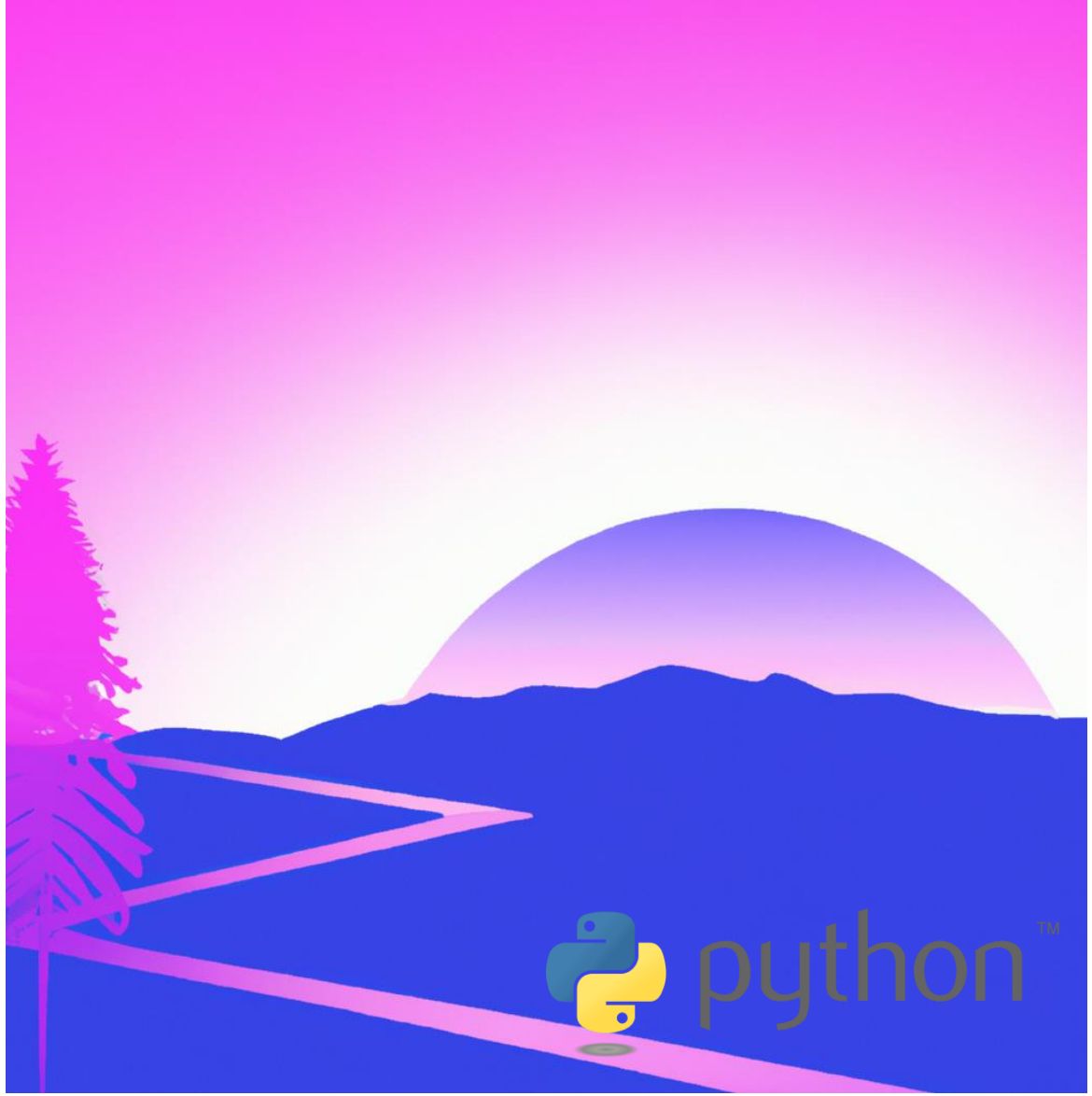
Programmation objet en Python



La programmation objet en Python

Les particularités du modèle objet de Python

➡ Tout est objet !

Les nombres, les fonctions, les classes, les modules, les types… Oui les classes aussi ! En Python, les classes sont instanciées par un métaclasse, en général `type`, ce qui veut dire que les classes sont aussi des instances d'une classe spéciale.

```
class A:
    pass

print(type(A))  # <class 'type'>
```

```
A = type("A", (), {"x": 42})
print(A.x)  # 42
```

La programmation objet en Python

Les particularités du modèle objet de Python

➡ Les attributs sont stockés dans un dictionnaire.

Les objets Python stockent leurs attributs dans un dictionnaire interne appelé `__dict__`.

```
class Person:
    def __init__(self, nom):
        self.nom = nom

p = Person("Alice")
print(p.__dict__)  # {'nom': 'Alice'}
```

La programmation objet en Python

L'écriture des classes et leur instanciation

Définir une classe

Une classe est un modèle qui définit la structure et le comportement de ses objets.

Syntaxe de base :

```
class ClassName:
    # constructeur (méthode spéciale appelée à la création)
    def __init__(self, param1, param2):
        self.param1 = param1
        self.param2 = param2

    # une méthode (comportement)
    def method_1(self):
        pass
```

```
class Car:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed

    def drive(self):
        print(f"The {self.brand} is going at {self.speed} km/h.")
```

La programmation objet en Python

L'écriture des classes et leur instanciation

Instancier (créer) des objets

L'instanciation consiste à créer un objet à partir de la classe, comme si on utilisait un « moule ».

```
v1 = Car("Peugeot", 120)
v2 = Car("Renault", 90)
```

v1 et v2 sont deux **instances** différentes de la classe Voiture. Chacune possède ses propres attributs (marque, vitesse).

Une fois un objet créé, il est possible d'accéder à ses attributs et à ses méthodes.

```
print(v1.brand)      # Peugoet
print(v2.speed)      # 90

v1.drive()           # The Peugeot is going at 120 km/h.
v2.drive()           # The Renault is going at 90 km/h.
```

La programmation objet en Python

Le rôle du paramètre self

Dans les méthodes d'une classe, le premier paramètre est toujours self. Il représente l'instance courante (l'objet sur lequel on travaille). Python le passe automatiquement lors de l'appel.

```
class Car:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed

    def drive(self):
        print(f"The {self.brand} is going at {self.speed} km/h.")
```

La programmation objet en Python

Modifier et ajouter des attributs

Il est possible de modifier et/ou ajouter des attributs à la volée.

```
v1.couleur = "Red"          # Ajout d'un nouvel attribut
print(v1.couleur)           # Red
```

La programmation objet en Python

Attribut de classe vs attribut d'instance

Jusque là nous avons vu des attributs dits « d'instance ». Un attribut de classe est partagé par toutes les instances d'une classe.

```
class Car:
    wheels = 4  # attribut de classe, toutes les voitures ont 4 roues

    def __init__(self, brand, speed):
        self.brand = brand   # attribut d'instance
        self.speed = speed   # attribut d'instance

    def drive(self):
        print(f"The {self.brand} is driving at {self.speed} km/h.")
```

La programmation objet en Python

TP fil rouge

TP 1 - Partie 1

OOP

La programmation objet en Python

L'héritage simple, l'héritage multiple, le polymorphisme

Héritage simple

Consiste à créer une classe fille qui hérite d'une seule classe parente. Cela permet à la classe fille de réutiliser les méthodes et attributs de la classe parente.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a noise")  # Affiche un bruit générique

# Classe fille héritant de Animal
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks")  # Redéfinition de la méthode speak

# Test
my_dog = Dog("Rex")
my_dog.speak()  # Affiche : Rex barks
```

La programmation objet en Python

L'héritage simple, l'héritage multiple, le polymorphisme

➡ **Le système d'héritage est dynamique !**

Python supporte une résolution d'ordre de méthode (MRO).

```
class A:
    pass
class B(A):
    pass
print(B.__mro__)
# (<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

L'héritage peut même être modifié dynamiquement à l'exécution.

```
class C:
    def hello(self):
        print("Hi from C")

B.__bases__ = (C,)  # changement de la classe parente à l'exécution

b = B()
b.hello()  # "Hi from C"
```

La programmation objet en Python

L'héritage simple, l'héritage multiple, le polymorphisme

Héritage multiple

Consiste à créer une classe qui hérite de plusieurs classes parentes. Python gère l'ordre de résolution des méthodes grâce au MRO.

```
class Flyable:
    def describe(self):
        print("I can fly")

class Swimmable:
    def describe(self):
        print("I can swim")

# Classe qui hérite de deux classes
class Duck(Flyable, Swimmable):
    pass

d = Duck()
d.describe()  # Affiche : I can fly (selon l'ordre MRO : Flyable est prioritaire)
```

La programmation objet en Python

L'héritage simple, l'héritage multiple, le polymorphisme

Polymorphisme

Permet d'utiliser la même interface pour des objets de classes différentes. Autrement dit, différentes classes peuvent répondre à la même méthode de façon spécifique.

```
class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows")  # Spécifique pour le chat

# Liste d'animaux
animals = [Dog("Rex"), Cat("Mimi")]

# Même méthode speak(), comportements différents
for a in animals:
    a.speak()
# Affiche :
# Rex barks
# Mimi meows
```

La programmation objet en Python

L'héritage simple, l'héritage multiple, le polymorphisme

Polymorphisme

Permet d'utiliser la même interface pour des objets de classes différentes. Autrement dit, différentes classes peuvent répondre à la même méthode de façon spécifique.

```
# Exemple avec héritage multiple et polymorphisme
class Robot:
    def describe(self):
        print("I am a robot")

class RobotDuck(Duck, Robot):
    def describe(self):
        print("I am a robotic duck")  # Redéfinition (override)

rd = RobotDuck()
rd.describe()  # Affiche : I am a robotic duck
```

La programmation objet en Python

TP fil rouge

TP 1 - Partie 2

OOP

La programmation objet en Python

La protection des attributs et des méthodes (notion de visibilité)

En Python, il n'existe pas de véritable encapsulation privée comme en Java ou C++, mais on utilise des conventions et des mécanismes internes pour indiquer le niveau d'accès.

Notation	Signification	Effet réel
name	Public	Accessible depuis l'extérieur, aucun contrôle
_name	Protégé	Convention : "ne pas utiliser en dehors de la classe ou du module", mais accessible quand même
__name	Privé	Python renomme l'attribut _Classe__nom pour éviter les collisions et l'accès direct

La programmation objet en Python

La protection des attributs et des méthodes (notion de visibilité)

```
class Car:
    def __init__(self):
        self.brand = "BMW"      # public
        self._speed = 120       # protected
        self.__secret = "xyz"   # private (name mangling)

v = Car()
print(v.brand)     # BMW
print(v._speed)    # 120 (accessible mais conventionnellement protégé)
# print(v.__secret)  # AttributeError
print(v._Car__secret)  # xyz (possible via name mangling)
```

La programmation objet en Python

La protection des attributs et des méthodes (notion de visibilité)

Même logique pour les méthodes :

```
class Car:
    def public_method(self):
        print("Méthode publique")

    def _protected_method(self):
        print("Méthode protégée")

    def __private_method(self):
        print("Méthode privée")

v = Car()
v.public_method()         # OK
v._protected_method()     # OK mais conventionnellement interne
# v.__private_method()    # AttributeError
v._Car__private_method() # OK via name mangling
```

La programmation objet en Python

La protection des attributs et des méthodes (notion de visibilité)

- Philosophie Zen de Python : « We are all consenting adults here ». Python fait confiance au développeur pour respecter les conventions.
- __name + name mangling protège surtout contre les collisions d'attributs dans l'héritage multiple.
- On peut néanmoins combiner avec __getattr__, __setattr__, __delattr__ pour contrôler dynamiquement l'accès aux attributs, et donc en particulier interdire la modification d'un attribut privé depuis l'extérieur :

```
class Car:
    def __init__(self):
        self.__speed = 120

    def __setattr__(self, name, value):
        if name == "_Car__speed":
            raise AttributeError("Accès interdit")
        super().__setattr__(name, value)

v = Car()
# v._Car__speed = 200  # AttributeError
```

La programmation objet en Python

Les méthodes spéciales

➡ Définition

- Une méthode spéciale est une fonction définie dans une classe avec des **double underscores** avant et après : __init__, __str__, __add__, etc.
- Python les utilise pour des **opérations intégrées** : création d'objets, affichage, opérateurs, comparaisons, conversions, etc.
- Elles ne sont **pas appelées directement** dans la plupart des cas ; Python les invoque automatiquement.
- Elles sont parfois appelées **dunder methods**.

La programmation objet en Python

Les méthodes spéciales

➡ Constructeurs et destructeurs : __init__, __new__, __del__

- __init__ n'est pas le vrai constructeur : il initialise après la création de l'objet.
- Le vrai constructeur est __new__, qui crée l'objet avant __init__. On le modifie rarement, sauf pour des classes immuables.

```
class Car:
    def __new__(cls, *args, **kwargs):
        print("Création de l'objet (avant __init__)")
        instance = super().__new__(cls)
        return instance

    def __init__(self, brand, speed):
        print("Initialisation de l'objet")
        self.brand = brand
        self.speed = speed
```

La programmation objet en Python

Les méthodes spéciales

Constructeurs et destructeurs : __init__, __new__, __del__

- __del__est la méthode appelée quand un objet est détruit, c'est-à-dire quand il n'a plus aucune référence et que le garbage collector le libère.
- C'est la méthode appelée quand un objet est détruit, c'est-à-dire quand il n'a plus aucune référence et que le garbage collector le libère.
- Même après del, __del__ n'est pas toujours garanti d'être appelé immédiatement. Le garbage collector peut retarder ou ignorer l'appel s'il reste des références circulaires. Dans les environnements comme CPython, la destruction se produit quand le compteur de référence tombe à 0.

```
class Car:
    def __del__(self):
        print("Car deleted")

v1 = Car()
v2 = v1   # deux noms pointent vers le même objet
del v1    # supprime une référence
del v2    # supprime la dernière référence
```

La programmation objet en Python

Les méthodes spéciales

Context managers : __enter__ et __exit__

Remarque : pour libérer des ressources, on n'utilise généralement pas __del__ mais __exit__ qui est appelée lors de l'utilisation de context managers (avec `with`).

Le pendant de __exit__, appelé en début de context manager, est __enter__.

```
class Connection:
    def __enter__(self):
        print("Connexion ouverte")
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Connexion fermée")

with Connection() as c:
    print("Travail avec la connexion")
```

La programmation objet en Python

Les méthodes spéciales

➡ Context managers : __enter__ et __exit__

Remarque : pour libérer des ressources, on n'utilise généralement pas __del__ mais __exit__ qui est appelée lors de l'utilisation de context managers (avec `with`).

	__del__	__exit__
Quand est-elle appelée ?	Quand l'objet est détruit (références à zéro)	Quand le bloc with se termine
Garantie d'appel	Non garanti	Toujours appelé, même en cas d'exception
Usage	Libération générale, nettoyage mémoire	Gestion fiable de ressources critiques (fichiers, connexions)
Dépend de	Garbage collector	Bloc with explicite

La programmation objet en Python

Les méthodes spéciales

➡ Représentation d'un objet : __repr__, __str__ et __format__

__repr__ : représentation officielle pour le développeur

__str__ : représentation lisible pour l'utilisateur (utilisée par `print`)

__format__ : contrôle le formatage dans f-strings ou format()

```
class Car:
    def __init__(self, brand):
        self.brand = brand

    def __str__(self):
        return f"Car: {self.brand}"

v = Car("Audi")
print(v)  # Car: Audi
```

La programmation objet en Python

Les méthodes spéciales

➡ **Opérateurs : __add__, __sub__, __mul__, __eq__, __lt__...**

Utilisé par les symboles opérateurs (+, -, *, ==, <...).

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):  # opérateur +
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3.x, p3.y)  # 4 6
```

La programmation objet en Python

Les méthodes spéciales

➡ **Accès aux attributs : __getattr__, __setattr__, __delattr__**

```
class Car:
    def __init__(self, brand):
        self.brand = brand

    def __getattr__(self, name):  # appelé quand l'attribut n'existe pas
        return f"{name} inexistant"

v = Car("BMW")
print(v.color)  # color inexistant
```

__setattr__, __delattr__ permettent quant à eux de contrôler l'écriture et la suppression d'attributs.

La programmation objet en Python

Les méthodes spéciales

➡ Le cas des conteneurs

Méthode	Utilisation	Exemple
__len__(self)	Retourne le nombre d'éléments	len(obj)
__getitem__(self, key)	Accès via index ou clé	obj[key]
__setitem__(self, key, value)	Assignation via index ou clé	obj[key] = value
__delitem__(self, key)	Suppression via index ou clé	del obj[key]
__iter__(self)	Retourne un itérateur	for x in obj:
__contains__(self, item)	Vérifie l'appartenance	item in obj

La programmation objet en Python

L'introspection

Capacité d'un programme à **examiner ses propres objets** (types, attributs, méthodes, classes, modules…) au moment de l'exécution (sans les modifier).

En Python, tout est un objet, donc l'introspection est très puissante et largement utilisée.

Cas d'usage typiques :

- Débogage et exploration d'objets inconnus.
- Sérialisation/désérialisation (ex. JSON).
- Création de frameworks ou de bibliothèques génériques (ORM, GUI, tests automatisés).
- Méthodes dynamiques ou proxy (ex. intercepter l'accès aux attributs).

La programmation objet en Python

L'introspection

	Utilité	Exemple
type(obj)	Retourne le type de l'objet	type(42) → <class 'int'>
id(obj)	Retourne l'identifiant unique en mémoire	id(obj)
dir(obj)	Liste les attributs et méthodes disponibles	dir([]) → ['append', 'clear', 'copy', ...]
hasattr(obj, "attr")	Vérifie si l'attribut existe	hasattr(obj, "append") → True
getattr(obj, "attr", default)	Accède dynamiquement à un attribut	getattr([], "append") → méthode append
setattr(obj, "attr", value)	Définit dynamiquement un attribut	setattr(obj, "x", 42)
isinstance(obj, cls)	Vérifie si l'objet est instance d'une classe	isinstance(42, int) → True
issubclass(Sub, Base)	Vérifie l'héritage	issubclass(bool, int) → True
callable(obj)	Vérifie si l'objet peut être appelé	callable(print) → True
help(obj)	Affiche documentation et signature	help(str)

La programmation objet en Python

L'introspection

```
class Car:
    wheels = 4
    def drive(self):
        pass

v = Car()
print(v.__class__)  # <class '__main__.Car'>
print(v.__class__.__name__)  # Car
```

```
v.brand = "BMW"
print(v.__dict__)  # {'brand': 'BMW'}
```

```
import inspect
print(inspect.getmembers(Car, predicate=inspect.isfunction))
# [('drive', <function Car.drive at 0x...>)]
```

```
hasattr(v, "__str__")  # True
```

La programmation objet en Python

L'implémentation des interfaces

Une interface définit un contrat : quelles méthodes une classe doit implémenter.

En Python, ce contrat peut être assuré par :

- Des **conventions** : créer une classe avec des méthodes « à implémenter » et documenter l'usage.
- Des **classes abstraites** avec le module abc (Abstract Base Classes) : plus sûr, car Python empêche l'instanciation si les méthodes abstraites ne sont pas implémentées.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def drive(self):
        pass

    @abstractmethod
    def stop(self):
        pass
```

La programmation objet en Python

L'implémentation des interfaces

Implémentation concrète :

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def drive(self):
        pass

    @abstractmethod
    def stop(self):
        pass
```

```
class Car(Vehicle):
    def drive(self):
        print("La voiture roule")

    def stop(self):
        print("La voiture s'arrête")

v = Car()  # OK
v.drive()  # La voiture roule
```

```
class Bike(Vehicle):
    def drive(self):
        print("Le vélo roule")

# v = Bike()
# TypeError: Can't instantiate abstract
class Bike with abstract methods stop
```

La programmation objet en Python

L'implémentation des interfaces

En Python, on n'utilise pas systématiquement de classe abstraite formelle. On se repose souvent sur le duck typing (« If it walks like a duck and quacks like a duck, it's a duck. »).

```
class Car:
    def drive(self):
        print("La voiture roule")

class Bike:
    def drive(self):
        print("Le vélo roule")

for v in [Car(), Bike()]:
    v.drive()  # Pas besoin d'interface formelle, tant que la méthode existe
```

((A titre personnel, je définis ou j'importe clairement mes classes abstraites.))

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Une exception est un signal qu'une erreur ou un événement inhabituel s'est produit.

Plutôt que de vérifier systématiquement des valeurs de retour (comme en C), Python lance une exception et permet de la capturer et gérer.

Cela sépare le code normal du code de traitement des erreurs.

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Syntaxe try/except :

```
try:
    x = int(input("Entrez un nombre : "))
    y = 10 / x
except ValueError:
    print("Erreur : ce n'est pas un nombre valide !")
except ZeroDivisionError:
    print("Erreur : division par zéro !")
```

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Syntaxe try/except/else/finally :

```
try:
    x = int(input("Entrez un nombre : "))
    y = 10 / x
except ValueError:
    print("Erreur : ce n'est pas un nombre valide !")
except ZeroDivisionError:
    print("Erreur : division par zéro !")
else:
    print(f"Résultat : {y}")
finally:
    print("Fin du bloc try/except")
```

- try : bloc contenant le code « à risque »
- except : capture l'exception spécifique et permet de la gérer
- else : exécuté **si aucune exception** n'a été levée
- finally : exécuté **dans tous les cas**, utile pour libérer des ressources (fichiers, connexions…)

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Lever une exception : `raise`

```
def inverse(x):
    if x == 0:
        raise ValueError("Impossible d’inverser zéro !")
    return 1 / x

print(inverse(2))  # OK
# print(inverse(0))  # ValueError levée
```

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Exceptions intégrées courantes

Exception	Quand est-elle levée ?
ValueError	Conversion ou valeur invalide
TypeError	Type inapproprié pour une opération
ZeroDivisionError	Division par zéro
IndexError	Index hors limites pour liste ou tuple
KeyError	Clé absente dans un dictionnaire
FileNotFoundError	Fichier inexistant

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Créer ses propres exceptions

```
class MyError(Exception):
    pass

def test(x):
    if x < 0:
        raise MyError("x doit être positif")

try:
    test(-1)
except MyError as e:
    print(f"Erreur personnalisée : {e}")
```

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Bonnes pratiques :

- Attraper des exceptions spécifiques, pas de « except: » tout court.
- Lever des exceptions pertinentes pour indiquer un problème.
- Utiliser finally ou un context manager (with) pour libérer les ressources.
- Ne pas utiliser les exceptions pour contrôler le flux normal ; elles sont pour les situations exceptionnelles. -> ?

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

Bonnes pratiques :

« Ne pas utiliser les exceptions pour contrôler le flux normal ;
elles sont pour les situations exceptionnelles. »

En Python, il existe un dilemme classique entre « tester avant d'agir » et « essayer d'agir et capturer les exceptions », appelé EAFP vs LBYL.

Acronyme	Signification	Exemple	Quand l'utiliser
EAFP	Easier to Ask Forgiveness than Permission (plus simple de demander pardon que permission)	Essayer directement une opération et capturer l'exception si elle échoue	Très pythonique, surtout pour les accès à des objets/dictionnaires, fichiers, etc.
LBYL	Look Before You Leap (regarder avant de sauter)	Tester si la condition est OK avant d'agir (if key in dict:)	Utile si la vérification est simple et peu coûteuse

La programmation objet en Python

L'utilisation du mécanisme d'exception pour la gestion des erreurs

```
d = {"a": 1, "b": 2}

try:
    value = d["c"]
except KeyError:
    value = 0

print(value)  # 0
```

EAFP

```
d = {"a": 1, "b": 2}

if "c" in d:
    value = d["c"]
else:
    value = 0

print(value)  # 0
```

LBYL

La programmation objet en Python

Les bonnes pratiques et les modèles de conception courants

Principes de conception OOP

- **Encapsulation** : protéger les attributs/méthodes sensibles (_nom, __nom)
- **Héritage et composition** : privilégier la composition (has-a) quand l'héritage (is-a) devient compliqué
- **Polymorphisme** : utiliser le duck typing ou des interfaces (abc)
- **Cohésion et responsabilité unique** : une classe/fonction doit avoir une seule responsabilité

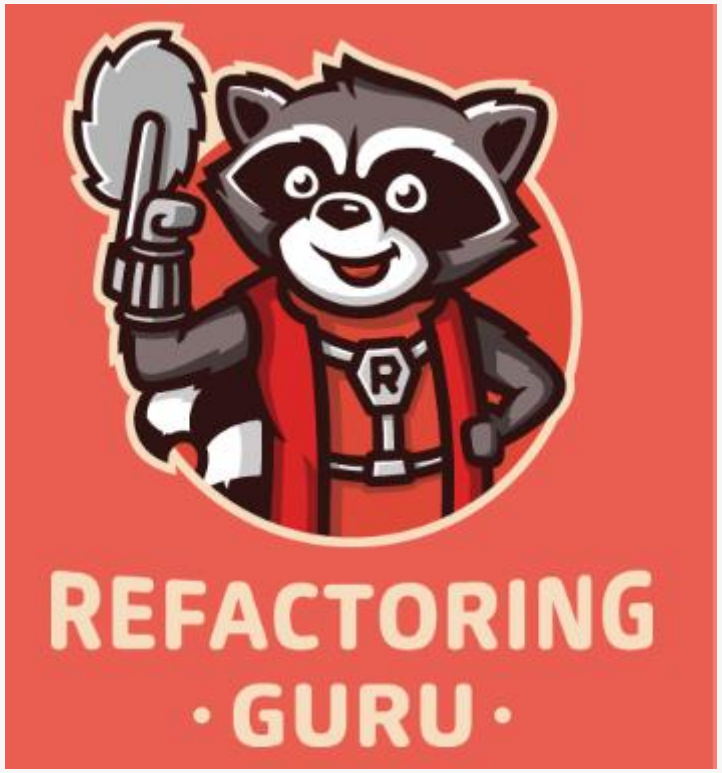
```
class Engine:
    def start(self):
        print("Moteur démarré")

class Car:
    def __init__(self):
        self.engine = Engine()  # composition : la voiture « possède » un moteur

    def drive(self):
        self.engine.start()
        print("La voiture roule")

v = Car()
v.drive()
```

Composition

La programmation objet en Python

Les bonnes pratiques et les modèles de conception courants

Design patterns en Python



https://refactoring.guru/design-patterns/python

Bonus

Les bonnes pratiques générales

Les bonnes pratiques générales

BONNE PRATIQUE 1 : Respecter la philosophie « Zen »

“Simple is better than complex”

“Readability counts”

Les bonnes pratiques générales

BONNE PRATIQUE 2 : Nommer clairement

Variables, fonctions et classes : noms explicites et cohérents

Conventions :

- **snake_case pour variables et fonctions**
- **CamelCase pour les classes**
- **_nom pour attribut/méthode interne,__nom pour privé**

Les bonnes pratiques générales

BONNE PRATIQUE 3 : Eviter la duplication de code

Factoriser dans des fonctions ou classes

Utiliser des modules et packages

Les bonnes pratiques générales

BONNE PRATIQUE 4 : Ecrire du code lisible et maintenable

Respecter le PEP 8

Ajouter des docstrings pour les fonctions/classes

Limiter la complexité cyclomatique des fonctions

Les bonnes pratiques générales

BONNE PRATIQUE 5 : Utiliser les exceptions correctement

Capturer des exceptions spécifiques

EAFP vs LBYL

Utiliser with pour gérer les ressources

Les bonnes pratiques générales

BONNE PRATIQUE 6 : Favoriser les structures Python natives (*)

Listes, dictionnaires, sets, tuples

Compréhensions ([x for x in ...]) pour plus de concision

Les bonnes pratiques générales

BONNE PRATIQUE 7 : Tester

Écrire des tests unitaires (unittest/pytest)

Vérifier les cas limites et les exceptions

Fin de cette partie

Merci !