



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

Baptiste Matrat
Marie Probert

Le voyage continue !

“ Félicitations, jeune dresseur ! Tu as appris les bases dans le laboratoire de Java. Mais le monde des Pokémons 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 Pokémons, 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 :

Le voyage continue !	1
Etape 0 : Configuration de VS Code	3
Étape 1: Rappel des classes existantes	6
Étape 2 : Tests fonctionnels avec fichiers .feature et steps	8
Étape 3 : Implémentation de la classe dresseur	11
Étape 4 : Techniques de refactoring	13
Etape 4.1 : Rename	13
Etape 4.2 : extractMethod	14
Etape 5 : Test Infected	16

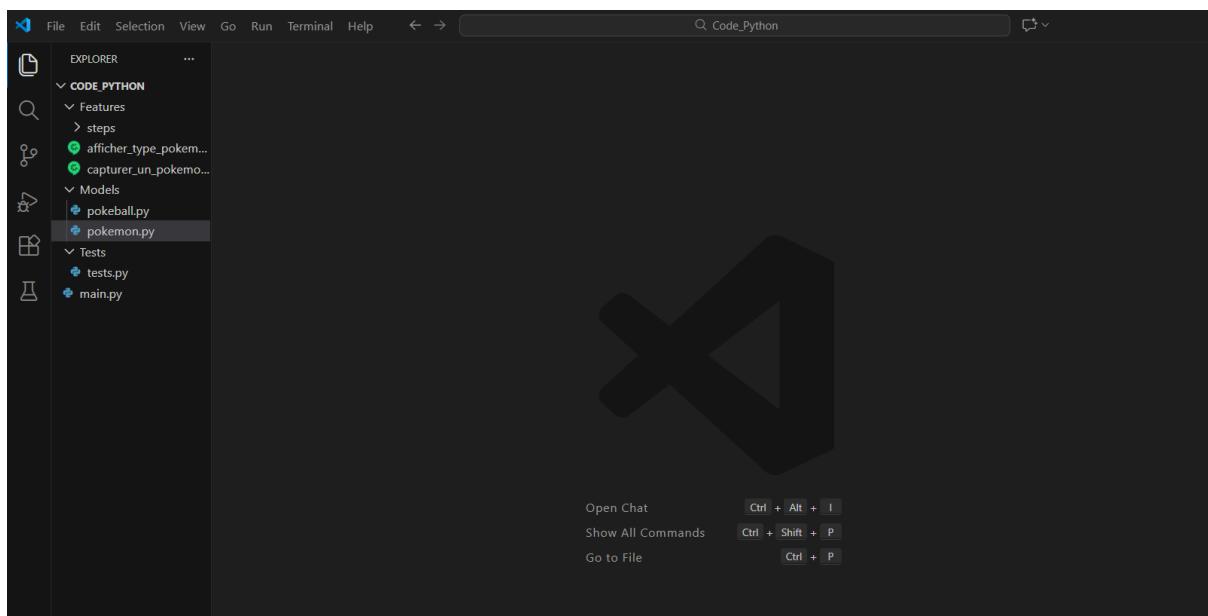


Etape 0 : Configuration de VS Code

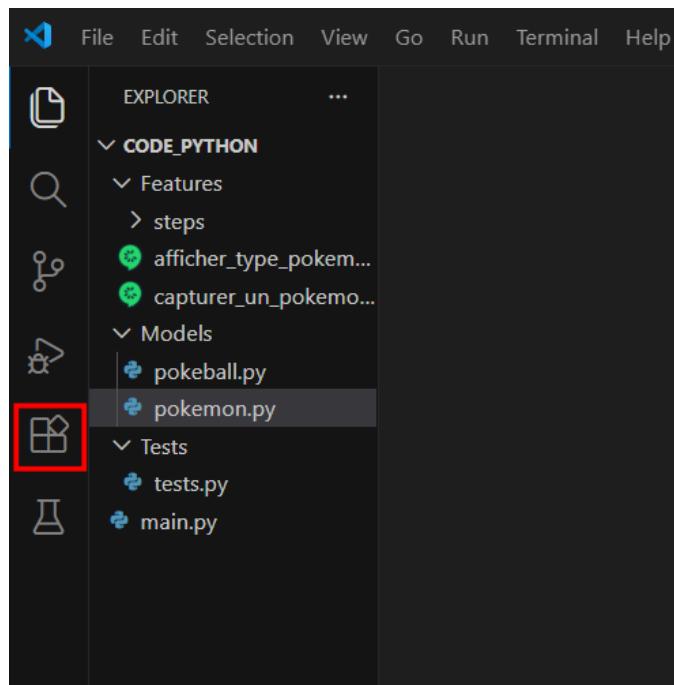
Bonjour jeune dresseur, pour ce tutoriel, nous avons changer de langage de code, avant si tu te souviens, nous utilisions Java sur BlueJ, maintenant nous allons utiliser Python pour continuer notre folle aventure.

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

EXTENSIONS: MARKETPLACE

python

Python 325ms
Python language support with...
Microsoft [Restart Extensions](#)

Python Debugger 90ms
Python Debugger extension u...
Microsoft

Pylance 588ms
A performant, feature-rich lan...
Microsoft

Python Environme... 135ms
Provides a unified python env...
Microsoft

Python Indent 16.2M ★4
Correct Python indentation
Kevin Rose [Install](#)

Python for VS... 6.4M ★2
Python language extension fo...
is Haakon Townsend [Install](#)

Python Enviro... 12.8M ★4
View and manage Python envir...
Don Jayamanne [Install](#)

Python
Microsoft [microsoft.com](#) | 196,544,142 | ★★★★★ (619)
Python language support with extension access points for IntelliSense (Pylance), Debugging (Python Debugger), lin...

[Restart Extensions](#) [Disable](#) [Uninstall](#) [Switch to Pre-Release Version](#) Auto Update

DETAILS FEATURES CHANGELOG EXTENSION PACK

Python extension for Visual Studio Code

A Visual Studio Code extension with rich support for the [Python language](#) (for all actively supported Python versions), providing access points for extensions to seamlessly integrate and offer support for IntelliSense (Pylance), debugging (Python Debugger), formatting, linting, code navigation, refactoring, variable explorer, test explorer, environment management (**NEW** Python Environments Extension).

Support for vscode.dev

The Python extension does offer some support when running on [vscode.dev](#) (which includes [github.dev](#)). This includes partial IntelliSense for open files in the editor.

Installed extensions

The Python extension will automatically install the following extensions by default to provide the best

Installation

Identifier	ms-python.python
Version	2025.26.1
Last Updated	26 seconds ago
Size	29.61MB
Cache	3.49KB

Marketplace

Published	9 years ago
Last Released	13 hours ago

Categories

Programming Languages



The screenshot shows the VS Code Marketplace interface. A search bar at the top has "Code_Python" typed into it. Below the search bar, a sidebar on the left lists extensions related to "cucumber", with one item highlighted: "Cucumber (Gherkin) Full Support" by Alexander Kechik. This highlighted item has a red border around its card. The main content area displays the details for this extension, including its name, developer, rating, and a brief description: "VSCode Cucumber (Gherkin) Full Language Support + Formatting + Autocomplete". It also shows the "Install" button, which is currently disabled. To the right of the main content, there are sections for "Installation" (listing identifier, version, last updated, and size), "Marketplace" (listing published and last released dates), and "Categories" (listing "Programming Languages").

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 Pokémons et une classe Pokeball, un Pokémons pouvant être associé à une Pokeball lors de sa capture.

Voici notre nouvelle classe Pokémons, créée là simplement dans un fichier nommé "Pokemon.py"

```
1  class Pokemon:
2      def __init__(self, type1, type2, pokeball=None):
3          self.type1 = type1
4          self.type2 = type2
5          self.pokeball = pokeball
6
7      def get_type1(self):
8          return self.type1
9
10     def set_type1(self, new_type):
11         self.type1 = new_type
12
13     def get_type2(self):
14         return self.type2
15
16     def set_type2(self, new_type):
17         self.type2 = new_type
18
19     def get_pokeball(self):
20         return self.pokeball
21
```



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

```
1  class Pokeball:
2      def __init__(self, prix, pokemon=None):
3          self.prix = prix
4          self.pokemon = pokemon
5          if pokemon is not None:
6              pokemon.pokeball = self
7      def afficher_type_pokemon(self):
8          if self.pokemon is None:
9              return "Cette Pokeball est vide. Aucun type à afficher."
10
11     t1 = self.pokemon.type1
12     t2 = self.pokemon.type2
13
14     if t1 and t2:
15         return f"De types {t1} et {t2}."
16     elif t1:
17         return f"De type {t1}."
18     else:
19         return "Le Pokémon dans cette Pokeball n'a pas de type défini."
20
21     def capturer_pokemon(self, pokemon):
22         if self.pokemon is not None:
23             raise Exception("La Pokeball contient déjà un Pokémon.")
24         if pokemon.pokeball is not None:
25             raise Exception("Le Pokémon est déjà dans une Pokeball.")
26         self.pokemon = pokemon
27         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.



```
1 Feature: Mettre un Pokéémon dans une Pokéball
2
3     En tant que dresseur Pokemon
4         Je veux mettre un Pokéémon sauvage dans une Pokéball
5             Afin de le capturer
6
7     Scenario: Mettre un Pokéémon dans une Pokéball
8         Given un Pokéémon sauvage
9             And une Pokéball vide
10            When je mets le Pokéémon dans la Pokéball
11            Then la Pokéball doit contenir le Pokéémon
12            And le Pokéémon doit référencer la Pokéball
13
14
```

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.

```
1 from behave import given, when, then
2 from Models.pokemon import Pokemon
3 from Models.pokeball import Pokeball
4
5 @given('un Pokéémon sauvage')
6 def step_given_pokemon_sauvage(context):
7     context.pokemon = Pokemon("Feu", "Spectre")
8
9 @given('une Pokéball vide')
10 def step_given_pokeball_vide(context):
11     context.pokeball = Pokeball(600)
12
13 @when('je mets le Pokéémon dans la Pokéball')
14 def step_when_mettre_pokemon(context):
15     context.pokeball.pokemon = context.pokemon
16     context.pokemon.pokeball = context.pokeball
17
18 @then('la Pokéball doit contenir le Pokéémon')
19 def step_then_pokeball_contient(context):
20     assert context.pokeball.pokemon == context.pokemon
21
22 @then('le Pokéémon doit référencer la Pokéball')
23 def step_then_pokemon_reference(context):
24     assert context.pokemon.pokeball == context.pokeball
25
```



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).

```
PROBLÈMES 2 SORTIE PORTS ... | [ ] X
> ✓ TERMINAL
ॐ baptistematrat@MacBook-Air-de-Baptiste Code_Python % behave
e:19
  Given une Pokéball sans Pokémon # features/steps/afficher_type_pokemon_
steps.py:24 0.000s
    When J'affiche le type du Pokémon via la Pokéball # features/steps/afficher_type_pokemon_
steps.py:15 0.000s
      Then afficher le message "Cette Pokeball est vide. Aucun type à afficher." # features/steps/afficher_type_pokemon_
steps.py:28 0.000s

Feature: Mettre un Pokémon dans une Pokéball # features/capturer_un_pokemon.feature:1
  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 # features/capturer_un_pokemon.feature:7
    Given un Pokémon sauvage # features/steps/capturer_un_pokemon_steps.py:5 0.000s
    And une Pokéball vide # features/steps/capturer_un_pokemon_steps.py:9 0.000s
    When je mets le Pokémon dans la Pokéball # features/steps/capturer_un_pokemon_steps.py:13 0.000s
    Then la Pokéball doit contenir le Pokémon # features/steps/capturer_un_pokemon_steps.py:18 0.000s
    And le Pokémon doit référencer la Pokéball # features/steps/capturer_un_pokemon_steps.py:22 0.000s

2 features passed, 0 failed, 0 skipped
5 scenarios passed, 0 failed, 0 skipped
20 steps passed, 0 failed, 0 skipped
Took 0min 0.001s
```



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 :

```
1  class Dresseur:
2      def __init__(self, nom):
3          self.nom = nom
4          self.pokeballs = []
5
6      def get_nom(self):
7          return self.nom
8
9      def get_pokeballs(self):
10         return self.pokeballs
11
12     def ajouter_pokeball(self, pokeball):
13         self.pokeballs.append(pokeball)
14
15     def capturer_pokemon(self, pokemon, pokeball):
16         if pokeball.capturer_pokemon(pokemon):
17             print(f"{self.nom} a capturé {pokemon.nom} dans une Pokéball!")
18         else:
19             print(f"La Pokéball est déjà pleine ou la capture a échoué.")
20
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
26         else:
27             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.

```
55  class TestDresseur(unittest.TestCase):
56      def setUp(self):
57          self.dresseur = Dresseur("Sacha")
58          self.pokemon = Pokemon("Pikachu", "Électrik", None)
59          self.pokeball = Pokeball(prix=200)
60
61      def test_modification_nom(self):
62          self.dresseur.nom = "Régis"
63          self.assertEqual(self.dresseur.nom, "Régis")
64
65      def test_ajouter_pokeball(self):
66          self.dresseur.ajouter_pokeball(self.pokeball)
67          self.assertIn(self.pokeball, self.dresseur.pokeballs)
68          self.assertEqual(len(self.dresseur.pokeballs), 1)
69
70      def test_capturer_pokemon_succes(self):
71          print("--- Test de capture de Pokémon ---")
72          self.dresseur.capturer_pokemon(self.pokemon, self.pokeball)
73          self.assertEqual(self.pokeball.pokemon, self.pokemon)
74          self.assertEqual(self.pokemon.pokeball, self.pokeball)
75
76      def test_liberer_pokemon(self):
77          self.dresseur.capturer_pokemon(self.pokemon, self.pokeball)
78
79          self.dresseur.liberer_pokemon(self.pokeball)
80          self.assertNone(self.pokeball.pokemon)
81          self.assertNone(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".

```
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.



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

    def get_n inventaire
    return Enter pour renommer, ⌘Enter pour afficher un aperçu
```

```
1  class Dresseur:
2      def __init__(self, nom):
3          self.nom = nom
4          self.inventaire = []
5
6      def get_nom(self):
7          return self.nom
8
9      def get_inventaire(self):
10         return self.inventaire
11
12     def ajouter_pokeball(self, pokeball):
13         self.inventaire.append(pokeball)
14
15     def capturer_pokemon(self, pokemon, pokeball):
16         if pokeball.capturer_pokemon(pokemon):
17             print(f"{self.nom} a capturé {pokemon.nom} dans une Pokéball!")
18         else:
19             print("La Pokéball est déjà pleine ou la capture a échoué.")
20
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
26         else:
27             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 libérer_pokémon(self, pokéball):
22         if pokéball.pokémon is not None:
23             print(f"{self.nom} a libéré {pokéball.pokémon.nom} de la Pokéball.")
24             pokéball.pokémon.pokéball = None
25             pokéball.pokémon = 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:

```
23     print(f"{self.nom} a libéré {pokéball.pokémon.nom} de la Pokéball.")
24     pokéball.pokémon.pokéball = None
25     pokéball.pokémon = None
26
27     Extraire
28     ↗ Méthode d'extraction
29
30     Réécrire
31     ↗ Modifier
32     ↗ Vérifier
```

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

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

Pourquoi faire ça ? 1. Lisibilité : Ta méthode libérer_pokémon 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