INFORMATIQUE DTL 1 - CORRECTION

Exercice 0.1. *Lorsque l'on utilise un éditeur, l'ajout d'une parenthèse fermante surnuméraire est immédiatement signalé (mise en surbrillance sous IDLE).*

Nous allons construire un vérificateur de parenthésage qui fonctionne avec une pile.

a) On considère l'algorithme suivant qui prend en argument une expression parenthésée, la parcourt de gauche à droite :

- *A chaque parenthèse, crochet ou accolade ouvrante rencontrée, on empile la fermeture correspondante.*
- *A chaque parenthèse, crochet ou accolade fermante rencontrée, on dépile la fermeture correspondante.*

Faites cela avec les exemples qui suivent :

1. $(3 + 2) \times 6$
2. $((3 + 2) \times 6$
3. $(3 + 2) \times 6 - 10 \times (5 - 6)$
4. $((3 + 2) \times 6 - 10 \times (5 - 6)$
5. $((3 + 2) \times 6 - 10) \times (5 - 6)$
6. $(5 + 7] \times 12$

Que se passe-t-il dans chacun des cas ?

b) Vous disposez du module `MinPiles` qui contient les fonctions de base sur les piles (créer, empiler, dépiler...). Programmer l'algorithme qui précède.

```
from MinPiles import*

def identification(a):
    if a==str('(') or a==str('[') or a==str('{'):
        return('ouverture')
    if a==str(')') or a==str(']') or a==str('}'):
        return('fermeture')
    else:
        return('other')
```

```

def err_detect(calcul):
    pile=[]
    err=0
    for i in range(len(calcul)):
        if identification(calcul[i])=='ouverture':
            empiler(pile,calcul[i])
        if identification(calcul[i])=='fermeture':
            if est_vide(pile)==True:
                err=1
                return(err)
            else:
                if [pile[0],calcul[i]]==[str('('),str(')')] or [pile[0],calcul[i]]==[str(')'),str('(')]:
                    depiler(pile)
                else:
                    err=1
                    return(err)
    if est_vide(pile)==False:
        err=1
    return(err)

```

```

calcul=str(input('entrez le calcul a tester :'))
print('calcul= ',calcul)
err=err_detect(calcul)
if err==1:
    print('SYNTAX_ERROR')
else:
    print('parenthesage_OK')

```

Exercice 0.2. 1. On donne les programmes Python P0 et P1 suivants.

```

def P0(N) :
    if N == 1 :
        return False
    if N==2 :
        return True
    for d in range(2,N) :
        if N % d == 0 :
            return False
    return True

def P1(N) :
    if N == 1 :

```

```

        return False
    if N == 2 :
        return True
    for d in range(2,N) :
        if N % d == 0 :
            return False
    return True

```

Que renvoient les appels $P0(5)$, $P1(5)$ et $P0(9)$, $P1(9)$?
Dire en une phrase ce que fait chacune des fonctions $P0$ et $P1$?

On remarque que la seule différence entre $P0$ et $P1$ provient de la place du dernier `return T` rue qui est au niveau du dernier `if` dans $P0$ et au niveau de `for` dans $P1$.

Cela signifie que s'il passe le cap de $N = 1$ et de $N = 2$, $P0$ va renvoyer `T` rue d'es que 2 ne divisera pas N . Il n'effectuera pas le reste de la boucle pour $P0$.

Donc $P0(N)$ renvoie `False` si N est pair supérieur ou égal à 4 et renvoie `T` rue si N est impair supérieur ou égal à 3.

Quant à $P1$, si N n'a pas d'autre diviseur que 1 et lui-même, il renvoie `T` rue.

Ainsi $P1(N)$ renvoie `True` si N est un nombre premier et `False` si N n'est pas premier ou 'gal à 1.

On a alors :

$P0(5)$, $P1(5)$, $P0(9)$, $P1(9)$ qui renvoie :

`True True True False`

- En une phrase, dire ce que fait le programme Python $P2$, qui utilise le programme $P1$ précédent ?

```

def P2(N) :
    L = [ ]
    k = 0
    n = k * k + 1
    while n <= N :
        if P1(n) :
            L.append(n)
        k = k + 1
        n = k * k + 1
    return L

```

Que renvoie l'appel $P2(127)$?

$P2$ fournit la liste de tous les nombres premiers du type k^2+1 inférieurs ou égaux à N , où k est un entier non nul.

On a alors :

$P2(7)$, $P2(20)$, $P2(127)$, $P2(345)$ qui renvoie :

$[2, 5]$ $[2, 5, 17]$ $[2, 5, 17, 37, 101]$ $[2, 5, 17, 37, 101, 197, 257]$

3. Écrire une fonction **nextPrime** en langage Python qui prend un argument entier N et qui retourne comme valeur le premier nombre premier qui est strictement supérieur à N .

```
def nextPrime(N) :
    n = N + 1
    while P1(n) == False :
        n = n + 1
    return (n)
```

Par exemple : $\text{nextPrime}(1)$, $\text{nextPrime}(7)$, $\text{nextPrime}(216)$ renvoie :

2, 11, 223

4. On appelle couple de nombres premiers jumeaux toute liste $[p, q]$ telle que p et q soient deux nombres premiers vérifiant $p < q$ et $q = p + 2$. Par exemple, $[3, 5]$ et $[11, 13]$ sont des couples de nombres premiers jumeaux.

- (a) Écrire, à l'aide de la fonction **nextPrime** précédente, une fonction Python nommée **jumeau**, prenant comme argument un entier N et renvoyant le couple $[p, q]$ de nombres premiers jumeaux tels que p soit strictement supérieur à N et le plus petit possible. Par exemple **jumeau(5)** renvoie comme valeur :

$[11, 13]$.

```
def jumeau(N) :
    M = N
    while nextPrime(nextPrime(M)) != nextPrime(M) + 2:
        M = M + 1
    return [ nextPrime(M), nextPrime(nextPrime(M)) ]
```

jumeau(4), **jumeau(5)**, **jumeau(225)**, **jumeau(349)** renvoie :

$[5, 7]$ $[11, 13]$ $[227, 229]$ $[419, 421]$

- (b) Écrire avec les mêmes consignes une fonction, **lesJumeaux**, prenant en argument un entier N et renvoyant la liste de tous les couples de nombres premiers jumeaux $[p, q]$ tels que q soit inférieur ou égal à N .

Par exemple :

lesJumeaux(18) retourne : $[[3, 5], [5, 7], [11, 13]]$
 (Le couple $[17, 19]$ n'en fait donc pas partie.)

Il s'agit donc de calculer tous les **jumeau(n)** pour n variant de 1 à N mais en ne mettant pas les nombres **jumeau(n)[0]** et **jumeau(n)[1]** qui dépasseraient N .

```
def lesJumeaux(N) :
    L = [ ]
    for n in range(1, N + 1) :
        if jumeau(n)[1] <= N :
            L.append(jumeau(n))
    return L
```

Le problème, c'est que l'on affiche un certain nombre de doublons.
 Ainsi, **lesJumeaux(8)** renvoie :

$[[3, 5], [3, 5], [5, 7], [5, 7]]$

Nous tapons alors une fonction qui enlève les doublons dans une liste :

```
def Non_doublon(L) :
    LL = [L[0]]
    p = len(L)
    for n in range(0, p-1) :
        if L[n + 1] != L[n] :
            LL.append(L[n + 1])
    return (LL)
```

On transforme maintenant **lesJumeaux** :

```
def lesJumeaux(N) :
    L = [ ]
    for n in range(1, N + 1) :
        if jumeau(n)[1] <= N :
            L.append(jumeau(n))
    return Non_doublon(L)
```

lesJumeaux(70) renvoie :

$[[3, 5], [5, 7], [11, 13], [17, 19], [29, 31], [41, 43], [59, 61]]$