éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

TYPES MUTABLES ET PROBLÈMES ASSOCIÉS

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

Nous supposons que le lecteur sait déjà effectuer des opérations basiques sur les listes (le type `list` de Python) et les p-uplets (le type `tuple`).
La principale différence entre ces deux types est que le premier est modifiable (« mutable » en anglais) alors que le second est persistant (« immutable » en anglais).
En effet, on peut modifier le contenu d'une liste, et même changer sa taille, ce qui est impossible pour un p-uplet. Essayez :

```
p = (1,2,3)
p[0] = 1
p.append(4)
p.pop()
```

Le caractère mutable d'un type aura plusieurs conséquences pratiques en Python, qui sont détaillées ci-dessous.

1. Phénomène d'alias

Tout élève tombe un jour ou l'autre dans ce piège... Rappelons-le brièvement :

```
t = [1,2,3]
s = t
t[0] = 5
# Que vaut s à présent ?
```

À la ligne 2, la liste `t` n'est pas copiée dans `s` : au contraire les identificateurs `s` et `t` désignent la *même* liste. Toute modification de l'une modifie aussi l'autre (d'ailleurs le mot « l'autre » est mal choisi puisqu'en fait c'est la même !) : on dit qu'ils sont en alias.
Notons que la question du comportement analogue pour un p-uplet ne se pose même pas puisque l'opération effectuée en ligne 3 n'existe tout simplement pas pour ce type. Prenons le code suivant :

```
p = (1,2,3)
q = p
p = p + (4,5)
# Que vaut p ?
```

Le point est ici que l'opération effectuée en ligne 3 ne modifie pas la valeur actuellement désignée par p mais modifie p en tant que telle. Il s'agit d'une affectation : elle commence par évaluer l'expression située à droite du symbole =, puis change l'environnement courant pour que l'identificateur p désigne désormais cette valeur. La nouvelle définition de p remplace alors l'ancienne mais on n'a pas touché à la valeur anciennement désignée.

2. Exercices autour de la copie

Voici quelques exercices de difficulté strictement croissante autour de la notion de type mutable, et de copie. Les deux premiers peuvent être traités par des élèves, les suivants nécessitent un peu de récursivité et sont plutôt destinés à l'enseignant qui veut prendre un peu de recul.

1. Écrire une fonction `copie_tab` qui renvoie une copie de la liste passée en argument.
2. M. X utilise la fonction précédente pour copier une matrice :

```
1  def copie_tab(t):
2      res = []
3      for x in t:
4          res.append(x)
5        return res
6
7  M = [ [1,2],
8      [3,4]]
9  N = copie_tab(M)
10 M[0][0] = 0
```

   Que vaut N à l'issue de ces opérations ? Expliquer.

3. On pourrait naturellement écrire une fonction `copie_mat` qui crée une copie d'une matrice. Quelle serait sa complexité ?

   Mais on propose de passer directement à l'étape générale : écrire une fonction `copie_profonde` qui renvoie une copie d'une liste de n'importe quelle dimension. Autrement dit, elle devra fonctionner pour une liste, une liste de listes, une liste de listes de listes...

   On pourra utiliser l'expression `isinstance(t, list)` pour savoir si un objet `t` est une liste.

Signalons que la copie profonde réalisée pour le cas des listes de listes par notre fonction `copie_profonde` peut être effectuée dans le cas le plus général par la fonction `deepcopy` du module `copy`.
Remarques : développements possibles : Écrire une fonction pour créer une matrice et voir qu'il y a plusieurs méthodes pour la rater.

3. Corrigé des exercices sur la copie

1. Voici une fonction pour copier un tableau :

```
1  def copie_tab(t):
2      res=[] # Création d'un *nouveau* tableau
3      for x in t:
4          res.append(x)
5      return res
```

2. M. X a réutilisé la même fonction que ci-dessus pour copier une matrice. Alors N est bien un nouveau tableau, découplé de M. Par contre, les deux lignes `M[0]` et `M[1]` n'ont pas été copiées ! Donc `M[0]` et `N[0]` désignent le même tableau, de même que `M[1]` et `N[1]`. Alors la modification de `M[0]` effectuée ligne 9 modifie aussi `N[0]`, de sorte que N désigne `[[0,2],[3,4]]`
3. Pour copier n'importe quel emboîtement de tableaux, créons une fonction récursive dont le cas d'arrêt est celui d'un objet qui n'est pas une liste.

```
def copie_tab_profonde(t):
    if isinstance(t, list):
        res=[]
        for x in t:
            res.append( copie_tab_profonde(x))
        return res
    else:
        return t
```

Un petit test :

```
M = [ [1,2],
      [3,4]]
N = copie_tab_profonde(M)
M[0][0] = 0
print(N)
```

4. Fonction dont un paramètre est mutable

4.1. Potentialité d'impureté

Dans la première section, on a présenté le problème de l'alias comme le résultat d'un programme étrange où on effectue `s = t` avec des structures mutables.
En fait, ce phénomène d'alias a lieu à chaque fois qu'on effectue un appel de fonction.
Analysons l'exemple suivant :

```
def truc(t):
    t.append(0)

t = [4]
s = [1]
truc(s)
# Maintenant, que vaut s ?
```

On constate que la modification de la liste effectuée par `truc` a été conservée en dehors de cette fonction. Ceci ne contrevient nullement au principe des variables locales. En effet, lors de l'exécution de `truc(s)`, Python :

1. calcule la valeur du paramètre effectif de l'appel, ici `s`,
2. crée un contexte local pour la fonction `truc` (dans ce contexte, le paramètre formel `t`, à l'instar d'une variable, est local ; en particulier ce n'est pas le même identificateur que le `t` global qu'on a également défini ligne 4),
3. crée une liaison entre la valeur calculée à l'étape 1 et le paramètre formel correspondant dans le contexte local : il y a donc alias entre le `s` du contexte global et le `t` du contexte de l'appel.

Ce phénomène d'alias n'est pas spécifique au cas des paramètres mutables : il se produit lors de tout appel de fonction. Cependant, comme on l'a dit, tant qu'on n'a que des objets immuables, ce phénomène peut être ignoré.
Lorsqu'une fonction modifie un paramètre mutable, on dit qu'elle a un effet sur ce paramètre. Plus généralement, une fonction peut réaliser des effets de bord comme un affichage à l'écran. Une fonction qui a des effets ou est susceptible d'en avoir est dite impure.

4.2. Points de vocabulaire, précautions

Lorsqu'on écrit une fonction réalisant une opération avec une structure mutable (comme trier une liste par exemple), il existe donc en général deux possibilités.

- ▷ On peut construire une nouvelle structure qui a le contenu souhaité et renvoyer cette structure. En ce cas, il est d'usage de ne pas modifier la structure de départ.

▷ On peut muter la structure de départ pour en réorganiser le contenu (opération en place). En ce cas, il est d'usage de ne pas renvoyer la structure modifiée (puisque, structurellement, c'est la même qu'au départ).

Certains auteurs réservent le terme « fonction » au premier cas et parlent de « procédure » pour désigner les fonctions qui ne renvoient pas de valeur. Cette distinction de vocabulaire peut avoir un intérêt pédagogique mais il faut savoir qu'en Python, toute fonction renvoie une valeur, fût-ce la valeur `None`, et qu'il n'existe donc pas de procédure au sens de Pascal par exemple.
Cela est lié au fait que tout appel de fonction en Python est une expression, et a pour conséquence que certaines constructions conceptuellement erronées sont néanmoins syntaxiquement valides et donc non décelées par l'interpréteur.
Cela se produit aussi avec les fonctions et méthodes de base des types mutables. Voici par exemple une erreur fréquente en début d'année :

```
t = t.append(1)
t[0]
```

À l'issue de la première ligne, `t` vaut désormais `None`, parce que c'est la valeur renvoyée par `t.append(1)`.
Une procédure et une fonction, si l'on veut utiliser ces termes, ne s'utilisent pas de la même manière, il est donc important que les élèves sachent quel type de programme ils sont en train d'écrire.
Pour les aider, on peut par exemple prendre pour convention d'utiliser des verbes dans les noms de procédure, et des noms ou participes passés dans les noms de fonctions. Par exemple `à_l_envers` serait le nom d'une fonction qui renvoie une liste obtenu en renversant celui passé en argument, et `renverser` serait une procédure qui retourne le contenu de la liste passée en argument.

5. Mise en garde sur += et consorts

On présente souvent l'opérateur `+=` comme une abréviation de `+` et `=`. Effectivement, pour des entiers, il revient au même de faire par exemple `n += 1` et `n = n + 1`.
Cependant, contrairement à l'affectation `=` dont la sémantique est précisément définie par le langage Python, les opérateurs `+=`, `-=` etc. ne sont que des notations pour appeler des méthodes des objets manipulés. Cela signifie que le sens précis de `x += y` dépend du type de `x` : en réalité, cette instruction exécute, quand elle existe, la méthode `x.__iadd__(y)`. C'est seulement si cette méthode n'existe pas pour l'objet considéré que Python évalue `x + y` (soit en fait `x.__add__(y)`) et affecte le résultat à `x`.
Or, pour les listes, la méthode `__iadd__` est implémentée comme la méthode `extend` : du point de vue des effets, `L += M` est équivalent à `L.extend(M)` et **modifie** la liste `L` pour lui ajouter, à droite, les éléments contenus dans `M`. Par conséquent, ce n'est pas la même chose que `L = L + M` dont l'effet est de créer une nouvelle liste par concaténation de `L` et `M` puis d'affecter à `L` cette nouvelle liste.
Par conséquent, la fonction `f` ci-dessous a un effet sur son paramètre mais pas la fonction g.

```
def f(L, n) :
  for k in range(n) :
    L += [k, k+1]

def g(L, n) :
  for k in range(n) :
    L = L + [k, k+1]
```

En outre, g a une complexité qui dépend de la taille initiale de `L` parce que chaque opération de concaténation effectuée recopie tous les éléments des listes concaténées, tandis que la fonction `f` se contente d'ajouter 2 termes à `L` à chaque tour de boucle et est de complexité $O(n)$ si l'on veut bien considérer que les opérations d'extensions de liste sont font en complexité amortie constante.
Nous recommandons pour ces raisons de ne jamais utiliser l'opérateur `+=`; le gain de place dans les cas simples comme les entiers ne vaut pas le risque des confusions lorsqu'on manipule des listes.

6. Intérêt de l'immuabilité

Les p-uplets peuvent apparaitre comme des listes présentant moins de fonctionnalités : les opérations permettant de modifier, d'ajouter, ou de supprimer un élément en sont absentes, et ils n'apportent aucune fonctionnalité supplémentaire.

Ces restrictions apparentes ont en fait des avantages de deux ordres :

- ▷ sémantiquement, l'immuabilité est une garantie : une valeur immuable se comporte comme une valeur mathématique ; elle nécessite moins de précautions d'emploi ; il est en général beaucoup plus agréable de démontrer des propriétés sémantiques (correction, terminaison) à propos de programmes dont toutes les structures sont immuables ;
- ▷ l'implémentation peut bénéficier de ces garanties sémantiques : l'interpréteur (ou le compilateur dans le cas d'un langage compilé) peut gérer l'allocation mémoire de l'objet plus finement, notamment en partageant la représentation mémoire d'objet immuable qui interviennent plusieurs fois (une telle optimisation n'est pas possible pour une structure mutable puisque chaque exemplaire doit pouvoir évoluer indépendamment des autres).

L'intérêt des structures immuables est particulièrement renforcé lorsqu'on pratique la programmation fonctionnelle, en OCaml, Haskell, F# par exemple. Ces langages sont optimisés pour la manipulation de telles structures.
Même dans des langages plus traditionnellement portés sur les structures mutables, comme C++, il y a un intérêt à se demander si telle fonction peut modifier ou non telle structure passée en argument. Le langage C++ dispose ainsi d'un mot-clé `const` pouvant être indiqué lors de la déclaration des paramètres formels d'une fonction : il constitue une promesse que la fonction ne modifie pas ce paramètre. Le compilateur force le programmeur à respecter cette promesse en n'autorisant, pour un tel paramètre, que des opérations ou appels à des fonctions qui elles-mêmes font cette promesse : on constitue ainsi une chaine de `const` qui permet au compilateur de prouver de façon automatique l'absence de modification de l'objet.

```
int g(const sometype& p) {
  // cette fonction promet de ne pas modifier p
  return p.something;
}

int h (sometype& p) {
  // aucune promesse ici
  return p.something;  // de fait on ne modifie pas p, mais on n'en a pas fait la
   ↪ promesse
}

void f(const sometype& param1, sometype& param2) {
  // f promet de ne pas modifier param1, mais ne promet rien pour param2
  int v = h(param2);   // licite car on fait ce qu'on veut de param2
  int w = g(param1);   // licite car g ne modifie pas son paramètre
  int x = h(param1);   // erreur car rupture de la chaine de const
}
```

En Python, on voit apparaître cette distinction lorsque l'on utilse des structures de données du type *ensemble* ou *dictionnaire*. Pour illustrer cela, essayons de vérifier que la fonction `random.shuffle` qui mélange une liste renvoie bien les différentes permutations possibles avec la même probabilité.
On peut programmer cela ainsi (en supposant que le module `random` est déjà importé), où `n\` est le nombre de listes à tester, et `p` la longueur de la liste :

```
def test(n, p):
  dico = {}
  for _ in range(n):
    # on créé la liste des entiers de 0 à p - 1
    l = list(range(p))
    random.shuffle(l)
    if l in dico:
      dico[l] = dico[l] + 1
    else:
      dico[l] = 1
  return dico
```

En utilisant cette fonction, on obtient le type d'erreur suivant :

```
>>> test(100000, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in test
TypeError: unhashable type: 'list'
```

La raison est très simple : une liste (ici `l`) étant mutable, Python n'est pas assuré que sa valeur ne va pas varier dans le futur. Son utilisation comme clé dans un dictionnaire n'a alors pas de sens[1]. En effet, si l'on a `l = [1]`, que l'on effectue `d[l] = "Bonjour"` puis que l'on modifie `l` en effectuant par exemple `l.append(2)`, est-ce que la valeur `"Bonjour"` doit être associée à la clé `[1]` ou `[1, 2]` ?
Pour modifier cela, il faut transformer la liste en donnée immuable, comme un p-uplet. On obtient la fonction suivante :

```
def test(n, p):
    dico = {}
    for _ in range(n):
        # on créé la liste des entiers de 0 à p - 1
        l = list(range(p))
        # on mélange la liste (qui est mutable)
        random.shuffle(l)
        # on transforme la liste mélangée en p-uplet
        t = tuple(l)
        # on utilise t comme clé dans le dictionnaire
        if t in dico:
            dico[t] = dico[t] + 1
        else:
            dico[t] = 1
    return dico
```

On a alors le comportement espéré :

```
>>> test(100000, 3)
{(2, 1, 0): 16667, (0, 1, 2): 16757, (0, 2, 1): 16711, (2, 0, 1): 16445, (1, 2, 0):
    ↪ 16719, (1, 0, 2): 16701}
```

On remarque que tous les nombres d'occurrences obtenus sont proches de $100\,000/6 = 16\,666.666\ldots$ ce qui montre que l'équiprobabilité des permutations semble acquise.

1. Techniquement, ce n'est pas une type de donnée *hachable*, on ne peut pas l'associer à un entier qui le représenterait, puisque sa valeur peut être modifiée.