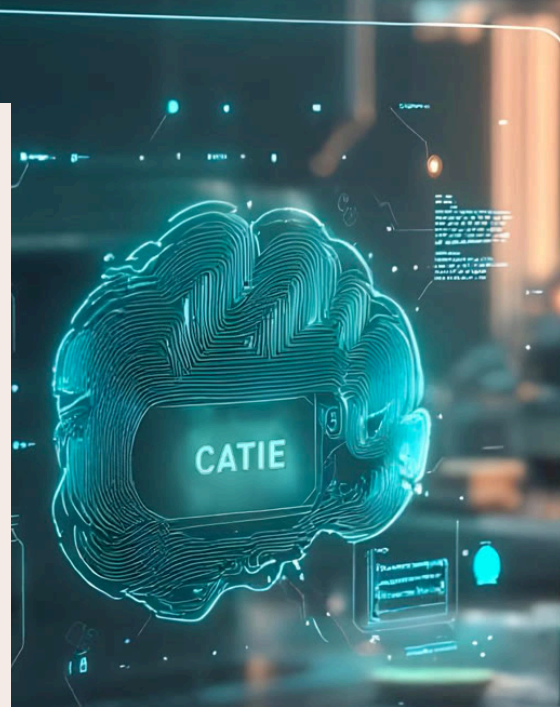




CATIE

Outils d'imagerie pour la
robotique
Python moderne

06/10/2026



Plan

1. Python en quelques slides
2. Typer et documenter
3. TP : découvrir le fil rouge
4. Structurer ses données et son code
5. Erreurs et logging
6. Trois outils du langage : **with**, générateurs, décorateurs
7. TP : fil rouge, étape 1

Bases : rappels du langage, sautés si tout le monde les connaît. **Pour aller plus loin** : détails pour ceux qui sont déjà à l'aise. Les deux se sautent sans perdre le fil.

Crédit : cours en partie inspiré de « Python Advanced » de Yoan Mollard, sous licence CC BY-NC-SA 4.0.



1. Python en quelques slides

Rappels du langage pour ceux qui n'ont pas encore pratiqué Python. Déjà à l'aise : on passe directement à la suite.

Écrire et lancer un programme

Bases

Un programme Python est un simple fichier texte `.py`. Pas de compilation comme en C : un **interpréteur** le lit et l'exécute directement, ligne par ligne, de haut en bas.

```
# hello.py (after a #, it's a comment)
name = "robot"
print("Hello", name)
$ uv run hello.py
Hello robot
```

- Le **terminal** : la fenêtre où l'on tape des commandes
- `uv run hello.py` lance le script ; uv installe Python s'il le faut
- `print(...)` affiche du texte
- Pas de `main()` obligatoire, pas de `gcc` : on lance, c'est tout

Un script Python = un fichier `.py`, lancé avec `uv run file.py`.

Variables et types

Bases

Une **variable** est une étiquette posée sur une valeur. En C, on déclare le type : `int width = 320;`. En Python, on écrit juste `width = 320` : le type vient de la valeur.

```
width = 320           # int: integer
factor = 1.2          # float: decimal number
name = "image_000.png" # str: text (a string)
color = True          # bool: True or False
file = None           # None: "nothing"
```

```
print(f"{name}: {width} px")
# prints: image_000.png: 320 px
```

- `f"...{x}..."` (une **f-string**) insère la valeur de `x` dans le texte
- `type(width)` donne `int`
- Une variable peut changer de type : `width = "large"` passe sans erreur... source de bugs, on y revient avec le typage

`name = value` crée une variable ; le type (`int`, `float`, `str`, `bool`) est porté par la valeur.

Listes et dictionnaires

Bases

Pour ranger plusieurs valeurs : la **liste** garde un ordre (comme un tableau C, mais de taille variable) ; le **dictionnaire** associe une clé à une valeur (comme un annuaire : nom $\rightarrow$ numéro).

<code>files = ["a.png", "b.png"]</code>	<code>size = {"width": 320, "height": 240}</code>
<code>files.append("c.png")</code> # add at the end	<code>size["width"]</code> # 320
<code>files[0]</code> # "a.png": index 0	<code>size["channels"] = 3</code> # add a key
<code>len(files)</code> # 3 elements	<code>position = (10, 20)</code> # a tuple

Un **tuple** (10, 20) est une liste figée, qu'on ne peut plus modifier : pratique pour une position ou une taille.

[...] : une liste, on y accède par position. {key: value} : un dictionnaire, on y accède par clé.

Conditions et boucles : l'indentation compte

Bases

En C, les accolades { } délimitent les blocs. En Python, un bloc commence par : et se reconnaît à son **indentation** : le décalage de 4 espaces en début de ligne.

```
files = ["a.png", "notes.txt", "b.png"]

for f in files:                # for each element
    if f.endswith(".png"):      # ends with .png?
        print("image:", f)
    else:
        print("skipped:", f)
print("done")                  # outside the loop
```

- `for x in items:` parcourt les éléments un par un, sans indice
- `for i in range(5):` compte de 0 à 4
- `if / elif / else`, avec `and`, `or`, `not`
- Indentation incohérente : `IndentationError`

Pas d'accolades : c'est le décalage de 4 espaces qui dit ce qui est dans la boucle ou dans le `if`.

Les fonctions

Bases

Une **fonction** donne un nom à un calcul pour le réutiliser : mot-clé **def**, paramètres entre parenthèses, **return** pour renvoyer le résultat.

```
def brighten(pixel, factor=1.2):  
    """Multiply a pixel, capped at 255."""  
    return min(round(pixel * factor), 255)  
  
brighten(100)           # 120 (factor is 1.2)  
brighten(250)           # 255: saturated  
brighten(100, factor=0.5) # 50
```

- **factor=1.2** : valeur par défaut, l'argument devient facultatif
- **factor=0.5** à l'appel : argument **nommé**, plus lisible
- Le texte entre `"""` est la **docstring** : la documentation de la fonction
- Sans **return**, une fonction renvoie **None**

```
def name(parameters): définit une fonction, return renvoie son résultat.
```

Importer du code

Bases

On ne réécrit pas tout. Un **module** est un fichier `.py` dont on réutilise le contenu ; une **bibliothèque** est un ensemble de modules prêts à l'emploi.

```
import time                # standard library
from pathlib import Path    # import a single name
import numpy as np          # third-party, aliased np
from config import Config   # our config.py file

start = time.perf_counter() # module.function
folder = Path("data")
```

- **Bibliothèque standard** : livrée avec Python (time, pathlib, logging...)
- **Bibliothèques externes** (numpy, pillow) : installées par uv, d'après la liste des **dépendances** du projet
- Un projet = plusieurs fichiers qui s'importent entre eux

`import` réutilise le code d'un autre fichier ou d'une bibliothèque.

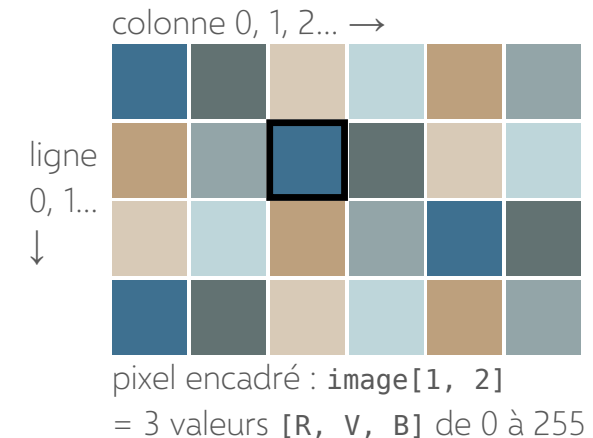
Une image en Python : un tableau NumPy

Une image couleur est un `numpy.ndarray` de forme (`hauteur`, `largeur`, 3) et de type `uint8` : une valeur de 0 à 255 par canal (R, V, B). Tout le fil rouge manipule ces tableaux.

```
import numpy as np
from PIL import Image      # Pillow

image = np.asarray(Image.open("data/image_000.png"))
image.shape, image.dtype   # (240, 320, 3), uint8
image[1, 2]                # pixel (1, 2): [R, G, B]
image[..., 0]              # red channel: (240, 320)
image[10:-10, 10:-10]      # 10 px less on each side
```

- Indices [`ligne`, `colonne`, `canal`] depuis 0 ; `a:b` exclut `b`, un indice négatif part de la fin
- Une tranche est une **vue** : pas de copie, elle partage la mémoire de l'image



« Python moderne » : ce qui a changé

Une version par an, en octobre. Une bonne part des tutoriels, des réponses en ligne et du code généré date d'avant 3.10 : vérifiez la date. Le module impose **Python ≥ 3.12**.

Versions récentes

- 3.10 : **match**, unions **X | Y** dans les types
- 3.11 : de 10 à 60 % plus rapide
- 3.12 : génériques **def f[T](...)**, mot-clé **type**
- 3.13 : build sans GIL, expérimental
- 3.14 : build sans GIL officiellement supporté

Le GIL est le verrou global qui empêche deux threads d'exécuter du Python en même temps (séance 3).

Les PEP

Python Enhancement Proposals : les documents qui spécifient le langage et ses conventions.

- PEP 8 : style de code
- PEP 257 : docstrings
- PEP 484, 604, 695 : typage
- PEP 723 : dépendances en tête de script (utilisé au TP)



2. Typage et documentation

Annoter les signatures, et faire vérifier leur cohérence par un outil avant l'exécution.

Annotations de type

Python est typé dynamiquement : `area("320", 240)` s'exécute et renvoie la chaîne "320" répétée 240 fois. Les **annotations** déclarent les types attendus. Elles ne sont **pas vérifiées à l'exécution** : ce sont des outils qui les exploitent.

```
def area(width: int, height: int) -> int:
    return width * height

def find(name: str, folder: Path) -> Path | None:
    path = folder / name
    return path if path.exists() else None

sizes: dict[str, tuple[int, int]] = {}
```

- Collections paramétrées : `list[str]`, `dict[str, int]`, `tuple[int, int]`
- `X | Y` : union ; `Path | None` : un chemin, ou rien
- On annote les **signatures** ; les variables locales sont inférées
- Syntaxe moderne uniquement : ni `List`, ni `Optional`, ni `Union`

Vérifier les types avec ty

Un vérificateur **statique** analyse le code sans l'exécuter. On utilise **ty** (Astral, mêmes auteurs que uv et ruff, en bêta, très rapide) ; **mypy** et **pyright** (moteur de Pylance dans VS Code) sont les références établies.

```
def mean(values: list[float]) -> float:  
    return sum(values) / len(values)
```

```
mean("12")
```

```
uvx ty check .
```

uvx exécute un outil dans un environnement temporaire, sans l'ajouter au projet.

```
error[invalid-argument-type]: Argument to  
    function `mean` is incorrect  
--> t.py:4:6  
4 | mean("12")  
  |      ^^^^ Expected `list[float]`,  
  |              found `Literal["12"]`
```

Dans le projet, toutes les fonctions sont annotées et **ty check** passe sans erreur.

Vérifier le style avec ruff

Un **linter** repère sans exécuter le code les erreurs probables et les mauvaises pratiques ; un **formateur** fixe la mise en page. **ruff** (Astral, comme uv et ty) fait les deux, en quelques millisecondes.

```
uvx ruff check .          # lint
uvx ruff check --fix .    # fix what can be fixed
uvx ruff format .         # layout
uvx ruff rule E722        # explain a rule
```

- Chaque règle a un code : E style, F erreurs, I imports, SIM simplifications...
- Configuration dans `pyproject.toml` et passage en CI : séance 4

Sur `depart/pipeline.py` :

```
pipeline.py:7:1: I001 Import block is
    un-sorted or un-formatted
pipeline.py:55:5: E722 Do not use bare
    `except`
pipeline.py:74:7: SIM115 Use a context
    manager for opening files
```

Dans le projet, `ruff check` et `ruff format --check` passent sans erreur.

Typage : fonctions, génériques, ancienne syntaxe

Une fonction qui prend une fonction ou une collection en paramètre se type avec `collections.abc` ; le code trouvé en ligne utilise souvent l'ancienne syntaxe de `typing`.

```
from collections.abc import Callable, Iterable
```

```
# type alias (3.12)
```

```
type Transform = Callable[[bytes], bytes]
```

```
def apply(files: Iterable[Path],  
          f: Transform) -> list[bytes]:  
    return [f(p.read_bytes()) for p in files]
```

```
# generic function (3.12)
```

```
def first[T](items: list[T]) -> T:  
    return items[0]
```

- En paramètre, le type le plus général utile (`Iterable`, `Sequence`) ; en retour, le plus précis
- `Callable[[A], B]` : une fonction de A vers B
- À remplacer : `List[int] → list[int]`,
`Optional[str] → str | None`, `Union[a, b] → a | b`, `TypeVar → def f[T]`
- `ruff` (séance 4, règles `UP`) fait ces réécritures automatiquement

Docstrings

La signature dit **quoi** passer ; la docstring dit **ce que fait** la fonction. Elle est lue par l'éditeur (au survol), par `help()` et par les générateurs de documentation (pdoc, Sphinx, makedocstrings).

```
def area(width: int, height: int) -> int:
    """Number of pixels in an image.

    Args:
        width: width in pixels.
        height: height in pixels.

    Raises:
        ValueError: if a dimension is negative.
    """
    if width < 0 or height < 0:
        raise ValueError("negative dimension")
    return width * height
```

- Première ligne : une phrase qui tient seule
- Puis **Args**, **Returns**, **Raises** (format Google) ; les types sont déjà dans la signature, inutile de les répéter
- Les commentaires **#** expliquent le **pourquoi** d'un choix, pas ce que fait la ligne

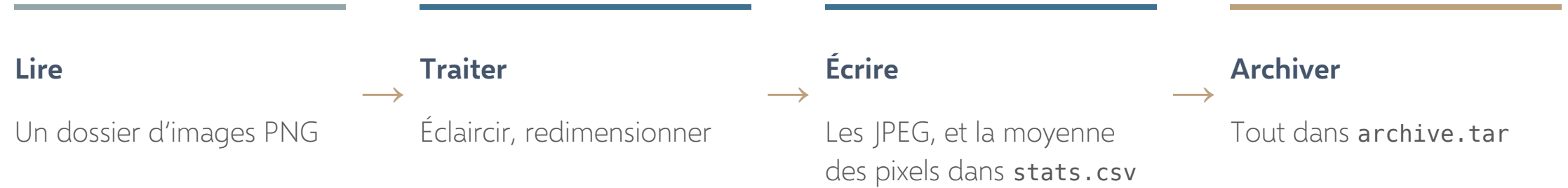


3. TP : découvrir le fil rouge

Lancer le code de départ et relever tout ce qui ne va pas, avant de voir comment l'écrire proprement.

Le fil rouge

Un petit programme qui sert de support à tous les TP :



■ Aujourd'hui ■ À venir

Le code de départ (`depart/`) fonctionne, mais il est volontairement mal écrit.

TP : démarrer

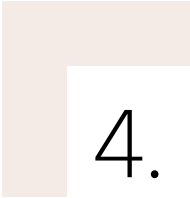
```
# "Fil rouge : départ" link, below these slides
uv unzip fil_rouge_depart.zip
cd fil_rouge/depart
uv run generate_images.py data 20 # 20 test images
uv run pipeline.py                # run the pipeline
```

L'en-tête PEP 723 déclare les dépendances du script : `uv run` crée son environnement.

```
# /// script
# requires-python = ">=3.12"
# dependencies = ["numpy", "pillow"]
# ///
```

Premier exercice (10 min)

Lisez `depart/pipeline.py` et listez tout ce qui vous gêne : il y a largement plus de dix problèmes. On met en commun : la suite du cours y répond.



4. Structurer ses données et son code

Une configuration validée, des chemins typés, un point d'entrée, et un pipeline d'opérations interchangeables.

Les classes : un moule à objets

Bases

Une **classe** regroupe des données et les fonctions qui les manipulent, un peu comme une **struct** C accompagnée de ses fonctions. Chaque **objet** fabriqué avec ce moule a ses propres valeurs.

```
class Robot:
    def __init__(self, name: str) -> None: # on creation
        self.name = name                 # attribute
        self.x = 0

    def move(self, step: int) -> None:      # method
        self.x += step

r2 = Robot("R2")    # create an object
r2.move(3)          # r2.x is now 3
```

- **self** : l'objet lui-même (proche du pointeur vers la **struct** en C)
- **__init__** : le **constructeur**, appelé par `Robot("R2")`
- **Attribut** : une donnée de l'objet (`r2.x`) ;
méthode : une fonction de l'objet (`r2.move(3)`)
- Les noms entourés de `__` sont des méthodes spéciales, appelées par Python

Une classe décrit un type d'objet : ses données (attributs) et ce qu'il sait faire (méthodes).

La configuration : une `dataclass` validée

`depart/pipeline.py` lit ses paramètres dans des variables globales. On les regroupe dans une **`dataclass`** : on déclare des champs typés, et `__init__`, `__repr__` et `__eq__` sont générés.

```
@dataclass(frozen=True)
class Config:
    input_dir: Path
    output_dir: Path
    width: int = 160
    quality: int = 85

    def __post_init__(self) -> None:
        if self.width <= 0:
            raise ValueError("width must be > 0")

    @property
    def images_dir(self) -> Path:
        return self.output_dir / "images"
```

- `frozen=True` : immuable (donc hashable) ; la configuration ne change pas en cours d'exécution
- `__post_init__` s'exécute juste après `__init__` : c'est là qu'on valide
- `@property` : attribut calculé, `config.images_dir`
- Une seule `config` passée aux fonctions, plus aucune globale

Valider à la construction : une erreur de paramètre apparaît au démarrage, pas au milieu du traitement.

Chemins : `pathlib`

`FOLDER + "/" + f` casse au premier séparateur oublié, et sous Windows. Un `Path` représente un chemin par un objet typé, avec ses opérations.

```
from pathlib import Path

folder = Path("data")
path = folder / "image_000.png"
path.stem, path.suffix      # "image_000", ".png"
path.with_suffix(".jpg")    # data/image_000.jpg
sorted(folder.glob("*.png"))
Path("output").mkdir(parents=True, exist_ok=True)
```

os.path	pathlib
os.path.join(d, f)	d / f
os.path.exists(p)	p.exists()
os.listdir(d) + endswith	d.glob("*.png")
splitext(f)[0] + ".jpg"	p.with_suffix(".jpg")
open(p).read()	p.read_text()

Point d'entrée : `main` et `argparse`

Les paramètres viennent de la ligne de commande (`uv run pipeline.py data output --width 320`), pas de constantes dans le code. `argparse` les analyse, les convertit et génère `--help`.

```
import argparse

def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument("input_dir", type=Path)
    parser.add_argument("output_dir", type=Path)
    parser.add_argument("--width", type=int, default=160)
    args = parser.parse_args()
    config = Config(args.input_dir, args.output_dir,
                    width=args.width)

    run(config)

if __name__ == "__main__":
    main()
```

- Arguments positionnels obligatoires, options `--x` avec valeur par défaut ; `type=` convertit et rejette les valeurs invalides
- `if __name__ == "__main__": main()` ne tourne que si le fichier est lancé, pas s'il est importé (par un autre module, par les tests)
- Seul `main` lit l'extérieur : le reste du code reçoit une `Config`

Le problème : tout est dans `process`

Dans `depart`, une seule fonction lit l'image, la transforme et l'écrit. Les étapes de traitement sont noyées au milieu des entrées-sorties.

```
def process(f): # simplified
    img = np.array(Image.open(FOLDER + "/" + f)) # read
    img = brighten(img) # step 1
    h = round(img.shape[0] * width / img.shape[1])
    small = Image.fromarray(img).resize((width, h))
    m = mean(np.array(small)) # stats
    small.save(OUTPUT + "/images/" + f) # write
```

- Ajouter un recadrage : il faut réécrire `process`
- Retirer ou réordonner une étape : idem
- Les réglages (`factor`, `width`) sont des variables globales
- Impossible de tester une étape seule

Objectif : séparer **quoi faire** à l'image (les étapes) de **comment la lire et l'écrire** (le pipeline).

Une opération = un objet qu'on appelle

Chaque étape devient une **opération** de même forme, image → image. Le pipeline applique dans l'ordre celles d'une liste.



```
class Brighten:
    def __init__(self, factor: float) -> None:
        self.factor = factor # setting, fixed once

    def __call__(self, image: np.ndarray) -> np.ndarray:
        ... # each pixel × factor, capped at 255

operations = [Brighten(1.2), Resize(160)]
for operation in operations:
    image = operation(image) # called like a function
```

- Les réglages vont dans `__init__`, l'action dans `__call__`
- `__call__` permet d'appeler l'objet comme une fonction : `operation(image)`
- Le pipeline enchaîne les opérations sans savoir ce qu'elles font
- Ajouter, retirer, réordonner : on modifie la liste, pas le pipeline

Une interface commune : l'héritage

Rien n'empêche encore de glisser dans la liste un objet sans `__call__` : le pipeline planterait au milieu du traitement. Une classe de base **abstraite** fixe le contrat que toute opération doit respecter.

```
from abc import ABC, abstractmethod

class Operation(ABC):
    @abstractmethod
    def __call__(self, image: np.ndarray) -> np.ndarray: ...

class Brighten(Operation): ... # same code as before

class Crop(Operation):
    def __init__(self, margin: int) -> None:
        self.margin = margin
        # __call__ forgotten
Crop(10)
# TypeError: Can't instantiate abstract class Crop
# without an implementation for abstract method '__call__'
```

- `Operation` ne fait rien : elle déclare ce que toute opération doit savoir faire
- `Brighten(Operation)` **hérite** de ce contrat et le remplit
- Un oubli se voit dès la création de l'objet, pas à la 150^e image
- La liste s'annote `list[Operation]` : ty vérifie qu'elle ne contient que des opérations

Ce découpage est le patron de conception **Strategy** : des algorithmes interchangeables derrière une même interface.
Catalogue : refactoring.guru/design-patterns



5. Erreurs et logging

Échouer proprement sur une image illisible, et tracer ce que fait le programme.

Les exceptions : quand ça se passe mal

Bases

Fichier absent, image corrompue, division par zéro : Python **lève une exception**, un signal d'erreur qui arrête le programme... sauf si le code l'**attrape**.

```
image = Image.open("data/corrupted.png") # a file that is not an image
```

Traceback (most recent call last):

```
File "demo.py", line 5, in <module>
```

```
    image = Image.open("data/corrupted.png")
```

```
File ".../PIL/Image.py", line 3715, in open
```

```
    raise UnidentifiedImageError(msg)
```

```
PIL.UnidentifiedImageError: cannot identify image file 'data/corrupted.png'
```

- La **traceback** : le chemin parcouru par le programme jusqu'à l'erreur
- Lire d'abord la **dernière ligne** (le type d'erreur et son message), puis remonter jusqu'à une ligne de **votre** code

Une traceback se lit de bas en haut : la dernière ligne dit ce qui s'est passé.

Capturer précisément, lever des exceptions métier

Une image corrompue sur 200 ne doit pas arrêter le traitement, mais elle doit être signalée, avec sa cause.

Dans depart

```
try:
    process(f)
except:
    print("error")
```

- Attrape tout, y compris `NameError` (une faute de frappe) et `KeyboardInterrupt` (Ctrl+C)
- Aucune information : quel fichier, quelle erreur ?
- Capturer le type le plus précis, là où l'on sait quoi faire : ici, ignorer l'image et le journaliser
- Exception **métier** : l'appelant gère `UnreadableImageError` sans connaître Pillow (`UnidentifiedImageError` hérite d'`OSError`) ; `from error` garde la cause dans la traceback

À faire

```
class UnreadableImageError(Exception):
    """The file is not a usable image."""

def load(path: Path) -> np.ndarray:
    try:
        with Image.open(path) as image:
            return np.asarray(image.convert("RGB"))
    except OSError as error:
        raise UnreadableImageError(path) from error
```

Exceptions : idiomes

Les réflexes attendus dans le code du projet.

EAFP plutôt que LBYL

```
if p.exists():                # LBYL
    image = load(p)

try:                          # EAFP
    image = load(p)
except FileNotFoundError:
    logger.warning("Missing: %s", p)
```

« Easier to Ask Forgiveness than Permission » : essayer, puis gérer l'échec. Le test préalable (« Look Before You Leap ») coûte un accès disque, laisse une condition de course et ne détecte pas un fichier corrompu.

Structurer ses erreurs

```
class PipelineError(Exception): ...
class UnreadableImageError(PipelineError): ...
```

- Une base par projet : `except PipelineError` attrape toutes les erreurs métier
- `try / except / else / finally` : `else` si aucune erreur, `finally` toujours
- Dans un `except`, `raise` seul relance l'exception telle quelle
- Un cas normal (clé absente, liste vide) ne passe pas par une exception : `dict.get`, `if not items`

logging plutôt que print

`print("ok", f)` : ni niveau, ni horodatage, impossible à filtrer ou à rediriger. **logging** associe à chaque message un **niveau** et le module d'origine ; la configuration décide de ce qui est affiché, et où.

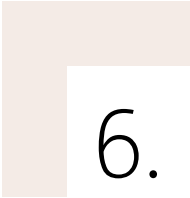
```
import logging

# one logger per module, named after it
logger = logging.getLogger(__name__)

def process(path: Path) -> None:
    logger.debug("Reading %s", path)
    try:
        image = load(path)
    except UnreadableImageError:
        logger.exception("Skipped image: %s", path)

def main() -> None:
    logging.basicConfig(level=logging.INFO)
```

- Niveaux `DEBUG < INFO < WARNING < ERROR < CRITICAL` ; `level=` fixe le seuil (- - `verbose` → `DEBUG`)
- `basicConfig` une seule fois, dans `main` : une bibliothèque ne configure jamais le logging
- `"%s"`, x plutôt qu'une f-string : le message n'est formaté que s'il est émis
- `logger.exception` ajoute la traceback au message
- `print` reste pour la **sortie** d'un outil, le résultat demandé



6. Trois outils du langage

`with`, générateurs, décorateurs : trois mécanismes omniprésents, tous utilisés dans le corrigé du TP.

with et les context managers

`with` garantit la libération d'une ressource (fichier, archive, verrou, caméra...) à la sortie du bloc, **même si une exception est levée** : une porte à fermeture automatique. Dans `depart`, si `tar.add` échoue, `tar.close()` n'est jamais appelé.

```
with tarfile.open(destination, "w") as tar:
    tar.add(folder, arcname="images")
# archive closed here, no matter what
```

- Un context manager implémente `__enter__` et `__exit__`
- `open`, `tarfile.open` et `Image.open` en sont

Écrire le sien

```
from contextlib import contextmanager

@contextmanager
def timer(name: str) -> Iterator[None]:
    start = time.perf_counter()
    try:
        yield # body of the with
    finally:
        elapsed = time.perf_counter() - start
        logger.info("%s: %.2f s", name, elapsed)
```

Générateurs : produire à la demande

Une fonction qui contient `yield` renvoie un **générateur** : les valeurs sont produites une par une, quand la boucle les demande. Comme le flux d'une caméra : on traite chaque image à son arrivée, sans stocker la vidéo entière. La mémoire reste constante, et les étapes se chaînent.

```
from collections.abc import Iterable, Iterator

def list_images(folder: Path) -> Iterator[Path]:
    for path in sorted(folder.glob("*.png")):
        yield path

def images(paths: Iterable[Path]) -> Iterator[Image]:
    for path in paths:
        yield load(path)

for image in images(list_images(folder)):
    ...    # only one image in memory at a time
```

- Rien ne s'exécute avant le premier tour de boucle
- Expression génératrice : `(p for p in d.iterdir() if p.is_file())`
- `yield from iterable` délègue à un autre itérable ; `itertools.batched(it, 32)` découpe en lots (3.12)
- Un générateur ne se parcourt qu'une fois

Décorateurs

Un décorateur prend une fonction et en renvoie une version enveloppée : `@timed` au-dessus de `def run` équivaut à `run = timed(run)`. Comme une gaine autour d'un câble : le conducteur ne change pas, on ajoute une couche autour (mesure, cache, journalisation) sans toucher au code de la fonction.

```
def timed(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        try:
            return func(*args, **kwargs)
        finally:
            elapsed = time.perf_counter() - start
            name = func.__name__
            logger.info("%s: %.2f s", name, elapsed)
    return wrapper
```

```
@timed
def run(config: Config) -> list[Result]: ...
```

- `*args, **kwargs` : transmet n'importe quels arguments
- `try / finally` : la durée est journalisée même si la fonction lève
- `functools.wraps` recopie nom et docstring ; sans lui, `run.__name__` vaudrait `"wrapper"`
- Déjà vus : `@dataclass`, `@property` ; dans la stdlib : `@functools.cache`

Typing un décorateur : ParamSpec

Pour aller plus loin

Le projet impose d'annoter toutes les fonctions, décorateur compris. Une annotation naïve (`Callable[..., Any]`) efface la signature de la fonction décorée ; la syntaxe générique de 3.12 la conserve (version du corrigé `etape_1`) :

```
def timed(**P, R)(func: Callable[P, R]) -> Callable[P, R]:
    name = getattr(func, "__name__", repr(func))

    @functools.wraps(func)
    def wrapper(*args: P.args, **kwargs: P.kwargs) -> R:
        start = time.perf_counter()
        try:
            return func(*args, **kwargs)
        finally:
            logger.info("%s: %.2f s", name, time.perf_counter() - start)

    return wrapper
```

- `**P` capture la liste des paramètres, `R` le type de retour : `run(config)` reste vérifié par `ty`
- `getattr(..., "__name__", ...)` : un `Callable` n'est pas forcément une fonction (un objet avec `__call__` n'a pas de `__name__`)

À retenir : structurer

Besoin	Réflexe	En une ligne
Dire ce qu'une fonction attend	Annotations, vérifiées par ty	<code>def area(w: int, h: int) -> int:</code>
Expliquer ce qu'elle fait	Docstring	<code>"""Area in pixels."""</code>
Regrouper les paramètres	<code>dataclass</code> figée et validée	<code>@dataclass(frozen=True)</code>
Manipuler des chemins	<code>pathlib</code>	<code>Path("data") / "img.png"</code>
Lire la ligne de commande	<code>argparse</code> dans <code>main</code>	<code>if __name__ == "__main__": main()</code>
Rendre des traitements interchangeables	Classe de base abstraite	<code>class Crop(Operation):</code>

À retenir : fiabiliser

Besoin	Réflexe	En une ligne
Réagir à une erreur	<code>except</code> précis, exception métier, <code>from</code>	<code>raise UnreadableImageError(p) from e</code>
Suivre ce qui se passe	<code>logging</code> , jamais <code>print</code>	<code>logger.info("%d images", n)</code>
Libérer une ressource	<code>with</code>	<code>with tarfile.open(p, "w") as tar:</code>
Traiter un flux sans tout charger	Générateur	<code>yield image</code>
Ajouter un comportement à plusieurs fonctions	Décorateur	<code>@timed</code>

Tout ce tableau est attendu dans le code du projet : c'est la note de qualité.



7. TP : fil rouge, étape 1

Réécrire proprement un petit programme de traitement d'images, sans changer son résultat.

TP : consignes

Réécrivez le pipeline dans `etape_1/`, **sans changer son résultat**. Tout est attendu : c'est le point de départ de la séance suivante.

Structure

1. **Config** : `dataclass` figée et validée, remplie par `argparse` dans `main`
2. `pathlib` pour tous les chemins
3. Opérations en classes qui héritent d'`Operation` (ABC) ; le pipeline est une liste d'opérations
4. Trois modules : `config.py`, `operations.py`, `pipeline.py`

Bonus : docstrings partout, décorateur typé avec `ParamSpec`, `list_images` en générateur.

Qualité

1. Toutes les fonctions annotées, `ty check` sans erreur
2. `logging` (option `--verbose`), plus aucun `print`
3. `UnreadableImageError` levée avec `from`, plus d'`except`: nu
4. `with` pour les fichiers et l'archive
5. `@timed` sur `run`

Ne cherchez pas à accélérer le code : c'est l'objet de la séance suivante.

TP : vérifier son travail

```
uvx ruff check . # lint
uvx ty check --python "$(uv python find --script pipeline.py)" . # types
uv run pipeline.py ../depart/data output
diff ../depart/output/stats.csv output/stats.csv # same result?
```

- `uv python find --script` donne à `ty` l'interpréteur de l'environnement du script, pour qu'il trouve NumPy et Pillow
- `diff` ne doit rien afficher : un refactoring ne change pas le résultat
- Robustesse : `echo x > ../depart/data/corrupted.png` ; le pipeline doit journaliser l'image ignorée et continuer

Corrigé : sous ce diaporama sur le site, après la séance. En retard, on repart de là à la séance suivante.

À retenir : obligatoire pour le projet

Types

Toutes les fonctions annotées, syntaxe moderne, vérifiées par `ty`

Exceptions

Précises, métier, chaînées avec `from` ; jamais d'`except` :
nu

Fortement recommandé : docstrings.

Logging

Un `logger` par module, aucun `print` pour tracer

Structure

`dataclass` validée pour la config, `pathlib`, opérations qui héritent d'une classe abstraite, pas de globales

QUESTIONS ?