TP #1 — max ; recherche d'élément ; primalité & factorisation

Exercice 1. *Max & ordre, deux éléments*

1. Écrivez une fonction `maxnum` qui prend deux arguments supposés être des entiers et renvoie le plus grand d'entre eux (pour l'ordre (non strict) usuel sur les entiers).
2. Écrivez une fonction `orderpair` qui prend un unique argument supposé être une paire d'entiers, et qui renvoie cette même paire « ordonnée » (pour l'ordre (non strict) usuel sur les entiers) : le premier élément de la paire renvoyée doit être inférieur ou égal au second.
   Construction utile :
   ```
   p = 3, 4
   a, b = p
   a, b = b, a # échange en une fois les valeurs contenues
               # dans les variables a et b
   ```
3. Testez vos fonctions dans un interpréteur interactif (idéalement `ipython`) sur de petits exemples construits à la main.

Exercice 2. *Max dans une liste*

Le but de cet exercice est de rechercher de plusieurs façons l'élément maximum d'une liste. En réalité, cette fonction est déjà implémentée en python, et s'appelle tout simplement `max`. Elle n'est cependant pas au programme.

1. Écrivez une fonction `maxinlistiter` qui prend un argument supposé être une liste d'entiers supposée être non vide et qui renvoie l'élément maximum trouvé dans cette liste. Cette fonction doit utiliser une approche itérative, et non récursive.
   Constructions utiles :
   ```
   len(L) # renvoie la longueur de la liste L

   assert(be) # interrompt l'exécution si be vaut False
              # ne fait rien si be vaut True
              # (à utiliser en cas d'erreur irrécupérable)

   for x in L:
       # corps de boucle permettant
       # de traiter itérativement (et un à un)
       # tout élément x de la liste L
   ```

2. Écrivez une fonction `maxinlistrec` de même spécifications que `maxinlistiter`, qui utilise une approche récursive, et *peut* modifier son argument.
   Constructions utiles :

```
def funrec(arg1):
    # ...
    # corps de la fonction
    # ...
    b = funrec(a) # appel récursif
    # ...
```

```
L.pop() # si L est non vide, supprime le dernier
        # élément ajouté à L et le renvoie
```

3. Écrivez une fonction `maxfromsort` qui prend un argument supposé être une liste d'entiers supposée être non vide, trie cette liste renvoie l'élément maximum trouvé.
   Construction utile :

```
L.sort() # trie L pour un ordre usuel sur ses éléments
         # techniquement hors programme
```

4. (Si vous vous ennuyez : écrivez vous-même une fonction de tri **efficace**.)
5. Testez vos fonctions dans l'interpréteur interactif sur des listes de tailles variables. Qu'observez-vous pour `maxinlistrec` pour des listes « suffisamment grandes » (par exemple de plus de 10000 éléments) ?
   Constructions utiles :

```
L = [] # définit une liste L vide
```

```
L = [1, 2, 3] # définit une liste L contenant
              # 1, 2, 3
```

```
L = [1, 2, 3]*10 # définit une liste L contenant
                 # 10 fois 1, 2, 3
```

```
L = [e for x in s] # définit une liste L contenant les
                   # éléments définis par l'expression
                   # e faisant (possiblement)
                   # intervenir l'élément x de la
                   # valeur « itérable » s
```

```
L = [2*x for x in range(10)] # illustration
```

```
L.append(x) # ajoute l'élément x à la fin de L
```

Exercice 3. *Recherche d'élément dans une liste*

1. Écrivez une fonction `searchlin` qui prend deux arguments : un supposé être une liste d'entiers et un supposé être un entier, qui renvoie `True` si cet entier est présent dans la liste, et `False` sinon.
   Notez qu'on pourrait très facilement implémenter cette fonction comme :
   ```
   def searchlinPy(L, x):
       return (x in L)
   ```
   mais cette réponse n'est pas acceptable pour cette question !
2. Écrivez une fonction `searchdich` de même spécifications que `searchlin`, à la différence que son premier argument est *garanti être trié*. On attend de cette fonction qu'*elle ne parcoure **pas*** tous les éléments de la liste.
   CONSEIL : Vous pouvez définir une fonction auxiliaire :
   ```
   _searchdich(L, x, bot, top)
   ```
   qui se contente de chercher (récursivement) la présence de `x` entre les indices `bot` et `top`.
3. *(Optionnel.)* Écrivez une fonction `searchdiciter` similaire à `searchdich`, mais non récursive.
4. Testez vos fonctions et comparez leur performances dans le pire cas, sur des exemples de taille significative. Faites attention à ne pas mesurer le temps d'exécution de la construction des exemples !
   Constructions utiles :

```
%timeit es # évalue l'expression e ou exécute
           # l'instruction s, possiblement plusieurs
           # fois et en mesure le temps d'exécution
           # *spécifique à ipython*

# import de la fonction timeit du module timeit
# (à ne faire qu'une fois)
from timeit import timeit
# évalue l'expression ou exécute l'instruction es
# n fois et en mesure le temps d'exécution total
timeit('es', number=n, globals=globals())
```

Exercice 4. *Primalité & factorisation*

1. Écrivez une fonction `isprime` qui prend un argument n supposé être un entier positif et renvoie `True` (resp. `False`) s'il est premier (resp. composé). Vous pouvez vous contenter d'utiliser un algorithme naïf (qui est *très loin* d'être le meilleur connu) qui pour tout entier $1 < i \leq \sqrt{n}$ teste si i divise n.

Constructions utiles :

```
# import des fonctions ceil (partie entière
# supérieure) et sqrt du module math
# (à ne faire qu'une fois)
from math import ceil, sqrt

for i in range(a):
    # corps de boucle
    # dans lequel i prendra successivement
    # les valeurs 0...a-1
for i in range(a, b):
    # pareil, entre a et b-1
for i in range(a, b, c):
    # pareil, entre a et b-1 (pas nécessairement
    # atteint) par incrément de c

a % b # opérateur « modulo »
      # pour a et b entiers positifs, renvoie
      # le reste positif de la division de a par b
```

2. Écrivez une fonction `listprimes` qui prend un argument `n` supposé être un entier positif et renvoie une liste contenant tous les nombres premiers positifs inférieurs (stricts) à `n`. Utilisez pour cette fonction la construction `L.append` pour une liste `L`.
3. Écrivez une fonction `listprimes2` de même spécifications que `listprimes`, qui utilise la construction :

   `[x for x in s if be]`

   qui construit la liste formée des éléments de la « valeur itérable » `s` (par ex. `range(10)`) tels que l'expression booléenne `be` (qui peut faire intervenir `x`) vaut `True`.
4. Écrivez une fonction `listprimessieve` de même spécifications que `listprimes`, qui utilise pour algorithme le crible dit d'Érathostène : on construit une liste `P` de `n` valeurs booléennes dont les deux premières valent `False` (0 et 1 ne sont pas premiers) et les suivantes valent toutes initialement `True` (on ne sait pas encore lesquels des nombres suivants sont premiers ou pas). Puis pour chaque valeur `i` de 2 à `n - 1`, si `P[i] = True` alors on sait (par induction) que `i` est premier et que ses multiples `2*i`, `3*i` etc. ne le sont pas, et on modifie donc `P` de façon appropriée. Une fois `P` ainsi construite, il suffit d'extraire les valeurs appropriées.
5. Comparez expérimentalement le temps d'exécution de `listprimessieve` et `listprimes2` (n'« affichez » pas les résultats) pour d'assez grandes valeurs de `n` (par exemple entre 100 000 et 5 000 000). Êtes-vous surpris du résultat ?

6. Écrivez une fonction `facto` qui prend un argument supposé être un entier positif et qui renvoie une liste de ses facteurs premiers (avec multiplicité) représentés comme des paires `(p,e)`. Par exemple, cette fonction doit renvoyer les résultats suivants :

```
> facto(2)
[(2,1)]
> facto(60)
[(2, 2), (3, 1), (5, 1)]
> facto(1337133713371337)
[(7, 1), (17, 1), (73, 1), (137, 1),
                           (191, 1), (5882353, 1)]
```

   Vous pouvez vous contenter d'utiliser un algorithme naïf.
   N.B. : Contrairement au test de primalité, on ne connaît pas à ce jour d'algorithme vraiment efficace pour la factorisation d'entiers.
   Construction utiles :

```
a // b # pour a et b entiers positifs, renvoie
      # le quotient de la division de a par b
```

7. Écrivez une fonction `isprimefacto` qui prend en argument une liste du format renvoyé par `facto` et qui renvoie `True` (resp. `False`) s'il s'agit bien d'une factorisation en nombre premiers.
8. Écrivez une fonction `prod` qui prend en argument une liste du format renvoyé par `facto` et qui renvoie le nombre dont celle-ci représente la factorisation.
9. Utilisez vos fonctions `isprimefacto` et `prod` pour vérifier le bon fonctionnement de votre fonction `facto`.