Introduction à la Programmation Orientée Objet en Python

Rédigé par : Fabien Poirier
Supervisé par : Geoffrey Groff

February 4, 2024

Plan du Chapitre

Les méthodes

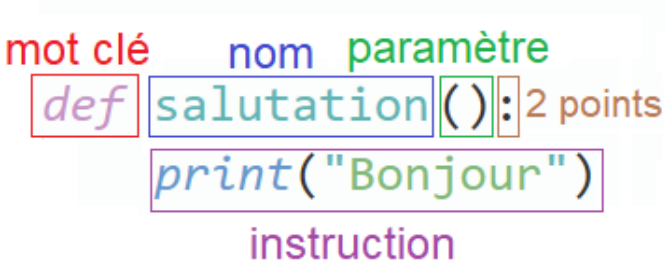


Figure: Procédure

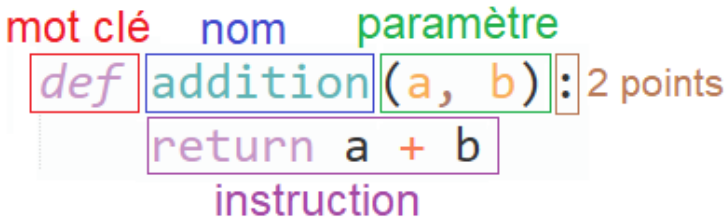


Figure: Fonction

- Lorsque vous utilisez le mot-clé **def**, suivi du nom de la fonction et de ses paramètres entre parenthèses, vous déclarez une méthode.
- Pour rappel, la différence entre une fonction et une procédure se situe au niveau du retour de valeur : les procédures n'en ont pas, contrairement aux fonctions.

Les paramètres optionnels

paramètre optionel

```
def saluer(nom="tout le monde"):
    print("Bonjour,", nom)

# Appel de la fonction sans spécifier de paramètre
saluer()  # Output: Bonjour, tout le monde

# Appel de la fonction en spécifiant un paramètre
saluer("Alice")  # Output: Bonjour, Alice
```

prototype

- Pour définir un paramètre optionnel, il suffit de lui attribuer une valeur par défaut lors de la déclaration du prototype.

Les paramètres optionnels (2)

```python
def saluer(nom, salutation="Bonjour", ponctuation="!"):
    print(salutation + ", " + nom + ponctuation)


# Appel de la fonction en spécifiant uniquement le nom
saluer("Alice")  # Output: Bonjour, Alice!

# Appel de la fonction en spécifiant le nom
# et une ponctuation personnalisée
saluer("Bob", ponctuation=".")  # Output: Bonjour, Bob.

# Appel de la fonction en spécifiant le nom,
# une salutation personnalisée et une ponctuation personnalisée
saluer("Charlie", "Coucou", ".")  # Output: Coucou, Charlie.
```

- Si vous avez plusieurs paramètres optionnels mais que vous souhaitez uniquement changer la valeur de l'un d'entre eux, vous devrez le spécifier via la syntaxe **nom_param=val** lors de l'appel de la méthode.

Passage par copie vs passage par référence

Passage par copie ?

Passage par référence ?

Mais que veulent dire ces termes ?

Passage par copie / Passage par référence

Passage par copie [X]

Une copie de la valeur de l'objet est passée à la méthode.
Cela signifie que toute modification de l'objet à l'intérieur de la méthode n'affectera pas l'objet original en dehors de la méthode.

Passage par référence [✓]

Une référence à l'objet est passée à la méthode.
Cela signifie que toute modification de l'objet à l'intérieur de la méthode affectera également l'objet original en dehors de la méthode.

Le multi return !

```python
def diviser_et_rester(a, b):
    quotient = a // b
    reste = a % b
    return quotient, reste

# Appel de la fonction
q, r = diviser_et_rester(10, 3)
print("Quotient:", q)  # Affiche: Quotient: 3
print("Reste:", r)      # Affiche: Reste: 1
```

- En Python, une fonction peut renvoyer plusieurs valeurs à la fois sous forme de tuple. Vous pouvez stocker ces valeurs dans des variables distinctes en les séparant par des virgules, ce qui est appelé "unpacking".
- Lors de l'affectation des valeurs retournées, vous pouvez également ignorer certaines valeurs en les remplaçant par un underscore (_), indiquant que vous n'avez pas l'intention d'utiliser ces valeurs.

return / yield ?

Les générateurs

Un générateur est une fonction spéciale qui permet de produire une séquence de valeurs de manière paresseuse, c'est-à-dire qu'il génère les valeurs au fur et à mesure de leur demande plutôt que de les calculer et de les stocker en mémoire en une seule fois. Cela permet d'économiser de la mémoire, en particulier lorsque vous travaillez avec de grandes séquences de données.

```
def generateur(start=0, end=100, step=1):
    for x in range(start, end, step):
        yield(x)


gen = generateur()
for _ in range(10):
    print(next(gen))
```

Avantages des générateurs ! (1/2)

- Syntaxe : ils sont définis comme des fonctions, mais utilisent l'instruction yield au lieu de return. L'utilisation de yield permet de suspendre temporairement l'exécution et de la reprendre là où elle s'était arrêtée lors du prochain appel.

- Paresseux : ils ne calculent et ne renvoient les valeurs qu'au moment de leur appel, ce qui évite de stocker toute la séquence en mémoire.

- Itérable : ce sont des objets itérables une seule fois de par leur nature.

Avantages des générateurs ! (2/2)

- Efficacité : ils sont plus adaptés que les listes ou les tuples lorsqu'il s'agit de manipuler de grandes séquences de données, car ils ne chargent en mémoire qu'une seule valeur à la fois. De plus, ils sont responsables de leur mnémonique en cas d'implémentation.

- Génération infinie : ils peuvent être utilisés pour générer des séquences infinies de valeurs.

Le docstring : normes

- Le docstring commence et se termine par trois guillemets.
- La première ligne est une description de la fonction.
- Chaque paramètre de la fonction est décrit sur une nouvelle ligne, précédé de :param suivi du nom du paramètre, puis d'une description du paramètre.
- Si la fonction renvoie quelque chose, vous pouvez également spécifier cela avec :return: suivi d'une description de ce qui est retourné.
- Les autres sections facultatives peuvent inclure :raises: pour les exceptions levées par la fonction, :rtype: pour le type de retour, etc.

Le docstring : exemple

```python
def saluer(nom, salutation="Bonjour", ponctuation="!"):
    """
    Fonction qui affiche une salutation personnalisée.

    :param nom: Le nom de la personne à saluer.
    :param salutation: La salutation à afficher (par défaut: "Bonjour").
    :param ponctuation: La ponctuation à ajouter à la fin de la salutation (par défaut: "!").
    :return: Aucun retour, affiche simplement la salutation.
    """
    print(salutation + ", " + nom + ponctuation)
```

La methode help()

La méthode **help()** est une fonction utilisée pour obtenir de l'aide sur les modules, les fonctions, les classes, etc. Elle fournit des informations sur leur utilisation, leur syntaxe et leurs arguments en affichant dans la console le commentaire docstring lié à l'élément fourni en paramètre. Exemple : **help(nom_fonction)**

Les annotations de type (type hints)

```python
def saluer(nom: str, salutation: str = "Bonjour", ponctuation: str = "!") -> None:
    """
    Fonction qui affiche une salutation personnalisée.

    :param nom: Le nom de la personne à saluer.
    :param salutation: La salutation à afficher (par défaut: "Bonjour").
    :param ponctuation: La ponctuation à ajouter à la fin de la salutation (par défaut: "!").
    :return: Aucun retour, affiche simplement la salutation.
    """
    print(salutation + ", " + nom + ponctuation)
```

- Lors de la déclaration d'une méthode, il est possible de spécifier le type de chaque paramètre en utilisant la syntaxe nom_paramètre: type. Le type de retour sera précisé à la fin des parenthèses via l'utilisation d'une flèche → type_de_retour
- Cette approche permet une meilleure documentation du code en indiquant explicitement les types de données attendus en entrée et le type de données retourné par la fonction.

La fonction Main

```python
if __name__ == "__main__":

    print("Hello World !")
```

Pourquoi cette structure ?

L'utilisation de `if __name__ == '__main__'` permet de rendre un fichier Python à la fois exécutable en tant que programme principal et importable comme module, en séparant clairement le code destiné à chaque cas d'utilisation.

Début TP1

TP 1 - Générateurs :

- Développez un générateur qui produira des nombres premiers en utilisant comme paramètres un début, une fin et un pas.
- Développez un générateur calculant la suite de Fibonacci en utilisant les mêmes paramètres que le précédent.

Plan du Chapitre

Les importations !

Les importations sont utilisés pour inclure dans votre programme actuel des modules ou des fonctions définies dans d'autres fichiers. La portée de ces fonctions ou éléments est limitée au fichier dans lequel ils sont importés. Cela vous permet d'utiliser du code réutilisable et d'organiser votre code de manière modulaire.

- **Importation de module:** *import nom_du_module*
- **Importation de fonction:** *from nom_du_module import nom_de_la_fonction*
- **Importation avec renommage de module:** *import nom_du_module as nouveau_nom*

Exemples d'importation

- **Importation de module:**

```python
import math

# Utilisation d'une fonction du module math
print(math.sqrt(25))  # Affiche la racine carrée de 25
```

- **Importation de fonction:**

```python
from math import sqrt

# Utilisation de la fonction directement sans préfixe
print(sqrt(25))  # Affiche la racine carrée de 25
```

- **Importation avec renommage de module:**

```python
import math as m

# Utilisation du module renommé
print(m.sqrt(25))  # Affiche la racine carrée de 25
```

Les espaces de noms (namespace)

espace de noms / namespace

Un espace de noms est un système qui garantit que les noms de variables sont uniques et identifiables dans un programme. Cela permet d'éviter les conflits de noms et de fournir un contexte clair.

- **Espace de noms local (local namespace)**
- **Espace de noms global (global namespace)**
- **Espace de noms intégré (built-in namespace)**

Installer un nouveau module ?

PIP

pip est un gestionnaire de paquets qui vous permet d'installer, de désinstaller et de gérer des paquets Python supplémentaires qui ne sont pas inclus dans la distribution standard.

- **Installer un paquet:** *pip install nom_du_package*
- **Installer une version précise:** *pip install nom_du_package==version*
- **Mettre à jour un paquet:** *pip install –upgrade nom_du_package*
- **Desinstaller un paquet:** *pip uninstall nom_du_package*
- **Lister les paquets installés:** *pip list*

Conflit : pip2 & pip3

Attention !

Si vous avez plusieurs versions de Python sur votre machine, notamment Python 2 et Python 3, vous avez également plusieurs versions de pip (pip2 / pip3), chacune associée à la version correspondante de Python.

1. Assurez-vous de préciser à chaque requête la version de pip : pip3
2. Vous pouvez configurer un alias dans votre ~/`.bashrc` en ajoutant la ligne suivante : alias pip='pip3'

Connaître son environnement de travail

pip freeze

pip freeze est une commande utilisée pour générer une liste des paquets installés dans un environnement virtuel ou global, ainsi que leurs versions. Cette liste est généralement utilisée pour sauvegarder l'état des dépendances d'un projet Python afin de pouvoir le reproduire sur un autre système ou de partager facilement les exigences du projet avec d'autres développeurs.

- **Afficher la liste des paquets:** *pip freeze*
- **Enregistrer cette liste dans un fichier:** *pip freeze > requirements.txt*
- **Installer les paquets à partir d'un fichier:** *pip install -r requirements.txt*

Plan du Chapitre

Ouvrir un fichier (Open vs with Open)

La fonction **open()** est utilisée pour ouvrir un fichier. Elle prend en paramètre le chemin du fichier à ouvrir ainsi que le mode d'ouverture. Une fois le fichier traité celui-ci doit être fermé.

le fichier fonction chemin du fichier type d'ouverture

```
fichier = open("exemple.txt", "r")
# Instructions
fichier.close()
```

L'instruction **with open()** est une syntaxe recommandée pour ouvrir et gérer des fichiers. Elle crée un contexte de gestion de fichier qui garantit que le fichier est correctement fermé à la fin, même en cas d'erreur pendant le traitement.

mot clé fonction chemin du fichier type d'ouverture le fichier

```
with open("exemple.txt", "w") as fichier:2 points
    # Instructions
```

Les types d'ouverture (1)

- "r" (read) : Ouvre le fichier en mode lecture. Le fichier doit exister, sinon une erreur sera levée.
- "w" (write) : Ouvre le fichier en mode écriture. Si le fichier existe, son contenu sera supprimé. Si le fichier n'existe pas, il sera créé.
- "a" (append) : Ouvre le fichier en mode ajout. Les nouvelles données seront ajoutées à la fin du fichier, sans supprimer le contenu existant. Si le fichier n'existe pas, il sera créé.

Les types d'ouverture (2)

- "x" (exclusive creation) : Ouvre le fichier en mode création exclusive. Si le fichier existe déjà, une erreur sera levée.
- "b" (binary) : Ouvre le fichier en mode binaire, ce qui est utile pour les fichiers non textuels (comme les images ou les fichiers binaires). Ce mode est souvent utilisé en combinaison avec les modes "r", "w" ou "a".
- "+" (read/write) : Ouvre le fichier en mode lecture et écriture. Ce mode est généralement utilisé en combinaison avec les modes "r", "w" ou "a".

Méthodes liées aux fichiers

Méthodes	**Description**
close()	Ferme le fichier.
open(path)	Ouvre le fichier situé au chemin spécifié.
read()	Lit le fichier entier.
readline()	Lit et retourne la première ligne du fichier sous forme de chaîne de caractères.
readlines()	Lit et retourne une liste contenant toutes les lignes du fichier.
write(string)	Écrit la chaîne de caractères spécifiée dans le fichier.
writelines(lines)	Écrit une liste de lignes dans le fichier.

Erreurs / Exceptions

Exception

Les exceptions sont des événements qui se produisent lorsqu'une condition inattendue ou indésirable se produit pendant l'exécution d'un programme. Lorsqu'une exception est levée, le programme peut soit la gérer pour continuer son exécution de manière contrôlée, soit laisser l'exception se propager, ce qui arrête l'exécution du programme et affiche un message d'erreur.

Les mots clés liés aux exceptions

Le traitement des exceptions

1. Lever une exception : Vous pouvez lever une exception à l'aide du mot-clé **raise**.

```
raise ValueError("Valeur incorrecte")
```

2. Gérer les exceptions : Vous pouvez gérer les exceptions à l'aide de blocs **try**, **except**, **else** et **finally**.

```
try:
    # Code susceptible de lever une exception
except SomeException:
    # Code pour gérer l'exception
else:
    # Code à exécuter si aucune exception n'est levée
finally:
    # Code à exécuter quel que soit le résultat
```

La gestion d'exception : explication

mot clé

```
try:
    # Tentative d'ouverture d'un fichier
    fichier = open("exemple.txt", "r")
    # Lecture du contenu du fichier
    contenu = fichier.read()
    print(contenu)

nom de l'exception
except FileNotFoundError:
    # Gestion de l'exception si le fichier n'est pas trouvé
    print("Le fichier spécifié est introuvable.")

else:
    # Code à exécuter si aucune exception n'est levée
    print("Lecture du fichier terminée avec succès.")

finally:
    # Code à exécuter quel que soit le résultat
    if 'fichier' in locals():
        # Fermeture du fichier s'il est ouvert
        fichier.close()
        print("Fermeture du fichier.")
    else:
        print("Aucun fichier n'a été ouvert.")
```

bloc d'instructions optionel

Les exceptions personnalisées !

- Créer son propre type d'exception :

```
class MonException(Exception):
    def __init__(self, message="Une erreur personnalisée s'est produite"):
        self.message = message
        super().__init__(self.message)
```

- La propager :

```
def fonction_critique(parametre):
    if parametre < 0:
        raise MonException("Le paramètre ne peut pas être négatif")
```

mot clé nom de l'exception Message d'erreur

Plan du Chapitre

Structure d'un code python !

```
"""
Ce fichier/module est utilisé pour effectuer des opérations mathématiques simples.
"""

# Imports de la bibliothèque standard de Python
import os
import sys

# Imports des modules externes (third-party)
import numpy as np
import pandas as pd

# Imports des modules locaux ou spécifiques à l'application
from my_module import my_function

# Variables globales
DEBUG = True
MAX_ITERATIONS = 1000

# Définition des fonctions/classes
def add(a, b):
    return a + b

class MyClass:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")

# Code exécutable
if __name__ == "__main__":
    my_obj = MyClass("Alice")
    my_obj.greet()
```

Importation

Variables globales

Méthodes / Classe

Main

La Programmation Orienté Objet

Les concepts clés :

- Classes et Objets
- Attributs et Méthodes
- Encapsulation
- Héritage
- Polymorphisme

Classe & Encapsulation

La classe

- Une classe définit un modèle pour la création d'objets. Elle encapsule des données (attributs) et des fonctions (méthodes) qui agissent sur ces données.

→ Il est recommandé de n'avoir qu'une seule classe par fichier !

mot clé nom 2 points

```
class Personne:
```

variable de la classe

```
    nombre_instances = 0
```

Méthodes de la classe

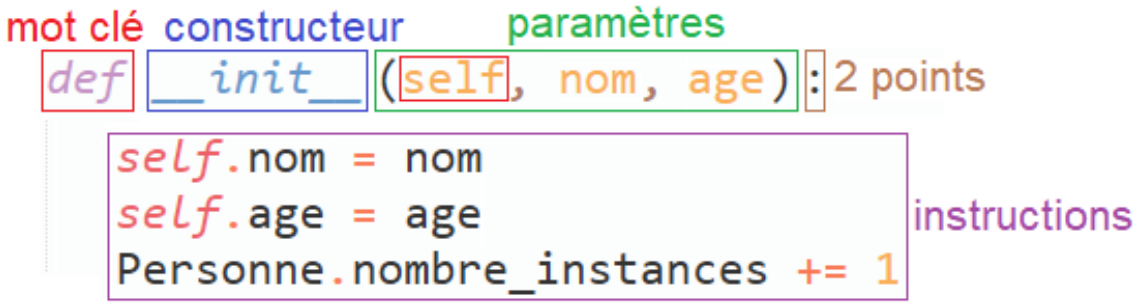
```
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
        Personne.nombre_instances += 1

    def presenter(self):
        print(f"Je m'appelle {self.nom} et j'ai {self.age} ans.")
```

La construction d'instance !

Le constructeur

- La méthode __init__() est le constructeur d'une classe. Elle est appelée automatiquement à chaque fois qu'une nouvelle instance de la classe est créée.



Objet / Instance

Instance de classe

- Les objets sont des instances spécifiques d'une classe et représentent des entités du monde réel ou des concepts abstraits.

```python
# Création de plusieurs objets de la classe Personne
alice = Personne("Alice", 30)
bob = Personne("Bob", 25)

# Récupération du type de l'objet alice
type_alice = type(alice)
print(type_alice) # Personne

# Vérification si l'objet bob est une instance de la classe Personne
is_bob_personne = isinstance(bob, Personne)
print(is_bob_personne) # True
```

self le this de Python !

Le mot clé self

- Le rôle de **self** est de permettre à la méthode d'accéder aux attributs et aux méthodes de l'objet sur lequel elle est appelée. Self est une référence à l'instance elle-même, et sera passée implicitement lors de l'appel à la méthode.

  Cela signifie que lorsque vous définissez une méthode dans une classe, vous devez inclure **self** comme premier paramètre dans la signature de la méthode pour indiquer qu'elle est une méthode de l'instance.

Le mot clé self

```
class Personne:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def presenter(self):
                  [alice]
        print(f"Je m'appelle {self.nom} et j'ai {self.age} ans.")
                             [alice]                  [alice]


# Création d'un objet personne
alice = Personne( "Alice", 30)

alice.presenter()
```

Figure: Explication du fonctionnement du mot clé self

Les méthodes de classe et les méthodes statiques (1)

Méthode de classe

- Ce sont les méthodes qui prennent implicitement l'instance de la classe (self) en premier paramètre.
- Elles ont accès aux attributs de l'instance via self.
- Elles sont utilisées pour manipuler les attributs spécifiques à chaque instance de la classe.

Méthode statique

- Ce sont des méthodes qui n'ont pas besoin de l'instance de la classe comme premier paramètre.
- Elles sont décorées avec @staticmethod.
- Elles sont utilisées lorsque la méthode ne dépend pas de l'état de l'instance, mais qu'elle est logiquement liée à la classe.

Les méthodes de classe et les méthodes statiques (2)

```python
class Personne:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def presenter(self):
        print(f"Je m'appelle {self.nom} et j'ai {self.age} ans.")

    @staticmethod
    def est_adulte(age):
        return age >= 18
```

Constructeur

Méthode d'instance

Méthode statique

Les décorateurs

- Un décorateur est une fonction qui prend une autre fonction en tant qu'argument et retourne une fonction modifiée ou une version améliorée de cette fonction. En d'autres termes, les décorateurs permettent de modifier le comportement d'une fonction sans modifier son code interne.
- La syntaxe de base d'un décorateur consiste à placer un "@" suivi du nom du décorateur au-dessus de la fonction à décorer. Cela permet d'appliquer le décorateur à cette fonction.
- Il est possible d'appliquer plusieurs décorateurs à une même fonction en les empilant les uns au-dessus des autres.

Une histoire de portée ?

Seriez-vous capable de définir les 3 portées utilisées en programmation orientée objet ?

Les portées : public, private et protected

- **La portée publique :** l'élément peut être accessible depuis n'importe où en dehors de la classe.
- **La portée protégée :** l'élément n'est accessible que depuis la classe ou les classes héritant de celle-ci. En Python sont considérés comme protégé les éléments définis avec un underscore comme préfixe (_)
- **La portée privée :** l'élément ne peut pas être accédé ou modifié depuis l'extérieur de la classe. En Python sont considérés comme privé les éléments définis avec deux underscores comme préfixe (__)

Les portées en Python

```python
class MyClass:
    def __init__(self):
        self.public_attribute = "public"  # Portée publique
        self._protected_attribute = "protected"  # Portée protégée
        self.__private_attribute = "private"  # Portée privée

    def public_method(self):
        print("Public method called")

    def _protected_method(self):
        print("Protected method called")

    def __private_method(self):
        print("Private method called")
```

Attention !

En Python, par défaut, tous les membres d'une classe (attributs et méthodes) sont considérés comme publics.

L'héritage en POO

L'héritage est un concept clé de la programmation orientée objet (POO) qui permet à une classe (appelée classe fille) d'hériter des attributs et des méthodes d'une autre classe (appelée classe mère). Cela permet la réutilisation du code et la mise en œuvre de la relation "est-un".

- **La classe mère (super classe) :** destinée à léguer toutes ses variables et méthodes marquées comme protégées.
- **La classe fille (sous classe) :** hérite des attributs et des méthodes de la classe mère. Elle peut également ajouter de nouveaux attributs et méthodes ou modifier ceux hérités.

→ En Python, pour hériter d'une classe mère, il suffit d'indiquer son nom entre parenthèses après le nom de votre classe.

Les classes filles

```python
class Animal:
    def __init__(self, nom):
        self.nom = nom


class Chien(Animal):
    def __init__(self, nom, race):
        super().__init__(nom)
        self.race = race
```

super()

- Le mot-clé super est utilisé pour appeler les méthodes de la classe parente à partir de la classe enfant : **super().methode_classe_mere()**. Si celui-ci est appelé seul il fait référence au constructeur de la classe mère.
- Si vous ne définissez pas de constructeur à votre classe fille celui de la classe mère sera automatiquement utilisé lors de la création d'une instance.

L'héritage en Python

```python
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def manger(self):
        print(f"{self.nom} mange")


class Chien(Animal):
    def aboyer(self):
        print(f"{self.nom} aboie")


class Chat(Animal):
    def miauler(self):
        print(f"{self.nom} miaule")



chien = Chien("Max")
chien.manger()  # Appel de la méthode de la classe Animal
chien.aboyer()  # Appel de la méthode de la classe Chien

chat = Chat("Minou")
chat.manger()   # Appel de la méthode de la classe Animal
chat.miauler()  # Appel de la méthode de la classe Chat
```

Les classes abstraites

Pour déclarer une classe abstraite en Python : Commencez par importer le module **abc** (Abstract Base Classes). Ce qui vous permettra de déclarer des méthodes abstraites qui seront à définir dans les classes filles. Chacune des méthodes abstraites devra être précédée du décorateur **@abstractmethod** et contenir l'instruction **pass**.

```python
from abc import ABC, abstractmethod

class Forme(ABC):
    @abstractmethod
    def aire(self):
        pass

class Rectangle(Forme):
    def __init__(self, longueur, largeur):
        self.longueur = longueur
        self.largeur = largeur

    def aire(self):
        return self.longueur * self.largeur
```

Le polymorphisme

Le polymorphisme est un principe de la programmation orientée objet où deux objets héritant de la même classe partagent un comportement commun mais peuvent l'exprimer chacun à leur manière.

- Polymorphisme ad hoc [X]: Il permet à une fonction ou méthode d'accepter différents types d'arguments et d'agir en conséquence. Cela peut être réalisé par surcharge de fonction ou par utilisation de paramètres génériques.
- Polymorphisme d'héritage [✓]: Il permet à des objets de différentes classes d'être traités de la même manière s'ils sont des instances de la même superclasse. Cela implique que les méthodes d'une classe parente peuvent être redéfinies dans les classes enfants pour fournir des fonctionnalités spécifiques à chaque classe.

Polymorphisme d'héritage

```python
class Animal:
    def faire_son(self):
        pass

class Chien(Animal):
    def faire_son(self):
        return "Woof!"

class Chat(Animal):
    def faire_son(self):
        return "Meow!"

def faire_son_animal(animal):
    return animal.faire_son()



chien = Chien()
chat = Chat()

# Appelle la méthode de Chien
print(faire_son_animal(chien))

# Appelle la méthode de Chat
print(faire_son_animal(chat))
```

Sources

- Geekforgeek
- Wikipedia
- ChatGPT

TP 2 à 5

- TP 2 : https://adventofcode.com/2017/day/15
- TP 3 (Objet demandé): https://adventofcode.com/2017/day/13
- TP 4 (Objet demandé): https://adventofcode.com/2017/day/20

TP Bonus (difficulté croissante)

- https://adventofcode.com/2018/day/2
- https://adventofcode.com/2017/day/11
- https://adventofcode.com/2021/day/6
- https://adventofcode.com/2020/day/21