

# Programmation Python

## une introduction complète

Jean-Pierre MESSAGER

`jp@xiasma.fr`

v2.0a - février 2020

**Attribution - Pas d'Utilisation Commerciale - Pas de Modification**

**CC BY-NC-ND**



## Conditions de distribution

*Licence Creative Commons*

*Attribution - Pas d'Utilisation Commerciale -*

*Pas de Modification 3.0 France*

(CC BY-NC-ND 3.0 FR)



Ceci est un **résumé** de la licence complète disponible à :

<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/legalcode>

Vous êtes autorisé à :

Partager — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats

L'offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

Selon les conditions suivantes :

**Attribution** — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.

**Pas d'utilisation commerciale** — Vous n'êtes pas autorisé à faire un usage commercial de cette œuvre, tout ou partie du matériel la composant. *Le titulaire des droits peut autoriser tous les types d'utilisation ou au contraire restreindre aux utilisations non commerciales (les utilisations commerciales restant soumises à son autorisation).*

**Pas de modifications** — Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'œuvre originale, vous n'êtes pas autorisé à distribuer ou mettre à disposition l'œuvre modifiée.

**Pas de restrictions complémentaires** — Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser l'œuvre dans les conditions décrites par la licence.

- **Chapitre 1. Bases du langage Python**
- **Chapitre 2. Classes, objets et modules**
- **Chapitre 3. Entrées/Sorties, bases de données, interfaces et Web**

# Chapitre 1

## Bases du langage Python

- Mise en œuvre
- Types de bases
- Collections
- Structures de contrôle
- Fonctions et générateurs
- Compréhensions

## **Chapitre 2**

### **Classes, objets et modules**

- Définition de classes et instanciation
- Héritage
- Surcharge d'opérateurs
- Décorateurs et méthodes abstraites
- UML et patrons de conception
- Modules
- Tests unitaires

## **Chapitre 3**

# **Entrées/Sorties, bases de données, interfaces et Web**

- Entrées/sortie sur des fichiers, exceptions et gestionnaire de contextes
- Format de données
- Bases de données
- Interfaces graphiques
- Développement Web
- Interfaçage C/Python

## Remerciements

L'auteur remercie Christophe, Mickaël, Simon, Stéphane et Vincent qui ont accompagné la rédaction de la version initiale de ce support en suivant une formation animée par l'auteur, à Lyon en 2018

L'auteur remercie Guido Van Rossum ainsi que tous les membres de la communauté des programmeurs et utilisateurs de Python

*à L-L.N.*

# **Chapitre 0**

## **Introduction au langage Python**



# Le langage Python

- Langage dont les implémentations sont des logiciels libres (*open source*)
  - CPython (implémentation de référence)
  - Jython (JVM), IronPython (.NET)
  - Pypy, Brython (JS) [brython.info](http://brython.info), MicroPython (C)
- Langage orienté objet, typé et dynamique
  - En Python TOUTE donnée est un objet
- Permet les styles procédural, fonctionnel, objet
- Portable : UNIX, macOS X, GNU/Linux, MS Windows
- Python 2 n'est plus maintenu depuis le 1er janvier 2020
  - Pensez à migrer votre code en Python 3 !

# Organisation du projet

- Guido Van Rossum est le créateur initial du langage et jusqu'à récemment le BDFL (*Benevolent Dictator for Live*)
- La *Python Software Foundation* gère le développement du projet
- D'où vient donc ce drôle de nom ?
- Le site python.org est le site officiel du langage
  - Documentation, sources, binaires
  - Wiki
  - PEPs: *Python Enhancement Proposals* (acceptés ou non...)

# Python 2 et 3

- On peut importer les fonctionnalités non rétro-compatibles de Python 3 dans Python 2

```
from __future__ import ...
```

- division, print\_function, unicode\_literals, ...
- Le script 2to3 convertit un script v2 en v3
- Le module six aide à l'écriture de scripts portables
- *10 awesome features of Python that you can't use because you refuse to upgrade to Python 3 or turning it up to 13!*  
<https://www.asmeurer.com/python3-presentation/python3-presentation.pdf>

# Python interactif

- Python fournit plusieurs environnements interactifs
  - Boucle REPL: Read Eval Print Loop
  - Lancement de python/python3 dans un terminal UNIX ou Windows
  - IDLE (écrit en Python)
  - IPython, Jupyter, bpython3, etc. et dans les IDE la "*Python Console*"

```
$ python3
>>> print("Hello world!")
Hello world!
>>> 42 + 1
43
>>> 'John Doe'
'John Doe'
>>> print
<built-in function print>
>>> quit()
```

# REPL

- En Python 2 on peut exécuter *from \_\_future\_\_* en interactif (comme au début d'un script)
- L'aide sur les objets est accessible par la fonction *help*
- La fonction *dir* permet de connaître les attributs d'un objet

```
>>> dir('spam')
...
>>> dir(42)
...
>>> dir(int)
```

- La fonction *dir* permet de connaître les attributs d'un objet

```
>>> help('spam') # objet chaîne de caractère
>>> help(str)    # type chaîne de caractère
```

# Modèle de script Python

- Le nom du fichier DOIT se terminer en `.py` certains caractères sont à proscrire comme le tiret (-)
- Première ligne « *she-bang* » `#!`

- Prise en compte sous UNIX et GNU/Linux si le fichier est exécutable :

```
$ chmod +x fichier.py  
$ ./fichier.py
```

- Sur tout système on peut lancer directement l'interpréteur

```
$ python3 fichier.py  
C:\MyProject> python3 fichier.py
```

- Spécification de l'encodage

- Obligatoire en Python 2 si vous voulez avoir des accents même dans les commentaires. UTF-8 est le défaut si *omis* en Python 3

```
#!/usr/bin/env python3  
# -*- encoding: utf-8 -*-  
'''Déjà de la documentation si on veut en fournir'''  
  
# Ici un commentaire  
print('Hello world!')
```

# Compilé ou interprété ?

- L'exécution d'un script ou d'un module Python se fait en deux étapes :
  - Analyse du code et compilation en « *Byte Code* »
    - Ce *Byte Code* peut se retrouver dans un fichier `.pyc`
  - Exécution du *Byte Code* (qui est indépendant de l'architecture et du système d'exploitation) dans la PVM (*Python Virtual Machine*)
- Le module *dis* permet d'accéder à une représentation du *Byte Code*

# Environnements de développement (Linux)

- Installer Python (2) et Python 3 à partir des paquetages de la distribution :
  - **Debian/Ubuntu:** (ouvrez un nouveau terminal) et lancez :  
`$ sudo apt install python python3`
  - **RHEL/CentOS/Fedora:** `sudo dnf install python...`
  - Activer des dépôts supplémentaires peut être utile (EPEL, IUS, PPAs)
  - **macOS X :** installer *brew* et faire `brew install python3 ...`
- Éditeurs : vim, vim-gnome, emacs  
pour vim créer le fichier `~/ .vimrc` avec quelques lignes au minimum :  
`syntax on`  
`filetype indent plugin on`  
`set tabstop=8`  
`set expandtab`  
`set softtabstop=4`  
`set shiftwidth=4`



# Environnements de développement sous MS Windows

- Télécharger sur [python.org](http://python.org) les installeurs officiels binaires
- Demander à modifier la variable PATH
- Éditeurs de texte : notepad++, gvim, emacs

# Environnements de développement intégrés

- **PyCharm Community Edition** : installation avec *snap*

```
$ sudo snap install pycharm-community --classic  
$ pycharm-community &  
configuration initiale et dans le dock : « Ajouter aux favoris »
```

- **Pycharm Community Edition pour Ubuntu** : installation avec *umake*

```
$ sudo add-apt-repository ppa:ubuntu-desktop/ubuntu-make  
$ sudo apt-get update  
$ sudo apt-get install ubuntu-make  
$ umake ide pycharm
```

Détails : <https://itsfoss.com/install-pycharm-ubuntu/>

- Eclipse + PyDev (extension d'Eclipse)
- Je vous suggère fortement d'installer **PyCharm** pour cette semaine, ensuite vous ferez ce que vous voulez (et moi aussi)

# Notre premier programme

- On part d'une configuration fraîche, choisissez le thème qui vous convient (Darcula ou Light), *Create New Project* : remplacez "untitled" par : **FormationPython**
- Clic-droit sur le nom du projet "**FormationPython**" dans le panneau de droite et faites :  
**New -> Directory -> "TP"** (Travaux Pratiques) (nous y mettrons nos fichiers)
- Clic-droit sur **TP / New -> Python File -> hello.py**  
Dans le fichier mettez :

```
#!/usr/bin/env python3
'''My first program'''
print('Hello world!')
```

Puis : **Bouton droit -> Run**

- Dans la console vérifiez que vous voyez le résultat de l'exécution de notre programme.

# Exécuter le script hors de PyCharm

- Ouvrir un terminal (bouton droit : Nouveau Terminal)

```
$ cd Py<tab>
  cd PycharmProjects/F<tab>
  cd PycharmProjects/FormationPython/TP/ <Entrée>
$ pwd
/home/stagiaire/PycharmProjects/FormationPython/TP
$ ls
hello.py
$ python3 hello.py # ou python si seule la v3 est installée
Hello World!
$ chmod +x hello.py # man chmod (rend exécutable UNIX)
$ ./hello.py
Hello World!
$ python3
>>> import hello
Hello World!
>>> help(hello)
```

# Exécuter le script hors de PyCham

- Importer notre script à partir de la "Python Console" de Pycharm

```
>>> import TP.hello  
Hello World!  
>>> help(TP.hello)
```

- Nous pouvons donc exécuter ou importer nos scripts Python aussi bien à partir du terminal Linux, UNIX ou MS Windows qu'à partir de PyCharm



Programmation Python — une introduction complète

# Chapitre 1

# Bases du langage Python

Mise en œuvre, types de bases, collections, structures de contrôle, fonctions, générateurs, compréhensions

Jean-Pierre MESSEAGER

`jp@xiasma.fr`

v2.0a - février 2020

**Attribution - Pas d'Utilisation Commerciale - Pas de Modification**

**CC BY-NC-ND**



- Types de données numériques et chaînes
- Collections
- Boucles
- Fonctions
- Compréhensions



# Type de données numériques

- Types intégrés à Python, compatibles entre eux (opérations usuelles)
  - Entiers (int), Flottant (float), Complexes (complex)
  - Représentations littérales :
    - Décimal: 42
    - Hexa: 0x10A
    - Binaire: 0b011001
    - Octal: 0o14 (zéro + o + chiffres)
    - Point décimal : 3.14
    - Notation scientifique : 1.23e42
    - Complexes : 2 + 3j
- La représentation que la REPL vous montre est TOUJOURS en base 10
  - Fonctions de conversion vers d'autres bases *oct()*, *hex()*, *bin()*

# Opérateurs sur les nombres

- Acceptent des combinaisons des différents types numériques
- Usuels : + - / \* ; exponentiation \*\* ; opérateurs en calcul binaire (not, décalage, et, ou binaires) : ~ << >> & |, comparaison <=, >=, <, >, ==, !=
- On peut les enchaîner (économise l'usage de l'opérateur *and*) :

```
>>> 5 < 10 <= 12
True
>>> 5 < 10 and 10 <= 12
True
```

- Ce sont en fait des fonctions, regardez :  
    `dir(42)` ou `dir(int)`  
    `dir` affiche la liste des attributs d'un objets  
    `42 + 1` est la même chose que :  
    `(42).__add__(1)` qui est lui même la même chose que :  
    `int.__add__(42, 1)`  
    C'est du « typage **canard** » (*Duck Typing*)  
    Nous reverrons ces conventions dans la partie sur la programmation orienté objet
- Règles de priorité usuelles et **parenthèses** pour grouper différemment

# Chaînes de caractères

- Littéralement plusieurs façon de les noter :
  - Entre guillemets ou apostrophes (**strictement équivalents**)
  - On peut utiliser \ pour *protéger* un caractère spécial (\\, \' dans une chaîne encadrée par des apostrophes, \" dans une chaîne encadrée par des guillemets) ou indiquer un caractère spécial \t, \n (retour chariot, tabulation, Unicode)
  - On peut aussi encadrer par ''' ou "" (longues chaînes, documentation interne)
  - On peut aussi demander à ne plus interpréter le \ de façon spéciale, en préfixant avec r (*raw*)  
`r'\t'` plutôt que `'\\t'`
    - Utile pour les chemins d'accès sous Windows (r"C:\Windows\System32\")
    - Utile pour les expressions régulières (nous y reviendrons)

# Opération sur les chaînes

- Opérateurs + et \* (concaténation et répétition), comparaison ==, <=, >=, <, >
- Beaucoup de méthodes : regardez `dir(str)`

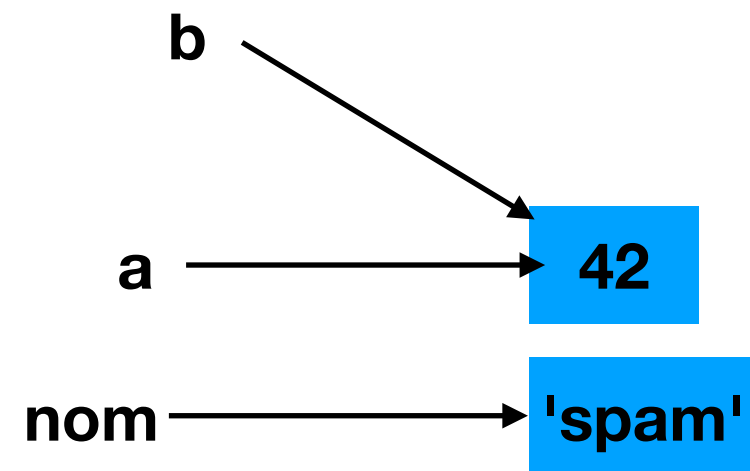
```
>>> a = 'spam'
>>> a.isupper()
False
>>> a.upper()
'SPAM'
>>> a.upper
<built-in method ...>
>>> a
'spam'
```

- *find()*, *replace()*, etc.
- Extraire un caractère (sous-chaîne) commence à 0, **négatif** pour partir de la fin  
a[0] -> 's'  
a[2] -> 'a'  
a[-1] -> 'm'

# Objets et nommage

- On peut associer un nom à un objet

```
>>> a = 42
>>> nom = 'spam'
>>> a + 1
43
>>> nom
'spam'
>>> b = a
```



- Le nom est **déréférencé** (l'objet est obtenu) dans une expression
- On parle d'affectation de variables habituellement, mais c'est un abus de langage : en Python il n'y a que des *noms* liées à des *références* et pas de variables au sens de C, Perl, ...
- Le nom est une *référence* vers un objet
- Un objet est un espace en mémoire avec un *type* et une *valeur*
- On peut combiner les opérateurs et l'affectation :  

```
>>> a += 1. # Idem a = a + 1
```

Adresse 0x03ea345  
Type: str  
'spam'

# Booléens (True/False)

- On peut comparer deux objets avec l'opérateur ==
- Compare les *valeurs* des objets concernées qui peuvent être *différents* (et même de différents types)

```
1 == 1
1.0 == 1
(1j) ** 2
(1j) ** 2 == -1
```

- type() renvoie le *type* d'un objet, id() renvoie son adresse en mémoire dans la PVM
- L'opérateur *is* vérifie l'identité de deux objets (en fait UN objet) (même adresse mémoire)

```
a = 1 ; b = 1.0
a is 1 ; a == b ; a is b
```

- Si même adresse en mémoire, nécessairement même type et valeur - ou pas ?  
a = float('nan') # Not A Number IEEE803 norme sur le calcul flottant  
b = a  
a == b (False)  
a is b (True) (Quoi ?!)

- Vidéo : <https://www.destroyallsoftware.com/talks/wat>

# Une note sur les types

- En Python type de donnée et classe c'est LA MÊME CHOSE
- Un type est aussi un objet (TOUT est un objet)
- **C'est aussi une fonction**, qualifié de *constructeur* du type en question, pour les types intégrés ce sont des fonction de conversion :

```
int('42') # comment convertir si la base
           # n'est pas 10 ? help(int)

str(42)
int('42')
int('234', base=8)
int('42abcd')
```

# Polymorphisme

- Une opération peut échouer sur un problème de type

```
>>> 'spam' + 1
```

- Il faut donc demander explicitement les conversions

```
>>> 'spam' + str(1)
```

- On peut additionner deux chaînes : concaténation

```
>>> 'spam' + ' ' + 'ham'  
'spam ham'
```

- + pour les chaînes existe bien on trouve `__add__` dans `dir(str)`

- On trouve `__mul__` aussi : `help(str.__mul__)`

```
>>> 'spam' * 7  
'spam spam spam spam spam spam spam '
```



# Tranches

- Tranches de chaînes (valable aussi pour les autres séquences) :

```
>>> s = 'SPAM_IS_BETTER'
      0   1   2   3   4   5   6   7   8   9 10   11   12 13
      S   P   A   M   _   I   S   _   B   E  T   T   E   R
-14 -13 -12 -11 -10 -9  -8  -7  -6  -5  -4  -3  -2  -1
>>> s[2:10]
'AM_IS_BE'
>>> s[:4]
'SPAM'
>>> s[5:]
'IS_BETTER'
>>> s[-6:]
'BETTER'
>>> s[2:10:2]      # debut:fin:pas
'A_SB'
>>> s[3::-1]
'MAPS'
```

# Les tests : *if*

- L'instruction if attend une expression qui sera convertie en booléen (bool()) et exécute conditionnellement du code
  - bool(0) est False, bool(42) est True, bool('') est False et bool('ham') est True, bool('0') est True aussi
- On peut combiner les tests avec and / or / not  
Un bloc est défini par ':' suivi d'une avancé d'indentation

```
if a == b:
    # c'est le changement d'indentation qui
    # fait le bloc
    print("Oui c'est pareil")
elif a > b: # pas obligatoire
    print("C'est plus grand")
else: # pas obligatoire
    print("C'est plus petit")
print("Suite du code, toujours exécuté")
```

# Les blocs : *off side rule*

- Les blocs de code en Python sont toujours définis par l'*indentation* !
- 4 espaces sont la valeur recommandée et suivie par les éditeurs sinon configurez les ainsi (en particulier la touche tabulation)
- Pour définir un bloc vide (aucun code exécuté) utilisez l'instruction *pass*

```
if error:
    pass # TODO
else:
    ...
```

- Dans l'interpréteur **interactif** pour indiquer qu'on a finit un bloc quand il n'y a pas de suite simplement **passez à la ligne** à l'invite ... (trois point)
- Dans un script on peut avoir des lignes blanche où on veut ça n'a pas d'impact

# **Exercice 1.1**

## **Chaînes de caractères et tests**

# Formater des chaînes

- Contaténer des données pour obtenir une chaîne bien formée est pas toujours pratique :  
`"l'âge de " + name + " est " + str(age)`
- La méthode `format()` est conçue pour faciliter cela :  
`"l'âge de {} est {}".format(name, age)`
- Entre accolades on peut indiquer le rang dans la liste d'argument, des indications de type et de largeur  
`"le mur {1} fait {0:.2f} mètres".format(12.3, 'nord')`
- Il existe aussi l'opérateur `%` (obsolète, mais disponible en Python 2 et 3)  
`"l'âge de %s est %d" % (name, age)`
  - Compatible à 100% avec la fonction C *sprintf*
- Dans les dernières versions de Python 3 une autre façon est proposée : le préfixe `f`  
`name = 'John'; oldage=42`  
`f"The age of {name} is {oldage+1}"`

# Plus d'opérations sur les chaînes et les nombres

- Deux modules de la bibliothèque standard fournissent plus d'opérations sur les nombres et les chaînes : *string* et *math*

```
>>> import math
>>> math.sqrt(42)
...
>>> dir(math)
>>> import string
>>> dir(string)
>>> string.capitalize('spam is good')
'Spam Is Good'
>>> help(math)
>>> help(string.capitalize)
```

- Plus de détails sur les modules et les diverses façons d'en importer des fonctionnalités plus tard...

# Modifiable ou immuable ?

- En Python il est important de distinguer les types *modifiables* des types *immuables*
- Tous les types numériques, les chaînes de caractères, les booléens sont **IMMUABLES**

```
>>> 42 + 1
43    # trois objets distincts sont en
      # mémoire : 42, 1 et 43
```

```
>>> s = 'spam'
```

```
>>> s[0] = 'S'
```

Erreur ... ne peut modifier une chaîne

```
>>> s = s[0].upper() + s[1:]
```

s référence maintenant un **NOUVEL** objet

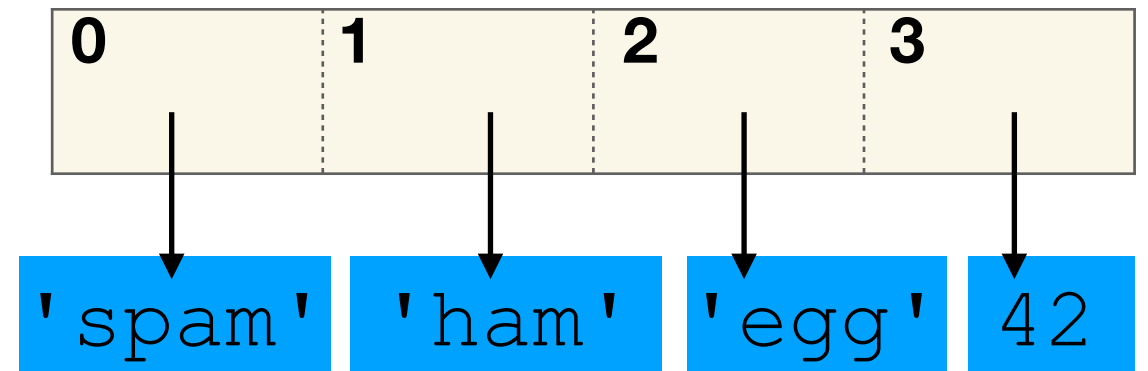
- Types de données numériques et chaînes
- **Collections**
- Boucles
- Fonctions
- Compréhensions



# Collections : Listes

- Une liste contient comme éléments des **références** vers des objets Python (quelconques)

```
>>> myList = [ 'spam', 'ham', 'egg', 42 ]  
>>> myList[1]  
'ham'
```



- Une liste est **modifiable** :

```
>>> myList[1] = 'bacon'  
>>> myList  
[ 'spam', 'bacon', 'egg', 42 ]
```

# Tranches de listes

- L'extraction de sous-listes par tranche suit exactement le même principe que pour les chaînes :

```
>>> myProducts = [ 'Soup', 12, 'Bread', 3 ]
>>> myProducts[::2]
[ 'Soup', 'Bread' ]
>>> myProducts[1::2]
[12, 3 ]
```

- Les tranches sont modifiables

```
>>> myList = [ 'spam', 'ham', 'bacon', 'egg', 42 ]
>>> myList[1:3]
['ham', 'bacon']
>>> myList[1:3] = ['orange', 'croissant', 'beurre']
>>> myList
['spam', 'orange', 'croissant', 'beurre', 'egg', 42]
```

# Méthodes sur les listes

- Beaucoup de méthodes, en particulier pour les modifier : list.**append**, list.push, list.pop, list.find, etc.

```
>>> dir(list)
>>> help(list.pop)
>>> myL = [1,2,42]
>>> myL.pop()
42
>>> myL
[1,2]
```

- *len()* est une fonction qui renvoie la longueur de n'importe quelle collection (chaînes, listes et les autres aussi)
- Une méthode intéressante *sur les chaînes* est *split()* : elle renvoie une *liste*  

```
>>> 'hello my old friend'.split()
['hello', 'my', 'old', 'friend']
```
- L'inverse de *split* devrait être une méthode *join*. Assez finement c'est une méthode de chaîne, qui est le séparateur !  

```
>>> '_'.join( [ 'ici', 'une', 'liste' ])
'ici_une_liste'
```

# Opérations sur les listes

- Similaires à celles sur les chaînes de caractères

- Addition

```
>>> [ 'egg', 'spam' ] + [ 'bacon', 'spam' ]  
[ 'egg', 'spam', 'bacon', 'spam' ]  
>>> [ 'spam', 'egg' ] * 3  
[ 'spam', 'egg', 'spam', 'egg', 'spam', 'egg' ]
```

- Comparaison : applique == sur les éléments

```
if myOrder == [ 42, "spam" ]:  
    print("You've ordered 42 pieces of spam")
```

# Collections : tuples

- Un tuple est similaire à une liste (collection ordonnée d'objets) :

```
>>> myBkfst = ( 'spam', 'egg', 'bacon' )  
>>> myBkfst[1]  
'egg'
```

- Les parenthèses ne sont là que pour mettre en évidence la séquence, ce sont les virgules qui construisent un tuple :

```
>>> myBkfst = 'spam', 'egg', 'bacon'  
>>> myBkfst[1]  
'egg'
```

- Comment construire un tuple à un seul élément ? Quelle est le type de *myBkfst* ci-dessous ?

```
>>> myBkfst = ( 'spam' ) # une chaîne...
```

# Tuples

- Pour construire un tuple à **un** élément ajouter une virgule à la fin:

```
>>> myBkfst = 'spam',          # ou encore :  
>>> myBkfst = ( 'spam', ) # Plus lisible !
```

- Pour construire un tuple vide juste les parenthèses :

```
>>> emptyTuple = ( )
```

- Toutes les opérations en lecture sont IDENTIQUES à celle des listes (sélection d'élément, tranches)

```
>>> myBkfst = ('spam', 'egg', 'bacon', 'sausage', 'ham')  
>>> myBkfst[1]  
'egg'  
>>> myBkfst[2:4]  
( 'bacon', 'sausage' )
```

- Les tuples sont **IMMUABLES** :

```
>>> myBkfst[2] = 'spam'  
Erreur...
```

# Tuples et listes

- Comme TOUJOURS en Python le type (la classe) est une fonction **constructeur**, les fonctions tuple et list prennent les éléments de toute autre collection
- Convertir tuples en listes et vice-versa est simple :

```
>>> myBkfst = tuple( 'egg-pain beurre-spam'.split('-') )
>>> prices = ( 42, 'spam', 12, 'bread' )
>>> list(prices)
[ 42, 'spam', 12, 'bread' ]
```
- Une liste ou un tuple peuvent contenir des tuples, des listes, tout objet Python...

```
>>> myBkfst = [ ( 'spam', 42), ( 'egg', 21) ]
>>> myBkfst[1] = ( 'bread', 3 )
```
- Une chaîne est *aussi* une collection (celle de ses caractères) :

```
>>> list('spam')
[ 's', 'p', 'a', 'm' ]
```

# Déballage de séquences

- Tuples et listes sont tous deux des séquences (collections ordonnées d'objets), leurs tranches aussi
- On peut assigner tous les éléments d'une séquence en une fois (les nombres d'éléments doivent correspondre !) :

```
>>> myData = [ 42, 'spam', 12 ]  
>>> price, product, quantity = myData  
>>> print(price, product, quantity)  
42 spam 12
```

- Comment permuter deux « variables » ?

```
a, b = b, a                # Ouah !
```



# Plus fort encore : déballage imbriqué de séquences

Cela fonctionne aussi à plusieurs niveaux :

```
>>> myData = [ 42, ('spam','ham'), [1,2] ]
>>> a, (s1,s2), (n1,n2) = myData
>>> a
42
>>> s2
'ham'
>>> n1
1
```

# Déballage élastique (déballage et remballage)

- Penser à utiliser des tranches si le nombre d'éléments ne correspond pas

```
>>> myData = [ 42, 'spam', 12, 0 ]
>>> price, product, quantity = myData[:3]
>>> print(price, product, quantity)
42 spam 12
```

- On peut indiquer **une** séquence pour récupérer une quantité inconnue d'éléments avec une étoile (Python 3 seulement) :

```
>>> price, product, *rest = myData
>>> rest
[ 12, 0 ]
```

- Au début, au milieu, à la fin mais *une seule fois*. On peut la nommer `_` si son contenu ne nous intéresse pas :

```
>>> first, *_, last = myData
```

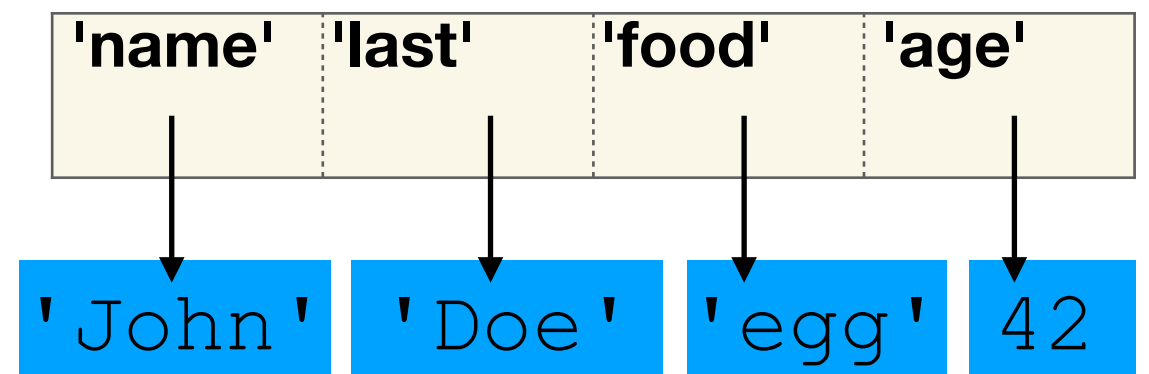
# Dictionnaires

- Dictionnaire : association de clés *presque* quelconques à des objets quelconques

- Les clés peuvent être de n'importe quelle type **immuable**

```
>>> myData = { 'name': 'John', 'last': 'Doe',  
               'food': 'egg', 'age': 42 }
```

```
>>> myData['last']  
'Doe'
```



- Ils sont modifiables :

```
>>> myData['age'] += 1
```

```
>>> myData['city'] = 'Nantes'
```

# Construction de dictionnaires

- Une suite de *cle:valeur* entre accolades

```
>>> myData = { 'name': 'Doe', 'age':42 }
```

- Le constructeur *dict()* attend une séquence de couples :

```
>>> myData = dict( [ ('name','Doe'), ('age',42) ] )
```

- ... ou encore des paramètres passé par mot-clés :

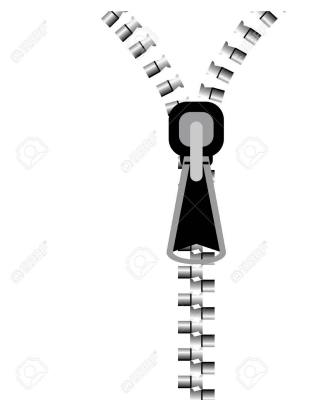
```
>>> myData = dict( name='Doe', age=42 )
```

- Utilisez *zip()* si vous avez clés et valeurs *séparément* :

```
>>> data1 = ['name', 'age']
```

```
>>> data2 = ['Doe', 42]
```

```
>>> myData = dict( zip(data1, data2) )
```



# Méthodes de dictionnaires

- Beaucoup de méthodes d'accès et de modification

```
>>> myData['color']  
... KeyError: 'color'  
>>> myData.get('color', 'n/a') # 'n/a' si clé absente  
                                # elle n'y est pas  
'n/a'
```

- On peut ajouter une nouvelle entrée (clé inexistante) :

```
>>> myData['color'] = 'red'  
>>> myData.get('color', 'n/a') # 'n/a' si clé absente  
                                # elle est là !  
'red'
```

- Le second argument de *get()* est optionnel (sa valeur par défaut est *None*, objet de type *NoneType*)

```
if myData.get('color') is None:  
    myData['color'] = 'red' # No! Blue!
```

# Dictionnaires et séquences

- Des méthodes permettent de récupérer les séquences de clés, de valeurs, de couples (clé, valeur)

```
>>> myData.keys()  
>>> myData.values()  
>>> myData.items()
```

- En Python 2 ce sont des listes de tuples, en Python 3 des *vues* qui restent à jour, occupent peu de place en mémoire et se traitent de la même façon (itérables)

```
[ 'name', 'last', 'age', ... ]  
[ 'John', 'Doe', 42, ... ]  
[ ('name', 'Joe'), ('last', 'Doe'),  
  ('age', 42) , ... ]
```

# Ensembles

- Collection d'objets tous différents (pas comme une liste) et non ordonnés (type *set*)

```
>>> bkfst = { 'spam', 'ham', 'egg', 'ham' }  
>>> bkfst  
{ 'egg', 'ham', 'spam' }
```

- **Modifiable** (méthodes *add()*, *update()*, *remove()* )
- Opérations ensemblistes : intersection (&), union (|), différence (-), ...
- Pratique pour supprimer des doublons dans des listes :  

```
>>> list_of_products += new_products  
>>> list_of_products = list(set(list_of_products))
```
- Le type *frozenset* est similaire mais **immuable**

# Opérateur *in*

- Fonctionne avec TOUTES les collections (chaînes, listes, tuples, dictionnaires, ensembles, ...), renvoie *True* ou *False*.
- Teste la présence d'un objet *égal* à celui cherché pour les collections générales (celles qui contiennent des objets variées : listes, tuples, clés pour un dictionnaire)
- Il y a aussi l'opérateur *not in*
- Teste la présence d'une *sous-chaîne* pour les *chaînes* (comme `str.find()`)

```
>>> 'ham' in [ 'spam', 'ham', 'egg' ]           # True
>>> 'ham' in { 'ham':34, 'spam':42, 'egg':12 }  # True
>>> 'needle' in 'a needle in a haystack'       # True
>>> 'needle' not in 'a needle in a haystack'   # False
```



- Types de données numériques et chaînes
- Collections
- **Boucles**
- Fonctions
- Compréhensions

# Protocole d'itération

- Un protocole est un ensemble de règles qu'un objet Python peut observer
- Le protocole d'itération consiste à :
  - Renvoyer un objet sur la fonction *iter()* (méthode spéciale `__iter__`)

```
>>> it = iter([1, 2])
```
  - cet objet peut être l'argument de *next()* et renvoie un élément

```
>>> next(it)
```

```
1
```
  - s'il n'y a rien à renvoyer l'exception *StopIteration* est levée (pas obligé)

```
>>> next(it)
```

```
2
```

```
>>> next(it)
```

```
...
```

```
StopIteration
```
  - Que renvoient une liste, un tuple, une vue, un dictionnaire ?  
(*éléments, éléments, éléments, clés*)

# Boucle for

- La boucle *for ... in ...* utilise le protocole d'itération

```
for food in ['spam', 'egg', 'bacon', 'pudding']:
    if food == 'bacon':
        break
    if food == 'egg':
        continue
    print(food)
else:
    print('No bacon today')
```

- On peut sauter à l'étape suivante avec *continue*
- On peut interrompre la boucle avec *break*
- Une clause optionnel *else* exécute du code si on est **pas** sorti par un *break*
- **ATTENTION** : un itérateur qui **ne** lève **pas** *StopIteration* conduit à une boucle infinie !

# Boucle for et déballage

- *for* réalise des affectations successives, similaire à :

```
food = next( ... )
```

- On peut déballer une *séquence* à ce moment là si les *éléments* sont des séquences (ici des tuples) :

```
for food,price in [('spam',42), ('egg', 12)]:  
    print('Le prix de {} ' \  
          'est {}'.format(food,price))
```

- Les déballages peuvent s'imbriquer :

```
data = {'a01':('spam',42), 'a02':('egg',12)}  
#items() -> [ ('a01',('spam',42)) , (a02,( ... )) ... ]  
for ident,(food,price) in data.items():  
    print('Identifiant {} : nom = {} ' \  
          'prix = {}'.format(ident,food,price))
```

# Énumération

- La fonction `range()` (`xrange` en Python 2) fournit un **itérateur** entre `n` et `m` (`m` non-inclu) avec un certain pas :

```
for i in range(10): # va de 0 à 9
    print(i)
for i in range(5,12,2): # va de 5 à 11 de 2 en 2
    print(i)
```

- On peut souhaiter énumérer nos entrées dans une boucle :

```
i = 1
ingredients = 'spam ham egg bacon'.split()
for food in ingredients:
    print('{}\t{}'.format(i,food))
    i += 1 # So old school...
```

- On peut faire mieux :

```
for i,food in enumerate(ingredients,1):
    print('{}\t{}'.format(i,food))
```

- Que fait `enumerate()` ?

```
>>> list(enumerate(['spam','ham','egg']))
[ (0, 'spam'), (1, 'ham'), (2, 'egg') ]
```

# Plus sur les itérateurs

- Un module nommé *itertools* fournit des fonctions utiles pour construire des itérateurs

```
>>> import itertools  
>>> dir(itertools) # pour voir...
```

- *chain()* fusionne deux itérateurs sans stocker tout en mémoire (comme le ferait + sur deux listes)

```
from itertools import chain  
bkfst = ['ham', 'spam']  
lunch = ['lamb', 'potatoe']  
for food in chain(bkfst, lunch):  
    print(food)
```

# Boucle while

- La boucle *while* vérifie juste une condition pour sortir du corps de la boucle

```
index = 0
while index < 10: # même chose que for index in range(10)
    index += 1
    if index == 8:
        break
    if index == 5:
        continue
    print(index)
else:
    print('No 8 found')
```

- On peut sauter à l'étape suivante avec *continue*
- On peut interrompre la boucle avec *break*
- Une clause optionnel *else* exécute du code si on est **pas** sorti par un *break*

# **Exercice 1.2**

## **Collections, Boucles et dictionnaires**



- Types de données numériques et chaînes
- Collections
- Boucles
- **Fonctions**
- Compréhensions

# Fonctions

- On définit une fonction par un bloc de code qui a un nom qui *va référencer* une *fonction* qui attend d'éventuels arguments
- Une fonction appelée renvoie **toujours** quelque chose, implicitement c'est *None*

```
from math import sqrt
```

```
def length(x,y):  
    return sqrt(x**2 + y**2)
```

```
print(length(2,3) + length(4,5))  
print(length(1,2,3)) # erreur  
print(length(1))      # erreur
```

# Paramètres

- On peut passer les paramètres par position

```
length(2, 3)
```

- On peut passer les paramètres par mot-clés

```
length(x=1, y=2)
length(y=2, x=1) # Pareil !
length(1, z=2) # Erreur ! (z est inconnu)
```

- Si vous panachez : par position d'abord, par mot-clés les suivants :

```
>>> print('ham', 'spam', 'egg', sep='-', \
          end=' END\n')
ham-spam-egg END
```

# Paramètres optionnels

- Des paramètres peuvent être optionnels mais dans ce cas il **faut** spécifier leurs valeurs par défaut
- Si il n'y a pas de valeur évidente (*0, False, '', []*) on utilise volontiers *None* (attention à tester la valeur par *is / is not None* si *False, chaîne vide, 0, [],()*, etc. sont possibles et légitimes !)

```
def length(x, y, debug=False) :  
    if debug:  
        print('DEBUG length : ', x, y)  
    return sqrt(x*x + y*y)
```

```
r    = length(1, 3)                # comme si (1, 3, False)  
l1   = length(1, 3, debug=True)    # par mot-clé  
l2   = length(1, 3, True)          # par position
```

# Documentez vos fonctions

- Commencez par la documentation !

```
def length(x, y, debug=False) :  
    '''Calcule la longueur d'un vecteur en  
        géométrie euclidienne. debug permet  
        d'afficher les arguments reçus.'''  
    if debug:  
        print('DEBUG length :', x, y)  
    return sqrt(x*x + y*y)
```

- `help(length)` va afficher le prototype de la fonction suivi de ce texte

# Règles de résolution des noms

- La règle de base de résolution d'un nom pour le déréférencer en tant qu'objet (lecture) est LEGB :
  - **L**ocal (fonction, classe)
  - **E**nclosing (fonction englobante)
  - **G**lobal (script ou module complet)
  - **B**uilt-in (interne à Python)
- Si vous affectez une "variable" (association d'un nom à une référence) dans une fonction cette affectation n'est visible que **localement** (dans notre espace de nom)
- Conclusion : Si vous *modifiez* un objet reçu en argument la modification est visible, si vous réaffectez elle ne l'est pas
- Il existe une instruction *global* pour outrepasser la règle LEGB (ne l'utilisez JAMAIS)

# Suite variable d'arguments

- Si vous voulez accepter une suite de longueur variable d'arguments passés par position il faut le spécifier à la déclaration de la fonction :

```
# args est un nom habituel ici
def showargs(*args):
    for i,arg in enumerate(args):
        print('Argument ',i,arg)
```

```
showargs(1,2,3,4,5,'ham')
```

- `args` collecte dans une séquence (tuple) les arguments reçus
- On peut combiner avec des arguments spécifiés

```
def showinfo(x,*args):
    ...
```

  - *x doit* être passé, les autres sont optionnels
- On *emballe* dans le tuple **args** les *arguments* supplémentaires reçus

# Arguments quelconques par mot-clés

- Une fonction peut être définie de façon à accepter une suite quelconque d'arguments par mot-clés (noms inconnus à l'avance)
- Vous les collectez dans un dictionnaire contenant les noms comme mot-clés et les valeurs associées

```
# kwargs est le nom habituel ici
def showargs(**kwargs):
    for name, val in kwargs.items():
        print('Argument ', name, '=', val)

showargs(nom='John', age='42', debug=True)
```



# Combiner tout ça...

- Vous pouvez panacher arguments obligatoires, optionnels, quelconques par position et quelconques par mots-clés

```
def showargs(x, y, dbg=False, *args, **kwargs) :  
    print('x =', x, 'y= ', y, 'dbg = ', dbg)  
    print('args = ', args)  
    print('kwargs = ', kwargs)
```

- Une définition comme ci-dessous définit une fonction **générique** (accepte toute forme d'arguments possible) :

```
def showargs(*args, **kwargs) :  
    print('  args = ', args)  
    print('kwargs = ', kwargs)
```

# Forcer le passage par mot clef ou par position

- En Python 3 on peut indiquer que les arguments par positions sont interdits à partir d'une certaine position

```
>>> def length(x, y, *, debug=False):  
...     if debug:  
...         print('DEBUG : ', x, y)  
...     return sqrt(x**2 + y**2)  
...  
>>> length(2, 4, True) # Erreur, debug par position est interdit  
>>> length(2, 5, debug=True) # Ok
```

- Avec les plus récentes versions de Python 3 (3.8 et ultérieures) on peut aussi obliger le passage d'arguments par position :

```
>>> def length(x, y, /): # x, y par position seulement  
...     # code  
...  
>>> length(1, y=42)  
TypeError: length() got an unexpected keyword argument 'y'
```

- On peut combiner les deux marqueurs / et \* dans une définition de fonction  
def f(pos1, pos2, /, pos\_ou\_kwd, \*, kwd1, kwd2):

# Déballage de collections dans des arguments

- On a souvent besoin de *déballer* des éléments d'une collection comme arguments d'une fonction

```
def length(x,y,debug=False):  
    ...  
vector = (3,4) # tuple ou liste, peu importe (séquence)  
length(vector[0],vector[1]) # Old school!  
length(*vector)             # Better, isn't it?  
data = (3,4,'vecteur vitesse') # Too much information...  
length(*data[:2])           # Nice! Defensive programming!
```

- De même on peut déballer un dictionnaire comme arguments passés par mots-clés :

```
def insertData(name,age,food): # Insère une personne en base  
    ...  
myData = { 'age':42, 'name':'John', 'food':'spam' }  
insertData(**myData)
```

# **Exercice 1.3**

## **Fonctions**

# Paramètres et référence

En Python les noms sont toujours des *références* vers des objets, les arguments de fonctions aussi

```
def morespam1(the_list):  
    the_list = the_list + ['spam'] # Nouvelle liste  
    return the_list
```

```
def morespam2(the_list):  
    the_list.append('spam') # Modifie la liste reçue  
    return the_list
```

```
myBkfst = [ 'egg', 'sausage' ]  
print(morespam1(myBkfst)) # [ 'egg', 'sausage', 'spam' ]  
print(myBkfst) # [ 'egg', 'sausage' ]
```

```
print(morespam2(myBkfst)) # [ 'egg', 'sausage', 'spam' ]  
print(myBkfst) # [ 'egg', 'sausage', 'spam' ]
```

# Générateurs

- Un *générateur* est une fonction renvoyant un *itérateur* (utilisable dans une boucle, par exemple) c'est-à-dire un objet obéissant au *protocole d'itération*
- Contient l'instruction **yield** (*relâcher*, réponse à **next()**), l'itérateur est l'objet *renvoyé* par l'appel à la fonction. Sortir de la fonction (*return* ou implicite) lève **StopIteration**

```
# Attention ! Itérateur infini si end n'est pas spécifié !
def genSquares(start=0, end=None):
    while True:
        # on pourrait sortir ici au lieu du return
        yield start**2 # Répondra à next()
        start += 1
        if end is not None and start >= end:
            return # Lèvera StopIteration, pas obligatoire...

sqrsto5 = genSquares(end=5) # l'itérateur est généré ici
print(next(sqrsto5))
print(next(sqrsto5))
for n in sqrsto5: # Attention d'être sûr qu'il se termine !
    print(n)
```

# **Exercice 1.4**

## **Générateurs**

# Fonctions *lambda*

- Une fonction est un objet comme un autre, juste défini avec un nom par *def* et *appelable* (i.e. *nom\_de\_fonction(arguments...)* fonctionne)
- Il est possible de décrire une fonction *simple* (qui renvoie une expression, pas de blocs de code) sans utiliser ni instruction *def* ni *return...* sans nom

```
>>> from math import sqrt
>>> f = ( lambda x,y: sqrt(x**2 + y**2) )
>>> f(3,4)
5.0
>>> ( lambda n,s: s + str(n) ) (42, 'spam')
'spam42'
```

- Utile pour définir *rapidement* une fonction simple, passer une fonction en argument à une autre fonction (*callbacks* ou *functools.reduce*), renvoyer une fonction *simple*, mettre des fonctions dans une *collection* sans se donner la peine de lui *donner un nom*, exprimer des fonctions dans des *compréhensions*, etc.



# Fonctions recevant des fonctions en paramètre

- Vous pouvez écrire des fonctions attendant des fonction en paramètres (application : décorateurs)
- Il en existe intégrés ou dans des modules standards : *map*, *filter*, *functools.reduce*
- On peut aussi renvoyer une fonction comme résultat (application : décorateurs)

```
def logfunc(anyfunc):  
    def _(*args,**kwargs):          # _ ou autre, peu importe !  
        print('appel de', anyfunc.__name__)  
        return anyfunc(*args,**kwargs)  
    return _
```

```
@logfunc  
def myFunc(...):  
    ...
```

Dès lors, une expression *myFunc(...)* évaluera en réalité *logfunc(myFunc)(...)*

- Types de données numériques et chaînes
- Collections
- Boucles
- Fonctions
- **Compréhensions**

# Une autre façon de parcourir une séquence

- Une autre forme de *for ... in ...* qui n'est pas une instruction : c'est une construction fonctionnelle construisant un objet !
- *Liste en compréhension* :

```
>>> bkfst = [ 'spam', 'ham', 'egg' ]  
>>> [ food.upper() for food in bkfst ]  
[ 'SPAM', 'HAM', 'EGG' ]
```

- *Générateur en compréhension* (évaluation *paresseuse*) :

```
>>> bkfst = [ 'spam', 'ham', 'egg' ]  
>>> it = ( food.upper() for food in bkfst )  
...  
>>> for food in it:      # évalués à la demande  
    print(food)
```

# Flirter/Tester en toute compréhension

- On peut *filtrer* les éléments pris en compte avec *if* en position finale

```
>>> bkfst = [ 'spam', 'ham', 'egg' ]  
>>> [ food.upper() for food in bkfst if food != 'egg' ]  
[ 'SPAM', 'HAM' ]
```

- On peut faire des tests dans une expression (« *if* fonctionnel », *else* obligatoire) :

*expression\_si\_vrai if condition else expression\_si\_faux*

```
>>> bkfst = [ 'spam', 'ham', 'egg' ]  
>>> [ food.upper() if food != 'spam' else 'POUAHAHAH'  
      for food in bkfst if food != 'egg' ]  
[ 'POUAHAHAH', 'HAM' ]
```

- Cette forme de *if* (similaire à l'opérateur ternaire de C, PHP, Perl, ...) est souvent utile en dehors des compréhensions :

```
s = 'Commande de {} produit{}'.format(n, 's' if n>1 else '')  
msg = 'cheese' if not vegan else 'no cheese'
```

# Autres compréhensions

- Cette forme fonctionne aussi pour les dictionnaires entre accolades...

```
>>> myData = dict(ham=12, spam=42, egg=21)
>>> meats = [ 'ham', 'egg' ]
>>> { k:v for k,v in myData.items() if k in
      meats }
{ 'ham':12, 'egg':21 }
```

- Pour les ensembles : accolades mais *pas* d'expression *clé:valeur*

```
>>> { v for k,v in myData.items() if k in meats }
{ 12, 21 }
```

- Syntaxe identique (filtrage avec *if*, alternatives avec *... if ... else ...*)

# Compréhensions vs code impératif

- Le code suivant ( $f, g$  : traitement,  $alt$  : alternative,  $cond$  : condition de filtrage) :

```
res = [ f(x) if alt(x) else g(x) for x in coll
        if cond(x) ]
```

- Est équivalent à :

```
res = []
for x in coll:
    if cond(x):
        if alt(x):
            res.append(f(x))
        else:
            res.append(g(x))
```

- Lequel préférez-vous ?

# Activité : compréhensions

- Reprenez (copiez le fichier) le résultat de l'exercice précédent, comment utiliser plutôt des compréhensions en lieu et place des instructions *for* + bloc ?
- **Réflexion** : Quand éviteriez-vous l'usage de compréhensions en lieu et place de boucles classiques ?

## Éviter les compréhensions ?

- Si c'est moins lisible (oui mais on peut faire des compréhensions successives et utiliser *itertools* au lieu de les imbriquer ou de mettre plusieurs *for* à la suite)
- Si vous construisez plus d'une séquence à partir d'une seule et que vous voulez le faire en une seule boucle (on peut avec des tuples et deux générateurs en compréhension)
- Si c'est moins lisible pour *vous*
- Si vous êtes allergique à la programmation fonctionnelle
- **Si leur évaluation a des effets de bords (*side effect*, effet secondaire) !!! (Un *print provisoire* pour déboguer est ok) DON'T DO THAT!**



# Programmation fonctionnelle

- Python n'est pas un langage fonctionnel *pur* (comme le sont LISP, Scheme, CaML, Haskell, F#, ...)
- Il intègre des concepts fonctionnels (fonctions comme objets, itérateurs, compréhensions, lambdas)
- Il en viole des aspects clés, et c'est assumé :
  - Lambdas limitées (pas de blocs)
  - Distinction instructions/expressions
  - Pas de récursivité terminale (récursivité classique bien sûr)

# Ontologie des types Python

<i>Collection ?</i>  <i>Immuable ?</i>	<b>Pas collection</b> <b>(pas d'éléments)</b>	<b>Collection</b> <b>(éléments)</b> <i>Séquence</i>	<b>Collection</b> <b>(éléments)</b> <i>Pas séquence</i>
<b>Immuable</b>	<code>int, float,</code> <code>complex, bool,</code> <code>NoneType</code>	<code>str, tuple</code>	<code>frozenset</code>
<b>Modifiable</b>	<i>Vos classes</i> <i>« métiers »,</i> <i>probablement</i>	<code>list</code>	<code>dict, set</code>

# Caractères ouvrants/ fermants

- ( ) servent à ...
  - grouper une expression (règles de priorité) : `42 * ( 2 + 3 )`
  - appels de fonction : `dir(whatever)`
  - faire joli autour d'un tuple : `date = ( 1, 2, 3, 4 )`
  - encadrer un générateur en compréhension  
`( x for x in ... if ... )`
  - souvent autour des *lambdas* pour regrouper : `( lambda x: x+1 )`
- { } servent à ...
  - étiquettes pour *str.format()* (dans une chaîne) :  
`'to {0} or not to {0} that is the question.'.format('be')`
  - dictionnaires littéraux `{ 'a':1 , 'b':2, ... }` ou en compréhension
  - ensembles : `{ 1, 2, 42 }` ou en compréhension
- [ ] servent à ...
  - listes *littérales* `[1, 2, 3, 42]` ou en compréhension `[ x + 1 for x in ... if ... ]`
  - *sélection* d'un élément ou d'une tranche dans une *collection* (sauf les ensembles) :  
`myList[4], myTuple[2], myDico['name']`

# D'autres collections ?

- Il existe d'autres types fournis par des modules, en particulier dans la bibliothèque standard le module *collections*, comme *defaultdict*, *namedtuple*, *Counter*, ...
- Il fournit des variation sur le thème des séquences et des dictionnaires

```
>>> import collections
>>> help(collections)
>>> dir(collections)
```

- Parmi ces nouveaux types de données on trouve *defaultdict* (dictionnaire avec valeur par défaut)

```
>>> from collections import defaultdict
>>> help(defaultdict)
>>> food = defaultdict( lambda: 'spam' ) # On lui passe une fonction
>>> food
>>> print(food.get('Brian')) # C'est un dictionnaire vide ...
None
>>> food['Brian']                # ... qui se remplit tout seul !
'spam'                         # en appelant la fonction
>>> food.get('Brian')
'spam'
>>> food
```

# Application : compter encore les mots

- Reprenez le code de l'exercice 1.2, partie A où l'on comptait le nombre d'occurrences des mots d'un texte
- Comment se comporte une structure `defaultdict(int)` lorsqu'on lit une entrée absente ?

```
>>> int() # constructeur appelé sans argument
0
>>> d = defaultdict(int)
>>> d['be']
>>> d
0
>>> d['to'] += 1
>>> d
1
```

- On peut donc compter les mots d'un texte encore plus simplement !

```
occurrence = defaultdict(int) # 0 par défaut
for word in words:
    occurrence[word] += 1
```

- Encore plus simple : la classe *Counter* du module *collections* !



# Chapitre 2

# Classes, objets et modules

Définition de classes, instances, méthodes, attributs d'instances et de classe, héritage, surcharge d'opérateurs, classes abstraites, création de modules, modules de la bibliothèque standard

Jean-Pierre MESSEAGER

`jp@xiasma.fr`

v2.0a - février 2020

**Attribution - Pas d'Utilisation Commerciale - Pas de Modification**

**CC BY-NC-ND**



- **Définition de classes et instanciation**
- Héritage
- Surcharge d'opérateurs
- Décorateurs et méthodes abstraites
- UML et patrons de conception
- Modules
- Tests unitaires



# Terminologie Programmation Orientée Objet

- Plutôt de manipuler des données avec des fonctions on met ensemble les deux
- **Classe** (type) : patron de conception qui décrit les opérations possibles sur une données (str)
- **Instance** : un objet donné d'une certaine classe avec ses données à lui ('spam')
- **Attribut** : donnée ou une méthode. En Python c'est un *nom* dans l'espace de noms (classe ou instance)
- **Attribut** d'instance : propriété d'une instance donnée
- **Attribut** de classe : propriété partagée par toutes les instances car accédé comme tel
- **Méthode** : fonction qui agissent ou renvoient des information pour une instance donnée (les méthodes sont des attributs de classe) *upper, lower, etc.*
- **Classe abstraite** : classe qui définit uniquement les noms des certaines méthodes mais pas leur contenu effectif
- **Héritage** : une classe hérite d'une autre si elle récupère tous ses attributs (méthodes comprises) et en redéfinit ou en ajoute de nouveaux (relation *être un*)
- **Composition** : une instance d'une certaine classe peut être un attribut d'une instance d'une autre classe (relation *avoir un*)

# Définissons une classe vide

```
>>> class MyClass:
    pass

>>> MyClass
...
>>> myO = MyClass() # instance !
>>> myO
...
>>> type(myO)
>>> type(myO) == MyClass
True
>>> myO.food
... Erreur
>>> MyClass.food = 'spam' # zut j'ai oublié un attribut
>>> myO.food # les instances le récupèrent
'spam'
>>> myO.food = 'ham' # l'instance change l'attribut
>>> MyClass.food # ça change rien pour la classe
'spam'
>>> myO.food
'ham'
>>> MyClass.beverage = 'coffee' # j'ai oublié un autre attribut
>>> myO.beverage # cette fois l'instance n'y touche pas
'coffee'
>>> MyClass.beverage = 'tea' # je change d'avis
>>> myO.beverage # l'instance n'y ayant pas touché, ça change aussi pour elle
'tea'
```

# Définition des attributs et du constructeurs

- On déclare habituellement les attributs de classe dans le bloc qui suit class, dont les méthodes, et les attributs d'instance sont initialisés dans le *constructeur* (mais d'autres peuvent être créés ensuite !)
  - Le constructeur est défini comme fonction `__init__(self, ...)` dans la classe
  - Il est appelé quand vous appelez la classe comme une fonction `ClassName( ... )`

```
>>> class Breakfast:
    forbidden = 'beer wine whisky'.split() # attribut de classe
    def __init__(self,num,bev='coffee'):    # constructeur
        self.num = num # attribut d'instance
        if bev not in Breakfast.forbidden: # le nom de la classe c'est
                                                # donc un attribut de classe
            self.beverage = bev # attribut d'instance
        else:
            self.beverage = 'coffee'

>>> myBkfst = Breakfast(301,'tea') # appelle __init__(self,301,'tea')
>>> myBkfst.beverage # attributs d'instance
'tea'
>>> myBkfst.num
301
```

# Autres méthodes

Les autres méthodes se définissent de la même façon mais sont appelées à travers leur nom directement

```
>>> class Breakfast:
    '''Classe représentant un petit déjeuner d'un client de l'hôtel'''
    forbidden = 'beer wine whisky'.split() # attribut de classe
    def __init__(self,num,bev='coffee'):    # constructeur
        self.num = num # attribut d'instance
        ... # comme avant
    def showInfo(self):
        print('Numéro de chambre :',self.num)
        print('Boisson :', self.beverage)
    def getInfo(self):
        return dict(num=self.num,bev=self.beverage)
    def changeBeverage(self,bev):
        if bev not in Breakfast.forbidden:
            self.beverage = bev

>>> myBkfst = Breakfast(301,'tea') # appelle __init__(self,301,'tea')
>>> myBkfst.changeBeverage('whisky') # comme
                                     # Breakfast.changeBeverage(myBkfst,'whisky')

>>> myBkfst.beverage
'tea'
>>> myBkfst.changeBeverage('chocolate')
>>> myBkfst.beverage
'chocolate'
>>> myBkfst.showInfo() # comme Breakfast.showInfo(myBkfst)
>>> myBkfst.getInfo()  # comme Breakfast.getInfo(myBkfst)
{ 'num':301, 'bev':'chocolate' }
```

# Convention d'appel sur les méthodes

- Un appel de méthode sur une instance :

```
myO.do_something( ... )
```

est réécrit et exécuté comme :

```
ClassName.do_something(myO, ... )
```

- Vrai aussi pour les classes des types intégrées :

```
'abcd'.upper()    pareil que    str.upper('abcd')
```

- Pièges courants :
  - Oublier de déclarer *self* dans le *def do\_something*
  - Oublier de préfixer *self.* dans les méthodes pour accéder aux attribut d'instances
  - Oublier de préfixer *ClassName.* pour accéder aux attributs de classes

# Visibilité des attributs

- Les attributs d'une classe ou d'une instance sont **toujours accessibles** de l'extérieur (hors des méthodes) *myO.the\_attr*
- Cependant des conventions sont bien connues :
  - SANS mettre de `__` à la fin !
    - préfixer par UN `_` (souligné) : *il vaut mieux pas toucher*
    - préfixer par DEUX `__` (soulignés) : *il ne vaut VRAIMENT mieux pas toucher* (dans les deux cas l'attribut n'est pas visible via *help()* par contre il l'est via *dir()* )
  - Note** : Double souligné AVANT et APRÈS, c'est **tout autre chose** ce sont les méthodes spéciales (*dunder methods*) appelées hors de leur nom (constructeurs, `__add__`, etc.)
- Voir le sketch des Monty Python : *How not to be seen ?*
- Il y a un truc pour le double souligné initial, mais un voisin vous dira comment le trouver (*It is Mister "dir" by the way...*)

# (In)Visibilité en action

- Examinons ce qui se passe avec les attributs dont le nom débute par un ou deux "\_" :  
On peut voir que l'aide ne montre pas tous les attributs :

```
>>> from visibility import MyClass
>>> help(MyClass)
>>> dir(MyClass)
```

```
# Fichier visibility.py
class MyClass:
    '''And now something completely different...'''
    data = 'spam'
    _hidden = 'ham'
    __veryhidden = 'egg'
    def __init__(self, data=None):
        self.data = data
    def show(self):
        print(self._mymethod())
        print('data :', self.data)
        print('_hidden :', self._hidden)
        print('__veryhidden :', self.__veryhidden)
    def _mymethod(self):
        return 'Now, something completely different'

myO = MyClass('sausage')
myO.show()
print(myO.data, myO._hidden) # Ok
print(myO.__veryhidden) # Erreur
print(dir(myO)) # commentez la ligne supérieure pour voir...
print(myO._MyClass__veryhidden) # accessible si on fouille un peu...
```

# **Exercice 2.1**

## **Classes et instances**



- Définition de classes et instanciation
- **Héritage**
- Surcharge d'opérateurs
- Décorateurs et méthodes abstraites
- UML et patrons de conception
- Modules
- Tests unitaires

# Hériter d'une classe parente

- L'héritage est la relation « est un•e » : la classe fille hérite de tous les attributs (y compris les méthodes) de la classe parente
- La classe fille peut modifier des méthodes, ajouter des attributs, ajouter des méthodes :

```
from datetime import date
class HotelService:
    def __init__(self,num,when=None,cost=0):
        self.num = num
        self.cost = cost
        self.date = when if when is not None else date.today()

class Breakfast(HotelService):                # héritage
    forbidden = 'beer wine whisky'.split()
    price = 9
    def __init__(self,num,bev='coffee',when=None):
        self.beverage = bev if bev not in
            Breakfast.forbidden else 'coffee'
        # HotelService.__init__(self,num,when,Breakfast.price)
        super().__init__(num,when,Breakfast.price) # mieux!
```

# Appeler les méthodes de la classe parente

- Il convient de limiter la duplication de code en appelant explicitement les méthodes de la classe parente dans les méthodes surchargées de la classe fille
- Applicable pour le constructeur ou toute autre méthode
- *super()* permet de ne pas spécifier explicitement la classe parente pour qualifier la méthode à appeler
- Parfois il est plus commode de ré-implémenter entièrement la méthode de la classe fille sans appeler la méthode de la classe parente

# **Exercice 2.2**

## **Héritage**

# Héritage multiple

- Python supporte l'héritage multiple (Beaucoup d'autres langages objet ne le proposent pas) :

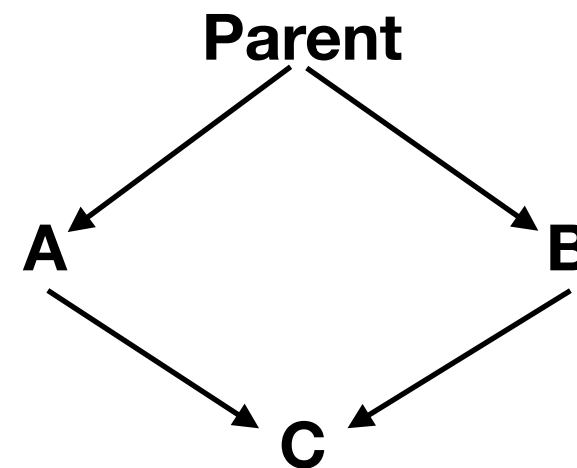
```
class MyClass(FirstClass,SecondClass, ...):  
    ...
```

- Une alternative est la *composition*, un attribut de la classe est du type d'une autre classe :

```
class MyClass(FirstClass):  
    def __init__(self, ...):  
        self.something = SecondClass(...)
```

- Un souci potentiel est le *diamond problem* :

```
class A(Parent):  
    ...  
class B(Parent):  
    ...  
class C(A,B):  
    ...
```



- Et ça peut être plus complexe sur le même principe. Quelle ordre de résolution de nom d'attribut appliquer ?  
C ? A ? Parent ? B ? (Python 2 *old style*) ou bien C ? A ? B ? Parent ? (Python 2 *new style*, Python 3)
- Il vaut mieux éviter les problèmes pour ne pas les rencontrer, en pratique on utilise l'héritage multiple surtout des classes complètement étrangères et fournissant des fonctionnalités génériques (*Mix In*)

- Définition de classes et instanciation
- Héritage
- **Surcharge d'opérateurs**
- Décorateurs et méthodes abstraites
- UML et patrons de conception
- Modules
- Tests unitaires

# Les opérateurs et même plus correspondent à des méthode spéciales

- Ce sont ces méthodes dont le nom commence et se termine par double souligné (*dunder methods*)

```
a + b    est    a.__add__(b)    est    classA.__add__(self,other)
*, -, /  sont  __mul__, __sub__, __div__, ...
a[b]    est    a.__getitem__(b)  est implémenté comme
                classA.__getitem__(self,thing)
a[b] = c  est  a.__setitem__(b,c)
a == b    est  a.__eq__(b)
str(a)    est  a.__str__()
sortie dans le REPL (repr) est a.__repr__
etc.
```

- On peut même oublier certaines méthodes, par exemple si vous avez <, > et == (\_\_lt\_\_, \_\_gt\_\_, \_\_eq\_\_) automatiquement <=, >= vont marcher
- Si vous êtes pas cohérent vous vous tirez dans le pied

# Exemple : une liste additionnable avec un entier

- On veut pouvoir écrire :

```
>>> myL = IntList([1,2,3])
>>> myL + 42
IntList([43,44,45]) # appel de IntList__repr__(myL)
```

```
class IntList:
    def __init__(self,data):
        self.data = [int(x) for x in data]
    def __add__(self,other):
        return IntList([x + int(other) for x in self.data])
    def __repr__(self):
        return 'IntList({})'.format(repr(self.data))
```

- Évidemment il faudrait logiquement avoir aussi `__sub__`, `__mul__`, `__getitem__`, `__setitem__`, `__iter__`, `__next__`, etc...
- Hériter du type *list* plutôt que de composer avec lui serait une bonne idée (moins de méthodes à implémenter)



- Définition de classes et instanciation
- Héritage
- Surcharge d'opérateurs
- **Décorateurs et méthodes abstraites**
- UML et patrons de conception
- Modules
- Tests unitaires

# Décorateurs

- Souvenez-vous de l'exemple donné rapidement dans le chapitre précédent :

```
from math import sqrt
def logfunc(anyfunc):
    def _(*args,**kwargs): # _ ou autre, peu importe !
        print('*** appel de', anyfunc.__name__,args,kwargs)
        res = anyfunc(*args,**kwargs)
        print('*** resultat',res)
        return res
    return _

@logfunc # length est "décoré"
def length(x,y):
    return sqrt(x**2 + y**2)

print(length(2,3) + length(4,5))
```

- Testez le. Que se passe-t-il en commentant ou non la ligne *@logfunc*. En quoi c'est utile ?
  - Les appels à la fonction sont logués (ligne print), la fonction est exécutée normalement
  - Au lieu d'exécuter *length*, lorsque sa définition est décorée c'est la fonction *logfunc(length)* qui est appelée. On peut décorer ainsi n'importe quelle fonction.

# Classe abstraite de base

- Une classe abstraite de base est une classe qui définit une partie de ses méthodes mais qui n'est pas censée être instanciée
- Les classe filles DOIVENT implémenter ces méthodes

```
class HotelService:
    def showLabel(self):
        pass # On pourrait mettre raise NotImplentedError qui
            # lève une exception à l'execution de la méthode
```

- On aimerait être prévenu au plus tôt si une classe fille omet d'implémenter une de ces méthodes, au moment de **l'instanciation** et pas au moment des tests d'appels

```
from abc import ABC, abstractmethod
class HotelService(ABC):
    @abstractmethod          # décorateur
    def showLabel():
        pass
```

- Si on instancie une classe fille de *HotelService* qui n'implémente pas *showLabel* on aura une exception *TypeError* dès l'instanciation

# Propriétés et accesseurs

- On peut déguiser des appels de méthodes lors des accès à un attribut (lecture ou écriture). Un tel attribut est appelé une propriété et se manipule à travers un *getter* et un *setter*
- Le véritable attribut est caché ! Évaluer ou assigner *obj.attribut* appelle automatiquement les fonctions spécifiées (*getter* et *setter*)

```
class Breakfast:
    def __init__(self, thefood):
        self.food = thefood    # appelle le setter
    @property                  # getter
    def food(self):
        print('*** Reading attribute food')
        return self._food
    @food.setter                # setter
    def food(self, thefood):
        print('*** Setting attribute food')
        if thefood == 'spam':
            raise ValueError('No spam please')
        self._food = thefood

bk = Breakfast('egg')
print(bk.food)    # appelle la fonction @property
bk.food = 'ham'   # appelle la fonction @food.setter
```

# Autres types de méthodes

Vous pouvez déclarer des méthodes qui seront appelées de façon différente des méthodes habituelles avec des décorateurs

- Méthodes **statiques** : pas de *self*

```
@staticmethod
def isforbidden(drink) :
    ...
```

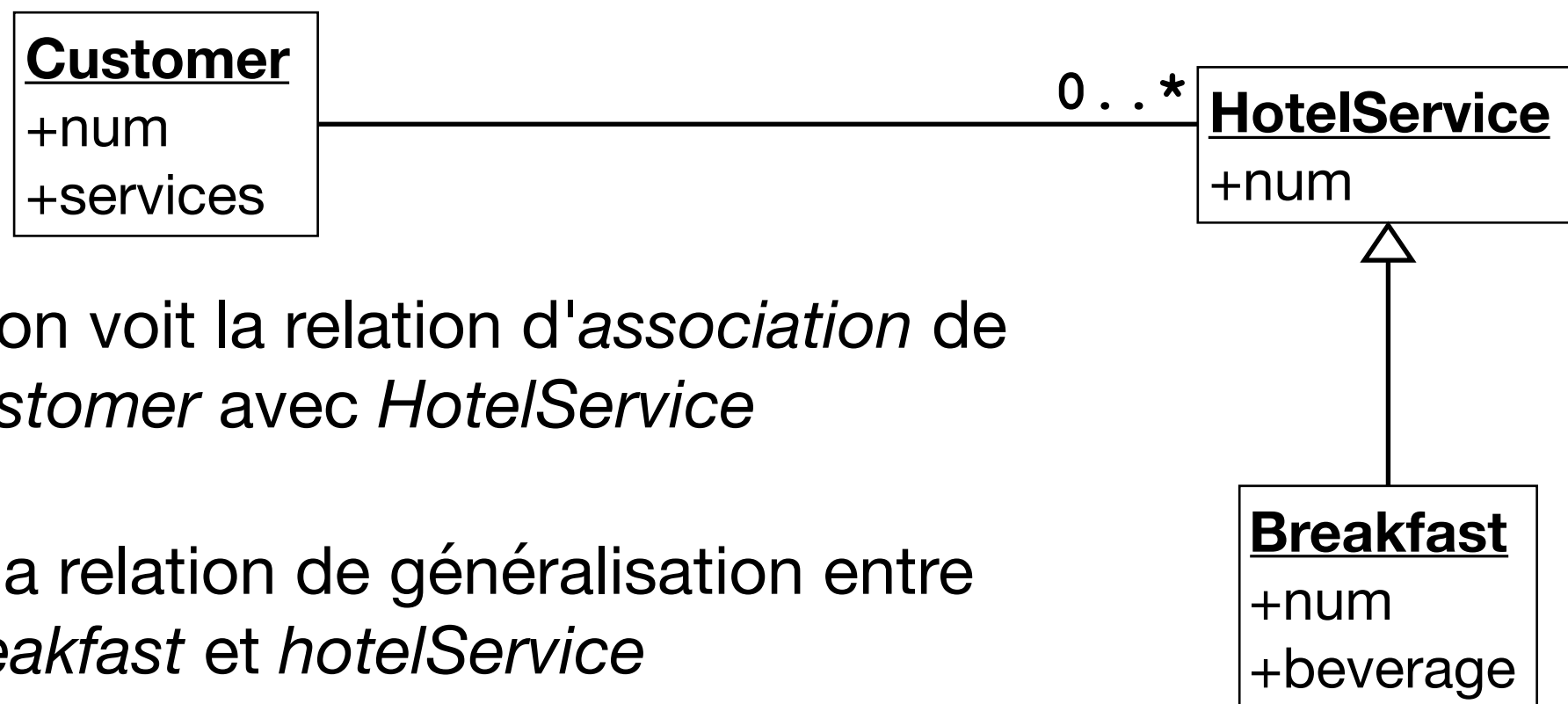
- **Méthodes de classes** : reçoivent la classe *c/s* au lieu de l'instance *self*

```
@classmethod
def addForbiddenDrink(cls, drink) :
    ...
```

- Définition de classes et instanciation
- Héritage
- Surcharge d'opérateurs
- Décorateurs et méthodes abstraites
- **UML et patrons de conception**
- Modules
- Tests unitaires

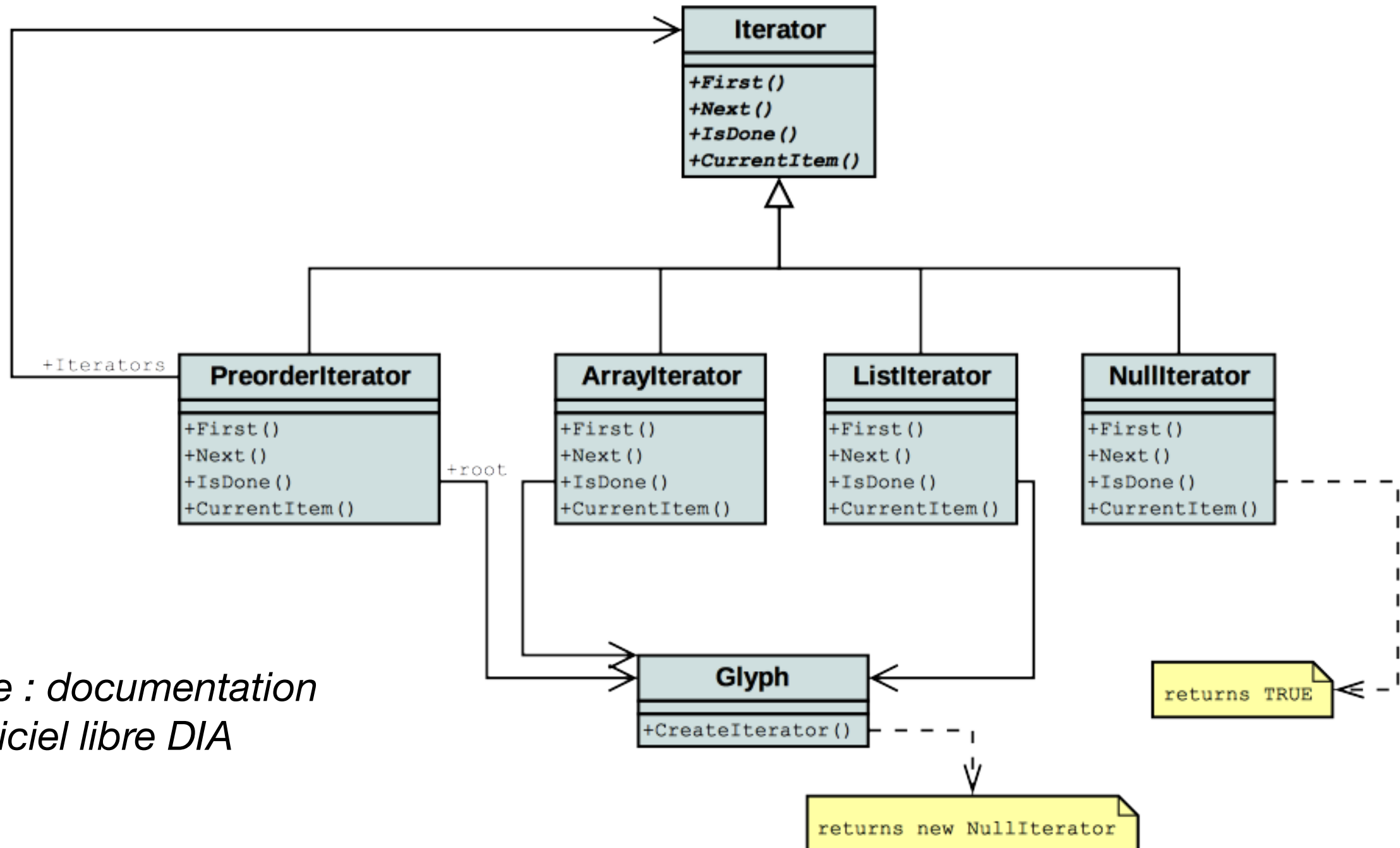
# Qu'est-ce que UML ?

- *Unified Modeling Language* : un langage de modélisation des relation entre classes et entre instances basé sur des **pictogrammes** :



- Ici on voit la relation d'*association* de *Customer* avec *HotelService*
- Et la relation de généralisation entre *Breakfast* et *hotelService*
- *UML* permet aussi d'exprimer *réalisations* et *dépendances*

# Un diagramme UML complet



Source : documentation  
du logiciel libre DIA



# Patrons de conception

- Un patron de conception est une implémentation *réutilisable* d'une solution à une problématique générale
- Souvent réalisés sous forme de classes abstraites ou de décorateurs de fonction ou de classes
- Popularisés par l'ouvrage du *Gang of Four* (GoF) : ***Design Patterns: Elements of Reusable Object-Oriented Software*** par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides en 1994
- Patrons usuels : *Template, Observer, Decorator, Proxy, Iterator, ...*

# **Exercice 2.3**

## **UML et patrons de conception**

- Définition de classes et instanciation
- Héritage
- Surcharge d'opérateurs
- Décorateurs et méthodes abstraites
- UML et patrons de conception
- **Modules**
- Tests unitaires

# Créer un module

- Rien à faire : un script Python est toujours utilisable comme module
- Si vous y faites autre chose que définir des fonctions, des classes, des objets, vous pouvez introduire un test :

```
print('Script ou module')
if __name__ == '__main__':
    # ignoré si on est importé
    print('Ici on est pas dans un import')
```

# Importer un module

- `breakfast.py` est un fichier présent dans le répertoire courant OU un des répertoires où Python va chercher ses modules

```
>>> import sys  
>>> sys.path
```

- Importer le module entier :

```
import hotelServices  
myBkfst = hotelService.Breakfast(42)
```

- Importer un module en le renommant :

```
import hotelServices as hls  
myBkfst = hls.Breakfast(42)
```

- Importer des noms à partir d'un module :

```
from hotelServices import Breakfast, Meal  
myBkfst = Breakfast(42)  
# Ceci est déconseillé :  
from hotelServices import *
```

- Importer des nom à partir d'un module en le renommant (*rarement utile*) :

```
from hotelServices import Breakfast as Bkf, Meal as Ml
```

# Modules imbriqués et paquetages

- Si vous importez un module *modA* qui lui même importe un module *modB* vous pouvez accéder à *modB* à travers *modA* :

```
>>> import modA
>>> modA.modB.whatever()
```

- Un paquetage est un ensemble de modules stocké dans une arborescence

```
    hotelservice/hotelservice.py
    hotelservice/breakfast.py
    hotelservice/meal.py
>>> import hotelservice.breakfast as Bkf
>>> from hotelservice.breakfast import Breakfast
```

- Si un fichier `__init__.py` existe dans un répertoire traversé il est automatiquement importé (dans Python 2 ce fichier est obligatoire et, du coup, souvent vide)

# **Exercise 2.4**

## **Modules**

# Les modules de la bibliothèque standard

- Python est fourni « avec les piles » une large collection de modules fournis d'office : section *Python Standard Library* du manuel sur [python.org](https://python.org)
- Le site [pypi.org](https://pypi.org) *Python Package Index* est un dépôt géré par la PSF de modules tiers validés, installable automatiquement avec `pip` ou `pipenv`
- *math, string, collections, itertools, functools, os, sys, abc*
- *argparse, re, struct, datetime, array, copy, pprint, enum, decimal, random, operator, stat, glob, shutil, csv, logging, getpass, ctypes, socket, email, json, timeit, dis, ...*



# Installer les modules du dépôt officiel PyPI

- Un module présent dans PyPI est sans doute disponible sous GNU/Linux sous forme de paquet RPM ou DEB pour python3 ou python2

```
$ apt show python3-requests python-requests
```

- Si vous êtes administrateur vous pouvez installer le module facilement :

```
$ sudo apt install python3-requests
```

- Sinon vous pouvez installer un module de PyPI avec l'outil *pip* (*pip3*) (à terme remplacé par *pipenv*) :

```
$ sudo pip install requests # install au niveau système
```

```
$ python3 -m venv myenv --without-pip # crée l'env
```

```
$ cd myenv
```

# Installer les modules du dépôt officiel PyPI

- Une fois dans votre environnement (pour un venv déjà créé par PyCharm on se place dans le bon répertoire)

```
$ cd PycharmProjects/FormationPython/venv
$ source ./bin/activate      # active un virtual env
(venv) $ pip install requests # au niveau du venv
```

- On peut déclencher un pip install dans un projet PyCharm :  
**File->Settings->Project: Formation Python->Project interpreter,**  
bouton '+' et donner le nom du module PyPI (par exemple *requests* et  
bouton **"Install"**
- Si voulez des paquets RPM ou DEB avec des versions différentes des  
versions disponibles dans les dépôts habituels : **FPM** est votre ami

# Modules sys, os et subprocess

- Module sys : environnement Python

```
>>> import sys
>>> dir(sys)
>>> sys.version, sys.executable
```

- Modules os : interface avec le système d'exploitation

```
>>> import os
>>> os.environ['PATH']
>>> os.environ['LANG']
```

- Modules *os.path* : manipulation de chemins vers des fichiers
- *os.popen* était la façon normale de lancer un processus extérieur, il est obsolète. Il convient d'utiliser le module *subprocess*

```
>>> from subprocess import Popen, PIPE
>>> with Popen(['ping', '-c', '3', 'localhost'], stdout=PIPE) as proc:
    for line in proc.stdout:    # oh un itérateur !
        print(line.rstrip())
```

- Sous MS Windows pas de '-c', '3' (la commande *ping* est un peu différente)

## **Exercice 2.5**

**Utiliser des modules de la bibliothèque standard**

# Types supplémentaires de la bibliothèque standard

- Plusieurs modules standards fournissent des types de données intéressants
- Le module *collections* fournit, par exemple, *deque*, *namedtuple* et *defaultdict*
- Le module *enum* fournit *Enum* :

```
class Choice(Enum)
    YES = 'oui'
    NO  = 'non'
    NA   = 'nsp'
```

```
>>> myAnswer = Choice.YES
>>> myAnswer
<Choice.YES: 'oui'>
>>> myAnswer.value
'oui'
```

# Le module *typing*

- Depuis Python 3.5 il existe un module **expérimental** *typing* qui introduit des annotations de type

```
def breakfast(food: str) -> str:  
    return 'spam spam spam ' + food
```

```
from typing import List  
Vector = List[float]  
def scale(s: float, vector:Vector) -> Vector:  
    return [ s * v for v in vector ]
```

- Python 3 (pour l'instant) ignore totalement ces indentations, pour en profiter on passer le type checker **mypy** (installable dans un venv avec pip ou via PyCharm)
- Il reste possible (et conseillé) d'utiliser *isinstance(object,class)* ou *instance(object,tuple\_de\_class)* au début des fonctions critiques et déclenche un `raise ValueError` en cas de passage d'une donnée invalide

## **Exercice 2.6**

### **Révision sur classes, objets, héritage et modules**

- Définition de classes et instanciation
- Héritage
- Surcharge d'opérateurs
- Décorateurs et méthodes abstraites
- UML et patrons de conception
- Modules
- **Tests unitaires**



# Tests unitaires

- Tester en continu le bon fonctionnement de votre code est crucial
- Les tests *unitaires* vérifie le comportement individuel de chaque fonction ou méthode isolément
  - Les tests *fonctionnels* valident l'application entière dans un contexte de production
  - *pylint* et *pychecker* analyse le style du code
- Plusieurs modules permettent d'écrire des tests unitaires : unittest, nose et pytest

# Exemple avec *unittest*

- Nous avons défini une classe *Point* dans un module *Points* qui a trois attributs dont deux attributs en lecture seule :

```
class Point:
    def __init__(self, x, y, label):
        self._x = x
        self._y = y
        self.label = label
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value): # toute cette partie peut être omise
        # sauf si on veut un message spécifique
        raise AttributeError('Read-only attribute: x')
    ... # idem pour y
    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)
```

- Nous voulons tester le constructeur et si `__eq__`, qui est appelée quand on évalue `p1 == p2`, fonctionne bien...

# Test d'une méthode

- La première méthode à tester est le *constructeur*
- Il faut tester deux choses pour valider la méthode `__eq__` : que l'on obtient *True* pour deux points égaux et que l'on obtient *False* pour deux point différents

```
import unittest
from Points import Point
```

```
class PointTestCase(unittest.TestCase):
    def test_init(self):
        p = Point(3, 4.0, 'spam')
        self.assertEqual(p.x, 3.0)
        self.assertEqual(p.y, 4.0)
        self.assertEqual(p.label, 'spam')
    def test_eq_instances_equal(self):
        p1 = Point(3, 4, 'spam')
        p2 = Point(3, 4.0, 'ham')
        self.assertEqual(p1, p2)
# suite à droite :
```

```
# suite...
def test_eq_instances_not_equal(self):
    p1 = Point(3, 4, 'spam')
    p2 = Point(4, 3, 'ham')
    self.assertNotEqual(p1, p2)

if __name__ == '__main__':
    unittest.main()
```

```
$ ./unittest_Point.py
.....
-----
Ran 5 tests in 0.001s

OK
```

# Tester les exceptions

- Nous devons aussi tester que x et y sont bien accessibles en lecture seule :

```
def test_verify_readonly_x(self):  
    with self.assertRaises(AttributeError):  
        p = Point(3, 4.0, 'spam')  
        p.x = 42
```

- On peut aussi valider le message d'erreur avec une expression régulière : *assertRaisesRegex(...)*
- D'autres types de tests sont implémentés par *unittest.TestCase* :
  - *assertTrue, assertIsNotNone, assertRegex, ...*

# Autres modules de test unitaire

- *pytest* et *node* sont aussi disponibles dans PyPI
- Historiquement construits sur la même base (*pytest*) et tests sensiblement plus simples à écrire qu'avec *unittest*
- Il suffit de définir des fonctions dont le nom contient le mot *test* !

```
'''Script test_Point.py'''
def test_init():
    p = Point(3, 4.0, 'spam')
    assert (3.0, 4.0, 'spam') == (p.x, p.y, p.label)

def test_eq_instances_equal():
    p1 = Point(3, 4.0, 'spam')
    p2 = Point(3, 4.0, 'ham')
    assert p1 == p2

@raises(AttributeError):
def test_verify_read_only_x():
    p = Point(3, 4, 'spam')
    p.x = 42
```

```
$ nosetests test_Point.py
...
-----
Ran 3 test in 0.001s
OK
```

# Autres fonctionnalités de *nose*

- Le décorateur *@timeit* teste la durée d'exécution maximal d'une fonction
- *nose* peut vous indiquer le degré de couverture de votre code par vos tests
  - Quelles parties sont couvertes et...
  - surtout quelle parties ne le sont pas !

```
$ nosetest --with-coverage --cover-html test_Point.py
```

- Fichier `index.html` généré, consultable dans un navigateur Web
- *nose* peut aussi exécuter tous les scripts contenant le mot *test* dans leur nom dans une arborescence

# **Exercice 2.7**

## **Tests unitaires**





Programmation Python — une introduction complète

# Chapitre 3

## Entrées/Sorties, bases de données, interfaces et Web

Accès à des fichiers, exceptions et gestionnaire de contexte, formats de données, bases de données relationnelles, bibliothèques d'interface graphique, développement Web, APIs

Jean-Pierre MESSENGER

jp@xiasma.fr

v2.0a - février 2020

**Attribution - Pas d'Utilisation Commerciale - Pas de Modification**

**CC BY-NC-ND**



- **Entrées/sortie sur des fichiers, exceptions et gestionnaire de contextes**
- Format de données
- Bases de données
- Interfaces graphiques
- Développement Web
- Interfaçage C/Python

# L'objet fichier

- Un appel à *open* : *open("chemin/vers/un/fichier",mode)* renvoie un objet
- Le mode peut être "r" (défaut), "w" (écrire en **vidant** le contenu !), "a" (écrire à partir de la fin). Ajouter un "+" pour lire et écrire. Ajouter "b" sous MS Windows pour les fichiers binaires.
- Cette objet est un ***itérateur*** fournissant les lignes d'un fichier texte
- Il a aussi des méthodes spécifiques : *read()*, *readline()*, *readlines()*, *seek()*, *write()*

```
>>> f = open('/etc/hosts')
>>> for l in f:                # lecture ligne par ligne
    print(l.rstrip())          # enlève \n à la fin
...
>>> f = open('/etc/hosts')
>>> print(f.read())            # lecture complète
```

# Gérer les erreurs

- Une erreur à l'exécution est une instance d'une classe fille de *Exception*, c'est un objet en Python
- On peut intercepter une Exception pour traiter l'erreur

```
try:
    f = open('/etc/hosts'):
except (FileNotFoundError, PermissionError) as e:
    print("Impossible d'ouvrir le fichier",e.args)
# except: # ou mieux Exception as e:
#     print("Problème, mais je ne sais pas lequel DU TOUT")
#     print("Des infos sur le pb :",e.args)
else:
    f.close() # only if NO exception
finally:
    # Toujours exécuté, exception ou non
    print('Toujours executé !')
print('Si une exception non interceptée a lieu, on arrive jamais ici')
```

- Un block *except:* sans spécifier d'exception capturera TOUTES les exceptions possibles :  
Attention, n'y supposez jamais *rien* (même une erreur de frappe sur le nom d'une méthode ou de fonction EST une exception !)

# Gérer vos propres exceptions

- Vous pouvez lever explicitement une exception :

```
raise NotImplementedError('Fonctionnalité non  
implémentée')
```

- Les exceptions sont des classes, vous pouvez créer les vôtres :

```
class ClientHotelNotFoundError(Exception) :  
    def __init__(self, message, errors):  
        super().__init__(message)  
        self.errors = errors
```

- Un simple bloc *pass* peut suffire (pas de comportement particulier)
- Vous pouvez choisir d'hériter d'une exception plus spécifique, *IOError* par exemple

# Gestionnaire de contexte

- Un protocole utilisé par l'instruction *with*, implémenté par *open*, *Popen*, et beaucoup d'autres objets qui décrivent l'accès à une ressource du système (ouverture de fichier, connection à une base de données, verrou, ...)
- Permet d'éviter d'utiliser *try: / finally:* pour libérer les ressources
- Appelle `__enter__` sur chaque objet, n'exécute le bloc que si *tous* réussissent et appelle `__exit__` sur ceux qui ont réussi en sortant, que le bloc ait été exécuté ou pas

```
with open('input.txt')          as input, \
     open('output.txt', 'w') as output:
    for l in input:              # bloc exécuté seulement si les
        output.write(l)         # deux fichiers sont ouverts
# ici dans tous les cas, seuls les fichiers
# ouverts seront fermés !
```

# try/except et with

- On combine try/except et with facilement:

```
try:
    with open('in.txt')          as input,
          open('out.txt','w') as output:
        for line in input:
            output.write(line)
    # en sortie de bloc les fichiers ouverts seront
    # fermés et seulement eux
except (FileNotFoundError, PermissionError) as e:
    print("Impossible d'ouvrir le fichier",e.args)
```

- Try/except prend en charges les erreurs (les exceptions) et with gère proprement les succès, y compris partiels (il va exécuter les `__exit__` nécessaires, c'est-à-dire *ici* fermer les fichiers ouverts)

# **Exercice 3.1**

## **Entrées/sorties sur des fichiers**



- Entrées/sortie sur des fichiers, exceptions et gestionnaire de contextes
- **Format de données**
- Bases de données
- Interfaces graphiques
- Développement Web
- Interfaçage C/Python

# Formats textes courants

- Pour la plupart des formats usuels de fichiers un module existe
- CSV : module csv
- **XML** : module standard xml, dans PyPI : **lxml**, BeautifulSoup, ...
- .ini : module ConfigParser
- **JSON** : module json, yaml
- Pickle et Shelve : format binaire propre à Python, efficace pour des structures de listes, tuple, dictionnaires, objets, mais un peu passé de mode en faveur de JSON
- Pour les formats binaires aussi : images (PIL et Pillow), audio (wav, flac, ogg, mp3, ...), PDF, etc. quelques formats propriétaires (bureautique)

# Accès aux fichier CSV

- Format courant de données (tableur, base de données, etc.)

```
"John", "Cleese", 42  
"Connie", "Booth", 34
```

- Le module `csv` fournit les fonctions `reader` et `writer` :

```
>>> import csv  
>>> f = open('test.csv')  
>>> rdr = csv.reader(f)  
>>> for t in rdr:    # itérateur qui contient des tuples  
...     print(t)  
( 'John', 'Cleese', '42' )  
( 'Connie', 'Booth', '34' )
```

- `csv.writer` a une méthode `writerow` qui écrit dans le fichier une séquence (liste, tuple, ...) quelconque

# **Exercice 3.2**

## **Lecture d'un fichier CSV**

- Entrées/sortie sur des fichiers, exceptions et gestionnaire de contextes
- Format de données
- **Bases de données**
- Interfaces graphiques
- Développement Web
- Interfaçage C/Python

# Base de données

- Toutes les systèmes de gestion de bases de données relationnelles (SGBDR) sont gérés
  - Libres : *mysql/mariadb, PostgreSQL, sqlite3*
  - Non libres : *Oracle, IBM DB2, Microsoft SQL Server, ...*
- Principe (la plupart des modules offrent la *même* API : DB API cf. PEP 249) :
  1. Obtenir une connexion (PostgreSQL, MySQL, etc.) ou ouvrir un fichier (sqlite)
  2. Créer un curseur
  3. Préparer une requête (langage SQL) et l'exécuter
  4. Si c'est une écriture on "*commit*" la transaction
  5. Parcourir les résultats ou examiner le code de retour (succès ou échec)
- Des bases non relationnelles sont gérées aussi : LDAP, MongoDB, Redis, ...

# Exemple avec sqlite3

- Exemple de lecture simple (à partir du fichier créé avec le client sqlite3 précédemment), le fichier **cities.sqlite3** doit être dans le même répertoire.

```
import sqlite3    # pour PostgreSQL : psycopg2
                  # pour MySQL/MariaDB : Mysqldb ou pymysql

with sqlite3.connect('cities.sqlite3') as conn:
    curs = conn.cursor()
    curs.execute('SELECT * FROM cities')
    for line in curs:
        print(line)
```

- On récupère un tuple pour chaque enregistrement renvoyé par la requête !
- Autre méthode de récupération des données : *fetchone()*, *fetchall()*, *fetchmany(n)*

# Requêtes préparées

- Il ne faut jamais construire une requête avec des éléments variables (surtout si venant de l'utilisateur) dans la clause WHERE (Pour se protéger contre l'injection de SQL)
- Une requête préparée est exécutée en deux temps
  1. Une requête "à trou" est envoyé au serveur
  2. On envoie les données pour compléter la requête et on l'exécute

```
pop = 100000
name = 'B%'
curs.execute('SELECT * FROM cities' \
              ' WHERE pop > ?', (pop,) )
curs.fetchall() # récupère tout d'un coup
curs.execute('SELECT * FROM cities' \
              ' WHERE name LIKE ?', (name,) )
curs.execute('SELECT * FROM cities' \
              ' WHERE name LIKE ? AND pop > ?', (name,pop) )
```



# Commit de transaction

- Si vous écrivez et que votre SGBDR supporte les transactions (ce que font tous les SGBDR sérieux, sauf MySQL par défaut, mais le commit marche quand même)

```
curs.execute('UPDATE cities SET pop=? WHERE  
id=1', (pop,))  
curs.execute('UPDATE cities SET area=?  
WHERE id=2', (area,))  
conn.commit() # l'écriture est validé
```

- Un source d'info sur SQL, le blog "Use the index Luke" de Markus Winand (et son livre)

# **Exercice 3.3**

## **Bases de données**

- Entrées/sortie sur des fichiers, exceptions et gestionnaire de contextes
- Format de données
- Bases de données
- **Interfaces graphiques**
- Développement Web
- Interfaçage C/Python

# Interfaces graphiques

- Plusieurs modules permettent la création d'interfaces graphiques, basés sur des bibliothèques tierces :
    - Tkinter (existe partout, simple et efficace)  
<https://likegeeks.com/python-gui-examples-tkinter-tutorial/>
    - GTK (GIMP Toolkit, lié au bureau Gnome), Qt Lié au bureau KDE, Kivy pour Android
    - wxWindows (portable), PyGame pour développer des jeux (et pas seulement)
    - GTK et Qt ont des générateurs d'interfaces : Glade, Qt Designer
- ```
import tkinter as tk # Tkinter in Python 2
top = tk.Tk() # Fenêtre principale
# Création des widgets (bouton, menus, ...)
top.mainloop() # Boucle d'évènements
```
- Sous Ubuntu/Debian le apt install python-gtk-doc contient gtk-demo qui montre plein de possibilité avec leur code source Python (v2)

# Demo Python2/Gtk2 et kivy

- Installer la documentation de Python 2 / Gtk

```
$ sudo apt install python-gtk2-doc  
$ pygtk-demo &
```

- Exemples du panneau de gauche s'exécutent en double-cliquant et le source est visible à droite

- Demo de Qt/Pyside :

Installer le module *pyside2* à partir des settings de PyCharm  
code d'exemple : <https://codeshare.io/5P0rBX>

- Démo de Kivy :

```
$ sudo apt install libgl1-mesa-dev
```

Dans PyCharm (utilise pip) : installer :

- pip
- cython
- Kivy
- Kivy-examples
- pygame

On peut alors charger dans PyCharm et exécuter (par exemple) :

```
venv/share/kivy-examples/demo/pictures/main.py  
.../3Drendering/main.py  
.../tutorial/pong/main.py
```

# Documentation disponible

Ressources sur pour d'autres bibliothèques de widgets :

- *Tkinter* :

<https://python.doctor/page-tkinter-interface-graphique-python-tutoriel>

- *Themed Tk (ttk)* :

<https://docs.python.org/3/library/tkinter.ttk.html>

- **Gtk3** :

<https://python-gtk-3-tutorial.readthedocs.io/en/latest/>

- **Qt** :

<https://www.qt.io/qt-for-python>

- **WxWindows** :

<https://wxpython.org>

# **Exercise 3.4**

## **Tkinter**

- Entrées/sortie sur des fichiers, exceptions et gestionnaire de contextes
- Format de données
- Bases de données
- Interfaces graphiques
- **Développement Web**
- Interfaçage C/Python



# Le Web

- Inventé en 1991 au CERN par Tim Berners-Lee *et al.* pour publier des documents (fichiers statiques) avec des liens hypertexte (HTML)
  - Une requête HTTP est décrite principalement par une URL de la forme :  
`http[s]://nom_serveur[:num de port]/vers/la/ressource`
1. Le client (navigateur ou autre) ouvre une connexion TCP sur le port 80 par défaut
  2. Le client envoie une ligne de requête et une en-tête (et une ligne vide suivi d'un contenu dans certains cas) :  
`GET /vers/la/ressource HTTP/1.1`  
`Host: nom_serveur`
  3. Le serveur répond avec une en-tête et un contenu s'il y en a :  
`HTTP/1.1 200 OK`  
`Content-Type: text/html; charset=utf-8`  
(ligne vide)  
`<html><head><title>Titre</title><body> ...`
  4. L'échange est terminé, tout recommence pour chaque page...

# Web dynamique

- Le serveur Web (Apache, NGINX, ...) peut exécuter un programme plutôt que de lire un fichier
- CGI : *Common Gateway Interface* (décrit la communication entre le serveur Web et le programme)
- Python fournit un serveur HTTP pour le développement (ne pas le mettre en production !!!)  
En lançant en ligne de commande (UNIX ou MS Windows)  
`python3 -m http.server --cgi`
- Pour arrêter le serveur HTTP de développement faites CTRL-C dans le terminal

# Démo : Web dynamique en mode CGI

- Dans un terminal : (en se plaçant dans le répertoire de TP)  
\$ **mkdir** cgi-bin  
\$ **gedit** cgi-bin/hello.py & # ou dans PyCharm

**hello.py** contient :

```
#!/usr/bin/env python3
```

```
print('Content-Type: text/html;' \
      'charset=utf-8', end='\n\n') # en-tête HTTP
print(' '<html><head><title>Titre</title></head>
<body>
<h1>Titre</h1>
<p>Du texte ...
</body>
</html>' '')
```

# Démo : Web dynamique en mode CGI (suite)

- Rendez le script exécutable, placez vous là où le répertoire `cgi-bin` a été créé et lancez le serveur HTTP de développement

```
$ chmod +x cgi-bin/hello.py
$ ./cgi-bin/hello.py # Test, il marche !
$ python3 -m http.server --cgi
```
- Testez : ouvrez un navigateur et vous allez à l'url :  
`http://localhost:8000/cgi-bin/hello.py`
- Si ça fonctionne (vous **voyez** la page **html**) ajouter du code dans *hello.py* qui affiche le résultat de l'appel de la fonction `date.today()` (*date* étant importé du module *datetime*) AVANT `</body></html>`. Testez en rechargeant la page Web dans le navigateur.
- **Ctrl-C** dans le terminal où tourne se serveur HTTP !!!

# Framework Web

- Un framework Web fournit
  - Un système de gabarit HTML
  - Un routeur d'URL vers des vues (fonction python qui traitent la requête)
  - La possibilité d'associer ces vues à des composants du modèle métier (fonctions, classes qui représentent le métier indépendamment du Web)
  - un ORM (*Object Relational Mapper*) utilisable en option
  - > séparer les préoccupations
- *Django, Flask, Pyramid*, et bien d'autres. cf. `wiki.python.org`

# Django : vues, modèle et gabarits

- Le distributeur d'URL associe des URLs selon leur forme (motif) avec des vues
- Les vues sont des fonctions exécutées dans le contexte d'une requête HTTP
- Les vues reçoivent les données extraites de l'URL (GET) ou provenant d'un formulaire (POST)
- Les vues font appel aux fonctions et classes du modèle pour manipuler les données métiers (base de données ou autre)
- Les vues combinent les informations reçues du modèle avec des gabarits HTML pour générer les pages Web

# Installer Django dans un venv (PyCharm)

- Dans un terminal extérieur allez dans le répertoire de votre projet PyCharm qui est un venv (ou dans le Terminal de PyCharm)  
\$ cd  
\$ cd PycharmProjects/*Nom du projet*/venv  
\$ source bin/activate  
(Active le venv dans le Shell courant)  
(sous Windows `bin\activate.cmd`)  
(Les commandes ci-dessus sont **inutiles** dans le Terminal de PyCharm)
- On installe django (le prérequis technique pour toute la suite)  
(venv) ... \$ pip install django
- **TOUTES les commandes qui suivent sont à saisir dans un Shell ou le venv est ACTIF (on voit *(venv)* au début de l'invite du Shell)**
- **Rappel : pour créer un venv (ici c'est INUTILE, PyCharm l'a fait lors de la création du projet) :**  
\$ python3 -m venv nom\_du\_venv [--without-pip]

# Construction d'un projet Django et d'une application

- Créez le projet `mysite` :

```
(venv) ... $ django-admin startproject mysite  
(venv) ... $ cd mysite
```

- Le script `manage.py` est créé dans le projet et permet d'y créer des applications, ici l'application **`countries`** :

```
(venv) ... $ python3 manage.py startapp countries
```

- Le répertoire `countries` est créé et rempli de fichiers de départ pour contenir votre application
- PyCharm vous permet de modifier et créer des scripts Python dans les répertoire `mysite/mysite` (projet Django) et `mysite/countries` (une application)



# Construction d'une application simple

## relier les distributeurs d'URLs

- **Créez** le distributeur d'URL de votre application **countries/urls.py** :

```
from django.urls import path
from . import views # importe countries/views.py
urlpatterns = [
    # nom de la fonction définie dans countries/views.py
    path('', views.index, name='index'),
]
```

- Référez le dans le distributeur d'URL (**déjà en place**) du site : ***mysite***/urls.py (en gras le code à **ajouter**) (attention PyCharm "cache" les lignes d'import)

```
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    # Associe les urls en /countries/... au distributeur
    # countries/urls.py
    path('countries/', include('countries.urls')),
    path('admin/', admin.site.urls),
]
```

# Une première vue pour notre index

- Créez une première vue dans **countries/views.py** (en gras code à **ajouter**) :

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def index(request):
    return HttpResponse("Index de l'application countries")
```

- Exécutez le serveur de développement dans le terminal :

```
(venv) ... $ python3 manage.py runserver
```

- Testez en allant avec un navigateur sur les URLs (vous devrez voir le texte émis par la vue index avec la seconde url) :

```
http://localhost:8000/
http://localhost:8000/countries
```

- Vous allez voir des avertissements concernant des migrations, résolvons le problème, CTRL-C puis :

```
(venv) ... $ python3 manage.py migrate
(venv) ... $ python3 manage.py runserver
```

# Récapitulons le flux applicatif

- Une connexion entrante correspond à une URL

`http://localhost:8000/countries/`

- Elle passe par le routeur d'URL du *site* (`mysite/urls.py`) qui référence (include) le routeur d'URL de l'application (`countries/urls.py`) pour toutes les URLs qui commencent par `countries`, elle y est identifiée par la suite (`/`, donc vide)
- Le routeur d'URL de l'application associe cette URL à la fonction `index` définie dans `views.py`
- La fonction vue (pour l'instant) se contente d'envoyer une donnée textuelle constante (même pas du html)
- La suite consiste à renvoyer une page d'accueil plus utile (fonction `index`) et d'ajouter tous les schémas d'URLs menant à des fonctions plus riches (info sur une ville précise)

# Enregistrer notre application dans le projet et créer un gabarit HTML

- **Ajoutez** une ligne enregistrant notre application au niveau du projet (ce qui permet à Django de retrouver tous les fichiers de gabarits automatiquement) dans le fichier `mysite/settings.py` (référence le fichier déjà créé par Django : `countries/apps.py`) :

```
INSTALLED_APPS = [  
    'countries.apps.CountriesConfig',  
    'django.contrib.admin',  
    ...
```

- Créez le **répertoire** des gabarits HTML (dans le répertoire `countries`):

```
$ mkdir countries/templates  
$ mkdir countries/templates/countries
```

- Créez dans `countries/template/countries` le fichier `index.html` suivant ...

# Le gabarit HTML

- Utilise un langage particulier de macro pour afficher des information provenant d'un dictionnaire Python (fichier `countries/templates/countries/index.html`) :

```
{% if countries %}
<h2>Liste des pays</h2>
<ul>
    {% for country in countries %}
    <li>{{ country }}</li>
    {% endfor %}
</ul>
{% else %}
    <p>Aucun pays</p>
{% endif %}
```

# Modifier la vue pour combiner le gabarit HTML avec un dictionnaire Python

- On modifie la vue (fonction) d'index `countries/views.py`, (ajouter/modifiez ce qui est **gras**)

```
from django.template import loader # chargeur de gabarit
# On importe une fonction du modèle métier
from .model.codes import countries_list
def index(request):
    countries = countries_list() # ma fonction du modèle métier
    template = loader.get_template('countries/index.html')
    context = { # les clés correspondent aux étiquettes dans les balises
        'countries': countries
    }
    return HttpResponse(template.render(context, request)) # motif
```

- Créez le répertoire `countries/model` (**NOTRE CHOIX** pour la partie modèle métier)

```
$ mkdir countries/model
```

- Créez un fichier `countries/model/codes.py` (modèle métier !)

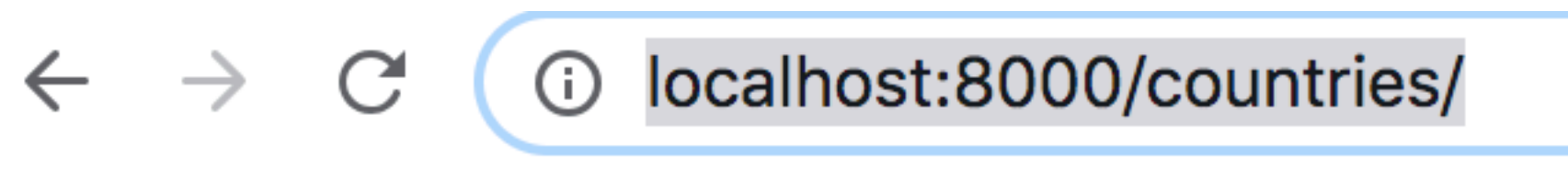
```
def countries_list():
    return [ 'France', 'UK', 'Italie' ]
```

# Le flux application est complet

- La vue combine un dictionnaire obtenu à partir du modèle métier `model/codes.py` avec un gabarit html qui contient des balises ayant des noms correspondant aux clés du dictionnaire passé (*context*)
- On peut maintenant facilement enrichir le modèle métier (base de données SQL) et ajouter des schémas d'URLs correspondant à de nouvelles vues et de nouvelles fonctions ou classes métier.

# La chaîne complète est en place

- En rechargeant la page Web on voit l'effet de la vue qui utilise une fonction du modèle métier pour combiner l'information avec un gabarit HTML :



- France
- UK
- Italie



# Une url pour les infos de chaque pays

- Dans `countries/urls.py` on **ajoute** une ligne qui extrait une information de l'url dans un paramètre si elle la contient : *countries/<Nom du pays>* et associe la vue à créer ensuite :

```
urlpatterns = [ path('', views.index, name='index'),  
                path('<str:country>/', views.show_country_info,  
                    name='country'),  
              ]
```

- Dans `countries/views.py` on *définit* la fonction citée ci-dessus et on importe la fonction métier (tout est à ajouter)

```
from .model.codes import countries_list, country_info  
def show_country_info(request, country): # paramètre transmis par le  
   # mot-clé country cf. ci-dessus  
  
    info = country_info(country)  
    template = loader.get_template('countries/city.html')  
    context = {  
        'name': country,  
        'info': info  
    }  
    return HttpResponse(template.render(context, request))
```

# Une url pour les infos de chaque pays (suite)


- Dans `countries/model/codes.py` on ajoute la fonction métier `country_info` utilisée dans la vue `show_country_info`:

```
def country_info(country):  
    myDB = { 'France': "67,13 millions d'habitants",  
             'UK': "66,02 millions d'habitants",  
             'Italie': "60,59 millions d'habitants" }  
    return myDB.get(country, 'Inconnu')
```

- Le modèle HTML est à créer dans `countries/templates/countries/city.html`:

```
<b>Informations sur {{ name }} : {{ info }}</b>
```

- Les URL comme `http://.../countries/France/` affichent maintenant les informations sur la France (et autres pays)



← → ↻ ⓘ localhost:8000/countries/France/

**Informations sur France : 67,13 millions d'habitants**

# Récapitulons les composants

- L'URL `http://.../countries` est associée à la fonction Python `index` (vue) à travers `urls.py` du projet qui inclut `urls.py` de l'application `countries`
- Cette fonction combine le fichier de gabarit `index.html` (répertoire `templates/countries/`) avec un dictionnaire obtenu en appelant une composante métier importé du module `.model.codes`
- La composante métier est la fonction `countries_list` définie dans `model/codes.py`
- Pour chaque application on a modifié deux fichiers du projet : `mysite/urls.py` et `mysite/settings.py`
- Pour chaque url de l'application *countries* on a :
  - une ligne dans `countries/urls.py`
  - une fonction (vue) dans `countries/views.py` (qui importe des fonctions ou des classes de *model/fichier.py*)
  - une fonction (métier) dans *countries/model/fichier.py*
  - un gabarit HTML dans *countries/templates/countries/fichier.html*

# **Exercice 3.5**

## **Construire une application Django**

# Autres fonctionnalités de Django, autres frameworks

- Django offre de nombreuses autres fonctionnalités décrite dans sa documentations et divers tutoriels
  - Construction automatique de tables SQL et de requêtes à partir d'objets métiers Python
  - Génération et traitement de formulaires Web
- D'autres frameworks Web sont intéressants
  - Flask : utilise des décorateurs pour relier les requêtes HTTP à des fonctions Python
  - Pyramid, CherryPy, ... liste complète sur [wiki.python.org](http://wiki.python.org)

# API Rest

- Une API REST (Representational State Transfer) est souvent basé sur HTTP
- Des requêtes HTTP qui demandent des informations, demande l'ajout d'information, la suppression ou la modification
- GET et POST, souvent aussi DELETE et PUT
  - GET /api/**list** -> renvoie la liste de villes
  - GET /api/**info**/Nantes -> infos sur Nantes
  - GET/DELETE /api/**delete**/Donaldville -> efface Donaldville
  - PUT/POST /api/**add**/Strasbourg -> ajoute Strasbourg  
+ infos dans le corps de la requête
  - POST/PUT /api/**modify**/Strasbourg -> modifie les infos  
+ infos dans le corps de la requête
- Les infos renvoyées par le serveur HTTP ainsi que celle envoyé dans le corps de requêtes sont dans un format normalisé : texte brut, **XML** ou **JSON**

# API : Côté serveur avec le framework Flask

Une API minimale (*list, add, get*) : Installer flask (pip ou à partir de PyCharm)

```
from flask import Flask, jsonify, request

cities = { 'Lyon':'lyonnais', 'Paris':'parisien', 'Nantes':'nantais' }
app = Flask(__name__)

@app.route('/api/list', methods=['GET'])
def get_list():
    return jsonify(list(cities.keys()))

@app.route('/api/add', methods=['POST', 'PUT'])
def add():
    payload = request.json
    cities.update(payload)
    return "Added: {}\n".format(payload)

@app.route('/api/get', methods=['GET'])
def get_none():
    return 'Name Required: /api/get/name\n'

@app.route('/api/get/<string:city>', methods=['GET'])
def get(city):
    info = cities.get(city, 'Unknown')
    return jsonify(info)

if __name__ == '__main__':
    app.run()
```

# Tester l'API en ligne de commande (UNIX/Windows)

- Des outils existent pour envoyer des requêtes HTTP très précises (curl et wget) :

```
$ curl -XGET -H 'Content-Type: application/json' http://localhost:5000/api/list
["Lyon", "Paris", "Nantes"]
$ curl -XPOST -H 'Content-Type: application/json' http://localhost:5000/api/add -d '{"Morlaix": "morlaisiens"}'
$ curl -XGET http://localhost:5000/api/list
["Lyon", "Paris", "Nantes", "Morlaix"]
$ curl -XGET -H 'Content-Type: application/json' http://localhost:5000/api/get/Donaldville
"Unknown"
$ curl -XGET -H 'Content-Type: application/json' http://localhost:5000/api/get/Morlaix
"morlaisiens"
```

- En production on aurait à utiliser sans doute la classe *Value* du module *from multiprocessing* ou une **base de donnée** gérant les transactions et les verrous (pour éviter les soucis en cas d'accès concurrents en écriture)



# Appel d'API : Côté client

Les modules *requests* (et *json*) font presque tout le travail (installer *requests* avec pip ou PyCharm)

```
import requests

with requests.get('http://localhost:5000/api/list') as r:
    if r.ok:
        print(r.json()) # idem json.loads(r.content)
    else:
        print('Erreur: {}'.format(r.status_code))
        exit(1)

with requests.post('http://localhost:5000/api/add',
                   json={ 'Brest': 'brestois' } ) as r:
    if r.ok:
        print('Brest ajouté !')

with requests.get('http://localhost:5000/api/get/Brest') as r:
    if r.ok:
        info = r.json() # idem json.dumps(r.content)
        print('Les habitants de {}' \
              ' sont les {}'.format('Brest', info))
```

- Entrées/sortie sur des fichiers, exceptions et gestionnaire de contextes
- Format de données
- Bases de données
- Interfaces graphiques
- Développement Web
- **Interfaçage C/Python**

# Interfaçage C/Python

- Première approche : le module *cdev*
  - Permet de charger une bibliothèque partagée (`.so` ou `.dll`) en mémoire et d'en appeler les fonctions
  - Ne nécessite pas de disposer des sources de la bibliothèque, uniquement de connaître les prototypes des fonctions
  - Le code qui fait le lien C-Python est écrit côté Python
- Seconde approche : importer `python.h` et écrire le code de lien (*binding*) côté C pour obtenir un module binaire

# Cython

- *Cython* (fork de *pyrex*) permet de traduire du code Python en C
- Supporte un sous-ensemble très large du langage Python
- Des annotations permettent d'utiliser des types C natifs au lieu de types Python à des endroits choisis
- On obtient des performances quasiment identiques à du code C natif

**Demo** : comparaison de code Python avec *CPython* et *pypy*, C avec *cdev*, module écrit en C et *Cython*

# Pour aller plus loin

- *Cython* : un sous-ensemble de Python 3 compilable en C avec la possibilité de spécifier quelle données doivent être stockées dans le type C correspondant et non le type Python. On obtient 99% des performances du C
- *Pypy* : le supposé successeur de CPython qu'on attend toujours, tous les modules passent pas... Il est plus rapide que CPython dans certains cas
- `async/await`, `yield from`, la programmation asynchrone
- **Analyse de données** : <https://bigdata-madesimple.com/step-by-step-approach-to-perform-data-analysis-using-python/>

