TP 1: mise en place de tests unitaires

1 Configuration générale pour le TP

Q1.1 - Vérifier l'accès à son compte Github, puis y créer un dépot nommé `tp-r504`. Dans les options, le laisser en public et valider la création automatique du fichier "readme.md".

Q1.2 - Dans la machine Linux, cloner le dépot dans votre "home" avec le protocole `git` :

```
$ cd ~; git clone git@github.com:NOM/PROJET.git
```

Q1.3 - Se placer dans le dossier du projet et créer un dossier `TP1`, dans lequel vous créerez un fichier `prog1.py` contenant ceci :

```
print("Hello, World!")
```

Q1.4 - Vérifiez le fonctionnement avec : `$ python3 prog1.py`

Q1.5 - Ajouter ce fichier aux fichiers suivis avec : `$ git add *.py`

Q1.6 - Vérifier avec `$ git status` que le fichier est bien dans la zone d'index ("*staging area*").

Q1.7 - Enregistrer les changements localement : `$ git commit -m "création fichier"`
(Note : git va vous demander préalablement de configurer votre compte avec un email et un nom, et va vous donner la syntaxe pour le faire)

Q1.8 - Pousser les changements au repo distant avec : `$ git push`

2 Création d'une fonction

Q2.1 - Ajouter les lignes permettant de saisir un nombre au clavier et d'afficher son carré, et vérifier le fonctionnement.

Q2.2 - Insérer ceci dans une boucle infinie, dont on ne pourra sortir que avec CTRL-C (utiliser "`while True`").

Q2.3 - Dans un **autre fichier** nommé `fonctions.py`, créer une fonction `puissance` prenant 2 arguments entiers a et b et qui va renvoyer le nombre entier a^b.
(note : pour l'instant, le calcul sera fait avec la notation `a**b`).

Q2.4 - Modifier le programme principal : faites la saisie de deux nombres et afficher le résultat du 1er élevé à la puissance du second, en appelant la fonction ci-dessus.
Il faudra ajouter une ligne en-tête du fichier indiquant que vous allez utiliser le code qui se trouve dans le fichier `fonctions.py` :
`import fonctions as f`
Vous pourrez ensuite appeler cette fonction en utilisant "f" comme identifiant :
`res = f.puissance(a,b)`
Vérifier le fonctionnement en exécutant le programme : `$ python3 prog1.py`

Q2.5 - Dans la fonction puissance(), ajouter le code qui va vérifier que les deux arguments sont entiers, et qui va lancer une exception dans le cas contraire.

> En Python, on teste le type d'une variable avec `type(var)`, qui renvoie `int` ou `float` pour un nombre. On pourra tester ceci avec :
> `if not type(var) is int:`
> On peut lancer une exception avec le mot clé `raise`, par exemple :
> `raise TypeError("Only integers are allowed")`

Q2.6 - Vérifier que on déclenche bien l'erreur attendue en appelant depuis le programme principal la fonction "puissance" en passant des flottants.

3 Ajout du code de test

Python dispose de multiples bibliothèques de tests, les principales sont :

- `unittest`, qui est intégrée à Python3.
- `pytest`, bibliothèque tierce, mais plus facile d'usage au début.

On utilisera ici la seconde, qui doit d'abord être installée avec :
`$ pip3 install pytest`

Q3.1 - Créer dans le même dossier un 3e programme `test.py`, contenant ceci :

```
import pytest
import fonctions as f

def test_1():
        assert f.puiss(2,3) == 8
        assert f.puiss(2,2) == 4
```

Q3.2 - Lancer le test avec : `$ pytest test.py` [1]
ou `$ python3 -m pytest test.py`
et vérifier que les tests de la fonction `test_1` "passent" sans erreur.

Q3.3 - Dans ce fichier, ajouter une 2è fonction de test nommée `test_2` dans laquelle vous ajouterez des tests avec des valeurs négatives. Vérifier le bon fonctionnement.

Q3.4 - "Committer" les fichiers, en les ayant préalablement ajouté avec "git add"

4 Création d'un *workflow* d'intégration continue

Dans cette partie, vous allez mettre en place un "workflow" qui va lancer automatiquement le code de test à chaque modification du code.

Sur Github Actions, ceci se fait en spécifiant les actions à effectuer dans des fichiers au format yaml. Un certain nombre de concepts sont à identifier : *events*, *jobs*, *steps*, *actions*, et *runners*.

- On peut avoir plusieurs fichiers yaml, qui chacun correspondront à un "workflow".
- L'exécution d'un workflow est déclenchée par un "*event*", qui pourra être typiquement une opération de "push" sur le dépot.
- Chaque workflow peut spécifier plusieurs "*jobs*", qui seront exécutés en parallèle.
- Chaque job définit une séquence d'actions, qui peuvent être soit des tâches prédéfinies disponibles sur la plateforme via le mot-clé `uses`, soit une commande shell via le mot-clé `run`

Pour plus de détails, se référer à la doc officielle, ou à cette page :
`https://everhour.com/blog/github-actions-tutorial/`

Q4.1 - Créer un fichier `pytest.yml` dans un sous-dossier `.github/workflows` de la racine du projet :

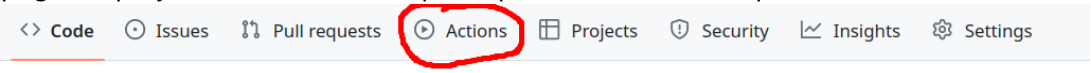
```
$ cd ~/tp-r504; mkdir -p .github/workflows
$ cd .github/workflows
$ touch pytest.yml
```

1. Il est possible que le binaire "pytest" ne soit pas dans le path, auquel cas il faut utiliser la 2e méthode d'appel.

Q4.2 - Editer ce fichier et y copier ce qui suit :

```
name: tests unitaires
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python for Demo
        uses: actions/setup-python@v4
      - name: Run tests
        run: pytest TP1/test.py
```

Q4.3 - L'ajouter au dépot et committer, puis faire un "push". Se connecter avec un navigateur sur la page du projet, et examiner ce qui se passe dans la rubrique "Actions" :



Vous devez voir que le test échoue. Pourquoi ?

Corrigez le problème en modifiant le fichier .yaml, et vérifier que les tests passent en CI.

Q4.4 - On peut avoir une visualisation du résultat du dernier test via un "badge" généré automatiquement, qui sera vert ou rouge en fonction du résultat de la dernière suite de tests.
Ce badge se trouve à l'URL :
`https://github.com/USER/PROJET/actions/workflows/FICHIER_YAML/badge.svg`
En Markdown, on affiche une image avec la syntaxe : `![text ALT](lien-vers-l'image)`
Ajouter dans votre "readme" une ligne qui affiche ce badge, avec devant le texte "Test results". Committer les changements, faites un push, et vérifier sur la page du projet que le badge apparait.

Q4.5 - On pourra de façon astucieuse transformer ce badge en lien de type "image", qui pointera sur la page "actions" du projet. Ainsi, un visiteur pourra observer le résultat du dernier test en détail en cliquant simplement sur le badge.
En Markdown, un lien image sera de la forme :
`[![ALT textGH](https://lien/sur/image)](https://lien/destination)`
Faites la modification et vérifier qu'en cliquant sur le badge on arrive sur la page "actions" du projet.

Q4.6 - Ajouter ensuite d'autres tests dans le programme de test, couvrant les cas limites (cf. slides de cours), et vérifier que les tests automatisés passent correctement.

5 Ecriture de la fonction sans l'opérateur intégré

Dans la fonction "puissance", remplacer l'usage de l'opérateur intégré "**" par une boucle for qui va itérer pour réaliser l'opération. Essayez votre code, puis le pousser dans le dépot. Si les tests automatisés échouent, corriger votre code jusqu'à passage complet de tous les tests.

Faites valider par l'enseignant.

6 Compléments sur GH Actions

6.1 Variables et "Contextes"

Dans le fichier yaml de configuration, il est possible d'utiliser des variables, pouvant être des variables d'environnement ou des variables utilisateurs (comme dans un shell classique) Par ailleurs, GH Actions dispose aussi du concept de "contextes", permettant d'utiliser des informations dans les scripts de CI.

Contextes et variables peuvent répondre aux mêmes besoins, mais les premiers permettent de "hierarchiser" les informations : un contexte est un objet qui pourra contenir une valeur (de type chaîne) ou d'autres objets, et on accède au sous-élement avec le caractère "point".

Ils s'utilisent avec la syntaxe `${{ <context> }}`

Par exemple, on pourra afficher le nom de la branche et le nom du repo avec les "steps" ci-dessous :

```
    - run: echo "Nom branche: ${{ github.ref }}"
    - run: echo "Nom repo:${{ github.repository }}."
```

Voir ce lien pour une liste des contextes disponibles :
`https://docs.github.com/en/github-ae@latest/actions/learn-github-actions/contexts`

Q6.1 - Dans votre projet du TP1, ajouter l'affichage du nom de la branche ainsi que l'affichage de l'OS de la VM sur lequel ces jobs sont lancés.

6.2 Matrices

Il existe des contextes de type "matrices", permettant de spécifier que l'on souhaite effectuer les "jobs" avec différentes "configurations" : différentes versions de l'OS, ou différentes versions d'une bibliothèque logicielle, ou autre.

Par exemple, pour faire l'ensemble des tests avec plusieurs versions de Python, on spécifie sous la clé "build" une clé "matrix" :

```
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.8", "3.10"]
```

Ceci permet d'avoir dans l'énumération des "steps" une ligne indiquant qu'on va utiliser les versions de Python indiquées dans cette variable.

```
    - name: Set up Python for Demo
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python-version }}
```

Q6.2 - Dans votre projet du TP1, ajouter la mise en place des tests avec ces 2 versions de Python.

Q6.3 - De façon à vérifier que c'est bien ces deux versions de Python qui sont utilisées, ajouter une "step" qui affichera la version de Python, afin qu'on puisse vérifier sur le log du job (via un simple "echo").
`python --version`