 179 181 183
185 187 189 191 193 195
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)

<ipython-input-13-a5f8f2ac3a07> in <module>()
 13     print(x,end=" ")
 14
---> 15 a2.__next__()


StopIteration:
```

[4]:
```
print(list(a2)[:100])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-4-88ac85517cab> in <module>
----> 1 print(list(a2)[:100])


NameError: name 'a2' is not defined
```

[7]:
```
for (i,x) in enumerate(["a","d","e"]):
    print(i,x)
```

```
0 a
1 d
2 e
```

[10]:
```
[x+10 for x in range(10) if x % 2==0]
```

[10]:
```
[10, 12, 14, 16, 18]
```

[51]:
```
# On peut appliquer les méthodes de générateur
from random import randint
maliste = (randint(-1000,1000) for i in range(10000))

print(next(i for (i,x) in enumerate(maliste) if x==0))

# 3 générateurs et complexité espace en O(1) !

# question: qu'a renvoyé cette expression ? --> l'indice du premier zero
```

```
1890
```

[52]:
```
# attention quand même :
print(sum(next(i for (i,x) in enumerate(maliste) if x==0) for j in range(1000))/
 ↪1000.)
```

```
5.827
```

```
/anaconda3/envs/snorkel/lib/python3.6/site-packages/ipykernel_launcher.py:2:
DeprecationWarning: generator '<genexpr>' raised StopIteration
```

Revenons à notre classe ; on aurait pu écrire l'itérateur nous mêmes:

[31]:
```
class Equipe:
    def __init__(self,nom,membres):
        self.nom = nom
        self. membres = membres

    def __iter__(self):
        for x in self.membres:
            yield x

team = Equipe("Preum",["Adam","Eve"])
for x in team:
    print(x)

# on peut gérer nous mêmes la fin de l'itération
a = iter(team)
next(a),next(a)
try:
    next(a)
except StopIteration:
    print("ayé")
```

```
Adam
Eve
ayé
```

A quoi ça sert ? à contrôler plus finement ce qui se passe

[Aussi et surtout: économie de mémoire]

[14]:
```python
class Equipe:
    def __init__(self,nom,membres):
        self.nom = nom
        self.membres = membres
        self.bannis = set()

    def __iter__(self):
        for x in self.membres:
            if x not in self.bannis:
                yield x

team = Equipe("Preum",["Adam","Eve","Cain","Abel"])
team.bannis.add("Cain")

for x in team:
    print(x)
```

```
Adam
Eve
Abel
```

[ ]:
```python
```

[ ]:
```python
```

Exercice:

Analyse d'ADN

- codon = 3 'lettres, fin seulement {TAA, TAG, TGA}; début TTG ATG GTG
- 'code génétique': acide aminé codé par chaque codon

Faire un générateur qui analyse l'ADN codon par codon, en commençant au premier début valide et en s'arrêtant à la première fin.

Faire un générateur qui envoie toutes les acides aminés codées sur un brin (bonus: il peut y avoir plusieurs début/fin)

[ ]:
```python
# codon -> acide aminé
code = {'TCA':'S', # Sérine
 'TCC':'S', # Sérine
```

```
 'TCG':'S', # Sérine
 'TCT':'S', # Sérine
 'TTC':'F', # Phénylalanine
 'TTT':'F', # Phénylalanine
 'TTA':'L', # Leucine
 'TTG':'L', # Leucine
 'TAC':'Y', # Tyrosine
 'TAT':'Y', # Tyrosine
 'TAA':'*', # Codon Stop
 'TAG':'*', # Codon Stop
 'TGC':'C', # Cystéine
 'TGT':'C', # Cystéine
 'TGA':'*', # Codon Stop
 'TGG':'W', # Tryptophane
 'CTA':'L', # Leucine
 'CTC':'L', # Leucine
 'CTG':'L', # Leucine
 'CTT':'L', # Leucine
 'CCA':'P', # Proline
 'CCC':'P', # Proline
 'CCG':'P', # Proline
 'CCT':'P', # Proline
 'CAC':'H', # Histidine
 'CAT':'H', # Histidine
 'CAA':'Q', # Glutamine
 'CAG':'Q', # Glutamine
 'CGA':'R', # Arginine
 'CGC':'R', # Arginine
 'CGG':'R', # Arginine
 'CGT':'R', # Arginine
 'ATA':'I', # Isoleucine
 'ATC':'I', # Isoleucine
 'ATT':'I', # Isoleucine
 'ATG':'M', # Méthionine
 'ACA':'T', # Thréonine
 'ACC':'T', # Thréonine
 'ACG':'T', # Thréonine
 'ACT':'T', # Thréonine
 'AAC':'N', # Asparagine
 'AAT':'N', # Asparagine
 'AAA':'K', # Lysine
 'AAG':'K', # Lysine
 'AGC':'S', # Sérine
 'AGT':'S', # Sérine
 'AGA':'R', # Arginine
 'AGG':'R', # Arginine
 'GTA':'V', # Valine
```

```
  'GTC':'V', # Valine
  'GTG':'V', # Valine
  'GTT':'V', # Valine
  'GCA':'A', # Alanine
  'GCC':'A', # Alanine
  'GCG':'A', # Alanine
  'GCT':'A', # Alanine
  'GAC':'D', # Acide Aspartique
  'GAT':'D', # Acide Aspartique
  'GAA':'E', # Acide Glutamique
  'GAG':'E', # Acide Glutamique
  'GGA':'G', # Glycine
  'GGC':'G', # Glycine
  'GGG':'G', # Glycine
  'GGT':'G', # Glycine
  }

adn = "GGTGGGGTTGCAGCCCTGAAGCTTACACCCTGA"

debut = {"TTG","GTG","ATG"}

["G","C","..."]
```

[17]:
```
def sequencage(adn):
    # trouver le début
    i = 0
    while adn[i:i+3] not in debut:#
        i = i + 1
    # cracher les codons
    i = i + 3
    while i <= len(adn)-3:
        codon = adn[i:i+3]
        i = i + 3
        yield codon


#for codon in sequencage(adn):
#    print(codon)


def decodage(adn,dico_code):
    for codon in sequencage(adn):
        proteine = dico_code[codon]
        if proteine != "*":
            yield proteine
        else:
            # ou bien return None
```

```
            raise StopIteration


for proteine in decodage(adn,code):
    print(proteine)
```

```
G
L
Q
P
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:98:
DeprecationWarning: generator 'decodage' raised StopIteration
```

2.1 Retour sur les définitions de classes

- attributs de classes
- classes abstraite

on peut avoir des attributs de classes, partagés par toutes les instances

[16]:
```
class Bidon:
    # attribut de classe
    general = [1,2,3]

    def __init__(self,x):
        self.particulier = x


a = Bidon(5)
b = Bidon(6)

print(a.general, b.general)
a.general[0] = a.particulier
print(b.general)
print(a.general is b.general)
print(a.particulier,b.particulier)
```

```
[1, 2, 3] [1, 2, 3]
[5, 2, 3]
True
5 6
```

Mais comment faire pour avoir une fonction partagée par toutes les instances sans l'instance elle-même ? Impossible comme les attributs car le premier argument est toujours l'objet créé (self)

-> On a besoin d'autre chose (plus tard)

3 Classes Abstraites

Rappel: une classe abstraite est une classe qui définit une API susceptible de recevoir plusieurs implémentations différentes (dans des sous-classes), et qui laisse donc certaines méthodes non implémentées. En Java, on parle d'"Interfaces".

Il est facile d'implémenter une telle classe en utilisant l'exception "NotImplementError"

[3]:
```
class AbstractMaClasse:

    def __init__(self,x):
        self.base = x

    def a_method(self,x):
        return x*2

    def another(self,x,y):
        raise NotImplementedError("appel à une classe abstraite")
```

[4]:
```
class MaClasse(AbstractMaClasse):
    # note: la liste d'argument n'est bien sur pas figée
    def another(self,x,y):
        return self.base + x*y

a = AbstractMaClasse(1)
print("objet a créé")
b = MaClasse(2)
print("objet b créé")
print(b.another(2,3))
print(a.another(2,3))
```

```
objet a créé
objet b créé
8
```

```
---------------------------------------------------------------------------

NotImplementedError                       Traceback (most recent call last)

<ipython-input-4-08153b9f976e> in <module>()
 10 print("objet b créé")
 11 print(b.another(2,3))
---> 12 print(a.another(2,3))
```

```
        <ipython-input-3-099e7e766afa> in another(self, x, y)
          8
          9     def another(self,x,y):
    ---> 10         raise NotImplementedError("appel à une classe abstraite")
         11
         12


        NotImplementedError: appel à une classe abstraite
```

Un inconvénient de cette méthode est qu'il n'y a aucun contrôle que les sous-classes définissent bien toutes les méthodes au moment de la création d'une instance.

[27]:
```
class Oops(AbstractMaClasse):
    pass

a = Oops(1) # ne devrait pas marcher !
print("----- instance créée")

a.another(3,4)
```

```
----- instance créée


        ---------------------------------------------------------------------------

        NotImplementedError                       Traceback (most recent call last)

        <ipython-input-27-409c86027e6b> in <module>()
          5 print("----- instance créée")
          6
    ----> 7 a.another(3,4)


        <ipython-input-3-099e7e766afa> in another(self, x, y)
          8
          9     def another(self,x,y):
    ---> 10         raise NotImplementedError("appel à une classe abstraite")
         11
         12


        NotImplementedError: appel à une classe abstraite
```

3.1 Décorateurs

On a vu que l'on pouvait facilement définir des fonctions avec des fonctions en arguments ou même en résultat. Les fonctions sont des objets comme les autres (ou presque).

Python fournit de plus une syntaxe spéciale pour faciliter les transformations de fonctions: les décorateurs.

Ce mécanisme va permettre de traiter les problèmes mentionnés auparavant sur la définition des objets: méthode statique, méthode de classe, attributs protégés pratiques

Exemple: faisons une fonction qui aide à déclarer des fonctions "obsolètes" (en donnant un avertissement, mais en laissant le programme continuer)

[53]:
```
def obsolete(func):
      def new_func(*x,**opt):
        print("!! attention, appel à fonction obsolete:",func.__name__)
        return func(*x,**opt)
    return new_func

def mafonction(x):
    return x*2

mafonction = obsolete(mafonction)


mafonction(12)
```

```
!! attention, appel à fonction obsolete: mafonction
```

[53]:
```
24
```

Inconvénient: ne marche que si la fonction a un seul argument.

Avec des arguments quelconques cette fois :

[7]:
```
def obsolete(func):
    def new_f unc(*args,**kwargs):
        print("!! attention, appel à fonction obsolete:",func.__name__)
        return func(*args,**kwargs)
    return new_func

def mafonction(x,y):
    return x**2+y**2

mafonction = obsolete(mafonction)
mafonction(12,12)
```

```
!! attention, appel à fonction obsolete: mafonction
```

[7]:
```
288
```

Python fournit en fait une syntaxe plus pratique

[27]:
```
def obsolete(func):
    def new_func(*args,**kwargs):
        print("!! attention, appel à fonction obsolete:",func.__name__)
        return func(*args,**kwargs)
    return new_func

@obsolete
def mafonction(x,y):
    return x**2+y**2

mafonction(12,12)
```

```
!! attention, appel à fonction obsolete: mafonction
```

[27]:
```
288
```

Avantages des décorateurs : - plus concis -> plus clair ! - ne sépare pas la définition de la fonction de ses modifications - permet de les empiler de façon lisible - autorise les arguments

–> un outil d'abstraction très puissant

avec un argument : revient à empiler un autre "emballage" (wrapper) par dessus le premier

[25]:
```
def obsolete(date):
    def obsolete_decorator(func):
        def new_func(*args,**kwargs):
            print("!! attention, appel à fonction",
                  func.__name__,"obsolete depuis",date)
            return func(*args,**kwargs)
        return new_func
    return obsolete_decorator


@obsolete(2005)
def mafonction(x,y):
    return x**2+y**2

@obsolete(1980)
def oldfonction(x):
    return x*2

print(mafonction(12,12))
print(oldfonction(15))
```

```
!! attention, appel à fonction mafonction obsolete depuis 2005
```

```
288
!! attention, appel à fonction oldfonction obsolete depuis 1980
30
```

Trois décorateurs utiles sont prédéfinis:

- @property: définit un attribut avec des accès/modifications contrôlées.
- @staticmethod: méthode d'une class partagée par toutes les instances
- @classmethod méthode de classe -> utiles pour héritage, permet de définir des méthodes statiques sans mettre le nom de classe explicitement et donc peut être repris par les sous-classes telle quelle

Et un autre du module abc permet de définir des méthodes de classes abstraites

- @abc.abstractmethod

3.1.1 Attributs protégés

et si l'on voulait

- protéger des attributs
- garder une syntaxe simple pour l'accès et la modification ?

réponse: le décorateur @property

Reprenons l'exemple de la classe Date

[2]:
```python
# lourd et redondant ....
class Date:

    def __init__(self,jour,mois,annee):
        if jour>=1 and jour<=31:
            self.jour = jour
        else:
            self.jour = None
        if mois>=1 and mois<=12:
            self.mois = mois
        else:
            self.mois = None
        if annee!=0:
            self.annee = annee
        else:
            self.annee = None

    def set_jour(self,new):
        if new>=1 and new<=31:
            self.jour = new
            return True
        else:
            self.jour = None
```

```
            return False

    def get_jour(self):
        return self.jour

    # idem mois / annee

# serait déjà mieux avec attributs protégés: self.__jour, etc
# et en utilisant set/get dans l'__init__
# mais toujours un peu lourd
```

[30]:
```
# avec property
class Date:

    def __init__(self,jour,mois,annee):
        self.jour = jour
        self.mois = mois
        self.annee = annee

    @property
    def jour(self):
        return self.__jour

    @jour.setter
    def jour(self,new):
        if new>=1 and new<=31:
            self.__jour = new
        else:
            self.__jour = None


d = Date(32,1,2019)
#d.jour =
print(d.jour)
```

```
None
```

[28]:
```
# note: l'attribut est bien protégé:
print(d.__jour)
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-28-0cf40fa4cc10> in <module>
  1 # note: l'attribut est bien protégé:
```

```
----> 2 print(d.__jour)


AttributeError: 'Date' object has no attribute '__jour'
```

[29]:
```
d._Date__jour
```

[29]:
```
31
```

3.1.2 Méthode statique:

- methode d'une classe qui ne dépend pas d'une instance
- si on déclare normalement, on a une copie de la fonction à chaque instance

Exemple : supposons qu'on redéfinisse une classe complexe, et qu'on veut une méthode qui donne les racines enièmes de 1, i.e.

$$\exp(i2k\pi/n) = cos(2k\pi/n) + i \cdot sin(2k\pi/n)$$

[7]:
```
from math import cos,sin, pi

class Complexe:
    def __init__(self,r,i):
        self.reel = r
        self.imaginaire = i

    def __repr__(self):
        return "%f + i*(%f)"%(self.reel,self.imaginaire)

    def racine_unite(self,n,k):
        return  Complexe(cos(2*k*pi/n),sin(2*k*pi/n))

a = Complexe(1,1)
b = Complexe(-1,3)
print(a.racine_unite(3,1))
a.racine_unite is b.racine_unite
```

```
-0.500000 + i*(0.866025)
```

[7]:
```
False
```

En fait l'instance est inutile ici, on voudrait juste avoir les arguments n et k;

On définit alors la méthode comme "statique" pour ne pas avoir besoin d'instance

[8]:
```
from math import cos,sin, pi

class Complexe:
    def __init__(self,r,i):
        self.reel = r
        self.imaginaire = i

    def __repr__(self):
        return "%f + i*(%f)"%(self.reel,self.imaginaire)

    @staticmethod
    def racine_unite(n,k):
        return Complexe(cos(2.0*k*pi/n),sin(2.0*k*pi/n))

a = Complexe(1,1)
b = Complexe(-1,3)
print(Complexe.racine_unite(3,1))
a.racine_unite is b.racine_unite
```

```
-0.500000 + i*(0.866025)
```

[8]:
```
True
```

Exercice: faire une méthode pour créer un complexe à partir d'une forme polaire (θ, r)

3.1.3 Méthodes de classe

Imaginons maintenant qu'on veuille définir un nouveau conteneur, que l'on peut initialiser avec une liste, mais aussi à partir des éléments d'un dictionnaire.

On a deux solutions :

[13]:
```
class Table:
    def __init__(self,items):
            self.contenu = items

    def from_dict1(self,dico):
            self.contenu = dico.values()

    def from_dict2(self,dico):
            return Table(dico.values())
```

Pas très élégant, car il faut soit initialiser deux fois une instance ou bien créer une instance pour créer une autre instance

[14]:
```
a = Table([])
a.from_dict1({1:2,3:4})
# ou bien
```

```
c = a.from_dict2({5:6,7:8})
```

Déjà mieux avec une méthode statique

[15]:
```
class Table:
    def __init__(self,items):
        self.contenu = items

    @staticmethod
    def from_dict(dico):
        return Table(dico.values())

a = Table.from_dict({})
print(a)
```

```
<__main__.Table object at 0x7f5ec00df320>
```

Problème : si on veut maintenant faire une sous classe de Table, on doit tout réécrire si on veut garder la cohérence des constructeurs (from_dict faisant explicitement appel à la surclasse Table)

[16]:
```
# une table qui contient plus d'information
class AutreTable(Table):
    def __init__(self,items):
        self.contenu = items
        self.nb = len(items)

    @staticmethod
    def from_dict(dico):
        return AutreTable(dico.values())
```

Ne serait-il pas mieux d'avoir une méthode de classe à la place ? Du coup rien à réécrire.

[28]:
```
class Table:
    def __init__(self,items):
        self.contenu = items

    @classmethod
    def from_dict(cls,dico):
        return cls(dico.values())

class AutreTable(Table):
    def __init__(self,items):
        self.contenu = items
        self.nb = len(items)
```

```
a = AutreTable.from_dict({1:2})
a.__class__
```

[28]:
```
__main__.AutreTable
```

Exercices: définir des décorateurs pour

- garder en cache des résultats de fonction (“memoisation”)
- compter les appels de certaines fonctions
- répéter une fonction jusqu’à atteindre une condition [exam 2017_2018]

Indice: définir le décorateur comme objet et redéfinir la méthode __call__

[35]:
```
# exemple de méthode __call__
class Bidon:
    def __init__(self):
        self.ct = 0

    def __call__(self,x):
        self.ct += 1
        return x**2

call_me = Bidon()
print(call_me(3))
print(call_me(6))
print(call_me.ct)
```

```
9
36
2
```

[2]:
```
# avec fonction à un seul argument

class memoise:

    def __init__(self,func):
        self.func = func
        self.memoire = {}

    def __call__(self,x):
        if x in self.memoire:
            return self.memoire[x]
        else:
            val = self.func(x)
            self.memoire[x] = val
            return val
```

```
# decorateur équivalent à :
#fibo = memoise(fibo)
#puis appel normal fibo(50)

@memoise
def fibo(n):
    if n<2:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)


fibo(400)
```

[2]:
```
2848122981084896117579889376814609956153800887823048909864771956459692714040323239
01
```

[ ]:
```

```

[1]:
```
# version plus générale
class memoise:

    def __init__(self,f):
        self.func = f
        self.cache = {}

    def __call__(self,*args):
        if args in self.cache:
            return self.cache[args]
        else:
            val = self.func(*args)
            self.cache[args] = val
            return val

@memoise
def fibo(n):
    if n<2:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)

fibo(100)
```

[1]:
```
573147844013817084101
```

[9]:
```
class comptage:
    counts = {}
```

```
    def __init__(self,f):
        self.func = f
        self.name = f.__name__
        self.counts[self.name] = 0

    def __call__(self,*args):
        self.counts[self.name] = self.counts[self.name] + 1
        return self.func(*args)

    def __repr__(self):
        return self.func.__doc__

    @staticmethod
    def resultats():
        return comptage.counts


@comptage
def add(x,y):
    """addition"""
    return x+y

@comptage
def sub(x,y):
    """soustraction"""
    return x-y

add(3,sub(4,5))
add(5,3)
add(6,7)
print(comptage.resultats())
print(add)
```

```
{'sub': 1, 'add': 3}
addition
```

[10]:
```
# répétition

def repeat(times):
    def wrapper(func):
            def newfunc(*args):
                n = 0
                success = False
                while not(success) and n < times:
                    success = func(*args)
                    n = n + 1
```

```
                return success
            return newfunc
    return wrapper


from random import random

# version de base
def test(seuil):
    a = random()
    print(a)
    return a>seuil

print(test(0.7))
print('-'*5)
# version transformée
@repeat(5)
def test(seuil):
    a = random()
    print(a)
    return a>seuil

print(test(0.7))
```

```
0.5322311982626257
False
-----
0.03862136340891276
0.16786192224019492
0.8983522905338306
True
```

3.1.4 Classes abstraites

On a vu que définir des classes abstraites seulement avec des exceptions NotImplemtend.

Ceci est aussi permit par le module abc, qui définit le décorateur de méthode abstraite

[18]:
```
import abc

class AbstractMaClass(object):
    __metaclass__  = abc.ABCMeta

    def __init__(self):
        pass

    @abc.abstractmethod
```

```
    def another(self):
        pass


a = AbstractMaClass()
b = AbstractMaClass()
print(a,b)
```

```
<__main__.AbstractMaClass object at 0x109f7cb50> <__main__.AbstractMaClass
object at 0x109f7c9d0>
```

[19]:
```
class Vrai(AbstractMaClass):
    pass

c = Vrai()
print(c.another())
```

```
None
```

3.2 Héritage multiple

On peut faire hériter une classe de plusieurs classes pour récupérer tous les traits et attributs en les combinant.

La syntaxe est simple ... la sémantique un peu moins.

Imaginons un héritage multiple avec des classes ayant des méthodes de même nom ...

[20]:
```
class A:
    pass
    def m(self):
        print("appel de A.m")

class B(A):
    pass
    def m1(self):
        print("appel de B.m")

class C(A):
    def m(self):
        print("appel de C.m")

# héritage multiple
class D(B,C):
    pass

a = D()
a.m()
```

```
appel de C.m
```

Python résoud les héritages de fonction en commençant par la première superclasse si possible, puis les autres en suivant.

Quand faire de l'héritage multiple ?

- jamais ? On peut toujours s'en tirer en composant les classes B et C class D: def **init**(self): self._b = B() self._c = C()
- en ne mélangeant pas hiérarchie et utilisation. dans l'exemple ci-dessus soit B soit C ne devrait pas être transmis à D, en tout cas pas avec des parents directs

Exemple: imaginons une classe polygone, avec des sous-classes rectangle, losange, carré.

Il est tentant de considérer le carré comme héritant à la fois de losange et rectangle.

En pratique il suffit d'hériter d'une seule classe pour "faire le job".