a is b → vérifie si a et b pointent vers le même objet mémoire.
***m →** appelé **unpacking :** permet d'englober plusieurs valeurs en tant que liste.
****kw →** Le double astérisque ** avant kw est un opérateur de déballage qui indique à Python de collecter tous les arguments nommés restants dans un dictionnaire nommé kw.

Décorateur : Un décorateur en Python est une fonction qui prend une autre fonction en tant qu'argument et qui étend ou modifie le comportement de cette fonction sans la modifier directement.

•**Annotation :** Les annotations en Java sont généralement utilisées par le compilateur pour détecter des erreurs ou ignorer des avertissements, pour la documentation, pour la génération de code ou la génération de fichiers, Techniquement, une annotation est une interface spéciale noté avec l'annotation @interface,

```
def fn_bavard(f):
    def f_interne(*p, **k):
        print('debut de f_interne()')
        f(*p, **k)
        print('fin de f_interne()')
    print('dans fn_bavard')
    return f_interne

@fn_bavard
def exemple(x, y='ok'):
    print('exemple:', y, x)

print('Appel à exemple')
exemple('?')
print(exemple.__qualname__)
```

```
import functools

def deprecated(func):
    """Décorateur pour marquer une fonction
    @functools.wraps(func)
    def wrapper_deprecated(*args, **kwargs):
        print(f"La fonction {func.__name__}
        return func(*args, **kwargs)

    return wrapper_deprecated

# Exemple d'utilisation du décorateur deprec
@deprecated
def fonction_depreciee():
    print("Cette fonction est dépréciée.")

# Appel de la fonction dépréciée
fonction_depreciee()
```

L'inversion de contrôle (IoC) est un principe de conception en programmation informatique où le contrôle du flux d'exécution est délégué à un conteneur ou un mécanisme externe. Au lieu que les composants soient responsables de la création ou de la gestion de leurs dépendances, celles-ci sont injectées dans les composants par un mécanisme externe. L'IoC favorise la modularité, la réutilisabilité et la testabilité du code en réduisant le couplage entre les composants."

Sans Ioc

```
class LecteurMusique:
    def __init__(self):
        self.source = ServiceDeStreaming()  # C

    def jouer(self):
        musique = self.source.obtenir_musique()
        # Code pour jouer la musique

class ServiceDeStreaming:
    def obtenir_musique(self):
        # Obtenir la musique du service de stre
```

Avec Ioc

```
class LecteurMusique:
    def __init__(self, source):
        self.source = source  # Injection de la

    def jouer(self):
        musique = self.source.obtenir_musique()
        # Code pour jouer la musique

class ServiceDeStreaming:
    def obtenir_musique(self):
        # Obtenir la musique du service de stre
```

Proxy : un objet qui agit comme une interface ou un substitut pour un autre objet afin de contrôler l'accès à celui-ci. Le proxy intercepte les appels à l'objet qu'il représente et peut effectuer des actions supplémentaires avant ou après la transmission de l'appel à l'objet réel.

Python définit les fonctions **hasattr(object, name)**, **getattr(object, name)**, **setattr(object, name, value)** et **delattr(object, name)** qui permettent respectivement de savoir si l'objet a un attribut du nom indiqué, de récupérer la valeur de l'attribut, de la modifier et de supprimer l'attribut. Comprendre le programme suivant (dont l'exécution se fait sans erreur).

TP3 : ProtectionHandler.java

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.*;

/**
 * ProtectionHandler implements InvocationHandler
 * @author Hamza Mouddene
 * @version 1.0
 */
public class ProtectionHandler implements InvocationHandler {
   /** Attributes of ProtectionHandler */
   private Object object;
   private String[] methodNames;

   /**
    * Constructio of ProtectionHandler.
    * @param object we will handle.
    * @param methodNames we will disable for object.
    */
   public ProtectionHandler(Object object, String[] methodNames) {
      this.object = object;
      this.methodNames = methodNames;
   }

   @Override
   public Object invoke(Object arg0, Method arg1, Object[] arg2) throws Thro
      if (Arrays.asList(this.methodNames).contains(arg1.getName())) {
         System.out.println("The proxy doesn't support " + arg1.getName() + "
                                                    method.");
         throw new UnsupportedOperationException();
      }
      return arg1.invoke(this.object, arg2);
   }
```

TP3 : MapProxy.java

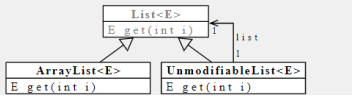
```
import java.util.*;

public class MapProxy {

   public static void main(String args[]) throws NoSuchMethodException
SecurityException,
                                                  Throwable {
      Random rand = new Random();
      Map<String, Integer> map = new HashMap<String, Integer>();
      System.out.println("Create an empty map : " + map);
      map.put("a", 2);
      map.put("b", 3);
      map.put("c", 5);
      map.put("d", 7);
      System.out.println("Fill the map : " + map);
      map.remove("d");
      System.out.println("Remove the key d (value = 7), map : " + map);

      String[] methodNames = {"put", "clear"};
      ProtectionHandler proxy = new ProtectionHandler(map,
methodNames);
      System.out.println("Create a proxy object of map : " + map);

      if (rand.nextInt(1000) % 2 == 0) {
         proxy.invoke(map, Map.class.getMethod("put",new Class[]
{Object.class,                                    Object.class}),          new
Object[]{"c", 5});
      }

      else {
         proxy.invoke(map, Map.class.getMethod("clear", new Class[]{}),
new Integer[]{});
      }

   }
```

TP3 : Proxy.py

```
class C:
    pass

class Proxy:

    def __init__(self, object, method_names):
        self.object = object
        self.method_names = method_names

    # Instance method
    def invoke(self, arg0, arg1, arg2):
        if (arg1 in self.method_names):
            raise AttributeError("The proxy doesn't support %s
method.",
                arg1.__name__)
        return getattr(C, arg1 + arg2)(arg0)

c = C()
proxy = Proxy(c, dir(c))
c.x = 5
assert c.x == 5
proxy.invoke(c, 'hasattr', "(c, 'x')")
assert hasattr(c, 'x')
assert getattr(c, 'x') == 5
assert not hasattr(c, 'y')
setattr(c, 'y', 7)
assert hasattr(c, 'y')
assert getattr(c, 'y') == 7
assert c.y == 7
assert vars(c) == {'x': 5, 'y': 7}
delattr(c, 'x')
assert not hasattr(c, 'x')
assert vars(c) == {'y': 7}
```

Dans les méthodes de collections (unmodifiedList,,,), le patron de conception utilisé est le patron décorateur qui permet d'attacher dynamiquement des fonctionnalités supplémentaires aux objets,

Cela est illustré dans le diagramme de classes ci-dessous, dans lequel on fait l'hypothèse que la méthode `unmodifiableList` retourne une instance d'une classe (non publique) `UnmodifiableList`.



L'introspection en java pour implémenter les unmodifiedlist

```
public class CollectionsProxy {

    public static <T> List<T> unmodifiableList(List<T> list) {
        return (List<T>) java.lang.reflect.Proxy.newProxyInstance(
            CollectionsProxy.class.getClassLoader(),
            new Class<?>[] { List.class },
            (proxy, method, args) -> {
                // Intercepte les appels aux méthodes de la liste
                if (method.getName().equals("add") ||
                    method.getName().equals("addAll") ||
                    method.getName().equals("clear") ||
                    method.getName().equals("remove") ||
                    method.getName().equals("removeAll") ||
                    method.getName().equals("retainAll") ||
                    method.getName().equals("set")) {
                    throw new UnsupportedOperationException("List is unmodifiable")
                }
                // Appel des autres méthodes sans modification
                return method.invoke(list, args);
            });
    }
```

L'introspection dans java pour l'implémentation de tels méthodes (unmodifedlist,,,) n'est pas utilisée principalement pour des raisons de performances, de complexité, de sécurité et de compatibilité. Au lieu de cela, les méthodes sont implémentées de manière statique en utilisant des techniques de programmation conventionnelles pour garantir la fiabilité, la performance et la compatibilité du code.

Pytest

si une fonction de test (fonction dont le nom commence par test) a un paramètre de même nom qu'un contexte de test, il sera automatiquement initialisé par `pytest` avec le résultat de la fonction de même nom.

Programmation par aspect

Aspect : Un aspect représente une préoccupation transversale ou une fonctionnalité qui peut être répétée à travers différentes parties du code. Par exemple, la journalisation, la gestion des transactions, la sécurité sont des exemples d'aspects.
Point de jonction (Join Point) : Un point de jonction est un endroit précis dans le code où une fonctionnalité transversale peut être appliquée. Cela peut être un appel de méthode, le lancement d'une exception ou d'autres événements du programme.
Coupe (Pointcut) : Une coupe définit un ensemble de points de jonction. C'est une expression qui permet de sélectionner les points de jonction où un aspect doit être appliqué. Par exemple, une coupe pourrait sélectionner tous les appels de méthodes dans une classe spécifique.
Greffon (Advice) : Un greffon est le code qui est exécuté à un point de jonction donné. Il représente la fonctionnalité transversale spécifique que vous souhaitez appliquer à votre application. Par exemple, un greffon de journalisation enregistrera des informations sur l'exécution d'une méthode.
Mécanisme d'introduction : Un mécanisme d'introduction permet d'ajouter de nouveaux attributs ou méthodes à une classe existante sans modifier son code source. Cela peut être utile pour ajouter des fonctionnalités à des classes sans avoir à les modifier directement.
Tissage (Weaving) : Le tissage est le processus de liaison d'un aspect à un point de jonction dans le code. Cela peut être fait à la compilation, au chargement ou à l'exécution du programme. Le tissage permet d'appliquer les aspects sélectionnés aux points de jonction spécifiés dans l'application.

Avantages :
1.Séparation des préoccupations : Facilite la modularité en séparant les préoccupations métier.
2.Réutilisabilité du code : Permet de réutiliser facilement le code à travers différentes parties de l'application.
3.Centralisation de la logique transversale : Simplifie la gestion des fonctionnalités communes en les regroupant dans des aspects distincts.
4.Amélioration de la lisibilité et de la maintenance : Rend le code plus lisible en se concentrant sur les fonctionnalités métier spécifiques.
Inconvénients :
1.Complexité accrue : Peut rendre le code plus complexe en introduisant des dépendances implicites entre les différents modules.
2.Perte de visibilité : Peut rendre le comportement de l'application moins évident en cachant certains aspects du fonctionnement interne.
3.Surcharge cognitive : Nécessite une courbe d'apprentissage plus longue pour les développeurs moins familiers avec ce paradigme.
4.Difficultés de test : Peut rendre les tests unitaires plus complexes en raison de la difficulté à isoler et tester les fonctionnalités métier individuelles.

L'advice ou le greffon

```
@Before("methodesPubliquesDesComptes()")
public void tracerAppel(JoinPoint joinPoint) {
    String methodName = joinPoint.getSignature().getName();
    System.out.println("Appel à la méthode " + methodName);
}
```

Définir une coupe

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TracerAppelsAspect {
    @Pointcut("execution(public * CompteSimple.*(..)) || execution(public * CompteCourant.*(..))"
    public void methodesPubliquesDesComptes() {}
}
```

JAspect : Exemple Complet

```
*********Pointcut*******************
pointcut publicMethod():
call(void CompteCourant.*(double)) ||
call(void CompteCourant.debiter(double)) ||
call(void CompteCourant.editerReleve()) ||
call(String CompteCourant.toString()) ||
call (String CompteSimple.getNumero()) ||
call (double CompteSimple.getSolde()) ||
call (Personne CompteSimple.getTitulaire()) ||
call(void CompteSimple.crediter(double)) ||
call(void CompteSimple.debiter(double)) ||
call(String CompteSimple.toString())
/**
pointcut publicMethod():
call(public CompteCourant.*(..)) ||
call (public CompteSimple.*(..))
*/

**********advice**********************
before() : publicMethod() {
        System.out.println(thisJoinPoint.getSignature() + " " + t
        hisJoinPoint.getArgs());
}

************Aspect****************
aspect DebitEleve {
        public static LIMITE = 450;

        before(double val) : debiter: (call(void *.debiter(double))) && args(val) {
                call (void DebitEleve.func(val))
        }

        public void func(double somme) {
                If (somme > LIMITE) {
                        System.out.println("send a notification in process ...")
                }
        }
}
```