DESIGN PATTERNS - FICHE PRATIQUE

Cheatsheet : Répartir actions, états et collaborations

Les onze patrons comportementaux expliqués : interactions, transitions, parcours, exemples Python et exercices corrigés.

Par Ali ZAINOUL · Cours gratuit

Les onze patrons comportementaux du catalogue GoF (Gang of Four) répartissent les opérations et les interactions. Ils se distinguent par la question posée : choisir une politique, représenter une action, parcourir, notifier, coordonner, restaurer ou changer d'état.

Un contrat indique les paramètres, le résultat et les erreurs attendus. Un contexte est l'objet qui utilise une stratégie ou possède un état ; un collaborateur est l'objet auquel il délègue. Les exemples supposent les fonctions, les classes, l'héritage, les listes, les dictionnaires et les exceptions Python.

Chaque exemple est autonome et utilise uniquement Python 3 et sa bibliothèque standard. Les sorties sont commentées en anglais ; les explications et exercices sont en français. Les paiements et messages sont fictifs. Les annotations de types décrivent les valeurs attendues mais ne réalisent pas, à elles seules, une validation à l'exécution.

Chain of Responsibility

Faire traverser une chaîne de gestionnaires à une requête jusqu'à celui qui peut la traiter.

Quand le choisir ? Plusieurs responsables sont possibles et leur ordre de priorité doit rester configurable.

Exemple : Chain of Responsibility

Les requêtes sont ici des entiers. Le premier gestionnaire traite 1 à 10, le second 11 à 20, puis DefaultHandler signale les autres cas. Chaque gestionnaire décide de traiter ou de transmettre.

Python

```python
# chain-of-responsibility.py

class Handler:
    """
    Abstract handler class.
    """
    def __init__(self, successor=None):
        self._successor = successor

    def handle_request(self, request):
        """
        Handles the request if possible, otherwise defers to the successor.
        """
        if self._successor is not None:
            self._successor.handle_request(request)

class ConcreteHandler1(Handler):
    """
    Concrete handler that handles requests in the range 0 to 10.
    """
    def handle_request(self, request):
        if 0 < request <= 10:
            print(f"Handler 1 handled request {request}")
        else:
            super().handle_request(request)

class ConcreteHandler2(Handler):
    """
    Concrete handler that handles requests in the range 11 to 20.
    """
    def handle_request(self, request):
        if 10 < request <= 20:
            print(f"Handler 2 handled request {request}")
        else:
            super().handle_request(request)

class DefaultHandler(Handler):
```

```
    """
    Default handler that handles any request not handled by previous handlers.
    """
    def handle_request(self, request):
        print(f"End of chain, no handler for request {request}")

# Client code
handler1 = ConcreteHandler1()
handler2 = ConcreteHandler2()
default_handler = DefaultHandler()

handler1._successor = handler2
handler2._successor = default_handler

requests = [2, 5, 14, 22, 18, 3, 35, 27, 20]
for request in requests:
    handler1.handle_request(request)

# Expected output:
# Handler 1 handled request 2
# Handler 1 handled request 5
# Handler 2 handled request 14
# End of chain, no handler for request 22
# Handler 2 handled request 18
# Handler 1 handled request 3
# End of chain, no handler for request 35
# End of chain, no handler for request 27
# Handler 2 handled request 20
```

Résultat expliqué. 2 et 5 vont au premier gestionnaire, 14 au second et 22 au gestionnaire final. Les autres nombres suivent les mêmes bornes, sans notification à tous les maillons.

Erreur à éviter. L'ordre de la chaîne détermine la priorité. Un maillon final absent peut perdre une requête ; un cycle peut empêcher la terminaison. La chaîne n'est pas un envoi à tous les participants.

▼ **Qui traite 0, 10, 11 et 21 ?**

0 et 21 atteignent DefaultHandler ; 10 appartient au premier gestionnaire ; 11 au second. Ces quatre valeurs vérifient les limites des intervalles.

Command

Représenter une action et son destinataire par un objet que l'appelant peut déclencher plus tard.

Quand le choisir ? Une interface de commande doit être séparée des détails du récepteur : bouton, file de travail ou historique d'actions.

Exemple : Command

La télécommande associe un nom à un objet fournissant execute(). TurnOnCommand et TurnOffCommand conservent la lampe destinataire. Les méthodes de la lampe affichent une simulation.

Python

```python
# command.py

class Command:
    """
    Abstract command class.
    """
    def execute(self):
        """
        Executes the command.
        """
        raise NotImplementedError

class Light:
    """
    Receiver class that performs the actual actions.
    """
    def turn_on(self):
        print("Light turned on")

    def turn_off(self):
        print("Light turned off")

class TurnOnCommand(Command):
    """
    Concrete command to turn on the light.
    """
    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_on()

class TurnOffCommand(Command):
    """
    Concrete command to turn off the light.
    """
    def __init__(self, light):
```

```
        self._light = light

    def execute(self):
        self._light.turn_off()

class RemoteControl:
    """
    Invoker class that holds and executes commands.
    """
    def __init__(self):
        self._commands = {}

    def set_command(self, command_name, command):
        """
        Assigns a command to a specific button.
        """
        self._commands[command_name] = command

    def press_button(self, command_name):
        """
        Executes the command associated with the button.
        """
        if command_name in self._commands:
            self._commands[command_name].execute()
        else:
            print(f"No command associated with {command_name}")

# Client code
light = Light()
turn_on_command = TurnOnCommand(light)
turn_off_command = TurnOffCommand(light)

remote = RemoteControl()
remote.set_command("on", turn_on_command)
remote.set_command("off", turn_off_command)

remote.press_button("on")
remote.press_button("off")
```

```
# Expected output:
# Light turned on
# Light turned off
```

Résultat expliqué. Le bouton on affiche Light turned on ; le bouton off affiche Light turned off. La télécommande ne connaît pas ces méthodes, seulement execute().

Erreur à éviter. Encapsuler une action ne fournit pas automatiquement une annulation, une transaction ou une persistance. Annuler demande de capturer ou de reconstruire correctement l'état antérieur.

▼ Une commande allumer peut-elle s'annuler toujours par éteindre ?

Non. Si la lampe était déjà allumée, éteindre ne restaure pas son état antérieur. Une annulation correcte doit connaître l'état précédant l'exécution.

Iterator

Séparer une collection de la manière d'obtenir ses valeurs successives.

Quand le choisir ? Le client doit parcourir des éléments sans gérer directement la représentation interne de la collection.

Exemple : Iterator

L'itérateur conserve son indice. __iter__ retourne cet itérateur ; __next__ retourne une valeur ou lève StopIteration. La fonction Python next() appelle __next__. Numbers.create_iterator() crée un nouveau parcours.

Python

```python
# iterator.py

class Numbers:
    """
    Aggregate class holding a collection of numbers.
    """
    def __init__(self, numbers):
        self._numbers = numbers

    def create_iterator(self):
        """
        Creates and returns an iterator for the numbers collection.
        """
        return NumberIterator(self._numbers)

class NumberIterator:
    """
    Iterator class for traversing the numbers collection.
    """
    def __init__(self, numbers):
        self._numbers = numbers
        self._index = 0

    def has_next(self):
        """
        Checks if there are more elements to iterate over.
        """
        return self._index < len(self._numbers)

    def __iter__(self):
        return self

    def __next__(self):
        """
        Returns the next element in the collection.
        """
        if self.has_next():
```

```
            value = self._numbers[self._index]
            self._index += 1
            return value
        else:
            raise StopIteration

# Client code
numbers = Numbers([1, 2, 3, 4, 5])
iterator = numbers.create_iterator()

try:
    while True:
        print(next(iterator))
except StopIteration:
    pass

# Expected output:
# 1
# 2
# 3
# 4
# 5
```

Résultat expliqué. Les nombres 1 à 5 sont imprimés sur cinq lignes. À la fin, StopIteration interrompt la boucle try ; cette exception signifie la fin normale du parcours.

Erreur à éviter. Un itérateur consommé ne recommence pas. Cette version conserve la liste reçue plutôt qu'un instantané : une modification de cette liste peut affecter le parcours.

▼ Après épuisement, comment retrouver le premier élément ?

Créez un nouvel itérateur avec numbers.create_iterator(). Appeler encore next() sur l'ancien doit continuer à lever StopIteration.

Observer

Notifier plusieurs abonnés lorsqu'un sujet produit un événement ou change d'état.

Quand le choisir ? Plusieurs réactions indépendantes doivent être ajoutées ou retirées sans intégrer leur logique au sujet.

Exemple : Observer

Signal.connect enregistre une fonction acceptant un message ; send appelle les fonctions d'un instantané de la liste ; disconnect retire une fonction. Aucun paquet externe ni service de messagerie n'est utilisé.

Python

```python
# observer.py

class Signal:
    def __init__(self):
        self._listeners = []

    def connect(self, listener):
        if listener not in self._listeners:
            self._listeners.append(listener)

    def disconnect(self, listener):
        self._listeners.remove(listener)

    def send(self, message):
        for listener in tuple(self._listeners):
            listener(message)


def observer_function(message):
    print(f"Observer received: {message}")

signal = Signal()
signal.connect(observer_function)
signal.send("State changed")
signal.disconnect(observer_function)
signal.send("No listener receives this message")

# Expected output:
# Observer received: State changed
```

Résultat expliqué. Le premier message est affiché une fois. Après disconnect, le second n'a plus de destinataire et ne produit aucune sortie.

Erreur à éviter. Les appels sont synchrones. Une exception dans un observateur interrompt la suite ; les références conservées par le sujet doivent être libérées lorsque l'abonnement cesse.

▼ Pourquoi parcourir tuple(self._listeners) plutôt que la liste directement ?

L'instantané stabilise les destinataires de la notification en cours. Un désabonnement pendant un appel ne décale pas les éléments que le parcours doit encore visiter.

Strategy

Confier une opération à une politique interchangeable reçue par le contexte.

Quand le choisir ? Une même opération possède plusieurs variantes et le client doit pouvoir choisir laquelle utiliser.

Exemple : Strategy

ShoppingCart reçoit un objet fournissant pay(amount). ABC et abstractmethod, du module abc, déclarent ce contrat abstrait. Les deux implémentations affichent uniquement des paiements fictifs ; float est choisi pour la simulation, pas comme convention monétaire réelle.

Python

```python
# strategy.py

from abc import ABC, abstractmethod

class PaymentStrategy(ABC):
    """
    Abstract base class for payment strategies.
    Defines the common interface that all strategies must implement.
    """
    @abstractmethod
    def pay(self, amount: float):
        """
        Abstract method to perform payment.
        Subclasses must implement this method.
        """
        pass

class CreditCardPayment(PaymentStrategy):
    """
    Concrete strategy for credit card payments.
    """
    def __init__(self, card_number: str, cvv: str):
        self.card_number = card_number
        self.cvv = cvv

    def pay(self, amount: float):
        print(f"Paying {amount} using Credit Card: {self.card_number}")

class PayPalPayment(PaymentStrategy):
    """
    Concrete strategy for PayPal payments.
    """
    def __init__(self, email: str):
        self.email = email

    def pay(self, amount: float):
        print(f"Paying {amount} using PayPal: {self.email}")
```

```python
class ShoppingCart:
    """
    Context class that uses the strategy.
    It doesn't know the concrete implementation of the strategy.
    """
    def __init__(self, payment_strategy: PaymentStrategy):
        self.payment_strategy = payment_strategy

    def checkout(self, amount: float):
        self.payment_strategy.pay(amount)

# Example usage
credit_card = CreditCardPayment(card_number="1234-5678-9012-3456", cvv="123")
paypal = PayPalPayment(email="user@example.com")

cart1 = ShoppingCart(payment_strategy=credit_card)
cart1.checkout(100.0)  # Paying 100.0 using Credit Card: 1234-5678-9012-3456

cart2 = ShoppingCart(payment_strategy=paypal)
cart2.checkout(50.0)   # Paying 50.0 using PayPal: user@example.com

# Expected output:
# Paying 100.0 using Credit Card: 1234-5678-9012-3456
# Paying 50.0 using PayPal: user@example.com
```

Résultat expliqué. Un panier délègue 100.0 au paiement par carte fictive ; l'autre délègue 50.0 à la politique PayPal fictive. Le panier garde la même opération checkout.

Erreur à éviter. Le choix de la politique reste à la charge du client. Une fonction suffit pour une stratégie sans état. Ne conservez ni cryptogramme ni coordonnées sensibles dans une application réelle ; définissez une unité et des arrondis pour les montants.

▼ Qu'est-ce qui distingue principalement Strategy de State ?

Strategy représente une politique choisie pour réaliser une opération. State représente une phase du cycle de vie qui modifie les opérations permises et organise des transitions.

Template Method

Fixer le squelette d'un algorithme et laisser les sous-classes réaliser certaines étapes.

Quand le choisir ? Des variantes partagent un ordre d'opérations stable mais diffèrent dans plusieurs étapes.

Exemple : Template Method

build_house appelle fondations, murs, toit, fenêtres et finition dans cet ordre. ABC et abstractmethod rendent trois étapes obligatoires. Le nom HouseBuilder n'en fait pas un Builder : le déroulement est ici hérité, sans objet produit assemblé par un collaborateur indépendant.

Python

```python
# template-method.py

from abc import ABC, abstractmethod

class HouseBuilder(ABC):
    """
    Abstract class defining the template for building a house.
    """

    def build_house(self):
        """
        Template method to define the order of building steps.
        """
        self.prepare_foundation()
        self.build_walls()
        self.install_roof()
        self.install_windows()
        self.finish_house()

    @abstractmethod
    def prepare_foundation(self):
        """
        Abstract method to prepare the foundation.
        Subclasses must implement this.
        """
        pass

    @abstractmethod
    def build_walls(self):
        """
        Abstract method to build the walls.
        Subclasses must implement this.
        """
        pass

    @abstractmethod
    def install_roof(self):
```

```
        """
        Abstract method to install the roof.
        Subclasses must implement this.
        """
        pass

    def install_windows(self):
        """
        Concrete method to install windows.
        """
        print("Installing windows")

    def finish_house(self):
        """
        Concrete method to finish the house.
        """
        print("Finishing house")

class WoodenHouseBuilder(HouseBuilder):
    """
    Concrete class to build a wooden house.
    """

    def prepare_foundation(self):
        print("Preparing wooden foundation")

    def build_walls(self):
        print("Building wooden walls")

    def install_roof(self):
        print("Installing wooden roof")

class StoneHouseBuilder(HouseBuilder):
    """
    Concrete class to build a stone house.
    """

    def prepare_foundation(self):
        print("Preparing stone foundation")
```

```
    def build_walls(self):
        print("Building stone walls")

    def install_roof(self):
        print("Installing stone roof")

# Usage
wooden_house_builder = WoodenHouseBuilder()
wooden_house_builder.build_house()

stone_house_builder = StoneHouseBuilder()
stone_house_builder.build_house()

# Expected output:
# Preparing wooden foundation
# Building wooden walls
# Installing wooden roof
# Installing windows
# Finishing house
# Preparing stone foundation
# Building stone walls
# Installing stone roof
# Installing windows
# Finishing house
```

Résultat expliqué. Chaque construction imprime cinq étapes. Les trois premières varient selon bois ou pierre ; fenêtres puis finition sont partagées et conservent leur position.

Erreur à éviter. L'héritage lie les variantes à la classe de base. Une variation qui doit changer pendant la vie d'une instance justifie souvent la composition et Strategy.

▼ Comment personnaliser les fenêtres sans changer l'ordre général ?

Redéfinissez install_windows dans une sous-classe et conservez build_house. Le point de variation est l'étape ; le squelette commun contrôle toujours l'ordre des appels.

Mediator

Centraliser les interactions de plusieurs collègues dans un coordinateur explicite.

Quand le choisir ? Les composants auraient sinon des références directes vers de nombreux autres composants et des règles de coordination dispersées.

Exemple : Mediator

Chaque User connaît le ChatRoom. Le salon conserve les inscrits et distribue send(sender, message) aux autres utilisateurs. Les boîtes de réception sont des listes locales : aucun message réseau n'est envoyé.

Python

```python
# mediator.py

class ChatRoom:
    def __init__(self):
        self._users = []

    def register(self, user):
        if user not in self._users:
            self._users.append(user)

    def send(self, sender, message):
        if sender not in self._users:
            raise ValueError("Unknown sender")
        for user in tuple(self._users):
            if user is not sender:
                user.receive(sender.name, message)

class User:
    def __init__(self, name, room):
        self.name = name
        self.room = room
        self.inbox = []
        room.register(self)

    def send(self, message):
        self.room.send(self, message)

    def receive(self, sender, message):
        self.inbox.append((sender, message))

room = ChatRoom()
alice = User("Alice", room)
bob = User("Bob", room)
alice.send("Hello")
print(bob.inbox)    # [('Alice', 'Hello')]
print(alice.inbox)  # []
```

```
# Expected output:
# [('Alice', 'Hello')]
# []
```

Résultat expliqué. Bob reçoit le couple ('Alice', 'Hello') ; Alice ne reçoit pas son propre message. Le salon choisit les destinataires sans que l'utilisateur parcoure une liste de collègues.

Erreur à éviter. Le médiateur peut concentrer trop de responsabilités. Observer insiste sur la diffusion aux abonnés ; Mediator insiste sur la coordination de collègues. Ces intentions peuvent coexister.

▼ Ajoutez Charlie au salon puis envoyez le même message. Combien de nouvelles réceptions sont attendues ?

Bob et Charlie reçoivent chacun le message une fois, Alice aucune. Le nombre de destinataires est donc deux pour ce salon de trois utilisateurs.

Memento

Capturer un état puis permettre à son propriétaire de le restaurer sans demander à l'historique de le reconstruire.

Quand le choisir ? Une fonction de retour arrière doit retrouver plusieurs champs cohérents et l'historique doit rester indépendant de leur logique de modification.

Exemple : Memento

Editor est l'origine de l'état, Snapshot le cliché, history son gardien. dataclass génère notamment le constructeur ; frozen=True interdit la réaffectation des champs. Texte et curseur sont immuables, donc le cliché ne partage pas de liste mutable avec l'éditeur.

Python

```python
# memento.py

from dataclasses import dataclass

@dataclass(frozen=True)
class Snapshot:
    text: str
    cursor: int

class Editor:
    def __init__(self):
        self.text = ""
        self.cursor = 0

    def insert(self, fragment):
        self.text = self.text[:self.cursor] + fragment +
self.text[self.cursor:]
        self.cursor += len(fragment)

    def save(self):
        return Snapshot(self.text, self.cursor)

    def restore(self, snapshot):
        self.text = snapshot.text
        self.cursor = snapshot.cursor

editor = Editor()
editor.insert("Hello")
history = [editor.save()]
editor.insert(" world")
editor.restore(history.pop())
print(editor.text, editor.cursor)  # Hello 5

# Expected output:
# Hello 5
```

Résultat expliqué. L'éditeur ajoute Hello puis world, avant de restaurer le cliché. Le texte redevient Hello et le curseur retrouve 5 : les deux champs sont restaurés ensemble.

Erreur à éviter. frozen=True ne rend pas les objets imbriqués immuables. Les champs restent publics en Python ; le gardien doit les traiter comme opaques. Capturer des états volumineux coûte de la mémoire.

▼ Pourquoi ne pas sauvegarder seulement le texte ?

Après restauration, un curseur resté à une autre position pourrait rendre l'édition suivante incohérente. Le contrat du cliché doit inclure tout l'état nécessaire à la reprise.

State

Déplacer les comportements propres à une phase de vie dans un objet état et organiser les transitions.

Quand le choisir ? Plusieurs opérations changent ensemble selon une phase, comme brouillon puis document publié.

Exemple : State

Document commence avec Draft. edit et publish délèguent à l'état. Le brouillon vérifie le texte avant de passer à Published ; le document publié refuse l'édition et une nouvelle publication.

Python

```python
# state.py

class Draft:
    def publish(self, document):
        if not document.text.strip():
            raise ValueError("An empty document cannot be published")
        document.state = Published()

    def edit(self, document, text):
        document.text = text

class Published:
    def publish(self, document):
        raise RuntimeError("Already published")

    def edit(self, document, text):
        raise RuntimeError("Published documents cannot be edited")

class Document:
    def __init__(self, text):
        self.text = text
        self.state = Draft()

    def publish(self):
        self.state.publish(self)

    def edit(self, text):
        self.state.edit(self, text)

document = Document("First draft")
document.edit("Ready")
document.publish()
print(type(document.state).__name__)  # Published
try:
    document.edit("Too late")
except RuntimeError as error:
    print(error)
```

```
# Expected output:
# Published
# Published documents cannot be edited
```

Résultat expliqué. La première édition prépare Ready, la publication remplace l'état, puis une nouvelle édition lève RuntimeError. Le bloc try affiche l'erreur attendue sans arrêter le programme.

Erreur à éviter. Deux conditions stables peuvent être plus simples que plusieurs classes. Les transitions et les opérations interdites doivent être testées ; l'attribut state public peut être modifié hors protocole dans cette démonstration.

▼ Quel état doit rester après une publication refusée pour texte vide ?

Draft : la validation échoue avant la transition. Le document doit encore pouvoir être corrigé puis publié.

Visitor

Ajouter des opérations à une famille de types en plaçant chaque opération dans un visiteur.

Quand le choisir ? Les types d'éléments sont relativement stables mais de nouveaux traitements doivent être ajoutés.

Exemple : Visitor

Text et Picture sont des dataclasses : ce décorateur génère notamment leur constructeur à partir des champs annotés. accept choisit explicitement visit_text ou visit_picture. Python ne sélectionne pas automatiquement une surcharge selon le type de l'argument.

Python

```python
# visitor.py

from dataclasses import dataclass

@dataclass
class Text:
    value: str

    def accept(self, visitor):
        return visitor.visit_text(self)

@dataclass
class Picture:
    caption: str

    def accept(self, visitor):
        return visitor.visit_picture(self)

class WordCount:
    def visit_text(self, element):
        return len(element.value.split())

    def visit_picture(self, element):
        return len(element.caption.split())

class KindLabel:
    def visit_text(self, element):
        return "text"

    def visit_picture(self, element):
        return "picture"

elements = [Text("Hello world"), Picture("A small tree")]
print(sum(element.accept(WordCount()) for element in elements))  # 5
print([element.accept(KindLabel()) for element in elements])     # ['text',
'picture']
```

```
# Expected output:
# 5
# ['text', 'picture']
```

Résultat expliqué. WordCount additionne deux mots de texte et trois mots de légende, soit 5. KindLabel produit une liste contenant text puis picture, avec les mêmes éléments.

Erreur à éviter. Ajouter un visiteur évite de modifier les éléments, mais ajouter un type d'élément oblige à étendre les visiteurs. L'accès à l'état interne doit respecter les contrats d'encapsulation.

▼ Si vous ajoutez un élément Audio, quelles parties doivent apprendre à le traiter ?

Audio doit fournir accept et chaque visiteur doit définir visit_audio. Ce coût révèle le compromis : ajout d'opérations facile, ajout de types plus transversal.

Interpreter

Représenter les constructions d'une petite grammaire par des objets qui évaluent une expression dans un contexte.

Quand le choisir ? Un langage de règles restreint a une grammaire stable et ses expressions peuvent être représentées par de petits arbres.

Exemple : Interpreter

La grammaire est variable, NOT expression ou expression AND expression. Les dataclasses génèrent les constructeurs des nœuds. Variable lit un booléen du dictionnaire ; Not le nie ; And combine avec court-circuit. L'arbre est construit à la main, sans analyse de texte ni eval.

Python

```python
# interpreter.py

from dataclasses import dataclass

@dataclass(frozen=True)
class Variable:
    name: str

    def interpret(self, context):
        value = context[self.name]
        if type(value) is not bool:
            raise TypeError("Rule values must be booleans")
        return value

@dataclass(frozen=True)
class And:
    left: object
    right: object

    def interpret(self, context):
        return self.left.interpret(context) and self.right.interpret(context)

@dataclass(frozen=True)
class Not:
    expression: object

    def interpret(self, context):
        return not self.expression.interpret(context)

rule = And(Variable("has_ticket"), Not(Variable("is_blocked")))
print(rule.interpret({"has_ticket": True, "is_blocked": False}))  # True
print(rule.interpret({"has_ticket": True, "is_blocked": True}))   # False

# Expected output:
# True
# False
```

Résultat expliqué. has_ticket et non is_blocked donne True pour un ticket valide sans blocage, puis False lorsque le blocage est vrai. Seuls les booléens réellement évalués sont vérifiés.

Erreur à éviter. Une variable évaluée absente lève KeyError ; une valeur évaluée non booléenne lève TypeError. Le court-circuit peut éviter une branche. Une grammaire complexe justifie un analyseur et une validation séparés.

▼ Dans And(Variable('left'), Variable('missing')), que se passe-t-il si left vaut False ?

Le résultat est False sans lire missing : and court-circuite. Si left vaut True, la branche droite est évaluée et l'absence de missing lève KeyError.

Télécharger cette fiche en PDF · **Toutes les fiches par famille**

Version web : https://www.clearcode.fr/courses/programming/design-patterns/cheatsheet/behavioral