Python : programmation fonctionnelle

Ressources

- https://python.developpez.com/tutoriels/apprendre-programmation-fonctionnelle/

Présentation

Le **paradigme de programmation fonctionnelle** se caractérise essentiellement par l'**absence d'effets de bord**.

Pour cela, le code défini à l'intérieur d'une fonction :

- **ne dépend pas** de données se trouvant à l'extérieur de cette fonction courante ;
- et le code à l'intérieur de la fonction **ne modifie pas** des données à l'extérieur de cette fonction.

Exemples

- voici une fonction **qui n'utilise pas ce paradigme de programmation fonctionnelle** :

```
a = 0
def augmenter():
    global a
    a = a + 1
```

Explications :

- la variable **a** est définie à l'extérieur de la fonction **augmenter()** ;
- Il est précisé dans le corps de la fonction **augmenter()** que la variable **a** est définie comme **globale** ;
- lors de l'exécution de la fonction **augmenter()**, la valeur de la variable **a** est **changée** pour **passer de 0 à 1**. Comme il s'agit d'une variable **a globale**, sa valeur est maintenant **changée pour l'ensemble du programme** et pas seulement à l'intérieur de la fonction **augmenter()**.

- Voici la même fonction une fonction **augmenter()** en utilisant la programmation fonctionnelle :

```
def increment(a):
    return a + 1
```

Explications :

- la variable **a** est passée en paramètre à la fonction **augmenter()** ;
- cette fonction retourne la **valeur de a augmentée de 1** mais **sans modifier** la valeur intiale de a qui reste alors à 0.
- Il n'y a pas d'effet de bord : à l'issue de l'exécution de la fonction **augmenter()**, la valeur de la variable **a n'a pas changée**.

Itérer sur des listes

La programmation fonctionnelle est particulièrement intéressant pour intervenir sur des listes. Les exemple qui suivent s'appuie sur la liste de Todos suivantes :

```
taches = [{"id":"1","libelle":"Préparer mon sac", "accomplie":True},
          {"id":"2","libelle":"Prendre mon petit-déjeuner", "accomplie":False},
          {"id":"3","libelle":"Partir au lycée", "accomplie":False}]
```

La fonction map

- La fonction **map** prend en argument une **fonction** et une **collection de données** ;
- Elle **crée** une nouvelle collection vide ;
- **applique** la fonction à chaque élément de la collection d'origine et insère les valeurs de retour produites dans la nouvelle collection ;
- elle renvoie alors la nouvelle collection.

Voici l'utilisation de la fonction **map** pour avoir une nouvelle liste **tachesfinies** avec toutes les tâches finies :

```
# definition de la fonction qui met la valeur True pour la donnée acccomplie
def fini(tache):
  return {"id":tache["id"],"libelle":tache["libelle"], "accomplie":False}

#nouvelle liste avec toutes taches accomplies
tachesfinies = list(map(fini,taches))
```

Il est possible d'utiliser une **fonction anonyme lambda** directement dans la fonction map :

```
# utilisation d'une fonction anonyme lambda
#nouvelle liste avec toutes taches accomplies
tachesfinies = list(map(lambda tache: {"id":tache["id"],"libelle":tache["libelle"], "accomplie":False},
taches))
```

> Les **fonctions lambda** sont des **fonctions anonyme**s, c'est à dire des fonctions qui **n'ont pas de nom. Une fonction anonyme : * est définie à l'aide du mot-clef lambda ; * les paramètres de la fonction lambda sont définis à gauche du caractère deux-points** :
>
> - le **corps** de la fonction est **défini à sa droite** ;
> - le **résultat** de l'exécution du corps de cette fonction, ce qui correspond à l'instruction **return** est **renvoyé implicitement**.
>
> Une fonction anonyme peut être **placée** :
>
> - **directement** dans une fonction qui accepte en **paramètre** une fonction ;
> - dans une **variable** pour être utiliser ultérieurement.
>
> Exemple avec une variable :
>
> ```
> # definition de la fonction anonyme et affectation dans la variable fois2
> fois2 = lambda x: x * 2
>
> #utilisation
> >>> print(fois2(4))
> >>> 8
> ```

Même résultat avec un générateur :

```
#nouvelle liste avec les taches modifiées pour les indiquer accomplies (finies)
tachesfinies = [fini(tache) for tache in taches]
```

La fonction filter

- La fonction **filter** prend en argument une **fonction** qui est une **condition** et une **collection de données** ;
- Elle **crée** une nouvelle collection vide ;
- **applique** la fonction à chaque élément de la collection d'origine et **insère dans la nouvelle collection uniquement les éléments qui répondent à la condition ; * Elle renvoie alors la nouvelle collection. Voici l'utilisation de la fonction filter pour avoir une nouvelle liste tachesfinies qui ne contient que les tâches finies : <code python> # definition de la fonction qui teste la valeur True pour la donnée acccomplie def fini(tache): return tache[“accomplie”] == True #nouvelle liste avec uniquement les taches accomplies (finies) tachesfinies = list(filter(fini,taches)) </code> Même résultat avec une fonction anonyme : <code python> #nouvelle liste avec uniquement les taches accomplies (finies) tachesfinies = list(filter(lambda tache:tache[“accomplie”] == True, taches)) </code> Même résultat avec un générateur : <code python> #nouvelle liste avec uniquement les taches accomplies (finies) tachesfinies = [tache for tache in taches if tache[“accomplie”] == True] </code> ===== La fonction reduce ===== La fonction reduce prend en entrée une fonction et une collection d'éléments. Elle renvoie une valeur créée en combinant les éléments de la**

collection. Exemple qui calcule la somme des éléments d'un tableau <code python> # importation de la fonction reduce from functools import reduce tableau = [0, 1, 2, 3, 4] somme = reduce(lambda a, x: a + x, tableau) # somme contient 10 </code>

Explications :

- **x** est l'élément courant de l'itération et **a** est **l'accumulateur** ;
- l'accumulateur est la valeur renvoyée par l'exécution de la fonction lambda sur l'élément précédent ;
- la fonction **reduce()** parcourt les éléments de la liste et, pour chacun d'eux, exécute la fonction lambda sur les valeurs courantes de a et de x et renvoie le résultat qui devient le a de l'itération suivante ;
- la valeur de a lors de la première itération est la première valeur de la liste ; la première valeur de x est donc le second élément de la liste.
- pour préciser une valeur initiale différente de a, on le précise en 3ème paramètre de la fonction reduce :

```
somme = reduce(lambda a, x: a + x, tableau, 50)
```

====== Trier des listes ====== <code python> taches.sort(key=lambda tache: tache[“id”], reverse=True) </code> * le paramètre **key** permet de préciser sur quelle données trier ; * le paramètre **reverse** permet de changer **l'ordre** du tri. ==== Retour au cours : Les instructions du langage Python ====

- Cours : Les instructions du langage Python