UNIVERSITÉ PAUL SABATIER — 2018-2019 / semestre 5
Licence L3 E — TD ordinateurs

# Séances 1 et 2

# Séance 1

## 1 Variables

Les noms de *variables* sont des noms que l'on choisit librement pour les objets que l'on va manipuler. Voici quelques exemples d'affectation de *variables* :

```
In [ ]: a=3
        b=3.0
        xy=-5.7
        xz=234567889
        je_ne_suis_pas_une_variable=23
        x=2,5,7
        xx="2,5,7"
        azerty="coucou"
        poiuyt='true'
        ma_variable=True
```

Pour s'assurer que nos variables existent bien, demandons leur valeur ! :

```
In [ ]: a,b,xy,xz,je_ne_suis_pas_une_variable,x,xx,azerty,poiuyt,ma_variable
```

On aurait pu créer ainsi un nouvel objet

```
In [ ]: nouvel_objet = a,b,xy,xz,je_ne_suis_pas_une_variable,x,xx,azerty,poiuyt,ma_variable
```

```
In [ ]: nouvel_objet
```

Si l'on veut faire apparaître les valeurs successives de nos variables, on voit que la commande suivante ne le permet pas :

```
In [ ]: a
        b
        xy
        je_ne_suis_pas_une_variable
        azerty
        ma_variable
```

Seule la valeur de la dernière variable demandée apparaît. On peut par contre demander (et on note, au passage, la possibilité avec le caractère # d'introduire des commentaires, c'est-à-dire, du texte qui n'est pas pris en compte dans l'exécution ; c'est très utile, voire indispensable quand un programme contient un nombre important de commandes) :

```
In [ ]: print(a)
        print(b)
        print(xy)
        print(je_ne_suis_pas_une_variable)  # ici, j'écris un commentaire...
                                            # et mon commentaire est aussi long que je
                    # veux, il peut même faire plusieurs lignes
                #(à condition de ne pas oublier le symbole # au début
        print(azerty)
        print(ma_variable)
```

On peut aussi vouloir plus de clarté ou de précision :

```
In [ ]: print('a =',a)
        print("b =",b)
        print(xy)
        print("je_ne_suis_pas_une_variable =",je_ne_suis_pas_une_variable)
        print(azerty)
        print("le texte ou commentaire que l'on veut",ma_variable)
```

On remarque la différence d'affichage :

```
In [ ]: print(azerty)
        azerty
```

Notons que tous les caractères ne sont pas permis dans les noms de variables :

```
In [ ]: nouvel-objet=7
```

Certains noms sont réservés :

```
In [ ]: if=28
```

Python fait la différence entre minuscules et majuscules :

```
In [ ]: valeur=3.1416
        print(valeur)
        print(Valeur)
```

et certains caractères (comme les accents) peuvent ne pas être bienvenus ; tout dépend de l'environnement, voici un test :

```
In [ ]: a="élémentaire, mon cher Benoît !"
        a
```

Ces quelques exemples montrent qu'il existe plusieurs sortes d'objets ; chaque objet a un type :

```
In [ ]: print(a,type(a))  # coome ça, ce sera plus clair
        print(b,type(b))
        print(xy,type(xy))
        print(xz,type(xz))
        print(x,type(x))
        print(xx,type(xx))
        print(azerty,type(azerty))
        print(je_ne_suis_pas_une_variable,type(je_ne_suis_pas_une_variable))
        print(poiuyt,type(poiuyt))
```

S'il ne connaît pas la variable, `Python` nous le dit :

```
In [ ]: print(babar,type(babar))
```

Voici donc déjà quatre sortes de types :

- `int`, `float` : type *numrique*
- `tuple`, `str` : type `sequence`. Une séquence est une collection ordonnée ; on verra aussi le type `list` et on reviendra très rapidement sur les types `sequence`, ils sont incontournables !!
- `bool` : type *boolen* ; les deux seules valeurs possibles sont `True` et `False`

Cette liste n'est pas exhaustive... Notons juste pour terminer cette introduction que, comme leur nom l'indique, le contenu d'une *variable* peut changer (réaffectation) :

```
In [ ]: print(a)
        a=18
        b=-6
        print(b)
        b=a
        print(b)
```

On n'oubliera pas que la notation des décimaux se fait avec le point :

```
In [ ]: d=3.4
        dd=3,4
        print(d,type(d))
        print(dd,type(dd))
```

Enfin, la bibliothèque `math` permet d'utiliser des décimaux célèbres ; exemples :

## 2 Opérations élémentaires

```
In [1]: pi = 256
        Pi=1000
        print("pi=", pi)
        print("Pi=", Pi)
        print("est-il vrai que pi=256 ?", pi==256)
        from math import *
        print("pi=", pi)
        print("Pi=", Pi)
        print("est-il vrai que pi=256 ?", pi==256)
        print("e=",e)
```

```
pi= 256
Pi= 1000
est-il vrai que pi=256 ? True
pi= 3.141592653589793
Pi= 1000
est-il vrai que pi=256 ? False
e= 2.718281828459045
```

Dans ce paragraphe, nous voyons comment Pythion exécute les opérations arithmétiques élémentaires.

```
In [ ]: a=13
        b=5
        print('a+b =',a+b)
        print('ab=',a*b)
        print('3a-4b=',3*a-4*b)
        print(2**10)
        print(2**1000)
        print(a**2-b**2)
        print((a-b)*(a+b))
```

Attention à la division !! :

```
In [ ]: a=13.4
        b=5
        c=13
        print('a =',a,type(a))
```

```
print("b =",b,type(b))
print('c =',c,type(c))
print(a/b,type(a/b))
print(a//b,type(a//b))
print(c/b, type(c/b))
print(c//b, type(c//b))
print(c//a, type(c//a))
print(c/a, type(c/a))
```

```
In [ ]: a,b,c,c%a,a%c,b%a,a%b,c%b,b%c
```

Pour vérifier qu'on a compris ces différents opérateurs :

```
In [ ]: A,B,C=25,10,8.4
        print('A =', A//B, '*',B,'+',A%B)
        print('A =', A//C, '*',C,'+',A%C)
        print('C =', C//B, '*',B,'+',C%B)
```

ou encore, en utilsant les booléens et les opérateurs de comparaisons :

```
In [ ]: A1=A/B*B+A%B
        A2=A//B*B+A%B
        print("est-ce que A=A1 ?",A==A1)
        print("est-ce que A=A2 ?",A==A2)
```

Profitons de l'occasion pour regarder d'autres opérateurs de comparaison :

```
In [ ]: print(3>4)
        print(3!=4)
        print(3<4)
        print(3<=4)
        print((3+1)>4)
        print((3+1)>=4)
        print(3==4)
        print((3*5)==(4+11))
```

Pour terminer ce paragraphe, notons la différence :

```
In [ ]: print(A//B, '*',B,'+',A%B)
        print(A//B*B+A%B)
```

Si vous rentrez l'heure h, la minute m et la seconde s actuelles, vous obtenez ainsi directement

```
In [ ]: h,m,s=8,28,17
        print("Il s'est écoulé", 3600*h+60*m+s,"secondes depuis minuit")
```

# 3 Exercices

## 3.1 Exercice 1 - Conversion de secondes

a) Ecrire un programme qui convertit en années, mois, jours, heures, minutes et secondes une durée de 123456789 secondes et afficher le résultat sous la forme 123456789 secondes = $a$ années $j$ jours $h$ heures $m$ minutes $s$ secondes

b) Ecrire un programme qui convertit en millénaires, siècles, années, mois, jours, heures, minutes et secondes une durée de 123456789987654321 secondes et afficher le résultat sous la forme 123456789987654321 secondes = $M$ millénaires $S$ siècles $a$ années $j$ jours $h$ heures $m$ minutes $s$ secondes

### 3.2 Exercice 2 - Conversion de degrés en radians

Ecrire un programme qui convertit en radians un angle $A$ de 55 degrés 36 minutes et 28 secondes en affichant le résultat sous la forme : un angle de 155 ° 8 ′ 13 " vaut $x$ radians

## 4 Un mot sur les chaînes (string)

Il s'agit du type str (pour string) que l'on a déjà rencontré. Il fait partie de la famille des `sequences` (i.e., des *suites*) comme les `tuples` (déjà rencontrés) ou les `listes` (qui feront l'objet du paragraphe suivant). Une *chane* est délimitée par des *guillemets* (" ... ") ou par des *apostrophes* (' ... '). Voici un dialogue qui montre notamment comment écrire des apostrophes ou des guillemets :

```
In [ ]: off1='Le voilà qui s\'approche'
        dial1="\"Salut\", c\'est moi"
        dial2='T\'es qui, toi ?'
        print(off1)
        print(dial1)
        print(dial2)
```

Les *chanes* peuvent se concaténer :

```
In [ ]: alpha_1='abcdef'
        alpha_2='ghijklm'
        alpha_3='nop'
        alpha_4='qrstuvwxyz'
        print("alpha_1=",alpha_1,"\n     alpha_1 est une chaîne :", type(alpha_1))
        print('vous avez vu comment on peut passer à la ligne ? ; ceux qui connaissent C ne sont pas su
        print('et maintenant je vais sauter deux lignes \n\n')
        print("et voilà ! \n Bon, revenoons aux alpha_i et à la concaténation \n")
        alphabet=alpha_1+alpha_2+alpha_3+alpha_4
        print("Voilà enfin un alpahbet complet : ", alphabet)
```

Dans une *chane*, chaque caractère à une place précise, donné par son indice. ATTENTION : le premier indice est 0 (règle générale pour les `sequences`)

```
In [ ]: print(alphabet[0],alphabet[1],alphabet[6],alphabet[19],alphabet[25])
```

```
In [ ]: alphabet[26]
```

L'indice 26 n'existe pas car *alphabet* ne contient que 26 caractères (et on commence à 0). Il y a une commande simple et très utile pour connaître le nombres d'indices affectés :

```
In [ ]: len(alphabet)
```

Et on peut sélectionner les indices que l'on veut dans une *chane* :

```
In [ ]: print(off1[8:18])
        print(alphabet[3:13])
        print(alphabet[:3])
        print(alphabet[19:])
```

On peut aussi compter à partir de la droite :

```
In [ ]: print(alphabet[-2])
        print(alphabet[-26])
        print(alphabet[-5:])
        print(alphabet[:-5])
        print('alphabet est égal à alphabet[:-5] + alphabet[-5:] ? ', alphabet==alphabet[:-5]+alphabet[
```

```
In [ ]: alphabet=alphabet+'un-de-plus'
        print (alphabet[26])
        print(alphabet[25:])

In [ ]: len(alphabet)
```

Observez le résultat des deux lignes de commmandes suivantes :

```
In [ ]: a=int("7")+int("8")
        b=str(7)+str(8)
        a,b
```

Que font les instructions int et str ? (si ce n'est pas clair, vérifier quels sont les types de a et b).

# 5 L'instruction conditionnelle while

Cette instruction `while` (ou `tant que`) permet de répéter une opération tant qu'une certaine condition est vérifiée. Exemple :

```
In [ ]: n=1
        while (n<20):     # Ne pas oublier le double-point !!!
            n=2*n         # Ne pas oublier l'indentation !!!
            print(n)
```

On notera l'importance des indentations. Certains éditeurs feront d'eux-mêmes des indentations mais, même si c'est le cas, le résultat dépend de la disposition des indentations dans le texte :

```
In [ ]: n=1
        while (n<20):
            n=2*n         # n double de valeur
        print(n)
```

Et parfois, le code ne sera tout simplement pas exécutable si les indentations ne sont pas respectées :

```
In [ ]: n=1
            while (n<20):     # Ne pas oublier le double-point !!!
            n=2*n         # Ne pas oublier l'indentation !!!
            print(n)

In [ ]: n=1
        while (n<20):     # Ne pas oublier le double-point !!!
            n=2*n         # Ne pas oublier l'indentation !!!
            print(n)
```

Le calcul des puissances de 2 nous donne l'occasion de remarquer que `Python` n'est pas effrayé par les très très grands entiers... :

```
In [ ]: k=0
        while (k<10000):
            n=2**k # n double de valeur
            k=k+1
        print("2 à la puissance ", k, " est égal à\n", n)
```

# 6 Exercices

## 6.1 Exercice 3 - Alphabets

a) A partir de la chaîne `abcdefghijklmnopqrstuvwxyz` (qu'on appellera `alpha`), utiliser la boucle while pour construire une chaîne `a b c d e f g h i j k l m n o p q r s t u v w x y z` (qu'on appellera `nouvel_alpha`).

b) Recommencer le même exercice avec la chaîne `saperlipopette`.

## 6.2 Exercice 4 - Tables de multiplication

a) Utiliser la boucle while pour afficher la table de multiplication de 6 sous la forme :

1 x 6 = 6 2 x 6 = 12 3 x 6 = 18 etc...
jusqu'à 10 x 6 =60

b) Sur le même modèle, utiliser deux boucles while pour afficher les tables de multiplication de 2 à 12 ; les tables de multiplications de deux entiers successifs seront séparées par une ligne vide.

## 6.3 Exercice 5 - La plus petite puissance négative de 2

Utiliser la condition `while` pour trouver le plus petit réel strictement positif de la forme $1/2^n$ et indiquer pour quel entier $n$ il est atteint.

## 6.4 Exercice 6 - Les nombres de Fibonacci

On rappelle que la suite de Fibonacci est définie par $F(0) = F(1) = 1$ et $F(n+2) = F(n) + F(n+1)$.

a) Utiliser la boucle `while` pour calculer $F(100)$.

b) Trouver le plus grand entier $n$ tel que $F(n) < 10^{20}$ puis afficher $n$ et $F(n)$.

## 6.5 Exercice 7 - Calcul du PGCD

Pour obtenir le PGCD de a et de b, on effectue la division euclidienne $a = b * q_1 + r_1$, puis $b = r_1 * q_2 + r_2$, puis $r1 = r_2 * q_3 + r_3$, etc... Le dernier reste non nul dans cette suite de divisions est le PGCDde a et b. Utiliser l'instruction `while` pour calculer le PGCD de 573804 et 348096 à l'aide de cette méthode.

# Séance 2

## 7 Les listes

Les `listes` sont un autre type de `sequences`. Elles sont notées entre crochets et leurs éléments ne sont pas forcément du même type :

```
In [1]: L=["a","b",'c',23,"salut",-56.34,12/29,1234]
        print(L)
```

Comme pour les *chaînes*, on a différentes façons d'en connaître des éléments ou des sous-séquences :

```
In [4]: print (L[0])
        print (L[2:])
        print (L[-2])
        print (L[3:6])
```

On peut également savoir si un élément est (ou n'est pas) dans une liste :

```
In [5]: 1 in L, 0 in L, 'B' in L, "b" in L, 100 not in L
```

Il y a aussi diverses possibilités de concaténer :

```
In [6]: M=[-1,1,-1,1,0,1]
        print (L+M+L)
        print (len(L+M))
        print (L[0:24:3])
        print ((L+M)[0:24:3])
        print ((L+M+L)[0:24:3])
        print(L+['au revoir']+['s','t','o',"p"])
```

ATTENTION !! La syntaxe peut changer sur certains points, suivant qu'on utilise une version 2.x ou 3.x.; dans l'exemple précédent, la commande `print "le nombre d'éléments de L est", len(L)` ne fonctionne que sur les versions 2.x. Avec les versions3.x, c'est la syntaxe suivante qui donnera le même résultat (et qui fonctionnera aussi avec la version 2) :

```
In [7]: print("le nombre d'éléments de L est", len(L))
```

L'addition $L + L + ... + L$ (où $L$ apparaît $n$ fois) se donne aussi $n * L$ ou $L * n$ :

```
In [10]: L+L+L==3*L, L*4==4*L
```

```
In [7]: print (3*L)
```

Dans une liste, on peut modifier la valeur d'une entrée :

```
In [8]: print ("L[2]=",L[2])
        L[2]="je suis nouveau"
        print ("L[2]=",L[2])
        print (L)
        L[4]=M
        print (L)
```

Il est parfois utile de créer une liste vide :

```
In [11]: liste=[]
         print("liste=",liste)
```

On peut aussi tester si un élément figure dans une liste, l'insérer en fin de liste ou à un endroit fixé ou encore suppprimer l'élément correspondant à un indice fixé :

```
In [12]: A=[1,2,3,4,5,16,7]
         print ("A=",A)
         A.append(1.11)
         print ("A=",A)
         A.insert(5,-2)
         print ("A=",A)
         A.pop(6)
         print ("A=",A)
         A.remove(2)        ### cette fonction ne marchera pas si 2 n'est pas dans A...
         print ("A=",A)
```

On note la syntaxe du type `x.fonction(...)` qui signifie qu'on applique la fonction (avec les arguments donnés entre parenthèses) à l'objet `x`. Il existe aussi des fonctions prédéfinies pour les listes :

```
In [13]: print (max(A), "est le plus grand nombre de la liste A")
         print (min(A), "est le plus petit nombre de la liste A")
         A = sorted(A) # la même liste en réordonnant les éléments
         print(A)
```

Et des `listes` prédéfinies qui permettent de définir des suites très rapidement :

```
In [9]: A=range(10)
        print(len(A))
        print(A[0],A[1],A[2],A[3],A[4],A[5],A[9])
        print(A[4])
        B=A[0:8:3]
        print(len(B))
        print(B[0],B[1],B[2])
```

# 8 La boucle for

La commande `for` permet d'itérer une instruction ; par exemple, on peut l'utiliser avec toute liste de nombres :

```
In [26]: for i in A:
             print (i**2)
```

ou comme indice classique, éventuellement avec une incrémentation différente de 1 :

```
In [33]: for i in range(10):
             print(i**3)
         print ("\n")

         for i in range(1,10,2):
             print (i**3),      ### A quoi sert la virgule ??
         print ("\n")
```

On peut ainsi se donner très rapidement les termes d'une suite (définie par une formule explicite) :

```
In [34]: print([i**3 for i in range(10)])
         print ([n+n**2 for n in range(1,10)])
```

On peut imbriquer des boucles :

```
In [35]: L=[]                          #### je vais ranger toutes mes valeurs dans une liste L
         for i in range (5):
             for j in range (5):
                 L.append((i+j)**2)
         print (L)
```

# 9 Définir de nouvelles fonctions

Il est temps de définir nos propres *fonctions* ; par exemple, la *fonction* $f(x) := x^5 + 12x^3 - 6x + 7$ :

```
In [36]: def f(x):              # f est le nom de notre fonction
                                # elle s'applique à une variable (qu'on appelle x)
             return(x**5+12*x**3-6*x+7)  # ce que renverra f(x)
```

A présent, je peux chercher les valeurs des images que je veux :

```
In [37]: print(f(1),f(-2),f(0),f(4))
```

Attention, l'instruction `return` est importante ; c'est elle qui nous dit ce que vaut $f(x)$. Par exemple, la fonction $g$ suivante semble faire la même chose que $f$ :

```
In [38]: def g(x):
             print(x**5+12*x**3-6*x+7)
         g(1),g(-1),g(0),g(4)
```

Mais c'est ne pas du tout le cas comme le petit calcul suivant peut nous en convaincre :

```
In [39]: print(f(1)+f(-2))
```

```
In [40]: g(1)+g(-1)
```

Voici l'exemple de la fonction qu'on va appeler `facto` qui calcule la *factorielle* en utisant l'instruction conditionnelle `while` :

```
In [11]: def facto(x) :    # je donne un nom à la fonction
             f=1           # une variable intermédiaire que je vais utiliser
                           # pour faire le calcul
             while(x>0) :
                 f=f*x
                 # je calcule x!... C'est clair ??
                 x=x-1
             return f      # ma fonction va renvoyer x !
```

Et maintenant, je peux utiliser ma fonction `facto` :

```
In [13]: print ("100!=",facto(100))
         for i in range(8):
             print (i,"!=",facto(i))
```

Il peut bien entendu exister plusieurs façons de calculer une même fonction :

```
In [43]: def facto2(x):              ##### méthode itérative
             a=1
             for i in range(2,x+1):
                 a *= i               ### noter le raccourci de a=a*i
             return a

In [44]: facto2(6)
```

Une fonction peut avoir plusieurs arguments ; par exemple, si on veut une fonction qui calcule le PGCD (cf. Exercice 7) :

```
In [45]: def PGCD(a,b):
             A,B=a,b
             while B!=0:
                 A,B=B,A%B
             return(A)

In [46]: print(PGCD(28,45))
         print(PGCD(48,45))
         print(PGCD(3**7*5**2*11*4,3*22*5**4*13))
```

## 10 L'instruction if

L'instruction conditionnelle `if` permet de s'assurer qu'une condition est vérifiée. Par exemple, la fonction `facto2` que l'on vient de voir nécessite des entiers (est-ce aussi la cas de `facto` ?) :

```
In [47]: facto2(3.5)
```

On peut donc intégrer une vérification que l'argument rentré est bien un entier :

```
In [49]: def facto2(x):             ##### méthode ITERATIVE
             if type(x) != int:                 ## si la condition est vérifiée, on fait...
                 return("hé, il faut donner un entier !!")
             else:                      ## si la condition n'est pas vérifiée, on fait...
                 a=1
                 for i in range(2,x+1):
                     a *= i               ### noter le raccourci de a=a*i
                 return a

In [50]: facto2(3.5)
```

L'instruction `if` est très utile pour définir les fonctions ; voici une nouvelle version du calacul de la fonction *factorielle* :

```
In [51]: def facto3(x):         ###### méthode RECURSIVE
             if x==0:
                 return(1)
             else :
                 return(x*facto3(x-1))    ##### Une fonction récursive est une fonction qui fait
                                          ##### appel à elle-même

In [52]: facto3(8)
```

Dans l'exemple suivant, on va calculer le nombre de solutions du trinôme $ax^2 + bx + c$ avec $a \neq 0$ et $a, b, c$ réels :

```
In [53]: def nb_de_racines(a,b,c):
             if a==0:
                 print("ce n'est pas une équation du second degré")
             else :
                 delta=b**2-4*c
                 if delta > 0:
                     print("L'équation a deux solutions réelles")
                 else :
                     if delta == 0:
                         print("L'équation a une unique solution réelle (racine double)")
                     else :
                         print("L'équation n'a pas de solution réelle")

In [54]: nb_de_racines(1,0,-1)
         nb_de_racines(1,0,1)
```

Dans l'exemple précédent, on peut trouver déplaisante la cascade de `if`. L'instruction `elif` est faite pour éviter ce genre de cascade :

```
In [55]: def nb_de_rac(a,b,c):
             delta=b**2-4*c
             if a==0:
                 print("ce n'est pas une équation du second degré")
             elif delta >0 :
                 print("L'équation a deux solutions réelles")
             elif delta == 0 :
                 print("L'équation a une unique solution réelle (racine double)")
             elif delta < 0 :
                 print("L'équation n'a pas de solution réelle")

In [56]: nb_de_rac(1,0,1)
         nb_de_rac(1,-2,1)
```

# 11 Exercices

## 11.1 Exercice 8 : années bissextiles

Ecrire une fonction booléenne `bissectile(n)` qui renvoie 1 si $n$ est une année bissectile et 0 sinon. On rappelle que les années bissextiles reviennent tous les 4 ans, sauf lorsqu'elles sont un multiple de cent qui n'est pas un multiple de 400.

## 11.2 Exercice 9 : le crible d'Eratosthène

Ecrire une liste qui contient les nombres premiers inférieurs à 100, obtenus en appliquant la méthode du crible d'Eratosthène.

## 11.3 Exercice 10 : somme de carrés

Ecrire une fonction qui, pour tout $M$ réel postif, retourne le couple $(n, m)$ où $n$ est le plus petit entier tel que $m = 1 + 2^2 + 3^2 + ... + n^2 >= M$.

## 11.4 Exercice 11 : présent ou pas

Ecrire une fonction `present(x,L)` qui retourne `True` si $x$ est dans $L$ et `False` sinon (et qui vérifie que le second argument est bien une liste)

## 11.5 Exercice 12 : présence d'une lettre dans un mot

Ecrire une fonction `presente(a,M)` qui retourne l'indice de la lettre $a$ dans la chaîne $M$ si $a$ est dans $M$, qui renvoie -1 sinon et qui vérifie que le second argument est bien une chaîne de caractères.

## 11.6 Exercice 13 : écriture en base 16

Ecrire une fonction `H(n)` qui renvoie l'écriture en base 16 de l'entier n (les chiffres retenus pour l'écriture en base 16 sont $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$).

## 11.7 Exercice 14 : Test de primalité

Ecrire une fonction `est_premier(N)` qui retourne `True` si l'entier $N >= 2$ est premier et `False` sinon en testant si $N$ admet un diviseur parmi les entiers inférieurs ou égaux à sa racine carrée. On utilisera la fonction `sqrt` (qu'il faudra peut-être importer, par exempel en tapant `from math import *`)

## 11.8 Exercice 15 : décomposition en produit de facteurs premiers

Ecrire une fonction `decomposition_en_facteurs_premiers(N)` qui retourne la décomposition en facteurs premiers de l'entier `N>=2` (nota bene : le résultat est attendu sous la forme d'une chaîne de caractères).