Agilité : Développement et tests unitaires

TP2 Agilité

Développement du simulateur de forêt

Jérôme Buisine
jerome.buisine@univ-littoral.fr

22 septembre 2022

Durée : 3 heures

L'objectif de ce TP est la mise en production de la première phase (sprint 1) de développement de notre simulateur d'évolution de forêt. Ce sprint concernera la mise en place de la structure basique d'un arbre et du monde dans lequel il peut évoluer.

1 Travail

Ce support est le premier d'une suite de travaux pratiques qui ont pour objectif la mise en place d'un simulateur d'évolution de forêt. Vous allez dans ce TP développer en respectant des consignes spécifiques qui sont les suivantes :

— 1. Créer un milestone sur Gitlab pour ce TP, nommé « tree management » ;

— 2. Au sein de ce milestone définir l'ensemble des tâches de développement (important donc de lire l'ensemble du sujet pour comprendre ces différentes tâches) ;

— 3. Pour chaque tâche développée (avec sa branche **`feature/Sprint1_TaskX`** spécifique), vérifier la qualité de votre code avec `pylint` puis réaliser les tests unitaires et fonctionnels de couverture ;

— 4. Si la couverture de code est correcte, c'est-à-dire avec un minimum de pourcentage de couverture $> 80\%$ (à vérifier localement), alors un commit est à réaliser ;

— 5. Soumettre en ligne la modification apportée et vérifier que le pipeline d'intégration continue fonctionne correctement. Vérifiez également que la qualité du code est préservée. Si besoin réaliser un commit avec des correctifs ;

Pour vous aider : voici un ensemble de ressources qui peuvent vous être utiles pour ce TP :

— 1. Documentation officielle de l'outil `Git` ;

— 2. Gitlab : interface web pour la gestion de projets versionnés sous `Git` ;

— 3. pyenv : permet la gestion d'environnement Python ;

— 4. pylint : permet l'analyse de code relative à des conventions du langage ;

— 5. pytest : permet d'écrire facilement des tests simples et peut évoluer pour prendre en charge des tests fonctionnels complexes.

Remarque importante : faites attention au copier/coller à partir du PDF qui peut prendre en compte des caractères inattendus et causer des erreurs.

2 Conventions adoptées pour le projet

2.1 Configuration du pipeline CI/CD

En l'état, la version Sonarqube community que nous utilisons ne permet pas de spécifier une branche à analyser. Nous sommes limités à la branche `master`.

De ce fait, on précisera au sein de la configuration CI/CD de Gitlab que :

— Les rapports Sonarqube sont à générer uniquement pour les branches `master` et `develop`. Même si les rapports ne sont pas visibles pour la branche `develop`, le code reste analysé ;

— Faire attention à ce que le job de Sonarqube soit réalisé avant ceux de déploiements `fly.io` ;

— L'instruction `allow_failure` associée à Sonarqube sera à supprimer. On souhaite que les déploiements sur l'environnement de développement et production sur `fly.io` ne soient pas effectués si la « *Qualite Gate* » de Sonarqube n'est pas valide.

2.2 Les bonnes pratiques en Python

L'ensemble du projet sera donc développé en Python. Pour ceux qui souhaient revoir les bases de la programmation orientée objet en Python, un document est disponible à cet effet.

Ce document reprend également les bonnes pratiques de gestion des imports des modules d'une librairie.

2.3 Conventions adoptées par le projet `treevolution`

Avant toute chose, il faut au sein d'un projet définir les conventions de développement. Voici les différentes conventions demandées :

— Le nom du fichier de classe en minuscule et nom de la classe en « pascal case » ;

— Utilisation du « snake case » comme convention de nommage des variables et fonctions/méthodes ;

— Les attributs d'instance possèdent un « _ ». Par exemple : « `self._mon_attribut` » ;

— Noms des classes, attributs et fonctions/méthodes en anglais. La documentation peut elle rester en français si l'anglais reste un frein ;

— Les variables de classe sont en majuscules (voir des exemples dans le code déjà mis à disposition) ;

— Chaque classe possédera sa méthode `__str__` pour proposer l'affichage d'une instance ;

— Exploiter au maximum la metadata « `property` » pour les getteurs/setters d'attributs d'une classe (notamment si l'attribut est abstrait et spécialisé) ;

— Ajouter la docstring pour chaque : module/classe/fonction/méthode... ;

— Plus généralement, respecter les conventions de bonne conduites de codage proposées par la *PEP* (Python Enhancement Proposals) et qui seront à vérifier par l'outil pylint.

3 Développements

Tâche 1 : Représentation d'un monde (`world`) définit par les fonctionnalités suivantes :

— est composé d'une hauteur (`height`) ;

— est composé d'une largeur (`width`) ;

— d'une date de début (`start_date`) ;

— d'une méthode `step` qui avance la date du monde d'un jour ;

— d'une méthode `date` (potentiellement `property`) qui retourne le jour courant du monde.

Indications :

- La classe `World` sera définit dans le module `world` présent à la racine du package ;
- La gestion des dates sera réalisée avec la librairie `datetime` de Python et notamment la fonction `timedelta`.

À tester : `world_test.py`

→ Vérifier que la longueur et hauteur du monde est bien accessible ;

→ Vérifier que la méthode `step` fonctionne correctement (utiliser la méthode `date`).

Tâche 2 : Un arbre sera tout d'abord représenté de manière abstraite :

— Créer une classe `Tree` dans le module `models.tree` avec un constructeur composé de 3 paramètres : `coordinate` (un point), `birth` (la hauteur de l'arbre) et `world` (le monde auquel il sera associé) ;

— Ce constructeur définira plusieurs attributs :

- `specie` : qui contient le nom de la classe de l'instance courante ;
- `height` : la hauteur de l'arbre fixée par défaut à 0 ;
- `coordinate` : la position actuelle de l'arbre dans le monde à sa naissance ;
- `nutrient` : un indicateur de nutrition de l'arbre, initialisée à 100 ;
- `fallen` : spécifie si l'arbre a chuté ou non, initialisée à `False` ;
- `birth` : date de naissance de l'arbre ;
- `age` : l'âge courant de l'arbre, pour le moment 0 ;
- `max_age` : indicateur de l'âge maximal de l'arbre courant et spécifique aux capacité de l'espèce. Par défaut fixé à `None` ;
- `world` : le monde auquel l'arbre est associé ;
- `days_in_humus` : le nombre de jours où l'arbre sera en état `HUMUS` après sa chute (âge maximal atteint ou chute). Par défaut inconnu, soit `None`.

— Seul les attributs `height` et `fallen` peuvent être modifiées depuis l'extérieur ;

— Plusieurs `property` abstraites sont à définir dans la classe `Tree` :

- `health` : qui déterminera en fonction de plusieurs critères propres à l'espèce l'état de santé de l'arbre ;
- `width` : qui déterminera la largeur du tronc de l'arbre pour chaque espèce.

— Une méthode abstraite `evolve` qui permet de préciser comment évolue chaque arbre jour après jour. Pour cela, cette méthode prend en paramètre un contexte d'environnement (définit dans `context.context`).

Puis spécifiquement pour chaque espèce, avec comme premier exemple :

— Créer une classe `Oak` qui hérite de `Tree`. Elle sera définie dans le module `models.trees.oak` ;

— Des variables de classes sont définies pour borner l'âge maximal et la taille maximale de l'arbre à son initialisation. Chaque valeur respective est choisie aléatoirement et uniformément, avec :

```
MIN_HEIGHT, MAX_HEIGHT = 5, 7
MIN_AGE, MAX_AGE = 5, 10
```

— Tant que l'arbre n'a pas dépassé sa taille maximale, alors il évolue de 0.005 par jour ;

— Pour le moment la santé de l'arbre est définie par sa valeur nutritionnelle actuelle ;

— La largeur du tronc est quand à elle définie par : $height * 0.08$.

Indications :

- Suivez au maximum les indications proposées ;
- Utiliser la fonction `type` de Python pour connaître le type d'une instance ;
- Utilisez au maximum les `metadata` proposées par Python.

À tester : `test_tree.py`

→ Vérifier que la méthode `evolve` fonctionne correctement en proposant un contexte par défaut pour chaque jour simulé ;

→ Vérifier que la largeur du tronc et la hauteur de l'arbre soit respectée.

Tâche 3 : Gestion de l'âge maximal atteint d'un arbre.

— Au sein de la méthode abstraite `evolve` de `Tree`, déterminer un comportement commun aux arbres : l'évolution de l'âge en fonction de la date courante. Appeler en début de la méthode spécifiée `evolve` par `Oak`, la méthode mère à l'aide de l'instruction `super()` ;

— Définir la méthode `state` comme `property` qui retourne l'état d'un arbre en fonction de son âge (`HUMUS` ou `TREE`, état définit dans `models.state.TreeState`). Faites le choix judicieux de l'ajout de cette méthode à la classe attendue ;

— Un arbre n'évolue plus si son état atteint un stade `HUMUS` et est considéré comme tombé (`fallen`).

Indications :

- Utilisez la fonction `relativedelta` proposée par la librairie `datetime` de Python.

À tester : `tree_test.py`

→ Vérifier que l'état d'un arbre change une fois son âge maximal atteint et qu'il est considéré comme tombé ;

→ Vérifier que l'arbre n'évolue plus une fois son âge maximal atteint.

Tâche 4 : Actuellement la gestion de l'évolution d'un arbre se fait en le simulant manuellement (en appellant la méthode `evolve`). Nous allons maintenant faire en sorte que ce soit notre représentation du monde qui simule l'évolution des arbres.

— Définir une liste d'arbres comme attribut d'instance de `World` (par défaut vide) ;

— Définir une méthode dans la classe `World` qui ajoute un arbre à cette liste des arbres connus ;

— La méthode `step` de `World` va maintenant simuler l'ensemble des arbres. Pour chaque jour un temps météo aléatoire est demandé basé sur la date du jour. Pour qu'un arbre soit simulé, on considère un contexte sans filtre d'éclairage et une quantité de humus fixée à 0. La méthode `step` retournera un tuple comprenant : la date actuelle, la météo du jour simulée, la liste des arbres ;

— Un attribut de `World` va maintenant stocker la dernière météo obtenue ;

— À partir du précédent attribut, définir une méthode `state`, qui sans simuler retourne l'état du monde tout comme `step`.

À tester : `world_test.py`

→ Fixez une graine aléatoire de la librairie random à 42. Cela vous permet de toujours posséder un cas attendu de simulation ;

→ Proposez de simuler un monde composé deux arbres de type `Oak` sur 1000 jours dont les points de coordonnées sont fixés aléatoirement ;

→ Vérifiez que la date de la simulation est bien celle attendue ;

→ Vérifiez que les arbres ont bien l'âge attendu.

Pour cela, vous pouvez vous inspirer du fichier `src/main.py` pour proposer une simulation ne prenant pas encore en compte les graines d'arbres. Par exemple, en fixant :

— La graine aléatoire à 42 ;

— Une date de départ fixée à « 2022-09-10 » ;

— Un monde de taille 200×200 ;

— Un nombre d'arbres de 2 placés aléatoirement ;

— Une simulation de 1000 jours.

À ce stade de développement et avec les mêmes paramètres, vous devriez obtenir :

```
At 2025-06-06 00:00:00:
-- (name: Oak, height: 5.00, width: 0.40, coordinate: (x: 147.29, y: 135.34),
        health: 100.00, nutrient: 100.00, age: 2, max_age: 9.46, humus_days: None)
-- (name: Oak, height: 5.00, width: 0.40, coordinate: (x: 84.38, y: 5.96),
        health: 100.00, nutrient: 100.00, age: 2, max_age: 6.09, humus_days: None)
```

Tâche 5 : Gestion du vieillissement d'un arbre.

— Lorsqu'un arbre a atteint son âge maximal, il chute. Il faudra mettre à jour son état en conséquence ;

— À la chute d'un arbre, on considère qu'il passe à un stade de humus. Il moisit tout simplement et devient ressource pour toute espèce vivante avoisinante. Nous déterminons le nombre de jour de humus disponible par l'entier obtenu par le produit de la largeur et du carré de la hauteur de l'arbre ($humus = width * height^2$) ;

— De manière commune à tous les arbres, le nombre de jours en état humus est décompté de jour en jour, mais ne peut pas être inférieur à 0 ;

— Déterminer une méthode `consumed` qui permet de savoir si un arbre à été consumé. Il est considéré comme consumé seulement s'il est tombé et que le nombre de jour en stade humus a été atteint ou dépassé ;

— Lorsqu'un arbre est consumé, il est supprimé de la représentation du monde.

Indications :

- Faire attention à ne fixer qu'une seule fois (au moment de la chute) le nombre de jours que doit passer l'arbre en état humus.

À tester : `world_test.py`

→ Toujours en fixant un contexte de simulation (seed aléatoire), vérifiez la suppression d'un arbre une fois son âge maximal atteint et sa transition en état humus effectuée ;

Tâche 6 : Prise en compte de la météo pour l'amélioration de l'évolution d'un arbre de type `Oak`.

- Un arbre dans la nature n'a pas tendance à croître linéairement dans le temps comme le fait notre espèce `Oak` actuellement. Ajoutez une méthode `youth_ratio`, qui fournit un ratio de jeunesse d'un arbre déterminé par : $1 - (age/age^{max})$. Ce ratio permet de simuler que l'arbre grandit beaucoup plus quand il est jeune que le contraire ;
- Exploiter ce ratio de sorte à ce que l'évolution de la hauteur de l'arbre soit de $0.005 * youth$ par jour ;
- Chaque jour, la nutrition de l'arbre baisse de 0.2 ;
- Inversement, la nutrition de l'arbre est renforcée par l'humidité présente dans la journée, qu'elle se voit ajouter. De plus, la quantité/présence de humus dans le contexte permet d'obtenir un bonus de nutrition fixé par le produit de l'humidité et de la quantité de humus présent. On considère que le humus en plus d'être nutritif, emmagasine de l'eau. À noter que ce bonus pour le moment n'apporte rien (le humus vaut par défaut 0 dans un contexte), mais permet déjà de se fixer d'une idée de comment il sera activé quand la notion de humus disponible sera géré ;
- La valeur de nutrition d'un arbre ne peux pas être inférieur à 0 ou excéder 100, soit bornée dans l'intervalle $[0, 100]$.

À tester : `tree_test.py`

→ Modifier si besoin les tests précédents suite à cette modification ;

→ Vérifiez la valeur de nutrition de l'arbre (notamment en fin de vie).

En l'état le simulateur devrait proposer une première version stable. Certains éléments simulés ne sont pas totalement complets, mais fonctionnels. Il s'agit du principe du livrable qui permet de fournir un logiciel opérationnel, de qualité et vérifié. Les prochains TPs, permettront d'ajouter de nouvelles fonctionnalités telles que la gestion de branches et de graines d'arbres (propagation d'une espèce). Ainsi que tout ce qui en découle, comme l'ombre qu'un arbre peut faire à un autre, la consommation de nutriment pour évoluer, l'impact du vent...

À ce stade, si vous simulez une forêt de 5 arbres avec le contexte suivant :

- La graine aléatoire fixée à 42 ;
- Une date de départ fixée à « 2022-09-10 » ;
- Un monde de taille 200×200 ;
- Les arbres placés aléatoirement ;
- Une simulation de 3000 jours.

Avec un affichage des arbres tous les 1000 jours simulés, vous devriez obtenir (ou être proche de) :

```
At 2025-06-06 00:00:00:
-- (name: Oak, height: 4.52, width: 0.36, coordinate: (x: 147.29, y: 135.34),
        health: 92.67, nutrient: 92.67, age: 2, max_age: 9.46, humus_days: None)
-- (name: Oak, height: 4.26, width: 0.34, coordinate: (x: 84.38, y: 5.96),
        health: 92.67, nutrient: 92.67, age: 2, max_age: 6.09, humus_days: None)
-- (name: Oak, height: 4.45, width: 0.36, coordinate: (x: 5.31, y: 39.77),
        health: 92.67, nutrient: 92.67, age: 2, max_age: 8.25, humus_days: None)
-- (name: Oak, height: 4.50, width: 0.36, coordinate: (x: 44.09, y: 117.85),
        health: 92.67, nutrient: 92.67, age: 2, max_age: 9.05, humus_days: None)
-- (name: Oak, height: 4.32, width: 0.35, coordinate: (x: 161.16, y: 139.63),
```

```
        health: 92.67, nutrient: 92.67, age: 2, max_age: 6.70, humus_days: None)
At 2028-03-02 00:00:00:
-- (name: Oak, height: 5.18, width: 0.41, coordinate: (x: 147.29, y: 135.34),
        health: 100.00, nutrient: 100.00, age: 5, max_age: 9.46, humus_days: None)
-- (name: Oak, height: 6.01, width: 0.48, coordinate: (x: 84.38, y: 5.96),
        health: 100.00, nutrient: 100.00, age: 5, max_age: 6.09, humus_days: None)
-- (name: Oak, height: 6.09, width: 0.49, coordinate: (x: 5.31, y: 39.77),
        health: 100.00, nutrient: 100.00, age: 5, max_age: 8.25, humus_days: None)
-- (name: Oak, height: 5.02, width: 0.40, coordinate: (x: 44.09, y: 117.85),
        health: 100.00, nutrient: 100.00, age: 5, max_age: 9.05, humus_days: None)
-- (name: Oak, height: 5.31, width: 0.42, coordinate: (x: 161.16, y: 139.63),
        health: 100.00, nutrient: 100.00, age: 5, max_age: 6.70, humus_days: None)
At 2030-11-27 00:00:00:
-- (name: Oak, height: 5.18, width: 0.41, coordinate: (x: 147.29, y: 135.34),
        health: 91.68, nutrient: 91.68, age: 8, max_age: 9.46, humus_days: None)
-- (name: Oak, height: 6.09, width: 0.49, coordinate: (x: 5.31, y: 39.77),
        health: 91.68, nutrient: 91.68, age: 8, max_age: 8.25, humus_days: None)
-- (name: Oak, height: 5.02, width: 0.40, coordinate: (x: 44.09, y: 117.85),
        health: 91.68, nutrient: 91.68, age: 8, max_age: 9.05, humus_days: None)
```

---

4 Remise des travaux

N'oubliez pas de réaliser un dernier commit de vos travaux. Taguez ensuite votre projet avec le tag « tp2 » et soumettez-le sur le serveur Gitlab. Vérifiez que le pipeline reste fonctionnel. Cette version de projet fera office de rendu.