Résumé

qkzk

Type abstrait : tableau

Type abstrait

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

On doit pouvoir :

- les créer, vide ou avec des valeurs,
- mesurer leur taille,
- accéder à un élément par son indice,
- modifieur leur contenu (ajouter, retirer, changer un élément) par leur indice

Implémentation Python : `list`

L'implémentation la plus fidèle de ce type abstrait en Python est le type `list`

```
# créer
tab = [3, 5, 1, 7] # une list de longueur 4

# longueur
print("longueur", len(tab)) # 3

# accéder par un indice
x = tab[2] # 1
print("element d'indice 2", x)

# modifier
tab[2] = 0 # [3, 5, 0, 7]
print("apres modification d'un élément", tab)

# ajouter à la fin
tab.append(9) # ne renvoie rien !!!
print("apres ajout d'un élément", tab)

# retirer le dernier
last = tab.pop() # sort et renvoie le dernier élément
print("le dernier était", last)
print("apres retrait du dernier élémént", tab)

# insérer par exemple au début
tab.insert(0, 11) # ne renvoie rien
print("après ajout d'un élément en tête", tab)

# contient ?
print("tab contient 5 ?", 5 in tab)
```

Variantes : `tuple`, `str`

Les `tuple` c'est comme les `list` mais non mutable et plus rapide.

Une fois crées on ne peut les modifier.

```
# création avec des ( ) plutôt que des [ ]
potes = ("Pierre", "Paul", "Jacques")
print(potes)

# accéder
print("premier", potes[0])

# contient ?
print("Frank est mon pote ?", "Franck" in potes)


# modifier : plantage commenter cette ligne !!!
potes[0] = "Fanny"
```

Les `str` pour *string* ou *chaînes de caractères* se comportent comme des `tuple`.

On peut les créer, accéder aux élément, tester la présence d'une sous-chaîne, itérer etc. mais pas les modifier.

```
# créer
phrase = "Tant va la cruche à l'eau..."
print(phrase)

# accéder
print("Troisieme lettre", phrase[2])

# présence ?
print("La phrase contient le mot eau ?", "eau" in "phrase")
```

Type abstrait : tableau associatif

Type abstrait

Un *tableau associatif* ou *dictionnaire* est une collection qui associé des clés à des valeurs.

L'intérêt est de pouvoir accéder *en temps constant* à une valeur par sa clé et de tester, toujours en temps constant, la présence d'une valeur dans la clé.

Ce sont des collections mutables, on peut changer leur contenu.

Implémentation Python : `dict`

L'implémentation la plus fidèle en Python est le type `dict`

```python
# créer : { } et : entre clé et valeur
questions = {
    "1+1 =": 2,
    "3-5 =": -1000,
    "127 % 2 =": 1,
    "13 // 3 =": 4,
}
# on va rectifier l'erreur un peu plus tard...
print(questions)

# accéder
print("1+1 =", questions["1+1 ="])

# modifier : rectifions l'erreur
questions["3-5 ="] = -2
print(questions)

# contient ?
print("1+1 =" in questions)
```

Itération

sur les `str`, `tuple`, `list`

La même syntaxe pour les trois :

- sur les valeurs directement (on n'a pas besoin de l'indice):

```
for elt in collection:
    faire un truc à elt
```

```python
total = 0
for x in [12, 13, 14]:
   total = total + x
print(total) # 39

for ami in ("Pierre", "Paul", "Jacques"):
   print(ami, "est mon pote")

for lettre in "aeiouy":
   print(lettre, "est une voyelle")
```

- sur les indices (lorsqu'on en a besoin)

```
for i in range(len(collection)):
    val = collection[i]
    faire un truc à val ou à i
```

```python
temperatures = [12, 20, 14, 17]
for i in range(len(temperatures)):
    temp = temperatures[i]
    print("jour", i, "températeure", temp)

voyelles = "aeiouy"
for i in range(len(voyelles)):
    lettre = voyelles[i]
    print(i, lettre)
```

sur les `dict`

```python
ages = {"pierre": 4, "rémi": 2, "yolande": 12}

for personne in ages:
    print(personne, ages[personne])

print("anniversaires !")

for personne in ages.keys():
    ages[personne] += 1
    print(personne, "a", ages[personne], "ans")
```

```python
ages = {"pierre": 4, "rémi": 2, "yolande": 12}

for personne, age in ages.items():
    print(personne, age)
```

```python
ages = {"pierre": 4, "rémi": 2, "yolande": 12}

for age in ages.values():
    print("mon ami a", age, "ans")
```

Il n'y a pas d'indice dans un dict donc pas de :

```python
dictionnaire = {"a": 1, "b": 2}

for i in range(len(dictionnaire)):
    val = dictionnaire[i]
    # provoque une KeyError
```

Comparaison `list` vs `dict`

Avec :

```python
tab = [12, 10, 9, 20, 14]

repertoire = {
    "Paul": "0775654412",
    "Jacques": "0677744111",
    "Martine": "0122334455"
}

# notez la syntaxe différente :
# list: [ ... ]
# dict: { ... } et : entre les clés et valeurs
```

- Les `list` sont plus légers que les `dict`,
- `tab[4]` : on accède aux éléments d'une `list` par leur indice,
- `repertoire["Jacques"]` : on accède aux valeurs des `dict` par leur clé,
- `9 in tab` : fonctionne mais est lent. Python doit vérifier un par un les éléments,
- `"Paul" in repertoire` : très rapide, temps constant,
- `for elt in tab` : rapide,
- `for cle in dictionnaire` : lent.

`list` et `dict` par compréhension

`list` par compréhension

```python
doubles = [2 * i for i in range(10)]
print(doubles)
```

```python
doubles = [2 * i for i in range(10) if i % 2 == 0]
print(doubles) # il n'y a que 5 éléments !!!!
```

```python
produits = [i * j for i in range(5) for j in range(5)]
print(produits)
```

```python
produits = [[i + j for i in range(4)] for j in range(6)]

# pour afficher un peu mieux
from pprint import pprint
pprint(produits)
```

`dict` par compréhension

Même syntaxe.

```python
carres = {x: x ** 2 for x in range(6)}
print(carres)

triple_des_impairs = {x: 3 * x for x in range(10) if x % 2 == 1}
print(triple_des_impairs)
```