Cours

Types construits

Qu'est-ce ?

Par type construit, on entend, tout objet composé de plusieurs objets simples

Tuples

Qu'est-ce qu'un tuple ?

Un *tuple* est une série de valeurs séparées par des virgules.

Exemple : `tup = (1, 2, 3)` ou `tup = ('a', 'b', 'c')`

En python, les tuples peuvent être constitués de valeurs de type différent.

Manipuler les tuples

Les tuples ne sont pas mutables : on ne peut en changer le contenu

On accède à un élément par son indice :

```
>>> tup = ('a', 'b', 'c')
>>> tup[2]
'c'
```

Fonction qui renvoie un tuple

Une fonction peut renvoyer un tuple !

```
def oppose_vecteur(x, y):
    return -x, -y
```

et cela donne :

```
>>> oppose_vecteur(1, 3)
(-1, -3)
```

Tableaux : le type `list`

Qu'est-ce qu'un tableau ?

Un **tableau** est une collection *numérotée* et *mutable* d'objets.

- Numérotée : chaque élément dispose d'un indice qui permet de le retrouver rapidement
- Mutable : Contrairement aux tuples, on peut en changer le contenu. On peut aussi ajouter ou retirer des éléments à un tableau.

En python, pour représenter les tableaux on utilise le type `list`

Abstrait vs concret

Un *tableau* désigne une structure de donnée *abstraite*, disposant d'un minimum d'opérations possibles (appelées *primitives*).

Il existe énormément de structures de données différentes, généralement introduites pour résoudre efficacement un problème.

Lorsqu'on programme une telle structure de données abstraite (on parle *d'implémentation*) on ne respecte pas toujours ces contraintes théoriques :

- parfois on ajoute des méthodes particulières par commodité,
- parfois ce n'est pas possible ou pas efficace.

Il convient donc de distinguer la théorie (ex: les tableaux) de la pratique (ex: les `list` Python).

Le type abstrait "tableau"

Un tableau est donc : une collection numérotée et mutable d'objets d'un même type (ex. tous entiers).

Primitives on doit pouvoir :

- le créer vide : `T = []` ou le créer avec des valeurs : `T = [5, 1, 3]`
- accéder à un élément par son indice : `T[0] -> 5`
- mesurer sa longueur : `longueur([5, 1, 3]) = 3`
- ajouter un élément au tableau, généralement à la fin : `ajouter(T, 7)` et `T = [5, 1, 3, 7]`
- retirer un élément du tableau, généralement par son indice : `retirer(T, 0) = 5` et `T = [1, 3, 7]`

Contrainte importante d'implémentation Pour être efficace, une implémentation d'un tableau doit permettre d'accéder à un élément *en temps constant*.

Accéder au premier ou 1327^{eme} élément prend exactement autant de temps.

Comparaison Tableau vs `list` Le type `list` de Python implémente toutes ces opérations mais bien d'autres encore.

Les `list` Python ne sont pas *exactement* des tableaux...

- elles peuvent contenir des éléments de type différents,
- on peut insérer, retirer où on veut dans les `list`

Remarque importante sur les types abstraits

L'informatique est une science récente et *ses définitions varient beaucoup d'un auteur à l'autre.* Si l'on a besoin d'être précis, il faut définir exactement les notions avant d'exposer quelque chose.

Ce n'est pas le cas en mathématiques ou en physique, par exemple. La notion de *masse* est la même pour tous, une *fonction* mathématique a une définition précise identique pour tous les auteurs. En informatique, si l'on vous parle de *tableau* il faut faire attention à ce que l'auteur à défini un peu plus tôt.

`list` construites à la main

On peut créer, de plusieurs manières un tableau :

```
>>> tab = ["pierre", "paul", "jacques"]
>>> tab[1]
"paul"
```

À l'aide d'une boucle :

```
>>> tab = [] # tableau vide
>>> for i in range(5): # i de 0 à 4
...   tab.append(i ** 2) # ajouter un élément à la fin de tab
>>> tab
[0, 1, 4, 9, 16]
```

`list` construites par compréhension

Il existe une manière beaucoup plus simple d'écrire les tableaux : par compréhension

Pour construire la liste des carrés des entiers de 0 à 4 :

```
[i ** 2 for i in range(5)]
```

soit :

```
[0, 1, 4, 9, 16]
```

Liste par compréhension complexe

On peut imbriquer plusieurs boucles ou ajouter des conditions :

`[carres des entiers inférieurs à 10 et multiples de 3] = [0, 9, 81]`

En python :

```
[i ** 2 for i in range(10) if i % 3 == 0]
```

soit

```
[0, 9, 81]
```

`i % 3` est le reste de la division de i par 3 (se lit `i` *modulo* `3`).

Dictionnaire

Qu'est-ce qu'un dictionnaire ?

Un dictionnaire est un enregistrement de **valeurs** associées à des **clés**.

Exemple : répertoire téléphonique

Nom	Téléphone
Marcel	0320666666
Robert	0320123456
Amandine	0320987654

Dictionnaire par clés et valeurs

En python cela donne :

```
tel = {
  "Marcel":  "0320666666",
  "Robert":  "0320123456",
  "Amandine":  "0320987654",
}
```

Accéder à une valeur

On accède à une **valeur** par sa **clé**

`dictionnaire[cle] ----------> valeur`

```
>>> tel["Amandine"]
"0320987654"
```

Dictionnaire : mutable

Les dictionnaires sont mutables.

Si Robert change de numéro :

```
tel["Robert"] = "0320445566"
```

Remarquez bien la différence de syntaxe : on utilise `:` pour déclarer le dictionnaire et `=` pour changer une valeur

---

Type abstrait : les tableaux associatifs

Parfois appelés *table d'association* ou simplement *dictionnaires*.

C'est une collection de paires (`cle`, `valeur`). Chaque clé est associée à une valeur.

Cela correspond donc à une *application* en mathématique.

Primitives Généralement on attend les opérations :

- créer vide,
- savoir s'il est vide
- **rechercher une valeur par sa clé** : c'est l'opération la plus importante et elle doit être très rapide.
- ajouter une paire (`clé`, `valeur`),
- modifier une valeur associée à une clé,
- supprimer une paire (`clé`, `valeur`).

tableau associatif vs `dict` Les `dict` emploient une *fonction de hachage*, appelée `hash` qui prend un objet (**non mutable**) et lui associe un entier.

La contrainte immédiate et qu'on ne peut insérer dans un `dict` que des clés non mutables.

Ecrire une telle fonction de hachage est un exercice délicat.

Il existe une autre approche, totalement différente, utilisant des *arbres équilibrés* (HP NSI).

tableau vs dictionnaire Dans un *tableau*, les éléments sont **numérotés**. Dans un *dictionnaire*, les éléments sont **repérés par une clé**.

- Si l'on veut itérer (= faire des boucles) : tableau.
- Si l'on veut connaître une valeur par sa clé ou savoir si élément figure dans la collection rapidement : dictionnaire.

En Python, une `list` est assez légère tandis qu'un `dict` est plutôt lourd.

Itérer

Collections

En python (mais aussi dans beaucoup de langages), les `str`, `tuple`, `list` et `dict` sont des collections. Cela signifie qu'on peut itérer dessus.

Itérer sur une collection : *parcourir à l'aide d'une boucle*

On peut écrire des boucles `for element in objet_construit:`

Cas simple

Pour les :

- chaînes de caractères,
- tuples,
- listes Python (=tableau)

La syntaxe est la même et `element` désigne l'objet contenu dans `objet_construit`

```
>>> chaine = "aZe"
>>> for lettre in chaine:
...   print(lettre)
a
Z
e
```

```
>>> tuple = (6, 4, 2)
>>> for t in tuple:
...   t ** 2
36
16
4
```

```
>>> liste = [a-1 for a in range(3)] # [-1, 0, 1]
>>> for x in liste:
...    x + 2
1
2
3
```

Cas particulier propres aux dictionnaires

Il existe plusieurs manières d'itérer sur un dictionnaire.

Mais ATTENTION dans **Python < 3.6** il n'y a pas d'ordre particulier.

Itération simple :

```
tel = {
  "Marcel":  "0320666666",
  "Robert":  "0320123456",
  "Amandine":  "0320987654",
}
```

```
>>> for personne in tel:
...  tel[personne]
"0320666666"
"0320123456"
"0320987654"
```

Itération avec `.keys()`

`keys()` : collection des clés (les noms dans l'exemple plus haut.)

```
>>> for personne in tel.keys():
...  tel[personne]
"0320666666"
"0320123456"
"0320987654"
```

C'est identique à l'itération normale !

Itération avec `.items()`

`items()` : collection des TUPLES (clé, valeur)

```
>>> for personne, tel in tel.items():
...  print("le numéro de ", personne, " est ", tel)
```

```
Le numéro de Marcel est 0320666666
Le numéro de Robert est 0320123456
Le numéro de Amandine est 0320987654
```

Itération avec `.values()`

Cette fois on ne récupère que les *valeurs*. C'est la moins utilisée en pratique.

Dictionnaire par compréhension.

On peut créer des dictionnaires par compréhension :

```
>>> carres = {a: a ** 2 for a in range(4)}
>>> carres
{
  0: 0,
  1: 1,
  2: 4,
  3: 9
}
```

`list` & `dict` Python : comparaison

Ces deux structures ont des points communs :

- Collection d'objets
- Accéder à un élément avec `collection[ obj ]`
- Effacer avec `del collection[ ob ]`
- Itérer avec `for obj in collection: ...`

Mais aussi des différences majeures

- Les `dict` sont beaucoup plus complexes que les `list`
- L'itération est rapide pour une `list`, très lente pour un `dict`
- Les `dict` n'ont pas d'ordre : pas de premier, second, dernier élément...

Vitesses

Généralement, les `list` sont beaucoup plus rapides que les `dict`.

Une exception majeure, l'appartenance : `obj in collecion`. Plus rapide pour les dict que les list.

Pour les `list` il faut tout parcourir et comparer élément par élément, pour les `dict` c'est en temps constant.

Implantation

Les `list` python sont des *tableaux dynamiques* mutables qui acceptent des éléments de n'importe quel type.

Les `dict` python sont des *tableaux d'associations* mutables dans lesquels on ne peut ajouter que des clés *non mutables*. Par exemple `(1, 2)` peut être un clé de dictionnaire mais `[1, 2]` ne le peut.

Le type `dict` fait l'objet d'un chapitre complet de terminale aussi on s'arrête là.