INSTITUT
de MATHEMATIQUES
de TOULOUSE

Python intermédiaire pour les scientifiques

Session 4/5 : Python par l'objet

Laurent Risser

Ingénieur de Recherche à l'Institut de Mathématiques de Toulouse

lrisser@math.univ-toulouse.fr

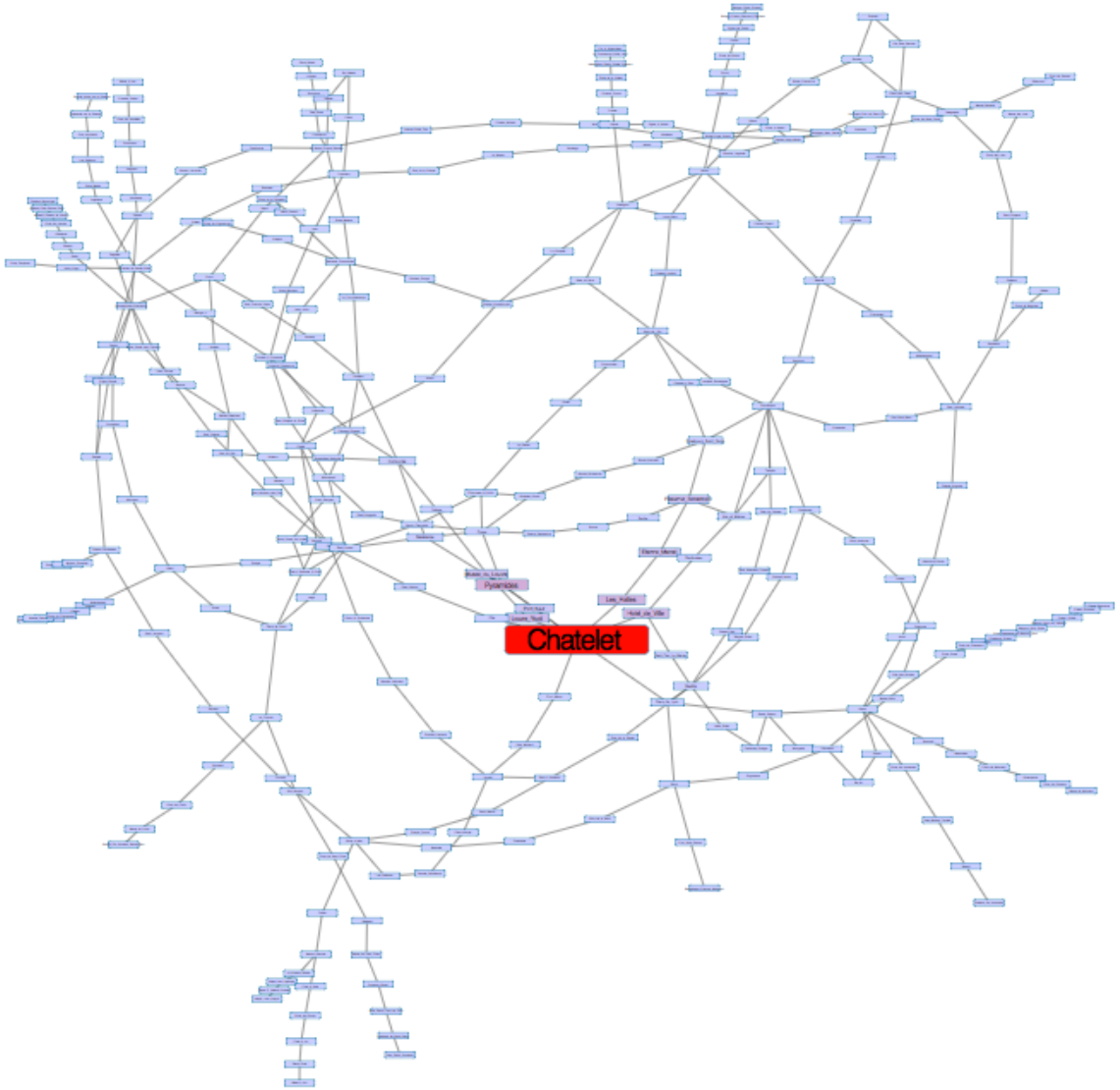
Résumé

En bref :

- Exemple introductif
- Principaux concepts et intérêt de la programmation orienté objet (POO).
- POO sous Python.
  - Définition de classe
  - Constructeurs/Destructeurs
  - Propriétés
  - Attribut __dict__
  - Attribut __doc__
  - Fonction dir
  - Attributs privés/publiques
  - Mangling
  - L'héritage de classe
  - Surcharge d'opérateurs
- Itérateurs.
- Décorateurs.
- Comment penser une classe.
- TP : création et instanciation de classes.

1) Exemple introductif

Code qui nécessite la programmation orientée objet (POO) pour grossir proprement



Travail avec Ioana Gavra (doctorante IMT) et Sébastien Gadat (Pr TSE/IMT)

→ Estimation du barycentre de graphes

Code d'estimation du barycentre :

EstimGraphBarycenter.py :

```
import networkx as nx
…
import numpy
```

```
def brownianMotion(…):
        …

def DrawAnID(…):
        …
```

```
RefGraph=nx.Graph()
csvfile=sys.argv[1]
with open(self.GraphFile,'r') as csvfile:
        data = csv.reader(csvfile, delimiter=' ', quotechar='%')
        self.RefGraph.add_weighted_edges_from([(row[0],row[1],float(row[2]))])
…
```

```
for i in range(IterationNb):
    …
```

→ Appel de modules

→ Définition de fonctions

→ Initialisation de l'algorithme

→ Algorithme stochastique itératif. On se rapproche peu à peu du centre.

Code d'estimation du barycentre :

EstimGraphBarycenter2.py :

```
import networkx as nx
…
class GraphCenterEstimator(…):
        def __init__(graph):
                …
        def ReturnBarycenter():
                …
        …
def returnSubgraph(graph,clusters,clusterID):
        …
…

RefGraph=nx.Graph()
csvfile=sys.argv[1]
with open(self.GraphFile,'r') as csvfile:
          data = csv.reader(csvfile, delimiter=' ', quotechar='%')
          self.RefGraph.add_weighted_edges_from([(row[0],row[1],float(row[2]))])
…

…
```

→ Appel de modules

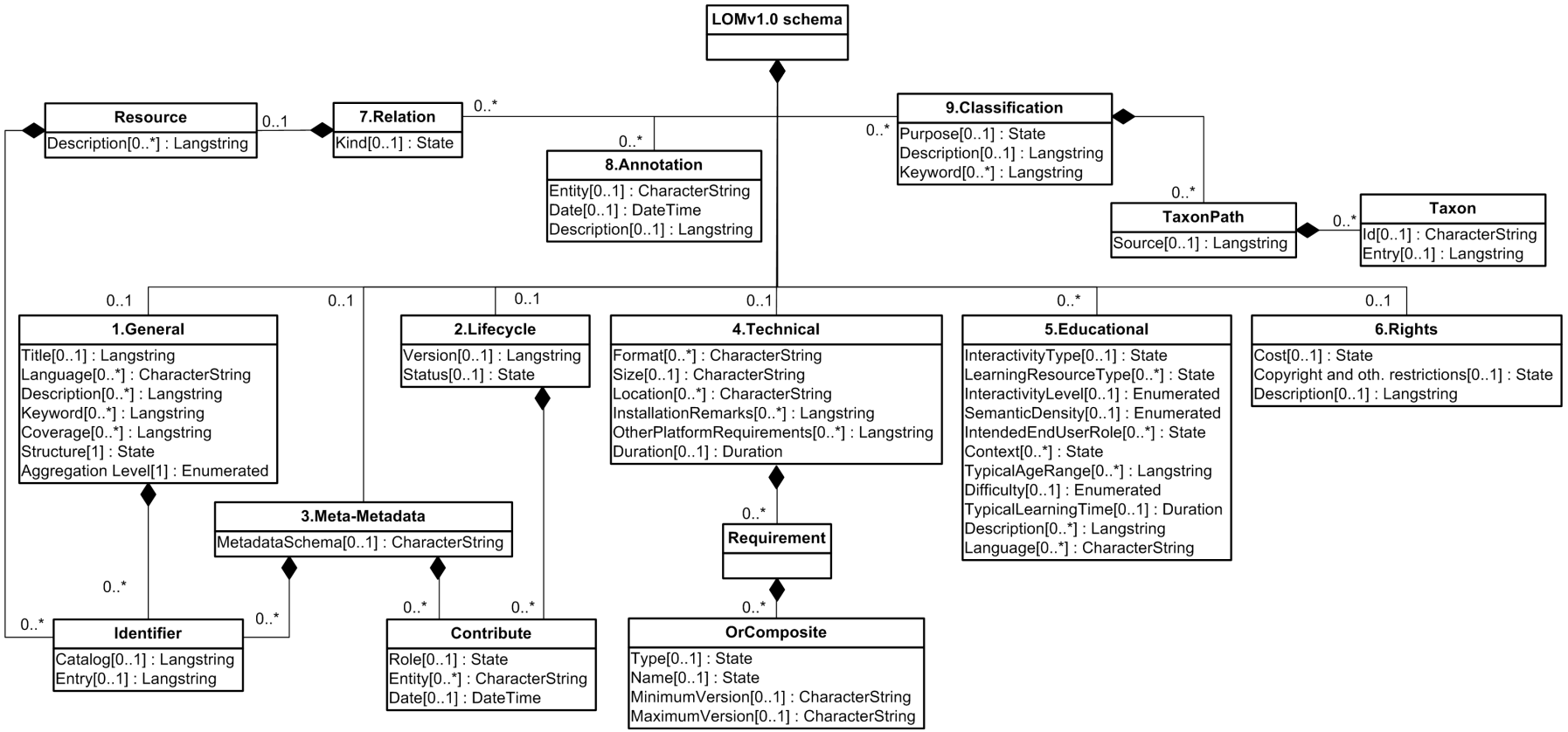
→ Tout ce qui est lié à l'algo d'estimation de base (pourrait être dans un autre fichier)

→ Définition de fonctions liées à l'algorithme multi-échelle

→ Initialisation de l'algorithme

→ Algorithme multi-échelles

Pour de plus gros codes : exemple de diagramme de classes (wikidepia)



Pour résumer :

- La **programmation orientée objet** (POO) permet de créer des entités objets que l'on peut
- aisément manipuler sans nécessairement avoir besoin d'en comprendre ses détails.

- La programmation orientée objet impose des structures solides et claires.

- Les objets peuvent interagir entre eux, cela facilite grandement la compréhension du code
- et sa maintenance.

- Une **classe** regroupe des **méthodes** (fonctions) et des **attributs** (variables).

- Un **objet** est une instance de classe.

Par rapport aux modules Python sur des codes de taille raisonnable :

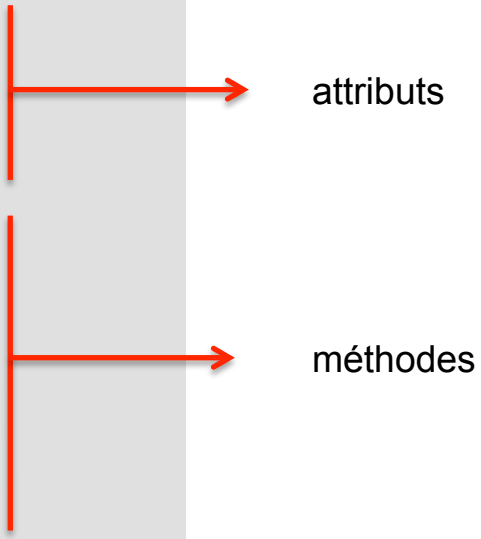
- Recours à des variables globales beaucoup moins pertinent

- Facilité de sauvegarder facilement l'état d'une classe avec pickle

- Cadre de programmation plus standardisé

Concepts : méthodes (fonctions) et des attributs (variables)

```
classe Image :
  tailleX
  tailleY
  MatriceDesIntensites
  function construct(TX,TY)
  function destruct()
  function get(x,y)
  function put(value,x,y)
  function read(fileName)
  function write(fileName)
```



Concepts : Héritage

```
classe Image :
  tailleX
  tailleY
  MatriceDesIntensites
  function construct(TX,TY)
  function destruct()
  function get(x,y)
  function put(value,x,y)
  function read(fileName)
  function write(fileName)
```

```
classe ImageMedicale(Image) :
  resolutionX
  resolutionY
  OrigineX
  OrigineY
  DateAcquisition
  NomPatient
  ...
```

Concepts : Utilisation de classes dans d'autres classes

```
classe Image :
  tailleX
  tailleY
  MatriceDesIntensites
  function construct(TX,TY)
  function destruct()
  function get(x,y)
  function put(value,x,y)
  function read(fileName)
  function write(fileName)
```

```
Classe FlouteurImage:
  public:
    function construct(Image,niveauDeFlou)
    function run()
  private:
    tailleImageX
    tailleImageY
    ImagesTemporaires
    function DefineKernel(niveauDeFlou)
    function FFT(image)
    function IFFT(image)
    function MultKernelPtAPt()
```

Type d'accès :

- Private : accessible qu'à l'intérieur de la classe
- Public : accessible partout (classe et instance, i.e. objet)
- Protected : accessible dans la classe et dans les sous-classes (héritées)

3.1) Définition de classes en Python :

```
class ClassName(superclass):
        def __init__(self, args):
                self.variable=0
                ...

        def method_name(self, args):
                ...
```

- La classe *ClassName* hérite en option de la classe *superclass*.
- *__init__* est le constructeur de la classe (optionnel)
- *functionname* est une méthode (fonction) de la classe
- self.variable est un attribut (variable) de la classe

3.1) Définition de classes en Python :

```
class ClassName(superclass):
        def __init__(self, args):
                self.variable=0
                …

        def method_name(self, args):
                …
```

Utilisation d'une classe :

```
var = ClassName()
var.variable =10
var.method_name(args)
print(var.variable)
```

Création de l'objet var

Modification d'un attribut

Utilisation d'une méthode de var

Accès à la valeur d'un attribut

3.1) Définition de classes en Python :

```
class ClassName(superclass):
        def __init__(self, args):
                self.variable=0
                …

        def method_name(self, args):
                …
```

Utilisation d'une classe :

```
var = ClassName()
self.variable =10
var.method_name(args)
print(self.variable)
```

Création de l'objet var

Modification d'un attribut

Utilisation d'une méthode de var

Accès à la valeur d'un attribut

Convention sur les noms :

```
var = ClassName()
```

Nom de classe commençant par des majuscules

Nom d'objet sans majuscules

3.2) Constructeurs et destructeurs :

Python gère automatiquement la destruction des objets mais il peut être utile de s'en occuper manuellement (envoi de message, libération explicite de mémoire)

```
class DelTester():
    def __init__(self, tailleImageX, tailleImageY):
        print("cree une image")
        self.Image = np.ones((tailleImageX, tailleImageY))
    def showsize(self):
        print("Image de taille :")
        print(self.Image.shape)
    ...
    def __del__(self):
        print("supprime une image")
        del self.Image
```

Constructeur (optionnel mais courant)

Fonctions

Destructeur (optionnel et peu courant)

```
import numpy as np

Im=DelTester(1000,2000)
→ cree une image
Im.showsize()
→ Image de taille :
→ 1000,2000
del Im
→ supprime une image
```

3.3) Les propriétés :

Il est préférable de passer par des propriétés pour changer les valeurs des attributs :

Convention de passer par des **getter** (ou **accesseur**) et des **setter** (**mutateurs**)

```
class Voiture():
    def __init__(self):
        self.couleur="rouge"
    def get_couleur(self):
        print  "Recuperation de la couleur"
        return self.couleur
    def set_couleur(self, v):
        print "Changement de couleur"
        self.couleur  =  v
```

Lors de l'instanciation de la classe, self.couleur a la valeur 'rouge'.

accesseur

mutateur

3.4) L'attribut spécial __dict__ :

Cet attribut spécial vous donne les valeurs des attributs de l'instance :

```
class Voiture():
        def __init__(self):
            self.couleur="rouge"
        def get_couleur(self):
            print  "Recuperation de la couleur"
            return self.couleur
        def set_couleur(self, v):
            print "Changement de couleur"
            self.couleur  =  v
```

```
ma_voiture=Voiture()
ma_voiture.__dict__

→ {'couleur' : 'rouge'}
```

3.5) L'attribut spécial __doc__ :

Cet attribut spécial vous donne une documentation sur la classe d'un object :

```
class Voiture():
    """
    Classe voiture avec option couleur
    """
    def __init__(self):
        self.couleur="rouge"
    def get_couleur(self):
        print  "Recuperation de la couleur"
        return self.couleur
    def set_couleur(self, v):
        print "Changement de couleur"
        self.couleur  =  v
```

```
ma_voiture=Voiture()
ma_voiture.__doc__

→ '\n    Classe voiture avec option couleur\n '
```

3.5) L'attribut spécial __doc__ :

Cet attribut spécial vous donne une documentation sur la classe d'un object :

```
class Voiture():
        """
        Classe voiture avec option couleur
        """
        def __init__(self):
            self.couleur="rouge"
        def get_couleur(self):
            print  "Recuperation de la couleur"
            return self.couleur
        def set_couleur(self, v):
            print "Changement de couleur"
            self.couleur  =  v
```

Dans un code destiné a être réutilisé, il faut absolument définir ce que fait la classe et ses entrées et sorties.

```
ma_voiture=Voiture()
ma_voiture.__doc__

→ '\n    Classe voiture avec option couleur\n '
```

3.5) L'attribut spécial __doc__ :

Cet attribut spécial vous donne une documentation sur la classe d'un object :

```
class Voiture():
    """
    Classe voiture avec option couleur
    """
    def __init__(self):
        self.couleur="rouge"
    def get_couleur(self):
        print  "Recuperation de la couleur"
        return self.couleur
    def set_couleur(self, v):
        print "Changement de couleur"
        self.couleur  =  v
```

```
ma_voiture=Voiture()
ma_voiture.__doc__

→ '\n    Classe voiture avec option couleur\n '
```

Petit test :

```
a=1
a.__doc__

→ "int(x=0) -> int or long
    ...   base=0)\n4"
```

Toute variable est un objet en Python !!!

3.6) Fonction dir :

La fonction dir vous donne un aperçu des méthodes de l'objet:

```
class Voiture():
        def __init__(self):
            self.couleur="rouge"
        def get_couleur(self):
            print  "Recuperation de la couleur"
            return self.couleur
        def set_couleur(self, v):
            print "Changement de couleur"
            self.couleur  =  v
```

```
ma_voiture=Voiture()
dir(ma_voiture)

→ ['__doc__', '__init__', '__module__','get_couleur', 'set_couleur']
```

3.7) Méthodes/attributs privés et publiques :

Toutes les méthodes et tous les attributs d'une classe Python sont publiques !

```
class Voiture():
        def __init__(self):
            self.couleur="rouge"
        def get_couleur(self):
            print  "Recuperation de la couleur"
            return self.couleur
        def set_couleur(self, v):
            print "Changement de couleur"
            self.couleur  =  v
        def _fonction_interne(self,val)
                self._tempVal=val
        _tempVal=0
```

Les méthodes/attributs commençant par _ sont considérés comme privés (détails d'implémentation intra-classe).

3.8) Le mangling :

Utilisé pour les membres de classes (pseudo) privés avec des noms que l'on peut retrouver dans différentes classes → Evite les confusions de noms, vu que tout est publique.

```
class Voiture():
        def __init__(self):
            self.couleur="rouge"
        def get_couleur(self):
                ...
        ...
        def _fonction_interne(self,val)
                self._tempVal=val
        _tempVal=0
        def __show():
                print(self.couleur)
```

Le nom commence par __ et ne termine pas par __

```
ma_voiture=Voiture()
dir(ma_voiture)
→ ['__doc__', '__init__', '__module__','get_couleur', ... ,'_Voiture__show']
ma_voiture._Voiture__show
→ rouge
```

3.9) L'héritage de classe :

Permet de créer de nouvelles classes à partir d'une classe existante :

```
class Voiture():
        self.couleur="rouge"
        def __init__(self):
            ...
        def get_couleur(self):
            print  "Recuperation de la couleur"
            return self.couleur
        ...
```

```
class VoitureCabriolet(Voiture):
        def __init__(self):
            ...
        def get_type_de_toit(self):
            ...
```

3.9) L'héritage de classe :

Permet de créer de nouvelles classes à partir d'une classe existante :

```
class Voiture():
        self.couleur="rouge"
        def __init__(self):
            …
        def get_couleur(self):
            print  "Recuperation de la couleur"
            return self.couleur
        …
```

ATTENTION : __init__ est écrasé (surcharge de méthode) lors d'un héritage. Il vaut mieux définir les valeurs par défaut ailleurs ou bien faire attention à les redéfinir dans les nouveaux __init__ si l'héritage est utilisé.

```
class VoitureCabriolet(Voiture):
        def __init__(self):
            …
        def get_type_de_toit(self):
            …
```

3.9) L'héritage de classe :

Permet de créer de nouvelles classes à partir d'une classe existante :

```
class Voiture():
        self.couleur="rouge"
        def __init__(self):
            …
        def get_couleur(self):
            print  "Recuperation de la couleur"
            return self.couleur
        …
```

```
class VoitureCabriolet(Voiture):
        def __init__(self):
            …
        def get_type_de_toit(self):
            …
        def get_couleur(self):
          return [self.couleur_carrosserie, self.couleur_toit]
        …
```

NOTE : D'autres méthodes peuvent intentionnellement être écrasées par surcharge de méthode

3.10) Surcharge d'opérateurs :

On se souvient que tout est objet en Python… et on remarque que de même opérateurs peuvent être utilisés sur des objets différents.

```
a=1
b=2

a+b
→ 3
```

```
a='hello '
b='world'

a+b
→ 'hello world'
```

Comment définir de tels opérateurs sur nos objets ?

→ Utilisation de méthodes __add__, __sub__, __mul__, __eq__, …

3.10) Surcharge d'opérateurs :

```
import numpy as np
class Vec2D:
        def __init__(self, x, y):
                self.x=x
                self.y=y
        def __eq__(self, vB):
                return (self.x==vB.x) and (self.y==vB.y)
        def __add__(self, vB):
                return Vec2D(self.x+vB.x,self.y+vB.y)
        def __sub__(self, vB):
                return Vec2D(self.x-vB.x,self.y-vB.y
        def __mul__(self, c):
                if isinstance(c,Vec2D):
                        return self.x*c.x+self.y*c.y
                else:
                        return Vec2D(c*self.x,c*self.y)
        def __div__(self, c):
                return Vec2D(self.x/c, self.y/c)
        def __abs__(self):
                return np.linalg.norm(np.array([self.x, self.y]))
        def __str__(self):
                return '('+str(self.x)+','+str(self.y)+')'
```

```
a = Vec2D(4,5)
b = Vec2D(6,7)
print a==b
False
print a+b
(10,12)
print a-b
(-2,-2)
print a*b
59
print abs(a)
6.40312423743
```

Ce qu'affiche print

Containers : Objets spéciaux qui possèdent un nombre arbitraire d'autres objets.

→ exemple : listes, dictionnaires, …

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,
```

Containers : Objets spéciaux qui possèdent un nombre arbitraire d'autres objets.

→ exemple : listes, dictionnaires, …

```
for element in [1, 2, 3]:
        print element
for element in (1, 2, 3):
        print element
for key in {'one':1, 'two':2}:
        print key
for char in "123":
        print char
for line in open("myfile.txt"):
        print line,
```

Pour se faire, for utilise la classe des itérateurs : iter()

Classe des itérateurs : Utilise massivement la méthode next()

```
s = 'abc'
it = iter(s)
it
→ <iterator object at 0x00A1DB50>
it.next()
→ 'a'
it.next()
→ 'b'
it.next()
→ 'c'
it.next()
→ Traceback (most recent call last):
→   File "<stdin>", line 1, in ?
→     it.next()
→ StopIteration
```

Créer un itérateur :

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def next(self): # Python 3: def __next__(self)
        if self.current > self.high:
            raise StopIteration
        else:
            self.current = self.current+1
            return self.current - 1
```

Constructeur

Retourne un objet itérateur et est lancé au début de la boucle

Retourne la *prochaine* valeur et est lancé à chaque Incrément de la boucle. Remonte l'exception StopIteration à la fin de la boucle

```
for c in Counter(3, 8):
    print c
→ 3
→ 4
→ ...
→ 8
```

Générateurs: Il est aussi souvent possible et plus simple d'utiliser un générateur.

```
def counter(low, high):
    current = low
    while current <= high:
        yield current
        current = current+1
```

```
for c in counter(3, 8):
    print c
→ 3
→ 4
→ ...
→ 8
```

- Plus simple en pratique
- Moins customisable (un générateur est un sous-type d'itérateur)

Expressions générateurs: Elles sont une forme compacte de générateurs.

```
sum(i*i for i in range(10))
→ 285

data = 'golf'
list(data[i] for i in range(len(data)-1,-1,-1))
→ ['f', 'l', 'o', 'g']
```

Proche des compréhensions qui ne font que créer une liste :

```
Squares=[i*i for i in range(10)]
Squares
→ [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Décorateurs: Permettent de modifier le comportement *par défaut* des fonctions.

```
def mon_decorateur(function):
    def fonction_enveloppante():
        if user != 'Laurent':
            print('Action refusee')
        else:
            return function()
    return fonction_enveloppante
```

```
@mon_decorateur
def fonction_coucou():
    print('hello world')

user = 'Laurent'
fonction_coucou()
→ hello world

user = 'Martin'
fonction_coucou()
→ Action refusee
```

Agit comme :

mon_decorateur(fonction_coucou())

```
def mon_decorateur(func):
    def func_wrapper(name):
        global FirstTime
        print 'Temps depuis le debut : '+str(time.time()-FirstTime)
        return func(name)
    return func_wrapper

@mon_decorateur
def RajouteDeux(val):
    return val+2
```

Utilisation d'une variable globale pour connaître le temps écoulé

```
import time

global FirstTime
FirstTime=time.time()

print RajouteDeux(3)
→ Temps depuis le debut : 2.14576721191e-06
→ 5
print RajouteDeux(30)
→ Temps depuis le debut : 3.40938568115e-05
→ 32
```

6) Penser une classe en pratique

Le plus souvent, une classe sera comme une brique d'un gros code dans laquelle :

- On a bien spécifié ses entrées
- On a bien spécifié ses sorties
- On ne veut pas se soucier de ses détails d'implémentations

6) Penser une classe en pratique

Le plus souvent, une classe sera comme une brique d'un gros code dans laquelle :

- On a bien spécifié ses entrées
- On a bien spécifié ses sorties
- On ne veut pas se soucier de ses détails d'implémentations

Si la classe est utilisée pour un gros calcul ou est un *modificateur* :

- Initialisation des variables principales dans le `__init__()`
- Initialisation de variables secondaires avec des setters
- Definition d'une méthode `run()` qui lance les calculs et retourne le résultat

6) Penser une classe en pratique

Le plus souvent, une classe sera comme une brique d'un gros code dans laquelle :

- On a bien spécifié ses entrées
- On a bien spécifié ses sorties
- On ne veut pas se soucier de ses détails d'implémentations

Si la classe est utilisée pour un gros calcul ou est un *modificateur* :

- Initialisation des variables principales dans le `__init__()`
- Initialisation de variables secondaires avec des setters
- Definition d'une méthode `run()` qui lance les calculs et retourne le résultat

Si la classe contient des données que l'on manipule au cours du temps :

- Initialisation des données dans le `__init__()`
- Modification et récupération de données avec des getters et setters
- Définition d'une série de méthodes faisant appel à d'autres méthodes pseudo-privées

6) Penser une classe en pratique

Le plus souvent, une classe sera comme une brique d'un gros code dans laquelle :

- On a bien spécifié ses entrées
- On a bien spécifié ses sorties
- On ne veut pas se soucier de ses détails d'implémentations

Si la classe est utilisée pour un gros calcul ou est un *modificateur* :

- Initialisation des variables principales dans le `__init__()`
- Initialisation de variables secondaires avec des setters
- Definition d'une méthode `run()` qui lance les calculs et retourne le résultat

Si la classe contient des données que l'on manipule au cours du temps :

- Initialisation des données dans le `__init__()`
- Modification et récupération de données avec des getters et setters
- Définition d'une série de méthodes faisant appel à d'autres méthodes pseudo-privées

Dans tous les cas :

Bien commenter les classes et méthodes

6) Penser une classe en pratique

Le plus souvent, une classe sera comme une brique d'un gros code dans laquelle :

- On a bien spécifié ses entrées
- On a bien spécifié ses sorties
- C

S4_TP1 : modifier S3_TP2 en :

(1) Implémentant une classe *TF_Image* qui va contenir les images et englober les fonctions de Image_IO.py, Image_Treatment.py et Image_Visualisation.py

(2) Implémentant une classe *TF_Image_Mapper* qui permet de calculer la translation optimale entre deux images de type *TF_Image* et *TF_Image*.

Si la

- In
- In
- D

Si la classe contient des données que l'on manipule au cours du temps :

- Initialisation des données dans le `__init__()`
- Modification et récupération de données avec des getters et setters
- Définition d'une série de méthodes faisant appel à d'autres méthodes pseudo-privées

S4_TP2 : jouer avec des décorateurs du fichier S4_TP2.py

Dans tous les cas :

Bien commenter les classes et méthodes

Références :

- http://docs.python.org/2/tutorial/
- http://apprendre-python.com (exemple qui ne marche pas sur les décorateurs)
- http://thecodeship.com/patterns/guide-to-python-function-decorators/
- https://openclassrooms.com/courses/apprenez-a-programmer-en-python/
- https://python.developpez.com/
- http://fr.wikipedia.org/wiki/Python_%28langage%29

Remerciements :

- Etienne Gondet
- Sébastien Déjean
- Jérome Fehrenbach
- Vous

MERCI !!!