TP 3 : instructions de bouclage et listes

Lorsqu'on veut faire répéter à un ordinateur une instruction donnée un grand nombre de fois, on veut éviter d'avoir écrire cette instruction autant de fois dans l'interpréteur ou bien dans notre script. Il y a deux grandes catégories de répétitions :

— les répétitions conditionnelles : le bloc d'instructions est à répéter indéfiniment, tant qu'une certaine condition est vérifiée
— les répétitions inconditionnelles : le bloc d'instructions est à répéter un nombre de fois fixé à l'avance.

Instruction while

Nous commençons par les répétitions conditionnelles. On utilise le mot-clé `while` suivi de la condition que l'on veut tester, suivie de deux points. Vient ensuite, avec indentation, le bloc d'instruction à répéter. Par exemple si l'on veut écrire une fonction qui prenant en argument un entier n et renvoyant le plus petit entier k tel que $2^k > n$, on écrira :

```
def essai(n):
    k=0
    while 2**k <= n:
        k+=1
    return k
```

Voici le déroulement de l'exécution :
— on teste la condition
— si la condition est le booléen `True`, les instructions dans le bloc indentées sont exécutées, puis on revient au début de la boucle
— si la condition est le booléen `False`, on sort de la boucle

Attention : Si la condition ne devient jamais fausse, le bloc est répété indéfiniment ! Pour interrompre un programme mal conçu qui ne se termine pas, on fait la combinaison de touches Ctr-C. Essayez par exemple de terminer l'exécution de cette boucle :

```
def essai(n):
    k = 0
    while True:
        k += 1
    return k
```

Certains programmes peuvent avoir une boucle infinie (c'est-à-dire dont la condition ne devient jamais fausse) écrite dans une fonction, mais qui néanmoins se termine, grâce au mot-clé `return`. Voici à droite un exemple. Cette fonction a exactement le même effet que la première fonction de cette feuille de TP.

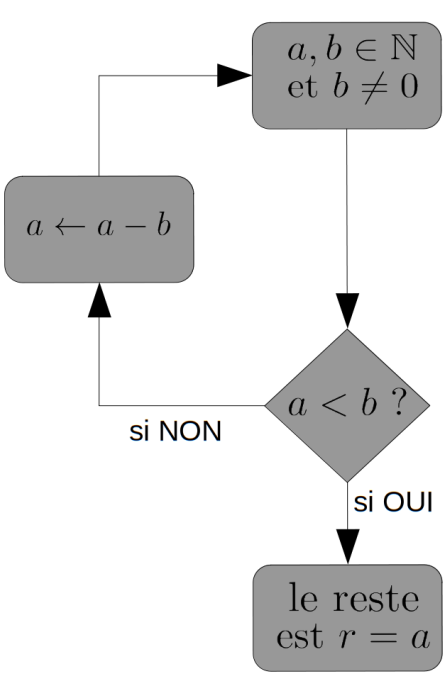
```
def essai(n):
    k = 0
    while True:
        if 2**k > n:
            return k
        k += 1
```

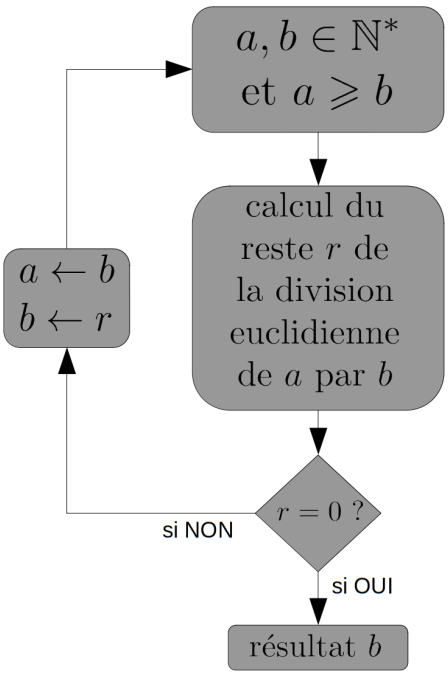
Exercice 1.

1. Écrire une fonction `somme(n)` qui renvoie la somme des carrés des n premiers entiers.
2. Écrire une fonction `depasse(M)` qui, pour tout entier M renvoie le plus petit entier n tel que $1^2 + 2^2 + \cdots + n^2 \geqslant M$.
3. Testez ces fonctions. Vous devez obtenir 4324 pour $n = 23$ dans la question 1. et $n = 718$ pour $M = 123456789$ dans la question 2.

Exercice 2.

Voici deux algorithmes présentés sous forme de « flowchart »(diagramme de flux).





1. Que calcule le premier algorithme ? Si vous ne devinez pas, exécutez avec papier crayon l'algorithme pour quelques valeurs de a et b.
2. Codez cet algorithme dans une fonction prenant a et b comme argument et renvoyant r.
3. Que calcule le deuxième algorithme ? (plus difficile, et il n'est pas certain que vous deviniez en faisant des essais si vous n'avez pas fait maths expertes !)
4. Codez cet algorithme dans une fonction prenant en argument a et b et renvoyant le résultat b. Vous aurez besoin de la fonction de la question précédente, ou bien de l'opérateur %.
5. Testez vos deux fonctions. Le résultat de la première dans le cas $a = 123456789$ et $b = 23456$ est 7861. Le résultat de la deuxième fonction pour $a = 123456789$ et $b = 23456$ est 1.

Listes en Python

Nous introduisons ici un nouvel objet Python : la **liste**. Les boucles sont utiles pour faire traiter un grand nombre d'instructions par un ordinateur. Mais si nous avons un grand nombre de **données** à faire traiter par l'ordinateur, il sera très pénible d'affecter à chaque donnée une variable : cela nécessitera autant d'instructions d'affectation que nous avons de données à disposition. Par exemple pour calculer la moyenne à un examen d'une classe, il faut ajouter les notes de chaque étudiant et diviser par le nombre de notes. Si il y a 37 notes, nous aurons besoin de 37 variables `n1`, `n1`, ..., `n37`, puis de 37 affectations `n1=...`, `n2=...`, ..., `n37=...` avant d'enfin pourvoir exécuter l'instruction `(n1+n2+...+n37)/37` qui calcule la moyenne.

Il est bien plus avantageux d'utiliser un seul objet, une **liste** de valeurs, chaque note étant repérée par un indice. Le type d'objet correspondant est le type `list`. Par exemple :

```
>>> etudiants = ['ECG3', 12, 17, 13.6, 14]
>>> etudiants, id(etudiants), type(etudiants)
(['ECG3', 12, 17, 13.6, 14], 140423365213760, list)
```

Une liste est une succession d'éléments, rangés dans un ordre fixé. De plus en Python, les différents éléments de la liste ne sont pas nécessairement du même type : dans l'exemple précédent, le premier élément est de type `str`, les deuxièmes troisièmes et cinquièmes de type `int` et le quatrième de type `float`.

La première chose que l'on peut vouloir faire avec une liste est d'en extraire un élément. Un utilise alors la syntaxe `etudiants[2]`, où 2 est l'indice de l'élément voulu dans la liste. Par exemple, toujours avec la liste de l'exemple précéedent

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste[2]
18
```

Le résultat peut paraître surprenant. Mais les éléments d'une liste en Python (et dans beaucoup de langages de programmation) sont indexés **à partir de 0** et non à partir de 1. Exemples :

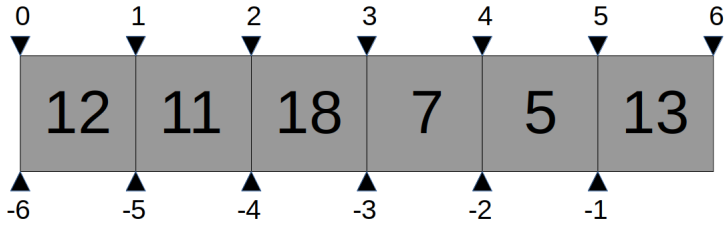
```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste[0], liste[1], liste[4], liste[5]
(12, 11, 15, 3)
>>> list[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Si on tente d'extraire de la liste un élément avec un indice dépassant la taille de la liste, le message d'erreur est très clair : `IndexError: list index out of range`. La longueur d'une liste nommée `nom-de-liste` peut être obtenue à l'aide de l'instruction `len(nom-de-liste)`.

On peut extraire une partie de la liste (ce qui donnera à nouveau une liste) en déclarant l'indice de début souhaité et l'indice de fin souhaité. J'appellerai ceci le « saucissonnage de liste »pour des raisons qui seront un peu plus claires ensuite. Pour extraire les éléments numéros trois quatre et cinq, on essaie naturellement de donner en indice de départ l'indice 2 (puisque les indices commencent à 0, l'indice 2 correspond bien au 3ème élément) et en indice de fin l'indice 4 (qui correspond bien au cinquième élément) :

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste[2:4]
(18, 7)
```

À nouveau une surprise : nous n'obtenons pas le résultat attendu. En fait, il faut penser aux éléments comme des tranches de saucisson, et aux indices comme l'endroit où l'on doit mettre un coup de couteau pour obtenir la tranche souhaitée. Le petit schéma suivant représente ceci clairement.



On peut même se servir d'indices négatifs. Les résultats suivants doivent devenir clairs.

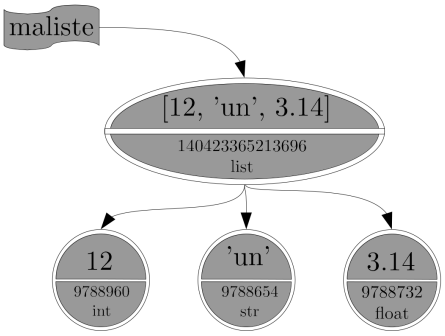
```
>>> liste[2:]
[18, 7, 15, 3]
>>> liste[:2]
[12, 11]
>>> liste[0:len(liste)]
[12, 11, 18, 7, 15, 3]
>>> liste[:]    # meme resultat
[12, 11, 18, 7, 15, 3]
>>> liste[2:5]
[18, 7, 15]
```

```
>>> liste[2:7]
[18, 7, 15, 3]
>>> liste[-2:-4]
[]
>>> liste[-4:-2]
[18, 7]
>>> liste[len(liste)-1]
3
>>> liste[-1]   # meme resultat
3
```

Important à noter : les dépassements d'indice sont tolérés lorsqu'on utilise le saucissonnage . Notez l'apparition de la liste vide `[]`, qui nous sera bien souvent utile. Notez finalement que l'indice -1 est l'indice du dernier élément, ce qui permet parfois d'éviter d'avoir recours au calcul de la longueur de la liste par `len`.

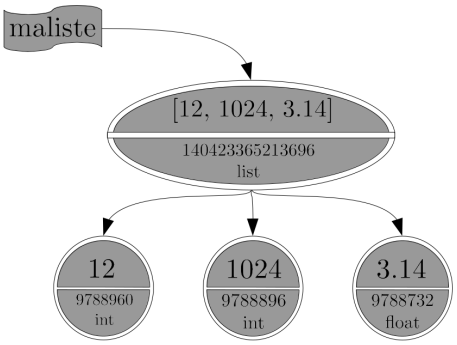
Intéressons nous maintenant à la position en mémoire des éléments d'une liste.

```
>>> maliste = [12, 'un', 3.14]
>>> liste, id(liste), type(liste)
([12, 'un', 3.14], 140423365213696, list)
>>> maliste[0], id(maliste[0]), type(maliste[0])
(12, 9788960, int)
>>> maliste[1], id(maliste[1]), type(maliste[1])
('un', 9788654, str)
>>> maliste[2], id(maliste[2]), type(maliste[2])
(3.14, 9788732, float)
```



Une liste est en fait un carnet d'adresses : elle contient les emplacements mémoire des différents éléments de la liste. Une propriété remarquable des listes, propriété que ne possèdent pas les `tuple`, est le fait que l'on peut modifier un ou plusieurs éléments de la liste sans modifier son emplacement mémoire et donc son identifiant. Voici un exemple :

```
>>> maliste[1] = 2**10
>>> liste, id(liste), type(liste)
([12, 'un', 3.14], 140423365213696, list)
>>> maliste[0], id(maliste[0]), type(maliste[0])
(12, 9788960, int)
>>> maliste[1], id(maliste[1]), type(maliste[1])
(1024, 9788896, int)
>>> maliste[2], id(maliste[2]), type(maliste[2])
(3.14, 9788732, float)
```



La liste n'a pas changé d'identifiant, on a simplement modifié sa **valeur** : le contenu de la liste est toujours à la même adresse dans la mémoire. Les concepteurs de Python on prévu ce comportement afin d'éviter que de grandes listes soient déplacées dans la mémoire à chaque modification d'un seul élément. On dit que le type `list` est **mutable**. Le type `tuple` n'est quand à lui pas mutable. Voyez par exemple ce que donne une tentative de modification d'un élément d'un `tuple` :

```
>>> a = (1,2,3,4)
>>> a, id(a), type(a)
((1, 2, 3, 4), 139650946155440, tuple)
>>> a[2] = 34
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
    assignment
```

Une modification d'un objet **non mutable** aura pour conséquence la désignation d'un nouvel emplacement mémoire. Voyez par exemple dans l'exemple précédent le changement d'identifiant de `maliste[1]`. En effet cet objet est de type `int` donc non mutable. La modification de sa valeur entraîne la modification de son emplacement mémoire.

Les techniques de saucissonnage permettent de modifier les listes à souhait.

```
>>> liste = [12, 11, 18, 7, 5, 13]
>>> liste[2:5] = ['a', 'b', 'c']
>>> liste
```

Le problème des copies de listes

On peut vouloir créer une copie d'une liste. Mais cela nous amène à quelques surprises.

```
>>> liste = ['HEC', 'ESSEC', 'ESCP']
>>> copie = liste    # creation d'une copie
>>> liste[0] = 'EDHEC'   #modification de l'original
>>> liste, copie
>>> copie[1] = 'AUDENCIA'
>>> liste, copie
```

Une modification de la liste originale se propage à la copie, et une modification de la liste copie se propage à l'originale. Ce n'est pas conforme à nos habitudes Python : une affectation est une affectation une fois pour toute pas une identification comme en mathématiques. L'explication se trouve du côté des identifiants (c'est même ce problème de copies de listes qui nous oblige à nous pencher ainsi en détail sur ces questions d'identifiants).

Lors de l'affectation `copie = liste`, c'est l'adresse mémoire de la liste d'origine qui est stockée dans la nouvelle variable `copie`. Cette nouvelle variable pointe donc au même endroit dans la mémoire, et toute modification d'une liste affecte l'autre liste au même moment. On dit que l'on a créé un **alias** de liste. Ce mécanisme a été mis en place afin d'éviter d'avoir à souvent recopier des grandes listes dans la mémoire, ce qui est très coûteux en temps de calcul. Pour tenter d'éviter ce problème, on peut extraire d'abord **tous les éléments** de la liste grâce à l'instruction `liste[:]` puis on affecte le résultat dans la nouvelle variable `copie`.

```
>>> copie = liste[:]
>>> copie[1]='nouveau'
>>> liste, copie
>>> copie[1] = 'AUDENCIA'
>>> liste, copie
```

Le résultat semble satisfaisant, mais en réalité ne l'est pas. On dit qu'on a créé une **copie superficielle** (en anglais : « shallow copy ») comme nous pouvons le voir sur les exemples suivants :

```
>>> x = ['a', 'b', [4, 5, 6]]
>>> y = x              # creation d'un alias
>>> z = x[:]           # creation d'une copie superficielle
>>> x[0] = 'c'     # affectera x et y mais pas z
>>> z[2][2] = 765    # affecte z ET x ET y !!
>>> x, y
>>> z
```

Le problème est le même que dans la copie superficielle : puisque le troisième élément de la liste `x` est une liste, donc une adresse, c'est cette adresse qui est copiée dans la copie superficielle `z`. Et donc une modification d'un élément de cette liste via `z` affectera bien `x` aussi. Pour éviter ce problème, on réalise des **copies profondes** (en anglais « deep copy »). On procède ainsi :

```
>>> import copy
>>> x = ['a', 'b', [4, 5, 6]]
>>> y = x              # creation d'un alias
>>> z = x[:]           # creation d'une copie superficielle
>>> w = copy.deepcopy(x)  # copie profonde
>>> x[0] = 'c'     # affectera x et y mais pas z ni w
>>> z[2][2] = 765    # affecte x,y,z mais pas w
>>> x, y
>>> z
>>> w
```

Quelques outils pour les listes

Pour ajouter un élément à une liste, on utilise la méthode `append`. On procède ainsi :

```
>>> malist = [1, 2, 45, 'ok', 0.69]
>>> malist, id(malist)
>>> malist.append("toto")
>>> malist, id(malist)
```

C'est la première fois que nous rencontrons cette opération « point ». Certains objets Python (comme les listes) possèdent ainsi des **méthodes**. La méthode s'appelle **append**, on l'applique à l'objet `malist` avec comme paramètre effectif la chaîne de caractères `"toto"`. L'objet `malist` est alors modifié. Mais vous pouvez remarquer que son identifiant mémoire n'a pas changé (toujours pour économiser des déplacements en mémoire). Voici quelques méthodes utiles sur les listes (dans cet exemple la liste que l'on cherche à modifier s'appelle `malist`).

méthode	**effet**
`malist.append(x)`	ajoute l'élément `x` en fin de liste
`malist.extend(L)`	ajoute les éléments de `L` en fin de liste
`malist.insert(i, x)`	ajoute l'élément `x` en position i
`malist.remove(x)`	supprime la première occurence de `x`
`malist.pop(i)`	supprime l'élément d'indice i et le renvoie
`malist.index(x)`	renvoie l'indice de la première occurence de `x`
`malist.count(x)`	renvoie le nombre d'occurences de `x`
`malist.sort()`	modifie la liste en la triant
`malist.reverse()`	modifie la liste en inversant l'ordre de ses éléments

La méthode `sort` modifie la liste. Si l'on veut obtenir une nouvelle liste, triée, sans modification de l'ancienne, on peut avoir recours à la fonction **sorted**. Comparez les exécutions de ces deux codes.

```
>>> maliste = [78, 12, 0, 57]
>>> sorted(maliste)
```

```
>>> maliste = [78, 12, 0, 57]
>>> maliste.sort()
>>> maliste
```

Finalement, nous avons déjà rencontré la fonction **len** qui renvoie la longueur d'une liste. On peut aussi tester l'appartenance d'un élément à une liste grâce à l'opérateur **in**.

```
>>> maliste = [78, 12, 0, 57]
>>> len(maliste)
>>> 5 in maliste
>>> 0 in maliste
```