Sous-programmes
Corrigé

Objectifs
- Savoir écrire un sous-programme
- Comprendre les paramètres en Python
- Savoir écrire des sous-programmes itératifs et récursifs

1 Vocabulaire

Exercice 1 : Vocabulaire
On considère la fonction suivante.

```
1  def a(b, c):
2      '''
3      Bla bla bla...
4      '''
5      d = b + c
6      return d
7
8  e = 7
9  a(1, e)
```

Indiquer (on donnera les numéros de ligne et/ou les éléments concernés) ce qui est :

1. signature d'une fonction
   Solution : Ligne 1 (ligne de `def`).
2. spécification (ou interface) d'une fonction
   Solution : Lignes 1 à 4. C'est la signature (ligne 1) + la sémantique (la docstring des lignes 2 à 4).
3. implantation (ou corps) d'une fonction
   Solution : Lignes 5 à 6.
4. paramètre formel
   Solution : b ou c (ligne 1).
5. paramètre effectif
   Solution : 1 ou e (ligne 9).

6. variable locale
   Solution : d (ligne 5).
7. variable globale
   Solution : e (ligne 8).

2 Une première fonction

Exercice 2 : Permuter deux éléments d'une liste
On souhaite disposer d'un sous-programme qui permute deux éléments d'une liste, ces deux éléments étant identifiés par leurs indices. On utilisera les fichiers suivants : `permuter.py` et `test_permuter.py`.

2.1. Donner deux exemples d'utilisation de ce sous-programmes en indiquant les données manipulées par ce sous-programme.

Solution :

1. Si `liste = [5, 1, 6, 2, 9]; i1 = 0; i2 = 2;` alors on aura : `liste == [6, 1, 5, 2, 9]`
2. Si `liste = [5, 1, 6, 2, 9]; i1 = 1; i2 = -1;` alors on aura : `liste == [5, 9, 6, 2, 1]`
   Ce test montre que l'on peut utiliser des indices négatifs.
3. On pourrait prendre un autre exemple avec deux indices égaux.

2.2. Écrire la spécification (l'interface) de ce sous-programme.

Solution :

```
1  def permuter(liste, i1, i2):
2      '''
3      Permuter les éléments aux indices i1 et i2 de la liste.
4
5      :pre: isinstance(liste, list)
6      :pre: -len(liste) <= i1 < len(liste)
7      :pre: -len(liste) <= i2 < len(liste)
8      '''
```

2.3. Écrire un programme de test (avec `pytest`) de ce sous-programme. Exécuter le programme de test qui devrait révéler des erreurs puisque le sous-programme précédent n'a pas été implanté.

Solution :

```
1  import pytest
2  from permuter import permuter
3
4  @pytest.fixture
5  def l1():
6      return [5, 1, 6, 2, 9]
7
8  def test_indices_positifs(l1):
```

```
9       permuter(l1, 0, 2)
10      assert l1 == [6, 1, 5, 2, 9]
11
12  def test_indices_negatifs(l1):
13      permuter(l1, 1, -1)
14      assert l1 == [5, 9, 6, 2, 1]
15
16  def test_indices_identiques(l1):
17      permuter(l1, 3, 3)
18      assert l1 == [5, 1, 6, 2, 9]
```

Quelques remarques :

— Le sous-programme `permuter` ne peut pas être directement utilisé avec **`assert`** car il renvoie toujours **`None`** (procédure). Il faut donc initialiser une liste, permuter deux de ses éléments et, enfin, vérifier que la liste a été correctement modifiée.

— On écrit plusieurs fonctions de tests, chaque fonction correspondant à un cas de test. Ici les cas de test identifiés sont 1) indice positif, 2) indice négatif et 3) indices identiques. Le nom de la fonction de test reflète le cas de test. Notons que pour un cas de test donné, plusieurs données de tests pourraient être choisies donnant à lieur à plusieurs **`assert`** dans la même fonction de test (ou à plusieurs fonction de test avec un suffixe qui identifie les données de test).

— On aurait pu initialiser `l1` avec `[5, 1, 6, 2, 9]` en début de chaque fonction de test. Définir une fonction `l1` sans paramètre, préfixée de `@pytest.fixture` (un décorateur), indique que c'est une contexte de test. Une fonction de test qui a un paramètre nommé `l1` sera automatiquement initialisé par `pytest` avec le résultat de la fonction qui porte le même nom si elle a été déclarée comme contexte de test.
On peut ainsi factoriser la création des objets utilisés dans les tests. Ceci est particulièrement utile si la création de l'objet nécessite plusieurs instructions.

2.4. Écrire l'implantation (le corps) de ce sous-programme.

Solution :

```
1   def permuter(liste, i1, i2):
2       '''
3       Permuter les éléments aux indices i1 et i2 de la liste.
4
5       :pre: isinstance(liste, list)
6       :pre: -len(liste) <= i1 < len(liste)
7       :pre: -len(liste) <= i2 < len(liste)
8       '''
9       liste[i1], liste[i2] = liste[i2], liste[i1]
```

Remarque : Le cas de test, indices identiques, serait utile si on avait réaliser la permutation sans utiliser l'affectation multiple de Python, ni une variable supplémentaire :

```
1       liste[i1] = liste[i1] + liste[i2]
2       liste[i2] = liste[i1] - liste[i2]
3       liste[i1] = liste[i1] - liste[i2]
```

3 Spécification de sous-programmes

Exercice 3 : Spécifier des sous-programmes
Pour chacun des énoncés suivants, donner la spécification du sous-programme correspondant. On écrira les spécifications dans le fichier `signatures.py`.

1. Calculer la puissance entière d'un réel

   Solution :

   (a) **Objectif :** La puissance entière d'un réel

   (b) **Exemples :**
   — nominal puissance positive : $2^4 = 16$
   — nominal puissance nulle $5^0 = 1$
   — nominal puissance négative $2^{-3} = 0.125$
   — nominal puissance négative, nombre négatif $(-2)^{-3} = -0.125$
   — nominal avec un vrai nombre réel $(-1.2)^2 = 1.44$
   — nominal nombre négatif, puissance positive $(-3)^3 = -27$
   Quand on envisage un exposant négatif, on doit se rendre compte que si le nombre est nul, la puissance ne peut pas être calculée (division par zéro). Nous avons donc une fonction partielle.
   Ceci permet de bien identifier les informations fournies au sous-programme : le nombre et l'exposant et les informations attendues la puissance.

   (c) **Paramètres :** (on identifie d'abord le rôle, le mode, le type puis le nom)
   — nombre : in Réel – – nombre réel
   — exposant : in Entier – – l'exposant
   — puissance : out Réel – – la puissance (nombre ** exposant)

   (d) **Type de sous-programme :** Fonction car un seul paramètre en sortie et les autres en entrée.
   Remarque : le nom du paramètre en sortie est un bon candidat pour le nom de la fonction !

   (e) **Préconditions :** Non (x = 0 Et n <= 0) <=> x /= 0 Ou n > 0

   (f) **Postcondition :** puissance == nombre ** exposant = Produit(nombre) n fois si n > 0...

```
-- La puissance entière d'un nombre réel
-- Paramètres
--      - nombre   : in  Réel   -- le nombre réel
--      - exposant : in  Entier -- l'exposant
-- Retourne
--      - puissance : out Réel  -- nombre à la puissance exposant
--
-- Nécessite  : x /= 0 Ou n > 0
-- Assure : pas si simple à exprimer !
-- Exemples :
--      On peut reprendre quelques uns de ceux qui sont donnés avant
Fonction Puissance (Nombre : in Réel; exposant : in Entier) retourne Réel
```

```python
def puissance (nombre: float, exposant: int) -> float:
    '''
    Retourner la puissance entière d'un réel.

    Paramètres :
        nombre: in Float     -- le nombre réel
        exposant: in int     -- l'exposant
    Résultat : Réel  -- nombre à la puissance exposant
    Nécessite :
        (exposant >= 0) or (nombre != 0)               -- pas de division par zéro
    Assure
        Résultat == nombre ** exposant (à epsilon près)
    '''
    pass

"""
Remarque : Pour trouver la précondition, on peut d'abord identifier les cas
où on ne pourra pas calculer la puissance.  On ne peut pas calculer la
puissance si :

  exposant < 0 and nombre == 0

La précondition est la négation de cette condition :

  not (exposant < 0 et nombre == 0)
  <=>  not (exposant < 0) or not (nombre == 0)  -- de Morgan
  <=>  exposant >= 0 or nombre != 0
"""
```

2. Connaître le nombre de jours d'un mois.

   Solution : On pourrait dans un premier temps penser que l'on a deux données :

   (a) le mois (in)

   (b) le nombre de jours du mois (out)

   Mais si on prend des exemples, on va certainement considérer le mois de février qui peut avoir 28 ou 29 jours suivant que l'année est bisseextile ou non. On en déduit qu'il y a une donnée supplémentaire à considérer, l'année (in).

```python
def nb_jours_mois(mois: int, annee: int) -> int:
    '''
    Retourner le nombre de jours d'un mois pour une année donnée.

    Paramètres
        mois : in int      -- le mois considéré
        annee: in int      -- l'année considérée
    Retourne
        le nombre de jours de ce mois pour cette année
    Nécessite
        1 <= mois <= 12
        annee >= 0
```

```
13      '''
14      pass
```

3. Saisir un entier au clavier.

   Solution :

   (a) **Objectif :** L'énoncé ci-dessus.

   (b) **Exemples :**
   — l'utilisateur tape sur les touches 1 2 5 3 «Entrée», l'entier est 1253.
   — ...
   —

   (c) **Paramètres :**
   — entier_lu : out Entier – l'entier saisi par l'utilisateur

   (d) **Précondition :** —

   (e) **Postcondition :** —

```
1  -- Saisir une entier.
2  --
3  -- Paramètre :
4  --      nombre : out Entier     -- l'entier saisi par l'utilisateur
5  --
6  -- Nécessite : ---
7  -- Assure :  ----
8  -- Exemples : 1 2 5 3 «Entrée» -> Nombre = 1253
9  Procédure Saisir(Nombre : out Entier);
```

```
1  def entier_lu() -> int:
2      '''
3      Retourner un entier lu au clavier.
4
5      Retour :
6         nombre: int   -- l'entier saisi
7
8      Nécessite : ---
9      Assure : -- le résultat est un entier
10     '''
11     pass
```

Remarque : On aurait pu imaginer un paramètre supplémentaire (en in) correspondant à la consigne affichée à l'utilisateur quand il est sollicité.

4. Obtenir le quotient et le reste d'une division entière

   Solution :

```
1  -- Le quotient et le reste d'une division entière
2  --
3  -- Paramètres :
4  --      Dividende: in Entier
5  --      Diviseur : in Entier
6  --      Quotient : out Entier
7  --      Reste : out Réel
8  --
```

```
-- Nécessite :
--      Diviseur >= 0
--      Diviseur > 0
--
-- Assure :
--      0 <= Reste
--      Reste < Diviseur
--      Dividende = Quotient * Diviseur + Reste
--
-- Exemples :
--      Nominal (reste non nul) : Dividende = 11, Diviseur = 4 -> Quotient = 2 et Reste = 3
--      Nominal (reste nul) : Dividende = 12, Diviseur = 4 -> Quotient = 3 et Reste = 0
Procédure Div_Mod (Dividende, Diviseur : in Entier ;
                   Quotient, Reste : out Entier)
```

```
def div_mod(dividende: int, diviseur: int) -> (int, int):
    '''
    Calculer le quotient et le reste de la division entière d'un dividende
    par un diviseur.

    Paramètres
        dividende: in int
        diviseur: in int
    Résultats
        quotient: int
        reste: int

    Nécessite :
        dividende >= 0
        diviseur > 0
    Assure
        dividende == quotient * diviseur + Reste
        0 <= reste and reste < diviseur
    '''
    pass
```

5. Savoir si une année est bissextile

   Solution :

```
def est_bissextile(annee: int) -> bool:
    '''
    Est-ce qu'une année est bissextile ?

    Paramètres
       annee : in int -- l'année à analyser
    Résultat : bool   -- Vrai ssi annee est bissextile

    Nécessite : ---
    Assure :
         Résultat == (annee Mod 4 == 0)
            and ((annee Mod 100 != 0) or (annee Mod 400 == 0))
    '''
    pass
```

6. Saisir un entier compris entre une borne inférieure et une borne supérieure. Avant chaque demande à l'utilisateur, une consigne lui est affichée pour lui expliquer ce qui est attendu.
   Solution :
   (a) **Objectif :** L'énoncé ci-dessus.
   (b) **Exemples :**
   — nominal un seul essai inf = 10, sup = 15, utilisateur : 10. Nombre = 10.
   — nominal trop petit : inf = 10, sup = 15, utilisateur 9, 16, 13. Nombre = 13
   — limite : borne inférieure choisie : inf = 10, sup = 15, utilisateur 10. Nombre = 10
   — limite : borne supérieure choisie : inf = 10, sup = 15, utilisateur 15. Nombre = 15
   —
   On pourrait ajouter la consigne et montrer précisément ce qui doit se passer. Ici l'idée est plutôt de spécifier l'interface du programme avec son utilisateur, en particulier en précisant les messages.
   (c) **Paramètres :**
   — inf : in Entier – borne inférieure
   — sup : in Entier – borne supérieure
   — consigne : in Chaine – le message à afficher à l'utilisateur
   — entier_lu : out Entier – l'entier saisi par l'utilisateur
   (d) **Précondition :** inf < sup
   (e) **Question :** faut-il prévoir une contrainte sur le message ?
   (f) **Postcondition :** inf <= nombre Et nombre <= sup

```
-- Saisir une entier compris entre une borne inférieure et une borne
-- supérieure.  Un message d'afficher la consigne à l'utilisateur.
--
-- Paramètre :
--      Inf, Sup: in Entier  -- borne dans lesquelles doit être l'entier saisi
--      Consigne: in Chaine  -- le message à afficher à l'utilisateur
--
-- Nécessite : Inf <= Sup
-- Assure :  Inf <= Nombre Et Nombre <= Sup
-- Exemple : Saisir
Procédure Saisir(Nombre : out Entier ; Inf, Sup: in Entier ; Consigne: in Chaine);
```

```
def entier_lu (inf: int, sup: int, consigne: str) -> int:
    '''
    Saisir un entier au clavier, entier qui doit être compris entre inf et
    sup inclus.  Une consigne est affiché à l'utilisateur avant la saisie.
    En cas de saisie incorrecte, une explication est affiché à
    l'utilisateur et une nouvelle saisie a lieu.

    Paramètres
      inf, sup: in int      -- Bornes entre lesquelles doit se trouver l'entier saisi
      consigne: in str      -- Le message à afficher à l'utilisateur
    Résultat:  int       -- L'entier saisi

    Nécessite
```

```
        inf < sup  -- au moins deux éléments possibles
    Assure
        inf <= nombre <= sup
    '''
    pass
```

7. Classer une liste d'élèves dans l'ordre décroissant de leur moyenne.

   Solution :

```
def classer_eleves(eleves):
    '''
    Classer les élèves par ordre croissant de leur moyenne.

    Paramètres :
        - élèves : in out liste d'élèves (un élève pourrait être une liste, un
        couple...)

    Retour :
        - aucun (car la liste des élève est modifiée)

    Précondition : --
    Postcondition : les élèves sont classés suivant l'ordre croissant de leur moyenne
    '''
```

4 Signature d'un sous-programme

Exercice 4 Définissons la signature de différentes fonctions.

4.1. Donner une signature possible pour la fonction `f` sachant que les appels suivants sont valides :

```
f(1)
f(2, 3)
f(m = 5, c = 6)
f(7, m = 8)
```

et que les appels suivants sont refusés :

```
f()
f(9, 10, 11)
```

On complètera le fichier `test_f.py`.

Solution : Une signature qui convient est :

```
f(c, m = XXX)
```

avec `XXX` une valeur que les exemples donnés ne permettent pas de définir.

Notons, que cette signature n'est pas unique. Une autre signature qui conviendrait est :

```
f(c, m = XXX, *, y = YYY, z = ZZZ)
```

4.2. Donner la signature et le code d'une fonction `produit` qui permet de faire le produit d'un nombre quelconque de paramètres. Voici quelques exemples d'utilisation :

```
assert produit(5) == 5
assert produit(2, 5) == 10
assert produit() == 1
assert produit(1, 2, 3, 4, 5, 6) == 720
```

On complètera le fichier `produit.py` et on le testera avec `test_produit.py`.

4.3. Définir une signature pour la fonction g sachant que les appels suivants sont possibles :

```
g(a=1, b=1)
g(2, a=2)
g(a=3)
```

et que les appels suivants sont interdits :

```
g(10, 10)
g()
```

On complètera le fichier `test_g.py`.

Solution : Une signature possible est :

```
def g(b=XXX, *, a):
    pass
```

4.4. Définir la signature de `sp` pour que les tests de `test_sp_signature.py` passent.

5 Comprendre la récursivité

Exercice 5 : Comprendre la récursivité

Une fonction récursive est une fonction dont l'implantation contient un appel à elle-même. Un exemple classique est la factorielle définie en mathématiques de la manière suivante :

$$n! = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

On peut en déduire le code suivant.

```
def fact(n):
    ''' Factorielle d'un entier n positif...'''
    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)

if __name__ == "__main__":
    print('4! =', fact(4))
```

5.1. Indiquer les différents appels qui ont lieu quand on demande à calculer `fact(4)`.
Solution :

```
fact(4) --> 4 * fact(3) car 4 <= 1 faux
                  \--> 3 * fact(2) car 3 <= 1 faux
                             \--> 2 * fact(1) car 2 <= 1 faux
                                        \--> 1 car 1 <= 1 vrai
                                        <-- 1
                             <-- 2 * 1 = 2
                  <-- 3 * 2 = 6
         <-- 4 * 6 = 24
```

5.2. Exécuter le programme précédent sous Python tutor :

`http://www.pythontutor.com/visualize.html#mode=edit`.

Solution : On constate qu'il y a plusieurs appels à fact avec à chaque fois le paramètre formel m lié à un nouvel entier (le précédent - 1).
5.3. Est-ce que Python peut calculer `fact(1000)` ?
Pour exécuter ce programme, taper, depuis le dossier qui contient le fichier `fact.py` :

```
python fact.py
```

Solution : Non. On a l'exception RecursionError.
Voir `sys.getrecursionlimit()` et `sys.setrecusionlimit()` pour connaître ou modifier cette limite (qui correspond au nombre d'appels de fonction dans la pile d'exécution).
5.4. Rappeler ce qu'il est conseillé de faire lorsque l'on définit une fonction récursive, en particulier, pour garantir sa terminaison ?
Solution :

1. On définit un ou plusieurs cas de base pour lesquels aucun appel récursif n'est fait.
2. On définit un ou plusieurs cas généraux où un ou plusieurs appels récursifs sont faits. Pour montrer la terminaison, on peut exhiber une taille du problème (entier positif) et montrer que les appels récursifs se font sur une taille strictement inférieure.

6 Itératif et récursif : la fonction puissance

L'objectif de ces exercices est de proposer plusieurs implantations de la fonction puissance.

Exercice 6 : Puissance entière avec exposant positif
Intéressons nous d'abord à la puissance entière d'un nombre quand l'exposant est positif. Par exemple, si le nombre est 4 et l'exposant est 3, la puissance est 64 (4 * 4 * 4). On utilisera la convention généralement admise que 0^0 vaut 1.
6.1. Écrire le code de la fonction `puissance_positive_iterative` dans le module `puissance.py` qui calcule la puissance entière d'un nombre avec comme précondition que l'exposant est positif. On écrira le code de **manière itérative**, donc en utilisant une répétition.
Solution :

```python
def puissance_positive_iterative(nombre, exposant):
    '''
    Retourne nombre à la puissance exposant avec comme contrainte exposant
    positif.

    :param nombre: nombre à élever à la puissance (in)
    :type nombre: number (int ou float)
    :param exposant: l'exposant positif
    :type exposant: int
    :pre: exposant >= 0
    '''
    assert exposant >= 0
    produit = 1
    for i in range(exposant):
        produit = produit * nombre
    return produit
```

6.2. Tester avec le fichier `test_puissance_positive_iterative.py`.

Exercice 7 : Exposants négatifs

Prennons maintenant en compte le cas général où l'exposant peut être négatif.

7.1. Caractériser le domaine de définition de la fonction puissance.

Solution : La puissance n'est pas définie dans le cas où son exposant est strictement négatif et le nombre est nul. On a alors une division par 0. Par exemple, 0 puissance -2.

7.2. Écrire le code de la fonction `puissance_iterative`. On précisera ses préconditions et on les instrumentera en utilisant **`assert`**. On utilisera la fonction précédente (exercice 6).

Solution : Pour écrire cette fonction, on peut s'appuyer sur la première fonction écrite. En particulier, on peut remarquer que $x^n = 1/x^{-n} = (1/x)^{-n}$. Nous utilisons cette dernière formulation.

```python
def puissance_iterative(nombre, exposant):
    '''
    Retourne nombre à la puissance exposant.

    :param nombre: nombre à élever à la puissance (in)
    :type nombre: number (int ou float)
    :param exposant: l'exposant positif ou négatif
    :type exposant: int
    :pre: nombre != 0 or exposant >= 0   # division par zéro
    '''
    assert nombre != 0 or exposant >= 0   # sinon division par zéro

    if exposant >= 0:
        return puissance_positive_iterative(nombre, exposant)
    else:
        return puissance_positive_iterative(1 / nombre, - exposant)
```

Remarque : Il aurait été plus logique de lever l'exception ZeroDivisionError qui est levée par Python en cas de division par zéro.

7.3. Tester avec le programme de test `test_puissance_iterative.py`.

Exercice 8 : Puissance récursive

On se propose maintenant d'écrire une version récursive de la puissance.

8.1. Écrire le code de la fonction `puissance_recursive`.

Solution :

```python
def puissance_recursive(nombre, exposant):
    """
    Même spécification que puissance_iterative
    """
    assert nombre != 0 or exposant >= 0   # sinon division par zéro

    if exposant == 0:      # cas d'arrêt
        return 1
    elif exposant < 0:
        return puissance_positive_iterative(1 / nombre, - exposant)
    else:
        return nombre * puissance_recursive(nombre, exposant - 1)
```

8.2. La tester en utilisant le programme `test_puissance_recursive.py`.

Exercice 9 : Amélioration de la puissance

On peut améliorer le calcul de la puissance en remarquant que :

$$x^n = \begin{cases} (x^2)^p & \text{si } n = 2p \\ (x^2)^p \times x & \text{si } n = 2p+1 \end{cases}$$

Ainsi, pour calculer 3^5, on peut faire $3 * 9 * 9$ avec bien sûr $9 = 3^2$.

9.1. *Version récursive.* Écrire une version récursive (`puissance_recursive_mieux` dans `puissance.py`) et la tester (`test_puissance_recursive_mieux.py`).

Solution :

```python
def puissance_recursive_mieux(nombre, exposant):
    """
    Même spécification que puissance_recursive
    """
    if exposant == 0:      # cas d'arrêt
        return 1
    elif exposant < 0:
        if nombre == 0:
            raise ValueError()
        else:
            return puissance_recursive_mieux(1 / nombre, - exposant)
    elif exposant % 2 == 0:
        return puissance_recursive_mieux(nombre * nombre, exposant // 2)
    else:
        return nombre * puissance_recursive_mieux(nombre * nombre, exposant // 2)
```

9.2. *Version itérative.* Écrire une version itérative de la puissance (`puissance_positive_iterative_mieux` dans `puissance.py`) exploitant la remarque ci-dessus et la tester (`test_puissance_iterative_mieux.py`).

Solution :

```python
def puissance_positive_iterative_mieux(nombre, exposant):
    """
    Même spécification que puissance_positive_iterative
    """
    assert exposant >= 0

    produit = 1
    while exposant > 0:
        if exposant % 2 == 1:
            produit = produit * nombre
        nombre = nombre * nombre
        exposant = exposant // 2
    return produit
```

7 Récursivité

Exercice 10 : Les tours de Hanoï

C'est Lucas, mathématicien français, qui sous le pseudonyme de Claus a inventé ce jeu.

Il se présente sous la forme d'un support en bois sur lequel sont plantés trois tiges A, B et C. Sur ces trois tiges peuvent être enfilés des disques de diamètres différents (8 dans la version originale, mais N de manière générale). Dans la configuration initiale (figure 1), les disques sont empilés par ordre de taille décroissante sur la tige A.

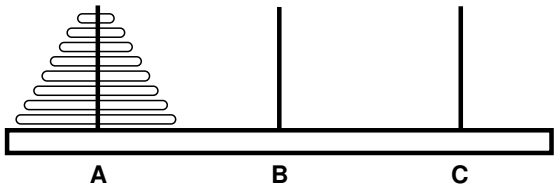

FIGURE 1 – Configuration initiale du jeu des tours de Hanoï

Le but est de déplacer tous les disques de la tige A vers la tige C sachant qu'on ne peut déplacer qu'un seul disque à la fois et qu'on ne peut pas le poser sur un disque plus petit que lui.

Ainsi, dans la configuration initiale, les deux seuls déplacements possibles sont `A -> B` et `A -> C`. On remarquera qu'un déplacement est complètement déterminé par la donnée de la tige de départ et de la tige d'arrivée, car on ne peut déplacer qu'un disque à la fois : celui qui se trouve en haut de la tige d'origine et qui sera placé au sommet de la tige destination.

Écrire un programme (fichier `hanoi.py`) qui donne la solution de ce jeu (les déplacements à effectuer). On commencera par appliquer un raisonnement par récurrence sur le nombre de disques N à déplacer :

1. Donner la solution pour $N = 0$?
2. Donner la solution pour $N = 1$?
3. Donner la solution pour $N = 2$?

4. Supposons que l'on sait résoudre un problème de Hanoï pour $N-1$ disques ($N > 1$). Montrer que l'on sait résoudre un problème de Hanoï de taille N.

On en déduira alors :

1. la spécification du sous-programme qui modélise le problème des tours de Hanoï,
2. l'implantation de ce sous-programme.

Solution : On peut raisonner par récurrence sur le nombre de disques n.

Si n = 0, on ne fait aucun déplacement.

Si n = 1, on déplace le disque de A en C.

Si n = 2, on déplace le petit disque de A en B, le grand disque de A en C et enfin le petit disque de B en C.

On suppose que l'on sait déplacer $n-1$ disques d'une tige A vers une tige C en utilisant une tige B. Pour déplacer n disques de A vers C en utilisant B, on peut faire :

1. déplacer $n-1$ disques de A vers B en utilisant C. On peut penser que c'est tricher puisque la règle impose de déplacer un seul disque à la fois et non $n-1$. Mais l'hypothèse de récurrence nous dit que l'on sait déplacer les $n-1$ disques **en respectant la règle**, donc un à la fois ;
2. déplacer le plus grand disque de A vers C ;
3. déplacer les $n-1$ disques de B vers C en utilisant A. On utilise à nouveau notre hypothèse.

Notons que quand on déplace les $n-1$ disques, nous sommes bien dans les hypothèses du jeu de Hanoï puisque les tiges ne contiennent que le grand disque qui est plus grand que tous les autres et ne gênera donc pas les déplacements.

On sait résoudre Hanoï avec 0 disque (ou 1 disque). On a montré que l'on sait résoudre Hanoï avec n disques en prenant comme hypothèse que l'on sait résoudre Hanoï avec $n-1$ disques. En conclusion, on a montré par récurrence qu'on sait résoudre Hanoï quelque soit le nombre de disques.

Une fois cette récurrence établie, la résolution du jeu se programme simplement en utilisant la récursivité.

```
'''Les tours de Hanoï.'''


def resoudre_hanoi(nb_disques, tige_depart, tige_arrivee, tige_intermediaire):
    '''
    Résoudre le jeu de Hanoï à nb_disques.  Les disques sont sur la tige de
    départ.  Ils doivent être amenés sur la tige d'arrivée en utilisant la tige
    intermédiaire.  Sur la tige de intermédiaire et la tige d'arrivée, il n'y a
    pas de disque plus petits que les nb_disques en haut de la tige de départ.
    '''
    if nb_disques <= 0:
        pass    # rien à faire
    else:
        resoudre_hanoi(nb_disques - 1, tige_depart, tige_intermediaire, tige_arrivee)
        print(tige_depart, '->', tige_arrivee)
```

```
16          resoudre_hanoi(nb_disques - 1, tige_intermediaire, tige_arrivee, tige_depart)
17
18  if __name__ == '__main__':
19      # Déterminer le nombre de disques    nb : out
20      import sys
21      if len(sys.argv) >= 2 and sys.argv[1].isdecimal():
22          nb = int(sys.argv[1])
23      else:
24          nb = 3
25
26      # Résoudre hanoï
27      resoudre_hanoi(nb, 'A', 'C', 'B')
```

8 Vers du fonctionnel

Exercice 11 : Vers une approche fonctionnelle : map

Dans cet exercice, on souhaite produire une nouvelle liste à partir d'une liste existante. Après avoir expérimenté sur quelques cas particuliers, nous mettons en œuvre une solution plus générale.

On utilisera les fichiers `map.py` et `test_map.py`.

11.1. Écrire une fonction qui, pour une liste d'entiers donnée, retourne une nouvelle liste qui contient tous les éléments de la première mis au carré. Appliquée sur la liste `[5, 2, 3, 0, 2]`, elle retournera donc `[25, 4, 9, 0, 4]`.

Solution :

```
1   def au_carre(liste):
2       '''
3       Retourne une liste qui contient les éléments de liste au carré.
4
5       :param liste: la liste d'origine
6       :type liste: un itérable de nombres
7       :return: la liste des carrés des éléments de liste
8       :type: une liste de nombres
9       '''
10      carres = []
11      for element in liste:
12          valeur = element ** 2
13          carres.append(valeur)
14      return carres
```

11.2. Écrire une fonction qui, pour une liste d'entiers donnée, retourne une nouvelle liste qui contient tous les éléments de la première divisés (division entière) par 2. Appliquée sur la liste `[5, 2, 3, 0, 2]`, elle retournera donc `[2, 1, 1, 0, 1]`|.

Solution :

```python
def div2(liste):
    '''
    Retourne une liste qui contient les éléments de liste divisés par 2.

    :param liste: la liste d'origine
    :type liste: un itérable de nombre
    :return: la liste des éléments de liste divisés par 2
    :type: une liste de réels
    '''
    moities = []
    for element in liste:
        valeur = element / 2
        moities.append(valeur)
    return moities
```

11.3. Écrire une fonction qui, pour une liste d'entiers donnée, retourne une nouvelle liste qui contient vrai si l'élément correspondant de la première liste est pair, faux sinon. Appliquée sur la liste `[5, 2, 3, 0, 2]`, elle retournera donc `[False, True, False, True, True]`.

Solution :

```python
def vers_est_pair(liste):
    '''
    Retourne une liste de booléens, vrai si l'élément au même indice de liste est pair.

    :param liste: la liste d'origine
    :type liste: un itérable
    :return: une liste de booléens, le ième élément est vrai ssi liste[i] est pair
    :type: une liste de booléens
    '''
    resultat = []
    for element in liste:
        valeur = element % 2 == 0
        resultat.append(valeur)
    return resultat
```

11.4. On constate que la structure du code des fonctions précédentes est la même. Proposer une fonction, nommée **`map`**, qui permet de généraliser les fonctions précédentes et d'autres qui seraient sur le même modèle comme par exemple produire la liste des factorielles des nombres d'une liste. Le principe est de représenter par un ou plusieurs paramètres les parties variables des premières fonctions écrites.

Solution :

```python
def map(liste, f):
    '''
    Retourner une liste telles que sont ième élément est le ième élément
    auquel est appliqué la fonction f.

    :param liste: la source de données
    :type liste: un iterable
    :param f: une fonction qui prend en paramètre un élément de liste
```

```
    :type f: fonction à un paramètre de même type que les éléments de liste
    '''
    resultat = []
    for element in liste:
        valeur = f(element)
        resultat.append(valeur)
    return resultat
```

Voici un exemple d'utilisation de `map`.

```
def exemple_map():
    def carre(x):
        return x ** 2

    assert map([1, 2, 3], carre) == [1, 4, 9]

    assert map([1, 2, 3], lambda x : x / 2) == [.5, 1.0, 1.5]
    assert map([1, 2, 3], lambda x : x % 2 == 0) == [False, True, False]

    print('ok.')
```