Session 1 : Les Fondamentaux de Python pour le Développement

Objectifs de la session

1. Créer une classe Python et comprendre les méthodes spéciales : `__init__`, `__repr__`, `__str__` et `__len__`
2. Comprendre la différence entre une classe et son instance
3. Utiliser les type hints et comprendre les types de données
4. Implémenter des fonctions et méthodes
5. Maîtriser les rudiments de Git

Quelques ressources supplémentaires que vous pouvez utilisez pour comprendre les différents concepts:

- Une intro à git en français : https://pythonds.linogaliana.fr/content/git/
- Un blog sur la CLI: https://dagshub.com/blog/effective-linux-bash-data-scientists/
- Le cours du MIT "Missing semester of your CS education": https://missing.csail.mit.edu/
- Ce site d'exercices en Python: https://www.w3resource.com/python-exercises/
- Un super site avec tutos et exercices: https://www.pythonmorsels.com/articles/
- Le fameux leetcode: https://leetcode.com/

Contexte du projet

Nous allons utiliser un projet de création de package Python comme prétexte pour découvrir les concepts de développement pythoniques essentiels. Cette librairie sera basée sur des concepts classiques de gestion de portefeuille.

Partie 1

Étape 1.0 : Configuration du projet avec Git

Avant de commencer à coder, configurons notre environnement de développement et initialisons notre dépôt Git.

```
# Initialiser le dépôt Git local
git init --initial-branch main

# Configurer votre identité Git
git config --global user.name "VotreNom"
git config --global user.email "votre.email@example.com"

# Vérifier la configuration
git config --global --list
```

```
# Créer le premier commit
git add .
git commit -m "Initial commit: project structure"

# Lier à un dépôt distant après création sur GitHub
git remote add origin https://github.com/username/pyvest.git
git push -u origin main #(ou --set-upstream)
```

Étape 1.1 : L'objectif de cette session

La classe que l'on va construire au cours de cette session :

```
import math

class PriceSeries:
    """
    Représentation d'une série temporelle de prix financiers.

    Attributes:
        values: Liste de prix indexés par le temps
        name: Identifiant de la série

    Class Attributes:
        TRADING_DAYS_PER_YEAR: Constante d'annualisation
        (convention US equities, peut varier selon l'actif)
    """

    TRADING_DAYS_PER_YEAR: int = 252

    def __init__(self, values: list[float], name: str = "unnamed") ->
None:
        self.values = list(values)  # Copie défensive
        self.name = name

    def __repr__(self) -> str:
        return f"PriceSeries({self.name!r}, {len(self.values)} values)"

    def __str__(self) -> str:
        if self.values:
            return f"{self.name}: {self.values[-1]:.2f} (latest)"
        return f"{self.name}: empty"

    def __len__(self) -> int:
        return len(self.values)

    def linear_return(self, t: int) -> float:
        """
        Calcule le rendement linéaire (arithmétique) entre t-1 et t.
```

```
        - Non ajusté des dividendes (utiliser le prix ajusté pour cela)
        - Additif entre actifs : r_portfolio = Σ(weight_i × r_i)

        Args:
            t: Position temporelle (doit être >= 1)

        Returns:
            Rendement en décimal (0.05 = 5%)
        """
        return (self.values[t] - self.values[t-1]) / self.values[t-1]

    def log_return(self, t: int) -> float:
        """
        Calcule le log-rendement entre t-1 et t.

        - Additif dans le temps : Σ(log returns) = log(P_T / P_0)
        - Permet d'approximer la variance multipériode par la somme des
variances

        Args:
            t: index temporel

        Returns:
            Log-rendement
        """
        return math.log(self.values[t] / self.values[t-1])

    @property
    def total_return(self) -> float:
        """
        Rendement total (non annualisé) sur toute la période.
        """
        if len(self.values) < 2:
            return 0.0
        return (self.values[-1] - self.values[0]) / self.values[0]
```

Étape 1.2 : Structure du projet

Comment démarrer rapidement le projet ? Profitez de la puissance de uv et de ses commandes intégrées:

```
uv init pyvest --lib --package
```

Cette commande crée la structure suivante :

```
pyvest/
├── pyproject.toml      # Configuration du projet
├── README.md           # fichier markdown pour présenter un repo git
└── src/                # Dossier qui contient le code source de la
```

```
librairie dans des modules / packages
    └── pyvest/
        ├── __init__.py # Fait de pyvest un package
        └── ...         # Nos modules iront ici
```

Créons ensuite notre environnement virtuel :

```
uv venv -p 3.12 .venv
source .venv/bin/activate
```

Module, Package et Librairie

> **Module** : Un fichier Python (`.py`) contenant du code (fonctions, classes, variables) qui peut être importé par d'autres fichiers.
>
> **Package** : Un répertoire contenant un fichier `__init__.py` et potentiellement d'autres modules ou sous-packages. Le `__init__.py` indique à Python que ce répertoire doit être traité comme un package importable.
>
> **Librairie** : Une collection de code créée par un tiers, composée d'un ou plusieurs packages. L'utilisateur final n'a pas besoin de connaître le code interne ; il utilise simplement l'interface publique (API).
>
> **API (Application Programming Interface)** : L'interface publique d'une application. Elle définit comment d'autres programmes peuvent communiquer avec votre code (appeler des fonctions, instancier des classes, etc.).

Étape 1.3 : Création de la première classe

On peut créer notre premier fichier : (attention toutes les commandes sont en bash, demandez à votre LLM préféré pour les commandes en powershell) (Surtout essayez de comprendre les commandes et pas juste bêtement copier/coller)

```
touch src/pyvest/priceseries.py
```

```
# src/pyvest/priceseries.py

class PriceSeries:
    pass
```

On peut tester dans le REPL (read-eval-print-loop) :

```
>>> from pyvest.priceseries import PriceSeries
>>> ps = PriceSeries()
>>> ps
<pyvest.priceseries.PriceSeries object at 0x...>
>>> type(ps)
<class 'pyvest.priceseries.PriceSeries'>
```

Qu'est-ce qu'une classe ?

Une **classe** est un patron / template qui définit la structure et le comportement d'un type d'objet. On peut la concevoir comme une usine capable de fabriquer des objets selon un modèle défini.

Les objets créés à partir d'une classe sont appelés **instances**. Chaque instance possède :

- Des **attributs** : variables propres à l'instance
- Des **méthodes** : fonctions ayant accès aux attributs de l'instance

```
# PriceSeries est la classe
# ps1 et ps2 sont des instances (les objets créés)
ps1 = PriceSeries([100, 105, 110], "close")
ps2 = PriceSeries([50, 52, 51], "close")

# Chaque instance a ses propres données
ps1.name
ps2.name
```

> **Note sur `pass`** : C'est une instruction qui ne fait rien. Elle permet juste de définir des structures vides (classes, fonctions) sans casser le code définit après.

Expression vs Statement (Instruction)

Petite distinction importante.

Une **expression** est une combinaison de valeurs, variables et opérateurs qui s'évalue pour produire une valeur :

```
# Expressions — chacune produit une valeur
3 + 4              # → 7
x * 2              # → valeur numérique
len(my_list)       # → int
a > b              # → booléen
"hello".upper()    # → "HELLO"
```

Un **statement** (instruction) est une unité de code qui effectue une action mais ne produit pas de valeur utilisable :

```
# Statements — effectuent des actions
if condition:      # Contrôle de flux
    pass
for x in items:    # Boucle
    pass
def ma_fonction(): # Définition de fonction
    pass
return valeur      # Retour de fonction (donc contrôle de flux)
import math        # Import
x = 5              # Assignation
```

Cas particulier : L'opérateur morse `:=` (walrus operator) permet une assignation qui est aussi une expression :

```
# Assignation classique (statement)
n = len(data)
if n > 10:
    print(n)

# Avec walrus operator (statement + expression)
if (n := len(data)) > 10:
    print(n)
```

Étape 1.4 : La méthode `__init__`

La méthode `__init__` est l'**initialiseur** de la classe. Elle est appelée automatiquement après la création de l'instance pour initialiser ses attributs.

> **Note (peu importante)** : On appelle souvent `__init__` le "constructeur" par abus de langage mais le constructeur en Python est `__new__`, qui crée l'instance. `__init__` reçoit cette instance déjà créée et l'initialise.

```
class PriceSeries:

    TRADING_DAYS_PER_YEAR: int = 252  # Attribut de classe

    def __init__(self, values: list[float], name: str = "unnamed") ->
None:
        self.values = values  # Attribut d'instance
        self.name = name      # Attribut d'instance
```

Le paramètre `self` est une référence à l'instance spécifique en cours de création. Python le passe automatiquement lors de l'appel.

```
# Ce que vous écrivez :
ps = PriceSeries([100, 105], "close")

# Ce que Python exécute en coulisses (approximativement) :
type.__call__(PriceSeries, [[100, 105], "close"])
# 1. Dans __call__, création de l'instance via __new__
instance = cls.__new__(PriceSeries)
# 2. Initialisation via __init__
PriceSeries.__init__(instance, [100, 105], "close")
# 3. Assignation à la variable
ps = instance
```

Test dans le REPL :

```
>>> ps = PriceSeries([100.0, 102.5, 101.0, 105.0], "close")
>>> ps.values
[100.0, 102.5, 101.0, 105.0]
>>> ps.name
'close'
>>> ps.TRADING_DAYS_PER_YEAR  # Accessible via l'instance
252
>>> PriceSeries.TRADING_DAYS_PER_YEAR  # Ou via la classe
252
```

Attributs de classe vs Attributs d'instance

Type	Définition	Partage	Exemple
Attribut de classe	Défini directement dans la classe	Partagé par toutes les instances	`TRADING_DAYS_PER_YEAR = 252`
Attribut d'instance	Défini via `self.xxx` dans `__init__`	Propre à chaque instance	`self.values = values`

Type Hints (Annotations de type)

Les **type hints** sont des annotations qui indiquent le type attendu des variables, paramètres et valeurs de retour :

```
def __init__(self, values: list[float], name: str = "unnamed") -> None:
#                  ↑                ↑              ↑              ↑
#            paramètre      type attendu    valeur défaut   type retour
```

Points importants :

1. **Pas d'effet à l'exécution** : Python n'applique pas les types au runtime. Ce code fonctionne même si on passe des types incorrects.
2. **Vérification statique** : Les outils comme `mypy`, `pyright` ou l'IDE (VS Code/PyCharm) analysent les types avant l'exécution.
3. **Documentation vivante** : Les types rendent le code plus lisible et auto-documenté.

```
# Ce code s'exécute sans erreur malgré les types incorrects
ps = PriceSeries("pas une liste", 12345)
# Mais un type checker signalerait le problème
```

Principe de Postel (utile pour le typage) :

> "Soyez conservateur dans ce que vous envoyez, libéral dans ce que vous acceptez."

Types courants :

```
# Types simples
x: int = 5
y: float = 3.14
s: str = "hello"
b: bool = True

# Types composés
numbers: list[float] = [1.0, 2.0]
mapping: dict[str, int] = {"a": 1}
optional: str | None = None
```

Étape 1.5 : Les méthodes `__repr__` et `__str__`

Ces méthodes spéciales définissent comment votre objet est représenté sous forme de chaîne de caractères.

```
class PriceSeries:
    # ... (code précédent)

    def __repr__(self) -> str:
        """Représentation pour les développeurs (debugging)."""
        return f"PriceSeries({self.name!r}, {len(self.values)} values)"

    def __str__(self) -> str:
        """Représentation pour les utilisateurs."""
        if self.values:
            return f"{self.name}: {self.values[-1]:.2f} (latest)"
        return f"{self.name}: empty"
```

Différence clé :

Méthode	Appelée par	Usage	Format
`__repr__`	`repr(obj)`, REPL, debugger	Développement	Doit ressembler au code pour recréer l'objet
`__str__`	`str(obj)`, `print(obj)`	Utilisateur final	Lisible et claire

```
>>> ps = PriceSeries([100.0, 102.5, 105.0], "adjusted")
>>> ps                    # REPL appelle __repr__
>>> print(ps)             # print() appelle __str__
>>> repr(ps)
>>> str(ps)
```

> **Astuce `!r` dans les f-strings** : L'expression `{self.name!r}` applique `repr()` à la valeur, affichant les guillemets pour les strings. Cela aide à distinguer `PriceSeries('AAPL', ...)` de `PriceSeries(AAPL, ...)`.

Le Python Data Model et les méthodes spéciales

Le Python Data Model c'est l'API qu'on utilise pour que nos objets customisés intéragissent avec les fonctionnalités pré-intégrées à Python.

Les **méthodes spéciales** (ou "dunder methods" pour "double underscore") permettent à vos objets de s'intégrer naturellement avec les opérations Python :

```
# Ce que vous écrivez          # Ce que Python appelle
len(obj)                       obj.__len__()
str(obj)                       obj.__str__()
repr(obj)                      obj.__repr__()
obj[key]                       obj.__getitem__(key)
obj1 + obj2                    obj1.__add__(obj2)
for x in obj:                  obj.__iter__()
```

Règle importante : Vous n'appelez généralement pas ces méthodes directement. C'est l'interpréteur Python qui les appelle pour vous. Écrivez `len(ts)`, pas `ts.__len__()`.

Étape 1.6 : Premier commit Git

Sauvegardons notre progression :

```
# Voir les fichiers modifiés
git status
```

```
# Ajouter les fichiers
git add src/pyvest/priceseries.py

# Créer un commit avec un message descriptif
git commit -m "feat: add PriceSeries class with __init__, __repr__,
__str__"

# Vérifier l'historique
git log --oneline
```

Convention de commit : Utilisez des messages clairs, par exemple suivant le format : `type: description`

- `feat`: nouvelle fonctionnalité
- `fix`: correction de bug
- `docs`: documentation
- `refactor`: refactorisation sans changement fonctionnel

Étape 1.7 : Implémentation des méthodes de calcul de rendement

```python
import math

class PriceSeries:
    # ... (code précédent)

    def linear_return(self, t: int) -> float:
        """Rendement linéaire (arithmétique) entre t-1 et t."""
        return (self.values[t] - self.values[t-1]) / self.values[t-1]

    def log_return(self, t: int) -> float:
        """Log-rendement entre t-1 et t."""
        return math.log(self.values[t] / self.values[t-1])
```

Test dans le REPL :

```
>>> ps = PriceSeries([100.0, 105.0, 103.0, 110.0], "TEST")
>>> ps.linear_return(1)  # (105 - 100) / 100
>>> ps.log_return(1)

# On peut vérifier l'additivité des log-rendements
>>> sum(ps.log_return(t) for t in range(1, len(ps.values)))
>>> math.log(110 / 100)
```

Propriétés des rendements :

Type	Additivité	Usage principal

Type	Additivité	Usage principal
Linéaire (arithmétique)	Entre actifs : `r_p = Σ(w_i × r_i)`	Portefeuille, cross-section
Logarithmique	Dans le temps : `r_total = Σ(r_t)`	Série temporelle, volatilité

Les fonctions en Python

Une **fonction** est un bloc de code réutilisable qui effectue une tâche spécifique.

```
def nom_de_fonction(param1: type1, param2: type2 = valeur_defaut) ->
type_retour:
    """Docstring décrivant la fonction."""
    # Corps de la fonction
    resultat = param1 + param2
    return resultat #optionnel
```

Composants :

- `def` : mot-clé introduisant la définition
- `nom_de_fonction` : identifiant (convention : `snake_case`, à la différence du `camelCase` par exemple)
- `param1, param2` : paramètres recevant des valeurs à l'appel
- `-> type_retour` : annotation du type de retour
- `return` : instruction optionnelle renvoyant une valeur

Appel de fonction :

```
# Appel positionnel
resultat = nom_de_fonction(10, 20)

# Appel avec arguments nommés (ordre modifiable)
resultat = nom_de_fonction(param2=20, param1=10)

# Mix des deux (positionnels d'abord)
resultat = nom_de_fonction(10, param2=20)
```

Portée des variables (Scope)

Python résout les noms de variables selon la règle **LEGB** :

```
L - Local      : Variables définies dans la fonction courante
E - Enclosing  : Variables des fonctions englobantes (closures)
G - Global     : Variables au niveau du module
B - Built-in   : Fonctions et constantes intégrées (len, print, True, ...)
```

```python
# GLOBAL
x = "global"

def externe():
    # ENCLOSING (pour interne)
    y = "enclosing"

    def interne():
        # LOCAL
        z = "local"
        print(z)  # → "local"
        print(y)  # → "enclosing" (trouvé dans Enclosing)
        print(x)  # → "global" (trouvé dans Global)
        print(len)  # → <built-in function len> (Built-in)

    interne()

externe()
```

Piège classique: Modifier une variable globale :

```python
compteur = 0

def incrementer():
    compteur = compteur + 1  # UnboundLocalError
    # Python crée une variable locale 'compteur' qui n'est pas encore
définie

def incrementer_correct():
    global compteur  # Déclare qu'on utilise la variable globale
    compteur = compteur + 1
```

Différence entre fonction Python et fonction mathématique

Aspect	Fonction mathématique	Fonction Python
Mapping	Un élément de A → un unique élément de B	Peut ne rien retourner (None)
Déterminisme	Même input → même output (toujours)	Peut dépendre d'état externe
Effets de bord	Aucun	Peut modifier l'environnement

Étape 1.8 : La méthode `__len__`

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

```
    def __len__(self) -> int:
        return len(self.values)
```

Cette méthode permet à nos instances de fonctionner avec la fonction built-in `len()` :

```
>>> ps = PriceSeries([100.0, 105.0, 103.0, 110.0], "TEST")
>>> len(ps)
4
>>> empty = PriceSeries([], "EMPTY")
>>> len(empty)
```

> **Duck Typing en action** : Python ne vérifie pas le type ; il vérifie la présence de la méthode. Tout objet avec `__len__` fonctionne avec `len()`. Pas besoin de type particulier.

Concept fondamental : Duck Typing

> "If it walks like a duck and quacks like a duck, it's a duck."

Le **duck typing** signifie que Python se soucie des **comportements** (méthodes/attributs) plutôt que des types. Un objet est compatible avec une opération s'il implémente les méthodes nécessaires.

```
def afficher_longueur(obj):
    """Fonctionne avec TOUT objet ayant __len__."""
    print(f"Longueur: {len(obj)}")

# Tous ces appels fonctionnent !
afficher_longueur([1, 2, 3])           # list
afficher_longueur("hello")              # str
afficher_longueur({"a": 1, "b": 2})     # dict
afficher_longueur(PriceSeries([100, 101, 102], "X"))  # notre classe !
```

C'est pourquoi implémenter `__len__` permet à `PriceSeries` de s'intégrer naturellement avec tout code utilisant `len()`.

Étape 1.9 : Le décorateur `@property`

```
class PriceSeries:
    # ... (code précédent)

    @property
    def total_return(self) -> float:
        """Rendement total sur toute la période."""
        if len(self.values) < 2:
            return 0.0
        return (self.values[-1] - self.values[0]) / self.values[0]
```

Le décorateur `@property` transforme une méthode en **attribut calculé** :

```
>>> ps = PriceSeries([100.0, 105.0, 103.0, 110.0], "TEST")
>>> ps.total_return  # Pas de parenthèses car la méthode est accessible
comme un attribut
>>> f"Rendement total: {ps.total_return:.2%}"
'Rendement total: 10.00%'
```

Avantages de `@property` :

1. **Syntaxe propre** : `ps.total_return` au lieu de `ps.total_return()`
2. **Encapsulation** : Le calcul est caché derrière une interface simple
3. **Lazy evaluation** : Calculé uniquement quand on y accède

Quand utiliser `@property` :

- Valeur dérivée d'autres attributs
- Pas d'effets de bord sur son environnement
- Calcul relativement léger

On verra plus en détails le concept de décorateur et son fonctionnement un peu plus tard dans le cours.

Les variables ne sont pas des boîtes

Il faut penser les variables comme des étiquettes et non des boîtes.

```
# On pourrait croire que b contient une COPIE de a
# En réalité, a et b sont des étiquettes pointant vers le MÊME objet

a = [1, 2, 3]
b = a           # b est une autre étiquette sur le même objet
a.append(4)
print(b)        # [1, 2, 3, 4] - b voit le changement !
```

Concept fondamental : Objets mutables vs immutables

Type	Mutabilité	Exemples
Immutable	Ne peut pas être modifié après création	`int`, `float`, `str`, `tuple`, `frozenset`
Mutable	Peut être modifié après création	`list`, `dict`, `set`, objets personnalisés

```
# Immutable: l'objet original n'est pas modifié
x = 5
```

```
y = x
x = x + 1  # Crée un NOUVEL objet int
print(y)   # 5 car y pointe toujours vers l'ancien objet car immutable

# Mutable: l'objet original EST modifié
# Voir l'exemple précédent de la liste pour démontré
# qu'une variable n'est pas une boîte
```

Implication pour `PriceSeries` :

```
# Danger potentiel
prices = [100, 105, 110]
ps = PriceSeries(prices, "TEST")
prices.append(115)  # Modifie aussi ps.values
print(ps.values)    # [100, 105, 110, 115]

# Solution : copie défensive dans __init__
def __init__(self, values: list[float], name: str = "unnamed") -> None:
    self.values = list(values)  # Crée une COPIE de la liste
    self.name = name
```

Disinction d'opérateur: `is` vs `==`

Opérateur	Compare	Question posée
`==`	Les **valeurs**	"Est ce que les variables pointent vers des objets avec la même valeur ?"
`is`	L'**identité** (adresse mémoire)	"Sont-ils le même objet en mémoire ?"

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a

a == b   # True  - mêmes valeurs
a is b   # False - objets différents
a == c   # True  - mêmes valeurs
a is c   # True  - même objet

# Vérification avec id()
id(a)    # 140234567890
id(b)    # 140234567891 - différent !
id(c)    # 140234567890 - identique à a
```

Slicing (découpage)

Le slicing permet d'extraire des sous-séquences avec la syntaxe `sequence[start:stop:step]` :

```
prices = [100, 105, 103, 110, 108, 112]
#          0    1    2    3    4    5   (indices positifs)
#         -6   -5   -4   -3   -2   -1   (indices négatifs)

# Quelques exemples d'extraction de base
prices[0]       # 100 - premier élément
prices[-1]      # 112 - dernier élément
prices[1:4]     # [105, 103, 110] - indices 1, 2, 3
prices[:3]      # [100, 105, 103] - du début à l'indice 2
prices[3:]      # [110, 108, 112] - de l'indice 3 à la fin
prices[::2]     # [100, 103, 108] - un élément sur deux
prices[::-1]    # [112, 108, 110, 103, 105, 100] - inversé

# Utilisé dans notre code
peak = max(self.values[:t+1])  # Maximum du début jusqu'à t (inclus)
last_price = self.values[-1]   # Dernier prix
```

Gestion des erreurs (Exceptions)

Les **exceptions** signalent des conditions anormales. Sans gestion, elles arrêtent le programme.

```
# peut planter
def linear_return(self, t: int) -> float:
    return (self.values[t] - self.values[t-1]) / self.values[t-1]
    # si t=0 ? → values[-1] donc calcul faussé
    # si t=100 et len(values)=50 on aura une IndexError

# robuste avec validation
def linear_return(self, t: int) -> float:
    if t < 1:
        raise ValueError(f"t doit être >= 1, reçu: {t}")
    if t >= len(self.values):
        raise IndexError(f"t={t} hors limites (max:
{len(self.values)-1})")
    return (self.values[t] - self.values[t-1]) / self.values[t-1]
```

Gestion avec `try/except` :

```
try:
    ret = ts.linear_return(0)
except ValueError as e:
    print(f"Erreur de valeur: {e}")
except IndexError as e:
    print(f"Index hors limites: {e}")
```

On peut également conclure un statement `try/except` par un `finally` pour exécuter une instruction dans tous les cas, qu'une exception soit catch ou pas.

Exceptions courantes :

Exception	Cause typique
`ValueError`	Valeur incorrecte mais type correct
`TypeError`	Type incorrect
`IndexError`	Index hors limites d'une séquence
`KeyError`	Clé absente d'un dictionnaire
`ZeroDivisionError`	Division par zéro
`FileNotFoundError`	Fichier introuvable

Étape 1.10 : Deuxième commit

```
git add .
git commit -m "feat(core): add return calculations and __len__ to
PriceSeries"
```

Partie 2 : Exercices Pratiques

Exercice 1 : Calcul du vecteur de rendements

Implémentez deux méthodes retournant la liste de tous les rendements :

```python
def get_all_linear_returns(self) -> list[float]:
    """Retourne la liste de tous les rendements linéaires.

    Returns:
        Liste de n-1 rendements pour n prix.
    """
    # Votre code ici
    pass

def get_all_log_returns(self) -> list[float]:
    """Retourne la liste de tous les log-rendements."""
    # Votre code ici
    pass
```

Exercice 2 : Volatilité annualisée

```python
def annualized_volatility(self) -> float:
    """
    Volatilité annualisée à partir des log-rendements.

    Formule: σ_annual = σ_daily × √252

    Note: Le scaling √252 suppose des rendements i.i.d
    Cette hypothèse est rarement vérifiée en pratique
    (clustering de volatilité).

    Pour une meilleure estimation, considérer:
    - Modèles GARCH
    - Moyenne mobile exponentielle (EWMA)
    """
    # Étapes:
    # 1. Obtenir tous les log-rendements
    # 2. Calculer la moyenne
    # 3. Calculer la variance d'échantillon (diviser par n-1)
    # 4. Prendre la racine carrée pour la volatilité quotidienne
    # 5. Annualiser: × √252
    pass
```

Exercice 3 : Ratio de Sharpe

```python
def annualized_return(self) -> float:
    """
    Rendement annuel moyen à partir des log-rendements.
    Formule: μ_annual = μ_daily × 252
    """
    pass

def sharpe_ratio(self, risk_free_rate: float = 0.0) -> float:
    """
    Ratio de sharpe annualisé :
            - ratio entre les rendements esperés d'une stratégie et sa
volatilité
            - rendement par unité de risque

        Formule: SR = (μ - r_f) / σ

        Args:
            risk_free_rate: taux sans risque annuel
    """
    pass
```

Exercice 4 : Max drawdown

```
    def drawdown_at(self, t: int) -> float:
        """
        Retourne le drawdown à l'instant t depuis le début de la série.
        Mesure le déclin par rapport à un pique historique.

        Args:
            t (int): index de position de la valeur supérieure de l'intervalle
    considéré.

        Returns:
            float: drawdown
        """
        pass

    def max_drawdown(self) -> float:
        """
        Drawdown maximum sur toute la série.

        Returns:
            Drawdown maximum (valeur négative ou zéro)
        """
        pass
```

Commit final

```
git add .
git commit -m "features: add volatility, sharpe, drawdown to PriceSeries"
```

Solutions

Solution - Exercice 1

```
    def get_all_linear_returns(self) -> list[float]:
        """Retourne la liste de tous les rendements linéaires."""
        return [self.linear_return(t) for t in range(1, len(self.values))]

    def get_all_log_returns(self) -> list[float]:
        """Retourne la liste de tous les log-rendements."""
        return [self.log_return(t) for t in range(1, len(self.values))]
```

Solution - Exercice 2

```
    def annualized_volatility(self) -> float:
        if len(self.values) < 3:
```

```
            raise ValueError("Not enough data points")

    log_returns = self.get_all_log_returns()
    n = len(log_returns)
    mean = sum(log_returns) / n
    var = sum((l_r - mean)**2 for l_r in log_returns) / (n - 1)
    daily_vol = math.sqrt(var)

    return daily_vol * math.sqrt(self.TRADING_DAYS_PER_YEAR)
```

Solution - Exercice 3

```
def annualized_return(self) -> float:
    """
    Retourne le rendement annualisé sur toute la période.
    """
    if len(self) < 2:
        raise ValueError("Not enough data points")
    r = self.get_all_log_returns()
    return (sum(r) / len(r)) * self.TRADING_DAYS_PER_YEAR

def sharpe_ratio(self, risk_free_rate: float = 0.0) -> float:
    """
    Ratio de sharpe annualisé :
        - ratio entre les rendements esperés d'une stratégie et sa vol
        - rendement par unité de risque

    Formule: SR = (μ - r_f) / σ

    Args:
        risk_free_rate: taux sans risque annuel

    """
    vol = self.annualized_volatility()
    if vol == 0:
        return 0.0
    excess_return = self.annualized_return() - risk_free_rate
    return excess_return / vol
```

Solution - Exercice 4

```
def drawdown_at(self, t: int) -> float:
    """Drawdown au temps t."""
    if t < 0 or t >= len(self.values):
        raise IndexError(f"index {t} is out of range for series of length
{len(self.values)}")
```

```
        peak = max(self.values[:t+1])
        if peak == 0:
            return 0.0
        return (self.values[t] - peak) / peak

    def max_drawdown(self) -> float:
        """Drawdown maximum sur toute la série."""
        max_dd = 0.0
        peak = self.values[0]

        for value in self.values[1:]:
            peak = max(peak, value)
            if peak > 0:
                dd = (value - peak) / peak
                max_dd = min(max_dd, dd)

        return max_dd
```