Gaël Penessot
Auteur Business Intelligence avec Python

7 APPROCHES AVANCÉES POUR OPTIMISER VOTRE CODE PYTHON

Les Générateurs

Optimisation des séquences infinies

```python
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Utilisation du générateur (économe en mémoire)
gen = fibonacci_generator()
first_10 = [next(gen) for _ in range(10)]
```

Les générateurs permettent de créer des séquences infinies sans surcharger la mémoire, réduisant l'utilisation de O(n) à O(1).

List Comprehensions Imbriquées

Transformations de données complexes

```
# Liste de commandes clients avec plusieurs articles
orders = [
    [("item1", 10), ("item2", 20)],
    [("item1", 30)],
    [("item3", 40), ("item1", 50), ("item2", 60)]
]

# Calcul du total par article en une expression
item_totals = {
    item: sum(price for order in orders
              for item_name, price in order if item_name == item)
    for item in {item for order in orders for item, _ in order}
}
```

Les list comprehensions imbriquées permettent d'exprimer des transformations complexes de données en peu de lignes.

Décorateurs Adaptatifs

Fonctions qui s'adaptent au contexte

```python
from functools import wraps
import time

def smart_retry(max_attempts=3, delay=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            attempts = 0
            while attempts < max_attempts:
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    attempts += 1
                    if attempts == max_attempts:
                        raise
                    time.sleep(delay * attempts)
            return None
        return wrapper
    return decorator

# Usage
@smart_retry(max_attempts=3, delay=2)
def fetch_data(url):
    # Simulated API call that might fail
    pass
```

Les décorateurs permettent d'ajouter des fonctionnalités à vos fonctions sans modifier leur code source.

Gestionnaires de Contexte

Gestion élégante des ressources

```python
from contextlib import contextmanager
import threading

class ResourcePool:
    def __init__(self, max_resources=3):
        self.semaphore = threading.Semaphore(max_resources)
        self.resources = []

    @contextmanager
    def acquire_resource(self):
        try:
            self.semaphore.acquire()
            resource = {"id": "resource"}
            self.resources.append(resource)
            yield resource
        finally:
            self.resources.remove(resource)
            self.semaphore.release()

# Usage
pool = ResourcePool(max_resources=2)
with pool.acquire_resource() as res:
    # Utiliser la ressource en toute sécurité
    pass
```

Les gestionnaires de contexte garantissent la libération propre des ressources, même en cas d'erreur.

Métaclasses

Génération dynamique de classes

```
class ValidationMeta(type):
    def __new__(cls, name, bases, attrs):
        # Ajoute validation aux méthodes
        for key, value in attrs.items():
            if callable(value) and not key.startswith('__'):
                attrs[key] = cls.validate_types(value)
        return super().__new__(cls, name, bases, attrs)

    @staticmethod
    def validate_types(func):
        def wrapper(*args, **kwargs):
            # Validation simplifiée
            return func(*args, **kwargs)
        return wrapper

# Usage
class ValidatedClass(metaclass=ValidationMeta):
    def process_data(self, data: list, factor: int) -> list:
        return [x * factor for x in data]
```

Les métaclasses permettent de modifier le comportement des classes à leur définition.

Programmation Asynchrone

Coroutines efficaces

```python
import asyncio
from typing import List

async def process_item(item: int) -> int:
    await asyncio.sleep(0.1)  # Simulation I/O
    return item * 2

async def process_batch(items: List[int]) -> List[int]:
    # Avec limite de concurrence
    semaphore = asyncio.Semaphore(3)

    async def bounded_process(item):
        async with semaphore:
            return await process_item(item)

    tasks = [bounded_process(item) for item in items]
    return await asyncio.gather(*tasks)

# Usage
async def main():
    items = [1, 2, 3, 4, 5]
    results = await process_batch(items)
```

L'async permet d'exécuter des opérations concurrentes sans bloquer, idéal pour les opérations d'I/O.

Programmation Fonctionnelle

Composition de fonctions

```python
from functools import reduce

def compose(*functions):
    """Compose right to left"""
    return reduce(lambda f, g: lambda x: f(g(x)), functions)

def pipe(*functions):
    """Compose left to right"""
    return reduce(lambda f, g: lambda x: g(f(x)), functions)

# Fonctions simples
def double(x): return x * 2
def add_one(x): return x + 1
def stringify(x): return str(x)

# Création d'un pipeline
pipeline = pipe(double, add_one, stringify)
result = pipeline(5)  # "11"
```

La composition de fonctions crée des pipelines de traitement réutilisables et élégants.