CBF, LUR, LRG

Projets de développement informatique

Des fonctions, des fonctions et encore... des fonctions !

1 Contexte

Le but de ce labo consiste à comprendre la programmation fonctionnelle et les décorateurs. Pour cela, une série de petits exercices vous sont proposés, le but étant de bien comprendre dans le détail et en profondeur le code que vous écrirez pour résoudre ces exercices.

2 Exercices

Commençons avec quelques exercices pour se remettre dans le bain de la programmation fonctionnelle :

1. Définir une fonction `call` qui reçoit en paramètre une fonction sans paramètres qui ne renvoie pas de valeur et qui l'appelle. On doit, par exemple, pouvoir écrire le code suivant :

```
# À compléter...

def hello():
    print('hello')

call(hello)                             # Affiche 'hello'
```

2. Modifier maintenant votre fonction `call` de sorte qu'elle puisse appeler une fonction qui prend des paramètres positionnels et qui renvoie une valeur. On doit, par exemple, pouvoir écrire le code suivant :

```
# À compléter...

def add(a, b):
    return a + b

print(call(add, 2, 9))                  # Affiche '11'
```

3. Enfin, définissez une fonction `compute` qui reçoit deux paramètres positionnels `a` et `b` et un troisième paramètre optionnel `op` qui est une fonction représentant l'opération à faire entre `a` et `b`. Par défaut, il faut faire une addition. Modifiez la fonction `call` en conséquence, pour qu'elle puisse accepter des paramètres optionnels. On doit, par exemple, pouvoir écrire le code suivant :

```
# À compléter...

def add(a, b):
    return a + b

def sub(a, b):
    return a - b

# À compléter...

print(call(compute, 2, 9))              # Affiche '11'
print(call(compute, 2, 9, op=sub))      # Affiche '-7'
```

Venons-en à des exercices sur les décorateurs :

1. En exploitant la fonction Python `sleep` définie dans le module `time`, définir une nouvelle annotation `delay` qui, lorsqu'appliquée à une fonction, provoque un délai de une seconde avant que la fonction ne soit effectivement appelée. On doit, par exemple, pouvoir écrire le code suivant :

```
# À compléter...

@sleep
def printnum(i):
    print(i)

cnt = 3
while cnt > 0:
    printnum(cnt)
    cnt -= 1
print('KA-BOOM!')
```

2. Améliorer votre décorateur pour que l'on puisse paramétrer le temps d'attente. On doit, par exemple, pouvoir écrire le code suivant :

```
# À compléter...

@sleep(5)
def printnum(i):
    print(i)

cnt = 3
while cnt > 0:
    printnum(cnt)
    cnt -= 1
print('KA-BOOM!')
```

Terminons avec deux petits exercices sur les générateurs et itérateurs :

1. Définissez une fonction `binrep` qui génère de manière fainéante la représentation binaire d'un nombre naturel, en commençant par le bit de poids le plus faible (exploitez le modulo 2). On doit, par exemple, pouvoir écrire le code suivant :

```
# À compléter...

b = binrep(12)
while True:
    try:
        print(next(b))
    except StopIteration:
        break
```

2. Définissez une classe `Natural` qui représente un nombre naturel et qui est itérable, permettant d'afficher sa représentation binaire. On doit, par exemple, pouvoir écrire le code suivant :

```
class Natural:
    # À compléter...

for e in Natural(42):
    print(e)
```

3. Généralisez ensuite votre code pour que l'on puisse créer autant d'itérateurs que l'on souhaite, en définissant donc une nouvelle classe `NaturalIterator`.