Les bases : exercices corrigés en Python
Corrigé

Consignes : Les exercices 2, 4, 6 et 7 sont facultatifs. Dans le cas de l'exercice 5, on pourra se limiter au cas des puissances positives (x^n avec $n \geq 0$).

Objectifs

— Raffiner des problèmes simples ;
— Écrire quelques algorithmes simples ;
— Savoir utiliser les types de base ;
— Savoir utiliser les instructions « élémentaires » : d'entrée/sortie, affichage, bloc...
— Manipuler les conditionnelles ;
— Manipuler les répétitions.

Exercice 1 : Résolution d'une équation du second degré 1
Exercice 2 : Résolution numériquement correcte d'une équation du second degré 5
Exercice 3 : Le nombre d'occurrences du maximun 7
Exercice 4 : Nombres amis 9
Exercice 5 : Puissance 14
Exercice 6 : Amélioration du calcul de la puissance entière 18
Exercice 7 : Nombres de Amstrong 21

Exercice 1 : Résolution d'une équation du second degré
Soit l'équation du second degré $ax^2 + bx + c = 0$ où a, b et c sont des coefficients réels. Écrire un programme qui saisit les coefficients et affiche les solutions de l'équation.
Indication : Les solutions sont cherchées dans les réels. Ainsi, dans le cas général, en considérant le discriminant $\Delta = b^2 - 4ac$, l'équation admet comme solutions analytiques :

$$\begin{cases} \Delta < 0 & \text{pas de solutions réelles.} \\ \Delta = 0 & \text{une solution double : } \frac{-b}{2a} \\ \Delta > 0 & \text{deux solutions : } x_1 = \frac{-b-\sqrt{\Delta}}{2a} \text{ et } x_2 = \frac{-b+\sqrt{\Delta}}{2a} \end{cases}$$

Quelles sont les solutions de l'équation si le coefficient a est nul ?
Solution : Voici un raffinage possible :

```
R0 : Résoudre l'équation du second degré

R1 : Raffinage De « Résoudre l'équation du second degré »
    |  Saisir les 3 coefficients                  a, b, c: out RÉEL
    |                              -- les coefficients de l' équation
```

```
      |  Calculer et afficher les solutions

R2 : Raffinage De « Saisir les 3 coefficients »
      |  Écrire("Entrer␣les␣valeurs␣de␣a,␣b␣et␣c␣:␣")
      |  Lire(a, b, c)

R2 : Raffinage De « Calculer et afficher les solutions »
      |  Si équation du premier degré Alors
      |      Résoudre l'équation du premier degré bx + c = 0
      |  Sinon
      |      Calculer le discriminant
      |          --> delta: RÉEL        -- le discriminant de ax² + bx + c = 0
      |      Si discriminant nul Alors           -- une solution double
      |          Écrire("Une␣solution␣double␣:␣", - B / (2 * A))
      |      SinonSi discriminant positif Alors  -- deux solutions distinctes
      |          Écrire("Deux␣solutions␣:␣")
      |          Écrire((-B + √Delta) / (2*A))
      |          Écrire((-B - √Delta) / (2*A))
      |      Sinon       { discriminant négatif }        -- pas de solution dans IR
      |          Écrire("Pas␣de␣solution␣réelle")
      |      FinSi
      |  FinSi

R3 : Raffinage De « Résoudre l'équation du premier degré bx + c = 0 »
      |  Si B = 0 Alors
      |      Si C = 0 Alors
      |          Écrire("Tout␣IR␣est␣solution")
      |      Sinon
      |          Écrire("Pas␣de␣solution")
      |      FinSi
      |  Sinon
      |      Écrire("Une␣solution␣:␣", - C / B)
      |  FinSi
```

Remarque : Notons que nous n'avons pas un premier niveau de raffinage en trois étapes, *saisir*, *calculer*, *afficher* car il est difficile de dissocier les deux dernières étapes car la forme des racines de l'équation du second est très variable et ne peut pas être facilement capturée par un type (deux solution distinctes, une solution double, pas de solution, une infinité de solution). Aussi, nous avons regroupé calcul et affichage.

Et l'algorithme correspondant :

```
Algorithme second_degré

    -- Résoudre l'équation du second degré

Variables
    a, b, c: Réel       -- les coefficients de l'équation
    delta: Réel         -- le discriminant

Début
```

```
    -- Saisir les 3 coefficients
    Écrire("Entrer les valeurs de a, b et c : ")
    Lire(a, b, c)

    -- Calculer et afficher les solutions
    Si a = 0 Alors                -- équation du premier degré
        -- Résoudre l'équation du premier degré bx + c = 0
        Si B = 0 Alors            -- équation constante
            Si C = 0 Alors
                Écrire("Tout IR est solution")
            Sinon
                Écrire("Pas de solution")
            FinSi
        Sinon                     -- équation réellement du premier degré
            Écrire("Une solution : ", - C / B)
        FinSi

    Sinon                         -- équation réellement du second degré
        -- Calculer le discriminant delta
        delta <- b*b - 4*a*c

        -- Déterminer et afficher les solutions
        Si delta = 0 Alors                -- une solution double
            Écrire("Une solution double : ", - B / (2 * A))
        SinonSi delta > 0 Alors -- deux solutions distinctes
            Écrire("Deux solutions : ")
            Écrire((-B + sqrt(delta)) / (2*A))
            Écrire((-B - sqrt(delta)) / (2*A))
        Sinon   { discriminant négatif }        -- pas de solution dans IR
            Écrire("Pas de solution réelle")
        FinSi
    FinSi
Fin.
```

```python
# -*- coding: utf-8 -*-
"""
ROLE : Résoudre une équation du second degré a.x^2 + b.x + c = 0
EXEMPLES :
    1 -1  1 --> Deux solutions : 1.61803398875 (nombre d'or) et -0.61803398875
    1  1  1 --> Pas de solution réelle
    1 -4  4 --> Une solution double : 2.0
    0  1  2 --> Une solution unique : -2.0
    0  0  1 --> Pas de solution
    0  0  0 --> Tous les réels sont solutions
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
a = float() ; b = float() ; c = float() # coefficients de l'equation
delta = float() # discriminant
#
```

```
print ("Résolution d'une équation du second degré :")
print ("a.x^2 + b.x + c = 0")

#1.Saisir les coefficients de l'équation
print ("Saisir la valeur des 3 coefficients :")
a = float(input("coefficient a : "))
b = float(input("coefficient b : "))
c = float(input("coefficient c : "))
#2.Resoudre l'equation et afficher les resultats
if a == 0 : #2.1.cas particulier : équation du 1er degré b.x + c = 0
    if b == 0 : #2.1.1.cas très particulier : équation constante c = 0
        if c == 0 :
            print ("Tous les réels sont solutions")
        else :
            print ("Pas de solution")
    else : # 2.1.2.équation réellement du 1er degré (b != 0)
        print ("Une solution unique :", -c / b)
else : #2.2.cas général : équation réellement du 2nd degré (a != 0)
    delta = b * b - 4 * a * c # le discriminant
    if delta > 0 : # deux solutions réelles
        print ("Deux solutions :", \
            (-b + sqrt(delta)) / 2 / a, "et", \
            (-b - sqrt(delta)) / (2 * a))
    # NOTA1 : 1 / 2 / a et 1 / (2 * a) sont 2 écritures équivalentes
    # NOTA2 : le caractère antislash permet de scinder une instruction
    #         sur plusieurs lignes
    elif delta == 0 : # une solution dite double
        print ("Une solution double :", (-b / 2 / a))
    else : # pas de solution réelle
        print ("Pas de solution réelle")
```

Exercice 2 : Résolution numériquement correcte d'une équation du second degré
Utiliser les formules analytiques de l'exercice 1 pour calculer les solutions réelles de l'équation du second degré n'est pas numériquement satisfaisant. En effet, si la valeur de b est positive et proche de celle de $\sqrt{\Delta}$, le calcul de x_2 comportera au numérateur la soustraction de deux nombres voisins ce qui augmente le risque d'erreur numérique.

Par exemple, considèrons l'équation $x^2 + 62,10x + 1 = 0$ dont les solutions sont approximativement :

$$x_1 = -62,08390$$
$$x_2 = -0,01610723$$

Dans cette équation, la valeur de b^2 est bien plus grande que le produit $4ac$ et le calcul de Δ conduit donc à un nombre très proche de b. Dans les calculs qui suivent, on ne tient compte que des 4 premiers chiffres significatifs.

$$\sqrt{\Delta} = \sqrt{(62,10)^2 - 4 \times 1 \times 1} = \sqrt{3856,41 - 4} = 62,06$$

On obtient alors :

$$x_2 = \frac{-62,10 + 62,06}{2} = -0,02$$

L'erreur relative sur x_2 induite est importante :

$$\frac{|-0,01611 + 0,02|}{|-0,01611|} = 0,24 \quad \text{(soit 24\% d'erreur).}$$

En revanche, pour le calcul de x_1 l'addition de deux nombres pratiquement égaux ne pose pas de problème et les calculs conduisent à un erreur relative faible $(3,210^{-4})$.

Pour diminuer l'erreur numérique sur x_2, il suffirait de réorganiser les calculs :

$$x_2 = \left(\frac{-b+\sqrt{\Delta}}{2a}\right)\left(\frac{-b-\sqrt{\Delta}}{-b-\sqrt{\Delta}}\right) = \frac{b^2-\Delta}{2a(-b-\sqrt{\Delta})} = \frac{-2c}{b+\sqrt{\Delta}}$$

Mais il existe une solution plus simple qui consiste à calculer d'abord x_1 par la formule classique puis en déduire x_2 en considérant que le produit des racines est égal à c/a ($x_1x_2 = c/a$).

Conclusion : En pratique, si b est négatif, on calcule d'abord la racine $\frac{-b+\sqrt{\Delta}}{2a}$, si b est positif, on calcule la racine $\frac{-b-\sqrt{\Delta}}{2a}$. L'autre racine est calculée à l'aide du produit c/a.

Écrire le programme qui calcule les racines réelles de l'équation du second degré en s'appuyant sur cette formule.

Solution :

```
# -*- coding: utf-8 -*-
"""
ROLE : Résoudre une équation du second degré a.x^2 + b.x + c = 0
EXEMPLES :
    1 -1  1 --> Deux solutions : 1.61803398875 (nombre d'or) et -0.61803398875
    1  1  1 --> Pas de solution réelle
    1 -4  4 --> Une solution double : 2.0
    0  1  2 --> Une solution unique : -2.0
```

```
    0  0  1 --> Pas de solution
    0  0  0 --> Tous les réels sont solutions
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
a = float() ; b = float() ; c = float() # coefficients de l'equation
delta = float() # discriminant
x1 = float() ; x2 = float # solutions de l'équation dans le cas général
#
print ("Résolution␣d'une␣équation␣du␣second␣degré␣:")
print ("a.x^2␣+␣b.x␣+␣c␣=␣0")

#1.Saisir les coefficients de l'équation
print ("Saisir␣la␣valeur␣des␣3␣coefficients␣:")
a = float(input("coefficient␣a␣:␣"))
b = float(input("coefficient␣b␣:␣"))
c = float(input("coefficient␣c␣:␣"))
#2.Resoudre l'equation et afficher les resultats
if a == 0 : #2.1.cas particulier : équation du 1er degré b.x + c = 0
    if b == 0 : #2.1.1.cas très particulier : équation constante c = 0
        if c == 0 :
            print ("Tous␣les␣réels␣sont␣solutions")
        else :
            print ("Pas␣de␣solution")
    else : # 2.1.2.équation réellement du 1er degré (b != 0)
        print ("Une␣solution␣unique␣:", -c / b)
else : #2.2.cas général : équation réellement du 2nd degré (a != 0)
    delta = b * b - 4 * a * c # le discriminant
    if delta > 0 : # deux solutions réelles
        #2.2.1.selon le signe de b, on calcule différemment la 1ère solution
        #   de manière à faire un calcul avec la meilleure précision
        if b < 0 :
            x1 = x1 = (-b + sqrt(delta)) / 2 / a
        else :
            x1 = x1 = (-b - sqrt(delta)) / 2 / a
        #2.2.2.et la seconde solution s'en déduit avec c/a :
        x2 = c / a / x1
        print ("Deux␣solutions␣:", x1, "et", x2)
    elif delta == 0 : # une solution dite double
        print ("Une␣solution␣double␣:", (-b / 2 / a))
    else : # pas de solution réelle
        print ("Pas␣de␣solution␣réelle")
```

Exercice 3 : Le nombre d'occurrences du maximun
Compléter le programme de l'exercice 2 (Exercices résolus en Python, Semaine 1) qui calcule les statistiques sur une série de valeurs réelles pour qu'il affiche également le nombre d'occurrences de la plus grande valeur, c'est-à-dire le nombre de valeurs de la série qui correspondent à la valeur maxiamle.

Par exemple, pour la série 1 2 3 1 2 3 3 2 3 1 0, le max est 3 et il y a quatre occurrences de 3. Dans la série 1 2 3 3 3 3 1 2 4 1 2 3 0, il y a une seule occurrence du maximun qui est 4.

Solution : Le principe est d'ajouter une nouvelle variable d'accumulation, `nb_max`, qui comptabilise le nombre d'occurrences du max rencontrées. Elle est initialisée à 1 sur la première valeur. Elle est remise à 1 dès qu'un nouveau maximum est identifié. Elle est augmentée de 1 si la valeur lue est égale au maximum précédent.

```python
# -*- coding: utf-8 -*-
"""
ROLE : Afficher la moyenne, la plus grande valeur, la plus petite valeur
    et le nombre d'occurences de la plus grande valeur d'une serie
    de valeurs reelles saisies au clavier. La série est terminée par la
    saisie de la valeur 0 qui ne sera pas incluse dans la série.
EXEMPLE :
    1 3 2 3 1 (0) --> moyenne=2.0, plus petite valeur=1.0, plus grande=3.0,
                      nombre d'occurrences du max = 2
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
x = float()         # chaque reel saisi au clavier
nb = int()          # le nombre de valeurs saisies de la serie
somme = float() # la somme des valeurs saisies de la serie
min = float()   # la plus petite des valeurs saisies de la série
max = float()   # la plus grande des valeurs saisies de la série
nbMax = int()   # le nombre d'occurrences du max
#
print ("Moyenne d'une série de valeurs")
#1.Afficher la consigne
print ("Donnez une serie d'entiers qui se termine par le nombre 0")
#2.Saisir le premier reel
x = float(input("Saisir un nombre (0 pour finir) : "))
if x == 0 : # indiquer que les statistiques demandées ne peuvent être faites
    print ("La série est vide.")
    print ("Les statistiques demandées n'ont pas de sens !")
else :
    #3.Initialiser les variables statistiques avec x
    nb = 1 ; somme = x ; min = x ; max = x ; nbMax = 1
    #4.Saisir une nouvelle valeur x
    x = float(input("Saisir un nombre (0 pour finir) : "))
    #5.Traiter les éléments de la série
    while x != 0 : # traiter le réel venant d'être saisi
        #5.1.Mettre à jour les variables statistiques en fonction de x
        nb = nb + 1
        somme = somme + x
```

```
        if x > max : max = x
        elif x == max : nbMax = nbMax + 1
        elif x < min : min = x
        #5.2.Saisir un nouveau réel x
        x = float(input("Saisir␣un␣nombre␣(0␣pour␣finir)␣:␣"))
    #6.Afficher le résultat
    print ("Moyenne␣=␣", somme/nb)
    print ("Plus␣petite␣valeur␣=", min)
    print ("Plus␣grande␣valeur␣=", max)
    print ("Nombre␣d’occurrences␣du␣max␣=", nbMax)
```

Exercice 4 : Nombres amis

Deux nombres N et M sont amis si la somme des diviseurs de M (en excluant M lui-même) est égale à N et la somme des diviseurs de N (en excluant N lui-même) est égale à M.

Écrire un programme qui affiche tous les couples (N, M) de nombres amis tels que $0 < N < M \leq MAX$, MAX étant lu au clavier.

Indication : Les nombres amis compris entre 1 et 100000 sont (220, 284), (1184, 1210), (2620, 2924), (5020, 5564), (6232, 6368), (10744, 10856), (12285, 14595), (17296, 18416), (66928, 66992), (67095, 71145), (63020, 76084), (69615, 87633) et (79750, 88730).

Solution : L'idée est de faire un parcours de tous les couples possibles et de se demander s'ils forment un couple de nombres amis ou non. Pour un n et un m donnés, il s'agit alors de calculer la somme de leurs diviseurs. C'est en fait deux fois le même problème. Il s'agit de calculer la somme des diviseurs d'un entier p. Naïvement, on peut regarder si les entiers de 2 à $p - 1$ sont des diviseurs de p.

Une première optimisation consiste à remarquer que p n'a pas de diviseurs au delà de $p/2$. En fait, il est plus intéressant de remarquer que si i est diviseur de p alors p/i est également un diviseur de p. Il suffit donc de ne considérer comme diviseurs potentiels que les entiers compris dans l'intervalle 2..$\sqrt{p}$. Il faut toutefois faire attention à ne pas comptabiliser deux fois $\sqrt{p}$.

```
R0 : Afficher les couples de nombres amis (N, M) avec 1 < N < M <= Max

R1 : Raffinage De « R0 »
      | Pour m <- 2 JusquÀ m = Max Faire
      |   | Pour n <- 2 JusquÀ n = m - 1 Faire
      |   |   | Si n et m amis Alors
      |   |   |   | Afficher le couple (n, m)
      |   |   | FinSi
      |   | FinPour
      | FinPour

R2 : Raffinage De « n et m amis »
      | Résultat <- (somme des diviseurs de N) = M
      |         Et (somme des diviseurs de M) = N

R3 : Raffinage De « somme des diviseurs de p »
      | somme <- 1
      | Pour i <- 2 JusquÀ i = racine_carrée(p) - 1 Faire
      |   | Si i diviseur de p Alors
      |   |   | somme <- somme + i + (p Div i)
      |   | FinSi
      | FinPour
      | Si p est un carré parfait Alors
      |   | somme <- somme + i
      | FinSi
```

L'algorithme est alors le suivant. Notez quelques optimisations qui font que l'algorithme de respecte pas complètement le raffinage.

```
Algorithme nb_amis

```

```
    -- Afficher les couples de nombres amis (N, M) avec 1 < N < M <= MAX

Variables
    n, m: Entier          -- pour représenter les couples possibles
    somme_n: Entier       -- somme des diviseurs de n
    somme_m: Entier       -- somme des diviseurs de m

Début
    Pour m <- 2 JusquÀ m = MAX Faire
        -- calculer la somme des divieurs de m
        -- Remarque : on peut déplacer cette étape à l'extérieur de la
        -- boucle car elle ne dépend pas de n (optimisation).
        somme_m <- 1
        Pour i <- 2 JusquÀ i = racine_carrée(m) - 1 Faire
            Si i diviseur de m Alors
                somme_m <- somme_m + i + (m Div i)
            FinSi
        FinPour
        Si m est un carré parfait Alors
            somme_m <- somme_m + i
        FinSi


        Pour n <- 2 JusquÀ n = m - 1 Faire
            -- calculer la somme des divieurs de n
            somme_n <- 1
            Pour i <- 2 JusquÀ i = racine_carrée(n) - 1 Faire
                Si i diviseur de n Alors
                    somme_n <- somme_n + i + (n Div i)
                FinSi
            FinPour
            Si n est un carré parfait Alors
                somme_n <- somme_n + i
            FinSi

            -- déterminer si n et m sont amis
            Si (somme_n = m) Et (somme_m = n) Alors { n et m sont amis }
                Afficher le couple (n, m)
            FinSi
        FinPour
    FinPour

Fin.
```

```python
# -*- coding: utf-8 -*-
"""
ROLE : Afficher les couples de nombres amis n et m inférieurs à une borne
    max saisie au clavier. n et m vérifient 0 < n < m <= max.
    Deux nombres n et m sont amis si la somme des diviseurs de n
    (lui-même exclu) est égale à m et la somme des diviseurs de m
    (lui-même exclu) est égale à n. Pour chaque couple trouvé,
    on affichera aussi les listes des diviseurs de chaque nombre.
```

```python
NOTA : cette version contient deux boucles 'Pour' imbriquées et n'est pas
    très rapide. Sous SPYDER, il est posible de "suivre" plus ou moins
    l'avancée de l'exécution en cliquant sur l'onglet Explorateur de variables
    dans lequel on peut voir la valeur de m et n (entre autres). On peut
    interrompre l'exécution en cliquant sur le bouton 'Carré rouge' situé
    en haut à droite du volet de la console IPython.
EXEMPLE : couple de nombres amis inférieurs à 1000 : 220 et 284
    220 = 1+2+4+71+142 qui sont les diviseurs de 284
    284 = 1+2+4+5+10+11+20+22+44+55+110 qui sont les diviseurs de 220
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
max = int()               # borne superieure de recherche
n = int() ; m = int() # pour representer les couples possibles
sommeDivsN = int()        # somme des diviseurs de n
sommeDivsM = int()        # somme des diviseurs de m
listeDivsN = str()    # liste des diviseurs de n
listeDivsM = str()    # liste des diviseurs de m
diviseur = int()          # variable de boucle (diviseur courant potentiel)
#
print ("Recherche de couples de nombres amis")
#1.Saisir la borne maximale de recherche
max = int(input("Borne maximale de recherche : "))
#2.Chercher et afficher les couples d'amis inférieurs à cette borne
for n in range(2,max) : # tester si n peut être ami d'un autre nombre
    #2.1.calculer la somme des diviseurs de n (lui-même exclu)
    sommeDivsN = 1 ; listeDivsN = "1" # 1 divise tout nombre
    for diviseur in range(2,n//2+1) : # tester si diviseur divise n
        if n % diviseur == 0 : # si oui, cumuler ce diviseur
            sommeDivsN = sommeDivsN + diviseur
            listeDivsN = listeDivsN + "+" + str(diviseur)
    #2.2.tester si n peut être ami avec un nombre m strictement supérieur
    for m in range(n+1,max+1) : # tester si m peut être ami avec n
        #2.2.1.calculer la somme des diviseurs de m (lui-même exclu)
        sommeDivsM = 1 ; listeDivsM = "1" # 1 divise tout nombre
        for diviseur in range(2,m//2+1) : # tester si diviseur divise m
            if m % diviseur == 0 : # si oui, cumuler ce diviseur
                sommeDivsM = sommeDivsM + diviseur
                listeDivsM = listeDivsM + "+" + str(diviseur)
        #2.2.2.tester si m est ami avec n
        if (m == sommeDivsN) and (n == sommeDivsM) :
            print (n, "et", m, "sont amis :")
            print (" ", n,"=",listeDivsM,"qui sont les diviseurs de",m)
            print (" ", m,"=",listeDivsN,"qui sont les diviseurs de",n)
```

Une solution plus efficace consiste à constater que pour un entier M compris entre 2 et MAX, le seul nombre ami possible est la somme de ses diviseurs que l'on note `somme_m`. Il reste alors à vérifier si la somme des diviseurs de `somme_m` est égale à M pour savoir si `somme_m` et M sont amis. Le fait de devoir afficher les couples dans l'ordre croissant nous conduit à ne considérer que les sommes de diviseurs inférieures à M.

À titre indicatif, pour Max = 1000000, ce deuxième algorithme termine en 1 minute 23 alors que dans le même temps, seuls les sept premiers résultats sont trouvés avec le premier algorithme. Les solutions suivantes sont trouvées au bout d'une minute 32 secondes, 2 minutes 46, 5 minutes 35, 20 minutes, etc.

```
R0 : Afficher les couples de nombres parfaits (N, M) avec 1 < N < M <= Max

R1 : Raffinage De « R0 »
        | Pour m <- 2 JusquÀ m = Max Faire
        |   | Déterminer la somme des diviseurs de m
        |   |                                           m: in ; somme_m: out Entier
        |   | n <- somme_m
        |   | Si n < m Alors       { n est un nb amis potentiel }
        |   |   | Déterminer la somme des diviseurs de n (somme_n)
        |   |   | Si somme_n = m Alors      { somme_m et m amis }
        |   |   |   | Afficher le couple (n, m)
        |   |   | FinSi
        |   | FinPour
        | FinPour

R2 : Raffinage De « somme des diviseurs de p »
        | somme <- 1
        | Pour i <- 2 JusquÀ i = racine_carrée(p) - 1 Faire
        |    | Si i diviseur de p Alors
        |    |    | somme <- somme + i + (p Div i)
        |    | FinSi
        | FinPour
        | Si p est un carré parfait Alors
        |    | somme <- somme + i
        | FinSi
```

Voici l'algorithme correspondant.

```
Algorithme nb_amis

   -- Afficher les couples de nombres amis (N, M) avec 1 < N < M <= MAX

Variables
    m: Entier              -- pour parcourir les entiers de 2 à MAX
    n: Entier              -- l'amis candidat
    somme_m: Entier        -- somme des diviseurs de m
    somme_n: Entier        -- somme des diviseurs de somme_m correspondant à n

Début
    Pour m <- 2 JusquÀ m = MAX Faire
        -- Déterminer la somme des divieurs de m
        somme_m <- 1
        Pour i <- 2 JusquÀ i = racine_carrée(m) - 1 Faire
            Si i diviseur de m Alors
                somme_m <- somme_m + i + (m Div i)
            FinSi
        FinPour
```

```
        Si m est un carré parfait Alors
            somme_m <- somme_m + i
        FinSi

        { somme_m est la candidat pour n }

        n <- somme_m
        Si n < m Alors  { on s'intéresse au cas  n <= m }
            -- Déterminer la somme des diviseurs de n (somme_n)
            somme_n <- 1
            Pour i <- 2 JusquÀ i = racine_carrée(n) - 1 Faire
                Si i diviseur de n Alors
                    somme_n <- somme_n + i + (n Div i)
                FinSi
            FinPour
            Si n est un carré parfait Alors
                somme_n <- somme_n + i
            FinSi


            Si somme_n = m Alors        { somme_m et m amis }
                Afficher le couple (somme_m, m)
            FinSi
        FinPour

    FinPour
Fin.
```

Exercice 5 : Puissance
Calculer et afficher la puissance entière d'un réel.

Solution : On peut, dans un premier temps, se demander si la puissance entière n d'un réel x a toujours un sens. En fait, ceci n'a pas de sens si x est nul et si n est strictement négatif. D'autre part, le cas $x = 0$, $n = 0$ est indéterminé. On choisit de contrôler la saisie de manière à garantir que nous ne sommes pas dans l'un de ces cas. Nous souhaitons donner à l'utilisateur un message le plus clair possible lorsque la saisie est invalide.

Nous envisageons les jeux de tests suivants :

```
   x   n  -->   x^n

   2   3  -->     8      -- cas nominal
   3   2  -->     9      -- cas nominal
   1   1  -->     1      -- cas nominal
   2  -3  --> 0.125      -- cas nominal (puissance négative)
   0   2  -->     0      -- cas nominal (x est nul)
   0  -2  --> ERREUR     -- division par zéro
   0   0  --> Indéterminé
 1.5   2  --> 2.25       -- x peut être réel
```

Voyons maintenant le principe de la solution. Par exemple, pour calculer 2^3, on peut faire $2 * 2 * 2$. Plus généralement, on multiplie n fois x par lui-même (donc une boucle **Pour**).

Si on essaie ce principe, on constate qu'il ne fonctionne pas dans le cas d'une puissance négative. Prenons le cas de 2^{-3}. On peut le réécrire $(1/2)^3$. On est donc ramené au cas précédent. Le facteur multiplicatif est dans ce cas $1/x$ et le nombre de fois est $-n$.

Nous introduisons donc deux variables intermédiaires qui sont :

```
facteur: Réel     -- facteur multiplicatif pour obtenir les puissances
                  -- successives de x
puissance: Réel -- abs(n).  On a : facteur^puissance = x^n
```

On peut maintenant formaliser notre raisonnement sous la forme du raffinage suivant.

```
R0 : Afficher la puissance entière d'un réel

R1 : Raffinage De « R0 »
     | Saisir avec contrôle les valeurs de x et n      x: out Réel; n: out Entier
     | { (x <> 0) Ou (n > 0) }
     | Calculer x à la puisssance n                    x, n: in ; xn: out Réel
     | Afficher le résultat                            xn: in Réel

R2 : Raffinage De « Saisir avec contrôle les valeurs de x et n »
     | Répéter
     |    | Saisir la valeur de x et n            x: out Réel; n: out Entier
     |    | Contrôler x et n                      x, n: in ; valide: out Booléen
     | JusquÀ valide                              valide: in

R1 : Raffinage De « Calculer x à la puisssance n »
     | Si x = 0 Alors
     |    | xn := 0
     | Sinon
```

```
        |   | Déterminer le facteur multiplicatif et la puissance
        |   |                    x, n: in ;
        |   |                    facteur out Réel ;
        |   |                    puissance: out Entier
        |   | Calculer xn par itération (accumulation)
        |   |                    n, facteur, puissance: in ; xn : out
        | FinSi

R3 : Raffinage De « Calculer xn par itération (accumulation) »
        | xn <- 1;
        | Pour i <- 1 JusquÀ i = puissance Faire
        |   | { Invariant : xn = facteur^i }
        |   | xn <- xn * facteur
        | FinPour
```

On peut alors en déduire l'algorithme suivant :

```
Algorithme puissance

    -- Afficher la puissance entière d'un réel

Variables
    x: Réel                 -- valeur réelle lue au clavier
    n: Entier               -- valeur entière lue au clavier
    valide: Booléen         -- x^n peut-elle être calculée
    xn: Réel                -- x à la puissance n
    facteur: Réel           -- facteur multiplicatif pour obtenir
                            -- les puissances successives
    puissance: Entier;      - abs(n).  On a facteur^puissance = x^n
    i: Entier               -- variable de boucle

Début
    -- Saisir avec contrôle les valeurs de x et n
    Répéter
        -- saisir la valeur de x et n
        Écrire("x␣=␣")
        Lire(x)
        Écrire("n␣=␣")
        Lire(n)

        -- contrôler x et n
        valide <- VRAI
        Si x = 0 Alors
            Si n = 0 Alors
                ÉcrireLn("x␣et␣n␣sont␣nuls.␣x^n est indéterminée.")
                valide <- FAUX
            SinonSi n < 0 Alors
                ÉcrireLn("x␣nul␣et␣n␣négatif.␣x^n n'a pas de sens.")
                valide <- FAUX
            FinSi
        FinSi

```

```
        Si Non valide Alors
            ÉcrireLn("  Recommencez !")
        FinSi
    JusquÀ valide

    -- Calculer x à la puisssance n
    Si x = 0 Alors      -- cas trivial
        xn <- 0
    Sinon
        -- Déterminer le facteur multiplicatif et la puissance
        Si n >= 0 Alors
            facteur <- x
            puissance <- n
        Sinon
            facteur <- 1/x
            puissance <- -n
        FinSi

        -- Calculer xn par itération (accumulation)
        xn <- 1;
        Pour i <- 1 JusquÀ i = puissance Faire
            { Invariant : xn = facteur^i}
            xn <- xn * facteur
        FinPour
    FinSi

    -- Afficher le résultat
    ÉcrireLn(x, "^", n, " = ", xn)
Fin.
```

```python
# -*- coding: utf-8 -*-
"""
ROLE : Afficher la puissance entière d'un nombre réel (sans l'opérateur **)
EXEMPLES :
    2 puissance 3 = 8, 2 puissance - 3 = 0.125, 2 puissance 0 = 0
    0 puissance 3 = 0, 0 puiss. 0 : indéterminé, 0 puiss. -2 : n'a pas de sens
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
x = float()       # valeur réelle saisie au clavier
n = int()         # valeur entière saisie au clavier
valide = bool()   # vrai si x^n peut être calculée
xn = float()      # x à la puissance n
facteur = float() # facteur multiplicatif pour avoir les puissances successives
puissance = int() # abs(n).  On a : facteur^puissance = x^n
i = int()         # variable de boucle
#
print ("Puissance entière d'un nombre")
#1.Saisir avec contrôle les valeurs de x et n
valide = False # a priori
```

```
while not valide : # saisir x et n, et tester leur validité
    #1.1.saisir la valeur de x et n
    print ("Saisir␣un␣nombre␣réel␣x␣et␣son␣exposant␣n␣:")
    x = float(input("x␣=␣")) ; n = int(input("n␣=␣"))
    #1.2.controler la validité de x et n
    valide = True # a priori
    if x == 0 : # cas particulier où la saisie peut être invalide
        if n == 0 :
            print ("x␣et␣n␣sont␣nuls␣:␣x␣puissance␣n␣est␣indeterminé).")
            valide = False
        elif n < 0 :
            print ("x␣nul␣et␣n␣négatif␣(x␣puissance␣n␣n’a␣pas␣de␣sens).")
            valide = False
        if not valide :
            print("Refaire␣la␣saisie␣:")
#2.Calculer x à la puissance n
if x == 0 : # cas trivial où x^n = 0
    xn = 0
else :
    #2.1.Déterminer le facteur multiplicatif et la puissance
    if n >= 0 :
        facteur = x     ; puissance =  n # puissance >= 0
    else :
        facteur = 1 / x ; puissance = -n # puissance > 0
    #2.2.Calculer xn par iteration (accumulation)
    xn = 1
    for i in range(1,puissance+1) : # invariant : xn = facteur^i
        xn = xn * facteur
#3.Afficher le résultat
print (x, "puissance", n, "=", xn)
```

Exercice 6 : Amélioration du calcul de la puissance entière
Améliorer l'algorithme de calcul de la puissance (exercice 5 du Exercices corrigés en Python, Semaine 1) 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 $9 = 3^2$.

Solution : Nous nous appuyons sur les mêmes variables que pour le calcul « naturel » de la puissance (exercice 5). Nous allons continuer à calculer x^n par accumulation. Nous avons l'invariant suivant :

$$x^n = \mathrm{xn} * \mathrm{facteur}^{\mathrm{puissance}}$$

Lors de l'initialisation, cet invariant est vrai (`xn` vaut 1 et facteur et puissance sont tels qu'ils valent x^n).

D'après la formule donnée dans l'énoncé, à chaque itération de la boucle, deux cas sont à envisager :

— soit `puissance` est paire. On peut alors l'écrire $2 * p$. On a alors :

$$\begin{aligned} x^n &= \mathrm{xn} * \mathrm{facteur}^{\mathrm{puissance}} \\ &= \mathrm{xn} * \mathrm{facteur}^{(2\,*\,\mathrm{p})} \\ &= \mathrm{xn} * (\mathrm{facteur}^2)^{\mathrm{p}} \end{aligned}$$

On peut donc faire :

```
facteur <- facteur * facteur
puissance <- puissance Div 2    -- car p = puissance Div 2
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car divisée par 2).

— soit `puissance` est impaire. On peut alors l'écrire $2 * p + 1$. On a alors :

$$\begin{aligned} x^n &= \mathrm{xn} * \mathrm{facteur}^{\mathrm{puissance}} \\ &= \mathrm{xn} * \mathrm{facteur}^{(2\,*\,\mathrm{p}\,+\,1)} \\ &= \mathrm{xn} * (\mathrm{facteur} * \mathrm{facteur}^{(2*\mathrm{p})}) \\ &= (\mathrm{xn} * \mathrm{facteur}) * \mathrm{facteur}^{(2*\mathrm{p})} \end{aligned}$$

On peut donc faire :

```
xn <- xn * facteur
puissance <- puissance - 1
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car diminuée de 1).

On peut alors en déduire le raffinage suivant :

```
R3 : Raffinage De « Calculer xn par itération (accumulation) »
        | xn <- 1;
        | TantQue puissance > 0 Faire
```

```
        |   | { Variant : puissance }
        |   | { Invariant : x^n = xn * facteur^puissance }
        |   | Si puissance Div 2 = 0 Alors        { puissance = 2 * p }
        |   |  | puissance <- puissance Div 2
        |   |  | facteur <- facteur * facteur
        |   | Sinon                               { puissance = 2 * p + 1 }
        |   |  | puissance <- puissance - 1
        |   |  | xn <- xn * facteur
        |   | FinSi
        | FinTQ
```

```python
# -*- coding: utf-8 -*-
"""
ROLE : Afficher la puissance entière d'un nombre réel (sans l'opérateur **)
    Version dite "indienne" avec usage du carré (exécution plus rapide)
EXEMPLES :
    2 puissance 3 = 8, 2 puissance - 3 = 0.125, 2 puissance 0 = 0
    0 puissance 3 = 0, 0 puissance 0 : indéterminé, 0 puissance -2 : pas de sens
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
x = float()       # valeur réelle saisie au clavier
n = int()         # valeur entière saisie au clavier
valide = bool()   # vrai si x^n peut être calculée
xn = float()      # x à la puissance n
facteur = float() # facteur multiplicatif pour avoir les puissances successives
puissance = int() # abs(n).  On a : facteur^puissance = x^n
#
print ("Puissance entière d'un nombre")
#1.Saisir avec contrôle les valeurs de x et n
valide = False # a priori
while not valide : # saisir x et n, et tester leur validité
    #1.1.saisir la valeur de x et n
    print ("Saisir un nombre réel x et son exposant n :")
    x = float(input("x = ")) ; n = int(input("n = "))
    #1.2.controler la validité de x et n
    valide = True # a priori
    if x == 0 : # cas particulier où la saisie peut être invalide
        if n == 0 :
            print ("x et n sont nuls : x puissance n est indeterminé).")
            valide = False
        elif n < 0 :
            print ("x nul et n négatif (x puissance n n'a pas de sens).")
            valide = False
        if not valide :
            print("Refaire la saisie :")
#2.Calculer x à la puissance n
if x == 0 : # cas trivial où x^n = 0
    xn = 0
else :
    #2.1.Déterminer le facteur multiplicatif et la puissance
```

```
    if n >= 0 :
        facteur = x     ; puissance =  n # puissance >= 0
    else :
        facteur = 1 / x ; puissance = -n # puissance > 0
    #2.2.Calculer xn par iteration (accumulation)
    xn = 1
    while puissance > 0 : # invariant : x puiss. n = xn * facteur puiss. puissance
        if puissance % 2 == 0 :
            facteur = facteur * facteur ; puissance = puissance // 2
        else :
            xn = xn * facteur ; puissance = puissance - 1
#3.Afficher le résultat
print (x, "puissance", n, "=", xn)
```

Exercice 7 : Nombres de Amstrong

Les *nombres de Amstrong* appelés parfois *nombres cubes* sont des nombres entiers qui ont la particularité d'être égaux à la somme des cubes de leurs chiffres. Par exemple, 153 est un nombre de Amstrong car on a :

$$153 = 1^3 + 5^3 + 3^3.$$

Afficher tous les nombres de Amstrong sachant qu'ils sont tous compris entre 100 et 499.

Indication : Les nombres de Amstrong sont : 153, 370, 371 et 407.

Solution :

```
R0 : Afficher les nombres de Amstrong compris entre 100 et 499.
```

On peut envisager (au moins) deux solutions pour résoudre ce problème.

Solution 1. La première solution consiste à essayer les combinaisons de trois chiffres qui vérifient la propriété. On prend alors trois compteurs : un pour les unités, un pour les dizaines et un pour les centaines. Les deux premiers varient de 0 à 9. Le dernier de 1 à 4. Ayant les trois chiffres, on peut calculer la somme de leurs cubes puis le nombre qu'ils forment et on regarde si les deux sont égaux.

Ceci se formalise dans le raffinage suivant :

```
R1 : Raffinage De « Afficher les nombres de Amstrong »
    Pour centaine <- 1 JusquÀ centaine = 4 Faire
        Pour dizaine <- 1 JusquÀ dizaine = 9 Faire
            Pour unité <- 1 JusquÀ unité = 9 Faire
                Déterminer le cube
                Déterminer le nombre
                Si nombre = cube Alors
                    Afficher nombre
                FinSi
            FinPour
        FinPour
    FinPour
```

On en déduit alors le programme Python suivant.

```
# -*- coding: utf-8 -*-
"""
ROLE : Afficher les nombres de Amstrong (nombres entre 100 et 499 égaux à la
    somme des cubes de leurs 3 chiffres).
NOTA : Version de base avec 3 boucles imbriquées
EXEMPLE : 153 = 1**3 + 5**3 + 3**3 = 1 + 125 + 27
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
nb = int()   # le nombre considéré
centaine = int() ; dizaine = int() ; unite = int() # les trois chiffres de nb
sommeCubes = int() # le somme des cubes des trois chiffres
#
print ("Nombres␣de␣Amstrong␣(version␣simple)")
```

```
for centaine in range(1,5) : # chiffre des centaines de 1 à 4 inclus
    for dizaine in range(0,10) : # chiffre des dizaines de 0 à 9 inclus
        for unite in range(0,10) : # chiffre des unités de 0 à 9 inclus
            #1.determiner la somme des cubes
            sommeCubes = centaine**3 + dizaine**3 + unite**3
            #2.determiner le nombre
            nb = centaine * 100 + dizaine * 10 + unite
            #3.tester
            if nb == sommeCubes : # c’est un nombre de Amstrong
                print (nb,"=",centaine,"**3 +",dizaine,"**3 +",unite,"**3 =", \
                    centaine**3, "+", dizaine**3, "+", unite**3)
```

On remarque que les opérations peuvent être réoganisées pour augmenter les performances en temps de calcul. En particulier, il est inutile de faire des calculs à l’intérieure d’une boucle s’ils ne dépendent pas de la boucle. Ainsi, le calcul du cube des centaines peut se faire dans la boucle la plus externe au lieu de le faire dans la plus interne.

Solution 2. La deuxième solution consiste à parcourir tous les entiers compris entre 100 et 499 et à regarder s’ils sont égaux à la somme des cubes de leurs chiffres. La difficulté est alors d’extraite les chiffres. L’idée est d’utiliser la division entière par 10 et son reste.

On obtient le raffinage suivant.

```
R1 : Raffinage De « Afficher les nombres de Amstrong »
    Pour nombre <- 100 JusquÀ centaine = 499 Faire
        Déterminer les chiffres de nb
        Déterminer le cube des chiffres
        Si nombre = cube Alors
            Afficher nombre
        FinSi
    FinPour

R2 : Raffinage De « Déterminer les chiffres de nb »
    unité <- nombre Mod 10
    dizaine <- (nombre Div 10) Mod 10
    centaine <- nombre Div 100
```