Programmation de spécialité : Python

Julien Velcin
https://eric.univ-lyon2.fr/jvelcin/

Plan du cours

- Généralités sur Python
- Eléments de base
- Programmation orientée objets
- Patrons de conception
- Cas pratique : la recherche d'information

Généralités sur Python

Python

```
# Petit script en Python 3
on_commence = input("On commence ? ")
if on_commence == "oui":
    print("C'est parti pour le cours de Python !")
    valeur = 42
print("Et voilà la suite...")
```

- Python est plus qu'un langage de script :
  - *built-in containers*
  - fonctions
  - classes et objets
  - Héritage, polymorphisme…

Pourquoi choisir Python ?

- Langage de haut niveau, interprété
- Open source
- Multi plateformes, inter-opérable
- Large communauté
- Nombreuses librairies

https://www.upgrad.com/blog/python-applications-in-real-world/

Exemples d'applications

- Jeux vidéos
- Développement Web
- Programmation numérique / scientifique
- IA et machine learning
- Apprentissage de la programmation
- Extraction de données depuis Internet
- Manipulation d'images

etc.

Environnements de développement

- Différentes implémentations : Jython, **CPython**, IronPython...
- Distributions de Python :
  - **anaconda** (avec le gestionnaire de paquets : conda)
  - PyPy
  - et bien d'autres...
- Quelques IDE :
  - Eclipse (avec PyDev), Sublim Text, PyCharm, Spyder, **VSC**...
- Environnements virtuels
  - installer les modules séparément

Mise en place

- Installation de la distribution Anaconda
- Logiciels qui nous seront utiles : Spyder, Visual Studio Code
- Création d'un environnement virtuel
- Installation de paquets avec conda

- Conda cheat sheet : https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf

Eléments de base

Programmation de spécialité : Python

Simple script en Python

pour importer un module

```
import sys

def main():
    print("Bonjour tout le monde (spéciale dédicace à Antoine Gourru !)")

    # on peut aussi accéder aux arguments ajoutés en ligne de commande
    args = sys.argv[1:]
    if (len(args)>0):
        print("avec les arguments...")
        for arg in args:
            print(arg)

if __name__ == "__main__":
    main()
```

lorsque votre script est exécuté comme programme principal (pas comme module)

Affichage dans la console

- Plusieurs solutions dans l'utilisation de `print` :

```
import math
x = math.pi
print("Valeur de x : " + str(x))
```

convertir en chaîne (*string*)

```
Valeur de x : 3.141592653589793
```

```
print("Valeur de x : %.2f" % (x))
```

précision

ancienne solution pour formater

```
Valeur de x : 3.14
```

```
print("Valeur de x : {:.2f}".format(x))

print(f"La valeur de x est {x:.2f}")
```

solution avec f-string

Plus de détails se trouvent sur : https://www.python-course.eu/python3_formatted_output.php

Branchements conditionnels : if… else…

```
x = 17
y = 42
if x > y and y != 42:
  print("x est plus grand que y")
elif x == y:
  print("x est égal à y")
elif x < y:
  print("x est plus petit que y")
else:
  print("x est plus grand que y et y=42")
```

Branchements conditionnels : match… case…

- Apparu avec Python 3.10
- Trouver une correspondance avec un motif :

```
s = "voiture"
match s:
    case "avion":
        print("Appareil qui vole")
    case "voiture":
        print("Appareil qui roule")
    case "pieds":
        print("Appareil qui marche")
```

commande *switch* dans d'autres langages de programmation

Boucles : for

```
for x in "informatique":
    print(x)

for x in range(10):
    print(x)

for x in range(3, 10):
    print(x)

for x in range(3, 10, 2):
    print(x)

for x in list(("un", "deux", "trois")):
    print(x)
```

- Possibilité d'utiliser **continue** et **break**

Boucles : while

```
val = int(input("On s'arrête quand ? "))
i = 1
while i < 10:
    print(i)
    if i == val:
        break
    i += 1
else:
    print("La boucle est allée à la fin")
```

- Le else s'exécute à la fin de la boucle
  … sauf si la boucle est interrompue (break, exception)

Structures simples et collections

- Variables
- Listes
- Tableaux
- Ensembles
- Dictionnaires

Variables et listes

- Le type est directement inféré

```
x = 17
nom = "Julien"
```

- Collection simple : la liste

```
villes = ["Lyon", "Bron", "Vénissieux"]
print(villes[0])
print("Taille du tableau : " + str(len(villes)))
print("Dernier élément : " + villes[len(villes)-1])
```

```
Lyon
Taille du tableau : 3
Dernier élément : Vénissieux
```


Liste (suite)

- Accès aux éléments de la liste par index :

```
villes = ["Lyon", "Bron", "Vénissieux", "Caluire"]
villes[0:2]
```

```
Out[2]: ['Lyon', 'Bron']
```

deux premiers éléments

```
villes[1::2]
```

```
Out[5]: ['Bron', 'Caluire']
```

un élément sur deux (à partir de 1)

```
villes[-2]
```

```
Out[10]: 'Vénissieux'
```

élément n°2 à partir de la fin

- La liste peut être naturellement utilisée dans une boucle :

```
for v in ["Lyon", "Bron", "Vénissieux"]:
    print("Ville " + v)
```


Liste (suite)

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Liste (suite)

- Attention, il faut **copier** la liste si on souhaite avoir un nouvel objet, sinon il s'agira d'une référence

```
villes_copy = villes.copy()
del villes_copy[0]
print(villes[0])
print(villes_copy[0])
```

- On peut initialiser une liste à partir d'un **tuple** :

```
ma_liste = list(("Lyon", "Bron", "Vénissieux"))
```

Listes « compréhensibles »

- Manière plus intuitive et compacte de définir une liste

```
S = [ x**2 for x in range(10) ]
V = [ 2**i for i in range(13) ]
M = [x for x in S if x % 2 == 0]


In [75]: print(S, V, M, sep='\n')
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
[0, 4, 16, 36, 64]
```


Ensembles

- Collection **non ordonnée** et **non indexée**, donc impossible de demander un élément en particulier

```
villes = { "Lyon", "Bron", "Vénissieux" }
villes.add("Francheville")
villes.update(["Tassin", "Oullins"])
villes.remove("Vénissieux")
villes.remove("Bron")
autres_villes = { "Vénissieux", "Bron", "Lyon" }
toutes_villes = villes.union(autres_villes)
print(toutes_villes)
```


Dictionnaires

- Accès par clef-valeur

```
client = {
  "nom": "Toto",
  "emploi": "kiné",
  "naissance" : [3, 2, 1970],
  "code" : 69001
}
print(client["nom"])
```

A noter que le dictionnaire ne retient pas l'ordre des clefs (il faut utiliser un OrderedDict)

- Parcourir la collection :

```
for x, y in client.items():
  print(x + " : " + str(y))
```

- Ajout à la volée :

```
client["premium"] = True
```


Tableaux

- Type non natif de Python, nécessite l'appel à une librairie
- Moins souple que les listes mais plus efficace pour les calculs si l'on connaît la taille à l'avance

```
from numpy import array
villes = array(["Lyon", "Bron", "Vénissieux", 42])
type(villes)
print(villes[0])
```

Itérables et énumérations

- Un **itérable** et un objet dont on peut parcourir les valeurs
  - *par ex. :* list, tuple, dict, str

  Vous pouvez implémenter vos propres itérables
- Construction d'une **énumération**
  - fonction pré-construite (built-in) permettant de faciliter l'énumération à partir d'un itérable

```
for i, v in enumerate(villes):
    print("Ville " + str(i) + " : " + v)
```

```
Ville 0 : Lyon
Ville 1 : Bron
Ville 2 : Vénissieux
Ville 3 : Caluire
```


Modules

- Fichier Python que l'on souhaite inclure dans d'autres programmes (fonctions *et* variables)
- Procédure :
  - placer le code dans un fichier monmodule.py
  - importer le module à l'aide de la commande import
- Renommer des modules à l'import :

```
import numpy as np
```

- Importer une partie d'un module :

```
from monmodule import mafonction
```


Fonctions et procédures

```
def mafonction(param1, param2=[], param3=42):
    res = param1 + param3
    param1 *= 2
    param2.append(8)
    return res
```

argument nommé

retourner une valeur n'est pas obligatoire

- Passage par référence pour les objets modifiables (ex. liste)

```
x = 10
y = [10]
print(mafonction(x, y))
print(x, "vs", y)
```

- Retourner plusieurs valeurs, sous forme de liste, tuple, dictionnaire

Note sur le typage

- Depuis la version 3.5, il est possible de typer les variables :

```
def mafonction(p : int, q : str) -> int:
```

- Attention, il s'agit juste d'une annotation

Expressions Lambda

- Permet de définir des fonctions anonymes :

```
x = lambda a, b : a * b
print(x(5, 6))
```

- Ces fonctions peuvent par ex. être implémentées dans d'autres fonctions :

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```


Les chaînes de caractères (str)

- De nombreuses méthodes déjà implémentées, comme :

```
s1 = "Lyon"
s2 = "La ville du bouchon"

s = s1 + ", "  + s2
# ou :
s = ", ".join((s1, s2))
print(s)

# trouver la position d'une sous-chaîne
s.index("ville")
# une exception est levée sinon

# découper la phrase selon un motif
sous_ch = s.split(", ")
print(sous_ch)
```

passer en minuscules avec lower()

→ Lyon, La ville du bouchon

→ ['Lyon', 'La ville du bouchon']

Les expressions régulières en Python

- Utilisation de la librairie re

ignorer la casse

motif

```
import re
p = re.compile("Python[a-z]*", re.IGNORECASE)

ch = "Apprendre le Python et ses codes pythoniques ?"
res = p.finditer(ch)
```

étendue de l'instance : i jusque j

```
for r in res:
    (i, j) = r.span()
    print("Trouvé en position {a} : {b}".format(a=i, b=ch[i:j]))
```

```
Trouvé en position 13 : Python
Trouvé en position 33 : pythoniques
```


*args et **kwargs

*args = nombre variable d'arguments dans une fonction

```
def print_liste(*args):
    for arg in args:
        print(arg)
```

```
In [189]: print_liste(1, "liste", "--", 27.4)
1
liste
--
27.4
```

**kwargs = arguments nommés

```
def print_dico(**kwargs):
    for arg in kwargs.values():
        print(arg)
```

```
In [187]: print_dico(a="Voilà", b="les",
c="arguments")
Voilà
les
arguments
```

Opérations d'entrée-sortie (1)

- écrire dans un fichier :

```
# plus sûr (le "close" est garanti)
with open('mon_fichier', 'w') as f:
    f.write("...")
```

```
f = open('mon_fichier', 'w')
f.write("Nous pouvons sauvegarder\n"
        + "des lignes et des lignes de texte\n")
f.flush()
f.close()
```

- et lire :

```
with open('mon_fichier', 'r') as f:
    for line in f:
        print(line, end="")
```


Opérations d'entrée-sortie (2)

- Pour sérialiser / désérialiser des données : pickle

```
with open('mon_fichier_binaire', 'wb') as f:
    pickle.dump(corpus, f)

with open('mon_fichier_binaire', 'rb') as f:
    objet = pickle.load(f)
```

b pour « binaire »

(attention, pickle n'est pas sécurisé, donc n'ouvrez pas des fichiers dont vous n'êtes pas sûr !)

- Possibilité de manipuler facilement des fichiers .json ou .csv

```
with open('mon_fichier_binaire', 'w') as f:
    json.dump(client, f)

with open('mon_fichier_binaire', 'r') as f:
    x = json.load(f)
```

```
import csv
with open('data/SMSSpamCollection') as csvfile:
    spamdata = csv.reader(csvfile, delimiter='\t')
    for row in spamdata :
        print(' ; '.join(row))
```


Gestion des dates

- Librairie datetime

```
# date d'aujourd'hui
madate = datetime.datetime.now()
# date fixée
madate = datetime.datetime(2020, 5, 28)
# date donnée dans une chaîne de caractère
date_time_str = "28/05/2020"
date_time_obj = datetime.datetime.strptime(date_time_str, '%d/%m/%Y')
```

- On peut ensuite récupérer les différentes informations

```
print("Date : " + str(madate.date()))
print("Heure : " + str(madate.time()))
print("Jour : " + madate.strftime("%d"))
print("Mois : " + madate.strftime("%B"))
print("Année : " + madate.strftime("%Y"))
```

str convertit en *string*

Gestion des erreurs

```
date_time_str = "28/05 2020"
date_time_obj = datetime.datetime.strptime(date_time_str, '%d/%m/%Y')
```

- Il faut anticiper ce genre de problème :

```
try:
    date_time_str = "28/05 2020"
    date_time_obj = datetime.datetime.strptime(date_time_str, '%d/%m/%Y')
except:
    print("La date est mal formatée, on met la date du jour")
    date_time_obj = datetime.datetime.now()
finally:
    print(date_time_obj.date())
```

optionnel

- Il est possible de générer ses propres exceptions avec raise

Analyse de données avec Pandas (1)

- Charger (et sauvegarder) depuis .csv

```
df = pandas.read_csv('data/dataconf.csv', sep="\t")

In [90]: type(df)
Out[90]: pandas.core.frame.DataFrame
```

objet DataFrame, comme dans R !

- Affichage des données

```
# afficher le nom des colonnes (variables)
df.columns
# voir les premières lignes
df.head()
# et les dernières
df.tail()
```

d'ailleurs on retrouve des commandes bien connues...

Analyse de données avec Pandas (2)

- Accéder à une partie du tableau

```
# accéder à une colonne en particulier
df.year # ou df["year"]
# accéder à une partie du tableau
df.loc[:, "title"]
df.loc[df.year==2000, :]
```

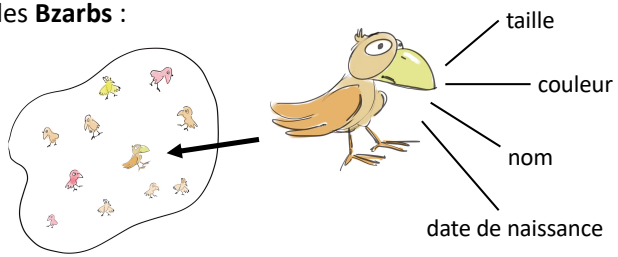
- Mais aussi plein de possibilité pour sélectionner, trier, calculer des statistiques simples, faire des jointures, etc.
  cf. : https://pandas.pydata.org/pandas-docs/stable/getting_started/

Programmation orientée objets

Programmation de spécialité : Python

Classes et objets

- Une **classe** regroupe des objets présentant des caractéristiques similaires *et* au « comportement » similaire
- Prenons le cas des **Bzarbs** :



Constructeur (1)

```
class Bzarbs:

    ''' inutile de déclarer les attributs :
     - nom
     - couleur
     - taille
     - date
    '''
    ### méthodes

    # constructeur le plus simple
    def __init__(self):
        self.nom = "toto"
        self.date = datetime.datetime.now()

    def print(self):
        print("Voilà le bzarb " + self.nom)
```

Le type des variables (champs) sera inféré « à la volée »

commentaires sur plusieurs lignes

Constructeur (2)

- Le constructeur peut comporter des paramètres, avec des valeurs par défaut possibles pour les champs

```
# constructeur avec passage des valeurs initiales
def __init__(self, nom, date=datetime.datetime.now(),
             taille=1.2, couleur="jaune"):
    self.nom = nom
    self.taille = taille
    self.date = date
    self.couleur = couleur
```

Date du jour par défaut

ne pas oublier d'indiquer qu'il s'agit de l'objet courant (self)

Constructeur (3)

- L'ordre ne doit pas nécessairement être respecté si les arguments sont nommés dans le constructeur :

```
b1 = Bzarbs("lolipop")
b2 = Bzarbs("Ploum", couleur="marron", taille=1.8)


In [3]: b1
Out[3]: <__main__.Bzarbs at 0x114ad6a10>

In [21]: b1.print()
Voilà le bzarb lolilo

In [17]: b2.taille
Out[17]: 1.8
```

pointeur (adresse) où se trouve l'objet

Portée des champs (1)

- Les champs sont accessibles à tous par défaut (**public**)

```
b1.__dict__
```

- Une bonne pratique consiste à « cacher » les champs qui ne doivent pas être visibles depuis l'extérieur (une autre classe par exemple)

```
def __init__(self, nom, date=datetime.datetime.now(),
             taille=1.2, couleur="jaune"):
    self.__nom = nom
    self.__taille = taille
    self.__date = date
    self.__couleur = couleur
```

on notera les deux *underscore* : _ _

Portée des champs (2)

- On ne peut plus accéder directement aux champs :

l'erreur génère une exception

```
In [33]: b2_Bzarbs.__nom
Traceback (most recent call last):

  File "<ipython-input-33-70ff72dbb59d>", line 1, in <module>
    b2_Bzarbs.__nom

NameError: name 'b2_Bzarbs' is not defined
```

- Même si rien n'est jamais totalement privé en Python...

```
In [34]: b2._Bzarbs__nom
Out[34]: 'Ploum'
```


Accesseurs / mutateurs

- Permet d'accéder aux champs privés :

```
def getNom(self):
    return self.__nom

def setNom(self, nom):
    if not (nom.lower().startswith("p")
        or nom.lower().startswith("l")):
        return
    self.__nom = nom
```

on a la possibilité d'ajouter des tests (ou d'autres opérations)

Représentation/affichage d'un objet

- La méthode **__str__** construit une représentation de l'objet destinée à l'affichage (par ex. via print)

affichage directement à l'écran

retourne une chaîne

```
def print(self):
    print("Voilà le bzarb " + self.__nom)

def __str__(self):
    return("Voilà le bzarb " + self.__nom)
```

- La méthode **__repr__** construit une représentation de l'objet destinée à l'affichage *du point de vue du programmeur*

```
coll_bzarbs = [b1, b2, b3]
print(coll_bzarbs)
```

utile pour afficher l'objet dans une liste

Variables de classe

- Un champ peut être partagé par tous les objets :

```
class Bzarbs:

    nombreTotal = 0

    ### méthodes
    def print(self):
        print("Voilà le bzarb " + self.__nom)

    # constructeur avec passage des valeurs initiales
    def __init__(self, nom=None,
                 date=datetime.datetime.now(),
                 taille=1.2, couleur="jaune"):
        if nom is None:
            nom = "toto"+str(Bzarbs.nombreTotal)
        self.__nom = nom
        self.__taille = taille
        self.__date = date
        self.__couleur = couleur
        Bzarbs.nombreTotal += 1
```

le nom de la classe doit être ajoutée

Utilisation d'une classe

- A partir d'un autre fichier (script, classe), pensez bien à importer le code Python :

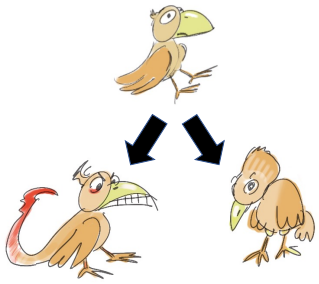
```
import classes_cours as cl

b1 = cl.Bzarbs("lolipop", couleur="marron", taille=1.3)
```


Remarques sur l'importation

- Un module n'est chargé qu'une fois. Il est ensuite mis à jour lorsque le fichier est modifié.
- Pour forcer un rechargement (par ex. pour assurer l'initialisation d'une variable de classe), on peut utiliser la librairie importlib

```
import Bzarbs
from importlib import reload
reload(Bzarbs)
```


Héritage (1)



« évolution »

Bzarb combattant Bzarb penseur

Héritage (2)

```
class Bzarbs:

    nombreTotal = 0

    ### méthodes
    def print(self):
        print("Voilà le bzarb " + self.__nom
              + " de type " + self.getType())

    # constructeur avec passage des valeurs initiales
    def __init__(self, nom=None,
                 date=datetime.datetime.now(),
                 taille=1.2, couleur="jaune"):
        if nom is None:
            nom = "lol"+str(Bzarbs.nombreTotal)
        self.__nom = nom
        self.__taille = taille
        self.__date = date
        self.__couleur = couleur
        Bzarbs.nombreTotal += 1

    def getType(self):
        pass
```

on a rajouté le type ici

cette méthode sera réécrite en fonction du type d'objet

ne fait rien pour le moment

Héritage (3)

on précise le nom de la classe « mère »

```
class BzarbsCombattant(Bzarbs):

    def __init__(self, nom=None,
                 date=datetime.datetime.now(),
                 taille=1.2, couleur="jaune",
                 force=10, endurance=50):
        Bzarbs.__init__(self, nom=nom, date=date, taille=taille,
                        couleur=couleur)
        self.__force = force
        self.__endurance = endurance

    def print(self):
        Bzarbs.print(self)
        print("Force : " + str(self.__force))
        print("Endurance : " + str(self.__endurance))

    def getType(self):
        return "combattant"
```

appeler une méthode de la classe mère on peut aussi utiliser **super()** *mais sans self*

mécanisme de surcharge (*override*)

la méthode « mère » est implémentée

Polymorphisme

```
# exemple simple de polymorphisme
choix = input("Quel bzarb voulez-vous recruter"
              + " (0=combattant, 1=penseur) ? ")
if choix == "0":
    print("Un bzarb combattant !")
    bb = BzarbsCombattant()
else:
    print("Un bzarb penseur !")
    bb = BzarbsPenseur()

bb.print()
```

```
Quel bzarb voulez-vous recruter (0=combattant,
1=penseur) ? 1
Un bzarb penseur !
Voilà le bzarb lol8 de type penseur
Intelligence : 42
```


Design patterns

"Design patterns help you learn from others' successes instead of your own failures." (Mark Johnson)

- Gang of four (1995): Gamma, Helm, Johnson and Vlissides
- Les « patrons de conception » indiquent des bonnes pratiques de programmation :
  - patrons d'usine (*factory*)
  - patrons singleton
  - patrons de décoration
  - etc.

Patron singleton (1)

- Un unique objet existe pour la classe

cette solution passe par le principe de « décorateur » : @

```
def singleton(cls):

    instance = [None]

    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

Patron singleton (2)

- Il suffit alors d'ajouter le décorateur en tête de classe :

```
@singleton
class Collection():

    def __init__(self,name):
        self.name = name
        self.collection = {}
        self.nbzarbs = 0

    def add_bzarb(self, b):
        self.collection[self.nbzarbs] = b
        self.nbzarbs += 1
```


Patron d'usine (*factory pattern*) (1)

- La création des objets est réalisée dans même endroit = l'usine

```
class BzarbsGenerator:

    @staticmethod
    def factory(type, nom):
        if type == "combattant": return BzarbsCombattant(nom)
        if type == "penseur": return BzarbsPenseur(nom)

        assert 0, "Erreur : " + type # si le type entré est inconnu
```

on teste les différents cas possibles

décorateur pour définir une fonction « statique »

génère un message d'erreur le cas échéant

Patron d'usine (*factory pattern*) (2)

- On peut ainsi créer des objets à la chaîne :

```
corpus.add_bzarb(BzarbsGenerator.factory("combattant", "toto"))
```

- Et vérifier qu'ils existent bel et bien :

autre manière d'écrire :
[v for v in corpus.get_coll().values()]

```
for b in [*corpus.get_coll().values()]:
    print(b.getNom())
```

ou directement avec une liste « compréhensible » :

```
[b.getNom() for b in [*corpus.get_coll().values()]]
```


Les décorateurs @

- Permet de modifier le **comportement** d'une classe ou d'une fonction (ce que nous venons de faire avec les patrons)
- Par exemple : calculer le temps d'exécution d'une fonction
- Très bonne explication sur OpenClassRooms : https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/233491-apprehendez-les-decorateurs

Tests unitaires

```python
import unittest
import mesclasses as mc

class BzarbsTest(unittest.TestCase):

    def test_name(self):
        b = mc.BzarbsCombattant("lolipop")
        self.assertEqual("lolipop", b.getNom())

    def test_liste_deux(self):
        b1 = mc.BzarbsCombattant("lolipop")
        b2 = mc.BzarbsPenseur("tutu")
        col = [b1, b2]
        self.assertLessEqual(len(col), 2)

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

de nombreuses fonctions disponibles

exécute toutes les fonctions préfixées par test

Quelques sources fiables

Eléments basiques en Python :

https://docs.python.org/3.7/tutorial/index.html
https://www.w3schools.com/python/

Notions plus avancées (générateurs, listes compréhensives, patrons...) :

https://python-3-patterns-idioms-test.readthedocs.io/en/latest/index.html