DESIGN PATTERNS - CHAPITRE 26

Décorateurs, contextes et parcours paresseux

Utiliser décorateurs, contextes, itérateurs et générateurs avec un cycle de vie explicite.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Prévoir l'ordre et le contrat des décorateurs.
- Distinguer résultat différé, chargement des données et durée de vie des fichiers.

Prérequis. Avoir étudié les patrons de création, de structure et de comportement.

Décorateurs, contextes et parcours paresseux

Observer l'ordre des décorateurs

`@italic_decorator` appliqué au-dessus de `@bold_decorator` correspond à `italic_decorator(bold_decorator(get_message))`. Prévoyez l'ordre des balises avant d'exécuter. Ces chaînes illustrent la composition ; n'insérez pas de texte non fiable en HTML sans l'échapper.

Python

```python
# example.py
from functools import wraps

# Define a decorator to make text bold
def bold_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return "<strong>" + func(*args, **kwargs) + "</strong>"
    return wrapper

# Define a decorator to make text italic
def italic_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return "<em>" + func(*args, **kwargs) + "</em>"
    return wrapper

# Apply the decorators to the function
@italic_decorator
@bold_decorator
def get_message():
    return "Hello, decorated world!"

# Using the decorated function
message = get_message()
print(message) # Output: <em><strong>Hello, decorated world!</strong></em>
```

Comparer un registre implicite à un registre explicite

Ce second registre rappelle le fonctionnement d'une métaclasse. Le nom `AbstractProduct` ne rend pas la classe abstraite : elle n'utilise ni `ABC` ni méthode abstraite. Pour deux produits connus, un dictionnaire littéral est une alternative plus directe.

Python

```python
# example.py
# Define a metaclass for abstract products
class AbstractProductMeta(type):
    def __init__(cls, name, bases, attrs):
        super().__init__(name, bases, attrs)
        if not hasattr(cls, 'products'):
            cls.products = {}
        else:
            # Register concrete products
            cls.products[name] = cls

# Define an abstract product class using the metaclass
class AbstractProduct(metaclass=AbstractProductMeta):
    pass

# Define concrete product classes
class ConcreteProductA(AbstractProduct):
    def __init__(self):
        self.name = "Product A"

class ConcreteProductB(AbstractProduct):
    def __init__(self):
        self.name = "Product B"

# Access registered products through the base class
product_a = AbstractProduct.products['ConcreteProductA']()
print(product_a.name)

product_b = AbstractProduct.products['ConcreteProductB']()
print(product_b.name)
```

Gestionnaire de contexte : garantir une sortie

Le protocole `with` appelle `__enter__` à l'entrée puis `__exit__` à la sortie, y compris lorsqu'une exception survient dans le bloc. `perf_counter()` mesure une durée écoulée. Ici `__exit__` ne retourne pas de valeur vraie : une éventuelle exception continue donc sa propagation.

Python

```python
# example.py
import time

# Define a context manager for timing code execution
class TimerContext:
    def __enter__(self):
        self.start_time = time.perf_counter()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.end_time = time.perf_counter()
        print(f"Elapsed time: {self.end_time - self.start_time:.4f} seconds")

# Define a function to be timed
def my_function():
    sum(range(1000))

# Using the TimerContext with 'with' statement
with TimerContext():
    my_function()
```

Décorer une méthode, pas une classe par erreur

Ces enveloppes attendent `self` et `message` : elles doivent décorer `send`. Les placer sur la classe remplacerait le constructeur par une fonction de signature incompatible. Les préfixes "Encrypted" et "Compressed" sont des simulations de transformation, sans chiffrement ni compression effectifs.

Python

```python
# example.py
from functools import wraps

class Notifier:
    def send(self, message):
        print(f"Sending notification: {message}")

def encrypt_message(func):
    @wraps(func)
    def wrapper(self, message):
        encrypted_message = f"Encrypted: {message}"  # Simulate encryption
        return func(self, encrypted_message)
    return wrapper

def compress_message(func):
    @wraps(func)
    def wrapper(self, message):
        compressed_message = f"Compressed: {message}"  # Simulate compression
        return func(self, compressed_message)
    return wrapper

class SecureNotifier(Notifier):
    @encrypt_message
    @compress_message
    def send(self, message):
        return super().send(message)

# Example Usage:
secure_notifier = SecureNotifier()
secure_notifier.send("This is a secret message.")
```

Enregistrer une fonction sans la remplacer

Le décorateur d'abonnement peut retourner la fonction reçue après l'avoir enregistrée. Conserver cette identité permet d'utiliser la même référence pour un désabonnement. Une enveloppe supplémentaire serait inutile tant que l'appel n'est pas modifié.

Python

```python
# example.py
class EventManager:
    def __init__(self):
        # Dictionary to store observers for each event
        self._observers = {}

    def subscribe(self, event_name, observer):
        # Add an observer to a specific event
        if event_name not in self._observers:
            self._observers[event_name] = []
        self._observers[event_name].append(observer)

    def unsubscribe(self, event_name, observer):
        # Remove an observer from a specific event
        if event_name in self._observers:
            self._observers[event_name].remove(observer)

    def notify(self, event_name, data=None):
        # Notify all observers of a specific event
        if event_name in self._observers:
            for observer in tuple(self._observers[event_name]):
                observer(data)

# Event Manager Instance
event_manager = EventManager()

def event_subscriber(event_name):
    # Decorator to subscribe a function to an event
    def decorator(func):
        event_manager.subscribe(event_name, func)
        return func
    return decorator

# Example Usage:
@event_subscriber("user_logged_in")
def log_user_login(user_data):
    print(f"User logged in: {user_data['username']}")
```

```python
@event_subscriber("order_created")
def send_order_confirmation(order_data):
    print(f"Order confirmation sent for order ID: {order_data['order_id']}")

# Simulate events
event_manager.notify("user_logged_in", {"username": "john.doe"})
event_manager.notify("order_created", {"order_id": "12345"})
```

Itérateur : préparer un jeu de fichiers complet

`TemporaryDirectory` crée un répertoire provisoire et le supprime à la sortie de `with`. `Path` crée ici deux fichiers de démonstration. Le filtre ne garde que les fichiers `.txt`. L'itérateur construit d'abord toute la liste des noms : il n'est donc pas paresseux pour cette étape.

Python

```python
# example.py
import os

class FileIterator:
    """
    Iterator that yields files with a specific extension from a directory.
    """
    def __init__(self, directory, extension):
        """
        Initializes the FileIterator.

        Args:
            directory (str): The directory to iterate through.
            extension (str): The file extension to filter by.
        """
        self.directory = directory
        self.extension = extension
        self.file_list = [f for f in sorted(os.listdir(directory)) if
f.endswith(extension) and os.path.isfile(os.path.join(directory, f))]
        self.index = 0

    def __iter__(self):
        """
        Returns the iterator object itself.
        """
        return self

    def __next__(self):
        """
        Returns the next file with the specified extension.

        Raises:
            StopIteration: If there are no more files with the specified
extension.
        """
        if self.index < len(self.file_list):
            file_name = self.file_list[self.index]
```

```
            self.index += 1
            return os.path.join(self.directory, file_name)
        else:
            raise StopIteration

from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory(prefix="patterns-files-") as directory_path:
    Path(directory_path, "notes.txt").write_text("Hello", encoding="utf-8")
    Path(directory_path, "image.png").write_bytes(b"demo")
    extension_filter = ".txt"
    file_iterator = FileIterator(directory_path, extension_filter)
    for file_path in file_iterator:
        print(Path(file_path).name)  # notes.txt
```

Générateur : différer la production des résultats

Une fonction contenant `yield` retourne un générateur, qui est un itérateur. Son corps commence au premier parcours et reprend après chaque `yield`. Ce générateur diffère la production des chemins, mais `sorted(os.listdir(...))` charge encore tous les noms : il ne garantit pas une mémoire constante. Le parcours reste à l'intérieur du bloc `with`, avant la suppression des fichiers.

Après consommation, `list(generateur)` est vide. Pour un nouveau parcours, appelez à nouveau la fonction. La lecture de ces fichiers est une expérience locale, pas un benchmark de performance.

Python

```python
# example.py
import os

def file_generator(directory, extension):
    """
    Generator that yields files with a specific extension from a directory.

    Args:
        directory (str): The directory to iterate through.
        extension (str): The file extension to filter by.

    Yields:
        str: The path to the next file with the specified extension.
    """
    for file_name in sorted(os.listdir(directory)):
        if file_name.endswith(extension) and
os.path.isfile(os.path.join(directory, file_name)):
            yield os.path.join(directory, file_name)

from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory(prefix="patterns-files-") as directory_path:
    Path(directory_path, "notes.txt").write_text("Hello", encoding="utf-8")
    Path(directory_path, "image.png").write_bytes(b"demo")
    extension_filter = ".txt"
    for file_path in file_generator(directory_path, extension_filter):
        print(Path(file_path).name)  # notes.txt
```

Version web : https://www.clearcode.fr/courses/programming/design-patterns/outils-python