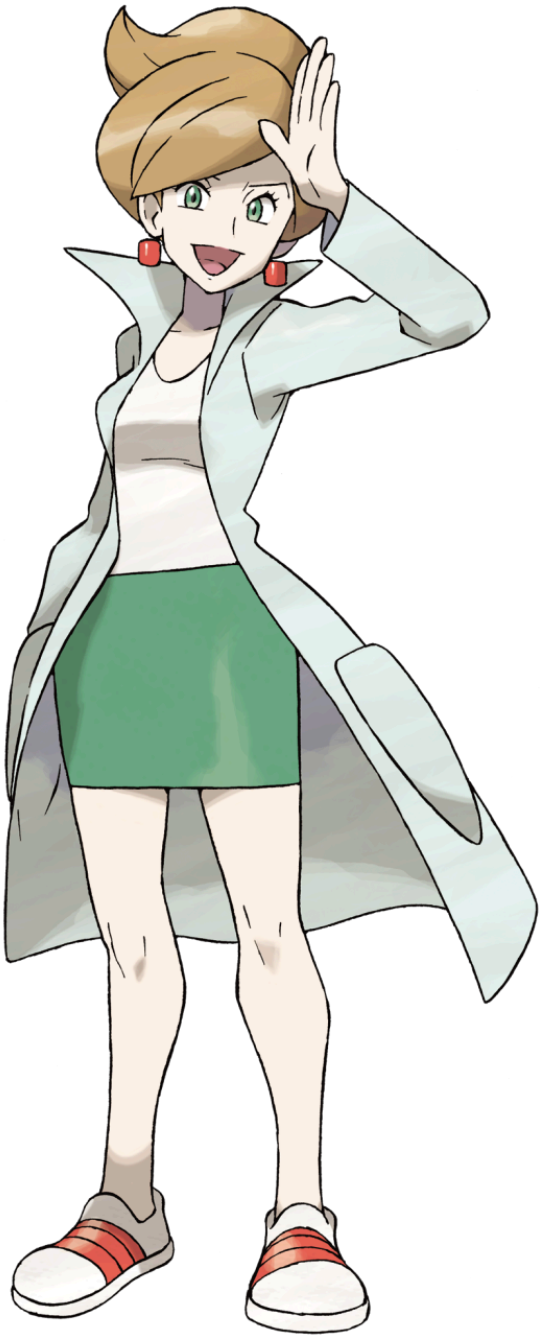
Tutoriel Python : découvrir les pratiques agiles de tests unitaires et refactoring avec Pokémon

Baptiste Matrat
Marie Probert

Le voyage continue !



“ Félicitations, jeune dresseur ! Tu as appris les bases dans le laboratoire de Java. Mais le monde des Pokémon s'agrandit et nous partons explorer une nouvelle région : les Terres de Python.

Ici, pas besoin de formules compliquées pour créer tes créatures. On écrit moins de lignes pour faire la même chose ! C'est l'endroit idéal pour perfectionner ta logique de dresseur : comment bien organiser tes Pokémon, comment gérer ton sac de Pokéballs et surtout, comment vérifier que tout fonctionne grâce aux tests.

Mais reste vigilant ! Dans ces terres plus libres, un code mal rangé peut vite devenir un vrai casse-tête. Ton défi sera de garder un code propre et efficace, pour que ton équipe soit toujours prête au combat.

Prépare tes Pokéballs et ton clavier : l'aventure continue !”

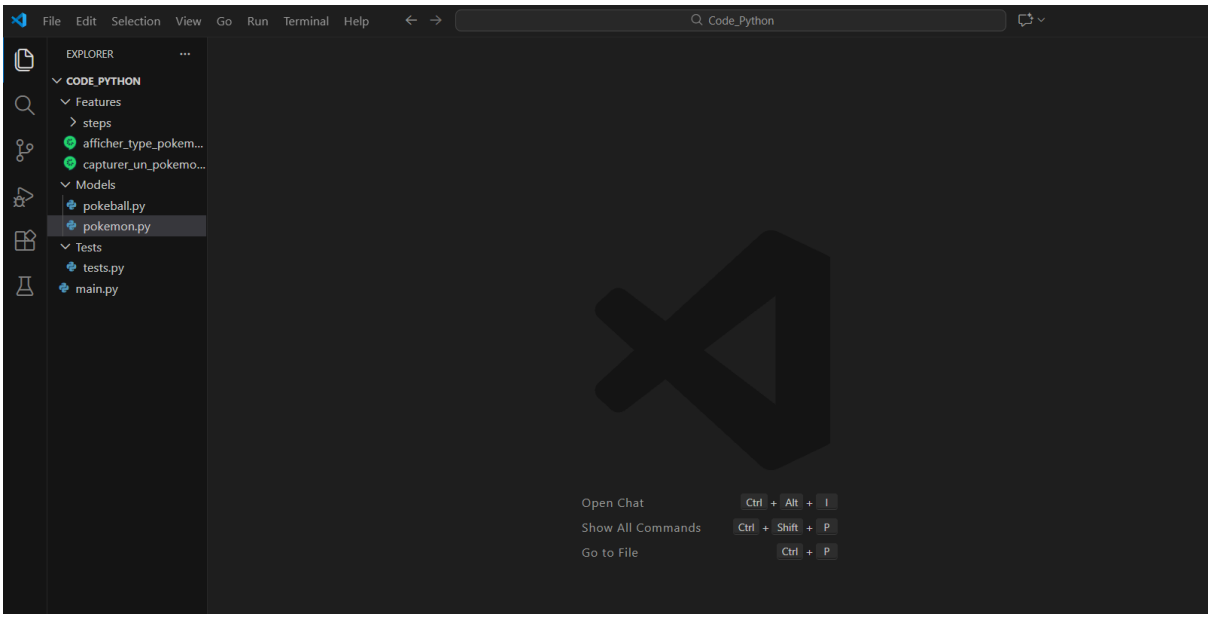
Sommaire :

Etape 0 : Configuration de VS Code

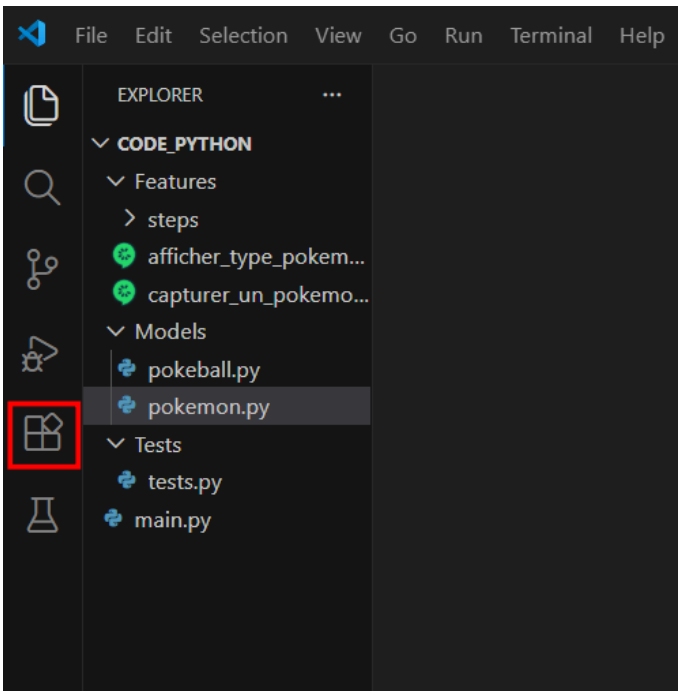
Bonjour jeune dresseur, pour ce tutoriel, nous avons changer de langage de code, avant si tu te souviens, nous utilisions Java sur BlueJ, maintenant nous allons utiliser Python pour continuer notre folle aventure.

Pour coder en Python, nous te conseillons de te télécharger VS Code, c'est un logiciel qui permet de coder dans tout un tas de langage différent en ajoutant différentes extensions.

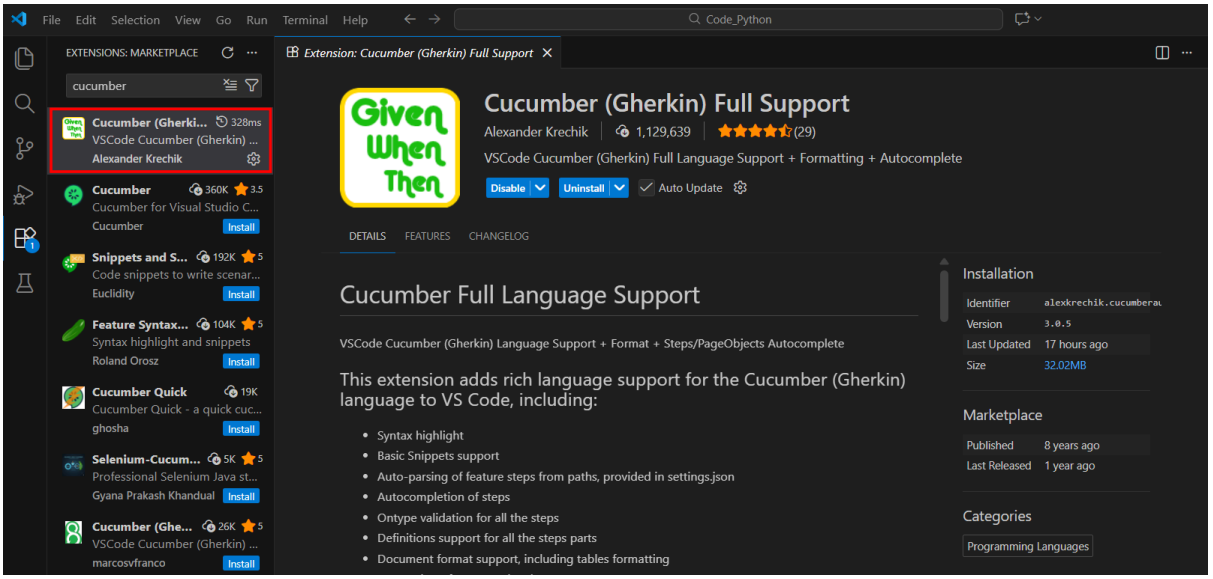
Pour télécharger VS Code suis ce lien : https://code.visualstudio.com/download



Ensuite une fois le téléchargement terminé, tu devras cliquer sur le bouton Extension (entouré en rouge sur l'image ci-dessous. Puis télécharge les extensions suivantes (on en aura besoin pendant ce tutoriel)



Insérer les extensions



Une fois le téléchargement terminé, bravo tu es fin prêt à commencer ton aventure Pokemon !

Étape 1 : Rappel des classes existantes

Si tu as suivi le tutoriel précédent je te conseille de tout de même suivre cette partie, car les classes que tu connaissais ont un peu changé.

Pour rappel, on avait une classe Pokémon et une classe Pokeball, un Pokémon pouvant être associé à une Pokéball lors de sa capture.

Voici notre nouvelle classe Pokémon, crée là simplement dans un fichier nommé “Pokemon.py”

```python
class Pokemon:
    def __init__(self, type1, type2, pokeball=None):
        self.type1 = type1
        self.type2 = type2
        self.pokeball = pokeball

    def get_type1(self):
        return self.type1

    def set_type1(self, new_type):
        self.type1 = new_type

    def get_type2(self):
        return self.type2

    def set_type2(self, new_type):
        self.type2 = new_type

    def get_pokeball(self):
        return self.pokeball

```

Voici notre nouvelle class Pokéball, crée là simplement dans un fichier nommé “Pokeball.py”

```python
class Pokeball:
    def __init__(self, prix, pokemon=None):
        self.prix = prix
        self.pokemon = pokemon
        if pokemon is not None:
                    pokemon.pokeball = self
    def afficher_type_pokemon(self):
        if self.pokemon is None:
            return "Cette Pokeball est vide. Aucun type à afficher."

        t1 = self.pokemon.type1
        t2 = self.pokemon.type2

        if t1 and t2:
            return f"De types {t1} et {t2}."
        elif t1:
            return f"De type {t1}."
        else:
            return "Le Pokémon dans cette Pokeball n'a pas de type défini."

    def capturer_pokemon(self, pokemon):
        if self.pokemon is not None:
            raise Exception("La Pokeball contient déjà un Pokémon.")
        if pokemon.pokeball is not None:
            raise Exception("Le Pokémon est déjà dans une Pokeball.")
        self.pokemon = pokemon
        pokemon.pokeball = self
```

Tu remarqueras que l’on a ajouté une fonction plus “propre” que la première fois pour la capture d’un Pokemon, nommé capturer_pokemon.

Étape 2 : Tests fonctionnels avec fichiers .feature et steps

Félicitations, tu viens d'enfiler ta casquette de dresseur de code. Pour que ton aventure Pokémon ne s'arrête pas au premier bug sauvage, tu dois t'assurer que ton application se comporte exactement comme prévu. C'est là qu'interviennent les **tests fonctionnels**.

Imagine que tu programmes un Pokédex. Un test unitaire vérifierait si la batterie fonctionne. Un **test fonctionnel**, lui, vérifie la fonction réelle : est-ce que si j'appuie sur le bouton, l'image de Pikachu s'affiche ? On teste l'expérience utilisateur, pas juste la mécanique interne.

Behave utilise un langage appelé **Gherkin**. C'est une manière d'écrire des tests en utilisant des phrases simples, presque comme si tu racontais une histoire à un ami. Cela permet aux dresseurs (développeurs) et aux professeurs (clients) de se comprendre.

Chaque scénario suit une structure logique :

- **Given (Étant donné)** : Le contexte initial (Tu as une Pokéball dans ton sac).
- **When (Quand)** : L'action que tu fais (Tu lances la Pokéball sur un Rattata).
- **Then (Alors)** : Le résultat attendu (Le Rattata est capturé).

Pour utiliser Behave, ton projet doit être organisé d'une manière précise. Voici l'arborescence :

- **Features/** : Le dossier principal.
    - **capturer_un_pokemon.feature** : C'est ici que tu écris ton histoire en langage humain.
    - **steps/** : Un sous-dossier obligatoire.
        - **capturer_un_pokemon_steps.py** : C'est ici que tu traduis ton histoire en code Python.

Dans ton fichier “capturer_un_pokemon.feature”, tu vas définir ton intention, tes scénarios.

```
Feature: Mettre un Pokémon dans une Pokéball

  En tant que dresseur Pokemon
  Je veux mettre un Pokémon sauvage dans une Pokéball
  Afin de le capturer

  Scenario: Mettre un Pokémon dans une Pokéball
    Given un Pokémon sauvage
    And une Pokéball vide
    When je mets le Pokémon dans la Pokéball
    Then la Pokéball doit contenir le Pokémon
    And le Pokémon doit référencer la Pokéball
```

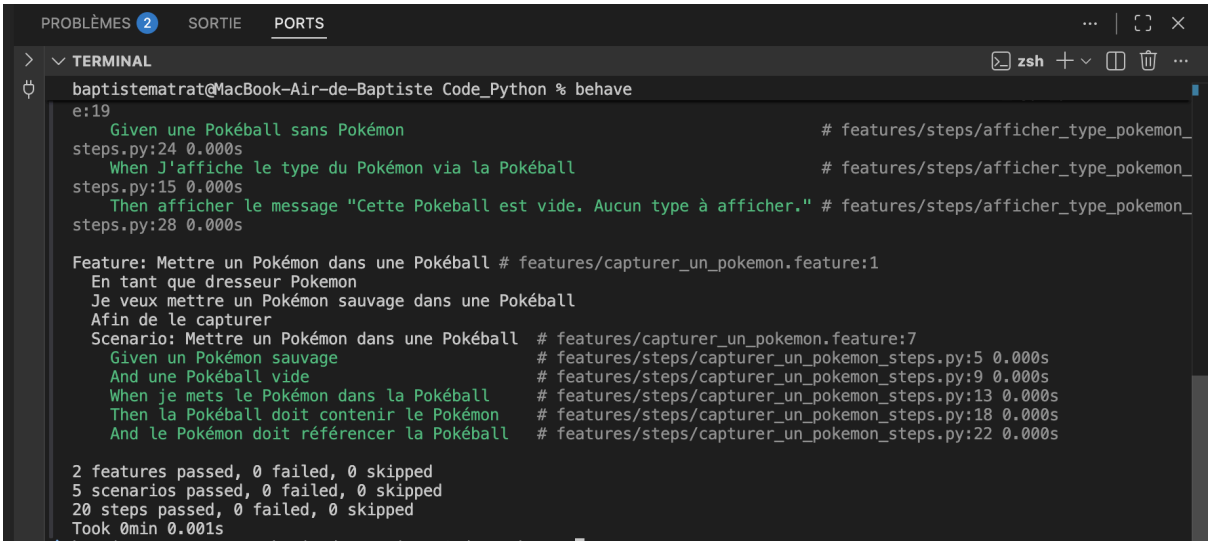
Behave va ensuite lire ton fichier texte et chercher les fonctions Python correspondantes. Dans “steps/capturer_un_pokemon_steps.py”, tu vas lier les phrases au code.

```python
from behave import given, when, then
from Models.pokemon import Pokemon
from Models.pokeball import Pokeball

@given('un Pokémon sauvage')
def step_given_pokemon_sauvage(context):
    context.pokemon = Pokemon("Feu", "Spectre")

@given('une Pokéball vide')
def step_given_pokeball_vide(context):
    context.pokeball = Pokeball(600)

@when('je mets le Pokémon dans la Pokéball')
def step_when_mettre_pokemon(context):
    context.pokeball.pokemon = context.pokemon
    context.pokemon.pokeball = context.pokeball

@then('la Pokéball doit contenir le Pokémon')
def step_then_pokeball_contient(context):
    assert context.pokeball.pokemon == context.pokemon

@then('le Pokémon doit référencer la Pokéball')
def step_then_pokemon_reference(context):
    assert context.pokemon.pokeball == context.pokeball
```

Une fois que tes fichiers sont prêts, retourne dans ton terminal et tape simplement : **behave**

Behave va parcourir ton dossier “Features”, lire tes scénarios et exécuter le code Python associé. Si tout est vert, ton Pokédex est prêt pour l'aventure. Si c'est rouge, c'est qu'un bug s'est glissé dans tes hautes herbes.

Tu peux voir ici que les lignes du fichier Feature sont en vert : elles sont toutes passées ! Aucun test n’a échoué ni n’a été sauté (skip).



Étape 3 : Implémentation de la classe dresseur

Pour devenir le meilleur, il te faut un personnage capable de porter des Pokéballs et de diriger les opérations : le **Dresseur**.

Dans cette étape, nous allons créer une classe simple qui représente l'humain de l'aventure. Un dresseur a un nom et un sac (une liste) pour ranger ses Pokéballs.

Crée un nouveau fichier nommé “Dresseur.py” dans ton dossier principal et ajoute ce code :

```python
class Dresseur:
    def __init__(self, nom):
        self.nom = nom
        self.pokeballs = []

    def get_nom(self):
        return self.nom

    def get_pokeballs(self):
        return self.pokeballs

    def ajouter_pokeball(self, pokeball):
        self.pokeballs.append(pokeball)

    def capturer_pokemon(self, pokemon, pokeball):
        if pokeball.capturer_pokemon(pokemon):
            print(f"{self.nom} a capturé {pokemon.nom} dans une Pokéball!")
        else:
            print(f"La Pokéball est déjà pleine ou la capture a échoué.")

    def liberer_pokemon(self, pokeball):
        if pokeball.pokemon is not None:
            print(f"{self.nom} a libéré {pokeball.pokemon.nom} de la Pokéball.")
            pokeball.pokemon.pokeball = None
            pokeball.pokemon = None
        else:
            print("La Pokéball est déjà vide.")
```

Ton dresseur n'est pas juste un spectateur, il agit sur son environnement :

- L'inventaire (pokeballs) : C'est une liste qui stocke tes objets. On utilise ajouter_pokeball pour la remplir.
- La méthode capturer_pokemon : C'est l'action principale. Le dresseur utilise une Pokéball précise sur un Pokémon précis. Si la Pokéball accepte le Pokémon (grâce à sa propre logique interne), le dresseur confirme la réussite.
- La méthode liberer_pokemon : Parfois, il faut savoir rendre sa liberté à un compagnon. Le dresseur vide la Pokéball et "déconnecte" le lien entre l'objet et le Pokémon.

Notre Dresseur est prêt ! Maintenant, tout comme pour le Pokémon et la Pokéball, nous devons vérifier que ses actions fonctionnent correctement. Pour cela, nous allons ajouter de nouveaux tests unitaires dans le fichier “tests.py” afin de valider chaque fonctionnalité ajoutée.

```python
class TestDresseur(unittest.TestCase):
    def setUp(self):
        self.dresseur = Dresseur("Sacha")
        self.pokemon = Pokemon("Pikachu", "Électrik", None)
        self.pokeball = Pokeball(prix=200)

    def test_modification_nom(self):
        self.dresseur.nom = "Régis"
        self.assertEqual(self.dresseur.nom, "Régis")

    def test_ajouter_pokeball(self):
        self.dresseur.ajouter_pokeball(self.pokeball)
        self.assertIn(self.pokeball, self.dresseur.pokeballs)
        self.assertEqual(len(self.dresseur.pokeballs), 1)

    def test_capturer_pokemon_succes(self):
        print("--- Test de capture de Pokémon ---")
        self.dresseur.capturer_pokemon(self.pokemon, self.pokeball)
        self.assertEqual(self.pokeball.pokemon, self.pokemon)
        self.assertEqual(self.pokemon.pokeball, self.pokeball)

    def test_liberer_pokemon(self):
        self.dresseur.capturer_pokemon(self.pokemon, self.pokeball)

        self.dresseur.liberer_pokemon(self.pokeball)
        self.assertIsNone(self.pokeball.pokemon)
        self.assertIsNone(self.pokemon.pokeball)
```

Étape 4 : Techniques de refactoring

Dans le monde Pokémon, un Pokémon évolue pour devenir plus fort. En programmation, le code a aussi besoin d'évoluer : c'est ce qu'on appelle le **Refactoring**.

Le refactoring, c'est l'art de modifier la structure interne de ton code pour le rendre plus propre et plus lisible, **sans changer son comportement**. Tes tests Behave sont là pour ça : si après tes modifications, `behave` affiche toujours du vert, c'est que ton code fonctionne toujours parfaitement !

Nous allons voir deux techniques essentielles.

Etape 4.1 : Rename

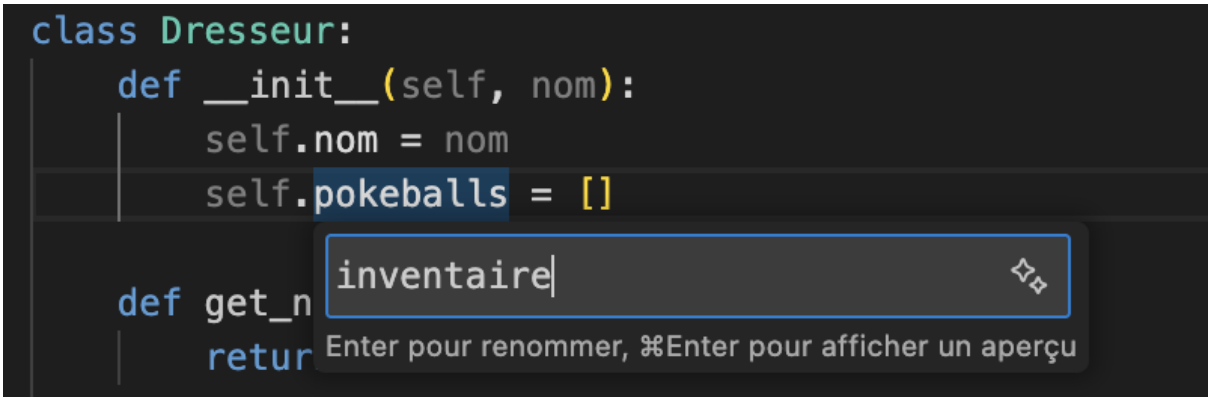
Imagine que le Professeur Chen vienne te voir et te dise : "Le terme pokeballs dans ton sac est trop vague, on devrait l'appeler inventaire car tu pourrais y ranger des potions plus tard".

```python
class Dresseur:
    def __init__(self, nom):
        self.nom = nom
        self.pokeballs = []
```

Si tu changes manuellement le nom de la variable partout, tu risques d'en oublier un et de créer une erreur. Dans VS Code, tu peux faire un Rename intelligent :

1. Clique sur la variable self.pokeballs dans ta classe Dresseur.
2. Appuie sur F2 (ou fais un clic droit puis clique sur renommer le symbole)
3. Tape le nouveau nom : inventaire.
4. Appuie sur Entrée.

VS Code a mis à jour self.inventaire partout, même dans tes méthodes get_pokeballs et ajouter_pokeball.




```python
class Dresseur:
    def __init__(self, nom):
        self.nom = nom
        self.inventaire = []

    def get_nom(self):
        return self.nom

    def get_inventaire(self):
        return self.inventaire

    def ajouter_pokeball(self, pokeball):
        self.inventaire.append(pokeball)

    def capturer_pokemon(self, pokemon, pokeball):
        if pokeball.capturer_pokemon(pokemon):
            print(f"{self.nom} a capturé {pokemon.nom} dans une Pokéball!")
        else:
            print(f"La Pokéball est déjà pleine ou la capture a échoué.")

    def liberer_pokemon(self, pokeball):
        if pokeball.pokemon is not None:
            print(f"{self.nom} a libéré {pokeball.pokemon.nom} de la Pokéball.")
            pokeball.pokemon.pokeball = None
            pokeball.pokemon = None
        else:
            print("La Pokéball est déjà vide.")
```

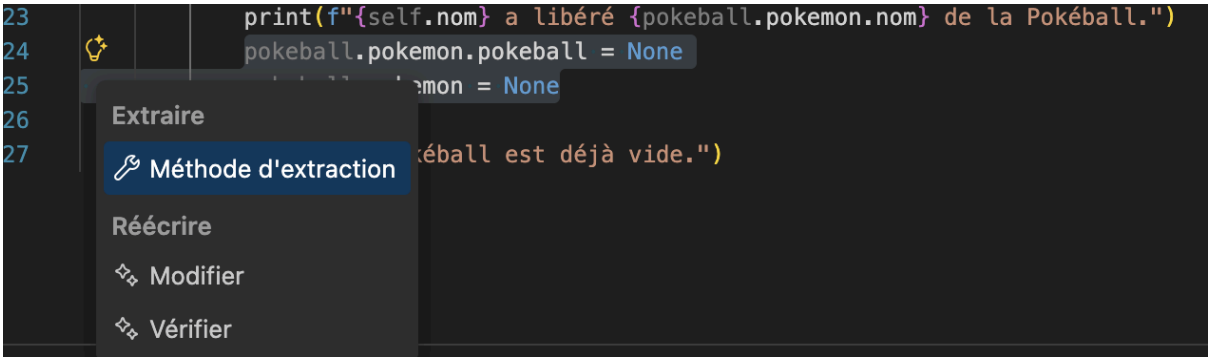
Étape 4.2 : extractMethod

Regarde ta méthode liberer_pokemon. Elle fait deux choses : elle affiche un message ET elle nettoie les liens entre la balle et le Pokémon. Pour rendre le code plus modulaire, on va extraire la partie "nettoyage".

Avant le refactoring :

```
21      def liberer_pokemon(self, pokeball):
22          if pokeball.pokemon is not None:
23              print(f"{self.nom} a libéré {pokeball.pokemon.nom} de la Pokéball.")
24              pokeball.pokemon.pokeball = None
25              pokeball.pokemon = None
```

Après l'Extract Method : On crée une petite méthode spécialisée pour le nettoyage, et on l'appelle dans la fonction principale.
Python
class Dresseur:




```
21      def liberer_pokemon(self, pokeball):
22          if pokeball.pokemon is not None:
23              self.nettoyer_lien_pokemon_pokeball(pokeball)
24              print(f"{self.nom} a libéré {pokeball.pokemon.nom} de la Pokéball.")
25          else:
26              print("La Pokéball est déjà vide.")
27
28      def nettoyer_lien_pokemon_pokeball(self, pokeball):
29          pokeball.pokemon.pokeball = None
30          pokeball.pokemon = None
```

```
21      def liberer_pokemon(self, pokeball):
22          if pokeball.pokemon is not None:
23              print(f"{self.nom} a libéré {pokeball.pokemon.nom} de la Pokéball.")
24              self.new_method(pokeball)
25          else:
26              print
27
28      def new_method(self, pokeball):
29          pokeball.pokemon.pokeball = None
30          pokeball.pokemon = None
```

nettoyer_lien_pokemon_pokeball
Enter pour renommer, ⌘Enter pour afficher un aperçu

Pourquoi faire ça ? 1. Lisibilité : Ta méthode liberer_pokemon se lit maintenant comme un livre. 2. Réutilisation : Si tu as besoin de nettoyer les liens ailleurs (par exemple lors d'un échange), tu as déjà la méthode toute prête.

Bonus : tu remarqueras que après avoir extrait la méthode, c'est rename qui est appelé pour que tu puisses changer son nom.

Etape 5 : Test Infected