Algorithmique en Python

Objectifs

- Savoir créer un environnement virtuel
- Savoir tester une fonction avec pytest
- Comprendre les couvertures de test avec coverage
- Comprendre les limites d'un langage typé seulement dynamiquement
- Savoir utiliser les structures de contrôle Python
- Savoir utiliser la méthode des raffinages
- Savoir utiliser les exceptions

1 Création d'un environnement virtuel Python

Exercice 1 L'outil virtualenv permet de créer un environnement Python propre à un projet, avec ses propres modules installés, indépendamment de ceux installés sur le système.

Voici les étapes pour créer une environnement virtuel dans le dossier `~/nosave/pyenv` :

```
1 which python3                # connaître l'emplacement de python3
2 mkdir -p $HOME/nosave/pyenv  # le dossier qui sera notre environnement virtuel
3 virtualenv -p $(which python3) $HOME/nosave/pyenv    # création de l'environnement
4 ${HOME}/nosave/pyenv/bin/python --version            # python3, normalement !
```

Pour l'activer, il suffit alors de faire :

```
1 source ${HOME}/nosave/pyenv/bin/activate
2 which python                 # Celui de pyenv !
3 which pip                    # Aussi celui de pyenv
```

L'outil `pip` permet d'installer des paquets Python. Par exemple, pour installer l'outil de test pytest, on peut faire :

```
pip install pytest
```

Pour désactiver l'environnement virtuel Python, il suffit de faire :

```
deactivate
```

Activer l'environnement oblige à taper une commande un peu longue. On peut définir un alias pour le faire plus rapidement :

```
alias p3="source ${HOME}/nosave/pyenv/bin/activate"
```

Pour que cet alias soit permanent, on peut l'ajouter dans le fichier `${HOME}/.bashrc`.

2 Comprendre les sous-programmes et leur test

Exercice 2 : Pourquoi et comment tester
Le langage Python s'appuie sur un typage dynamique. Ceci signifie que quand un programme Python est chargé par l'interpréteur, aucune vérification du programme n'est faite à part le respect de la syntaxe Python. Il est donc d'autant plus important de tester son programme et de s'assurer que l'ensemble de ses instructions ont été exécutées au moins une fois. Ceci ne garantira pas que le programme est correct mais permettra de détecter des erreurs !

Considérons la fonction du fichier `minimum.py` qui calcule le minimum de ses deux paramètres. Elle contient manifestement une erreur que nous ne corrigerons pas maintenant.

Le fichier `test_minimum.py` est un programme de test pytest de la fonction minimum. pytest considère les fichiers dont le nom commence par `test_` (ou se termine par `_test`) comme des programmes de test et, dans ces fichiers, les fonctions dont le nom commence par `test` sont des fonctions de test. Les tests s'appuient sur l'instruction `assert` qui lève une exception si l'expression booléenne est fausse et ne fait rien sinon.

2.1. *Installation de pytest.* L'outil *pytest* ne fait pas partie de la livraison standard de Python. On peut l'installer grâce à l'installateur de paquets fourni avec Python : *pip*. On peut installer *pytest* en faisant [1] :

```
pip install pytest
```

2.2. *Exécution des tests.* Exécuter [2] le programme de test avec `pytest`. Par exemple :

```
pytest test_minimum.py
```

Est-ce que le test réussi ?

2.3. *Couverture des instructions.* Un outil de couverture de code permet de recenser les instructions qui ont été exécutées et celles qui ne l'ont pas été pendant une exécution donnée. C'est donc un outil qui nous donne des indications pour identifier de nouveaux tests à faire en s'appuyant sur la structure du programme. On parle de tests structurels ou tests en boîte blanche.

Si `coverage` n'est pas déjà installé, faire : `pip install coverage`

Pour exécuter les tests sous *coverage*, on peut faire [3] :

```
coverage run -m py.test test_minimum.py
```

Les résultats sont conservées dans un fichier caché (.coverage sous Linux). La commande suivante [4] permet d'engendrer un rapport au format texte [5] :

1. On peut aussi faire : `python -m pip install pytest`. Cette version a pour avantage d'utiliser la version courante de python pour installer les nouveaux paquets.
2. Si `pytest` n'est pas déjà installé, il faut faire `pip install pytest`.
3. `-m` demande à Python d'exécuter le module dont le nom suit. `py.test` est le module qui correspond à `pytest`.
4. L'option `-m` affiche aussi les numéros des lignes manquantes.
5. On peut aussi engendrer un rapport au format HTML avec `coverage html minimum.py` qui engendre le dossier `htmlcov`. Il suffit alors d'ouvrir le fichier `htmlcov/index.html` avec un navigateur. On peut alors cliquer sur la ligne `minimum.py` pour consulter les résultats concernant ce module.

```
coverage report -m minimum.py
```

Consulter les résultats.

2.4. *Couverture des enchaînements.* En plus des instructions, `coverage` peut aussi analyser les enchaînements. Par exemple, pour **`if`** (même sans **`else`**), il faudrait faire un test pour lequel la condition est vraie et l'autre où elle est fausse. Ainsi, on teste les deux enchaînements possibles après l'évaluation de la condition du **`if`**.

Pour demander aussi l'analyse des enchaînements, il faut ajouter l'option `--branch` :

```
coverage run --branch -m py.test test_minimum.py
```

Bien sûr, il faut ré-engendrer les résultats :

```
coverage report -m minimum.py
```

Consulter les résultats et constater que la couverture du **`if`** est marquée partielle. Comment interpréter les numéros affichés à droite ?

2.5. *Exploiter les résultats de couverture.* Compléter les tests pour atteindre un taux de couverture de 100% des instructions et branchements. Que constate-t-on ? Que faire ?

2.6. *Complétude.* Est-ce qu'un taux de couverture de 100% des instructions et enchaînements avec des tests qui réussissent est suffisant pour conclure que le programme est correct ?

3 Algorithmique en Python

Exercice 3 : Tarif d'une place

Le tarif normal de la place est de 12,60 euros. Les enfants (moins de 14 ans) paient 7 euros. Les séniors (65 ans et plus) paient 10,30 euros. Étant donné son age, combien une personne doit-elle payer ? Compléter le fichier `tarif_place.py`.

Exercice 4 : Table de 7

Afficher la table de multiplication de 7 sous la forme :

```
7 x 1 =  7
7 x 2 = 14
...
7 x 9 = 63
```

Compléter le fichier `table7.py`.

Exercice 5 : Afficher les cubes consécutifs

Afficher les cubes des entiers de `debut` à `fin`.

Exemple : Si `debut` vaut 2 et `fin` vaut 4, le programme affiche :

```
8
27
64
```

Exercice 6 : Cubes dont la somme est inférieure à une limite
Afficher, dans l'ordre croissant, les cubes des entiers strictement positifs à condition que leur somme soit inférieure à une limite.

Exemples :
Si `limite` vaut 20, le programme affichera :

```
1
8
```

Si `limite` vaut 36, le programme affichera :

```
1
8
27
```

Exercice 7 : Suite de Fibonacci
Les termes de la suite de Fibonacci sont définis par la relation de récurrence suivante :

$fib(0) = 0$
$fib(1) = 1$
$fib(n) = fib(n-1) + fib(n-2)$ si $n \geq 2$

7.1. Écrire le corps de la fonction `fib` (fichier `fibonacci.py`) qui retourne le n^{e} terme de la suite de Fibonacci.
7.2. Écrire le corps de la fonction `rang_sup` qui retourne le rang du premier terme de la suite de Fibonacci qui est supérieur ou égal à une valeur donnée. On suppose cette valeur strictement positive.
7.3. Utiliser le fichier `test_fibonacci.py` pour tester ces deux fonctions.

4 Raffinage en Python

Exercice 8 : Jeu du devin
Le jeu du devin se joue à deux joueurs. Le premier joueur choisit un nombre compris entre 1 et 999. Le second doit le trouver en un minimum d'essais. À chaque proposition, le premier joueur indique si le nombre proposé est plus grand ou plus petit que le nombre à trouver. En fin de partie, le nombre d'essais est donné.
8.1. *La machine fait deviner le nombre.* Écrire un programme dans lequel la machine choisit un nombre et le fait deviner à l'utilisateur. Bien sûr, pour écrire ce programme on appliquera la méthode des raffinages.
8.2. *La machine joue.* Écrire un programme dans lequel l'utilisateur choisit un nombre et la machine doit le trouver. Pour chaque nombre proposé, l'utilisateur indique s'il est trop petit (`'p'` ou `'P'`), trop grand (`'g'` ou `'G'`) ou trouvé (`'t'`, `'T'`). Bien sûr, pour écrire ce programme on appliquera la méthode des raffinages.
Indication : On utilisera une recherche par dichotomie pour trouver le nombre.

8.3. *Le programme complet.* Écrire le programme de jeu qui donne le choix à l'utilisateur entre deviner ou faire deviner le nombre.
8.4. *On peut recommencer.* Compléter le programme précédent pour que l'ordinateur propose de faire une nouvelle partie lorsque la précédente est terminée.

Voici un exemple d'utilisation du jeu complet.

```
1- L'ordinateur choisit un nombre et vous le devinez
2- Vous choissez un nombre et l'ordinateur le devine
0- Quitter le programme
Votre choix : 1

J'ai choisi un nombre compris entre 1 et 999.
Proposition 1 : 900
Trop petit !
Proposition 2 : 10000
Trop grand !
Proposition 3 : 990
Trop grand !
Proposition 4 : 988
Bravo ! Vous avez trouvé en 4 essais.

1- L'ordinateur choisit un nombre et vous le devinez
2- Vous choissez un nombre et l'ordinateur le devine
0- Quitter le programme
Votre choix : 2

Avez-vous choisi un nombre compris entre 1 et 999 (o/n) ? n
J'attends...
Avez-vous choisi un nombre compris entre 1 et 999 (o/n) ? o
Proposition 1 : 500
Trop (g)rand, trop (p)etit ou (t)rouvé ? g
Proposition 2 : 250
Trop (g)rand, trop (p)etit ou (t)rouvé ? x
Je n'ai pas compris la réponse. Merci de répondre :
   g si ma proposition est trop grande
   p si ma proposition est trop petite
   t si j'ai trouvé le nombre
Trop (g)rand, trop (p)etit ou (t)rouvé ? t
J'ai trouvé en 2 essais.

1- L'ordinateur choisit un nombre et vous le devinez
2- Vous choissez un nombre et l'ordinateur le devine
0- Quitter le programme
Votre choix : 0
```

```
Au revoir...
```

5 Utilisation de repl.it

Exercice 9 : Utiliser repl.it
On peut utiliser repl.it (`https://repl.it/`) pour avoir un environnement de développement en ligne qui propose également un terminal avec un interpréteur de commande de type *bash*.

Même si ce n'est pas obligatoire, il est conseillé de créer un compte. L'intérêt est de retrouver ses projets (ses *repls*) entre deux utilisations de cette application (dans *My repls*). Sans compte créé, les fichiers seront effacés d'une utilisation à l'autre.

Une fois sur le site, on peut faire « start coding » en haut à droite, choisir Python et faire « create repl ».

Pour ouvrir le terminal *bash*, on peut faire F1 puis taper « Open Shell ». Si F1 ne fonctionne pas, on peut faire Ctrl-Shift-S (le raccourci) ou faire un clic droit dans l'éditeur (main.py) et choisi « command palette F1 » et taper « Open Shell ».

La figure 1 montre ce à quoi ressemble l'environnement repl.it avec :

- au centre, l'éditeur (avec le fichier `main.py` ouvert),
- à droite et en haut, l'interpréteur Python. Quand on clique sur Run le fichier `main.py` est exécuté dans cet interpréteur. On voit sur la première ligne le résultat de l'exécution du programme (« premier programme »). Sur la ligne suivante, on a appelé de nouveau la fonction `main`.
- à droite en bas, le terminal avec l'interpréteur de commandes *bash*. La première commande est le lancement du programme (`python main.py`) suivie du résultat de l'exécution

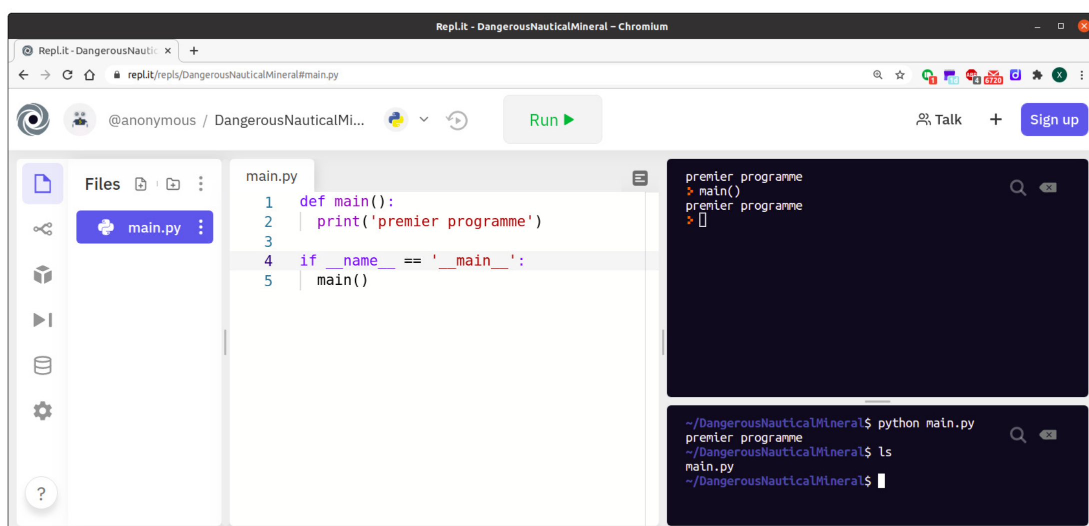

FIGURE 1 – L'environnement repl.it pour Python avec le shell ouvert

et la seconde (`ls`) qui affiche le contenu du dossier courant.

9.1. Créer un repl pour Python.
9.2. Saisir le programme qui est présent sur la figure 1.
9.3. L'exécuter en faisant « Run ».
9.4. Appeler la fonction `main()` depuis l'interpréteur Python.
9.5. Ouvrir le l'interpréteur de commande *bash*.
9.6. Exécuter le programme depuis l'interpréteur de commande *bash* en tapant :

```
python main.py
```