P-uplets nommés & dictionnaires

P-uplets nommés (ou enregistrements)

Un *p-uplet nommé*, on dit aussi *enregistrement*, est un p-uplet contenant plusieurs *champs* dont les *éléments* sont appelées grâce à un *descripteur* au lieu d'un indice.

On peut voir un *enregistrement* ou *p-uplet nommé* comme un meuble à tiroirs :

- chaque tiroir est ce qu'on appelle un *champ,*
- chaque tiroir est étiqueté par un *descripteur,*
- chaque tiroir contient un *élément.*

En tant que structure abstraite, les p-uplets nommés (ou enregistrements) sont *immutables* : ils ne peuvent pas être modifiés.

Regardons l'exemple ci-contre qui implémente un p-uplet nommé grâce à un dictionnaire en Python :

```
>>> fiche_eleve = { "Prénom" : "Alice",
                    "Nom"    : "Merveille",
                    "Age" : 14,
                    "Loisir" : "Rêver"}

>>> fiche_eleve["Nom"]
'Merveille'
>>> fiche_eleve["Loisir"]
'Rêver'
```

- L'enregistrement (ou p-uplet nommé) `fiche_eleve` est constitué de quatre *champs* dont les *descripteurs* sont : `"Prénom"`, `"Nom"`, `"Age"` et `"Loisir"`.

- Le *champ* décrit par le *descripteur* `"Nom"` contient l'*élément* `"Merveille"`

- On obtient le contenu du champ décrit par `"Nom"` en utilisant la syntaxe : `enregistrement["Nom"]`.

Remarque 1 :

On voit sur cet exemple que l'utilisation de descripteurs au lieu d'indices facilite la compréhension du code. Lorsqu'on écrit `..."Age" : 14...` ou bien `fiche_eleve["Age"]` il est difficile de faire une erreur d'interprétation sur la donnée correspondante (alors que `fiche_eleve[2]` est moins clair).

Remarque 2 :

On voit aussi sur cet exemple que les enregistrements ont vocation … à exister en masse. Ici, on imagine tout à fait qu'il existe plusieurs fiches élèves … qui pourraient toutes être stockées dans un même tableau. On obtiendrait ainsi un tableau d'enregistrements … Ce sera l'objet du cours suivant !

Implémentation en Python : utilisation de *dictionnaires* (ou *tableaux associatifs*)

Vocabulaire

Nous implémenterons les p-uplets nommés en utilisant des *dictionnaires* python. Le vocabulaire des dictionnaires est différent de celui des p-uplets-nommés. Sur l'exemple ci-dessus :

- Le dictionnaire fiche_eleve est constitué de quatre couples *clé-valeur* (ou *associations*) : les *clés* correspondent aux *descripteurs* alors que les *valeurs* correspondent aux *éléments* des *champs*.

- La syntaxe d'un dictionnaire est : `{ clé_1:valeur_1, clé_2:valeur_2, clé_3:valeur_3 ...}`
  On rencontre également cette syntaxe en Javascript avec le type `Object`. Dans un certain nombre de langages on utilise une flèche `->` ou `=>` à la place des deux points.

Quelle est la différence entre un p-uplet nommé et un dictionnaire ? Qu'est ce que l'implémentation ?
Le p-uplet nommé est une structure de donnée abstraite : en simplifiant, on y fait référence quand on fait de l'algorithmique, de la théorie.
Le dictionnaire `dict` est un type qui permet de programmer des p-uplets nommés en langage python, on y fait référence quand on fait de la programmation, de la pratique.

En python on aurait aussi pu utiliser le type `named tuple` qui fonctionne un peu différemment du type `dict`. Dans d'autres langages, on utiliserait par exemple des objets de type `container` ou de type `map`.

Selon le type utilisé pour implémenter (pour programmer) des p-uplets nommés, certaines méthodes seront ou pas disponibles (ajouter un élément, supprimer un élément, chercher un élément, compter un élément etc.). Et les performances (temps, mémoire) de ces méthodes peuvent varier selon l'implémentation...

Une analogie :
voiture = idée abstraite qui peut être "implémentée" par voiture électrique, par 4x4, par berline (ou par tracteur, ou par autobus …)

Opérations élémentaires supportées par les dictionnaires (ou tableaux associatifs) en python

On a vu précédemment comment créer – en une seule instruction – un dictionnaire contenant quatre paires clé-valeur. L'exemple précédent du dictionnaire `fiche_eleve` est typique de l'implémentation d'un p-uplet nommé. En particulier parce que le dictionnaire `fiche_eleve` a vocation à rester immutable.
Mais les dictionnaires python sont très riches et offrent de nombreuses possibilités de *mutabilité*. Cela les rend très pratiques pour beaucoup d'applications en dehors du cadre des p-uplets nommés.

On dispose des opérations élémentaires suivantes :

- Créer un dictionnaire vide
- Ajouter une paire clé-valeur (ou association)
- Renvoyer la valeur associée à une clé
- Modifier la valeur associée à une clé
- Supprimer l'association correspondant à une clé
- Tester si une clé est présente

```
>>> ages = {}
>>> ages["Ali"] = 74
>>> ages["Béa"] = 29
>>> ages["Célia"] = 32
>>> ages
{'Ali' : 74, 'Béa' : 29, 'Célia' : 32}

>>> ages["Célia"]
32

>>> ages["Ali"] = 45
>>> del ages["Béa"]
>>> ages
{'Célia' : 32, 'Ali' : 45}

>>> "Béa" in ages
False
```

Exercice : mettre une étoile $*$ devant les opérations élémentaires qui sortent du cadre des p-uplets nommés (c'est-à-dire qui permettent de la *mutabilité*)

Remarque 1 : Demander à accéder à une clé qui n'existe pas provoque une erreur à l'exécution. La méthode get permet de pallier à ce problème en ne levant pas d'erreur mais en renvoyant `None`.

Remarque 2 : Les temps d'accès (ajouter une paire clé-valeur ou renvoyer la valeur associée à une clé) des dictionnaires sont satisfaisants en moyenne, de complexité constante en la taille du dictionnaire.

```
>>> ages["Zoé"]
KeyError Tracebacl (most recent…
<ipython-input-30b-b59205 …
----> 1 ages["Zoé"]
KeyError: 'Zoé'

>>> ages.get("Zoé")
None
```

Méthodes `keys`, `items` et `values`

ATTENTION : Lors du parcours d'un dictionnaire, l'ordre de parcours est imprévisible ! Un dictionnaire n'est pas ordonné comme un tableau !

On peut parcourir un dictionnaire par clés, par valeurs ou par couples clés-valeurs.

Considérons le dictionnaire ci-contre (au passage on voit que la fonction `len` permet d'en obtenir la longueur).

```
>>> ages = { "Ali" : 15,
             "Béa" : 20,
             "Célia" : 23,
             "Diego" : 18,
             "Eva" : 17,
             "Fati" : 25}
>>> len(ages)
6
```

Parcours par Clé (2)

```
>>> for xx in ages.keys():
        print(xx)

Célia
Béa
Ali
Diego
Fati
Eva

>>> type(ages.keys())
dict_keys
>>> list(ages.keys())
['Célia', 'Béa','Ali'...]
>>> 'DieGo' in ages.keys()
False
```

Parcours par Valeur

```
>>> for xx in ages.values():
        print(xx)

18
20
25
15
17
23

>>> type(ages.values())
dict_values
>>> list(ages.values())
[18, 20, 25, 15, ...]
>>> 18 in ages.values()
True
```

Parcours par couple Clé-Valeur

```
>>> for xx in ages.items():
        print(xx)

('Eva', 17)
('Béa', 20)
('Célia', 23)
('Fati', 25)
('Ali', 15)
('Diego', 18)

>>> type(ages.items())
dict_items
>>> list(ages.items())
[('Eva', 17), ('Béa', 20)...]
>>> ('Eva', 25) in ages.items()
False
```

Remarque 1 :

`ages.keys()`, `ages.values()` et `ages.items()` ne sont pas transformés directement en `list` par python : cela demanderait de l'espace mémoire et du temps pour effectuer la copie. C'est pourquoi si on souhaite la liste des clés, des valeurs ou des 2-uplets (clé, valeur), il faut demander explicitement la conversion grâce à `list()`.

Remarque 2 :

Le choix de `xx` comme nom de variable est très mauvais dans les exemples donnés. Il aurait mieux valu choisir …

Parcours par couple Clé-Valeur (2)

```
>>> for (xx, yy) in ages.items():
        if yy % 5 == 0:
            print(xx)

'Ali'
'Béa'
'Fati'
```

Remarque 3 :

Lors du parcours par couple Clé-Valeur, on peut utiliser deux variables simultanément : une pour les clés et une pour les valeurs. Il suffit de se souvenir de son cours sur les tuples (voir ci-contre)

Copie d'un dictionnaire en python

Comme pour les listes, la "copie" d'un dictionnaire par affectation à une variable n'est pas une "vraie" copie.
Ainsi, sur l'exemple ci-contre, les noms de variables `dic_1` et `dic_2` désignent le même dictionnaire.

```
>>> dic_1 = {"A":['A', 'a', 'à', 'â'],
             "C":['C', 'c', 'ç'],
             "I":['I', 'i', 'ï', 'î']}

>>> dic_2 = dic_1
>>> dic_2["B"] = ['B', 'b']
>>> dic_1
{'A': ['A', 'a', 'à', 'â'],
 'C': ['C', 'c', 'ç'],
 'B': ['B', 'b'],
 'I': ['I', 'i', 'ï', 'î']}
```

Pour effectuer une vraie copie à l'issue de laquelle `dic_1` et `dic_2` sont indépendants, on utilisera `deepcopy`.

```
>>> import copy
>>> dic_2 = copy.deepcopy(dic_1)
>>> dic_2["O"] = ['O', 'o', 'ô', 'ò', 'ö']
>>> dic_1
{'A': ['A', 'a', 'à', 'â'],
 'C': ['C', 'c', 'ç'],
 'B': ['B', 'b'],
 'I': ['I', 'i', 'ï', 'î']}
```

Applications dont l'existence est à connaître (*abordées en TP*)

Construction de dictionnaires par compréhension (au programme)

```
>>> import string
>>> alphabet = string.ascii_letters
>>> alphabet
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> points_de_code_hexa = { car : hex(ord(car)) for car in alphabet }
>>> points_de_code_hexa
{'A': '0x41', 'B': '0x42', 'C': '0x43', 'D': '0x44', 'E': '0x45','F': '0x46' ...}
```

Exemple : points de code Unicode en hexadécimal des premières lettres de l'alphabet

Dictionnaires en tant que containers de données (culture générale - HP)

Format JSON pour les requêtes à des API ou pour la lecture/sauvegarde dans des fichiers

Grâce à une API, obtention d'une adresse à partir de ses coordonnées GPS :

```
>>> import requests

>>> url = https://api-adresse.data.gouv.fr/reverse/?lat=48.853196&lon=2.368886
>>> r = requests.get(url)
>>> dictionnaire = r.json()
>>> dictionnaire
{'attribution': 'BAN',
 'features': [{'geometry': {'coordinates': [2.368253, 48.853498], 'type': 'Point'},
     'properties': {'city': 'Paris',
    'citycode': '75104',
    'context': '75, Paris, Île-de-France',
    'distance': 57,
        ......... etc .........
    'type': 'Feature'}],
  'licence': 'ETALAB-2.0',
  'limit': 1,
  'type': 'FeatureCollection',
  'version': 'draft'}
```

Sauvegarde (`dump`) puis lecture (`load`) d'un dictionnaire au format JSON (clefs de type `str`) :

```
>>> import json

>>> appel = {
        "client" : 42787,
        "collaborateur" : 1298,
        "jour" : 3214,
        "appel" : 443,
        "duree" : 134}

>>> fichier = open('./appel.json','w')
>>> json.dump(appel, fichier)
>>> fichier.close()
```

```
>>> import json

>>> fichier = open('./appel.json', 'r')
>>> appel = json.load(fichier)
>>> appel
{'appel': 443,
 'client': 42787,
 'collaborateur': 1298,
 'duree': 134,
 'jour': 3214}
```

Les exemples ci-dessus sont en langage Python mais on peut faire la même chose en JavaScript ou en PHP.

Données EXIF des images numériques

```
>>> import PIL.Image

>>> img = PIL.Image.open('./souris_gps.jpg')
>>> img._getexif()

{271: 'samsung',
 272: 'SM-G930F',
 274: 6,
 282: (72, 1),
 283: (72, 1),
 296: 2,
```

On peut remplacer les clés numériques par leur signification en utilisant le module `PIL.ExifTags` et en particulier

```
>>> import PIL.ExifTags
>>> import PIL.Image

>>> img = PIL.Image.open('./souris_gps.jpg')
>>> exif_brut = img._getexif()
>>> exif = { PIL.ExifTags.TAGS[k]:v for k, v in exif_brut.items() if k in PIL.ExifTags.TAGS }
>>> exif
{'ApertureValue': (153, 100),
 'BrightnessValue': (-101, 100),
 'ColorSpace': 1,
 'ComponentsConfiguration': b'\x01\x02\x03\x00',
 'DateTime': '2020:02:12 23:57:25',
 'DateTimeDigitized': '2020:02:12 23:57:25',
 'DateTimeOriginal': '2020:02:12 23:57:25',
 'ExifImageHeight': 1440,
 'ExifImageWidth': 2560,
 'ExifInteroperabilityOffset': 852,
 'ExifOffset': 214,
 'ExifVersion': b'0220',
 'ExposureBiasValue': (0, 10),
```

le tableau `PIL.ExifTags.TAGS`.

Exemples d'utilisations pour effectuer du dénombrement (culture générale - HP)

Décompter des éléments d'une liste

```
>>> import random as rd

>>> cartes = [rd.choice(['As', 'Roi', 'Dame', 'Valet']) for i in range(10)]
>>> cartes
['As', 'As', 'Dame', 'As', 'Dame', 'As', 'Roi', 'Dame', 'As', 'Dame']

>>> compteurs = {}
>>> for carte in cartes:
        if carte in compteurs.keys():
            compteurs[carte] = compteurs[carte] + 1
        else:
            compteurs[carte] = 1
>>> compteurs
{'As': 5, 'Dame': 4, 'Roi': 1}
```