TP 3 - Optimisation de la mémoire.

Certaines questions ci-dessous vous demandent d'écrire des décorateurs. Pour tester votre code, vous pouvez créer une fonction bidon à décorer, par exemple :

```
def g(a, b, c, essai=None):
    """Fonction bidon."""
    return
```

Exercice 1 (Générateurs)

Écrivez un générateur infini de nombres premiers. Chaque avancement dans ce générateur nous fournira le nombre premier suivant. Par exemple :

```
>>> machine_a_nombres_premiers = premiers()
>>> next(machine_a_nombres_premiers)
2
>>> next(machine_a_nombres_premiers)
3
>>> next(machine_a_nombres_premiers)
5
>>> next(machine_a_nombres_premiers)
7
>>> next(machine_a_nombres_premiers)
11
```

Exercice 2 (Coroutines)

Écrivez une coroutine calculant la table de fréquences des caractères d'une chaîne. Utilisez-la dans un programme, et alimentez-la grâce aux lignes d'un fichier texte.

Exercice 3 (`lru_cache`)

Écrivez un décorateur qui suggère automatiquement l'utilisation de `lru_cache` dans les cas où cela est possible, c'est-à-dire quand tous les paramètres de la fonction sont d'un type *hashable*. Un objet est *hashable* s'il possède une méthode `__hash__`. Par exemple :

```
>>> g(range(40), [-200, 2, 3], "bonjour", essai=(3, 'a', 4, 2))
Appel de la fonction g :
    certains paramètres ne sont pas hashables, lru_cache est inutilisable
>>> g(42, (-200, 2, 3), "bonjour")
Appel de la fonction g :
    tous ses paramètres sont hashables, vous pourriez utiliser lru_cache
```

Utilisez `signature.bind()` (module `inspect`) pour intercepter les paramètres lors de l'appel à la fonction décorée. Pour simplifier, on suppose que la fonction décorée n'est pas définie de manière générique (il n'y a donc pas de `*args` ou `**kwargs` dans ses arguments).

Remarque : en pratique, il serait plus réaliste de suggérer le remplacement si **tous** les appels à la fonction satisfont cette propriété.

Exercice 4 (`array` ou pas `array` ?)

Le but de l'exercice est d'écrire un décorateur qui permet de détecter automatiquement, lors des appels de fonctions, quels paramètres peuvent être remplacés par des `array`.

(a) Écrivez une fonction à un seul paramètre qui renvoie :
- si possible : le type d'`array` à utiliser pour remplacer le paramètre donné par un `array` (reportez-vous à la documentation de `array`). Utilisez le plus petit type possible.
- `None` dans le cas contraire.

Par exemple :

```
>>> type_array("bonjour")
'u'
>>> type_array(range(10))
'b'
>>> type_array(range(-200, 10))
'h'
>>> type_array([-1, 3.5, True])
'f'
>>> type_array([-1, 3.5, True, "bonjour"]) is None
True  # pas possible à cause de mélange de types incompatibles
```

(b) Utilisez `signature.bind()` (module `inspect`) pour écrire un décorateur qui intercepte les paramètres lors de l'appel à la fonction décorée, et qui suggère automatiquement le remplacement des paramètres qui s'y prêtent par des `array`. Pour simplifier, on suppose que la fonction décorée n'est pas définie de manière générique (il n'y a donc pas de `*args` ou `**kwargs` dans ses arguments).

Une fois la fonction décorée, le résultat pourrait ressembler à ceci :

```
>>> g(range(40), [-200, 2, 3], "bonjour", essai=(3, 'a', 4, 2))
Appel de la fonction g :
    paramètre a = range(0, 40) (type <class 'range'>):
        remplaçable par un array de type b
    paramètre b = [-200, 2, 3] (type <class 'list'>):
        remplaçable par un array de type h
    paramètre c = bonjour (type <class 'str'>):
        remplaçable par un array de type u
    paramètre essai = (3, 'a', 4, 2) (type <class 'tuple'>):
        pas remplaçable par un array
```

(c) Si ce n'est déjà fait, améliorez votre fonction de la première question pour qu'elle détecte également les cas où l'utilisation de nombres réels est inutile. Par exemple :

```
>>> type_array([-1, 3.0, True])  # avant amélioration
'f'
>>> type_array([-1, 3.0, True])  # après amélioration
'b'
```

Exercice 5 (cyarray)

Écrivez un décorateur qui vérifie, pour chaque paramètre passé à une fonction, si ce paramètre est remplaçable par un des types du module `cyarray`. Pour rappel, il faut que :

1. le paramètre soit un itérable ne contenant que des nombres ;
2. la fonction n'utilise pas de *slices* sur cet itérable.

Pour vérifier la seconde propriété, vous aurez besoin des modules standard `ast` et `inspect`.

Une fois la fonction décorée, le résultat pourrait ressembler à ceci :

```
>>> g(range(40), [-200, 2, 3], "bonjour", essai=(3, 'a', 4, 2))
Appel de la fonction g :
    paramètre a = range(0, 40) (type <class 'range'>):
        remplaçable par un UIntArray
    paramètre b = [-200, 2, 3] (type <class 'list'>):
        remplaçable par un IntArray
    paramètre c = bonjour (type <class 'str'>):
        pas un itérable de nombres
    paramètre essai = (3, 'a', 4, 2) (type <class 'tuple'>):
        pas un itérable de nombres
```