```
#from trace import trace

def trace(f):
    def f_interne(*n):
        a = f.__qualname__
        print('--> ' + a + '(' + str(*n) +')' )
        resultat = f(*n)
        print('<--' + str(resultat))
        return(resultat)

    return f_interne

@trace
def fact(n):
    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)

@trace
def est_pair(n):
    return n == 0 or est_impair(n - 1)

def main():
    x = 3
    print(f'fact({x}) =', fact(3))

    print(f'{x} est', 'pair' if est_pair(x) else
'impair')

if __name__ == '__main__':
    main()
```

```
import functools
def fn_bavard(f):
    @functools.wraps(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__)
```

Un **décorateur** est une fonction qui modifie le comportement d'autres fonctions.

Les **décorateurs** sont utiles lorsque l'on veut ajouter du même code à plusieurs fonctions existantes.

L'inversion de contrôle (inversion of control, IoC) est un patron d'architecture commun à tous les frameworks (ou cadre de développement et d'exécution). Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework ou de la couche logicielle sous-jacente.

Un proxy (on parle aussi de procuration ou mandataire) est un objet qui est fourni à l'utilisateur à la place de l'objet réel qu'il attend. Le proxy sera donc un intermédiaire entre l'utilisateur et l'objet réel. Un proxy ajoute souvent des propriétés non fonctionnelles comme par exemple

```
public class ProtectionHandler implements
InvocationHandler {

    private Object obj;
    private String[] methods;

    public ProtectionHandler(Object o, String...
m) {
        obj = o;
        methods = m;
    }

    @Override
    public Object invoke(Object proxy, Method
method, Object[] args) throws Throwable {
```

	if (Arrays.asList(this.methods).contains(method.getName())) {
	System.out.println("The proxy doesn't support " + method.getName() + " method.");
	throw new UnsupportedOperationException();
	}
	return method.invoke(obj, args);
	}
	}

Java propose une classe Proxy qui s'appuie sur l'introspection pour définir des proxys : tous les appels de méthodes seront traités par un objet réalisant l'interface InvocationHandler qui réifie l'appel d'une méthode sous la forme d'une méthode invoke qui prend en paramètre la méthode appelée et ses paramètres effectifs. Un paramètre supplémentaire permet d'obtenir le proxy à l'origine de l'appel. La création effective du proxy se fait grâce à la classe Proxy. Elle permet de créer dynamiquement un proxy grâce à sa fabrique statique newProxyInstance qui prend en paramètre le chargeur de classe à utiliser (on prend celui qui a permis de charger la classe de l'objet qui nous intéresse, par exemple List.class.getClassLoader()), les interfaces que ce proxy réalisera (objet de type Class) et l'instance de InvocationHandler à utiliser pour traiter les appels de méthodes.	public class UnmodifiableList {
	
	public static void main(String[] args) {
	
	Map<Integer, String> M = new HashMap<>();
	ProtectionHandler phMap = new ProtectionHandler(M, "put", "clear");
	Map<Integer, String> B = (Map<Integer, String>) Proxy.newProxyInstance(Map.class.getClassLoader(),
	new Class[] { Map.class }, phMap);
	B.put(5, "y");
	}
	}