

Python 開發中的關鍵軟體工程原則

關注點分離 (Separation of Concerns, SoC)

關注點分離是一種設計原則，目的是將程式碼分割成不同部分，每個部分專注於特定功能或問題領域。

Python 中的實踐方式

不良實踐：混合關注點

```
def process_user_data(user_id):  
    # 資料庫邏輯  
    conn = sqlite3.connect('users.db')  
    cursor = conn.cursor()  
    cursor.execute("SELECT * FROM users WHERE id=?", (user_id,))  
    user = cursor.fetchone()  
  
    # 業務邏輯  
    full_name = f"{user[1]} {user[2]}"  
    age = calculate_age(user[3])  
  
    # 顯示邏輯  
    print(f"User: {full_name}, Age: {age}")  
  
    conn.close()
```

良好實踐：分離關注點

資料存取層

```
def get_user(user_id):  
    conn = sqlite3.connect('users.db')  
    cursor = conn.cursor()  
    try:  
        cursor.execute("SELECT * FROM users WHERE id=?", (user_id,))  
        return cursor.fetchone()  
    finally:  
        conn.close()
```

業務邏輯層

```
def process_user(user):  
    return {  
        'full_name': f"{user[1]} {user[2]}",  
        'age': calculate_age(user[3])  
    }
```

顯示層

```
def display_user_info(user_info):  
    print(f"User: {user_info['full_name']}, Age: {user_info['age']}")
```

協調功能

```
def user_workflow(user_id):  
    user = get_user(user_id)
```

```
user_info = process_user(user)
display_user_info(user_info)
```

常見實踐模式

1. **MVC/MTV 模式**：在 Django 中使用 Models (資料)、Templates (顯示)、Views (邏輯)
2. **分層架構**：資料存取層、業務層、表示層
3. **模組化設計**：將功能分解為獨立模組，如 `auth.py`, `database.py`, `api.py`

DRY 原則 (Don't Repeat Yourself)

DRY 原則強調每一個知識點在系統中都應該有單一、明確的表示，避免重複程式碼。

Python 中的實踐方式

```
# 違反 DRY 的程式碼
def validate_email(email):
    import re
    pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
    return bool(re.match(pattern, email))

def register_user(username, email):
    # 重複的電子郵件驗證邏輯
    import re
    pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
    if not bool(re.match(pattern, email)):
        raise ValueError("Invalid email")
    # 註冊邏輯...
```

```
# 遵循 DRY 的程式碼
def validate_email(email):
    import re
    pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
    return bool(re.match(pattern, email))

def register_user(username, email):
    if not validate_email(email):
        raise ValueError("Invalid email")
    # 註冊邏輯...
```

DRY 實踐技巧

1. **抽取共用函數**：將重複邏輯提取為獨立函數
2. **使用裝飾器**：重用橫切關注點的程式碼

```
def log_execution_time(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} executed in {time.time() - start}s")
        return result
    return wrapper
```

3. 類繼承與組合：使用繼承或組合來共享功能
4. 設定常數與配置：集中管理常數和配置值

測試 (Testing)

測試是確保程式碼質量、功能正確性和防止迴歸的關鍵實踐。

Python 測試工具與框架

1. **unittest**：Python 標準庫內建測試框架
2. **pytest**：更現代、靈活的測試框架
3. **mock**：用於模擬外部依賴

測試類型與實踐

```
# 單元測試範例 (pytest)
def calculate_discount(price, discount_rate):
    if discount_rate < 0 or discount_rate > 1:
        raise ValueError("Discount rate must be between 0 and 1")
    return price * (1 - discount_rate)

def test_calculate_discount():
    # 正常情況
    assert calculate_discount(100, 0.1) == 90
    # 邊界情況
    assert calculate_discount(100, 0) == 100
    assert calculate_discount(100, 1) == 0
    # 異常情況
    import pytest
    with pytest.raises(ValueError):
        calculate_discount(100, 1.5)
```

測試最佳實踐

1. 測試金字塔：
 - 更多的單元測試（快速且聚焦）
 - 較少的整合測試（驗證組件協作）
 - 更少的端到端測試（驗證整個系統）
2. 測試驅動開發 (TDD)：
 - 先寫測試

- 實現功能以通過測試
- 重構程式碼保持品質

3. Fixture 與測試資料：

```
@pytest.fixture
def sample_user():
    return {"id": 1, "name": "Test User", "email": "test@example.com"}

def test_user_processing(sample_user):
    result = process_user(sample_user)
    assert "Test User" in result["full_name"]
```

4. 模擬外部依賴：

```
def test_user_service(mock):
    # 模擬資料庫調用
    mock_db = mock.patch('module.database.get_user')
    mock_db.return_value = {"id": 1, "name": "Test"}

    result = get_user_details(1)
    assert result["name"] == "Test"
    mock_db.assert_called_once_with(1)
```

版本控制 (Version Control)

版本控制系統 (VCS) 追蹤和管理檔案的變更，Git 是目前最流行的版本控制系統。

Git 工作流程與最佳實踐

1. 分支策略：

- `main/master`：穩定的生產版本
- `develop`：開發分支
- 功能分支：特定功能開發
- 發布分支：準備發布版本
- 修補分支：緊急問題修復

2. 提交規範：

類型(範圍)： 簡短描述

詳細描述

解決 #123

類型可以是：feat, fix, docs, style, refactor, test, chore 等

3. Python 特定版本控制考量：

- 使用 `.gitignore` 排除 `__pycache__`, `*.pyc`, 虛擬環境

- 將依賴管理在 `requirements.txt` 或 `Pipfile` 中
- 考慮使用 `pyproject.toml` 作為現代 Python 專案配置

4. CI/CD 整合：

- 使用 GitHub Actions, GitLab CI, Jenkins 等
- 自動執行測試、Linting、代碼品質檢查
- 自動部署到不同環境

實際工作流示例

```
# 創建功能分支
git checkout -b feature/user-authentication

# 編寫程式碼和測試
# ...

# 提交變更
git add .
git commit -m "feat(auth): implement JWT authentication"

# 推送到遠端倉庫
git push -u origin feature/user-authentication

# 創建 Pull Request / Merge Request
# 代碼審核，自動化測試通過

# 合併到主要分支
git checkout develop
git merge --no-ff feature/user-authentication
git push origin develop
```

這些原則和實踐相輔相成，共同提升 Python 程式碼的可維護性、可靠性和開發效率。遵循這些最佳實踐，可以讓團隊開發更加順暢，產品質量更有保障。

pandas、NumPy 和 SciPy 詳細比較與解析

概述

這三個是 Python 數據科學和科學計算領域中最基礎和常用的函式庫，它們有不同的特點和用途：

NumPy

NumPy 是最基礎的科學計算函式庫，它提供了：

- 高效能的多維數組對象 (ndarray)
- 向量化操作，大幅提升計算速度
- 基本的數學函數和線性代數操作
- 隨機數生成器

NumPy 適合處理同質型數據（所有元素類型相同），是其他科學計算庫的基礎。

pandas

pandas 建立在 NumPy 之上，主要專注於數據處理和分析：

- 提供 DataFrame 和 Series 數據結構，可處理異質型數據（不同列可有不同數據類型）
- 強大的數據清洗、轉換和合併功能
- 時間序列功能和日期範圍生成
- CSV、Excel、SQL 等各種格式的數據輸入輸出
- 具有數據透視表等資料分析功能

pandas 非常適合數據清理、探索性分析和數據預處理。

SciPy

SciPy 建立在 NumPy 之上，提供更多高級科學和工程計算功能：

- 統計分析函數
- 訊號處理
- 圖像處理
- 最佳化算法
- 線性代數的進階操作
- 積分和微分方程求解
- 聚類算法等

SciPy 是進行科學計算和工程應用的高階工具箱。

三者關係

這三個函式庫可以被視為層次化關係：

- NumPy 是基礎，提供核心的數組計算能力
- pandas 建立在 NumPy 之上，專注於數據處理和分析
- SciPy 同樣建立在 NumPy 之上，提供科學計算的各種功能

在實際項目中，它們通常一起使用：使用 pandas 讀取和清理數據，NumPy 進行數值計算，SciPy 進行更高階的科學計算。

詳細比較表

特性	NumPy	pandas	SciPy
主要功能	高效能多維數組處理	數據處理與分析	科學與工程計算
核心數據結構	ndarray (多維同質數組)	DataFrame, Series (可處理異質數據)	基於 NumPy 數組
適用場景	數值計算、向量/矩陣運算	數據清理、轉換、分析	科學計算、統計分析、信號處理

特性	NumPy	pandas	SciPy
數據處理能力	基礎操作（切片、索引）	強大的數據處理（過濾、分組、合併）	較少直接數據處理功能
統計功能	基本統計（均值、標準差）	描述性統計和聚合	高級統計（分布、檢驗）
輸入/輸出	讀寫數組	多種格式（CSV、Excel、SQL 等）	專業格式（如 MATLAB 文件）
數據類型	同質（同類型）數據	異質（混合類型）數據	主要處理數值型數據
特殊功能	向量化運算、廣播功能	時間序列、數據透視表	積分、優化、信號處理、圖像處理
線性代數	基本操作	較少線性代數功能	完整且高級的線性代數庫
內存效率	高	中（因功能多而較 NumPy 低）	高
計算速度	快	較 NumPy 慢但仍高效	快（針對專門任務優化）
建立於	C 和 Python	NumPy、Cython	NumPy
典型用途	科學計算基礎	數據分析、數據準備	科學研究、工程應用
可視化整合	與 Matplotlib 配合	內建簡單繪圖功能	特定繪圖功能（如優化結果）

典型使用案例

NumPy

- 數學運算和向量計算
- 隨機數生成和操作
- 作為其他科學計算庫的基礎

pandas

- 數據清理和轉換
- 缺失值處理
- 時間序列分析
- 數據分析與探索

SciPy

- 信號處理與頻譜分析
- 圖像處理和操作
- 微分方程求解
- 最佳化問題

- 聚類和機器學習基礎算法
- 統計檢驗和分析

NumPy 向量化和廣播機制詳解

向量化 (Vectorization)

向量化是 NumPy 最重要的特性之一，它允許對整個數組執行操作，而無需使用顯式的循環。

向量化的概念

向量化是指使用單一操作處理整個數組的能力，而不是逐個元素處理。這種方法有幾個關鍵優勢：

- 性能提升**：向量化操作通常在底層使用優化的 C/C++ 代碼實現，比 Python 循環快數十倍甚至數百倍
- 代碼簡潔**：減少了代碼行數，提高可讀性
- 減少錯誤**：簡化的代碼降低了錯誤的可能性

向量化與循環的比較

以計算兩個數組元素和為例：

使用循環 (非向量化)：

```
import numpy as np
import time

# 創建兩個大型數組
a = np.random.random(1000000)
b = np.random.random(1000000)
c = np.zeros(1000000)

# 使用循環計算
start = time.time()
for i in range(len(a)):
    c[i] = a[i] + b[i]
print(f"循環耗時: {time.time() - start} 秒")
```

使用向量化：

```
# 使用向量化操作
start = time.time()
c = a + b
print(f"向量化耗時: {time.time() - start} 秒")
```

向量化版本通常比循環版本快 10-100 倍，視數組大小和操作而定。

常見的向量化操作

NumPy 提供了豐富的向量化操作：

- 算術運算：`+`, `-`, `*`, `/`, `**` (指數) 等
- 比較運算：`>`, `<`, `==`, `!=` 等
- 邏輯運算：`&` (與), `|` (或), `~` (非) 等
- 數學函數：`np.sin`, `np.cos`, `np.exp`, `np.log` 等
- 統計函數：`np.sum`, `np.mean`, `np.std` 等

向量化的應用場景

- 數據轉換與正規化
- 信號處理
- 金融計算
- 圖像處理
- 機器學習算法

廣播 (Broadcasting)

廣播是 NumPy 的另一個強大特性，它定義了不同形狀數組之間如何進行算術運算。

廣播的概念

廣播允許 NumPy 在執行操作時，自動處理不同維度或形狀的數組，無需實際複製數據。這使得向量化操作更加靈活，同時保持高效率。

廣播規則

NumPy 的廣播遵循以下規則：

1. 維度匹配：從尾部開始比較數組形狀
2. 相容條件：對應維度必須相等，或其中一個為 1，或其中一個不存在
3. 自動擴展：維度為 1 或不存在的數組會被"廣播"，表現得像它擁有與另一數組相匹配的尺寸

廣播示例

1. 標量與數組運算

```
# 將所有元素乘以 2
a = np.array([1, 2, 3, 4])
b = 2
c = a * b # [2, 4, 6, 8]
```

這裡，標量 `2` 被廣播到與數組 `a` 相同的形狀。

2. 一維與二維數組運算

```
# 為矩陣的每一列加上不同的值
a = np.array([[1, 2, 3], [4, 5, 6]]) # 2x3 矩陣
b = np.array([10, 20, 30])          # 1x3 向量
c = a + b # [[11, 22, 33], [14, 25, 36]]
```

這裡，`b` 被廣播成與 `a` 相同的形狀 `[[10, 20, 30], [10, 20, 30]]`。

3. 複雜的廣播示例

```
a = np.ones((3, 4, 1)) # 形狀為 (3, 4, 1)
b = np.ones((5, 1, 3)) # 形狀為 (5, 1, 3)
c = a + b               # 形狀為 (5, 4, 3)
```

廣播的視覺化

考慮兩個數組：

```
A:      (2, 1, 3)
B:      (   3, 1)
結果 c: (2, 3, 3)
```

廣播過程：

1. B 在前面加一個維度變成 (1, 3, 1)
2. 將 A 和調整後的 B 的每個維度分別廣播：
 - (2, 1, 3) → (2, 3, 3)
 - (1, 3, 1) → (2, 3, 3)

廣播的優勢

- 記憶體效率：不需要實際複製數據
- 計算效率：避免了顯式循環和臨時數組
- 代碼簡潔：可以用簡短直觀的代碼表達複雜操作

廣播的常見應用

- 數據標準化：從每列減去平均值
- 特徵縮放：矩陣的每列乘以不同係數
- 距離計算：計算點集之間的成對距離
- 圖像處理：應用濾鏡或顏色轉換
- 時間序列分析：與季節性因子相乘

向量化與廣播的結合使用

向量化和廣播通常一起使用，創造極具表現力和高效的代碼：

```
# 計算多個點到多個中心點的歐氏距離
points = np.random.random((1000, 3))    # 1000個3D點
centers = np.random.random((10, 3))     # 10個中心點

# 計算所有點到所有中心的距離
# 無需循環即可得到 1000x10 的距離矩陣
diff = points[:, np.newaxis, :] - centers[np.newaxis, :, :] # 形狀 (1000, 10, 3)
sq_dist = (diff ** 2).sum(axis=2) # 形狀 (1000, 10)
distances = np.sqrt(sq_dist)
```

這個例子結合了向量化運算和廣播，一次性計算了 1000 個點到 10 個中心點的所有距離（共 10,000 個距離值），而無需使用任何顯式循環。

最佳實踐與注意事項

- **了解數組形狀**：在使用廣播前，確保了解參與運算的數組形狀
- **避免意外廣播**：不當的廣播可能導致大量記憶體使用
- **使用 `reshape` 或 `newaxis`**：調整數組形狀以符合廣播規則
- **廣播診斷**：當廣播失敗時，仔細檢查錯誤訊息並分析數組形狀
- **使用 `axis` 參數**：在聚合函數中正確使用 `axis` 參數有助於維持所需維度