Initiation interactive à Python 5 : Listes

Aller vers Initiation interactive à Python 4 : les boucles

Cette page n'est pas tout à fait terminée 😎

Les listes sont les *conteneurs* Python les plus utilisés. Elle permettent de stocker de manière séquentielle des références vers d'autres objets : des nombres, des chaînes de caractères etc...

On peut ensuite accéder à un objet de la liste en donnant son numéro d'ordre (en faisant attention car la numérotation commence à 0).

Voici quelques exemples de manipulations de listes :

```
l = [2, 3, 5, 7, 11, 13]
print("Le plus petit nombre premier est : ", l[0])
print("Le sixième nombre premier est : ", l[5])
l.append(17)
print("Le septième est :", l[6]
```

Les listes sont des objets *mutables* ce qui signifie qu'on peut modifier les valeurs qu'elles contiennent.

On peut modifier le premier élément de la liste (par exemple) ainsi :

```
l[0] = 42
```

Il y a un certain nombre d'opérations disponibles sur les listes :

```
l = [1, 2, 5, 3, 1, 13, 8]
n = len(l) # nombre d'éléments de la liste
print("La liste",l,"contient",n,"éléments")
l.sort() # trie la liste
print(l)
```

Voici quelques autres opérations réalisables avec des listes :

Méthode	**Type retour**	**Description**
`list()`	list	Renvoie une liste vide
`[item[,item]+]`	list	Liste contenant les *items* donnés entre crochets
`del x[i]`		Efface l'objet en position `i` de `x`
`x[i]=e`		Affecte l'item `e` à la position `i` de `x`. L'item `i` doit déjà exister (la liste n'ets pas étirée).
`x+=y`		Modifie la liste `x` en lui ajoutant les éléments de `y`
`x*=i`		Modifie la liste `x` pour qu'elle contienne `i` copies concaténée de la liste originale.

Méthode	**Type retour**	**Description**
`x.append(e)`	None	Modifie `x` en lui ajoutant `e` comme dernire élément.
`x.count(e)`	int	Retourne le nombre d'apparitions de `e` dans `x`. Les apparitions sont détectées en utilisant ==.
`x.extend(iter)`	None	Modifie `x` en lui ajoutant les éléments de l'itérable `iter`.
`x.index(e,[i,[j]])`	int	Retourne l'indice du premier élément de `x` égale à `e` (au sens de ==), situé entre les indices `i` et `j-1`. L-ve l'exception `ValueError` si un tel élément n'existe pas.
`x.insert(i, e)`	None	Modifie `x` en insérant `e` de tel sorte qu'il se trouve à l'indice `i`.
`x.pop([index])`	item	Supprime (modifie donc `x`) et renvoie l'élément en position `index`. Si `index` n'est pas précisé, il est pris par défaut égal à `len(x)-1`.
`x.remove(e)`	None	Modifie `x` en supprimant la première occurence de `e`. Lève l'exception `ValueError` si `e` n'est pas trouvé.
`x.reverse()`	None	Modifie `x` en renversant l'ordre de ses éléments.
`x.sort()`	None	Modifie `x` pour qu'il soit trié en utilisant la méthode de comparaison `__cmp__` de ses éléments. La méthode accepte deux paramètres optionels : `key` et `reverse`. Si `reverse` vaut `True`, le tri est fait à l'envers (décroissant). Le paramètre `key` est une fonction qui prend en paramètre un élément et renvoie la valeur qui sera utilisée réellement pour le tri.

Comme d'autres conteneurs Pythons, les listes font partie de la famille des séquences. Toutes les types séquence (les listes, mais aussi les tuples, les chaînes de caractères...) disposent des opérations suivantes :

Méthode	**Type retour**	**Description**
`list(sequence)`	list	Renvoie une liste contenant les objets de `sequence`
x==y	bool	Retourne `True` si `x` et `y` contiennent les même objets (au sens de == pour chaque paire d'objets)
x>=y	bool	Vrai si `x` est avant `y` dans l'*ordre lexicographique* ou si x==y. Faux sinon.
x⇐y	bool	Vrai si `x` est après `y` dans l'*ordre lexicographique* ou si x==y. Faux sinon.
x>y	bool	Vrai si `x` est avant `y` dans l'*ordre lexicographique*. Faux sinon
x<y	bool	Vrai si `x` est après `y` dans l'*ordre lexicographique*. Faux sinon
x!=y	bool	Vrai si `x` et `y` sont de longueur différente ou si un item de `x` est différent d'un item de `y`. Faux sinon.
`x[i]`	item	Retourne l'item en position `i` de `x`. Si `i` est strictement négatif, vaut `x[len(x)+i]`
`x[i:j]`	list	Retourne la tranche d'items de la position `i` à la position `j-1`.
`x[:j]`	list	Retourne la tranche d'items de la position `0` à la position `j-1`.
`x[i:]`	list	Retourne la tranche d'items de la position `i` à la fin de la liste.
`iter(x)`	iterator	Retourne un itérateur sur `x`
`repr(x)`	str	Représentation *officielle* de `x` sous forme de chaîne de caractères

Aller vers Initiation interactive à Python : Défis de programmation - 1

Construire une suite

Une utilisation courante des listes est de stocker les différentes termes d'une suite. Reprenons l'exemple de la suite de Fibonacci. Chaque terme est le somme des deux précédents et les deux premiers termes sont 0 et 1.

On peut initialiser la liste des termes ainsi :

```
fibo = [0, 1]
```

Le premier terme est :

```
>>> fibo[0]
0
```

et le second est :

```
>>> fibo[1]
1
```

Le terme suivant est la somme de `fibo[0]` et de `fibo[1]` :

```
>>> fibo[0] + fibo[1]
1
```

Pour ajouter cette nouvelle valeur en fin de liste, on peut utiliser append :

```
>>> fibo.append(fibo[0] + fibo[1])
>>> fibo
[0, 1, 1]
```

On peut continuer sur notre lancée :

```
>>> fibo.append(fibo[1] + fibo[2])
>>> fibo
[0, 1, 1, 2]
```

Voyez comme on calcule les 30 permiers termes de la suite en exécutant le programme suivant :

L'erreur la plus courante est certainement de se tromper dans les indices des éléments de la liste. Si par exemple vous indiquez `fibo[i] + fibo[i-1]` à la place de `fibo[i-1] + fibo[i-2]`, ça ne fonctionnera plus. Essayez de bien comprendre pourquoi.

Les indices négatifs permettent d'accéder aux élément d'une liste en partant de la fin. L'indice -1 désigne le dernier élément, -2 l'avant dernier etc... Comme dans l'exemple précédent, nous voulons toujours sommer les deux derniers éléments ajoutés, on peut remplacer `fibo[i-1] + fibo[i-2]` par `fibo[-1] + fibo[-2]` ce qui veut littéralement dire : la somme des deux derniers éléments. À l'usage, c'est plus lisible.

Les nombres de Jacobsthal sont obtenus d'une manière presque identique aux nombre de Fibonacci. Nous avons aussi J(0)=0 et J(1)=1 mais ensuite, chaque nombre est obtenu comme la somme du nombre qui le précède par le double de celui qui le précède encore :

- J(2) = J(1) + 2* J(0) = 1
- J(3) = J(2) + 2* J(1) = 3
- ..

On obtient ainsi : 0, 1, 1, 3, 5, 11, 21...

Modifiez le programme qui précède pour calculer J(20). Vous devez trouver : 349525

Essayez de n'afficher que ce résultat, et non toute la suite.

Combien vaut J(60) ?