Session 2 : Composition de Classes et Chargement de Données

Objectifs de la session

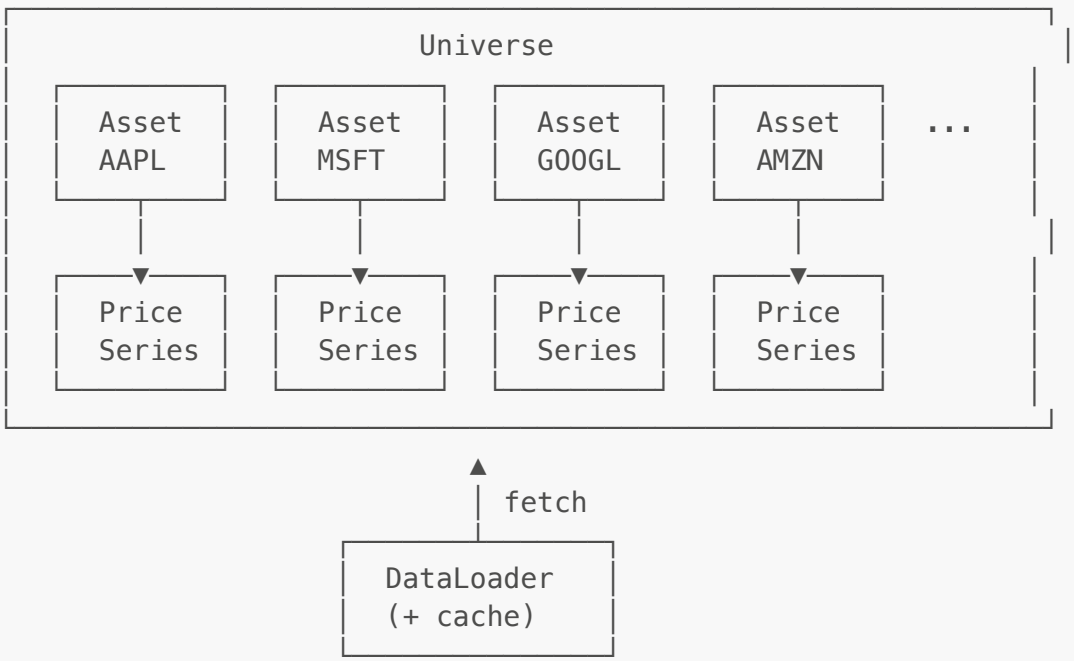
1. Introduire la programmation orientée-objet et les principes fondamentaux de conception de classes
2. Utiliser les types de collections intégrés à Python (listes, dictionnaires, ensembles)
3. Maîtriser la syntaxe des expressions de compréhension
4. Récupérer des données à partir d'APIs publiques
5. Implémenter un système de cache des données

Contexte du projet

Dans la Session 1, nous avons créé notre classe `PriceSeries` pour représenter et manipuler des séries de prix. Nous pouvons maintenant construire des abstractions de plus haut niveau :

1. Une classe **Asset** qui *contient* une `PriceSeries` (composition)
2. Une classe **DataLoader** pour récupérer des données de marché avec cache intelligent
3. Une classe **Universe** pour gérer une collection d'actifs

Architecture cible



Exemple d'utilisation finale : ce qu'on aimerait atteindre

```
from pyvest.src.priceseries import PriceSeries
from pyvest.src.asset import Asset
```

```python
from pyvest.src.loader import DataLoader
from pyvest.src.universe import Universe

# On instancie un "loader" à partir du DataLoader avec système de cache
loader = DataLoader(cache_dir=".cache")

# Récupération des données pour un actif via l'API Yahoo Finance
apple_ts = loader.fetch_single_ticker("AAPL", "Close", ("2024-01-01",
"2024-12-01"))

# Test du système de cache (le second appel devrait être instantané)
apple_ts = loader.fetch_single_ticker("AAPL", "Close", ("2024-01-01",
"2024-12-01"))

# Création de nos objets Asset
apple = Asset("AAPL", apple_ts, sector="Technology")
msft = Asset("MSFT", loader.fetch_single_ticker("MSFT", "Close", ("2024-
01-01", "2024-12-01")), sector="Technology")

# Communication entre Asset et son interface PriceSeries
print(f"Volatilité AAPL: {apple.volatility:.2%}")

# Corrélation entre deux actifs
correlation = apple.correlation_with(msft)
print(f"Corrélation AAPL-MSFT: {correlation:.2f}")

# Agrégation dans un objet Universe
universe = Universe([apple, msft])
for asset in universe:
    print(f"{asset.ticker}: {asset.total_return:.2%}")
```

Partie 1

Étape 1.1 : La classe Asset et le pattern de Composition

Quelques mots suppléementaires sur la POO

La **programmation orientée-objet (POO)** est un paradigme qui permet de structurer le code autour d'**objets**, des entités définies par leurs caractéristiques (attributs) et leurs comportements (méthodes). Une **classe** agit comme un patron (ou usine) capable de produire des objets d'un type donné.

Les trois piliers de la POO :

Pilier	Description	Exemple
Encapsulation	Regrouper données et comportements dans une même unité	`Asset` encapsule ticker, prix, secteur

Pilier	Description	Exemple
Abstraction	Exposer une interface simple, cacher la complexité	`asset.volatility` cache le calcul sous-jacent
Polymorphisme	Différents objets peuvent intéragir avec la même interface, que ces comportements découlent d'une superclasse (**héritage**s) ou non	Tout objet avec `__len__` fonctionne avec `len()`

> **Note importante** : La POO n'est pas toujours nécessaire. Un programme simple peut fonctionner plus clairement comme une combinaison de fonctions. Voir la présentation de Jack Diederich "Stop Writing Classes" (PyCon 2012). Cependant, pour une librairie structurée comme la nôtre, la POO apporte une organisation claire et maintenable.

Analyse des besoins

Avant de coder, identifions les classes principales et leurs responsabilités :

Classe	Responsabilité	Attributs	Méthodes
PriceSeries	Stocke et opère sur une série de prix	`values`, `name`	`get_log_return()`, `get_annualized_volatility()`
Asset	Représente un actif financier et ses métadonnées	`ticker`, `prices`, `sector`	`correlation_with()`
DataLoader	Récupère et met en cache les données de marché	`cache_dir`	`fetch_single_ticker()`, `fetch_multiple_tickers()`
Universe	Gère une collection d'actifs	`_assets`	`add()`, `get()`, `filter_by_sector()`

Les principes SOLID

Ces principes guident la conception de classes robustes et maintenables :

Principe	Description
Single Responsibility	Une classe = une responsabilité claire
Open/Closed	Concevoir des classes extensibles (généralement par héritage) sans devoir modifier le code interne
Liskov Substitution	Concevoir une sous-classe de telle manière qu'on puisse l'échanger à sa classe parente
Interface Segregation	Concevoir des interfaces minimales, ciblées et minimiser les dépendances entre classes qui collaborent

Principe	Description
Dependency Inversion	On devrait éviter de concevoir une classe importante de haut-niveau (proche de l'utilisateur finale / organise le comportement de l'app) comme dépendante de pleins de classes bas-niveau légérement différentes à gérer. On devrait plutôt inverser la situation: notre classe devrait dépendre d'une classe "abstraite" (une classe `ABC` en Python mais que vous pouvez comprendre par abstraite au sens large) et toutes les classes bas niveau dépendrait (=hériterait) de la classe abstraite pour les forcer à suivre le pattern général et à gérer en interne leurs détails

Types de relations entre classes

Relation	Question à se poser	Exemple
Composition	"B appartient-il à A ? Le cycle de vie de B dépend-il de A ?"	`Asset` *contient* une `PriceSeries`
Agrégation	"A contient-il B alors que B peut exister indépendamment ?"	`Universe` *contient* des `Asset`
Héritage	"A est-il un type spécialisé de B ?"	Non utilisé pour l'instant

Implémentation de la classe Asset

```python
# Fichier: pyvest/src/constant.py
from enum import StrEnum

class CurrencyEnum(StrEnum):
    """Énumération des devises supportées."""
    USD = "USD"
    EUR = "EUR"
    GBP = "GBP"
    JPY = "JPY"
```

```python
# Fichier: pyvest/src/asset.py

from pyvest.src.priceseries import PriceSeries
from .constant import CurrencyEnum


class Asset:
    """
    Représente un actif financier avec son historique de prix.
    Pattern de conception : COMPOSITION
    Asset POSSÈDE une PriceSeries (relation HAS-A, pas IS-A).
    ----------
    Attributes:
```

```
            ticker: Symbole (ex: 'AAPL')
            prices: Instance PriceSeries contenant l'historique
            sector: Classification sectorielle optionnelle
            currency: Devise des prix (défaut: USD)
        """

        def __init__(
            self,
            ticker: str,
            prices: PriceSeries,
            sector: str | None = None,
            currency: CurrencyEnum = CurrencyEnum.USD
        ) -> None:
            # Validation des entrées dans le constructeur
            if not ticker or not ticker.strip():
                raise ValueError("Le ticker ne peut pas être vide")
            if len(prices) == 0:
                raise ValueError("La série de prix ne peut pas être vide")

            self.ticker = ticker.upper()  # Normalisation en majuscules
            self.prices = prices  # Composition : Asset POSSÈDE une
    PriceSeries
            self.sector = sector
            self.currency = currency

        def __repr__(self) -> str:
            """Représentation pour le développement."""
            return f"Asset({self.ticker!r}, {len(self.prices)} prices)"

        def __str__(self) -> str:
            """Représentation pour l'utilisateur."""
            return f"{self.ticker}: ${self.current_price:.2f}"
```

Test dans le REPL :

```
>>> from pyvest.src.priceseries import PriceSeries
>>> from pyvest.src.asset import Asset
>>> ps = PriceSeries([100.0, 105.0, 103.0, 110.0], "Close")
>>> apple = Asset("AAPL", ps, "tech")
>>> apple
Asset('AAPL', 4 prices)
>>> apple.ticker
'AAPL'
>>> apple.prices.total_return
```

On pourrait vouloir ajouter un autre test dans l'initialisateur: par exemple on ne veut pas de prix négatifs.

```
class Asset:
    # ...
```

```
    def __init__(
        self,
        ticker: str,
        prices: PriceSeries,
        sector: str | None = None,
        currency: CurrencyEnum = CurrencyEnum.USD
    ) -> None:
        # Validation des entrées dans le constructeur
        if not ticker or not ticker.strip():
            raise ValueError("Le ticker ne peut pas être vide")
        if len(prices) == 0:
            raise ValueError("La série de prix ne peut pas être vide")
        if any(prices < 0):
            raise ValueError("La série de prix ne peut pas contenir de
valeurs négatives")

    # reste du code
```

Test dans le REPL :

```
from pyvest.src.pyvest import Asset, PriceSeries

asset = Asset("AAPL", PriceSeries([100, 102, 103, 104], "Close"), "tech")
>>> TypeError: '<' not supported between instances of 'PriceSeries' and
'int'
```

Plusieurs solutions s'offrent à nous mais la plus 'pythonique' pourrait être d'ajouter la dunder méthode `__lt__` (less than) à PriceSeries. On pourrait également convertir l'attribut `values` de notre classe en numpy array. Je vous laisse investiguer ces solutions.

Étape 2.2 : Propriétés et délégation vers PriceSeries

Le décorateur @property

Une **propriété** est un hybride entre un attribut et une méthode. Elle permet d'accéder à une valeur calculée avec la syntaxe d'un attribut (sans parenthèses), tout en exécutant du code derrière.

Avantages des propriétés :

- Syntaxe propre : `asset.volatility` au lieu de `asset.get_volatility()`
- Lecture seule par défaut (protection des données)
- Calcul à la demande (lazy evaluation)
- Possibilité d'ajouter de la validation

```
# Ajout des propriétés à la classe Asset
```

```python
class Asset:
    # ... (code précédent) ...

     @property
    def current_price(self) -> float:
        """Dernier prix connu."""
        return self.prices.values[-1]

    @property
    def volatility(self) -> float:
        """Volatilité annualisée (délègue à PriceSeries)."""
        return self.prices.get_annualized_volatility()

    @property
    def total_return(self) -> float:
        """Rendement total (délègue à PriceSeries)."""
        return self.prices.total_return

    @property
    def sharpe_ratio(self) -> float:
        """Ratio de Sharpe (délègue à PriceSeries)."""
        return self.prices.sharpe_ratio()

    @property
    def max_drawdown(self) -> float:
        """Drawdown maximum (délègue à PriceSeries)."""
        return self.prices.max_drawdown()
```

Comprendre les décorateurs

Un **décorateur** est une fonction qui prend une autre fonction en argument et retourne une version modifiée de celle-ci:

```python
@decorator
def ma_fonction():
    print("Ceci est ma fonction")

# Est équivalent à :
def ma_fonction():
    print("Ceci est ma fonction")
ma_fonction = decorator(ma_fonction)
```

Points clés :

1. **Exécution à l'import** : Le décorateur s'exécute quand le module est chargé, pas quand la fonction est appelée
2. **Closure** : Le décorateur est composé d'une fonction externe et d'une fonction externe. la fonction externe est la fonction qui porte le nom du décorateur et qui prend comme argument la fonction qui

va être décorée. La fonction interne du décorateur a accès aux variables de la fonction externe (scope `nonlocal`)
3. **Préservation des métadonnées** : Utiliser `@functools.wraps` pour conserver les attributs `__name__` et `__doc__` de notre fonction

Exemple : décorateur de chronométrage

```
import time
import functools

def chronometre(func):
    """Décorateur qui mesure le temps d'exécution d'une fonction."""
    @functools.wraps(func)  # Préserve __name__ et __doc__
    def wrapper(*args, **kwargs):
        debut = time.perf_counter()
        resultat = func(*args, **kwargs)
        duree = time.perf_counter() - debut
        print(f"[{duree:.4f}s] {func.__name__}")
        return resultat
    return wrapper

@chronometre
def calcul_lent():
    time.sleep(1)
    return 42
```

Étape 2.3 : Méthode de corrélation entre actifs

Contexte

La corrélation est une mesure fondamentale en finance qui permet de modéliser la structure conjointe de deux actifs. Les rendements ont généralement une corrélation positive car les actifs évoluent avec le reste du marché. La corrélation de Pearson mesure la relation linéaire entre deux variables. Pour deux actifs, nous calculons la corrélation de leurs log-rendements.

Formule de Pearson :

$$\rho_{X,Y} = \frac{\text{Cov}(X, Y)}{\sigma_X \cdot \sigma_Y} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}}$$

```
import numpy as np

class Asset:
    # ... (code précédent) ...

    def correlation_with(self, other: "Asset") -> float:
        """
        Calcule la corrélation de Pearson des log-rendements avec un autre
```

```
actif.

        Args:
            other: Un autre Asset

        Returns:
            Coefficient de corrélation entre -1 et 1
        """
        # Récupération des log-rendements
        x = np.array(self.prices.get_all_log_returns())
        y = np.array(other.prices.get_all_log_returns())

        # Alignement des longueurs (gestion des séries de tailles
différentes)
        n = min(len(x), len(y))
        if n < 2:
            raise ValueError(
                f"Pas assez d'observations communes: {n}. "
                "Minimum requis: 2."
            )

        x = x[:n]
        y = y[:n]

        # Centrage des valeurs (soustraction de la moyenne)
        x_centered = x - np.mean(x)
        y_centered = y - np.mean(y)

        # Covariance (numérateur)
        covariance = np.dot(x_centered, y_centered) / (n - 1)

        # Variances pour le dénominateur
        var_x = np.dot(x_centered, x_centered) / (n - 1)
        var_y = np.dot(y_centered, y_centered) / (n - 1)

        # Vérification de la variance nulle
        if var_x == 0 or var_y == 0:
            raise ValueError(
                "Variance nulle détectée. "
                "La corrélation n'est pas définie pour une série
constante."
            )

        return covariance / np.sqrt(var_x * var_y)
```

Étape 2.4 : Git Commit

```
git add pyvest/src/asset.py
git commit -m "feat: add Asset class with composition over PriceSeries"
```

Étape 3 : DataLoader avec système de cache

De manière générale, un système de cache est un système permettant de stocker temporairement la donnée résultant d'un calcul couteux, qu'on veut pouvoir accéder plus rapidement sans réiterer à chaque fois ce calcul.

Installation de yfinance

```
# Dans le terminal :
uv pip install yfinance
# ou
pip install yfinance
```

`yfinance` est une librairie open-source non officielle qui simplifie l'accès à l'API Yahoo Finance. Elle supporte les prix historiques, les données fondamentales et les données d'options.

Conception de l'interface du DataLoader

```
Interface publique :
├── fetch_single_ticker(ticker, price_col, dates) -> PriceSeries
├── fetch_multiple_tickers(tickers, price_col, dates) -> dict[str,
PriceSeries]
└── clear_cache() -> int

Interface privée :
├── _get_cache_path(ticker, price_col, dates) -> Path
├── _check_date_overlap(...) -> tuple[str, Timestamp, Timestamp]
├── _load_from_cache(...) -> tuple[DataFrame, str, tuple]
└── _save_to_cache(...) -> None
```

Pourquoi un système de cache ?

Raison	Explication
Performance	Éviter les appels API redondants, chargement local rapide
Rate limiting	Les APIs ont souvent des limites de requêtes
Reproductibilité	Garantir l'utilisation du même dataset en recherche

Architecture du cache

Le système de cache gère quatre scénarios de correspondance temporelle :

```
Cas 1: EXACT - La requête correspond exactement au cache
Cache:    |-------------|
Requête:  |-------------|
→ Retourner directement les données en cache

Cas 2: CONTAINS - Le cache contient la période demandée
Cache:    |-----------------|
Requête:      |---------|
→ Découper les données en cache

Cas 3: OVERLAP_AFTER - Chevauchement à droite
Cache:    |---------|
Requête:      |-----------|
→ Fusionner cache + nouvelles données à droite

Cas 4: OVERLAP_BEFORE - Chevauchement à gauche
Cache:          |---------|
Requête:   |-----------|
→ Fusionner nouvelles données à gauche + cache

Cas 5: MISS - Aucun chevauchement
Cache:    |-----|
Requête:              |-----|
→ Télécharger toutes les données
```

Implémentation du DataLoader

```python
# Fichier: pyvest/src/loader.py


from pathlib import Path
import logging
import pickle
from datetime import datetime
from typing import Sequence

import pandas as pd
import yfinance as yf

from .priceseries import PriceSeries

class DataLoader:
    """
    Charge des données de marché depuis Yahoo Finance avec un système de
cache.

    Le système de cache gère cinq scénarios de correspondance temporelle :
    1. EXACT : La requête correspond exactement aux données en cache
    2. CONTAINS : La requête est un sous-ensemble du cache
```

```
    3. OVERLAP_AFTER : Intersection partielle, fetch complémentaire à
droite
    4. OVERLAP_BEFORE : Intersection partielle, fetch complémentaire à
gauche
    5. MISS : Aucune donnée en cache, fetch complet nécessaire

    Attributes:
        cache_dir: Répertoire de stockage du cache
        logger: Logger pour le suivi des opérations

    """

    def __init__(self, cache_dir: str = ".cache") -> None:
        self.cache_dir = Path(cache_dir)
        self.cache_dir.mkdir(exist_ok=True)
        self.logger = logging.getLogger(self.__class__.__name__)

    def _get_cache_path(
        self,
        ticker: str,
        price_col: str,
        dates: tuple[str, str]
    ) -> Path:
        """
        Génère le chemin du fichier cache pour une requête donnée.

        Format: {ticker}_{price_col}_{start}_{end}.pkl
        """
        return self.cache_dir / f"
{ticker}_{price_col}_{dates[0]}_{dates[1]}.pkl"

    def _check_date_overlap(
        self,
        cached_start: pd.Timestamp,
        cached_end: pd.Timestamp,
        req_start: pd.Timestamp,
        req_end: pd.Timestamp
    ) -> tuple[str, pd.Timestamp | None, pd.Timestamp | None]:
        """
        Détermine le type de chevauchement entre le cache et la requête

        Returns:
            tuple: (status, gap_start, gap_end)
            - status: "exact" | "contains" | "overlap_before" |
"overlap_after" | "miss"
            - gap_start: Début de la période manquante (si overlap)
            - gap_end: Fin de la période manquante (si overlap)
        """
        # Cas MISS: Aucune intersection
        if cached_end < req_start or cached_start > req_end:
            return ("miss", None, None)

        # Cas exact: hit parfait du cache
        if cached_start == req_start and cached_end == req_end:
```

```
            return ("exact", None, None)

        # Cas CONTAINS: hit du cache qui contient complétement la requête
        if cached_start <= req_start and cached_end >= req_end:
            return ("contains", None, None)

        # Cas OVERLAP_AFTER: cache hit mais la requête débordre à droite
        if cached_start <= req_start and cached_end < req_end:
            gap_start = cached_end + pd.Timedelta(days=1)
            gap_end = req_end
            return ("overlap_after", gap_start, gap_end)

        # Cas OVERLAP_BEFORE: cache hit mais la requête déborde à gauche
        if cached_start > req_start and cached_end >= req_end:
            gap_start = req_start
            gap_end = cached_start - pd.Timedelta(days=1)
            return ("overlap_before", gap_start, gap_end)

        return ("miss", None, None)

    def _load_from_cache(
        self,
        ticker: str,
        price_col: str,
        start_date: pd.Timestamp,
        end_date: pd.Timestamp
    ) -> tuple[pd.DataFrame | None, str, tuple | None]:
        """
        Recherche et charge les données disponibles en cache.

        Parcourt les fichiers du répertoire cache pour trouver une
correspondance
        avec le couple (ticker, price_col) et détermine le type de
chevauchement.

        Args:
            ticker:
            price_col: Nom de la colonne prix ('Close', 'Open', etc.)
            start_date: Date de début de la requête
            end_date: Date de fin de la requête

        Returns:
            tuple: (dataframe, status, gap_range)
            - dataframe: Données en cache ou None
            - status: Type de correspondance
            - gap_range: (gap_start, gap_end) si overlap, sinon None
        """
        if not self.cache_dir.exists():
            return (None, "miss", None)

        # Itération sur les fichiers du cache pour match (ticker,
price_col)
        for file_path in self.cache_dir.iterdir():
            if not file_path.is_file() or file_path.suffix != '.pkl':
```

```
                continue

            try:
                # Parse le nom du fichier
                name_parts = file_path.stem.split('_')

                # Vérification du format attendu
                if len(name_parts) < 4:
                    continue

                cached_ticker = name_parts[0]
                cached_col = name_parts[1]
                cached_start_str = name_parts[2]
                cached_end_str = name_parts[3]

                # Vérifier la correspondance ticker + price_col
                if cached_ticker != ticker or cached_col != price_col:
                    continue

                # Parser les dates
                cached_start = pd.to_datetime(cached_start_str)
                cached_end = pd.to_datetime(cached_end_str)

                # Déterminer le type d'overlap
                status, gap_start, gap_end = self._check_date_overlap(
                    cached_start, cached_end, start_date, end_date
                )

                if status != "miss":
                    with open(file_path, 'rb') as f:
                        data = pickle.load(f)

                    # Reconstruire le DataFrame avec les dates
                    prices_list = data['prices']
                    dates_list = data.get('dates') # méthode pandas sur
dataframe

                    df = pd.DataFrame({price_col: prices_list})

                    if dates_list is not None:
                        # Utiliser les dates réelles stockées
                        df.index = pd.to_datetime(dates_list)
                    else:
                        # Fallback: utiliser les jours ouvrés
                        date_range = pd.bdate_range(
                            start=cached_start,
                            periods=len(df)
                        )
                        df.index = date_range

                    if status == "exact":
                        return (df, "exact", None)
                    elif status == "contains":
                        return (df, "contains", None)
```

```
                    elif status.startswith("overlap"):
                        return (df, status, (gap_start, gap_end))

            except (ValueError, KeyError, pickle.UnpicklingError) as e:
                # Ignorer les fichiers cache corrompus
                self.logger.warning(f"Fichier cache corrompu {file_path}:
{e}")
                continue

        return (None, "miss", None)

    def _save_to_cache(
        self,
        cache_path: Path,
        prices: list[float],
        dates: list,
        ticker: str,
        start: str,
        end: str
    ) -> None:
        """
        Sauvegarde les prix dans un fichier cache avec metadata
        """
        data = {
            "ticker": ticker,
            "start": start,
            "end": end,
            "fetched_at": datetime.now().isoformat(),
            "n_prices": len(prices),
            "prices": prices,
            "dates": dates
        }
        with open(cache_path, 'wb') as f:
            pickle.dump(data, f)
        self.logger.debug(f"Cache sauvegardé: {cache_path}")

    def fetch_single_ticker(
        self,
        ticker: str,
        price_col: str,
        dates: tuple[str, str]
    ) -> PriceSeries | None:
        """
        Récupère les données de prix d'un ticker unique avec système de
cache.
        Args:
            ticker: Symbole (ex: 'AAPL')
            price_col: Nom de la colonne prix (ex: 'Close', 'Open')
            dates: (start_date, end_date) au format 'YYYY-MM-DD'

        Returns:
            Instance de PriceSeries ou None si échec
        """
        # Conversion des dates en Timestamp
```

```
        # Vérifier le cache

            # Fetch la partie manquante

            # Fusionner le cache avec les nouvelles données

                # Concaténation à droite

                # Ou concaténation à gauche

            # Supprimer les doublons et trier par date

            # Sauvegarder le cache étendu

            # Retourner l'objet price series

        # else:  # miss: pas de données en cache
            # fetch toutes les données avec yfinance

            # Sauvegarder dans le cache

            # renvoyer PriceSeries avec la série de prix
        pass

    def fetch_multiple_tickers(
        self,
        tickers: Sequence[str],
        price_col: str,
        dates: tuple[str, str]
    ) -> dict[str, PriceSeries]:
        """
        Récupère les données de prix pour plusieurs tickers.

        Returns:
            Dictionnaire {ticker: PriceSeries}
        """
        results = {}
        for ticker in tickers:
            ps = self.fetch_single_ticker(ticker, price_col, dates)
            if ps is not None:
                results[ticker] = ps
        return results

    def clear_cache(self) -> int:
        """
        Supprime tous les fichiers du cache.

        Returns:
            Nombre de fichiers supprimés
        """
        # Itérer sur les fichiers d'un directory tout en vérifiant le
suffix
```

```
            # supprimer
        # Renvoyer le nombre de fichier supprimé
```

Test du système de cache

```
from pyvest.src.dataloader import DataLoader # si vous avez réussi à
installer pyvest avec 'uv pip install -e pyvest/'
# Sinon from pyvest.
import time

loader = DataLoader(cache_dir=".cache")

# Premier call
start = time.perf_counter()
ts1 = loader.fetch_single_ticker("AAPL", "Close", ("2024-01-01", "2024-06-
01"))
premier_temps = time.perf_counter() - start
print(f"Premier fetch: {premier_temps:.2f} secondes")

# deuxieme call depuis le cache
start = time.perf_counter()
ts2 = loader.fetch_single_ticker("AAPL", "Close", ("2024-01-01", "2024-06-
01"))
second_temps = time.perf_counter() - start
print(f"Second fetch: {second_temps:.4f} secondes")

print(f"Accélération: {premier_temps/second_temps:.0f}x plus rapide avec
le cache")
```

Étape 4 : Collections Python

Les listes : collections ordonnées

> Une liste en Python est une séquence mutable d'objets arbitraires, dans le sens ou plusieurs types de données peuvent coexister dans la structure.

Les listes sont la collection de base en Python. Elles maintiennent l'ordre d'insertion, supportent l'indexation et le slicing, et peuvent contenir n'importe quel type d'objet.

Caractéristiques principales :

Propriété	Description
Mutable	On peut ajouter, supprimer ou modifier des éléments
Ordonnée	L'ordre d'insertion est préservé
Indexable	Accès par position `[i]` et slicing `[a:b]`

Propriété	Description
Hétérogène	Peut contenir des types différents

```
# Opérations courantes sur les listes
assets = [apple, microsoft, google, amazon]

# Accès par index
premier = assets[0]           # 1er élément
dernier = assets[-1]          # 2eme élément
sous_liste = assets[1:3]      # indices 1 et 2

# Modification
assets.append(nvidia)
assets.insert(0, tesla)
assets.remove(msft)
element = assets.pop()  # Suppression et renvoie le dernier
assets.pop(1) # supprime + renvoie l'élément qui match l'index

# Itération
for asset in assets:
    print(f"{asset.ticker}: {asset.volatility:.2%}")

# Recherche
if apple in assets:
    print("Apple est dans la liste")
```

Syntaxe de compréhension

> Une compréhension de liste est simplement une autre manière de construire une liste, de façon plus explicite. Son but est toujours de construire une nouvelle liste

Les compréhensions offrent une syntaxe concise pour créer des collections à partir d'itérables

```
[expression for element in iterable if condition]
 ↑           ↑                       ↑
 sortie      boucle                  filtre optionnel
```

Exemples pratiques :

```
# Extraction des tickers
tickers = [asset.ticker for asset in assets]

# Tuples (ticker, rendement)
performance = [(a.ticker, a.total_return) for a in assets]
```

```
# Statement de condition dans la syntaxe
gagnants = [a for a in assets if a.total_return > 0]
```

Expressions génératrices : syntaxe de compréhension comme argument d'une fonction pour produire les éléments `paresseusement` (un à la fois), économisant la mémoire :

```
# Générateur : ne crée pas de liste en mémoire
total_vol = sum(a.volatility for a in assets)
```

Les dictionnaires : mapping clé-valeur

Les dictionnaires offrent une recherche en O(1) (temps d'éxécution constant) par clé.

Caractéristiques principales :

Propriété	Description
Clés uniques	Chaque clé apparaît une seule fois
Clés immuable	Les clés doivent être immutables (str, int, tuple)
Recherche O(1)	Accès quasi-instantané par clé
Ordre préservé	l'ordre d'insertion est garanti

```
# Création par compréhension
assets_par_ticker = {asset.ticker: asset for asset in assets}

# Recherche rapide O(1)
apple = assets_par_ticker["AAPL"]
nvidia = assets_par_ticker.get("NVDA")

# Vérifier l'existence
if "TSLA" in assets_par_ticker:
    tesla = assets_par_ticker["TSLA"]

# Itération sur les paires clé-valeur
for ticker, asset in assets_par_ticker.items():
    print(f"{ticker}: ${asset.current_price:.2f}")

# Itération sur clés
for ticker, asset in assets_par_ticker.keys():
    print(f"{ticker}: ${asset.current_price:.2f}")

# Itération sur valeurs
for ticker, asset in assets_par_ticker.values():
    print(f"{ticker}: ${asset.current_price:.2f}")
```

Les ensembles : collections uniques

> Un ensemble est une collection d'objets uniques

Les ensembles sont parfait pour le test d'appartenance (O(1) vs O(n) pour les listes) et les opérations ensemblistes de manière générale.

```
# Comparaison de portefeuilles comme des ensembles
mon_portefeuille = {"AAPL", "MSFT", "GOOGL"}
benchmark = {"AAPL", "MSFT", "AMZN", "NVDA", "META"}

# Intersection
commun = mon_portefeuille & benchmark

# Différence
manquant = benchmark - mon_portefeuille  # {'AMZN', 'NVDA', 'META'}

# Test d'appartenance rapide O(1)
"AAPL" in mon_portefeuille
```

Partie 2 : Exercices Pratiques

Exercice 1 : La classe Universe (25 min)

Objectif : Créer une classe `Universe` pour gérer une collection d'actifs avec le pattern d'**agrégation**.

```
# Fichier: pyvest/core/universe.py

from pyvest.core.asset import Asset
from typing import Iterator


class Universe:
    """
    Collection d'actifs représentant un univers d'investissement.

    Pattern de conception : AGRÉGATION
    ──────────────────────────────────
    Universe CONTIENT des Asset, mais les Asset peuvent exister
    indépendamment de l'Universe.

    La classe implémente le protocole d'itération (__iter__) et
    de conteneur (__contains__, __len__) pour une utilisation
    pythonique.
    """

    def __init__(self, assets: list[Asset] | None = None) -> None:
```

```
        self._assets: dict[str, Asset] = {}
        if assets:
            for asset in assets:
                self.add(asset)

    def add(self, asset: Asset) -> None:
        """Ajoute un actif à l'univers."""
        # Votre code ici
        pass

    def get(self, ticker: str) -> Asset | None:
        """Récupère un actif par son ticker."""
        # Votre code ici
        pass

    def remove(self, ticker: str) -> Asset | None:
        """Retire un actif de l'univers."""
        # Votre code ici
        pass

    def __len__(self) -> int:
        # Votre code ici
        pass

    def __iter__(self) -> Iterator[Asset]:
        # Votre code ici
        pass

    def __contains__(self, ticker: str) -> bool:
        # Votre code ici
        pass

    @property
    def tickers(self) -> list[str]:
        # Votre code ici
        pass

    def filter_by_sector(self, sector: str) -> list[Asset]:
        """Filtre les actifs par secteur."""
        # Votre code ici
        pass
```

Exercice 2 : Top K Corrélations

Objectif : Implémenter une fonction qui extrait les K paires d'actifs les plus corrélées d'un univers.

> **Note pédagogique** : Cette fonction sera remplacée par une version vectorisée plus performante dans la Session 3 (`Universe.top_correlations()`), mais l'implémentation avec boucles permet de bien comprendre l'algorithme sous-jacent.

```python
from itertools import combinations


def top_k_correlations(
    assets: list[Asset],
    k: int = 20,
    use_absolute: bool = False
) -> list[tuple[str, str, float]]:
    """
    Extrait les K paires les plus corrélées d'une liste d'actifs
    sur la base de la corrélation de Pearson.

    Args:
        assets:
        k: Nombre de paires
        use_absolute: Si True, trie par |corrélation| pour capturer aussi
les fortes corrélations négatives

    Returns:
        Liste de tuples (ticker_1, ticker_2, corrélation) triée
        par corrélation décroissante
    """
    correlations = []

    # itertools.combinations génère toutes les paires uniques
    # évitant les doublons (A,B) et (B,A) et les auto-corrélations (A,A)
    for asset_1, asset_2 in combinations(assets, 2):
        # Calculer la corrélation
        # Votre code ici...
        pass

    # Trier par corrélation (ou valeur absolue) et retourner les k
premières
    # Votre code ici...
    pass
```

Exercice 3 : Construction d'une matrice de corrélation

Objectif : Créer une fonction qui construit une matrice de corrélation complète.

La variance du portefeuille (risque) dépend à la fois des volatilités individuelles ET des corrélations. La matrice de covariance capture la façon dont les actifs évoluent ensemble.

Propriétés de la matrice de corrélation :

- **Symétrique** : corr(A,B) = corr(B,A)
- **Diagonale = 1** : corr(A,A) = 1 pour tout actif A
- **Valeurs dans [-1, 1]** : par définition
- **Semi-définie positive** : toute combinaison linéaire a une variance ≥ 0

```python
import pandas as pd
import numpy as np
from itertools import combinations


def build_correlation_matrix(assets: list[Asset]) -> pd.DataFrame:
    """
    Construit une matrice de corrélation pour tous les actifs.

    Returns:
        DataFrame symétrique avec tickers en index et colonnes
    """
    tickers = [a.ticker for a in assets]
    n = len(tickers)

    # Initialiser la matrice avec NaN
    matrix = np.full((n, n), np.nan)

    # Pre-remplir la diagonale avec 1.0
    np.fill_diagonal(matrix, 1.0)

    # Créer un mapping ticker -> index pour un accès rapide O(1)
    ticker_to_idx = {t: i for i, t in enumerate(tickers)}

    # Remplir le triangle supérieur et inférieur (symétrie)
    # Votre code ici

    return pd.DataFrame(matrix, index=tickers, columns=tickers)


def extract_upper_triangle(corr_matrix: pd.DataFrame) -> pd.DataFrame:
    """
    Extrait les paires uniques du triangle supérieur de la matrice.

    Utile pour éviter les doublons (AAPL-MSFT et MSFT-AAPL) et
    exclure la diagonale (auto-corrélations).

    Cette méthode est similaire à celle utilisée dans le projet
    "Global Multi-Asset Correlation Lab".

    Args:
        corr_matrix: Matrice de corrélation (DataFrame carré)

    Returns:
        DataFrame avec colonnes ['asset_1', 'asset_2', 'correlation']
        trié par corrélation décroissante
    """
    # Créer un masque pour le triangle supérieur (excluant la diagonale
k=1)
    mask = np.triu(np.ones(corr_matrix.shape, dtype=bool), k=1)

    # Votre code ici...
    pass
```

Solutions

Résumé de la Session 2

Concepts

Concept	Application
Composition	Asset POSSÈDE une PriceSeries
Agrégation	Universe CONTIENT des Asset
Décorateur `@property`	Attributs calculés en lecture seule
Décorateurs personnalisés	Chronométrage, logging
Listes	Collections ordonnées et mutables
Dictionnaires	Mapping clé-valeur
Ensembles	Collections uniques, opérations ensemblistes
Compréhensions	Création concise de collections
Expressions génératrices	Évaluation paresseuse
Système de cache	Persistance et optimisation des appels API

Références

1. Lott, S. & Phillips, D. (2021). *Python Object-Oriented Programming*, 4th Edition. Packt.
2. Ramalho, L. (2022). *Fluent Python*, 2nd Edition. O'Reilly.
3. Palomar, D. P. (2025). *Portfolio Optimization: Theory and Application*. Cambridge University Press.
4. Paleologo, G. (2024). *The Elements of Quantitative Investing*. Chapman & Hall/CRC.
5. Documentation yfinance : https://ranaroussi.github.io/yfinance/
6. Diederich, J. (2012). "Stop Writing Classes" - PyCon 2012