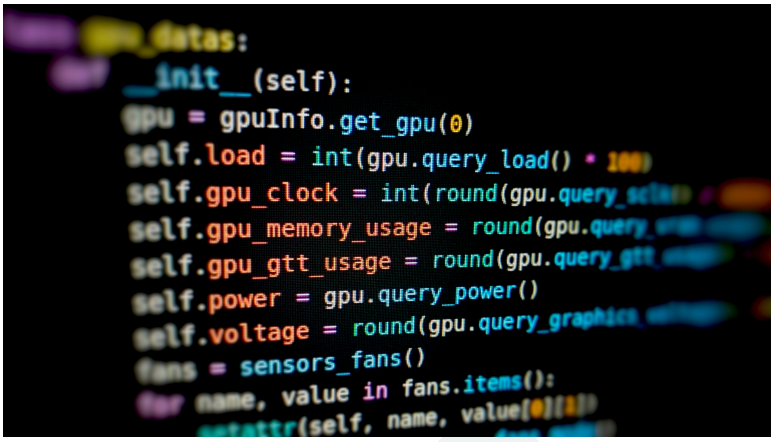
Patrick Nsukami

1/12/24

Contents

Programmation Orientée Objet:



Paradigmes de la programmation:

On appelle paradigme, la façon de **formuler** une solution dans un langage de programmation. Il existe plusieurs types de paradigmes, nous avons, *parmi tant d'autres*:

- la programmation impérative
- la programmation fonctionnelle
- la programmation orientée objet

Python nous permet de programmer dans n'importe lequel de ces paradigmes. Mais pour ce cours, nous nous concentrerons exclusivement sur le paradigme orienté objet.

POO:

La programmation orientée objet est un style de programmation centré sur le concept "d'objets". Chaque objet possède des caractéristiques et des opérations.

Mettre en place un système en utilisant le paradigme objet équivaut à peu près à ceci:

- Identifier les objets dans le monde réel.
- Définir ou bien modéliser ces objets
- Définir ou bien modéliser le comportement de ces objets
- Décrire comment ces objets interagissent entre eux

Pilliers de la programmation orientée objet:

La programmation orientée objet est caractérisée par 4 pilliers. Ces quatre pilliers sont:

- L'héritage
- L'encapsulation
- Le polymorphisme
- L'abstraction

Objet:

Les objets correspondent plus ou moins à des entités du monde réel. Par exemple, une boutique en ligne peut être modélisée avec des objets tels que: “panier”, “produit”, “client”, “facture”, etc ...

Un **objet** est une entité qui possède:

- un identifiant unique
- un ensemble **d’opérations** permettant soit de manipuler l’objet, soit d’offrir des services
  - une opération aura un nom unique
  - une opération renverra un type (integer, float, string, list, ...)
  - une opération pourra être accessible ou non
- un ensemble **d’attributs** permettant de distinguer un objet d’un autre
  - un attribut aura un nom unique
  - un attribut aura un type (integer, float, string, list, ...)
  - un attribut pourra être accessible ou non

Classe:

À l’intérieur de notre système, chaque objet, ou bien **chaque instance**, manipulé vient nécessairement d’une **classe**. Une classe est un **modèle** définissant quels seront les caractéristiques et les opérations qu’un objet doit avoir. Une classe est un **moule** permettant de fabriquer des objets.

Une **classe** est une entité logicielle qui possède:

- un identifiant unique
- un ensemble **d’opérations**
  - une opération aura un nom unique
  - une opération renverra un type (integer, float, string, list, ...)
  - une opération pourra être accessible ou non
- un ensemble **d’attributs**
  - un attribut aura un nom unique
  - un attribut aura un type (integer, float, string, list, ...)
  - un attribut pourra être accessible ou non

Voyons maintenant comment tout cela s’implémente en Python.

Les classes en Python:

En Python, on dit que tout est objet. Dans l’exemple suivant, a est un objet de type int. Cet objet possède au moins 2 attributs (caractéristiques): `imag` et `real`; cet objet possède au moins deux méthodes (opérations): `conjugate` et `as_integer_ratio`. Démonstration:

```
>>> a = 3
>>> type(a)  # notez le mot 'class'
<class 'int'>
>>>
>>> a.imag  # cet objet possède un attribut: imag (partie imaginaire)
0
>>> a.real  # cet objet possède un autre attribut: real (partie réelle)
3
>>> a.conjugate()  # cet objet possède une méthode
3
>>> a.as_integer_ratio() # cet objet possède une autre méthode
(3, 1)
```

Chaque fois que nous avons manipulé une variable, en fait nous manipulions un objet venant d'une *classe*, un objet possédant des *attributs*, et possédant des *méthodes*.

Créer nos propres types:

Pour pouvoir créer nos propres classes, nous utiliserons un nouveau mot clé: `class`.

Création d'une classe:

Voici comment créer une classe nommée Foobar et ne faisant absolument rien du tout. La convention nous recommande de toujours mettre en majuscule, la première lettre de la classe:

```
>>> class Foobar:
...     pass
...
>>>
>>> o1 = Foobar()  # instanciation d'un objet de la classe Foobar
>>> type(o1)
<class '__main__.Foobar'>
>>>
```

1. Opérations:

Mettre en place des opérations équivaut à définir des fonctions à l'intérieur de la classe. Une classe peut rien d'autre que des méthodes. Ces méthodes peuvent prendre autant de paramètres que nous voulons. Sauf dans certains cas bien définis, le premier paramètre d'une méthode sera toujours `self`. Attention, une méthode ne peut être exécutée en dehors du contexte de la classe. Exemple:

```
>>> class Barbaz:
...     """La docstring de la classe"""
...
...     def f(self):
...         """Une méthode nommée f qui ne prend aucun paramètre"""
...         return 'hello world'
...
...     def g(self, name="qux"):
...         """Une méthode nommée g qui prend un paramètre nommé name"""
...         return f'hello {name}'
...
>>> o6 = Barbaz()  # instanciation
>>> o6.f()  # appel d'une méthode
'hello world'
```

2. Caractéristiques:

Mettre en place des caractéristiques est équivalent à définir la méthode `__init__`. Cette méthode *spéciale*, nommée dunder `init` est la méthode en charge d'initialiser les attribut de notre objet. Exemple:

```
>>> class Quux:
...     def __init__(self, name, age, address):
...         """mise en place des attributs"""
...         self.name = name
...         self.age = age
...         self.address = address
...
>>> o2 = Quux("foo", 42, "Dakar")  # instanciation
>>> print(o2.name)  # lecture d'un attribut
'foo'
```

Types d'attribut:

Les attributs d'un objet ne sont pas tous les mêmes.

1. Attribut d'instance:

On appelle variables d'instance ou attributs d'instance, les données qui sont propres à l'objet courant. Les variables d'instance n'ont pas la même valeur d'un objet à un autre. Example:

```
...  class Cal:
...      def __init__(self, radius, pi):
...          self.radius = radius # radius est un attribut d'instance
...          self.pi = pi # pi est un attribut d'instance
...      def area(self):
...          return self.pi * (self.radius ** 2)
...
>>> c1, c2 = Cal(56, 3.142), Cal(45, 3.14)
>>> c1.pi
3.142
>>> c1.area()
9853.312
>>> c2.pi
3.14
>>> c2.area()
6358.5
```

2. Attribut de classe:

On appelle attribut de classe, les données qui peuvent être partagées par tous les objets de la classe. Les variables de classe ne changent pas de valeur d'un objet à un autre. Example:

```
...  class Cal:
...      # pi est un attribut de classe
...      # d'une instance à l'autre, pas besoin de changer cette valeur
...      pi = 3.14
...      def __init__(self, radius):
...          # radius est attribut d'instance
...          # le rayon est différent d'une instance à une autre
...          self.radius = radius
...          self.pi = pi
...
...      def area(self):
...          return self.pi * (self.radius ** 2)
...
>>> c1 = Cal(56)  # pas besoin de précier une valeur de pi
>>> c2 = Cal(45)  # pas besoin de préciser une valeur de pi
>>> c1.pi
3.14
>>> c1.area()
9847.04
>>> c2.pi
3.14
>>> c2.area()
6358.5
```

Types de méthodes:

De la même manière qu'il existe des attributs propres à une instance et des attributs propres à la classe, il existe également différents types de méthodes.

1. Méthode d'instance:

Une méthode d'instance est une méthode qui ne peut être executée qu'à travers une instance de la classe. Chaque fois que vous instanciez un objet et que vous appelez une méthode en faisant `objet.methode()`, vous êtes actuellement entrain d'executer une méthode d'instance.

2. Méthode de classe:

Les méthodes de classes sont des méthodes qui peuvent être invoquées sans passer par une instance (pas besoin nécessairement de créer un objet pour pouvoir appeler la méthode). La méthode peut être invoquée à travers la classe. Les méthodes de classe possèdent **une *référence* vers la classe courante: cls**; de la même manière que les méthodes d'instance possèdent **une *référence* vers l'objet courant: self**. Les méthodes de classe permettent la mise en place de constructeur alternatifs, exemple:

```
class Person:
    def __init__(self, first_name, last_name):
        # ici on va construire un nouvel objet en utilisant 2 données
        self.first_name = first_name
        self.last_name = last_name
```

```
    @classmethod # remarquez le décorateur
    def from_full_name(cls, name): # remarquez le premier paramètre: cls,
        # ici on va construire un nouvel objet en utilisant une seule donnée
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        # cls est une référence vers la classe courante
        # la ligne suivante est équivalente à Person(first_name, last_name)
        return cls(first_name, last_name)
```

```
>>> p1 = Person("foo", "bar")
>>> p2 = Person.from_full_name("corge qux") # utilisation de la méthode de classe
>>> p1.first_name
'foo'
>>> p1.last_name
'bar'
>>> p2.first_name
'corge'
>>> p2.last_name
'qux'
```

3. Méthode statique:

Une méthode statique est une méthode qui ne possède aucune référence ni à l'objet courant, ni à la classe courante. Elle peut être exécutée à travers un objet ou bien à travers la classe. Les méthodes statiques sont de **simples** fonctions, rassemblées dans une classe par besoin **d'organisation de code:**

```
class Person:
    def __init__(self, age):
        self.age = age

    @staticmethod # remarquez le décorateur
    def get_the_elder(a, b): # remarquez l'absence et self et l'absence de cls
        if isintance(a, Person) and isinstance(b, Person):
            if a.age < b.age:
                return b
            else:
                return c
        else:
            raise TypeError("Can't compare apples and oranges")
```

Pilliers:

Maintenant que nous avons exploré les différents styles d'attributs et les différents styles de méthodes. Parlons des pilliers de la programmation orientée objet.

Héritage:

L'héritage est le mécanisme grâce auquel une classe (**fille**) peut récupérer les attributs et les méthodes d'une autre (**mère**). Exemple, lisons et regardons attentivement le programme suivant:

```
>>> class Lemur:
...     def __init__(self, head, arms, fingers, locomotion, vision):
...         self.head = head
...         self.arms = arms
...         self.fingers = fingers
...         self.locomotion = locomotion
...         self.vision = vision
...     def move(self):
...         print("I'm moving myself")
...     def eat(self):
...         print("I'm eating fruits")
...
...
>>> class Monkey:
...     def __init__(self, head, arms, fingers, locomotion, vision):
...         self.head = head
...         self.arms = arms
...         self.fingers = fingers
...         self.locomotion = locomotion
...         self.vision = vision
...     def move(self):
...         print("I'm moving myself")
...     def eat(self):
...         print("I'm eating fruits")
...
```

Ensuite, faisons une comparaison avec le progamme qui suit:

```
>>> class Primate:
...     def __init__(self, head, arms, fingers, locomotion, vision):
...         self.head = head
...         self.arms = arms
...         self.fingers = fingers
...         self.locomotion = locomotion
...         self.vision = vision
...     def move(self):
...         print("I'm moving myself")
...     def eat(self):
...         print("I'm eating fruits")
...
>>> class Lemur(Primate):
...     pass
...
>>> class Monkey(Primate):
...     pass
...
```

Grâce à l'héritage, nous avons pu **factoriser** du code. Nous avons implémenté les fonctionnalités génériques dans la classe mère. Puis nous avons réutilisé ces fonctionnalités dans les classes filles sans duplication de code.

Ajout d'attributs dans la classe fille (fonction super):

Il est possible pour une classe fille, d'avoir des caractéristiques qui lui sont propres:

```
class Person:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

class Teenager(Person):
    def __init__(self, name, phone, website):
        # utilisation de la fonction super pour
        # invoquer le constructeur de la classe mère
        super().__init__(name, phone)
        # website est un attribut propre à la classe fille
        self.website=website
```

Définition ou redéfinition de méthode:

Il est possible pour une classe fille des méthodes qui lui sont propres:

```
class Fish:
    def __init__(self, name, age, skeleton, eyelids):
        self.name = name
        self.age = age
        self.skeleton = skeleton
        self.eyelids = eyelids
    def swim(self):
        print("I'm swimming.")
```

```
class Trout(Fish):
     # cette méthode n'existe pas dans la classe parente
     def swim_backwards(self):
         print("I can swim backwards.")
```

Il est possible pour une classe fille de redéfinir une méthode dont elle a hérité:

```
class Fish:
    def __init__(self, name, age, skeleton, eyelids):
        self.name = name
        self.age = age
        self.skeleton = skeleton
        self.eyelids = eyelids
    def swim(self):
        print("I'm swimming.")

class Trout(Fish):
     # cette méthode existe dans la classe parente, avec une implémentation différente
     def swim(self):
         print("I am Trout, I can swim too.")
```

Il est possible pour une classe fille d'étendre une méthode définie dans la classe mère:

```
class Fish:
    def __init__(self, name, age, skeleton, eyelids):
        self.name = name
        self.age = age
        self.skeleton = skeleton
        self.eyelids = eyelids
    def swim(self):
        print("I'm swimming.")

class Trout(Fish):
     # cette méthode réutilise l'implémentation de la classe mère
     def swim(self):
         print("I am Trout")
         super().swim()  # <--- ici
         print("I am Trout")
```

Encapsulation:

En programmation orientée objet:

- les attributs (les données) d'un objet n'ont pas besoin d'être toutes accessibles.
- les détails d'implémentation d'une méthode n'ont pas nécessairement besoin d'être connus
- les détails relatifs à la lecture d'un attribut n'ont pas nécessairement besoin d'être connus
- Certaines *informations et interactions* sont **privées**, d'autres **publiques**.

L'encapsulation est le mécanisme qui va nous permettre de restreindre l'accès à des attributs ou à des méthodes.

```
class Person:
    def __init__(self):
        # par défaut, un attribut est public
        self.public_name = "foobar"
```

```
        # remarquez le double underscore avant le nom pour le rendre privé
        self.__private_name = "barbaz"

    # par défaut, une méthode est publique
    def public_method(self):
        return 42
    # remarquez le double underscore pour rendre la méthode privée
    def __private_method(self):
        return 45
```

On utilise l'encapsulation pour en quelques sortes "cacher" les détails d'implémentation d'un service. Par exemple: pour démarrer sa voiture, un conducteur n'a besoin que de cliquer sur un bouton. Les détails (relatifs à tout ce qui doit s'opérer pour que la voiture démarre) sont cachés.

1. Attributs publics, attributs privés:

Les attributs publics sont directement accessibles. Les attributs privés ne le sont pas:

```
>>> p1 = Person()  # instanciation
>>> p1.public_name
'foobar'
>>> p1.public_method()
42
>>> p1.__private_name  # tentative d'accès
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [10], in <cell line: 4>()
      2 p1.public_name
      3 p1.public_method()
----> 4 p1.__private_name

AttributeError: 'Person' object has no attribute '__private_name'
```

2. Getters, Setters:

Un getter est une méthode qui renvoie la valeur d'un attribut. Un setter est une méthode qui modifie la valeur d'un attribut. Getters et setters seront utilisés pour permettre la lecture et la modification des attributs privés:

```
class Person:
    def __init__(self):
        # par défaut, un attribut est public
        self.public_name = "foobar"
        # remarquez le double underscore avant le nom pour le rendre privé
        self.__private_name = "barbaz"

    # get_private_name est une méthode publique, son rôle est de
    # permettre la lecture de l'attribut privé __private_name
    def get_private_name(self):
        return self.__private_name

    # set_private_name est une méthode publique, son rôle est de
    # permettre la modification de l'attribut privé __private_name
```

```
    def set_private_name(self, new_value):
        self.__private_name = new_value
```

Getters et setters seront également utilisés pour ajouter des traitements autour de la lecture et autour de l'écriture des attributs privés.

```
class Person:
    def __init__(self):
        # par défaut, un attribut est public
        self.public_name = "foobar"
        # remarquez le double underscore avant le nom pour le rendre privé
        self.__private_name = "barbaz"

    def get_private_name(self):
        # utilisation du getter pour effectuer une transformation
        result = self.__private_name.upper()
        # avant de renvoyer la valeur de l'attribut
        return result

    def set_private_name(self, new_value):
        # utilisation du setter pour effectuer une validation
        # avant de modifier la valeur de l'attribut
        if isinstance(new_value, str):
            self.__private_name = new_value
        else:
            raise TypeError("the value should be a string")
```

3. Propriétés:

Le problème avec les getters et les setters, c'est qu'ils ne sont pas intuitifs. Toute personne voulant manipuler l'objet devra retrouver les noms exacts des getters/setters permettant la manipulation de tel ou tel attribut. Pour implémenter des attributs offrant getters et setter, et pour éviter à l'utilisateur de se concentrer seulement sur le nom de l'attribut manipulé, nous utiliserons un autre mécanisme: les propriéts.

Les propriétés sont des attributs un peu particuliers. Ils permettent la manipulation de nos attributs de manière beaucoup plus élégante. L'utilisateur manipule un attribut, sans s'inquiéter des getters/setters qui agissent en arrière plan. Exemple:

```
class Person:
    def __init__(self):
        self.__private_name = "barbaz"

    @property # remarquez la présence du décorateur pour mettre en place le getter
    def name(self):
        result = self.__private_name.upper()
        return result

    @name.setter # remarqyez le décorateur pour mettre en place le setter
    def name(self, new_value):
        if isinstance(new_value, str):
            self.__private_name = new_value
        else:
            raise TypeError("the value should be a string")
```

```
>>> p2 = Person() # instanciation
>>>
>>> # remarquez cmt l'utilisateur manipule un attribut, derrière, le getter intervient
>>> p2.name
'BARBAZ'
>>> # remarquez cmt l'utilisateur modifie un attribut, derrière, le setter intervient
>>> p2.name = "foobar"
>>> p2.name
'FOOBAR'
```

Polymorphisme:

Le nom de polymorphisme vient du grec et signifie qui peut prendre plusieurs formes. Le polymorphisme est un autre pillier de la programmation orientée objet. Le polymorphisme est le concept consistant à fournir une interface unique à des entités pouvant avoir des types différents.

Par exemple, la fonction len adapte son comportement, suivant la nature de l'objet passé en argument. S'il s'agit d'une chaine de caractère, elle renverra le nombre de caractères. S'il s'agit d'une liste, elle renverra le nombre d'éléments dans cette liste. Etc ...

1. Polymorphisme utilisant des classes n'ayant aucune relation:

Ce type de polymorphisme est utile lorsque l'on doit manipuler une séquence d'objets qui sont tous regroupés dans un ensemble, mais venant de classes différentes. Pas besoin de tester la nature de l'objet, on invoque la méthode et on laisse le système déterminer la bonne implémentation. Exemple:

```
class Cat:
    def __init__(self, name):
        self.name = name
    def make_sound(self):
        print("Meow")

class Dog:
    def __init__(self, name):
        self.name = name
    def make_sound(self):
        print("Bark")

d1 = Dog("Bobby")
c1 = Cat("Kitty")
d2 = Dog("Roxy")

# animals est une variable contenant des objets
animals = [d2, d1, c1]
for animal in animals:
    # on ne s'inquiete pas du type de animal
    # on laisse le système invoquer la bonne implémentation
    animal.make_sound()
```

2. Polymorphisme utilisant des classes ayant une relation d'héritage:

Même principe. Ici, les classes ont une relation d'héritage et la classe fille peut même redéfinir la méthode de la classe mère. Exemple:

```
class Animal:
    def make_sound(self):
        print("Some sound")

class StrangeDog(Animal):...

class Cat(Animal):
    def make_sound(self):
        print("Meow")

class AnimalBot(Animal):
    def make_sound(self):
        super().make_sound()
        print("@_%^~#{ç")


o1 = Animal()
o2 = StrangeDog()
o3 = Cat()
o4 = AnimalBot()

# animals est une variable contenant des objets
animals = [o1, o3, o2, o4]
for animal in animals:
    # on ne s'inquiete pas du type de animal
    # on laisse le système invoquer la bonne implémentation
    animal.make_sound()
```

3. Polymorphisme utilisant la surcharge de méthode:

Python ne propose pas la surcharge de méthodes. Si une méthode est définie plus d'une fois dans une classe, seule la dernière définition sera retenue par le système, exemple:

```
In [34]: class A:
    ...:     def f(self):
    ...:         return 42
    ...:     def f(self, a, b):
    ...:         return 45
    ...:     def f(self, a, b, c):
    ...:         return 46
    ...:
```

```
In [35]: A().f()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [35], line 1
----> 1 A().f()

TypeError: A.f() missing 3 required positional arguments: 'a', 'b', and 'c'
```

Pour mettre en place ce genre de polymorphisme, il faudra installer et utiliser des bibliothèques telles que multimethod.

4. Polymorphisme utilisant le décorateur singledispathmethod:

Il est *en quelques sortes*, possible de mettre en place *une certaine forme* de polymorphisme, en utilisant une *certaine forme* de surcharge de méthode. Ce mécanisme utilise le décorateur singledispatchmethod. Exemple:

```
class A:
    @singledispatchmethod
    def f(self, a):
        # si paramètre inconnu, renvoyer erreur
        raise NotImplementedError("Unknown type")
    @f.register
    def _(self, a:int):
        # si paramètre entier, renvoyer 45
        return 45
    @f.register
    def _(self, a:str):
        # si paramètre str, renvoyer 46
        return 46


>>> o1 = A()
>>>
>>> o1.f(4)  # invocation de f avec un entier
45
>>>
>>> o1.f("foo")  # invocation de f avec un str
46
>>>
>>> o1.f({})
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In [48], line 1
----> 1 o1.f({})

File /usr/local/lib/python3.11/functools.py:946, in singledispatchmethod.__get__.<locals>._method(*args, **kwargs)
    944 def _method(*args, **kwargs):
    945     method = self.dispatcher.dispatch(args[0].__class__)
--> 946     return method.__get__(obj, cls)(*args, **kwargs)

Cell In [44], line 4, in A.f(self, a)
      2 @singledispatchmethod
```

```
      3 def f(self, a):
----> 4     raise NotImplementedError("Unknown type")

NotImplementedError: Unknown type
```

NOTA BENE: Remarquez comment le système choisit la bonne méthode à invoquer en s'appuyant seulement et uniquement sur le type du premier paramètre.

Abstraction:

L'abstraction est le moyen pour nous d'utiliser une classe afin d'exprimer **une intention** et non **une implémentation**.

1. Classe abstraite:

Une classe est dite abstraite si elle ne premet pas l'instanciation d'objets.

Supposons nous voulons écrire un programme manipulant des **Primates.** Le programme devra manipuler différents types de primates: **Humain, Lemurien, Singe, ...**

Ces primates ont des caractéristiques en commum. Il peut donc être intéressant pour nous de mettre en place une classe mère contenant toutes les caractéristiques communes. Et ensuite, mettre en place des classes filles.

La question devient: **Est ce cohérent pour notre système d'avoir des instances de la classe Primate?** La réponse est non. Dans ce cas là, nous ferons de la classe Primate, une classe **abstraite**.

2. Méthode abstraite

Une méthode est abstraite lorsqu'elle est déclarée mais ne fournit aucune implémentation. Pour qu'une classe soit déclarée abstraite, il faut absolument qu'elle définisse au moins une méthode abstraite. Pour déclarer une méthode abstraite, nous utiliserons le décorateur abstractmethod.

Une classe devient véritablement abstraite lorsque:

- Elle hérite de la classe ABC
- Elle définit au moins une méthode abstraite

Exemple:

```
from abc import abstractmethod, ABC  # remaquez les imports


class Animal(ABC): # remarquez l'héritage
    @abstractmethod  # remarquez le décorateur
    def make_sound(self):
        pass

class Cat(Animal):
    def make_sound(self):
        print("Meow")

class AnimalBot(Animal):...
```

```
In [53]: Animal()  # instancier la classe abstraite est interdit
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [53], line 1
```

```
----> 1 Animal()

TypeError: Can't instantiate abstract class Animal with abstract method make_sound
```

```
In [55]: AnimalBot() # instancier une classe qui n'implémente pas la méthode abstraite est interdit
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [55], line 1
----> 1 AnimalBot()

TypeError: Can't instantiate abstract class AnimalBot with abstract method make_sound
```

3. Pour résumer:

Pour résumer:

- Une classe abstraite ne peut être instanciée.
- Une classe abstraite doit hériter de la classe ABC.
- Une classe abstraite doit contenir **au moins** une méthode abstraite.
- Une classe héritant de la classe abstraite doit implémenter la ou les méthodes abstraites.

Pour en savoir plus:

Nous espérons que vous avez appris deux ou trois choses en lisant ce cours. Le sujet étant très vaste, il n'est pas possible pour nous de tout voir dans un seul document. Ci dessous, une liste de recommandations que nous suggérons à toute personne désireuse d'approfondir ses connaissances sur les concepts abordés jusqu'ici.

À lire:

- Les classes
- Les dataclasses
- Les classes abstraites
- Colab: chapter 6, OOP
- Les méthodes spéciales
- Les descripteurs

À regarder:

- Introduction aux classes
- Classes, instances, méthodes
- Variables et attributs
- Héritage
- Héritage multiple et mro
- Méthodes magiques
- Méthodes statiques

"Ce qui est dans la parole est dans le silence." proverbe Africain