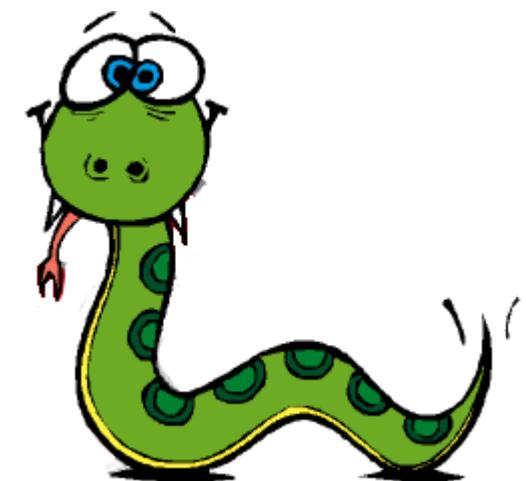
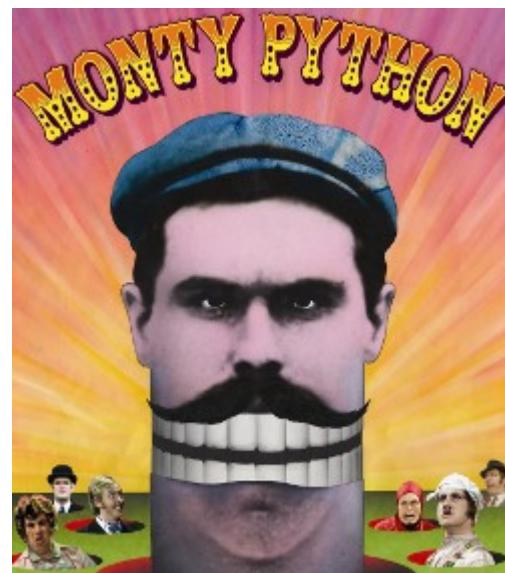


Initiation au langage Python



gael.pegliasco@makina-corpus.com

Sur une idée originale de Michel Matsumoto (Nokia)

Votre formateur

- Gaël Pegliasco <gael.peglasco@makina-corpus.com>
- Développe en Python depuis 2004
- Applications web : Django, Plone
- Calcul scientifique : Numpy, Scipy, Pandas, Matplotlib, Celery



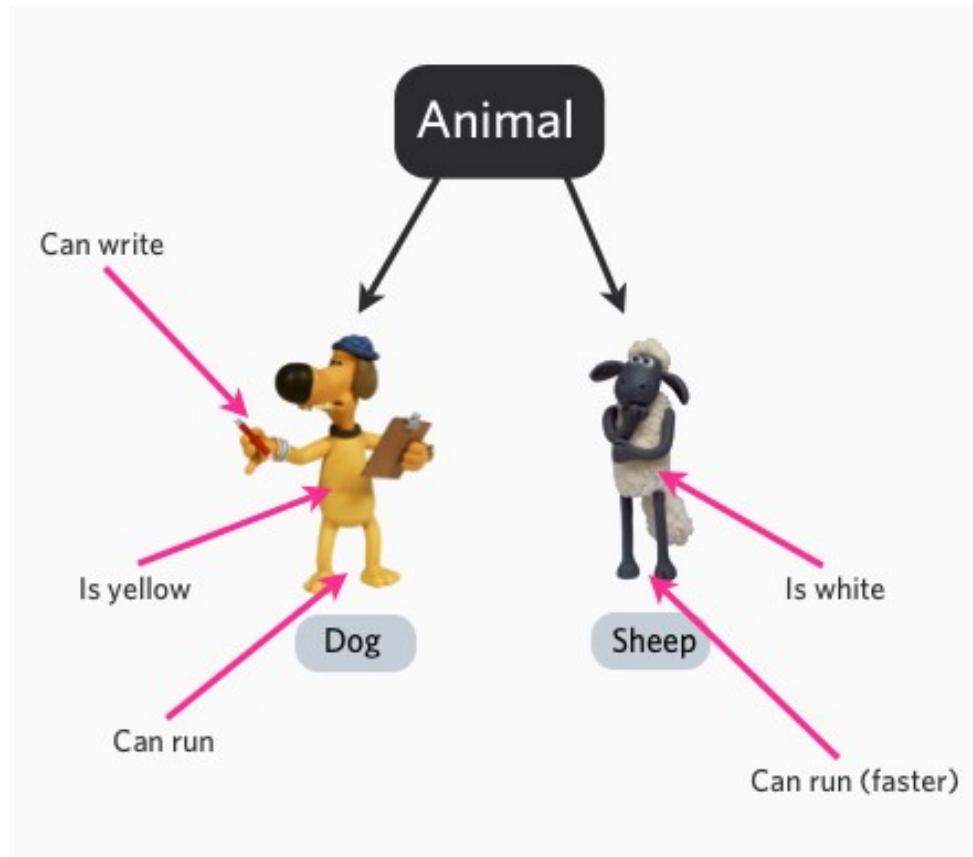
**Il est gentil et aura le plaisir de vous guider
dans la découverte de ce langage fabuleux...**



Plan de formation

- Présentation & Historique
- Syntaxe et bases du langage
- **Approche Orientée Objet**
- **Programmation Orientée Objet en Python**
- Librairies standard
- Outils de qualité et tests
- IHM TkInter
- Interfaçage avec le langage C

Programmation Orientée Objets



Approche Orientée Objet

Découverte par l'exemple

Sur une idée originale de Michel Matsumoto, Nokia, France

POO – Découverte par l'exemple

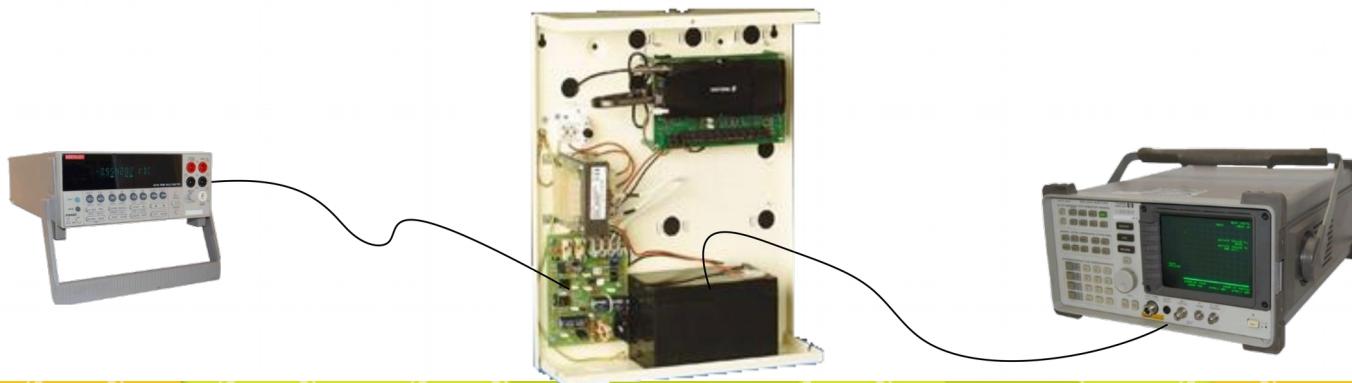
Vous êtes devant votre banc de mesure et vous avez décidé qu'il était grand temps de passer à la vitesse supérieure:

- faire en sorte que les mesures rébarbatives et répétitives soient réalisées automatiquement et que votre tache soit consacrée ... à la sieste !
- Votre chef veux des résultats – « bons » tant qu'à faire - alors, que ce soit vous ou votre banc automatisé qu'il les réalise ...

Vous avez 2 types d'équipements de mesure différents :

- un multimètre de table
- un analyseur de spectre

Les équipements sont tous les deux branchés sur votre appareil en test de la façon suivante:

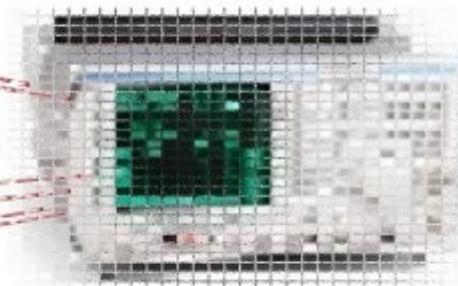


POO – Découverte par l'exemple

- Parce que vous êtes curieux, vous savez que vos équipements (même les anciens !) peuvent se piloter et être programmés via des commandes SCPI
https://fr.wikipedia.org/wiki/Standard_Commands_for_Programmable_Instruments
- Comme vous êtes encore plus curieux, vous savez que Python offre plusieurs librairies qui permettent de se connecter directement avec vos équipements via l'interface Visa
<https://pyvisa.readthedocs.org/en/stable>

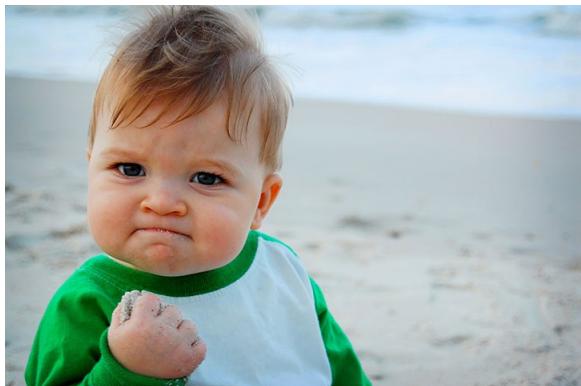


[SOURCE:] FREQUENCY [: SOURCE:] POWER: ALC [: SOURCE:]
[ROUTE:] CLOSE: STATUS [: ROUTE:]
OUTPUT [: STATUS]
SYSTEM: CAPABILITY
SYSTEM: ERROR [: NUMBER]
[ROUTE:] CLOSE: STATUS
INPUT [1] : COUPLING
[SENSE:] FUNCTION: CONDUCTANCE
INITiate [: IMMEDIATE]
TRIGger [: SEQUENCE]
TRIGger [: SEQUENCE]



POO – Découverte par l'exemple

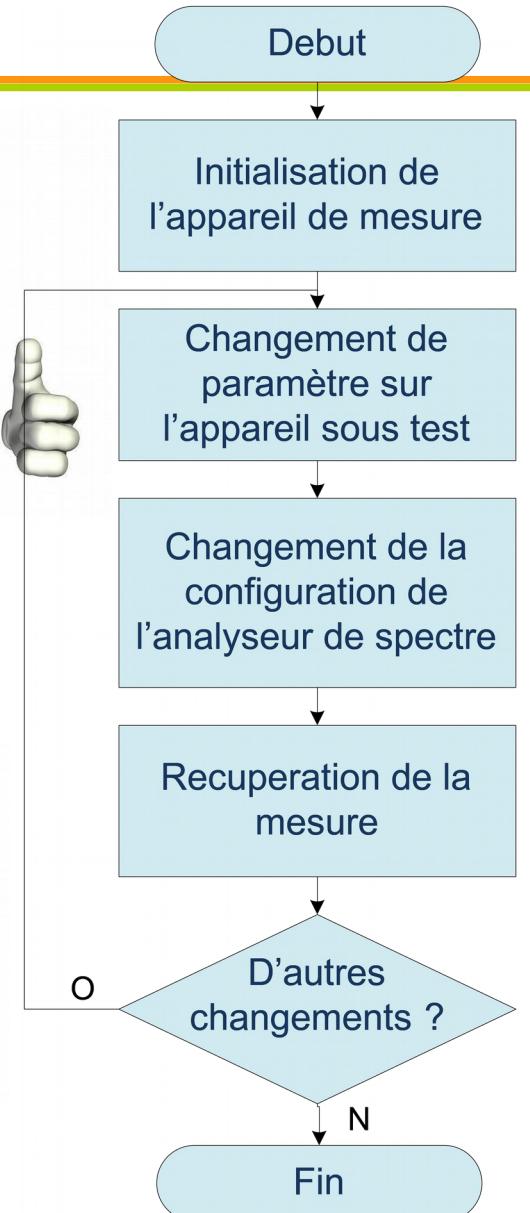
- Vous allez donc concevoir un programme qui permet :
 - de faire des relevés de mesures sur les deux équipements,
 - de changer un paramètre sur votre appareil sous test (disons : la fréquence) puis recommencer la mesure et ainsi de suite pendant 50 changements de fréquence.
- Qui dit changement de fréquence sur l'appareil en test dit également changement de fréquence sur l'analyseur de spectre (bien sur)



POO – Découverte par l'exemple

L'algorithme est très simple:

- Le programme est assez vite réalisé et d'autres personnes, sur d'autres bancs de mesures autour de vous voudraient bien récupérer ce superbe outil !
- Qu'à cela ne tienne, vous le leur installez.
- **Mais vous vous apercevez vite que cela ne fonctionnera pas:** votre collègue n'a pas le même équipement de mesure que vous !

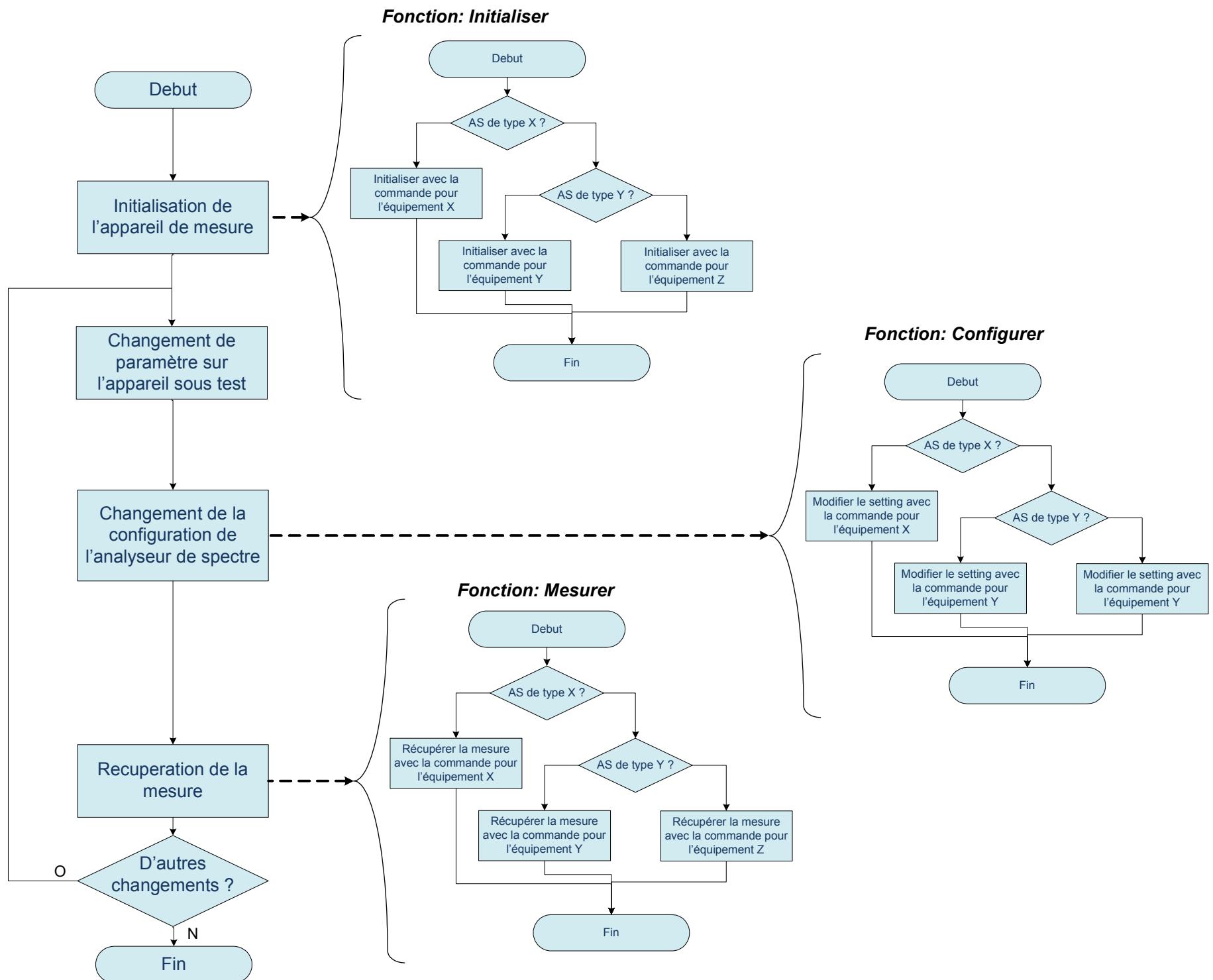


POO – Découverte par l'exemple

- Qu'à cela ne tienne, vous faites une version pour un Analyseur de Spectre Y.
- Et puis rebelote: un autre collègue vous demande votre logiciel et lui aussi a un AS différent du vôtre ou du type Y.
- Vous déclinez à nouveau votre outil pour un AS de type Z
- A ce stade, votre outil commence à ressembler à une usine à gaz avec des tests imbriqués sur le type d'équipement;

Quelque chose comme:





POO – Découverte par l'exemple

- N'y aurait-il pas une méthode qui permettrait de se « passer » de toutes ces tests « if ... then ... elseif ... then elif ... » ?
- Une méthode qui permettrait de manipuler un « type d'analyseur de spectre » sans se soucier de « comment dialoguer avec » ?
- Une méthode qui me permettrait de rajouter des AS différents sans modifier le programme de base ?

Disons que la POO répond très bien à ce genre de requêtes ...



POO – Découverte par l'exemple

Avec la Programmation Orientée Objets vous allez raisonner différemment :

- Vous allez créer un type de données générique « Analyseur de Spectre » qui possédera les fonctions :
 - Initialiser
 - Configurer
 - Mesurer
- Puis vous allez décliner ce type générique en sous ensembles, un par analyseur spécifique :
 - AS X
 - AS Y
 - AS Z

POO – Découverte par l'exemple

Chaque analyseur de spectre spécifique aura accès à tout ce qui est défini dans l'ensemble générique, soit les 3 fonctions :

- Initialiser
- Configurer
- Mesurer

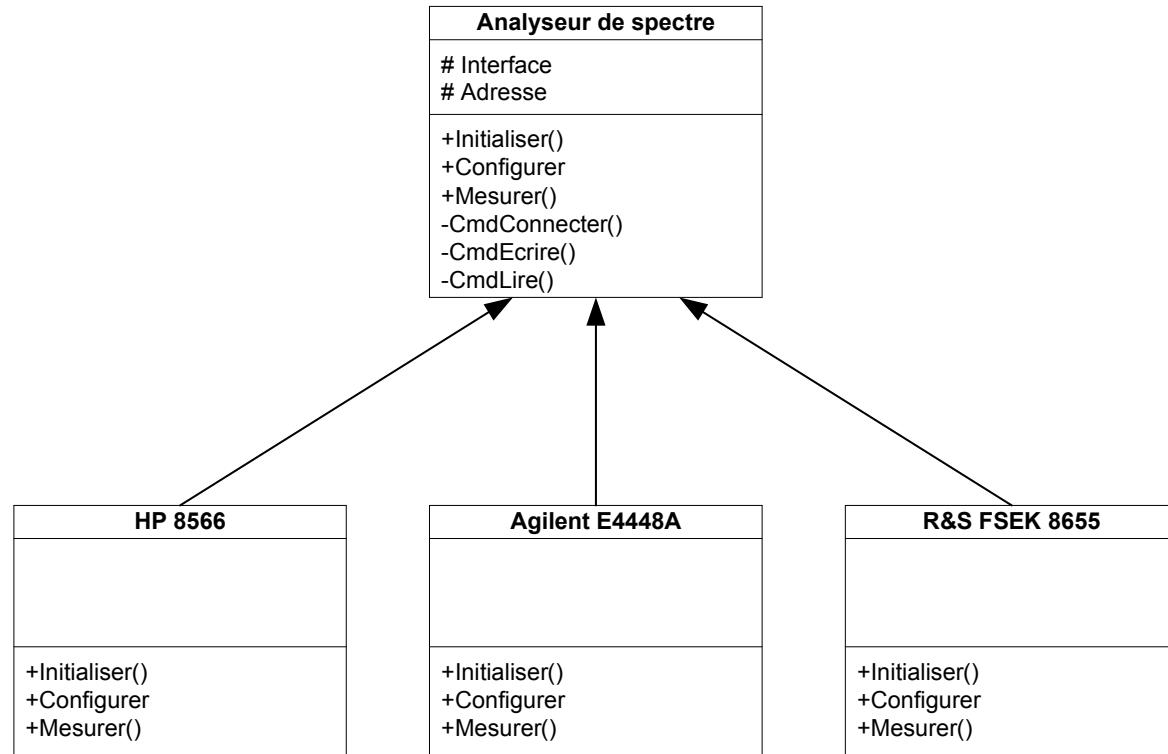
Mais :

- Chaque AS spécifique pourra personnaliser (créer sa propre version) de ces fonctions si la version générique n'est pas adaptée à son fonctionnement
- Et c'est le langage (Python ici) qui choisira automatiquement quelle est la bonne fonction à utiliser selon l'analyseur de spectre manipulé

POO – Découverte par l'exemple

Pour cela nous allons :

- Définir une classe de type AnalyseurDeSpectre qui contiendra tous les attributs (caractéristiques) et méthodes (fonctions) propres à ce type d'équipement de mesure
- Décliner 3 sous ensembles d'AS de marques différentes.
Chaque marque héritant des propriétés de la classe AnalyseurDeSpectre générique



POO – Découverte par l'exemple

- La classe de base AnalyseurDeSpectre contient les méthodes qui vont être « ré-écrites » (surchargées) dans les classes filles.
- Elle contiendra aussi d'autres méthodes (non modifiées par les classes filles) qui permettront des opérations de bas niveau
 - Ainsi, la méthode CmdConnecter() est une fonction que seule la classe de base à besoin de définir (car identique pour toutes les marques d'AS)
 - Alors que les méthodes Initialiser(), Configurer(), Mesurer() et d'autres seront surchargées par les classes filles
- Nous pourrons donc avoir une méthode de la classe de base qui fait une partie du « travail » commun à tous les équipements et chaque classe fille ferait sa partie spécifique ex:

POO – Découverte par l'exemple

- L'initialisation, dans le cas des équipements de mesure, repose avant tout sur un reset de tous les paramètres (settings), et ensuite, du réglage des différents calibres
 - En SCPI, la commande de reset est équivalente sur tous les équipements: *RST
 - Par contre, chaque équipement a ses propres commandes pour initialiser la fréquence de début et de fin (par exemple)



Analyseur de spectre
Initialiser()
CmdEcrire(" *RST ")

HP 8566
Initialiser()
AnalyseurDeSpectre.Initialiser() CmdEcrire(" CF 6.5 GHZ ")

Agilent E4448A
Initialiser()
AnalyseurDeSpectre.Initialiser() CmdEcrire(" FREQUENCY:CENTER 6.5 GHZ ")

R&S FSEK 8655
Initialiser()
AnalyseurDeSpectre.Initialiser() CmdEcrire(" FREQUENCY:CENTER 6.5 GHZ ")

POO – Découverte par la pratique

- Pour finir, chaque classe fille définirait sa propre version des fonctions de mesure et de configuration
- Il ne reste plus qu'à créer un type (on dit instancier en POO) d'analyseur de spectre vers une variable MonAS et utiliser celle-ci de la manière suivante:

```
MonAS = HP6566() ou AgilentE4448A() ou FSEK8655()  
MonAS.Initialiser()  
MonAS.Configurer()  
MonAS.Mesurer()
```

Encore mieux :

- Les commandes SCPI de base contiennent une commande « IDN? » qui permet de connaître l'identité de l'équipement à l'adresse spécifié.
- On pourrait très bien imaginer une fonction qui va nous « fabriquer » le bon objet « analyseur de spectre » en fonction du retour de la commande « *IDN? »

POO – Découverte par la pratique

Si vous souhaitez pousser la technique à son sommet, vous pourriez vous appuyer sur le motif de conception (design pattern) des fabriques (factory).

https://fr.wikipedia.org/wiki/Fabrique_%28patron_de_conception%29

Dans le cas de notre analyseur de spectre, nous pourrions très bien fournir à la fabrique :

- un nom d'interface
- une adresse

En effet: les différents équipements n'ayant pas tous le même type d'interface, (GPIB pour le HP, Ethernet pour l'Agilent et le R&S), nous pourrions très bien avoir une fonction AsFactory() (fabrique d'analyseur de spectre) qui prendrait 2 paramètres: l'interface et l'adresse sur cet interface.

- MonAS = AsFactory(« GPIB », « 0,4 ») ou
- MonAS = AsFactory(« ETH », « 129.222.12.9 »)

Mais nous nous contenterons d'un simple « constructeur », ce qui sera déjà très bien.

POO – Découverte par la pratique

En fonction du type d'interface, votre factory ou constructeur saura comment traiter l'adresse qui lui est fournie:

- GPIB : le second paramètre doit contenir l'adresse de l'interface et l'adresse de l'équipement sur cette interface (board address et equipment address dans le langage GPIB)
- ETH : le second paramètre contiendra l'adresse IP de l'équipement

La fabrique ou le constructeur :

- Se connectera vers l'équipement en utilisant les informations d'adresse.
- Enverra vers cet équipement la chaîne de demande d'info (*IDN?)
- Retournera un objet de type HP6566() ou AgilentE4448A() ou FSEK8655()

Et comme Python c'est magique, interroger une interface VISA avec ce langage c'est quelques lignes ...



POO – Découverte par la pratique

```
def AsFactory(interface,address):
    """
        pyvisa s'installe avec la commande shell/msdos
        pip install -U pyvisa
    """
    if interface == "GPIB":
        Board, Addr = address.split(" , ")
        AdresseComplete = " GPIB " + Board + " :: " + Addr + " ::INSTR "
    elif interface == "ETH":
        AdresseComplete = " TCPIP0:: " + Addr

    rm = visa.RessourceManager()
    Inst = rm.open(AdresseComplete)
    Equipment = inst.query("*IDN? ")
```

- A partir de là, `Equipment` contient la chaîne d'identification de l'équipement à l'adresse demandée sur l'interface spécifiée.
- Il ne reste plus qu'à retourner le type d'objet correspondant à la chaîne « `Equipment` »

POO – Découverte par la pratique

Mais cela, c'est la cerise sur le gâteau.



Auparavant, voyons comment fonctionne la POO en Python



POO – Historique

La programmation orientée objet a vu ses fondations puisées dans ces mêmes difficultés que vous venez de soulever.

Mais avant la programmation procédurale c'était encore bien plus laborieux...



POO – Historique – Fin des années 1950 - GOTO

- Avant l'apparition de la programmation par fonctions, dite « procédurale », à ne pas confondre avec la programmation fonctionnelle, les fonctions n'existaient pas...
https://fr.wikipedia.org/wiki/Paradigme_%28programmation%29
- Les langages conçus sur ce mode utilisaient généralement une instruction nommée GOTO pour éviter l'exécution de certaines instructions
https://fr.wikipedia.org/wiki/Goto_%28informatique%29
- Le Fortran (1954), puis le COBOL(1959) et« le BASIC » en sont les précurseurs (en dehors de l'assembleur)

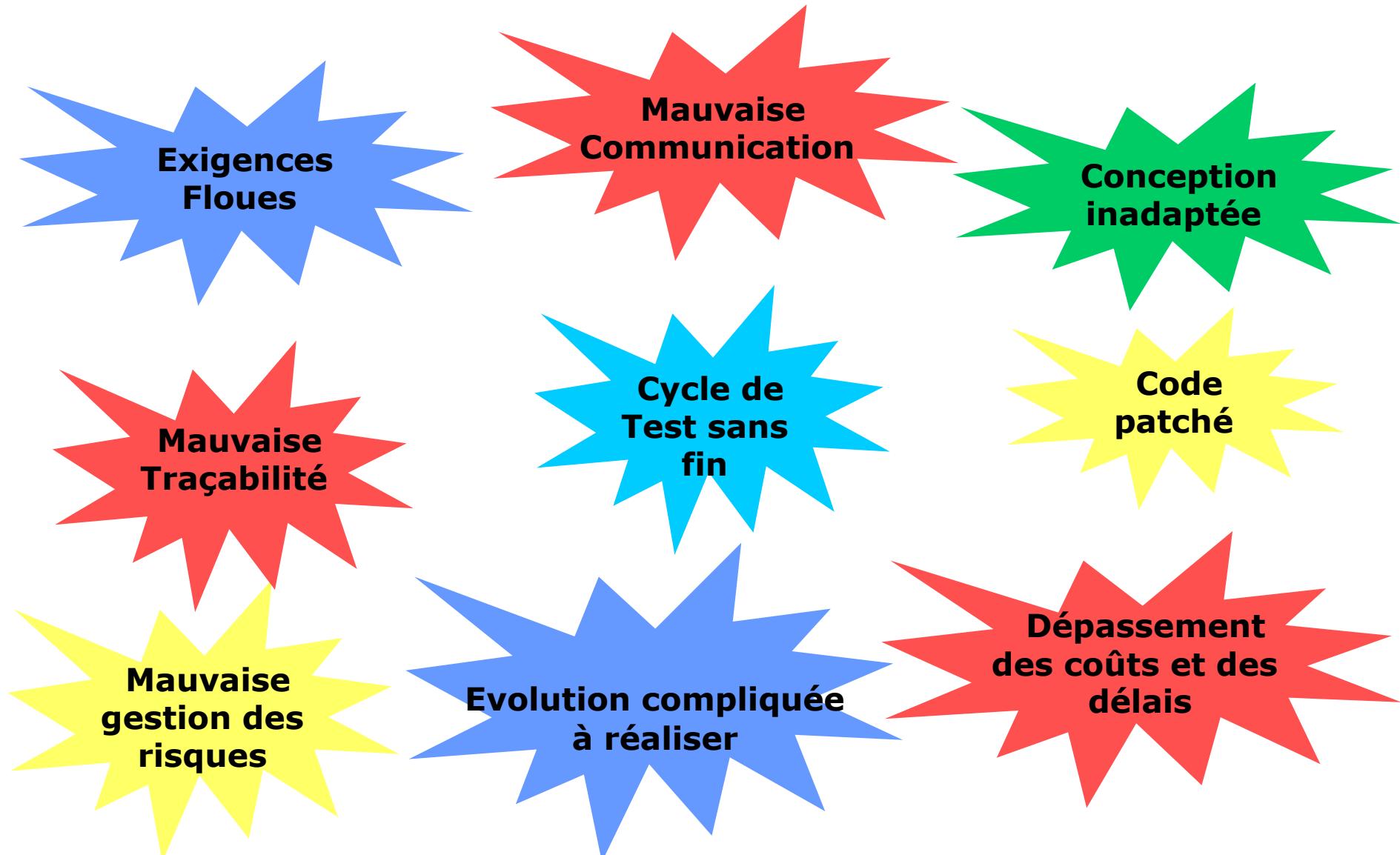
```
10 PRINT 'ENTREZ UN CHIFFRE '
20 LET $A = INPUT
30 IF $A % 2 <> 0 THEN GOTO 60
40 PRINT $A + ' EST PAIR'
50 EXIT
60 PRINT $A + 'EST IMPAIR'
```

POO – Historique – 1960/70 – Les fonctions

- Le tout premier langage à avoir introduit la notion de fonctions est l'ALGOL, en 1958.
https://fr.wikipedia.org/wiki/Programmation_proc%C3%A9durale
- Mais c'est avec l'arrivée de langages comme Fortran (1958), Pascal(1970) et C(1972) que la programmation procédurale a vraiment pris son envol et permis de quasiment éradiquer le GOTO de nos programmes
- Ces langages ont aussi apporté le typage de données, offrant ainsi une plus grande rigueur dans vos programmes.
- Les fonctions ont apporté de nouvelles techniques de programmation très avancées et prometteuses, comme la «récursivité»
- Plus de facilités pour certaines choses...

Mais les programmes avaient toujours tout un tas de problèmes récurrents...

Années 80/2000 - Les années noires de l'informatique

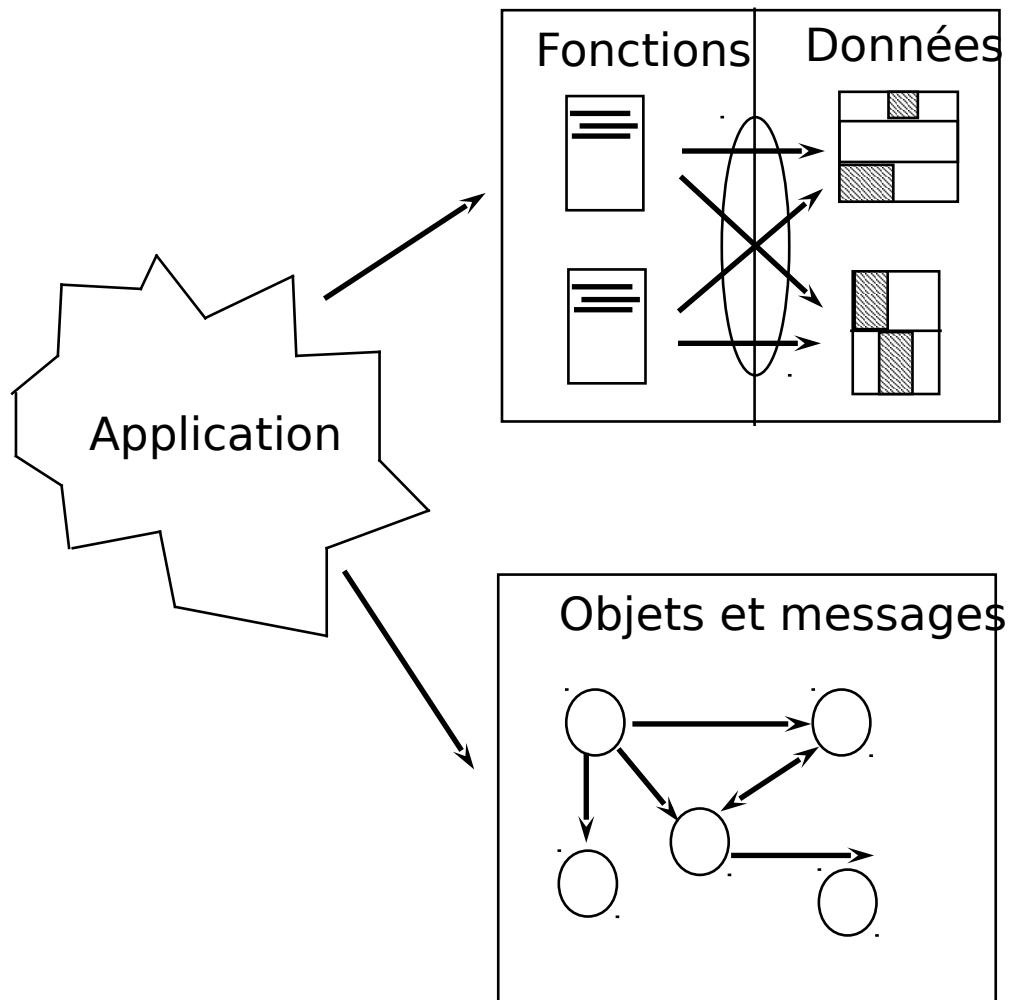


Les approches traditionnelles ne suffisent plus

- Les méthodes traditionnelles avec le cycle en V, telles Merise, SADT, montrent leurs limites
- Tout un tas de méthodologies de mesure de la qualité du code voient alors le jour :
Cocomo, Méthode B, Point de fonction, Métriques d'Halstead, ...
https://fr.wikipedia.org/wiki/M%C3%A9trique_%28logiciel%29
- La qualité logicielle et le génie logiciel affirment alors leur autorité
https://fr.wikipedia.org/wiki/Qualit%C3%A9_logicielle
https://fr.wikipedia.org/wiki/G%C3%A9nie_logiciel
- Un nouveau modèle de conception émerge :
Le modèle objet
Avec lui de nouvelles méthodes de conception



Les approches traditionnelles ne suffisent plus



- **robustesse**
- **évolutivité**
- **description de bas niveau**

- + **robustesse**
- + **évolutivité**
- + **modèle de haut niveau,
proche de la réalité**



POO - Historique

- La Programmation Orientée Objet voit le jour dans le début des années 1960 avec les travaux des Norvégiens Ole-Johan Dahl et Kristen Nygaard
- Ils sont complétés par les travaux d'Alan Kay dans les années 1970
- Elle « consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. », Wikipédia.
- Les principes de la POO ont concrètement vu le jour avec le langage Simula-67

https://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_objet

POO - Historique

- Mais c'est le langage SmallTalk qui réussit véritablement à mettre en exergue ce nouveau concept auprès des informaticiens
- Enfin, c'est avec l'arrivée de langages comme C++ (1983), Python(1990) et Java(1995) que l'industrie s'empare du concept pour en faire son cheval de bataille.
- Avec la programmation objet, de nouvelles méthodes de conception voient le jour :
 - Merise Objet
 - BOOCH,
 - OOSE
 - Unified Process (USDP)
 - Mais c'est UML qui s'imposera quasi unanimement dans l'industrie
https://fr.wikipedia.org/wiki/UML_%28informatique%29

POO – Historique – Méthodes agiles

- Enfin, tout comme les méthodes de conception, de nouvelles pratiques de programmation ont vu le jour ces dernières années sous le terme de méthodes agiles :
 - XP, eXtrem Programming
 - Kanban
 - Scrum
- Elles visent à simplifier le cycle de vie d'un logiciel en favorisant les interactions entre les intervenants et supprimant de nombreuses lourdeurs procédurales afin de faciliter le changement dans un programme



POO – Finalement

- Finalement la POO, accompagnée de bonnes méthodes de conception et d'une gestion de projet agile aura réussi à gagner son pari :
 - Des programmes plus simples
 - Plus maintenables
 - Plus évolutifs
 - Plus fiables
 - Moins coûteux
 - Plus vite développés
- L'incroyable réconciliation des utilisateurs et décideurs avec les informaticiens a lieu...



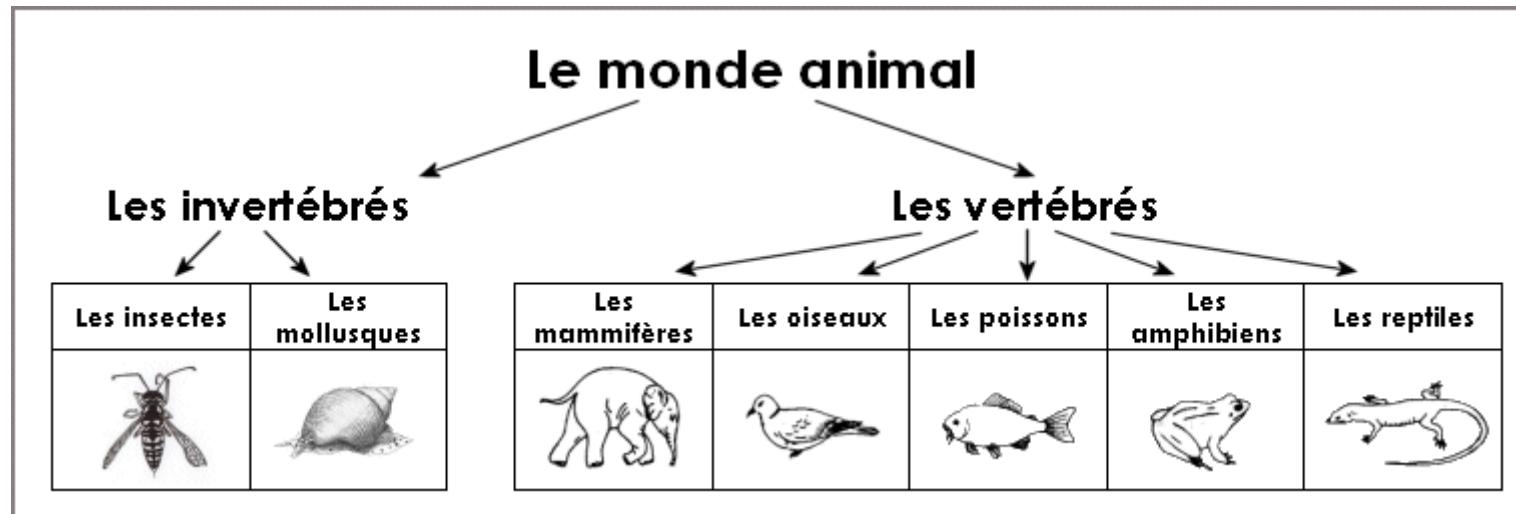
POO – Quelques termes avant de commencer

Quelques termes de qualité logicielle que vous pourrez rencontrer :

- **L'exactitude** : C'est l'aptitude d'un logiciel à fournir le résultat attendu, autrement dit, à répondre correctement au cahier des charges.
- **La robustesse** : C'est l'aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation
- **L'extensibilité** : C'est la facilité avec laquelle un programme pourra être adapté pour répondre des demandes d'évolution
- **La réutilisabilité** : C'est la possibilité d'utiliser certaines parties du logiciel dans un autre programme pour répondre à d'autres besoins.
- **La portabilité** : C'est l'aptitude d'un logiciel à fonctionner dans un environnement matériel ou logiciel différent de son environnement initial
- **L'efficience/performance** : C'est le rapport entre la quantité de ressources utilisées et la quantité de résultats délivrés
Temps de réponse, ressources utilisées (mémoire, disque, ...)

POO - Principes

- L'idée principale de la programmation Orientée Objet est d'essayer de classifier les types de données de votre programme en les regroupant selon leurs points communs
- Et créant des sous-ensembles dès que des objets partagent des données ou comportements communs
- Exactement comme dans le cas des classifications animales et végétales



POO - Principes

- Dans notre exemple nous avons un type de données « AnalyseurSpectre » qui se décompose en 3 sous-types : « HP » et « Agilent » et « H&S »
- Tous les AS, quelque soit leur modèle, possèdent des attributs communs :
 - Identifiant (modèle, numéro série)
 - Interface, adresse.
- Les AS possèdent aussi des comportements communs
 - Initialiser,
 - Configurer,
 - Mesurer
 - CmdEcrire

POO - Principes

- Chaque sous-type d'AS possède aussi des attributs qui lui sont propres :
 - Interface USB, wifi, Ethernet
 - Port série, option du constructeur
- Chaque sous type d'AS possède aussi des comportements qui lui sont propres
 - Soit des comportements communs mais qui sont réalisés de manière particulière pour ce sous type (un HP ne s'initialise pas comme un Agilent)
 - Soit des comportements propres
 - Réparer
 - Auto-diagnostic
 - ...

POO - Principes

La programmation objets a pour objectif de permettre de représenter toutes ces choses....

POO - Principes

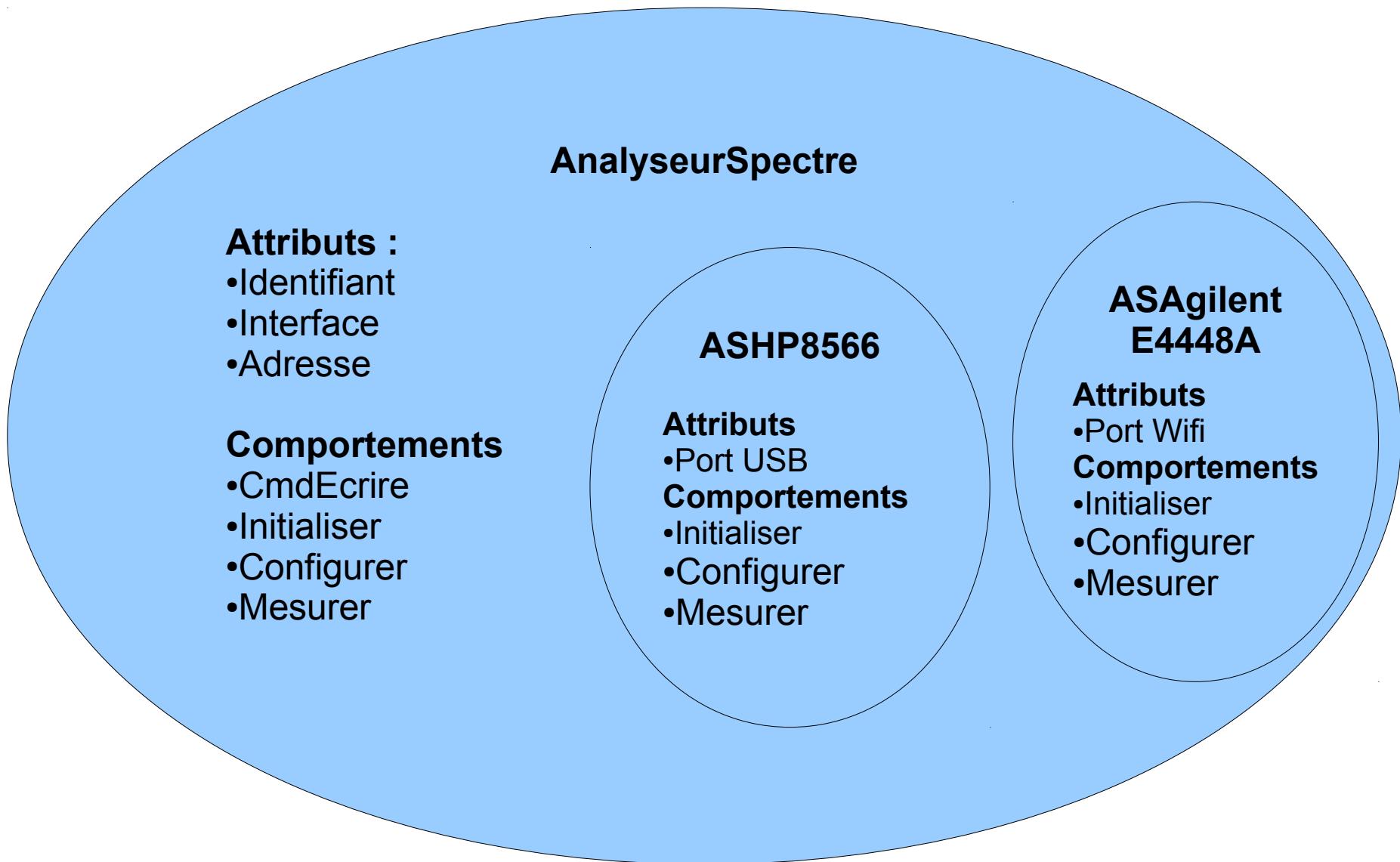


POO - Principes

- En définissant au niveau de chaque type/ensemble tout ce qui est commun à tous les sous-types
- Et en ne redéfinissant au niveau de chaque sous-type que ce qui est réalisé particulièrement pour ce sous-type ou unique à celui-ci
- Que ce soit au niveau des attributs (données) ou des comportements (actions)
- Cela s'appelle la « spécialisation »

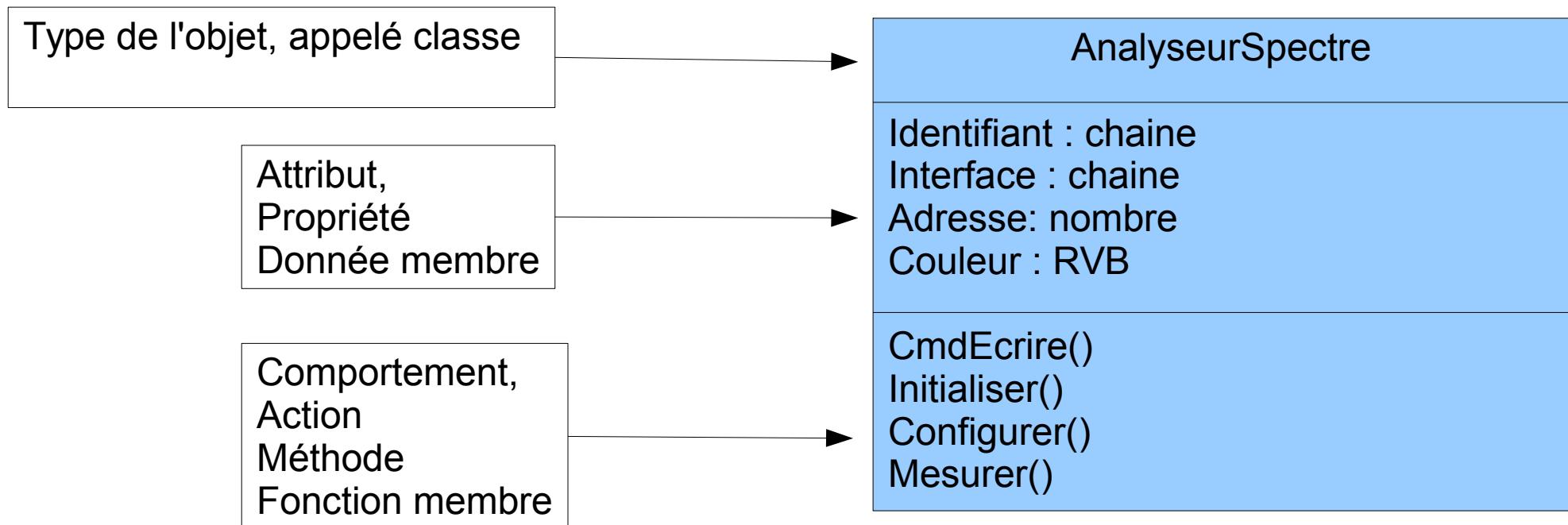


POO - Principes



POO - Terminologie

- Le fait de regrouper dans une même structure de données, le type de la donnée, ses attributs et ses comportements s'appelle **l'encapsulation**
- **En notation UML la symbolique est la suivante**

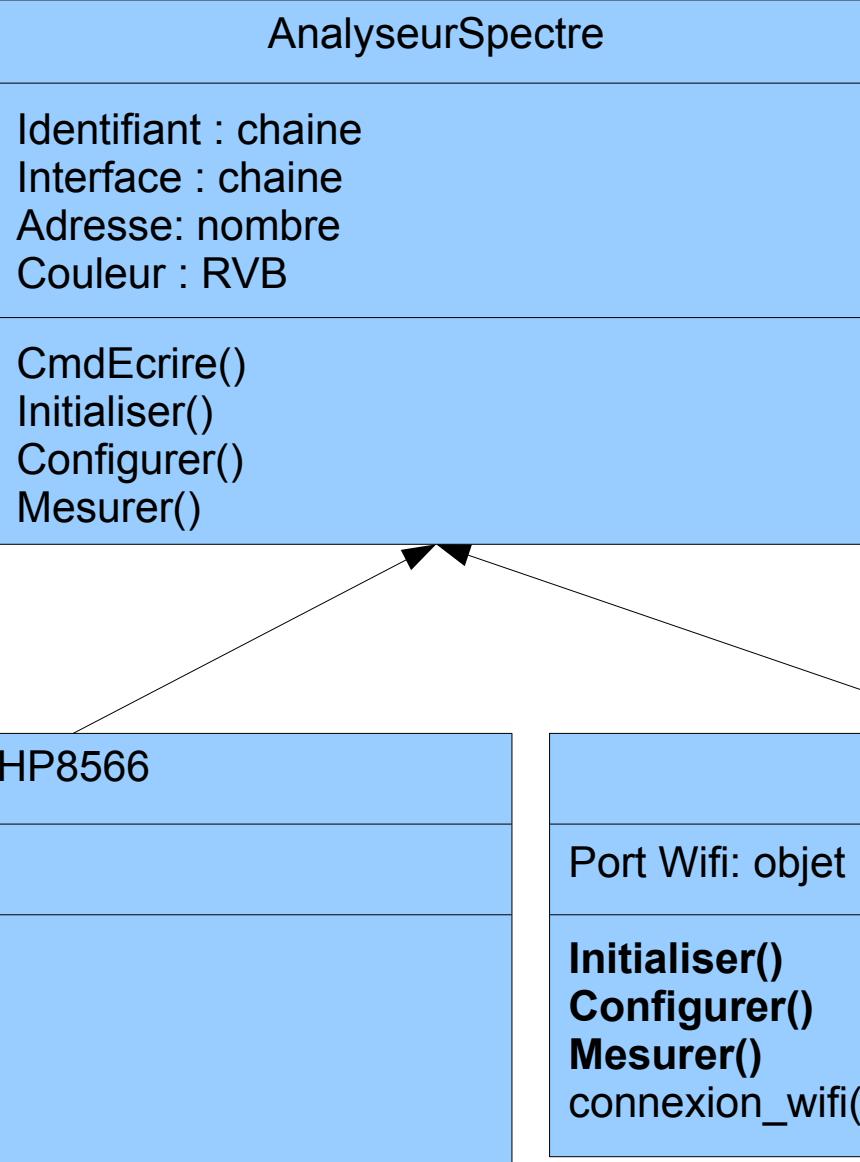


POO - Terminologie

- Définir un sous-ensemble dans un ensemble d'objets s'appelle la **spécialisation**
- Un sous ensemble est aussi appelé un sous-type, ou une **sous-classe**
- Une sous classe bénéficie automatiquement de tout ce qui est défini dans sa **classe parent**. On appelle cela **l'héritage**.
- Une sous classe a uniquement besoin de définir ce qui lui est propre : nouveaux attributs et nouvelles méthodes ainsi que les actions personnalisées.
- Redéfinir une action commune à tous les objets dans une sous classe, autrement dit, personnaliser une action s'appelle le **polymorphisme** ou la **surcharge**.

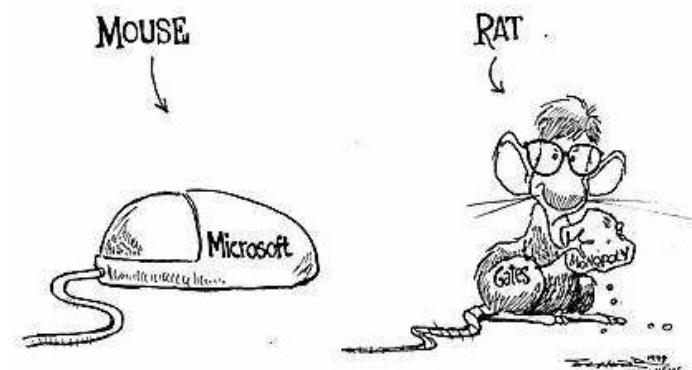
POO - Terminologie

Polymorphisme
Surcharge



POO - Terminologie

- Quand une classe B hérite d'une autre classe A on dit que :
 - La classe A est la **classe parent** de la classe B
 - La classe B est une **classe fille** de la classe A
 - La classe B **hérite** de A
 - La classe B est une **classe dérivée** de la classe A
 - La classe B est une **sous-classe** de la classe A
 - La classe A est une **super classe**
- Une variable issue d'une classe est appelée « **instance** » de la classe.



Programmation Orientée Objet en Python

Découverte par l'exemple

POO en Python – Creation d'une classe

- Pour deфинir une classe en Python on utilise le mot clef « class » suivi du nom de la classe.
- Tout comme pour les fonctions, tout ce qui est indente apres cette declaration fera partie de la classe

```
class AnalyseurSpectre:  
    pass
```

- Pour deфинir des attributs de classe on initialise des variables dans le bloc de code de la classe

```
class AnalyseurSpectre:  
    identifiant = None  
    couleur = 'blanc'
```

- Pour creer une instance de classe, on cre une variable en lui appelant la classe comme s'il s'agissait d'une fonction

```
v1 = AnalyseurSpectre()
```

POO en Python – Attributs de classe

Attention, les attributs de classe ne sont peut-être pas ce que vous croyez...
Quelle sera la couleur des 3 véhicules v1, v2 et v3 à la fin de ce programme ?

```
class AnalyseurSpectre:  
    identifiant = None  
    couleur = 'blanc'  
  
v1 = AnalyseurSpectre()  
print("Couleur de v1: %s" % v1.couleur)  
v2 = AnalyseurSpectre()  
print("Couleur de v2: %s" % v2.couleur)  
# Je repeins v1 en Rouge  
v1.couleur = 'rouge'  
print("Couleur de v1: %s" % v1.couleur) # rouge  
print("Couleur de v2: %s" % v2.couleur) # blanc, semble logique  
# Maintenant la couleur par défaut des véhicules sera le bleu  
AnalyseurSpectre.couleur = 'bleu'  
v3 = AnalyseurSpectre()  
print("Couleur de v1: %s" % v1.couleur)  
print("Couleur de v2: %s" % v2.couleur)  
print("Couleur de v3: %s" % v3.couleur)
```

Couleur de v1: blanc
Couleur de v2: blanc
Couleur de v1: rouge
Couleur de v2: blanc
Couleur de v1: **rouge**
Couleur de v2: **bleu**
Couleur de v3: **bleu**

POO en Python – Attributs de classe

C'est déroutant n'est-ce pas ?

On se serait attendu à ce que les 3 AS soient bleus ou que v2 reste blanc, mais pas vraiment à ce résultat.

Que s'est-il passé ?

- Quand vous définissez des attributs dans une classe, ce sont des attributs de classe, c'est à dire qu'ils ne sont pas spécifiques pour chaque instance, mais partagés entre toutes les instances
- C'est l'équivalent du mot clef « static » en C++ et Java
- Un peu comme les variables locales dans une fonction, pour qu'un attribut soit considéré comme un attribut d'instance, il convient de l'affecter à l'instance : v1.couleur = 'rouge'



POO en Python – Attributs de classe et d'instance

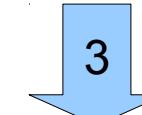
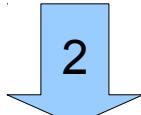
- Quand vous créez une instance de la classe AnalyseurSpectre, telle que définie précédemment, vous créez juste une variable de ce type.
- Quand vous écrivez `print(v1.couleur)` Python procède comme suit :
 - Il regarde si un attribut « couleur » a été affecté localement à `v1`
Si oui il retourne cet attribut d'instance
 - Sinon il regarde si la classe de `v1` possède un tel attribut
 - Si oui il retourne sa valeur
 - Si non il génère une erreur

POO en Python – Attributs de classe et d'instance

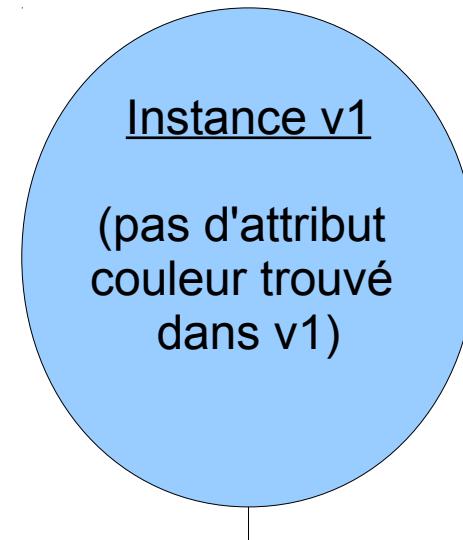
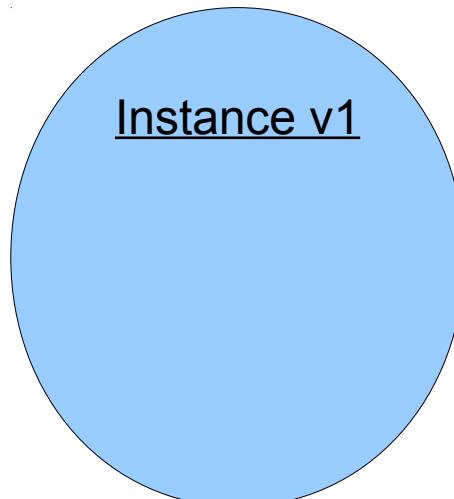
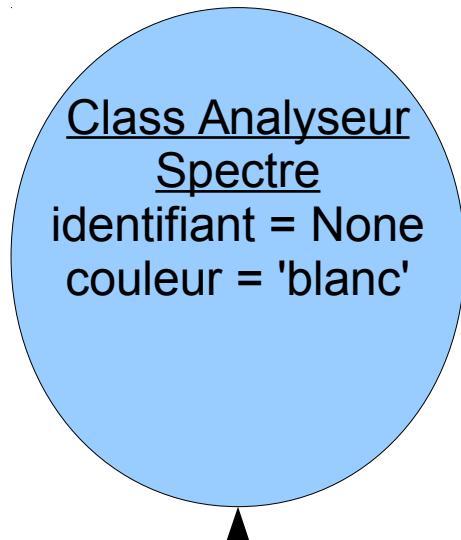
```
class AnalyseurSpectre:  
    identifiant = None  
    couleur = 'blanc'
```

v1 = AnalyseurSpectre()

print(v1.couleur)



Python cherche
dans v1

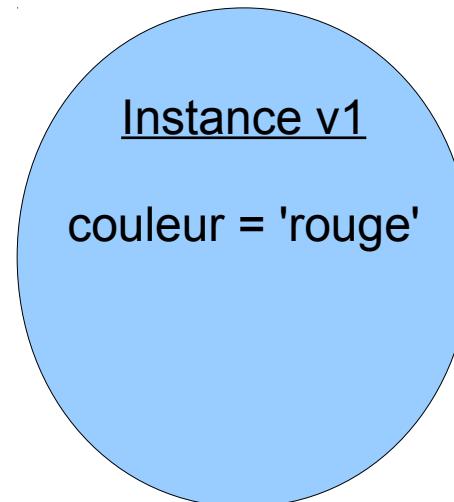
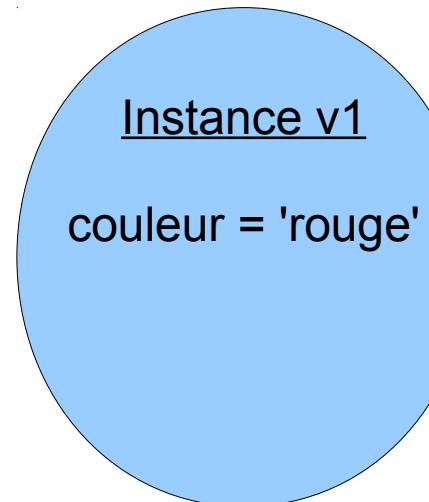
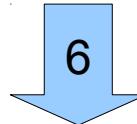
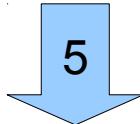


4 - Python cherche alors dans la classe de v1

POO en Python – Attributs de classe et d'instance

v1.couleur = 'rouge'

print(v1.couleur)



Python cherche
dans v1 et trouve l'attribut de
l'instance

POO en Python – Attributs de classe et d'instance

Questions :

- Aurait-on pu affecter un attribut à v1 qui ne soit pas un attribut de la classe AnalyseurSpectre?
- Doit-on affecter manuellement chaque attribut de la classe à chaque instance créée pour qu'il puisse avoir une valeur spécifique à l'instance ?
Quid s'il y en a 50, ce n'est pas très utilisable...
- A quoi servent les attributs de classe ?



POO en Python – Attributs de classe et d'instance

Aurait-on pu affecter un attribut à v1 qui ne soit pas un attribut de la classe AnalyseurSpectre?

Oui, évidemment... Il n'y a que rarement besoin de partager des attributs entre instances.

Mais Python va plus loin, il vous permet de créer des *moutons à 5 pattes* ou des *trèfles à 4 feuilles*...

```
v1.nb_portes = 5  
print(dir(v1))  
print(dir(v2))
```

```
[..., 'couleur', 'identifiant', 'nb_portes']  
[..., 'couleur', 'identifiant']
```

- Dans ce cas, l'instance v1 possède un attribut que ne possèdent pas les autres instances.
- Pour les puristes de l'objet, c'est une hérésie
- Pour les autres c'est plutôt cool.
La réalité n'étant jamais parfaite, c'est effectivement pratique...

POO en Python – Constructeur

Doit-on affecter manuellement chaque attribut de la classe à chaque instance créée pour qu'il puisse avoir une valeur spécifique à l'instance ?

Non, évidemment... Sinon ce serait ingérable

- Pour cela il existe une fonction spéciale, nommée « `__init__` »
- Elle est appelée implicitement par Python chaque fois que vous créez une instance
- On appelle ce type de fonction « un constructeur »
- Elle permet d'initialiser votre instance lors de sa création avec des valeurs par défaut
- Le constructeur peut accepter des paramètres.
- Son premier paramètre, obligatoire, représente l'instance
Par convention il est nommé « `self` ». C'est l'équivalent de « `this` » en Java et C++
- Contrairement à d'autres langages comme Java et C++, Python n'accepte qu'un seul constructeur



POO en Python – Constructeur

```
class AnalyseurSpectre:

    # Attribut directement sous la classe = attribut de classe
    # = attribut statique
    couleur = 'blanc'

    def __init__(self, couleur=None, identifiant=None):
        self.identifiant = identifiant
        if couleur is None:
            # Si paramètre couleur = None, on utilise la couleur de la classe
            self.couleur = AnalyseurSpectre.couleur
        else:
            # On affecte à l'attribut couleur la valeur du paramètre couleur
            self.couleur = couleur

v4 = AnalyseurSpectre()
v5 = AnalyseurSpectre(identifiant='HP 8566', couleur='vert')
```

```
v4: identifiant=None, couleur=blanc
v5: identifiant=1024 AB 02, couleur=vert
```

POO en Python – Attributs de classe et d'instance

A quoi servent les attributs de classe ?

- Ils peuvent servir à définir des constantes utiles/partagées par toutes les instances :
Math.PI=3.1415 ou encore Moto.nombre_roues = 2
- On les utilise souvent pour créer des compteurs permettant de connaître le nombre d'instances créées dans une classe.
- Ils peuvent aussi servir à définir des valeurs par défaut pour certains attributs de la classe
- ...

POO en Python – Attributs de classe et d'instance

Exercices

Premier exercice

- Utilisez les attributs de classe pour compter le nombre d'AS créés dans un attribut nommé « `nb_as_crees` »
- Affichez la valeur de cet attribut après avoir créé quelques instances

Second exercice

- Essayez de créer une classe Singleton
Une classe singleton est une classe ne permettant de créer qu'une seule instance de la classe
- Soit elle ne vous permet pas de créer plus d'une instance
- Soit les nouvelles instances créées sont en fait la première
Pensez à l'affectation par référence

POO en Python – Méthodes

Les méthodes d'une classe sont les actions que peuvent réaliser les objets de la classe. On dit plus généralement les actions qui peuvent être appliquées aux instances de la classe.

- Pour définir une méthode, on crée une fonction dans la classe
- Le premier paramètre de la fonction représente l'instance courante sur laquelle s'applique la fonction
- Par convention ce paramètre est nommé « self »
- Comme il existe des attributs de classe (dits statiques) il existe aussi des méthodes de classe en Python.
Contrairement à C++ et Java qui ne connaissent que les méthodes statiques, Python fait une différence entre méthode statique et méthode de classe.

<https://docs.python.org/3/library/functions.html#classmethod>

<https://docs.python.org/3/library/functions.html#staticmethod>

POO en Python – Méthodes

- Pour invoquer une méthode sur une instance on écrit :
`<instance>.< methode>(<params>)`
- Le paramètre « self » ne doit pas être passé en argument.
Écrire `v1.avancer(distance=10)` est équivalent à écrire
`Vehicule.avancer(v1, distance=10)`
- D'ailleurs les 2 syntaxes sont autorisées

```
class AnalyseurSpectre:  
  
    couleur = 'blanc'  
  
    def afficher(self):  
        print("Identifiant: %s" % self.identifiant)  
  
v4 = Vehicule()  
v4.afficher()
```

POO en Python – Méthodes spéciales

Il existe tout un tas de méthodes spéciales en Python.

Ces méthodes permettent de personnaliser les comportements des objets.

Par exemple d'utiliser les opérateurs mathématiques '+, -, *, /' entre des objets de votre classe :

BigAS = AS1 + AS2

Ou encore de convertir automatiquement votre instance en chaîne de caractères lorsque vous voulez l'afficher:

```
print('Mon AS est :%s' % v1)
```

Ces méthodes sont toutes celles qui sont encadrées par de double underscore « __ » lorsque vous appelez la fonction « dir » sur votre objet

POO en Python – Méthodes spéciales

Parmi ces méthodes, les plus utilisées sont:

- « **`__str__`** » pour convertir votre objet en chaîne
Exemple : `print("mon objet :%s" % mon_objet)`
- « **`__bool__`** » en Python 3 et « **`__nonzero__`** » en Python 2 permet de savoir si votre objet est évalué comme valant True ou False dans une expression booléenne
- « **`__del__`** » : Le destructeur. C'est le pendant du constructeur. Cette méthode est appelée quand votre objet est détruit.
Exemple : `del v1` ou `v1 = None`
- « **`__add__`** », « **`__sub__`** », « **`__mul__`** » : Pour utiliser les opérateurs mathématiques '+', '-' et '*' entre 2 objets
- Vous pouvez toutes les retrouver ici :
<https://docs.python.org/3/reference/datamodel.html>

POO en Python – Méthodes

Exercices :

Exercice 1 :

- Définissez une méthode « afficher » qui affiche tous les attributs d'un véhicule
- Implémentez la méthode « `__str__` » pour qu'elle affiche l'identifiant de votre véhicule. « Utilisez » la dans la méthode afficher

Exercice 2 :

- Ajoutez un attribut de classe « `nb_as_en_service` » qui compte le nombre d'instances actives de la classe
- Ajouter un attribut d'instance qui indique le numéro d'ordre de création de chaque véhicule
- Indication : Utilisez le destructeur

Exercice 3 :

- Implémentez les méthodes « initialiser », « configurer », « mesurer » et « `mesures_en_serie` » (algo du programme initial) de la classe AnalyseurSpectre en vous inspirant de leur version procédurale

POO en Python - Héritage

- Pour indiquer qu'une classe hérite d'une autre classe on liste entre parenthèses après le nom de la classe dans sa déclaration le nom de sa classe parent.
Exemple : class HP8566(AnalyseurSpectre)
- Une classe peut hériter de plusieurs classes.
Elles sont alors séparées par des virgules :
Exemple : classe VoitureAmphibie(Voiture, Bateau)
- Lorsqu'une classe hérite d'une autre classe elle dispose instantanément de toutes ses méthodes et attributs, sans qu'il soit nécessaire de les redéfinir dans la classe fille



POO en Python - Héritage

```
class AnalyseurSpectre:

    couleur = 'blanc'

    def __init__(self, couleur=None, identifiant=None):
        self.identifiant = identifiant
        if couleur is None:
            self.couleur = Vehicule.couleur
        else:
            self.couleur = couleur

    def afficher(self):
        print("Identifiant: %s" % self.identifiant)
        print("Couleur: %s" % self.couleur)

class HP8566(AnalyseurSpectre):
    pass

as1 = HP8566()
as2 = HP8566("rouge", "sn1234")
as2.afficher()
```

Identifiant: sn1234
Couleur: rouge

POO en Python – Héritage, surcharge

Pour modifier le comportement d'un objet dans une classe fille vous devez redéfinir la méthode de cet objet dans la classe dérivée.

- Si vous redéfinissez une méthode dans une classe fille, c'est cette méthode qui sera appelée au lieu de celle de la classe parent
- Vous pouvez toujours faire référence à la méthode de la classe parent en l'appelant directement via le nom de la classe parent
Exemple : AnalyseurSpectre.afficher(v1)
- Ou via le mot clef « super »
`super(HP8566, self).afficher()`
- Le mot clef « super » est préférable à l'appel par le nom de la classe parent comme nous le verrons

POO en Python – Héritage, surcharge

Personnalisons le constructeur des HP8566 pour qu'il accepte un nouveau paramètre : le type de port usb

```
class HP8566(AnalyseurSpectre):  
  
    def __init__(self, port_usb='USB3'):  
        self.port_usb = port_usb  
  
voit1 = HP8566()  
voit2 = HP8566("rouge", "SN 1234")  
voit2.afficher()
```

Traceback (most recent call last):

```
File "/home/makina/FormationPython/poo-initiation.py", line 25, in <module>  
    voit2 = HP8566("rouge", "SN 1234")  
TypeError: __init__() takes from 1 to 2 positional arguments but 3 were given
```

POO en Python – Héritage, surcharge

Que s'est-il passé ?

- Nous avons redéfini le constructeur dans la classe «HP8566».
- Python appelle maintenant ce dernier quand il crée une nouvelle Voiture car la fonction « `__init__` » existe maintenant dans la classe Voiture
- Auparavant il cherchait la fonction « `__init__` » dans la classe « HP8566 », et ne la trouvant pas il utilisait celle de la classe parent «AnalyseurSpectre»
- La nouvelle définition du constructeur « `__init__` » n'appelle pas implicitement sa version dans la classe parent et ne définit qu'un seul paramètre «`port_usb` »
- Python génère donc une erreur sur la création de la seconde voiture car un paramètre de trop est passé au constructeur qui n'accepte plus qu'un `port_usb` dans sa version actuelle

POO en Python – Héritage, surcharge

Pour corriger cela nous pouvons ajouter 2 nouveaux paramètres au constructeur de la classe « Voiture » : ceux de la classe « Vehicule »

```
class HP8566(AnalyseurSpectre):  
  
    def __init__(self, port_usb='usb3', couleur=None, identifiant=None):  
        self.port_usb = port_usb  
        # copie du code de la classe Vehicule  
        self.identifiant = identifiant  
        if couleur is None:  
            self.couleur = AnalyseurSpectre.couleur  
        else:  
            self.couleur = couleur  
voit1 = HP8566()  
voit2 = HP8566("rouge", "SN 1234")  
voit2.afficher()
```



Mais c'est une très mauvaise idée :

- A quoi sert la programmation objet si nous dupliquons le code de la classe parent ?
- Si la classe parent accepte 50 attributs (poids, longueur, hauteur, ...) faut-il vraiment tous les reprendre ?

POO en Python – Héritage, surcharge

Pour ne pas dupliquer il convient d'appeler explicitement le constructeur de la classe parent en début de notre constructeur pour être certain que tous les attributs de celle-ci sont bien initialisés au cas où nous en aurions besoin dans l'initialisation de ceux de la classe fille.

```
class HP8566(AnalyseurSpectre):  
  
    def __init__(self, port_usb='usb3', couleur=None,  
identifiant=None):  
        super(HP8566, self).__init__(couleur, identifiant)  
        #AnalyseurSpectre.__init__(self, couleur, identifiant)  
        self.port_usb = port_usb
```

- La syntaxe utilisant « super » est préférable à celle utilisant « AnalyseurSpectre »
 - « super » indique d'appeler la classe parent de « HP8566 », soit « AnalyseurSpectre » actuellement
 - Si vous décidez un jour que la classe « HP8566 » ne doit plus hériter de « AnalyseurSpectre » mais de « AnalyseurSpectreUSB » héritant lui-même de « AnalyseurSpectre » la syntaxe restera valide ; tandis que la syntaxe utilisant le nommage explicite de la classe parent vous obligera à modifier votre code.

POO en Python – Héritage, surcharge

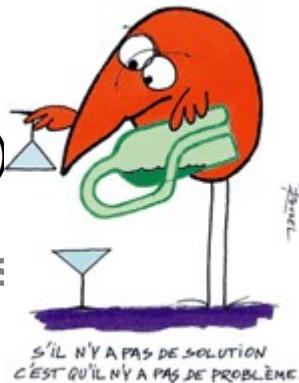
En Python 2 la syntaxe « super » n'est pas toujours disponible :

- En Python 3, toutes les classes n'ayant pas de classe parent héritent implicitement de la classe « object ».
C'est l'équivalent de « Object » en Java.
Ecrire « class AnalyseurSpectre: » est équivalent à « class AnalyseurSpectre(object) : »
- Ainsi, en Python 3, toutes les classes sont des sous-classes de la classe Python « object », mère de toutes les créations.
- En Python 2 ce n'est pas le cas, vous pouvez avoir plusieurs classes dites « racine ».
- Dans ce cas, en Python 2, si votre classe n'hérite pas de la classe « object » à un moment ou un autre, vous ne pourrez pas utiliser le mot clef « super »
- Autrement dit, en Python 2, « super » ne fonctionne qu'avec des classes héritant **explicitement** de la classe « object »

POO en Python – Héritage, surcharge

Et pour ne pas ressaisir tous les paramètres de la fonction surchargée dans la classe fille en plus des nouveaux que vous définissez, vous pouvez utiliser le passage de paramètres par « expansion » .

```
class HP8566(AnalyseurSpectre):  
  
    def __init__(self, port_usb='usb3', *t_args, **d_args)  
        super(HP8566, self).__init__(*t_args, **d_args)  
        #AnalyseurSpectre.__init__(self, *t_args, **d_args)  
        self.port_usb = port_usb
```



Tous les langages objets ne savent pas gérer ceci et certains vous obligeront à saisir manuellement les paramètres requis par la classe parent.

Cela complique la maintenabilité.

Ce n'est pas un problème en Python.

POO en Python – Héritage, surcharge

Exercices :

Exercice 1 :

- Surchargez la fonction « afficher » dans la classe HP8566 pour qu'elle affiche le type du port usb en plus des attributs de la classe parent. Ne dupliquez pas le code.

Exercice 2 :

- Redéfinissez les fonctions « initialiser », « configurer » et « mesurer » dans la classe HP8566 sans appeler les comportements de la classe parent
- Appelez la fonction « mesures_en_serie » que vous n'aurez pas surchargée sur une instance de la classe « HP8566 ». *Toutes les fonctions sont virtuelles en Python !*

POO en Python – Héritage multiple

L'héritage multiple est un cas particulier de l'héritage où une classe fille hérite de plusieurs classes parents.

Ceci pose des problèmes de choix : Si plusieurs classes parentes définissent une même méthode « afficher » mais pas leur classe fille, quelle(s) méthode(s) de quelle classe(s) parent(s) et dans quel ordre seront appelées en écrivant « fille1.afficher() »

- Celle de la première classe parent
- Celle de la dernière
- Toutes ?
- Dans quel ordre ?

L'algorithme gérant ces cas de figures s'appelle MRO en Python, pour **Method Resolution Order**

ATTENTION : Il ne fonctionne pas de la même manière en Python 2 et Python 3 !

Ce tutorial vous donnera tous les éléments pour bien comprendre cette problématique :

<http://makina-corpus.com/blog/metier/2014/python-tutorial-understanding-python-mro-class-search-path>

POO en Python – Visibilité des attributs et méthodes

Certains langages objets proposent des modificateurs de visibilité des attributs et méthodes entre classes parent/fille ou en dehors de la classe. Généralement il s'agit de :

- **Private**: les attributs et méthodes de ce type ne sont manipulables que dans la classe qui les définit
- **Protected** : les attributs et méthodes de ce type ne sont manipulables que dans la classe qui les définit et ses classes filles
- **Public** : les attributs et méthodes de ce type sont manipulables partout, notamment en dehors du bloc de la classe



POO en Python – Visibilité des attributs et méthodes

En Python ces modificateurs de visibilité n'existent pas. Tous les attributs et méthodes sont toujours publiques.
C'est une faiblesse de l'implémentation objet du langage.

Toutefois il existe des conventions :

- Préfix « `__` » : Si vous préfixez le nom de votre méthode ou attribut par 2 tirets bas, vous indiquez que vous souhaitez qu'il soit privé
- Préfix « `_` » : Si vous préfixez le nom de votre méthode ou attribut par 1 tiret bas, vous indiquez que vous souhaitez qu'il soit protégé

Mais, c'est comme en Perl, ce ne sont que des conventions d'écriture.
Vous les respecterez non pas parce que le langage vous interdit d'utiliser ces attributs en dehors de leurs contextes, mais parce que vous êtes un développeur bien élevé et que l'on vous a dit de ne pas le faire.

POO en Python – Accesseurs/Manipulateurs

En programmation objet on utilise généralement des fonctions pour accéder aux attributs d'une instance (lecture/écriture) plutôt que de les manipuler directement.

On appelle ces fonctions des « accesseurs/manipulateurs » ou « getters/setters ».

L'idée est de déclarer tous les attributs de type « privé » et de n'utiliser que ces getters/setters pour y accéder :

- Cela permet d'effectuer des contrôles avant de modifier ou retourner une valeur(droit d'accès, dépassement de bornes, etc.)
- Cela vous permet de faire évoluer plus facilement vos données : Si vous décidez en interne de représenter une couleur non plus par son nom, mais par une composante RVB, vous pourrez changer son type sans impacter le code des développeurs utilisant votre API



POO en Python – Accesseurs/Manipulateurs

```
class AnalyseurSpectre:  
    couleur = 'blanc'  
  
    def __init__(self, couleur=None, identifiant=None):  
        self.identifiant = identifiant  
        if couleur is None:  
            self.__couleur = Vehicule.couleur  
        else:  
            self.__couleur = couleur  
  
    def set_couleur(self, new_couleur):  
        self.__couleur = new_couleur  
  
    def get_couleur(self):  
        return self.__couleur  
  
v1 = AnalyseurSpectre("rouge", "Ma Ferrari")  
print('Identifiant de v1:' + v1.get_couleur())  
# Je repeins v1  
v1.set_couleur("Rouge flammes")
```



POO en Python – Properties

Pour simplifier l'accès aux attributs privés par des getters/setters, Python propose une mécanique nommée les « properties ».

Elle permet d'utiliser vos attributs par de faux noms qui appellent de manière transparent les getters et setters que vous avez définis. Mais vous continuez à utiliser une syntaxe plus naturelle.



POO en Python – Properties

```
class AnalyseurSpectre:

    def __init__(self, couleur=None, identifiant=None):
        self.identifiant = identifiant
        if couleur is None:
            self.__couleur = Vehicule.couleur
        else:
            self.__couleur = couleur

    def get_couleur(self):
        return self.__couleur

    def set_couleur(self, value):
        self.__couleur = value

    def del_couleur(self):
        del self.__couleur

    couleur = property(get_couleur, set_couleur, del_couleur, "couleur's docstring")

v1 = AnalyseurSpectre("rouge", "Ma Ferrari")
print('Identifiant de v1:' + v1.couleur) # appelle v1.get_couleur()
v1.couleur = "Rouge flammes" # appelle v1.set_couleur("Rouge flammes")
del v1.couleur # appelle v1.del_couleur()
```

POO en Python – Pour aller plus loin

- Le model objet de Python est décrit de manière exhaustive dans la documentation, chapitre « DataModel »
<https://docs.python.org/3/reference/datamodel.html>
<https://docs.python.org/2/reference/datamodel.html>
- Plusieurs évolutions posant parfois des soucis de compatibilité ont été introduites entre la version 2 & 3, notamment au niveau des destructeurs, des méthodes spéciales et du MRO.
Soyez vigilants lors des migrations.
- Python offre une mécanique pour simuler les classes abstraites, ce sont des classes pour lesquelles on ne peut pas créer d'instance. On peut juste des dériver.
<https://docs.python.org/3.3/library/abc.html>
- Le décorateur @property offre une autre manière de définir des getter/setter