Séquence 3 – La Programmation Orientée Objet (POO) et autres paradigmes de programmation

Objectifs
1. Distinguer sur des exemples les paradigmes impératif, fonctionnel et objet. 2. Choisir le paradigme de programmation selon le champ d'application d'un programme. 3. Écrire la définition d'une classe 4. Accéder aux attributs d'une classe 5. Accéder aux méthodes d'une classe 6. Distinguer sur des exemples les paradigmes impératif, fonctionnel et objet. 7. Choisir le paradigme de programmation selon le champ d'application d'un programme.

Cette séquence s'appuie sur :

- https://www.lecluse.fr/nsi/NSI_T/langages/paradigmes/
- https://pixees.fr/informatiquelycee/n_site/nsi_term_paraProg_poo.html
- https://isn-icn-ljm.pagesperso-orange.fr/NSI-TLE/res/res_les_classes.pdf

❶	Objectif : Écrire la définition d'une classe Accéder aux attributs d'une classe Accéder aux méthodes d'une classe Modalités : Utilisation d'un IDE python sur Raspberry	☐	☐
❶	## 1 Introduction La programmation orientée objet repose, comme son nom l'indique, sur le concept d'objet. Un objet dans la vie de tous les jours vous connaissez, mais en informatique, c'est un nouveau concept. Imaginez un objet, par exemple un moteur de voiture : il est évident qu'en regardant cet objet, on est frappé par sa complexité.	☐	☐

Imaginez que l'on enferme cet objet dans une caisse et que l'utilisateur n'ait pas besoin d'en connaître son principe de fonctionnement interne pour pouvoir l'utiliser. L'utilisateur a, à sa disposition, des boutons, des manettes et des écrans de contrôle pour faire fonctionner l'objet, ce qui rend son utilisation relativement simple.

La mise au point de l'objet (par des ingénieurs) a été très complexe, en revanche son utilisation est relativement simple.

Programmer de manière orientée objet, c'est un peu reprendre cette idée : utiliser des objets sans se soucier de leur complexité interne.

Un des nombreux avantages de la Programmation Orientée Objet (POO), est qu'il existe des milliers d'objets prêts à être utilisés. On peut réaliser des programmes extrêmement complexes uniquement en utilisant des classes préexistantes.

2 Les classes Python

Jusqu'à présent nous avons utilisé des objets dont le type est prédéfini : int,float,bool,str,list,tuple,dict

Nous avons même appris à utiliser des méthodes associées à ces types, comme *ma_liste.append(valeur).*

Python, comme d'autres langages (Java, C++, ...), est un langage orienté objet. **On peut même dire que tout y est objet.**

Une variable de type int est en fait un objet de type int donc construit à partir de la classe int. Pareil pour les float et string. Mais également pour les list, tuple, dict, etc.

2.1 Créer une classe et des instances

La mot-clé Python est *class*.

A faire vous même 1.

- Ecrivez votre première class, la plus simple possible

```
class Voiture:
    pass # signifie ne fait rien...
```

- Cette classe, vous pouvez l'instancier :

```
>>> voiture_de_leo=Voiture()
>>> voiture_de_Anna=Voiture()
```

On a une sorte de "moule" et avec on fabrique des voitures.

- Il n'est plus de type int, float, list ou autre, il est de type Voiture

>>> voiture_de_leo

<class'__main__.Voiture'>

- A noter : Pour poser les bases avant de programmer, il est possible d'utiliser le mot-clé python *pass* qui ne fait rien mais qui évite au code de se mettre en erreur.

2.2 Les attributs de classe

Un objet a une apparence, des caractéristiques qui lui sont propres : rouge, bleu, en plastique, 15 cm de hauteur, ... Ces caractéristiques sont des attributs.

Il y en a de deux types:

- Les attributs de classe : Ce sont des attributs qui seront identiques pour toutes les instances et n'ont pas vocation à être changés.
- Les attributs d'instance : Ils sont spécifiques à cette instance. Et d'une instance à l'autre il ne prendra pas forcément la même valeur.

A faire vous même 2.

- Complétez votre classe

```
class Voiture:
    # attributs de classe
    matière="acier"
    nombre_de_places=5
    longueur=4.5
```

- Cette classe, vous pouvez l'instancier :

  >>> voiture_de_Peter=Voiture()
- Vous avez accès à ses attributs de classe en écrivant le nom de l'instance UN POINT le nom de l'attribut :

  >>>voiture_de_Peter.matière

  "acier"
- Toutes les instances de Voiture auront ces mêmes attributs
- Affichez les 2 autres attributs de classe

2.3 Les attributs d'instance et le constructeur

Quand on créé une classe, on écrit une sorte de fonction spéciale (en fait c'est une méthode) appelée le *constructeur*.

Il est implicitement exécuté lors de la création de chaque instance.

Son nom est imposé : __init__ (2 underscores de chaque côté).

Son premier paramètre est *self*.

Ensuite, vient la liste des autres paramètres.

A noter : Il n'y a pas de *return* (pas de valeur retournée).

❶ **A faire vous même 3.** ☐ ☐

- Complétez votre classe

```
class Voiture:
    # attributs de classe
    matière="acier"
    nombre_de_places=5
    longueur=4.5
    def __init__(self, couleur_utilisateur):
        self.couleur = couleur_utilisateur
```

- Cette classe, vous pouvez l'instancier :

  >>> voiture_de_Titouan=Voiture()

  TypeError: __init__() missing 1 required positional argument: 'couleur_utilisateur'

  Il y a erreur, le constructeur a besoin absolument d'un paramètre.

  >>> voiture_de_Titouan=Voiture("bleu")

- Vous avez accès à ses attributs d'instance :

  >>> voiture_de_Titouan.couleur

  "bleu"

- Chaque instance de Voiture peut avoir ses propres attributs d'instance

  >>> voiture_de_Jackie=Voiture("rouge")

  >>> voiture_de_Jackie .couleur

  "rouge"

❶ ## 2.4 Les méthodes ☐ ☐

Un objet est (rouge, bleu, 15 cm de haut, ...) mais aussi FAIT/AGIT/ACTIONNE (démarre, roule, s'arrête, modifie, ...).

Une méthode est composée de commandes python et s' exécute. Il faut donc mettre des parenthèses.

Cela s' écrit comme une fonction qui fait partie de l'objet.

Comme dit plus haut, le *constructeur* est une méthode. Celle-ci n'a pas besoin d'être lancée, elle s' exécute automatiquement à la création.

❶ **A faire vous même 4.** ☐ ☐

- Complétez votre classe

```
class Voiture:
    # attributs de classe
```

```
        matière="acier"
        nombre_de_places=5
        longueur=4.5

        def __init__(self, couleur_utilisateur, nombre_de_km=0,
    a_charger=[ ]):
            self.couleur = couleur_utilisateur
            self.km=nombre_de_km
            self.coffre= a_charger[ :]

        def avance(distance) :
            self.km+=distance

        def charge(a_charger=[ ]) :
            self.coffre+= a_charger
```

- Cette classe, vous pouvez l' instancier et la manipuler :

```
>>> voiture_de_Ines=Voiture('verte')
>>> voiture_de_Alex=Voiture('noir', a_charger=['valise', 'caisse
en carton', 'livres'])
>>> voiture_de_Alex.km
0
>>> voiture_de_Alex.avance(77)
>>> voiture_de_Alex.km
77
>>> voiture_de_Alex.coffre
['valise', 'caisse en carton', 'livres']
>>> voiture_de_Alex.charge(['parapluie', 'poussette'])
>>> voiture_de_Alex.coffre
['valise', 'caisse en carton', 'livres', 'parapluie', 'poussette']
```

❶ **A faire vous même 5.** ☐ ☐

- Ajoutez une méthode estNeuve qui retourne True si le kilométrage est égal à 0 ou sinon False.
- Ajoutez une méthode videLeCoffre qui vide la liste correspondant au coffre.
- Il existe des méthodes cachées qui sont disponibles automatiquement pour toutes les classes. Par exemple : __str__ et __expr__. Elles sont toutes précédées et suivies de double underscores
- Ajoutez une méthode __str__ Cherchez comment fonctionne cette méthode et écrivez-la.

❶ ## 2.5 Encore et toujours la documentation ☐ ☐

Pour manipuler des objets, une documentation bien rédigée est vraiment nécessaire.

❶ **A faire vous même 6.**

```
class Voiture:
    """
    Permet de modéliser une voiture
    """
    matière="acier"
    nombre_de_places=5
    longueur=4.5

    def __init__(self, couleur_utilisateur, nombre_de_km=0,
a_charger=[ ]):
    """
    Constructeur
    Un arg. pos. – chaine de caractères pour la couleur de la voiture
    Deux arguments nommés :
    * nombre_de_km : Entier pour le nombre de km du véhicule
    * a_charger : Liste pour tous les objets à mettre dans le coffre
    """
        self.couleur = couleur_utilisateur
        self.km=nombre_de_km
        self.coffre= a_charger[ :]
```

- Complétez les autres méthodes de votre classe avec de la documentation
- Ajoutez-y quelques jeux de tests. A vous de voir s'il faut le mettre au niveau de la classe ou au niveau des méthodes.

❶

3 Mini-projet : Une classe Vecteur

Il va falloir vous rappeler ce que vous avez appris sur les vecteurs et leur manipulation. Si besoin, demandez au professeur des précisions sans handicap pour la note.

1. Créez une classe Vecteur2D pour un vecteur dans le plan
2. Créez un constructeur avec ses composantes en x et en y
3. Créez une méthode Norme qui retourne la norme du vecteur
4. Créez une méthode Multiplie qui modifie le vecteur en le multipliant pour un nombre donné en argument
5. Créez une méthode Ajoute qui modifie le vecteur en lui ajoutant un autre objet Vecteur donné en argument
6. Créez une méthode EstPerpendiculaire qui retourne True si un autre objet Vecteur donné en argument est perpendiculaire et sinon False
7. Créez une méthode EstColinéaire qui retourne True si un autre

objet Vecteur donné en argument est colinéaire et sinon False

8. Modifiez la méthode __str__ qui permette d'avoir une sortie qui ressemble à ceci :

   >>> print(mon_vecteur)

   vecteur de coordonnées : x=4 et y=5.6

9. Etudiez les méthodes __mul__ et __add__ et faites-les fonctionner pour les vecteurs

10. Mettez au point la documentation et les jeux de tests

❷ Objectif :

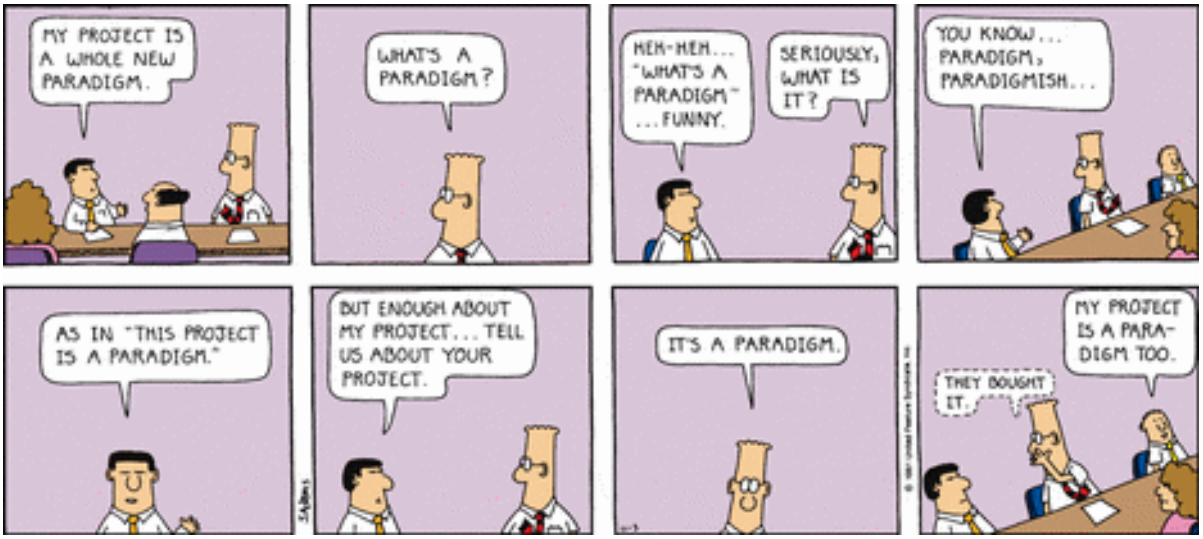
Distinguer sur des exemples les paradigmes impératif, fonctionnel et objet.

Choisir le paradigme de programmation selon le champ d'application d'un programme.

Modalités : Utilisation d'un IDE python sur Raspberry ☐ ☐

❷ ☐ ☐

4 Paradigmes de programmation



Le nombre de langages de programmation est gigantesque : on en dénombre plus de 2000. Pourquoi autant de langages ? Comment les choisir ?

Il y a bien sûr des langages plus populaires que d'autres - reste bien sûr à définir la notion de *populaire*. Si on prend comme critère les recherches des développeurs sur les forums d'entraide, on obtient ce classement qui fait de Python le langage le plus populaire.

http://pypl.github.io/PYPL.html

Python rentre dans la catégorie des langages généralistes : il peut s'adapter à beaucoup de situations et de *paradigmes* différents.

❷

4.1 Quelques exemples de paradigmes de programmation

Dans certaines situations, on va choisir un langage particulier parce qu'il est plus adapté qu'un langage généraliste pour effectuer certaines tâches. Pour prendre un exemple un peu extrême : pour gérer efficacement l'accès à un grand nombre de données, on va privilégier un langage *orienté requêtes* comme **SQL** plutôt que python.

Voici quelques exemples de paradigmes : Impératif, Orienté Objets, Fonctionnel, Logique, événementiel, Concurrent, Orienté requêtes etc... Cette liste n'est pas exhaustive.

❷

4.2 Programmation Impérative

Le paradigme Impératif est celui avec lequel vous avez probablement découvert la programmation : on y trouve toutes les structures comme les boucles (*for*, *while*), les conditionnelles (*if*, ...), les variables, les tableaux...

Python est bien sûr un langage impératif.

Le langage **BASIC** (Beginner's All-purpose Symbolic Instruction Code)) est un langage développé en 1963 et qui, comme son nom l'indique, était plutôt dédié aux débutants. Dans ce langage, les fonctions n'existent pas (il faut utiliser à la place des **GOSUB... RETURN**, et les lignes d'un programme doivent être numérotés. par ailleurs, l'absence de la notion de bloc d'instruction oblige à utiliser les **GOTO** à haute dose ce qui rend toute programmation rigoureuse et structurée extrêmement difficile.

Voici un exemple qui calcule les termes de la suite de Fibonacci (https://fr.wikipedia.org/wiki/Suite_de_Fibonacci) :

```
10 CLS
20 PRINT "PJ DEMO"
30 PRINT
40 LET I = 10
50 GOSUB 80
60 PRINT "RESULTAT : "  + R
70 END
80 IF I > 1 THEN GOTO 110
90 LET R = 1
95 PRINT "R=" + R
100 RETURN
110 LET A = 1
120 LET B = 1
```

```
130 FOR J = 2 TO I+1
140 LET T = A
150 LET A = B
160 LET B = A + T
170 PRINT T + "-" + A + "=>" + B
180 NEXT J
190 LET R = B
200 RETURN
```

❷ **A faire vous même 7.**

Testez le programme ci-dessus sur le simulateur de langage Basic :

http://www.quitebasic.com/

Pour aller plus loin : www.quitebasic.com/help/

☐ ☐

❷

4.3 Paradigme orienté Objets

Nous avons abordé la programmation orientée objet. Les langages qui permettent de manipuler les classes, propriétés et méthodes rentrent dans cette catégorie des langages objets. Le langage **C** est un langage impératif, mais pas orienté objet. Le langage **C++** est une évolution du langage **C** pour prendre en charge le paradigme objet. Les idées sous-tendant le paradigme objet datent des années 60. Mais il faudra attendre le début des années 70 et la mise au point du langage Smalltalk pour que le paradigme objet gagne en popularité chez les informaticiens. Ajourdh'hui de nombreux langages permettent d'utiliser le paradigme objet : C++, Java,...

☐ ☐

❷

4.4 Programmation événementielle

Vous avez étudié l'an passé le langage Javascript, spécifiquement conçu pour rendre les pages webs plus interactives. Il est donc optimisé pour répondre aux événements qui peuvent survenir sur une page web. Javascript obéit - entre autres - au paradigme de programmation événementielle.

Le langage **Scratch** est au autre exemple de langage événementiel dans lequel on associe des blocs de code à des actions de l'utilisateur.

Le développement d'interfaces graphiques (par exemple TKinter en Python) fait largement appel à la programmation événementielle.

☐ ☐

❷ Il ne s'agit pas ici de fournir une liste exhaustive de tous les paradigmes de programmation, simplement de vous sensibiliser au fait qu'il en existe de nombreux, parfois très différents (comme sont différents SQL et python). Certains problèmes sont particulièrement

☐ ☐

adaptés à l'utilisation de certains paradigmes, ce qui conditionne alors le choix du langage. Nous allons approfondir deux paradigmes qui se détachent un peu de ce que vous avez déjà pu rencontrer :

❷ 4.5 Programmation concurrentielle ou parallèle ☐ ☐

Un aperçu de la programmation parallèle en Python est donné ici : illustration de l'interblocage. La grosse problématique avec la programmation parallèle est le partage des ressources entre plusieurs processus. Certains langages sont mieux armés pour traiter efficacement cette problématique. Nous ne nous étendrons pas davantage sur ce sujet délicat.

❷ 4.6 Programmation Logique ☐ ☐

Certains langages sont spécialement conçus pour traiter des problèmes logiques, voire faire des preuves mathématiques. On peut citer **Prolog** ou **Coq** pour les preuves.
Développons le langage **Prolog** : ce langage a été créé en 1972 par un Français : *Alain Colmerauer*. Il est surtout utilisé dans le domaine de l'intelligence artificielle, mais aussi dans le traitement du langage.
Dans Prolog, un programme est en fait une base de faits et règles, et le programme est en quelque sorte exécuté en posant une question dans l'interpréteur.
Sans faire un tutoriel, regardons juste un exemple qui montre le fonctionnement totalement différent de ce langage très spécial :

❷ 4.6.1 Le problème à résoudre ☐ ☐

Les données du problème sont :

- Max a un chat.
- Eric n'est pas en pavillon.
- Luc habite un studio mais le cheval n'y est pas.
- Chacun habite une maison différente et possède un animal distinct.

La problématique à résoudre est : Qui habite le château et qui a le poisson ?

4.6.2 Le programme en Prolog

Source : http://www.gecif.net/articles/linux/prolog.html

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Réalisé par Jean-Christophe MICHEL
% Juillet 2011
```

```
% www.gecif.net
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------
% les faits :

% les 3 maisons :
maison(chateau).
maison(studio).
maison(pavillon).

% les 3 animaux :
animal(chat).
animal(poisson).
animal(cheval).

%-------------------------------------------------------
% les règles :

% le prédicat relation constitue la relation entre une
personne, son animal et sa maison :
relation(max,M,chat):-maison(M).
relation(luc,studio,A):-animal(A),A\==cheval.
relation(eric,M,A):-maison(M),M\==pavillon,animal(A).

% le prédicat different est vraie seulement si ses 3
paramètres sont différents :
different(X,X,_):-!,fail.
different(X,_,X):-!,fail.
different(_,X,X):-!,fail.
different(_,_,_).

% le prédicat "resoudre" indique les 4 inconnues à
retrouver :
resoudre(MM,ME,AE,AL):-

    relation(max,MM,chat),
    relation(eric,ME,AE),
    different(MM,ME,studio),
    relation(luc,studio,AL),
    different(AE,AL,chat).
```

4.6.3 La solution

Pour résoudre le problème, on tape `resoudre(MM,ME,AE,AL).` ce qui donne :

```
MM = pavillon,
ME = chateau,
AE = cheval,
AL = poisson .
```

qui s'interprète ainsi :

- la maison de Max (MM) est un pavillon
- la maison d'Eric (ME) est un chateau
- l'animal d'Eric (AE) est le cheval

- l'animal de Luc (AL) est le poisson

Sans chercher à rentrer dans les détails, un survol de cet exemple vous montre comment à partir de faits et de règles, le langage trouve une solution à un problème. Il utilise pour cela un **moteur d'inférence** qui simule des raisonnements déductifs.

❷ **A faire vous même 8.** ☐ ☐

* Livre Hatier Prépabac NSI term P. 77 ex 6

❷ ☐ ☐

4.7 Programmation fonctionnelle

4.7.1 Le principe du λ-calcul

Les langages fonctionnels sont des langages qui reposent sur le λ-calcul, créé par Church en 1925. Le principe du λ-calcul consiste à considérer les fonctions comme des données comme les autres. Ce concept ne vous est pas étranger.

Exemple : en λ-calcul, la fonction mathématique $x \longmapsto x+1$ s'écrirait $\lambda x \cdot (x+1)$

Python peut faire du λ-calcul comme le montre cet exemple :

```
>>> ma_fonction = lambda x:x+1
>>> ma_fonction(4)
```

Cet appel renvoie 5.

`ma_fonction` est une variable, déclarée comme telle, mais c'est aussi une fonction. Vous remarquerez au passage à quel point une fois de plus la syntaxe python est proche de la syntaxe mathématique !

❷ ☐ ☐

4.7.2 Les avantages de la programmation fonctionnelle

Les langages fonctionnels présentent en général les particularités suivantes :

1. Les données sont **immuables** : leur valeur n'est jamais modifiée ; les fonctions peuvent créer de nouvelles données, mais pas en modifier. On obtient ainsi un code beaucoup plus sûr ;
2. Les **effets de bord** sont clairement identifiés et en général séparés du cœur du langage : On sait ce qui rentre dans une fonction, ce qui en sort et ce que fait la fonction. En effet, ce sont eux qui peuvent entraîner un comportement erratique ;
3. Il est possible de mettre en place un mécanisme d'**évaluation**

paresseuse : les nouvelles données ne sont calculées que lorsque leur valeur devient nécessaire. Cela évite de calculer des données qui, au final, ne serviraient pas ;

4. Un mécanisme, appelé **inférence de type** permet à l'interpréteur du langage de déduire le type des données et fonction sans qu'il soit nécessaire de le spécifier.

❷ A faire vous même 9. ☐ ☐

* Livre Hatier Prépabac NSI term P. 77 ex 5

❷ ☐ ☐

4.7.3 quelques langages fonctionnels populaires

Un des premiers langages fonctionnel est **Lisp** (1958). Sa syntaxe particulière à base de parenthèses est très reconnaissable :)

```
(de inter (L M)
   (cond
      ((null L) nil)
      ((member (car L) M) (cons (car L) (inter (cdr L) M)))
      (t (inter (cdr L) M)) ))
```

Les poids lourds de la catégorie sont **CAML** (1985) et le plus récent **Haskell.**

La encore ce sont des langages à la syntaxe très particulière comme le montre cet exemple en **Haskell** qui calcule la somme des termes d'une liste :

```
somme liste = if (length liste == 0) then 0 else head liste +
somme (tail liste)
```

Si vous trouvez que Python a une syntaxe difficile, essayez les langages fonctionnels :)

❷ ☐ ☐

4.8 Conclusion

Des langages généralistes comme python permettent au développeur d'utiliser beaucoup de paradigmes. Cela peut paraître être la panacée : Si Python fait tout avec une syntaxe simple, pourquoi créer des langages spécialisés comme Haskell à la syntaxe obscure ? Avec Python On peut faire de la programmation fonctionnelle mais on peut aussi comme on vient de le voir faire des entorses à ce paradigme fonctionnel ! Laisser au développeur la possibilité de créer des variables globales et de faire des effets de bords peut le mener à des bugs parfois difficiles à déceler.

Un langage purement fonctionnel comme Haskell ne tolérera pas ces entorses et oblige donc à une rigueur absolue. Le développeur y gagne à terme car son code est plus fiable, moins susceptible d'avoir

	des bugs donc plus fiable. Un langage généraliste permet de tout faire, y compris des choses pas propres ! En fonction des projets le développeur fera des choix de paradigmes pertinents ce qui pourra le conduire naturellement vers tel ou tel langage.		

❶❷❸❹❺❻❼❽❾