Mathématique informatique

IEPS Mouscron, informatique de gestion
année 2019-2020
Herpoel Quentin

Exercice 1

On demande de réaliser un programme en Python. Celui-ci permettra l'entrée d'un message à crypter et/ou à décrypter (maximum 80 caractères en utilisant 27 possibilités, les 26 caractères majuscules + le blanc).

Pour cet exercice j'ai choisi de développer une solution d'encryptage RSA. Pour cela, j'ai eu besoin de pouvoir créer des clés primaires aléatoires.

```
def isPrime(n):
    for i in range(2, int(math.sqrt(n)+1)):
        if n % i == 0:
            return False
    return n>1

def keyGen():
    global p, q, e, r
    p = random.choice([m for m in range(minPrime, maxPrime) if isPrime(m)])
    q = random.choice([m for m in range(minPrime, maxPrime) if (isPrime(m)) & (m != p)])
    e = random.choice([m for m in range(minPrime, maxPrime) if (isPrime(m)) & (m > p) & (m > q)]) #clé d'encryptage
    r = p * q
    return (e, r)
```

Ces fonctions vont permettre de les générer sur un range ici défini :

```
minPrime = 10
maxPrime = 1000
```

Nous allons donc entrer notre message, et pour faciliter les transformations, nous allons le transformer en une liste de valeurs ASCII pour chaque caractère (qui sont un nombre unique représentant chaque valeur entrée, ex: "Q" vaut 81, et "q"vaut 113)

```
64    def getOrd(message):
65        messageList = list(message)
66        result = [ord(x) for x in messageList]
67        return result
```

A partir de là, il s'agit d'appliquer les formules d'encryptage RSA indiquées et de les formuler dans des fonctions :

```
46    def encrypt(messageToEncrypt, publicKeyE, publicKeyR): #ok
47        global encryptedMessage
48        encryptedMessage = ((messageToEncrypt ** publicKeyE) % publicKeyR)
49        return encryptedMessage
50
51    def encryptList(listToEncrypt, publicKeyE, publicKeyR):
52        encryptedList = [encrypt(x, publicKeyE, publicKeyR) for x in listToEncrypt]
53        return encryptedList
```

La première fonction est la méthode d'encryptage classique et la seconde sert à itérer l'encryptage pour chaque élément d'une liste, ici les codes ASCII des caractères.Nous les appliquons ici :

```
82          encryptedList = encryptList(messageInt, e, r)
```

Pour finir, il s'agit ensuite de réinterpréter les nouveaux codes ASCII en caractères classiques, grâce à la fonction getChr(), qui va nous servir aussi à "stringifier" la liste :

```
69  def getChr(message):
70      chrList = [chr(x) for x in message]
71      result = "".join(chrList)
72      return result
```

Exemple d'utilisation :

```
$ py exercice1.py
//Voulez-vous générer vos clés? (Y/N) :Y
//Ok ! Voici donc vos clés RSA générées aléatoirement : ( RSA, 569 , 234827 )
//Entrez maintenant le message que vous voulez encrypter : Coucou vous
//Admettons que votre message est [67, 111, 117, 99, 111, 117, 32, 118, 111, 117, 115] , je vais encrypter votre message
//Voici le message encrypté [181912, 3099, 149945, 60513, 3099, 149945, 23179, 128977, 3099, 149945, 184546]
//La clé de decryptage est : 78089
//Voici le message décrypté : [67, 111, 117, 99, 111, 117, 32, 118, 111, 117, 115]
//Et le voici sous forme finale : Coucou vous
```

Exercice 2

On demande de réaliser un programme en Python. Celui-ci permettra l'entrée d'un message à crypter et/ou à décrypter (maximum 80 caractères en utilisant 27 possibilités, les 26 caractères majuscules + le 0). Lors de la démarche de cryptage, faire en sorte que les blancs soient transformés en 0 afin de permettent le cryptage par bloc de huit caractères.

Le second exercice est une application de la méthode de transposition. Il va d'abord falloir transformer notre chaine de caractères en blocs de huit caractères. Pour cela, j'utilise cette fonction :

```
48  def eightBlocks(liste, lines):
49      while len(liste) % 8 != 0:
50          liste += zero
51      return [liste[i:i+ lines] for i in range(0, len(liste), lines)]
```

Celle-ci contrôle donc d'abord si la chaine est multiple de 8 et sinon, va la remplir de zéro jusqu'à l'être. Après, elle va ensuite créer une liste d'autant de caractères que la valeur de lines (dans notre application, ce sera donc 8).

Pour l'encodage même, celui-ci repose sur une clé de 8 valeurs allant de 0 à 8. Cette clé sera parcourue et va transposer chaque caractère à chaque valeur de chaque position de la clé.

Ce n'est pas clair ? Un exemple. Si la première valeur de la clé est 7, nous allons chercher la septième position du message (exemple, "e"). La transposition sera donc de déplacer le "e" à la position de la valeur 7 de la clé, ici 0 car c'est la première valeur. En définitive, le "e" sera déplacée à la position 0 d'une nouvelle chaine qui sera ici celle du message encodé.

Voici le code :

```
 8  def encodeStrings(key, message):
 9      encodedMessage = []
10      for j in range(len(key)):
11          for i in range(len(message)):
12              if int(key[j]) == i:
13                  encodedMessage.append(message[i])
14      return encodedMessage
15
16  def encodeList(key, lists):
17      encodedList = []
18      for i in range(len(lists)):
19          toEncode = lists[i]
20          encoded = encodeStrings(key, toEncode)
21          stringOutput = ''.join(encoded)
22          encodedList.append(stringOutput)
23      stringifiedList = ''.join(encodedList)
24      return stringifiedList
```

Encore une fois ici encodeStrings est l'encodage à proprement parler, et encodeList va l'itérer sur chaque message de la liste d'éléments de 8 caractères que nous avons créé avec eightBlocks().
A noter que encodeList() va aussi s'occuper de trasnformer les listes ainsi créé en chaine de caractère avec .join, ainsi nous pouvons mieux l'afficher en console et surtout retravailler directement dessus pour le décodage.

Concernant le décodage, il va falloir opérer de manière inverse. En parcourant chaque lettre du message encodé, il faudra la déplacer à la position de la valeur de la clé à sa même position. Ainsi, pour "e" qui est maintenant en position [0], il faudra aller voir la valeur de la clé en position [0] également (ici 7) et déplacer la lettre à la bonne position du nouveau message décodé.
Nous arrivons donc à créer ces fonctions :

```
26  def decodeStrings(key, message):
27      decodedMessage = []
28      for j in range(len(key)):
29          for i in range(len(key)):
30              if int(key[i]) == j:
31                  decodedMessage.append(message[i])
32      return decodedMessage
33
34  def decodeList(key, lists):
35      decodedList = []
36      for i in range(len(lists)):
37          toDecode = lists[i]
38          decoded = decodeStrings(key, toDecode)
39          stringOutput = ''.join(decoded)
40          decodedList.append(stringOutput)
41      stringifiedList = ''.join(decodedList)
42      return stringifiedList
```

Voici un exemple d'utilisation :

```
$ py exercice2.py
Sélectionnez votre clé (8 valeurs uniquement, toutes uniques et de 0 à 7): 74201356
Entre le message que vous voulez transposer :  Coucou vous
Votre message en longueurs multiples de 8 est :  Coucou vous00000
Voici le message encodé :  vouCocu 00sou000
Voulez-vous le décoder ? (Y/N)Y
Voici le message décodé :  Coucou vous00000
```

Exercice 3

Veuillez concevoir un programme en Python qui permettra de réaliser le cycle complet d'encryptage / décryptage d'un message et produira des résultats détaillés.

L'exercice 3 est une réutilisation du premier exercice, en ajoutant à cela l'utilisation de décryptage. Nous arrivons donc avec les mêmes méthodes d'encryptage (mais sans les limites de 80 caractères), à un message codé avec des clés privées et publiques générées aléatoirement.
Pour le décodage, il nous faut une nouvelle clé qui sera générée ici :

```
38    def getDecryptKey(randomChoiceP, randomChoiceQ, publicKeyE):
39        z = (randomChoiceP - 1) * (randomChoiceQ - 1)
40        n = 1
41        global d
42        while d != int(d):
43            d = ((z * n) + 1)/ publicKeyE
44            n += 1
45        return d
```

Ceci est une simple application de la formule d'obtention de cette clé de décodage. Cette clé sera ensuite en paramètre pour les fonctions de décodage suivantes :

```
56    def decrypt(messageToDecrypt, publicKeyR, secretKeyD):
57        global decryptedMessage
58        decryptedMessage = ((messageToDecrypt ** secretKeyD) % publicKeyR)
59        return decryptedMessage
60
61    def decryptList(listToDecrypt, publicKeyR, secretKeyD):
62        decryptedList = [decrypt(x, publicKeyR, secretKeyD) for x in listToDecrypt]
63        return decryptedList
```

Avec ceci nous travaillons à nouveau dans une liste de codes ASCII, et ceux-ci doivent forcément redevenir des caractères intelligibles pour l'humain standard :

```
70    def getChr(message):
71        chrList = [chr(x) for x in message]
72        result = "".join(chrList)
73        return result
```

Voici donc un exemple d'utilisation du code avec tout le processus d'encodage et décodage :

```
$ py exercice3.py
//Voulez-vous générer vos clés? (Y/N) :y
//Ok ! Voici donc vos clés RSA générées aléatoirement : ( RSA, 829 , 114139 )
//Entrez maintenant le message que vous voulez encrypter : Coucou vous
//Admettons que votre message est [67, 111, 117, 99, 111, 117, 32, 118, 111, 117, 115] , je vais encrypter votre message
//Voici le message encrypté [87308, 76106, 76448, 29512, 76106, 76448, 109579, 56741, 76106, 76448, 62997]
//La clé de decryptage est : 4645
//Voici le message décrypté : [67, 111, 117, 99, 111, 117, 32, 118, 111, 117, 115]
//Et le voici sous forme finale : Coucou vous
```

Exercice 4

Réaliser un programme en Python permettant la résolution du modèle d'équation du second degré $AX^2 + BX + C = o$. On prendra en compte toutes les possibilités (delta >o, =o, <o).

Nous allons entrer un par un chaque valeur de l'équation. Ainsi nous aurons a, b et c que nous ferons entrer dans une formule qui va d'abord calculer le delta :

```
 9    def calculateDelta(a = float, b = float, c = float):
10        global delta
11        delta = (float(b) * float(b)) - (4 * float(a) * float(c))
12        return delta
```

Une fois ce delta obtenu, le calcul des solutions se fait pour chaque cas (si delta est zéro, ou positif. S'il est négatif, il n'y a rien à calculer car il n'y a pas de solution à l'équation) :

```
14    def deltaZero(a = float, b = float):
15        result = ( - float(b) ) / (2 * float(a) )
16        return result
17
18    def positiveDelta(a = float, b = float, c = float, delta = float):
19        result1 = ( - float(b) + sqrt(delta) ) / ( 2 * a )
20        result2 = ( - float(b) - sqrt(delta) ) / ( 2 * a )
21        return result1, result2
```

Et ainsi, nous ajoutons dans le script l'application des trois cas de figure :

```
37    if (delta < 0):
38        print("Aucune solution n'est possible dans les réels! Le delta est négatif !")
39    elif (delta == 0):
40        print("Le delta est nul, une seule solution est donc possible. Réponse : ",deltaZero(number_a, number_b))
41    elif (delta > 0):
42        print("Le delta est positif, deux solutions sont donc possibles. Réponses : ", positiveDelta(number_a, number_b, number_c, delta))
```

Attention , il est à noter que la fonction modifySign() ne sert qu'à améliorer la visibilité de l'affichage de l'équation du second degré dans la console.
Voici donc un exemple d'utilisation :

```
$ py exercice4.py
Donnez a : -2
Donnez b: -3
Donnez c: 4
L'équation est comme suit :  -2.0 x²  -3.0 x + 4.0 = 0
Delta :  41.0
Le delta est positif, deux solutions sont donc possibles. Réponses :  (-2.350781059358212, 0.8507810593582121)
```

Exercice 5

Ecrire un programme en Python permettant de gérer les calculs sur les polynômes (somme de deux polynômes, produit de deux polynômes, produit d'un polynôme par un coefficient réel). Vous devez pouvoir gérer au moins le 4ème degré.

L'application ici ne se limite pas aux polynômes de degré 4 mais permet une infinité de degrés. Avec createPolynom(), nous allons pouvoir affecter les valeurs du polynôme dans une liste comme ceci :

```
1    def createPolynom(degree = int):
2        polynom = [0]*(int(degree) + 1)
3        for i in range(0,(int(degree) + 1)):
4            print("Votre indice pour X ^",i," :")
5            polynom[i] = input()
6        return polynom
```

Nous allons ensuite afficher le polynome en console de cette manière, avec notamment modifySign() qui nous sert à nouveau ici :

```
24   def polynomView(polynom = list):
25       for i in range(len(polynom)):
26           print("",modifySign(polynom[i]),polynom[i],"X ^",i, end="")
27
28   def modifySign(number = float):
29       if (float(number) >= 0):
30           result = "+"
31       else:
32           result = ""
33       return result
```

La console nous demandera l'opération à effectuer, et sera ensuite entrée en paramètre en même temps que les deux polynômes sur notre fonction principale, operateOnPolynoms :

```
8    def operateOnPolynoms(polynomA = list, polynomB = list, operation = str):
9        polynomC = []
10       if len(polynomA)>len(polynomB): polynomC = [float(0)]*len(polynomA)
11       else: polynomC = [float(0)]*len(polynomB)
12       if operation == "+":
13           for i in range(len(polynomA)): polynomC[i] += polynomA[i]
14           for i in range(len(polynomB)): polynomC[i] += polynomB[i]
15       elif operation == "-":
16           for i in range(len(polynomA)): polynomC[i] += polynomA[i]
17           for i in range(len(polynomB)): polynomC[i] -= polynomB[i]
18       elif operation == "*":
19           for i in range(len(polynomA)): polynomC[i] += polynomA[i]
20           for i in range(len(polynomB)): polynomC[i] = polynomA[i] * polynomB[i]
21       else: print("Vous n'avez pas sélectionné le bon caractère pour effectuer une opération!")
22       return polynomC
```

L'opération mathématique sera donc appliquée à chaque indice de la liste du ou des polynomes. polynomC étant le résultat de l'opération en fonction de la formule à appliquer.

Voici un exemple d'utilisation :

```
$ py exercice5.py
Vous pouvez effectuer des opérations sur les polynomes en choisissant les opérations suivantes : + pour addition, - pour soustraction, * pour multiplication
Il est temps de choisir votre opération, +, - ou *: +
Sélectionner le degré de votre premier polynome : 2
Votre indice pour X ^ 0  :
2
Votre indice pour X ^ 1  :
6
Votre indice pour X ^ 2  :
4
Voici votre premier polynome : [2.0, 6.0, 4.0]
 + 2.0 X ^ 0 + 6.0 X ^ 1 + 4.0 X ^ 2
Sélectionner le degré de votre second polynome : 3
Votre indice pour X ^ 0  :
16
Votre indice pour X ^ 1  :
7
Votre indice pour X ^ 2  :
-2
Votre indice pour X ^ 3  :
6
Voici votre second polynome : [16.0, 7.0, -2.0, 6.0]
Voici le résultat de l'opération : [18.0, 13.0, 2.0, 6.0]
Et le voici sous forme d'équation :
 + 18.0 X ^ 0 + 13.0 X ^ 1 + 2.0 X ^ 2 + 6.0 X ^ 3
```

Exercice 6

Ecrire un programme en Python qui permet d'analyser les attributions dans un diagramme de Venn composé de trois ensembles (voir exercice type fait au cours).

Pour pouvoir afficher un diagramme de Venn, nous allons devoir importer des modules spécifiques :

```
1   from matplotlib import pyplot as plt
2   import numpy as np
3   from matplotlib_venn import venn3, venn3_circles
```

Et nous allons créer un diagramme regroupant des valeurs divisibles par 3, 5 ou 7. Pour cela, nous allons créer des listes prédéfinies (même si la fonction permettant de les créer le permet très facilement) :

```
5   threeDividedList = ['3','105','21','15']
6   fiveDividedList = ['5','105','15','35']
7   sevenDividedList = ['7','105','35','21']
```

Voici, pour l'information, la fonction appendToLists() qui va déterminer si la valeur entrée par un utilisateur peut s'ajouter dans une des trois listes, et le faire si oui :

```
10  def appendToLists(number):
11      if int(number) % 3 == 0:
12          threeDividedList.append(number)
13      if int(number) % 5 == 0:
14          fiveDividedList.append(number)
15      if int(number) % 7 == 0:
16          sevenDividedList.append(number)
17      if int(number) % 3 != 0 and int(number) % 5 != 0 and int(number) % 7 != 0:
18          print("L'entrée",number,"n'est divisible ni par 3, ni par 5, ni par 7 ! Elle ne figurera pas dans le diagramme !")
```

Nous allons ensuite utiliser une fonction permettant de monter son diagramme et de l'afficher :

```
20  def vennConstructor(threeDividedNumbers, fiveDividedNumbers, sevenDividedNumbers):
21      plt.figure(figsize=(8,8))
22      plt.title("Diagramme représentant les chiffres divisibles par 3, 5 ou 7")
23      threeDivided = set(threeDividedNumbers)
24      fiveDivided = set(fiveDividedNumbers)
25      sevenDivided = set(sevenDividedNumbers)
26      labels = venn3([threeDivided, fiveDivided, sevenDivided], ('Divisible par 3', 'Divisible par 5', 'Divisible par 7'))
27      labels.get_label_by_id('001').set_text('\n'.join(sevenDivided-threeDivided-fiveDivided))
28      labels.get_label_by_id('010').set_text('\n'.join(fiveDivided-threeDivided-sevenDivided))
29      labels.get_label_by_id('100').set_text('\n'.join(threeDivided-sevenDivided-fiveDivided))
30      labels.get_label_by_id('101').set_text('\n'.join(threeDivided&sevenDivided-fiveDivided))
31      labels.get_label_by_id('110').set_text('\n'.join(threeDivided&fiveDivided-sevenDivided))
32      labels.get_label_by_id('011').set_text('\n'.join(sevenDivided&fiveDivided-threeDivided))
33      labels.get_label_by_id('111').set_text('\n'.join(threeDivided&fiveDivided&sevenDivided))
34      plt.show()
```

plt.figure va donner les dimensions de la fenêtre qui affichera le diagrame.

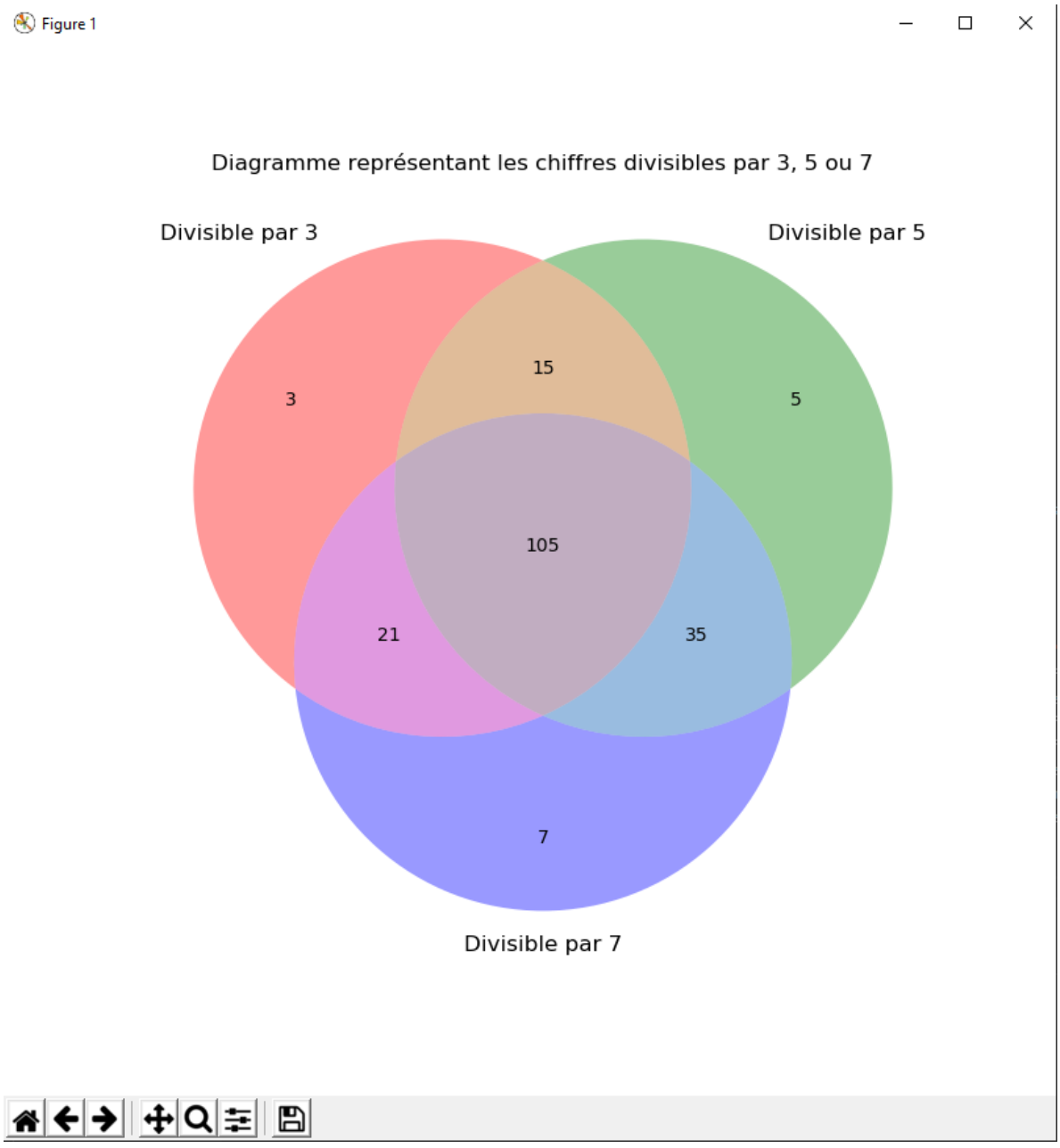
Ce qu'il y a de plus intéressant, c'est la détermination des labels qui seront utilisés pour créer les règles du diagramme. Chaque id ('001', '010', etc) vont correspondre à un des éléments du diagramme, et nous allons définir à quelle règle se lie tel élément du diagramme.
Exemple:

```
32      labels.get_label_by_id('011').set_text('\n'.join(sevenDivided&fiveDivided-threeDivided))
```

Va définir que l'élément 011 du diagramme est celui regroupant les listes divisibles par sept et cinq, mais pas par trois (grâce aux & et – derrière le join()).

Une fois tout ceci bien défini, nous lançons l'affichage du diagramme avec plt.show().

Utilisation :



Exercice 7

Réaliser un programme en Python permettant de structurer une suite de nombres réels sous forme d'arbre binaire en mémoire centrale. Les nombres seront entrés un par un en ordre aléatoire. La relecture de l'arbre binaire devra permettre une sortie en ordre croissant des nombre réels.

Pour la création des arbres binaires, j'utilise l'objet en créant la classe Node, qui va permettre de créer l'arbre. En effet, l'arbre est composé de noeuds (Nodes). Nous allons créer l'arbre de cette manière :

```
2       def __init__(self,data):
3           self.data = data
4           self.left = None
5           self.right = None
6           self.parent = None
```

Chaque noeud doit pouvoir pointer sur un noeud parent, et sur au plus 2 noeuds enfants. data correpond à la racine car il va être affecté à la création de l'objet (exemple, Nodes(10) détermine 10 comme racine, mais également comme nouveau noeud quand on utilisera insert() :

```
11      def insert(self, dataToInsert):
12          if dataToInsert <= self.data:
13              if self.left is None:
14                  self.left = Node(dataToInsert)
15                  self.left.parent = self
16              else:
17                  self.left.insert(dataToInsert)
18          elif dataToInsert > self.data:
19              if self.right is None:
20                  self.right = Node(dataToInsert)
21                  self.right.parent = self
22              else:
23                  self.right.insert(dataToInsert)
```

Il s'agit de la fonction principale de l'exercice, car les règles qui régissent l'arbre s'y trouvent. Si la valeur à insérer est inférieure ou égale à la racine, il est envoyé à gauche du noeud parent. S'il est supérieur, il est envoyé à droite du noeud parent. Chaque nouveau noeud ainsi créé va avoir la valeur insérée, et va pointer vers le noeud parent (avec self.right.parent = self et self.left.parent = self)

Ensuite, pour l'affichage, nous utiliserons la fonction pprint() :

```
25      def pprint(self, level=0):
26          if self.right:
27              self.right.pprint(level + 1)
28          print(f"{' ' * 4 * level}{self.data}")
29          if self.left:
30              self.left.pprint(level + 1)
```

Celle-ci va afficher l'arbre dans un ordre croissant en respectant les règles dudit arbre.

Voici un exemple d'utilisation de l'application :

```
$ py exercice7.py
Veuillez indiquer la valeur de la racine de l'arbre : 10
Veuillez maintenant indiquer le nombre de noeuds dans l'arbre : 8
Veuillez entrer une valeur : 4
Veuillez entrer une valeur : 11
Veuillez entrer une valeur : 13
Veuillez entrer une valeur : 10
Veuillez entrer une valeur : 3
Veuillez entrer une valeur : 7
Veuillez entrer une valeur : 8
Veuillez entrer une valeur : 99
            99
        13
    11
10
        10
                8
            7
    4
        3
```

Ici nous voyons bien que 10 étant la racine, 4 est à sa gauche et 11 à sa droite. Et ainsi de suite, 13 est à droite de 11 et 10 à gauche de 11. 4, qui ne possède qu'une seule branche, aura 3 à sa gauche. Et 3 qui n'a qu'une seule branche, aura 7 à sa droite. Etc… Etc…

Exercice 8

Réaliser un programme en Python permettant de résoudre les systèmes de deux équations à deux inconnues (x et y).

Ici il s'agira de d'abord former les deux équations avec chacun leurs indices :

```
19    a1 = (float(input('Veuillez entrer la valeur le coefficient de X pour la première équation: ')))
20    b1 = (float(input('Veuillez entrer la valeur le coefficient de Y pour la première équation: ')))
21    c1 = (float(input('Veuillez entrer la valeur de l\'égalité de la première équation : ')))
22    a2 = (float(input('Veuillez entrer la valeur le coefficient de X pour la seconde équation: ')))
23    b2 = (float(input('Veuillez entrer la valeur le coefficient de Y pour la seconde équation: ')))
24    c2 = (float(input('Veuillez entrer la valeur de l\'égalité de la seconde équation : ')))
```

Une fois ceci fait, nous allons tous les passer en paramètre sur la fonction reprenant la formule de résolution :

```
3     def equationsResolve(a1, b1, c1, a2, b2, c2):
4         A = np.matrix([[a1, b1], [a2, b2]])
5         B = np.matrix([[c1], [c2]])
6         solution = A.I * B
7         return solution
```

Pour cela nous utilisons les matrices du module numpy qui vont nous permettre de résoudre très rapidement les équations.

La solution sera également sortie en matrice, et devra être affichée de cette manière :

```
30    solution = equationsResolve(a1, b1, c1, a2, b2, c2)
31    print('X et Y ont été trouvées, les voici : ')
32    print('X = {}'.format(solution[0,0]))
33    print('Y = {}'.format(solution[1,0]))
```

Exemple d'utilisation :

```
$ py exercice8.py
Bonjour, nous allons résoudre des équations à deux inconnues
Veuillez entrer la valeur le coefficient de X pour la première équation: 13
Veuillez entrer la valeur le coefficient de Y pour la première équation: -2
Veuillez entrer la valeur de l'égalité de la première équation : 4
Veuillez entrer la valeur le coefficient de X pour la seconde équation: 7
Veuillez entrer la valeur le coefficient de Y pour la seconde équation: 77
Veuillez entrer la valeur de l'égalité de la seconde équation : 777
Voici la première équation : 13.0 X  -2.0 Y =  4.0
Voici la seconde équation : 7.0 X + 77.0 Y =  777.0
X et Y ont été trouvées, les voici :
X = 1.8344827586206898
Y = 9.924137931034483
```

Exercice 9

Réaliser un programme en Python permettant de résoudre les systèmes d'équations à trois inconnues (x, y et z).

```
3   def equationsResolve(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3):
4       A = np.matrix([ [a1, b1, c1],
5                       [a2, b2, c2],
6                       [a3, b3, c3]])
7       B = np.matrix([[d1], [d2], [d3]])
8       solution = A.I * B
9       return solution
```

L'exercice 9 a la particularité de fonctionner de la même manière que le 8, à ceci prêt que les matrices de numpy seront plus complexes à préparer :
Mais autrement, il ne s'agit que d'une redite du fonctionnement du 8 ...

```
21  a1 = (float(input('Veuillez entrer la valeur le coefficient de X pour la première équation: ')))
22  b1 = (float(input('Veuillez entrer la valeur le coefficient de Y pour la première équation: ')))
23  c1 = (float(input('Veuillez entrer la valeur le coefficient de Z pour la première équation: ')))
24  d1 = (float(input('Veuillez entrer la valeur de l\'égalité de la première équation : ')))
25  a2 = (float(input('Veuillez entrer la valeur le coefficient de X pour la seconde équation: ')))
26  b2 = (float(input('Veuillez entrer la valeur le coefficient de Y pour la seconde équation: ')))
27  c2 = (float(input('Veuillez entrer la valeur le coefficient de Z pour la seconde équation: ')))
28  d2 = (float(input('Veuillez entrer la valeur de l\'égalité de la seconde équation : ')))
29  a3 = (float(input('Veuillez entrer la valeur le coefficient de X pour la troisième équation: ')))
30  b3 = (float(input('Veuillez entrer la valeur le coefficient de Y pour la troisième équation: ')))
31  c3 = (float(input('Veuillez entrer la valeur le coefficient de Z pour la troisième équation: ')))
32  d3 = (float(input('Veuillez entrer la valeur de l\'égalité de la troisième équation : ')))
41   print('X, Y et Z ont été trouvées, les voici : ')
42   print('X = {}'.format(solution[0,0]))
43   print('Y = {}'.format(solution[1,0]))
44   print('Z = {}'.format(solution[2,0]))
```

Utilisation:

```
$ py exercice9.py
Bonjour, nous allons résoudre des équations à trois inconnues
Veuillez entrer la valeur le coefficient de X pour la première équation: 1
Veuillez entrer la valeur le coefficient de Y pour la première équation: 2
Veuillez entrer la valeur le coefficient de Z pour la première équation: 3
Veuillez entrer la valeur de l'égalité de la première équation : 4
Veuillez entrer la valeur le coefficient de X pour la seconde équation: -5
Veuillez entrer la valeur le coefficient de Y pour la seconde équation: 10
Veuillez entrer la valeur le coefficient de Z pour la seconde équation: -15
Veuillez entrer la valeur de l'égalité de la seconde équation : 20
Veuillez entrer la valeur le coefficient de X pour la troisième équation: -2
Veuillez entrer la valeur le coefficient de Y pour la troisième équation: 4
Veuillez entrer la valeur le coefficient de Z pour la troisième équation: -8
Veuillez entrer la valeur de l'égalité de la troisième équation : 16
Voici la première équation : 1.0 X + 2.0 Y + 3.0 X =  4.0
Voici la seconde équation : -5.0 X + 10.0 Y  -15.0 X =  20.0
Voici la troisème équation : -2.0 X + 4.0 Y  -8.0 X =  16.0
X, Y et Z ont été trouvées, les voici :
X = 11.999999999999998
Y = 2.0
Z = -4.0
```

Exercice 10

Réaliser un programme en Python permettant de réaliser les opérations de base sur des matrices de nombres réels (maximum 4X4) = addition de deux matrices, multiplication de deux matrices, multiplication d'une matrice par un scalaire, transposition d'une matrice.

Pour la création de matrices, nous allons encore avoir besoin du module numpy.

```
5   def createMatrix(lines, columns):
6       a = np.arange(lines * columns).reshape(lines , columns)
7       i = 0
8       while i < lines:
9           j = 0
10          while j < col:
11              print("colonne :",j+1," et ligne : ",i+1)
12              b = int(input("entrez la valeur : "))
13              a[i][j] = b
14              j += 1
15          i += 1
16      return a
```

Ici il s'agira de parcourir chaque colonne et chaque ligne et entrer les valeurs en fonction du nombre de colonnes et lignes que l'on a choisi. Ici j'ai fait le choix de ne pas me limiter à des matrices de maximum 4x4, car en soi le code fonctionnerait en tous les cas. Mais s'il le fallait, alors il suffirait de contrôler les valeurs de lines et columns entrées et empêcher qu'elles soient > 4.

Ensuite, une fois les deux matrices créées, nous pouvons enfin faire nos opérations. La fonction operationMatrix va donc d'abord contrôler l'opération, puis utiliser les paramètres correspondant à ses besoins.

```
24  def operationMatrix(operation, matrix1, matrix2 = 1, scalar = 1):
25      print(matrix1)
26      if(operation == 1):
27          print("+")
28          result = matrix1 + matrix2
29          print(matrix2)
30      elif(operation == 2):
31          print("*")
32          result = matrix1 * matrix2
33          print(matrix2)
34      elif(operation == 3):
35          print("*")
36          result = matrix1 * scalar
37          print(scalar)
38      elif(operation == 4):
39          result = np.transpose(matrix1)
40          print("Voici votre matrice :")
41          print(matrix1)
42          print("Voici sa transposée :")
43      if (1 <= operation <= 3):
44          print("=")
45      print(result)
```

Il est à noter que l'on détemine un paramètre comme facultatif en lui passant une valeur par défaut. Egalement, pour la transposition, nous utilisons la méthode np.transpose() déjà fournie par numpy.

Concernant la petite fonction printMatrixNumber(), il s'agit simplement de pouvoir adapter le texte de la console en fonction du choix, s'il faut créer une seconde matrice ou non. Cela permet de rendre l'appli plus agréable à suivre.

```
18  def printMatrixNumber(operation):
19      if (operation < 3):
20          print("Nous allons créer notre première matrice.")
21      else:
22          print("Nous allons créer notre matrice.")
```

Exemple d'utilisation :

```
$ py exercice10.py
1. Addition de deux matrices
2. Multiplication de deux matrices
3. Multiplication d'une matrice par un scalaire
4. Transposition d'une matrice
Veuillez choisir votre opération : 3
operation =  3
Nous allons créer notre matrice.
Entre le nombre de lignes :2
Entre le nombre de colonnes :2
colonne : 1  et ligne :  1
entrez la valeur : 1
colonne : 2  et ligne :  1
entrez la valeur : 2
colonne : 1  et ligne :  2
entrez la valeur : 3
colonne : 2  et ligne :  2
entrez la valeur : 4
Veuillez entrer une valeur pour le scalaire :15
[[1 2]
 [3 4]]
*
15
=
[[15 30]
 [45 60]]
```