1 Données de type composite

1.1 Les chaînes de caractères, type string `str`

Définition 1

- Une donnée (variable ou expression) de **type composite** est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples.
- Une **chaîne de caractères** est une donnée composite de type **string** `str` qui est une suite d'entités (ou items) plus simples, les caractères.
- En Python une chaîne de caractères est délimitée par des simples ou doubles quotes (voir des triples quotes). On peut la considérer comme une séquence ordonnée de variables qui contiennent chacune un caractère. Les caractères sont indexés à partir de 0.
- Si on crée une variable de type `str` avec l'affectation `chaine = 'Hello'` , on obtient sa longueur avec `len(chaine)` et on accède au caractère d'indice `i` par son nom qui est `chaine[i]`.
- Le type `str` est un type de variable non mutable, c'est pourquoi on ne peut modifier le premier caractère avec `chaine[0]='V'` par exemple.

```
>>> chaine = 'ISN'
>>> id(chaine)
3069607200
>>> for i in range(len(chaine)): #les caractères ont chacun un identifiant mémoire
...     print(chaine[i],':',id(chaine[i]))
...
I : 3070202912
S : 3070740608
N : 3070739648
>>> for c in chaine: #une variable de type str est un objet itérable
...     print(c,':',id(c))
...
I : 3070202912
S : 3070740608
N : 3070739648
>>> chaine = chaine+' au lycée du Parc en 2013'
>>> chaine, id(chaine)
('ISN au lycée du Parc en 2013', 3069620736)
>>> chaine = chaine+' au lycée du Parc en 2013'
>>> chaine, id(chaine)
('ISN au lycée du Parc en 2013', 3069620736)
```

Remarque 1

En Python toutes les données manipulées sont des objets au sens de la programmation orientée objet. Les objets sont des entités regroupant :

- *des données appelées attributs de l'objet ;*
- *des fonctions appelées méthodes de l'objet, ce sont les seules qui sont habilitées à manipuler les données d'un objet.*

Par exemple, si on déclare la variable `a = 'Hello'`*, le nom* `a` *pointe vers la référence mémoire d'un objet de type* `str`*.*

```
>>> a = 'Hello'
>>> id(a),type(a)
(45002496, <class 'str'>)
```

type string

Un objet est une instance (ou réalisation) d'une classe (ou modèle) qui décrit tous les attributs et toutes les méthodes communes aux objets de la classe. Pour les objets de type `str`*, il existe plusieurs méthodes particulières pour les manipuler. Pour applique une méthode à un objet on utilise le point :* `objet.methode()`*.*

Exemple 1

Exécuter le code suivant pour découvrir les opérations de manipulation d'une chaîne de caractères

```
>>> c = ' J\'aime l\'ISN '; b = "c'est \n très bien";print(c);print(b) #\ sert de caractère
    d'échappement ou pour insérer un saut à la ligne
>>>d = c+','+b; print(d) #On peut concaténer des cha\^ines
>>>len(b) #Longueur d'une cha\^ine
>>>b[0];b[len(b)];b[len(b)-1] #accès à un caractère
>>>b[1:2];b[0:2];b[:2];b[:len(b)];b[-1:len(b)];b[-1:];b[:len(b):2] #Découpage en tranches (
    slicing)
>>> for i in b: #Parcourir une chaine avec un itérateur
        print(i)
>>> dir(str) #Méthodes et attributs d'un objet de type str
>>> c.upper() #méthode pour mettre tous les caractères en capitales, l'application d'une
    méthode à une chaine crée un nouvel objet
>>> c = c.upper(); c #Pour modifier la variable c il faut la réaffecter
>>> c.index('J');b.index('J');c.strip();c.upper() #d'autres méthodes
>>> for j in 'ABC_12@Parc.fr': #code UNICODE d'un caractère
        print(ord(j))
>>>for i in range(600,610): #Réciproquement
      print(chr(i))
```

type string

Méthode *Fonctions et méthodes pour les variables de type str*

Soit `chaine` une variable associée à un objet de type `str`. Un objet de type `str` est non mutable, donc ses méthodes retournent forcément un nouvel objet du même type.

Commande	**Rôle**
`len(chaine)`	longueur de chaine
`chaine[i]`	caractère d'indice i dans chaine
`chaine[i:j]`	sous-chaine entre les positions i inclue et j exclue
`str(objet)`	convertit objet en `chaine` de type str
`eval(chaine)`	évalue `chaine` comme un code Python
`c in chaine`	retourne un booleen, teste si `c` dans `chaine`
`chaine.split()`	retourne une liste des éléments de `chaine` séparés par des espaces
`chaine.count('a')`	retourne le nombre de 'a' dans `chaine`
`chaine.replace('a','b')`	remplace 'a' par 'b' dans `chaine`
`chaine.find('a')`	retourne la position de 'a' dans `chaine` et −1 sinon
`chaine.rstrip('\r\n')`	supprime en fin de ligne les caractères '\r\n' ou les espaces par défaut
`chaine.lstrip()`	supprime en début de ligne le caractère passé en paramètre ou les espaces par défaut

Les caractères non imprimables `'\r\n'` (Carriage Return et Feed Line) codent les fins de lignes sous Windows alors que sous les systèmes UNIX elles sont codées par un simple `'\n'`.
Pour les différences de codage de fins de ligne selon les systèmes d'exploitation, on pourra consulter la page `http://fr.wikipedia.org/wiki/Fin_de_ligne`.

En pratique, on a souvent besoin de **formater** une chaîne de caractères pour y insérer le contenu d'une ou plusieurs variables.
En Python la méthode classique utilise **l'opérateur modulo** % mais en Python3 on peut aussi utiliser la méthode `format`.
Pour `format` on consultera la documentation `http://docs.python.org/3.3/library/string.html#formatspec`.
Ci-dessous, on donne quelques exemples de formatages avec l'opérateur modulo.
La syntaxe par défaut pour insérer les valeurs de deux variables `var1` et `var2` dans une chaîne est :

```
'bla %s bla %s bla'%(var1,var2)
```

Dans la chaîne formatée, %s insère l'affichage par défaut pour tous les types de variables (celui de `print`). %d permet d'afficher un entier (avec des directives supplémentaires pour régler la taille minimale ou l'affichage du signe) et %f permet d'afficher un flottant (réglage de la taille, du signe et de la précision). :

```
>>> a = 'python'
>>> 'ceci est un %s'%a
'ceci est un python'
>>> 'ceci est un %s'%'a'
'ceci est un a'
>>> b = 3
>>> 'Un %s de %s m de long'%(a,b)
'Un python de 3 m de long'
>>> '%d est un entier'%3
'3 est un entier'
>>> '%d n\'est pas un entier'%'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %d format: a number is required, not str
>>> '%+d est une température positive'%17
'+17 est une température positive'
>>> import math
>>> '%1.2f est un flottant avec une précision de %f'%(math.pi,0.01)
'3.14 est un flottant avec une précision de 0.010000'
>>> '%1.2f est un flottant avec une précision de %.2f'%(math.pi,0.01)
'3.14 est un flottant avec une précision de 0.01'
```

formatage de chaine

Il faut aussi savoir que les caractères sont des chaînes particulières de longueur 1 et comme l'ordinateur ne manipule que des nombres, une première abstraction est d'associer à chaque caractère un code numérique. En première approche, pour les caractères non accentués de l'alphabet romain, les entiers et les symboles de ponctuation classiques, on peut considérer qu'il s'agir du code ASCII.
Le code ASCII d'un caractère s'obtient alors avec la primitive `ord()` et réciproquement le caractère de code ASCII donné s'obtient avec la primitive `chr()`.
Pour obtenir la table de correspondance du code ASCII :

https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

```
>>> for i in range(97,123):
...     print(chr(i),end='')
...
abcdefghijklmnopqrstuvwxyz
>>> for i in 'abcdefghijklmnopqrstuvwxyz':
...     print(ord(i),end=',')
...
97,98,99,100,101,102,103,104,105,106,107,108,109,
110,111,112,113,114,115,116,117,118,119,120,121,122,
```

codes ASCII

Exercice 1

1. Ecrire un programme qui prend en entrée une chaîne de caractères et qui affiche en sortie son nombre d'espaces.
2. Écrire un programme qui lit au clavier une chaîne de caractères et compte les voyelles.
3. Ecrire un programme Python qui demande un mot à l'utilisateur et qui l'écrit en doublant toutes les voyelles

4. Ecrire un programme qui détermine si un caractère donné appartient à une chaîne de caractères donnée (Reprogrammation de l'opérateur `in`, algorithme de recherche séquentielle, voir aussi exo 7 3.).

5. Ecrire un programme qui détermine la (ou les) lettre(s) de l'alphabet la (les) plus fréquente(s) dans une chaîne de caractères. On pourra commencer par le bout de code :

```
chaine = input('Entrez une cha\^ine : ')
chaine = chaine.lower() #on convertit tout en minuscules
print(chaine)
lettremax, freqmax = '',0
for i in range(ord('a'),ord('z')+1): #Parcours des lettres de 'a' à 'z'
    lettre = chr(i) #caractère ASCII associé à i
```

6. Ecrire un programme Python qui demande un mot à l'utilisateur puis qui l'écrit à l'envers.

7. Ecrire un programme Python qui demande un mot à l'utilisateur et qui détermine si c'est un palindrome.

8. En utilisant la fonction `randint` du module `random` et la primitive `chr()`, écrire un programme qui génère une chaîne de longueur choisie, constituée de caractères tirés au hasard parmi les lettres romaines majuscules.

Exercice 2

Parfois, on a besoin de manipuler le temps. Le module `time` propose quelques outils, mais il ne permet pas de manipuler des dates avant l'EPOCH UNIX (01/01/1970) au au-delà de 2038. D'autres modules comme `datetime` ou `calendar` sont plus appropriés pour la gestion de calendriers. La documentation du module `time` est disponible à l'adresse suivante :

`http://docs.python.org/3.3/library/time.html`

```
>>> from time import*
>>> time() #temps en secondes écoulés depuis l'EPOCH UNIX (01/01/1970)
1372408311.689293
>>> localtime() #structure de temps suivant les règles locales, heure d'été compris
time.struct_time(tm_year=2013, tm_mon=6, tm_mday=28, tm_hour=10, tm_min=32, tm_sec=0,
      tm_wday=4, tm_yday=179, tm_isdst=1)
>>> print(localtime().tm_year)
2013
>>> strftime("Aujourd'hui nous sommes le %d/%m/%Y",localtime()) #formatage de chaine avec
      une structure de temps
"Aujourd'hui nous sommes le 28/06/2013"
>>> strptime('28/06/2013','%d/%m/%Y') #réciproque de la précédente fonction
time.struct_time(tm_year=2013, tm_mon=6, tm_mday=28, tm_hour=0, tm_min=0, tm_sec=0, tm_wday
      =4, tm_yday=179, tm_isdst=-1)
>>> time(),sleep(2),time() #sleep(n) permet de faire une pause de n secondes
(1372409358.127493, None, 1372409360.129556)
```

1. Ecrire un programme qui prend en entrée un entier n, déclenche un chronomètre, affiche toutes les secondes le nombre de secondes écoulées depuis le début puis qui affiche "C'est terminé" au bout de n secondes.

2. Ecrire un programme qui retourne le nombre de jours qu'il reste avant le prochain Noel ou "Noel est déjà passé".

3. Ecrire un programme qui retourne le nombre de jours restant avant le prochain vendredi 13.

4. Ecrire un programme qui retourne le nombre de jours écoulés depuis la date de naissance de l'utilisateur et le jour de la semaine où il est né (à condition que l'utilisateur soit né après l'EPOCH UNIX).

1.2 Les listes, type `list`

Définition 2

Un **tableau** est une structure de données composite constituée d'une séquence ordonnée d'éléments qui peuvent être des entiers, des flottants, des chaînes de caractère, des booleens ...Dans certains langages (Java ...), tous les éléments du tableau doivent être de même type et il faut déclarer la taille du tableau avant de l'utiliser pour réserver la place en mémoire.
En Python, il n'existe pas de type tableau mais un type `list`. Comme un tableau, une **liste** Python est une **séquence ordonnée** d'objets mais qui peuvent être de types différents. De plus les listes sont des objets Python modifiables, en particulier leur taille peut évoluer.
Dernier point, il est recommandé, en Python, de ne pas utiliser les caractères 'l', 'T' ou 'O' pour nommer des variables de type liste car dans certaines polices ils sont indiscernables de certains caractères numériques.

Exemple 2 *Liste en Python et gestion de la mémoire*

- Dans le code ci-dessous on déclare une liste L contenant dans l'ordre un entier `12`, un flottant `3.14`, une chaîne de caractère `'ISN'` et un booleen `True`.
- La longueur de la liste est `len(L)=4`.
- Les éléments sont indexés de 0 à `len(L)-1` et on accède à l'élément d'indice `i` avec `L[i]`.
- Comme les objets de type `string` ou `tuple`, les objets de type `list` sont itérables et on peut les parcourir avec une boucle `for` et le mot clef `in`.
- En affichant les identifiants mémoires de `L` et de ses éléments `L[i]` on observe qu'ils sont distincts. Le nom `L` ne pointe pas directement vers l'objet liste mais vers la référence mémoire de cet objet. Dans d'autres langages (C), on appelle **pointeurs** les variables dont la valeur est une référence mémoire qui pointe vers un certain type d'objet composite. Ainsi lorsqu'on affecte la valeur d'un pointeur à une autre variable, on copie juste une référence mémoire.
- Une liste est un objet **mutable** (contrairement à une chaîne de caractères).

  Pour modifier l'élément d'indice `i` on écrit `L[i] = valeur`.

  Ainsi l'instruction `K = L` affecte à la liste K, la valeur de `L` qui est la référence mémoire pointant vers le contenu de la liste `L`. `K` est donc juste un alias de `L`.

  Si on modifie `K` ou `L`, l'autre liste est aussi modifiée.
- Pour copier les éléments de la liste L dans la liste `M`, on peut écrire `M = L[:]`. L'identifiant mémoire de `M` est bien différent de celui de `L` et les éléments de `M` reçoivent une copie de la valeur des éléments de `L` de même indice. *Attention, si par exemple l'élément de `L[2]` est un pointeur, comme une liste, la valeur copiée sera une référence mémoire c'est pourquoi `M[2]` pointera vers le même objet liste et aura le même identifiant mémoire. Dans ce cas, pour réaliser une vraie copie, on utilisera la fonction `deepcopy()` du module `copy`.*
- On peut créer une liste vide avec le constructeur `list()` ou avec `[]`.
- On peut définir des **listes par compréhension** avec une syntaxe du type :

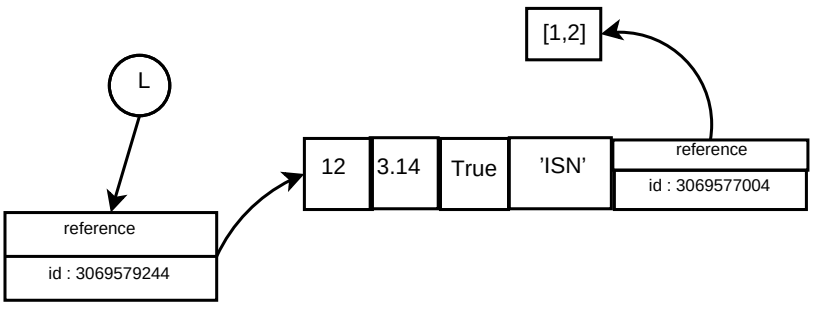
  `liste = [ expression for variable in iterateur if test(variable)]`

  telle que `pair = [ i for i in range(5) if i%2==0]`.

```
>>> L = [12,3.14,'ISN',True,[1,2]] #liste d'objets de types hétérogènes
>>> id(L)
3069579244
>>> len(L),L[0],L[len(L)-1],L[-1] #L[-1] pour accéder au dernier élément
(5, 12, [1, 2],[1,2])
>>> for i in L:
...     print(i,' d\'identifiant',id(i))
...
12  d'identifiant 137396176
3.14  d'identifiant 158385012
ISN  d'identifiant 3069357344
```

```
True d'identifiant 137231552
[1, 2] d'identifiant 3069577004

>>> L[0]=13;L[4][0]=3 #différence entre élément mutable (int) et élément non mutable (list)
>>> id(L[0]),id(L[4])
(137396192, 3069577004)
>>> K = L
>>> id(K),id(L) #K et L sont des alias de la même liste
(3069579244, 3069579244)
>>> M = L[:] #copie superficielle de la liste L
>>> M,id(M)
([13, 3.14, 'ISN', True, [3, 2]], 3069565548)
>>> M[2]='NSA'
>>> for i in M:
...     print(i,' d\'identifiant',id(i))
...
13  d'identifiant 137396192
3.14  d'identifiant 158385012
NSA  d'identifiant 3069357504
True  d'identifiant 137231552
[3, 2] d'identifiant 3069577004
>>> K[3] = False #effet de bord après modification de K alias de L
>>> L,K
([13, 3.14, 'ISN', False, [3, 2]], [13, 3.14, 'ISN', False, [3, 2]])
>>> M[3] = True #pas d'effet de bord pour les éléments non mutables avec M copie de L
>>> L,M
([13, 3.14, 'ISN', False, [3, 2]], [13, 3.14, 'NSA', True, [3, 2]])
>>> M[4][0]=5 #l'effet de bord demeure avec les éléments qui sont des pointeurs comme des
      listes
>>> L,M
([13, 3.14, 'ISN', False, [5, 2]], [13, 3.14, 'NSA', True, [5, 2]])
>>> lvide1 = list() ; lvide2 = []
>>> lvide1, lvide2, lvide1 is lvide2 #Deux listes vides différentes
([], [], False)
>>> carre = [i**2 for i in range(10)] #définition d'une liste en compréhension
>>> carre
[0, 1, 4, 9, 16]
```

Listes en Python



Exercice 0 *inspiré d'un TP de*[1] *Stéphane Gonnord*

Sans utiliser l'interpréteur Python, deviner les valeurs des listes `k,t,m` après exécution du script ci-contre.
Aller sur la page `http://pythontutor.com/visualize.html`, saisir le code et visualiser pas-à-pas l'exécution.

```
k = [10,15,[0,1]]
t = k
m = t[:]
m[1] = 17
t[0] = 19
m[2][0] = 2
```

Méthode *Manipulations de listes*

Exécuter le code suivant pour découvrir quelques techniques de génération de listes et des opérateurs et méthodes opérant sur les listes.

```
>>> L = [0,1,2,3,4,5]
>>> 6 in L #renvoie True si 6 dans L, False sinon
>>> M = [6,7,8,9,10]
>>> N = L+M #On peut concaténer des listes
>>> dir(list) #Attibuts et méthodes des objets de type list
>>> id(N); N.append(11) ; N ; id(N) #La méthode append ajoute un élément en fin de liste
#A l'inverse des objets de type int,float, bool ou str les objets de type list sont
      mutables, l'application d'une méthode à une liste change l'objet sur place
>>> N.extend([12,13]) #ajoute en fin de liste les éléments de [12,13]
>>> N.reverse() #renverse la liste
>>> P = list(range(10,21)) #Générer une liste d'entiers successifs
>>> Q = [] #avant de remplir une liste, créer une liste vide
>>> for j in range(20,31):
        Q.append(j) #Génération d'une liste avec une boucle
>>> R = [i for i in range(30,41)] #Sans boucle avec une liste par compréhension
>>> S = [j for j in range(40,51) if j%2 == 0] #On peut filtrer les valeurs avec un test
>>> N[0:5],N[0:5:1],N[:5] #Découpage en tranches
([0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4])
>>> N[1:5], N[1:5:2], N[1:2], N[7:len(N)], N[7:]
>>> N[-1:], N[-2:], N[-len(N):]
>>>del N[len(N)-1] #Suppression du dernier élément de N
>>> U = [10,6,7,9]; U.sort() #modifie la liste en la triant
>>> T = [1,'a',True,'a']
>>> T.index('a') #index de la première occurence de 'a' dans T
>>>T.count('a') #compte les occurences de 'a' dans T
>>> T.remove('a') #supprime la première occurence de l'élément 'a' dans T
```

Exercice 3

1. Écrire un programme qui trouve l'élément maximal dans un tableau d'entiers.
2. Écrire un programme qui compte le nombre d'éléments supérieurs à 15 dans un tableau d'entiers.
3. On se donne un tableau d'entiers. Écrire un programme qui range les éléments de ce tableau dans un autre tableau, en mettant les éléments impairs à gauche et les éléments pairs à droite. Même exercice en utilisant un seul tableau.
4. Le programme ci-après recherche la première occurence d'un entier dans un tableau. Expliquer son fonctionnement. Ecrire un programme similaire qui recherche la première occurence d'une lettre dans une chaîne de caractères.
5. Ecrire un programme qui prend en entrée deux tableaux d'entiers et qui retourne un message d'erreur s'ils ne sont pas de même longueur ou qui détermine s'ils contiennent les mêmes éléments dans l'ordre.

1. `http://www.mp933.fr/`

```
from random import randint

liste = [randint(0,150) for i in range(100)]
n = int(input('Entrez un entier entre 0 et 150 : '))
i = 0
while i<len(liste) and liste[i] != n:
    i += 1
if i == len(liste):
    print('%s n\'est pas dans la liste'%n)
else:
    print('%s apparait en position %i dans la liste'%(n,i))
```

Exercice 4 *Listes et tableaux bidimensionnels*

Considérons un élève qui a 4 notes par trimestre. On peut représenter ses notes sur 3 trimestres dans un tableau de 3 lignes et 4 colonnes. Une autre représentation peut être une matrice M à 3 lignes et 4 colonnes dont l'élément en ligne 2 et colonne 3 est $M_{23} = 12$.
En Python, on représentera un tableau bidimensionnel ou une matrice sous la forme d'une liste de listes.

Tableau

T1	12	9	13	10
T2	10	14	12	8
T3	15	16	17	7

Matrice

$$M = \begin{pmatrix} 12 & 9 & 13 & 10 \\ 10 & 14 & 12 & 8 \\ 15 & 16 & 17 & 7 \end{pmatrix}$$

Liste en Python

```
M = [[12, 9, 13,10], [10, 14, 12,8], [15, 16, 17,7]]
```

1. Pour accéder la note $M_{23} = 12$ l'instruction Python est $M[2][3]$, quelle est la valeur stockée en $M[3][2]$?

   Quel est le type de la valeur $M[0]$? et celui de $M[2][2]$?

2. Le programme ci-dessous affiche les coefficients de M ligne par ligne, en commençant par le haut et en parcourant chaque ligne de gauche à droite :

   ```
   for i in range(len(M)):
      for j in range(len(M[i])):
         print(M[i][j])
   ```

   Modifier ce programme pour afficher les coefficients de M colonne par colonne, en commençant par la gauche et en parcourant chaque colonne de haut en bas.

3. Ecrire un programme qui calculer la moyenne annuelle de l'élève.

4. Ecrire un programme qui diminue toutes les notes de l'élève stockée dans la matrice M de 8%.

5. Ecrire un programme qui réinitialise la matrice 3×4 du carnet de notes en affectant l'objet `None` à tous ses éléments.

6. Ecrire une fonction `zeros(n,m)` qui retourne une matrice de `n` lignes et `m` colonnes remplie de zéros. On pourra s'inspirer du code suivant et commenter la différence entre les deux méthodes proposées :

   ```
   >>> N = [[0,0,0]]*4
   >>> N
   [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
   >>> N[1][1] = 2
   >>> N
   [[0, 2, 0], [0, 2, 0], [0, 2, 0], [0, 2, 0]]
   >>> N = [[0,0,0] for i in range(4)]
   >>> N
   [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
   >>> N[1][1]=2
   >>> N
   [[0, 0, 0], [0, 2, 0], [0, 0, 0], [0, 0, 0]]
   ```

1.3 Manipulation de fichiers

Méthode

En Python, l'accès à un fichier se fait par l'intermédiaire d'un descripteur de fichier créé à l'aide de la primitive `open(nom,mode)`. Une fois que les manipulations sont terminées, il faut bien penser à fermer le descripteur de fichier avec `f.close()`. C'est une syntaxe classique dans la plupart des langages, Python propose une autre syntaxe qui délimite les manipulations de fichier par l'indentation avec un bloc introduit par le mot clef `with`.

```
f = open('fichier.txt','w')
f.write('Hello')
f.close()
```

syntaxe classique

```
with open('fichier.txt','w') as f:
    f.write('Hello')
```

syntaxe idiomatique

Par défaut les fichiers sont ouverts en mode texte 't' mais on peut aussi ouvrir des fichiers binaires en rajoutant le suffixe `'b'` aux modes de lecture `'r'` ou d'écriture `'w'`. Attention, pour l'ouverture en mode écriture, les modes `'w'` ou `'a'` créent le fichier s'il n'existe pas mais le mode `'w'` écrase le fichier s'il existe déjà. On peut rajouter un '+' (mode `'w+'`, `'a+'`, `'r+'`), qui permet l'accès à la fois aux fonctions d'écriture (`write`) et de lecture (`read`), mais le mode `'w+'` écrase toujours le fichier existant.
Mais pour bien manipuler un fichier texte, il faut d'abord connaître la façon dont il est structuré !!!

Fonctions	Rôle
`f = open('nom_fichier.txt','w')`	accès en écriture avec création d'un nouveau fichier
`f = open('nom_fichier.txt','r')`	accès à un fichier existant en mode lecture
`f = open('nom_fichier.txt','a')`	accès en écriture à un fichier existant en mode ajout
`f.write('texte')`	ajout de 'texte' dans le fichier
`f.writelines(liste)`	ajout d'une liste de lignes dans un fichier
`f.read()`	lecture de tout le fichier
`f.read(8)`	lecture des 8 premiers caractères du fichier
`f.readline()`	lecture de la ligne courante du fichier
`f.readlines()`	lecture de toutes les lignes stockées dans une liste
`f.tell()`	position courante du curseur
`f.seek(16)`	place le curseur sur le caractère en position 16
`for i in f`	itération sur les lignes du fichier
`f.close()`	fermeture du fichier

Exercice 5

Ouvrir le fichier `jeu.txt`, qui contient sur trois lignes successives : un nom de personnage, son niveau, son nombre de vies.
Ecrire un programme qui ouvre ce fichier en mode lecture puis qui affiche le message « Bonjour, vous êtes Personnage, vous êtes au niveau x et vous avez y vies. »

Exemple 3

On va manipuler un fichier texte `temperature.txt` dont chaque ligne contient un nom de ville, une tabulation, une température puis un retour à la ligne c'est-à-dire que chaque ligne est structurée ainsi : `'Ville\tTemperature\n'`.
Tous les scripts Python de manipulation de fichier doivent être écrits dans le même répertoire que le fichier, à moins d'indiquer un chemin absolu ou relatif pour le fichier. Pour faire des tests en mode console on peut changer le répertoire courant avec la commande `chdir` du module `os` :

```
>>> import os
>>> os.getcwd()
'/home/fjunier'
>>> os.chdir('/home/fjunier/Bureau/ISN/Cours/CoursJunier/Programmation2/Fichiers')
>>> os.getcwd()
```

```
'/home/fjunier/ISN/Cours/CoursJunier/Programmation2/Fichiers'
```

changement du répertoire courant

- Pour lire le fichier , on peut soit récupérer tout le contenu dans une chaîne de caractères ou le parcourir ligne à ligne :

```
f = open('temperature.txt','r')
lecture = f.read()
print(lecture)
f.close()
```

lecture de tout le fichier

```
f = open('temperature.txt','r')
liste_lignes = f.readlines()
print(liste_lignes)
for ligne in liste_lignes:
    liste = ligne.rstrip('\n').split('\t')
    print('A %s, la température est de %s '%(liste[0],liste[1]))
f.close()
```

lecture par ligne avec readlines

```
f = open('temperature.txt','r')
ligne = f.readline()
while ligne != '':
    liste = ligne.rstrip('\n').split('\t')
    print('A %s, la température est de %s '%(liste[0],liste[1]))
    ligne = f.readline()
f.close()
```

lecture par ligne avec readline

```
f = open('temperature.txt','r')
for ligne in f:
    liste = ligne.rstrip('\n').split('\t')
    print('A %s, la temperature est de %s '%(liste[0],liste[1]))
f.close()
```

lecture par ligne avec itération

On peut remarquer l'utilisation de méthodes de chaînes de caractères :

- `split()` pour récupérer dans une liste les différents champs d'une ligne séparés par un caractère de séparation
- `rstrip()` pour supprimer les caractères de fin de ligne (on peut aussi supprimer les espaces du début avec avec `lstrip()`)

- Pour rajouter des lignes à la fin du fichier, on peut l'ouvrir en mode ajout `'a'`. Si l'ajout ne se fait pas forcément à la fin, il est recommandé de lire le fichier initial et de remplir au fur et à mesure un autre fichier avec les modifications car l'insertion sans écrasement des données suivantes n'est pas commode :

```
f = open('temperature.txt','a')
while True:
    #0 a pour valeur booléenne False; tous les autres entiers ont pour valeur booleenne True
    test= int(input('Voulez-vous rajouter une ligne au fichier jeu2.txt ? (0 pour Non) \n'))
    if test:
        #on tape \\t pour échapper le caractère spécial \t
        ligne = input('Entrez votre ligne sous la forme Ville\\tTemperature\\n \n')
        f.write(ligne)
    else:
        break
f.close()
```

rajout à la fin du fichier

Exercice 6

Ouvrir le fichier `jeu2.txt`, identifier sa structure, puis écrire :

- Un programme qui lit le fichier en affichant pour chaque ligne un message du type « Personnage est au niveau x et il lui reste y vies. »
- Un programme qui propose à l'utilisateur de rajouter des lignes à la fin du fichier en lui indiquant la structure d'une ligne.

Exercice 7

À l'aide du fichier `premiers-1000.txt`, calculer la somme des 1 000 premiers nombres premiers.

Exercice 8 *Traitement d'un fichier de notes*

On considère le fichier texte `notes2014.csv`[2] qui contient une série de notes structurée comme ci-dessous. Aucun élève n'a été absent à l'un des six devoirs. :

```
ALBERT Marc,15,7,16,16,7,13
BRUN Pierre,6,11,7,5,12,11
```

notes.csv

1. Créer un script `notes.py` avec une fonction `moyenne(liste,coefs)` qui retourne la moyenne d'une liste de notes pondérée par une liste de coefficients.
2. Ecrire une fonction `affiche_moyennes(fichier,coefs)` qui lit un fichier de notes structuré comme `notes.csv` en affichant pour chaque ligne la moyenne de l'élève arrondie au centième. Pour l'arrondi de la moyenne on pourra utiliser la primitive `round` avec la syntaxe `round(moyenne,2)`.
3. Ecrire une fonction `fichier_moyennes(fichier,coefs)` dont les paramètres sont un fichier de notes et une liste `coefs` de coefficients. Cette fonction recopie dans un fichier `fichier+moyennes.csv` chaque ligne du fichier de notes complétée avec la moyenne de l'élève, plus une ligne en fin de fichier avec la moyenne générale de la classe. On doit trouver 9.29 de moyenne pour Pierre Brun et 11.31 pour la classe.

Exercice 9

Ecrire un programme qui réalise les actions suivantes :

1. Ouvrir le fichier texte `poeme_pi.txt` et stocker son contenu dans une variable de type chaîne de caractère.

   Le fichier `poeme_pi.txt` étant encodé en Latin1 et Python3 utilisant nativement l'Unicode (avec l'encodage UTF-8), il faudra passer l'encodage du fichier source comme paramètre de la primitive `open`.

   ```
   1 f = open('poeme_pi.txt','r',encoding='Latin1') #ouverture du fichier
   2 poeme = f.read() #on charge dans poeme le contenu du fichier
   3 f.close() #fermeture du fichier
   ```

2. Remplacer par un espace tous les caractères spéciaux de la variable de type liste :

   `caracteres_speciaux = [', ','! ','? ','_','-',' : ','; ',"\n","\t","\r","'",'(',')','... ','...','. ','.',' '].`

3. Transformer la chaîne en une liste de mots.
4. Retourner une chaîne qui représente l'écriture décimale d'un entier dont les décimales sont les longueurs des mots successifs du poème du premier au dernier.

   Normalement on doit obtenir la suite des décimales de π ...

2. Pour le format `csv` voir `http://fr.wikipedia.org/wiki/Comma-separated_values`

2 Fonctions

2.1 Définition d'une fonction

Exemple 4

- Imaginons que dans un programme on ait besoin de calculer plusieurs images de nombres par la fonction $f : x \mapsto x^2+3x-1$. En cours de mathématiques, on a pris l'habitude de définir la fonction f dans la calculatrice $Y1 = X^2+3*X-1$ puis d'appeler cette fonction lorsqu'on en a besoin pour calculer $Y1(3)$ l'image de 3 par exemple.

  En programmation on peut aussi définir des fonctions, par exemple en Python on déclare la **fonction** f ainsi :

```
>>>def f(x):
      return x**2+3*x-1
```

  On peut appeler `f` pour calculer les images d'expressions et les affecter à des variables.

```
>>> f(3);a =4;f(a);f(a-1);f(1); b = f(f(1)); b
```

  Dans la définition de la fonction `f`, `f` est son nom et `x` un paramètre formel qui joue le même rôle qu'une variable muette en mathématiques.

- Ecrire une fonction qui retourne la valeur absolue d'un flottant `x`.

- Une fonction retourne une valeur avec le mot clef `return`. Une fonction sans `return`, exécute un bloc d'instructions sans retourner de valeur autre que `None`. On parle de *procédure* ou de *fonction avec effet de bord.*. Par exemple, la fonction `bonjour` ci-dessous, effectue un affichage mais retourne la valeur `None`.
  Il ne faut pas confondre `return` (retourne une valeur utilisable par le programme) et `print` (retourne un affichage sur l'écran).
  De plus un `return` fait sortir du corps de la fonction, les instructions suivantes ne seront pas exécutées, ainsi on peut omettre le `else` dans une structure conditionnelle comme dans la fonction `max2` ci-dessous.

```
>>> def bonjour(nom):
      print('Hello '+nom)
>>> a = bonjour('Fred'); print(a)
Hello Fred
None
```

- On peut définir des fonctions avec plusieurs paramètres formels comme ci-dessous la fonction max2 (noter la documentation entre triple quotes après l'en-tête). :

```
def max2(a,b):
    """Fonction qui retourne le maximum de deux flottants """
    if a<b:
        return b
    return a
```

  Utiliser cette fonction pour écrire une fonction qui retourne le maximum de trois flottants .

- Ecrire une fonction qui retourne le quotient dans la division euclidienne d'un entier `a` par un entier `b`. (Reprogrammer `//`).

Définition 3

- Dans un programme on est souvent amené à utiliser plusieurs fois un même bloc d'instructions. Pour réduire le nombre de lignes de codes et clarifier le programme, on peut définir une fonction qui possède un nom, et qui exécute le bloc d'instructions souhaité lorsqu'elle est appelée par ce nom à n'importe quel endroit du programme et autant de fois qu'on le veut.

  Une fonction est caractérisée par sa **signature** (ou en-tête), c'est-à-dire son nom suivi de la liste des **paramètres formels** (ou arguments) de la fonction.

- En Python, pour déclarer une fonction, on écrit le mot clef **`def`** suivi du nom de la fonction, de la liste des paramètres formels entre parenthèses et des deux-points. Vient ensuite avec une indentation supérieure, le bloc d'instructions exécuté par la fonction.

  Lors de l'appel de la fonction, les paramètres formels vont recevoir les **valeurs** des expressions (ou paramètres effectifs) qui leur correspondront dans l'ordre défini par la signature de la fonction.

  Une fonction renvoie toujours une valeur unique. Par défaut il s'agit de **`None`** mais le mot clef **`return`** suivie d'une valeur permet de préciser la valeur à retourner.

- On peut insérer un commentaire de documentation de la fonction entre triple quotes juste après son en-tête.

```
def f(a,b,...):
    """Documentation de la fonction
    qu'on peut appeler avec help(f)
    """
    Bloc d'instructions
    return valeur
```

syntaxe d'une fonction

Exercice 10

1. Ecrire une fonction qui détermine s'il est possible de construire un triangle avec trois segments de mesures données. Elle retournera une valeur booleenne `True` ou `False` et on pourra réutiliser la fonction `max3(x, y, z)` définie précédemment.
2. On définit la fonction suivante :

```
def mystere(x):
    y = x
    z = y * x
    y = x + y + z
    z = z - y
    return y - z
```

Quelle sera la valeur renvoyée par la commande suivante ?

```
>>> mystere(10)
```

Et d'une manière générale, quelle est la valeur renvoyée par `mystere(x)` ?

Exercice 11

1. Écrire une fonction prenant en entrée deux entiers a et b avec $a \leq b$, et renvoyant $\sum_{k=a}^{b} k^5$. Ainsi $\sum_{k=831}^{944} k^5 = 63633265760661375$
2. Ecrire une fonction de paramètre n qui retourne le terme de rang n de la suite définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{2}{u_n}\right) \end{cases}$.
3. Ecrire une fonction de paramètre n qui retourne le terme de rang n de la suite définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = u_n + \sqrt{n} \end{cases}$.
4. Ecrire une fonction de paramètre n qui retourne le terme de rang n de la suite définie par $\begin{cases} f_0 = 0, f_1 = 1 \\ f_{n+2} = f_{n+1} + f_n \end{cases}$.
5. Ecrire une fonction de paramètre n qui retourne le terme de rang n de la suite définie par $\begin{cases} u_0 = 1, u_1 = 2 \\ u_{n+2} = u_{n+1} + n \times u_n \end{cases}$.

Exercice 12 *Fonctions et tableaux ou chaînes de caractères*

1. Ecrire une fonction `echange(tab,i,j)` réalisant l'échange des valeurs d'indices `i` et `j` dans un tableau. On utilise ici le fait que les tableaux/listes sont passés par référence comme arguments d'une fonction (voir paragraphe 3.2 ci-après).
2. Écrire une fonction `maximum(tab)` retournant le maximum des éléments d'un tableau passé en paramètre. De même, écrire une fonction `position_maximum` retournant le maximum et la liste des positions où ce maximum est atteint.
3. Ecrire une fonction `recherche_sequentielle(element, tableau)` qui retourne l'index de la première occurrence d'un élément dans un tableau ou `None` si l'élément n'appartient pas au tableau. L'algorithme consiste à en parcourir le tableau de la première à la dernière position tant que l'élément recherché n'est pas trouvé.
4. Ecrire une fonction `inversion(tab)` qui retourne un nouveau tableau dont les éléments sont dans l'ordre inverse par rapport au tableau passé en paramètre.
   Ecrire une fonction `inversion2(tab)` qui réalise une inversion sur place en modifiant le tableau passé en paramètre.
5. Ecrire une fonction calculant la `somme` des éléments d'un tableau d'entiers à une dimension. (redéfinition de la fonction `sum`).
6. L'itérateur `map(fonction,sequence)` permet d'appliquer une fonction à tous les éléments d'une séquence comme une liste.

```
>>> def carre(x):
...     return x**2
...
>>> L = [1,2,3,4]
>>> for i in map(carre,L):
...     print(i,end=',')
...
1,4,9,16
>>> sum(map(carre,L))
30
```

itérateur map

   Ecrire une fonction `sommeligne` qui prend en entrée un tableau de dimension 2 et qui retourne un tableau contenant les sommes de chaque ligne. En déduire une fonction `sommetableau` qui retourne la somme de tous les éléments du tableau initial sans faire de boucles. Ecrire de même une fonction `sommecarretableau` qui retourne la somme des carrés de tous les éléments d'un tableau de dimension 2.
7. Ecrire une fonction calculant la `moyenne` des éléments d'un tableau puis une autre calculant [3] l'`écart-type`.
8. Ecrire une fonction qui prend en entrée un tableau d'entiers et qui retourne un tableau contenant les sommes cumulées depuis le premier terme.
9. Écrire une fonction qui prend en argument une chaîne de caractères `s` et deux entiers `i` et `j`, et renvoie la sous-chaîne de s comprise entre le caractère d'index i inclus et le caractère d'index `j` exclu. (Reprogrammation de `s[i:j]`).
10. Ecrire une fonction qui prend en argument deux chaînes de caractère et qui détermine s'il s'agit de palindromes.
11. Ecrire une fonction `anagramme(mot1,mot2)` qui détermine si deux chaînes de caractères sont des anagrammes.

Exercice 13 *Fonctions et fichiers*

Ouvrir le fichier `mots.txt`, qui contient sur chaque ligne un mot puis écrire :

1. Une fonction `nombre_mots(chemin_fichier)` qui détermine le nombre de mots dans le fichier.
2. Une fonction `mot_plus_long(chemin_fichier)` qui détermine le mot le plus long dans ce fichier.
3. Une fonction `extraction7(chemin_fichier)` qui extrait tous les mots de 7 lettres du fichier `mots.txt` et les écrit dans un fichier `mots7.txt`.

3. Voir la formule de Koenig-Huygens sur Wikipedia, Variance =carré de la moyenne moins le carré de la moyenne

Exercice 14

Python permet aussi de définir des fonctions de façon concise en une seule ligne avec le mot clef **lambda**.C'est pratique pour définir des fonctions « jetables » et on peut éviter de les nommer.
Comme dans l'exercice 12 on utilise l'itérateur `map(fonction, liste)` qui permet d'appliquer une fonction à tous les éléments d'une liste.
On donne enfin un exemple où on a besoin de définir une fonction jetable pour sélectionner la première composante lorsqu'on veut trier une liste de coordonnées dans l'ordre décroissant des abscisses :

```
>>> def f(x):  #définition classique d'une fonction
...    return x**2
...
>>> list(map(f, [1, 2, 3]))
[1, 4, 9]
>>> f = lambda x : x**2 #définition avec lambda
>>> list(map(f, [1, 2, 3]))
[1, 4, 9]
>>> list(map(lambda x : x**2, [1, 2, 3])) #pas besoin de nommer la fonction
[1, 4, 9]
>>> [e**2 for e in [1, 2, 3]] #avec une liste en compréhension c'est + lisible
[1, 4, 9]
>>> points = [(i, j) for i in range(3) for j in range(3)]
>>> points
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> points.sort(key=lambda t : t[0], reverse=True) #tri après sélection par la fonction key
>>> points
[(2, 0), (2, 1), (2, 2), (1, 0), (1, 1), (1, 2), (0, 0), (0, 1), (0, 2)]
```

1. En combinant avec la fonction **eval**, on peut ainsi définir des fonctions génériques lorsque le corps de la fonction ne comporte qu'une seule instruction :

```
from math import*
fonction = input('Entrez l\'expression de la fonction :\n')
f = lambda x : eval(fonction)
```

fonction avec lambda

Compléter le programme précédent pour qu'il affiche un tableau de valeurs d'une fonction mathématique quelconque à partir d'une valeur initiale et d'un pas saisis par l'utilisateur.

2. On peut aussi écrire facilement des fonctions filtres.

Dans l'exemple qui suit, la fonction `g` retourne 255 si $i \leqslant 100$ et 0 sinon et la fonction `h` retourne 'Hello Fred' si la chaîne saisie est 'Fred' :

```
>>> g = lambda i : i<=100 and 255
>>> g(10),g(101)
(255, False)
>>> h = lambda c : c=='Fred' and 'Hello '+c
>>> h('Paul'),h('Fred')
(False, 'Hello Fred')
```

Si on écrit `f = lambda x : expression1 and expression2`, Python évalue d'abord l'expression logique associée à `x`. Si elle est vraie, il retourne la dernière valeur évaluée (ici expression 2) sinon il retourne `False`.

Sur le même modèle, écrire une fonction filtre qui prend en entrée une note et qui retourne 'admis' si cette note est supérieure ou égale à 10.

Exercice 15 *un peu de tortue Logo*

Compléter le programme ci-contre qui utilise le module `turtle` pour afficher une frise de $2n$ figures, carrés ou triangle alternés dont les côtés ont même longueur.
L'utilisateur doit entrer la longueur des côtés, le nombre de figures, les couleurs des tracés des carrés et triangles et l'écart entre chaque figure.

```
from turtle import*

def carre(cote,couleur):
    color(couleur)
    i = 0
    while i<4:
        forward(cote); right(90);i+=1
    forward(cote)

#fonction triangle(cote,couleur)

speed(5) #vitesse de la tortue
#initialisation des données par l'utilisateur
#boucle qui affiche la frise
mainloop()
```

programme_tortue

2.2 Structure d'un programme : programme principal et fonctions

Exemple 5

On donne ci-après le script d'un programme qui détermine un encadrement du nombre d'Euler e ≈ 2.718 avec un algorithme de dichotomie. On peut décomposer ainsi la structure du programme :

1. D'abord un commentaire spécifique pour préciser l'encodage à l'interpréteur
2. Ensuite la documentation du programme entre triple quotes. On pourra l'appeler avec `help(dichotomie)`
3. Puis les imports de modules, pour accéder à des fonctions définies dans d'autres programmes Python. On utilisera deux principales méthodes d'import :
   - avec `from math import*` on peut appeler la fonction log en écrivant simplement log.
   - avec import math il faut écrire math.log pour appeler la fonction log du module math.
4. Puis viennent les définitions des fonctions utilisées dans le programme, chaque en-tête étant suivi de sa documentation.
5. Enfin vient le programme principal.

```
#-*- coding: Utf-8 -*-
"""Encadrement du nombre d'Euler e avec un algorithme de dichotomie."""

import math #import du module de fonctions mathématiques

def f(x):
    return math.log(x)-1

def dichotomie(a,b,p):
    """a,b et p des flottants retourne un encadrement d'amplitude <=p de la solution de f(x)
          =0 dans [a;b] par l'algorithme de dichotomie"""
    while b-a>p:
        m = (a+b)/2
        if f(m)<0:
            a = m
        elif f(m)>0:
            b = m
        else:
            a = b = m
    return a,b
```

```
binf = float(input('Entrez la borne inférieure : '))
bsup = float(input('Entrez la borne supérieure : '))
amplitude = float(input('Entrez l\'amplitude souhaitée : '))
print('La solution de f(x)=0 sur [%s,%s]'
      'est dans [%s]'%(binf,bsup,dichotomie(binf,bsup,amplitude)))
print('Une valeur approchée de e à 0,000001 près est %.6f'%math.e)
```

2.3 Recherche de sous-mot

Exercice 16 *Recherche d'un sous-mot, d'après un énoncé d'Adrien Lalauze*

Le mot `format` est un sous-mot de `informatique`. De même la liste `[10,3,6]` est une sous-liste de la liste `[13,12,10,3,6,29]` (on ne s'intéresse qu'aux segments de termes consécutifs, dans ce cas `[10,3,6]` n'est pas une sous-liste).
On cherche à déterminer si un mot `m1` (respectivement une liste `t1`) est un sous-mot (respectivement une sous-liste) d ?un mot `m2` (respectivement une liste `t2`). Les manipulations de listes et de chaînes de caractères étant très proches, le programmes doit fonctionner indifféremment sur les listes et les chaînes de caractères.
Si on reprend l'exemple `m1 = "format"` et `m2 = "informatique"`, alors `m2[2:8] == m1`. Une solution avec le slicing[4] est alors :

```
def sousmot(m1,m2):
    L1, L2=len(m1),len(m2)
    for i in range(L2-L1+1):
        if m2[i:i+L1] == m1:
            return True
    return False
```

On se fixe désormais comme contrainte de ne pas utiliser le slicing.

1. Écrire une fonction `positionp` qui teste si le sous-mot `m1` apparait dans `m2` à la position `p`.

   On vérifie dans un premier temps que `p+len(m1)` n'est pas trop grand. Ensuite il faut que `m2[p]=m1[0]`, `m2[p+1]=m1[1]` ...dès qu'on voit une différence on peut sortir en renvoyant `False` sinon si tout est identique on renvoie `True`.

2. Ecrire une fonction `sousmot` prenant en entrée deux chaînes de caractères et retournant `True` ou `False` selon que la première est ou n'est pas un sous-mot de la seconde. (On testera suivant quatre cas : absence, présence au bord gauche, au bord droit, et au milieu et on testera aussi sur des listes).

   ```
   >>> sousmot('su','sumo')
   True
   >>> sousmot('um','sumo')
   True
   >>> sousmot('mo','sumo')
   True
   >>> sousmot('uma','sumo')
   False
   ```

3. Écrire une fonction prenant en entrée deux chaînes de caractères et retournant la liste (éventuellement vide) des positions dans `m2` où on trouve le mot `m1`. Par exemple `positions_sous_mot("bra", "abracadabra")` renvoie `[1, 8]`. (On pourra se servir de la fonction `positionp`)

4. Ecrire une fonction `recherche(fichier,mot,encodage='Utf8')` qui ouvre le `fichier` texte (avec éventuellement la précision de l'encodage), stocke son contenu dans une chaîne et retourne la position de la première occurence de `mot` dans le texte ou `None` si `mot` n'apparaît pas.

4. Possibilité qui n'est pas offerte par tous les langages

3 Portée des variables

3.1 Espaces de nommage

Exercice 17

Dans le corps de la procédure `echange` suivante, les valeurs des variables `a` et `b` doivent être échangées. Compléter le programme et le tester. Que remarque-t-on ?

```
#-*- coding: Utf-8 -*-

def echange(a,b):
    """procédure qui échange deux entiers"""
    ......

    print('Dans l\'espace de nommage local de la fonction : a= %d et b=%d'%(a,b))

if __name__ == "__main__":
    a = 2
    b = 3
    print('Initialement on a : a= %d et b=%d'%(a,b))
    echange(a,b)
    print('Retour à l\'espace de nommage global : a= %d et b=%d'%(a,b))
```

Définition 4

- La **portée d'une variable** définit où (dans quelle partie du programme) et comment une variable est accessible (en lecture, en écriture ?).
- En Python, les variables sont caractérisées par leur nom et leur valeur et comme dans la plupart des langages on distingue deux types d'espaces où les variables sont accessibles :
  - l'**espace de nommage global** contient toutes les variables définies dans le programme principal (`main`).
  - un **espace de nommage local** propre à un bloc d'instruction comme le corps d'une fonction qui contient toutes les variables définies dans ce bloc. Les blocs d'instructions conditionnels et itératifs ne définissent pas leur propre espace de nommage.

Propriété 1

- L'appel d'une fonction définit un **espace de nommage local**. Celui-ci contient juste les paramètres qui sont passés à la fonction et les variables définies dans son corps.
- Lorsqu'on appelle une fonction dans le programme principal, on lui passe en **arguments** des expressions calculées à partir des variables de l'espace global. Les valeurs de ces expressions sont affectées aux variables nommées par les **paramètres formels** de la fonction. Un paramètre formel est défini dans l'espace local de la fonction et constitue un alias vers la variable globale dont il reçoit la valeur et dont il partage l'identifiant mémoire (tant qu'il ne subit pas d'affectation).
- L'espace de nommage local de la fonction est effacé lorsque l'exécution de la fonction est terminée. Les variables d'un espace de nommage local ne sont pas visibles depuis l'espace de nommage global du programme principal.
- En Python, la fonction `globals()` retourne un dictionnaire des variables globales avec leurs valeurs dans l'état d'exécution courant du programme. La fonction `locals()` retourne un dictionnaire des variables locales avec leurs valeurs.

- Par défaut les variables de l'espace global sont toujours visibles depuis un espace de nommage local. Si une variable `var` a pour valeur 5 dans l'espace global et que dans le corps d'une fonction se trouve l'affectation `var = 6`, la valeur affichée pour `var` dans la fonction sera 6 mais lorsque son exécution sera terminée, `var` retrouvera sa valeur 5 de l'espace global.

  Une variable `var` de l'espace global ne peut être modifiée par une affectation dans un espace de nommage local, à moins d'écrire `global var` au début du corps de la fonction.

Exemple 6

Le code ci-dessous illustre les notions d'espace de nommage global et local.

```
>>> def f(x):
        print('Espace de nommage global à l\'appel : \n',globals())
        print('Identifiant du paramètre: ',id(x))
        print('Espace de nommage local à l\'appel : \n',locals())
        x = x+ 1
        print('Espace local en fin d'appel : \n',locals())
        print('Espace global en fin d\'appel : \n',globals())

>>> a = 2
>>> id(a)
505493912
>>> f(a)
Espace de nommage global à l'appel :
 {'a': 2, 'f': <function f at 0x02BFFC90>, '__builtins__': <module 'builtins' (built-in)>, '
      __package__': None, '__name__': '__main__', '__doc__': None}
Identifiant du paramètre: 505493912
Espace de nommage local à l\'appel :
 {'x': 2}
Espace local en fin d'appel :
 {'x': 3}
Espace global en fin d'appel :
 {'a': 2, 'f': <function f at 0x02BFFC90>, '__builtins__': <module 'builtins' (built-in)>, '
      __package__': None, '__name__': '__main__', '__doc__': None}
>>> globals() == locals()
True
```

nommage1

Après l'appel de la fonction l'espace de nommage local de la fonction est effacé et l'espace de nommage local est l'espace global du programme principal.

Exercice 18

Pour chacun des scripts ci-dessous, s'il est correct déterminer les valeurs des sorties écran et sinon expliquer pourquoi il y aura un message d'erreur. Vérifier ensuite avec la console.

```
#Fonction
def f():
    print(a)
#Programme principal
a = 2
f()
```

nommage2

```
#Fonction
def f():
    a = a+1
    print(a)
#Programme principal
a = 2
f()
```

nommage3

```
#Fonction
def f():
    a = 3
    print('Dans \'espace local a = ',a)
#Programme principal
a = 2
f()
print('Après l\'appel de la fonction a =
      ',a)
```

nommage4

```
#Fonction
def f():
    global a
    a = a+ 1
#Programme principal
a = 2
f()
print(a)
```

nommage5

Exercice 19

Pour chacun des scripts suivants,lorsqu'une instruction demande l'affichage de l'espace de nommage global ou local, prévoir le résultat.

```
#Fonction
def f():
    print(globals())
    print(locals())
    b = 3
    print(a)
    print(locals())
    print(globals())
#Programme principal
a = 2
f()
print(globals())
```

nommage6

```
#Fonction
def f(N):
    print(globals())
    print(locals())
    N = N+[3]
    print(locals())
    print(globals())
#Programme principal
L = [1,2]
f(L)
print(globals())
```

nommage7

3.2 Passage de paramètres et type de variables

Exercice 20

Le programme `parametre.py` a pour but d'examiner l'effet d'une modification d'une variable par une fonction selon le type de la variable passée en paramètre.
Tester ce programme avec des valeurs de variables de type `int, float, bool, str` ou `list`.Que remarque-t-on ?

Propriété 2

On distingue deux type de passage en paramètre d'une variable à une fonction :

- Lors d'un **passage par valeur** une nouvelle variable de même type que la variable passée en paramètre est créée dans l'espace de nommage de la fonction et on lui affecte une copie de la valeur de la variable passée en paramètre. Si cette dernière est de type composite (donc gourmande en espace mémoire) on double l'occupation mémoire. En revanche les deux variables étant distinctes, on peut modifier la variable paramètre dans le corps de la fonction sans altérer la variable initiale. Le passage par valeur concerne principalement les variables de type simple : `int, float, bool, str`.
- Lors d'un **passage par référence** la variable locale à l'espace de nommage de la fonction ne reçoit que la référence mémoire de la variable passée . C'est plus économe en espace mémoire mais toute modification du paramètre dans le corps de la fonction altère le contenu de la variable initiale puisque les deux pointent vers la même case mémoire. Le passage par référence est conseillé pour les variables de type composite (tableaux, listes, dictionnaires, données structurées).

En Python les paramètres formels de la fonction créent des alias vers les valeurs des variables de l'espace global. Les variables globales de type non mutable `int, float, bool, str, tuple`, ne peuvent être modifiées par la fonction , on peut considérer qu'elles sont passées par valeur. Mais celles qui sont des pointeurs et mutables comme les listes, les dictionnaires peuvent subir une modification à travers le paramètre formel. Ces variables sont passées par référence et si la fonction les altère, on parle d'**effet de bord**.

Exercice 21

1. Comparer le résultat donné par le script suivant avec celui du script `nommage7` .

```
1  def f(M):
2      print(globals())
3      print(locals())
4      M.append(3) #modification sur place, identique à M +=[3] mais pas à M = M+[3] qui
           est une réaffectation
5      print(locals())
6      print(globals())
7
8  L = [1,2]
9  f(L)
10 print(globals())
```

nommage8

2. Remplacer l'instruction de la ligne 5 par `M[0]=0`. Que peut-on dire de la liste `M` après l'appel de la fonction ?

Exercice 22

Ecrire une fonction sans paramètres qui échange les contenus de deux variables globales `a` et `b`.

Exercice 23 *Ordre d'évaluation des paramètres*

1. Tester le programme ci-contre, puis remplacer la dernière instruction par print(somme(f(1),a)) où l'ordre de passage des paramètres est inversé.

Que remarque-t-on ?

2. Que peut-on conjecturer sur l'ordre dans lequel Python évalue les paramètres passés à une fonction ?

```
1  def f(x):
2      global a
3      a = a+x
4      return a
5
6  def somme(x,y):
7      return x+y
8
9  a = 0
10 print(somme(a,f(1)))
```

Exercice 24

1. Ecrire une fonction qui prend entrée une liste `L` représentant une matrice de dimensions $n \times m$ et qui modifie cette liste en appliquant à tous ces éléments `L[i][j]` avec $1 \leqslant i \leqslant n$ et $1 \leqslant j \leqslant m$ de cette matrice en leur appliquant la fonction $f : x \mapsto 255 - x$.
2. Ecrire une fonction qui prend entrée une liste `L` représentant une matrice et retourne une nouvelle matrice construite comme en 1. sans que la matrice L soit modifiée.
3. Ecrire un programme avec une double boucle, qui prend en entrée une liste `M` représentant et qui retourne en sortie sa transposée c'est-à-dire la matrice N dont les lignes sont les colonnes de M.

   Par exemple la transposée de `M = [[12, 9, 13], [10, 14, 12], [15, 16, 17]]`, est :

   `N = [[12, 10, 15], [9, 14, 16], [13, 12, 17]]`.
   En Python, les listes en compréhension permettraient même de se passer d'une double boucle pour créer la transposée N de M :

```
1 N = [[M[j][i] for j in range(len(M))] for i in range(len(M[0]))]
```

4 Récursivité

4.1 Des fonctions qui s'appellent elles mêmes

Exemple 7

1. Dans le script ci-contre, les fonctions `h` et `m` font appel dans leur corps aux fonctions `f` et `g`. Quels résultats vont être affichés par les instructions des lignes 18 et 19 pour les entrées $x = 4$ et $x = -2.4$?

   Tester ce script pour vérifier.
2. Soient a un réel et n un entier naturel, on se propose de définir une fonction qui prend en argument un flottant a et un entier n et qui retourne le flottant a^n.

   On rappelle que a^n peut être défini à l'aide d'une suite récurrente en notant $a_n = a^n$:

$$\begin{cases} a_0 = 1 \\ a_{n+1} = a_n \times a \end{cases}$$

   a. Définir une fonction `puissance1` qui réalise le traitement souhaité avec un **algorithme itératif**.

   b. La fonction `puissance2` ci-dessous réalise aussi le traitement souhaité mais elle implémente un **algorithme récursif**, c'est-à-dire que la fonction s'appelle elle-même (en ligne 7).

   c. Programmer la fonction `puissance2` et la tester avec les arguments $(a,n) = (2,0)$, $(a,n) = (2,1)$, $(a,n) = (2,2)$, $(a,n) = (2,3)$.

   Expliquer le déroulement du traitement effectué par la fonction `puissance2` pour les arguments $(a,n) = (2,3)$.

```
1  import math
2
3  def f(x):
4      return x/2
5
6  def g(x):
7      return math.floor(x)
8
9  def h(x):
10     return f(g(x))
11
12 def m(x):
13     return g(f(x))
14
15 if __name__ == "__main__":
16     x = float(input('Entrez x : '))
17     print('f(g(x)) est égal à : ',h(x)
           )
18     print('g(f(x)) est égal à : ',m(x)
           )
```

composition de fonctions

```
1  def puissance2(a,n):
2      """retourne a^n avec un algorithme
             récursif
3      """
4      if n == 0: #cas de base
5          return 1
6      #appel récursif
7      return a*puissance2(a,n-1)
```

puissance2

Définition 5

1. Une fonction récursive est une fonction qui contient un appel à elle-même.
2. Une fonction récursive doit respecter les règles suivantes :
   - Une fonction récursive doit comporter un cas de base sinon le risque de non terminaison se pose comme pour les boucles.
   - Une fonction récursive doit modifier son état pour se ramener au cas de base.

Exercice 25

1. Ecrire une fonction récursive `factrec` qui prend en argument un entier naturel n et qui retourne $n! = 1 \times 2 \times \cdots \times (n-1) \times n$.
   Ecrire une fonction `factit` qui effectue le même traitement mais avec un algorithme itératif.
2. Écrire une fonction récursive qui calcule le quotient de la division euclidienne d'un entier naturel par un autre, sans utiliser l'opérateur / du langage.
3. Écrire une fonction récursive qui calcule le reste de la division euclidienne d'un entier naturel par un autre, sans utiliser l'opérateur % du langage.

4.2 Efficacité d'un algorithme récursif : calcul de termes de la suite de Fibonacci

Exercice 26 *Suite de Fibonacci*

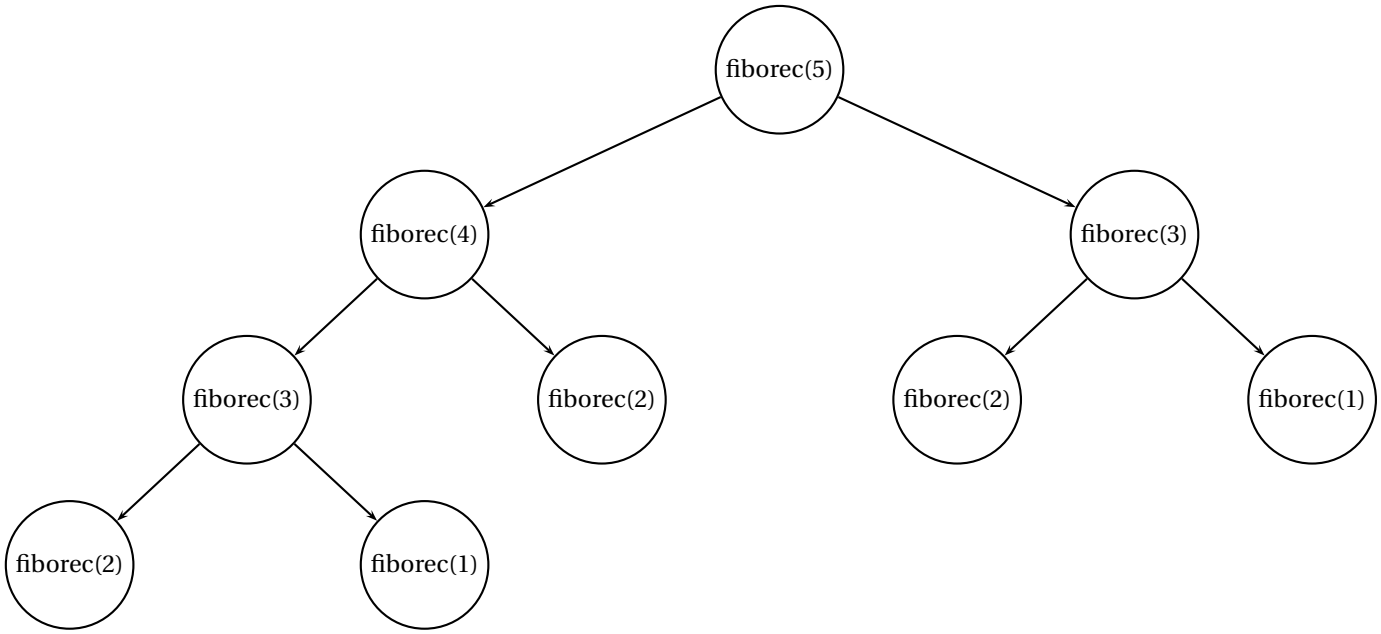
```
1  n = int(input('Entrez a : '))
2  import time #import du module time
3  #defintions des fonctions fibo et fiborec
4  #programme principal
5  n = int(input('Entrez n : '))
6  debut = time.time()
7  fibo_imp = fibo(n)
8  fin = time.time()
9  duree = fin - debut
10 print('Fn par un algorithme itératif : %s en %.6f secondes'%(fibo_imp,duree))
```

itératif versus récursif

La suite de Fibonacci $(F)_{n\geqslant 0}$ est définie par : $\begin{cases} F_0 = 1 \text{ et } F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \end{cases}$.

1. Ecrire une fonction `fiborec` qui prend en argument un entier naturel n et qui retourne F_n avec un algorithme récursif calqué sur la définition de la suite.
2. Ecrire une fonction `fibo` qui prend en argument un entier naturel n et qui retourne F_n avec un algorithme itératif.
3. En utilisant la fonction `time` du module `time`, compléter le script ci-dessus pour comparer les temps ce calcul des fonctions `fibo` et `fiborec` pour les arguments $n = 10$, $n = 20$, $n = 30$, $n = 35$ et $n = 40$. Commenter.
4. Compléter l'arbre ci-dessous qui recense les appels récursifs de `fiborec` nécessaires au calcul de `fiborec(5)`. Comment cet arbre permet d'expliquer la différence de[5] complexité entre l'algorithme itératif `fibo` et l'algorithme récursif `fiborec` ?

5. L'algorithme itératif de calcul de F_n a une complexité linéaire, proportionnelle à n, alors que l'algorithme récursif a une complexité exponentielle de l'ordre de F_n, proportionnelle à $\left(\frac{1+\sqrt{5}}{2}\right)^n$



4.3 Fonction récursive et Pile

1. Programmer la fonction `factrec2` ci-dessous et la tester avec l'argument $n = 4$.

2. L'appel de `factrec2(4)`, déclenche la chaîne d'instructions suivantes qui nous ramène au cas de base :

- **a.** demande d'évaluation de `4*factrec2(3)` donc appel de `factrec2(3)`
- **b.** demande d'évaluation de `3*factrec2(2)` donc appel de `factrec2(2)`
- **c.** demande d'évaluation de `2*factrec2(1)` donc appel de `factrec2(1)`
- **d.** cas de base, retourne la valeur 1 et ferme l'appel de `factrec2(1)`

A partir de là on remonte la chaîne pour évaluer toutes les expressions de retour des appels successifs de la fonction `factrec2` et fermer tous les appels de fonction. La dernière valeur retournée est celle de `4*factrec2(3)` = 4 × (3 × (2 × (1))).

- **a.** on peut évaluer `2*factrec2(1)=2*1=2` et fermer l'appel de `factrec2(2)`
- **b.** on utilise ce résultat pour évaluer `3*factrec2(2)=3*2=6` et fermer l'appel de `factrec2(3)`
- **c.** on utilise ce résultat pour évaluer `4*factrec2(3)=4*6=24` et fermer l'appel de `factrec2(3)`
- **d.** on retourne `4*factrec2(3)` = 4 × (3 × (2 × (1))) et on ferme l'appel de `factrec2(4)`.

3. L'ordinateur utilise une pile pour stocker toutes les données (ou opérandes) nécessaires au calcul. Le premier calcul n'est effectué que lorsque le cas de base `n == 0` est atteint. La pile est alors de taille $n+1$ soit 4 pour $n = 3$ L'ordinateur procède alors ainsi : il dépile les deux opérandes du dessus de la pile puis empile le résultat et ainsi de suite ....

4	3	2	1	*	*	*

$\Longrightarrow$

4	3	2	*	*

$\Longrightarrow$

4	6	*

$\Longrightarrow$

24

Une pile est une structure de données de type `LIFO` (`Last In First Out`), puisque les premières valeurs à sortir de la pile sont celles du dessus c'est-à-dire les dernières qui ont été empilées.
Pour des compléments sur les piles en informatique, on pourra consulter les pages web suivantes :
`http://fr.wikipedia.org/wiki/Pile_(informatique)` et `http://fr.wikipedia.org/wiki/Last_in,_first_out`
`http://fr.wikipedia.org/wiki/Notation_polonaise_inverse`.

```
1 def factrec2(n):
2     if n == 1: # ou n == 0: return 1 pour avoir factorielle 0
3         return 1
4     else:
5         print('--'*n+'>appel de factrec2(%s)'%(n-1))
6         f = n*factrec2(n-1)
7         print('--'*n+'>sortie de factrec2(%s)'%(n-1))
8         return f
```

factrec2

4.4 Quelques exercices

Exercice 27 *exponentiation rapide*

Soit a un réel positif. Pour calculer a^n, si on utilise l'algorithme naif : $a \times a \times \cdots \times a$ cela nécessite $n-1$ multiplications.
Pour réduire le coût en opérations, on peut remarquer que :

$$a = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ a \times \left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

On peut alors naturellement programmer un algorithme d'exponentiation rapide récursif qui nécessite au plus $\log_2(n)$ multiplications où $\log_2(n)$ est le logarithme en base 2 de n.
La partie entière de $\log_2(n)$ est le nombre de chiffres de l'écriture binaire de n moins 1, de la même façon que la partie entière du logarithme décimal d'un entier n est le nombre de chiffres de son écriture décimale moins 1.
Si on note $\lfloor x \rfloor$ la partie entière de x on a :

$$2^{\log_2(n)} = n \text{ et } 2^{\lfloor \log_2(n) \rfloor} \leqslant n < 2^{\lfloor \log_2(n) \rfloor + 1}$$

Petite vérification avec Python :

```
>>> from math import log
>>> log(45,2) #logarithme en base 2
5.491853096329675
>>> 2**log(45,2)
45.0
>>> bin(45)
'0b101101'
```

1. Programmer une fonction `exporapide_rec()` qui implémente l'algorithme d'exponentiation rapide de façon récursive.
2. Comparer les performances des fonctions `exporapide_rec()` et `puissance2rec()` (exponentiation récursive avec l'algorithme naif)en utilisant la fonction `time` du module `time`.

Exercice 28

1. Programmer une fonction qui prend en entrée un entier n et qui retourne le terme d'indice n de la suite (u_n) définie par $\begin{cases} u_0 = 2 \\ u_{n+1} = u_n + 2n \end{cases}$
2. Programmer une fonction qui prend en entrée un entier n et qui retourne le terme d'indice n de la suite (u_n) définie par $\begin{cases} u_0 = 2 \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{3}{u_n}\right) \end{cases}$
3. Programmer de façon récursive une fonction qui prend en argument une liste d'entiers positifs $[a_0, a_1, \cdots, a_n]$ et qui retourne $\sqrt{a_0 + \sqrt{a_1 + \sqrt{\cdots + \sqrt{a_n}}}}$.
4. Programmer une fonction qui effectue le même traitement que précédemment mais avec un algorithme itératif.

Exercice 29

Ecrire deux fonctions `expoexpo_iter(a, n)` et `expoexpo_rec(a, n)` qui calculent l'une de façon itérative, l'autre de façon récursive le nombre $a^{a^{\cdots^a}}$ qui est le résultat de n exponentiations successives d'un entier a par lui même.
On vérifiera que $2^{2^2} = 2^4 = 16$ et $2^0 = 1$.

Exercice 30 *algorithme de Horner pour l'évaluation d'un polynôme*

Soit un polynôme $P(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$ et soit un réel α.
Si on veut calculer $P(\alpha)$, la méthode naive consiste à calculer les α^i pour $i \in [\![0; n]\!]$ puis multiplier par a_i (soit i multiplications à chaque fois) puis sommer tous ces résultats.
La méthode de Horner consiste à effectuer les opérations en remarquant que :

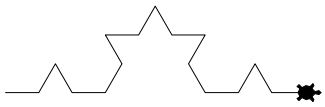
$$P(\alpha) = ((\cdots((a_n\alpha + a_{n-1})\,\alpha + a_{n-2})\,\alpha + \cdots)\,\alpha + a_1)\,\alpha + a_0$$

1. Déterminer le nombre d'opérations (additions et multiplications) nécessaires pour calculer $P(\alpha)$ avec la méthode naive puis avec la méthode de Horner.
2. Programmer deux fonctions qui prennent en entrée la liste des coefficients d'un polynôme et un flottant x et qui retournent l'évaluation $P(x)$ du polynôme en x : l'une avec méthode naive et l'autre avec la méthode Horner.

4.5 Fonctions récursives et dessin

Exercice 31 *Flocon de Von Koch*

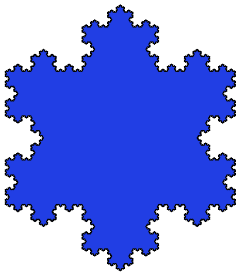
L'algorithme de Von Koch consiste à partir d'un segment donné, à le diviser en 3 segments de même longueur et à remplacer le segment médian par les 2 côtés d'un triangle équilatéral construits extérieurement.
On donne ci-dessous une figures illustrant les deux premières itérations de l'algorithme à partir d'un segment de longueur fixé. Pour faire le tracé on va utiliser le module `turtle`.



1. Construire la figure obtenue après 3 applications récursives de l'algorithme de Von Koch à un segment de longueur 27.
2. On veut d'abord écrire une fonction `koch_iter(x)` qui applique 4 fois l'algorithme de Von Koch à un segment de longueur `x` en utilisant des boucles imbriquées.

   Compléter le script `Von Koch itératif` donné ci-après.
3. Dans le script `Von Koch récursif` compléter l'écriture de la fonction récursive `koch_rec` qui applique n fois l'algorithme de Von Koch à un segment de longueur x.
4. Tester cette fonction avec les entrées $(x, n) = (300, 1)$, $(x, n) = (300, 2)$, $(x, n) = (300, 3)$, $(x, n) = (300, 5)$.
5. Le flocon de Von Koch est obtenu en appliquant l'algorithme de Von Koch aux trois côtés d'un triangle équilatéral. En utilisant la fonction `koch_rec` précédente, écrire une fonction `floconkoch_rec(x,n)` qui dessine un flocon de Von Koch.

   Tester cette fonction pour les entrées $(x, n) = (300, 1)$, $(x, n) = (300, 2)$, $(x, n) = (300, 3)$, $(x, n) = (300, 5)$.



```
#-*- coding: Utf-8 -*
from turtle import*

def koch_iter(x):
    def rotation(p):
        """rotation de la tortue selon la valeur
        de l'indice p de la boucle"""
        if p==1:
            right(120)
        elif p==0 or p==2:
            left(60)

    up()
    goto(-x/2,0)
    down()
    for i ..........:
        for j ..........:
            for k ..........:
                forward(x/81)
                left(60)
                .......
                rotation(k)
            rotation(j)
        rotation(i)

#test pour un flocon de coté 300 pixels
koch_iter(300)
```

Von Koch iteratif

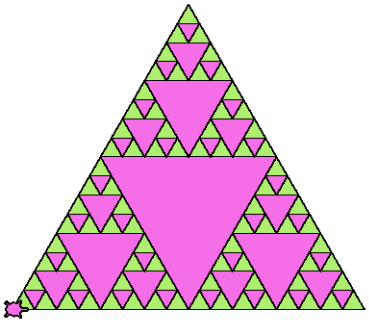
```
#-*- coding: Utf-8 -*
from turtle import*

def koch_rec(x,n):
    """n applications récursives de l'algorithme de Von Koch
    à un segment de longueur x"""
    if n==0:
        forward(x)
    else:
        koch_rec(x/3, n-1)
        left(60)
        koch_rec(x/3, n-1)
        #fonction à compléter ...

#programme principal
speed(0) #vitesse maximale de la tortue
x=int(input('Entrez la longueur du segment en pixel : '))
n=int(input('Entrez la profondeur de récursion : '))
koch_rec(x,n)
done()
```

Von Koch recursif

Exercice 32 *Mini-Projet*

On peut définir ainsi le triangle de Sierpinski.

Figure fractale obtenue en appliquant indéfiniment l'algorithme suivant à partir d'un triangle équilatéral initial pointe orientée vers le haut : à chaque itération on divise tous les triangles équilatéraux de la figure avec la point orientée vers le haut en quatre triangles équilatéraux en reliant les milieux des côtés (ce qui forme un triangle équilatéral avec point orientée vers le bas.)



On a représenté ci-contre un triangle de Sierpinski obtenu après 4 itérations de l'algorithme.

Le programme ci-dessous utilise le module `turtle`.
Dans le programme principal on trace d'abord (ligne 27 à 31) un triangle de couleur verte de notation (r,g,b)=(175,239,111) et de côté x.
Puis on applique récursivement n fois l'algorithme de Sierpinski pour découper des triangles de couleur color2 (ici '#F66EE7'en notation hexadécimale) à l'intérieur du triangle initial.
C'est la fonction `sierpinski(c,n,color2='#F66EE7')` qui réalise ce découpage récursif. Son paramètre `color2` a une valeur fixée par défaut.
La fonction `sauve(nomFichier)` enregistre l'image obtenue sous le format d'image vectorielle `eps` (PostScript Encapsulé).

```
1  from turtle import*
2
3  def sauve(nomFichier):
4      #on récupère l'écran de la tortue
5      ts = getscreen()
6      #on récupère la canevas (zone de tracé) de l'écran de la tortue c'est un canevas
7      #auquel on peut appliquer les méthodes des canevas du module Tkinter
8      #en particulier on peut le convertir en fichier postscript
9      ts.getcanvas().postscript(file="{0}.eps".format(nomFichier))
10
11 def sierpinski(c,n,color2='#F66EE7'):
12     """fonction récursive qui découpe des triangles de sierpinski à l'intérieur d'un
           triangle plein de coté c.
13     n est le niveau de récursion. color2 est la couleur de remplissage
14     des triangles découpés."""
15     ......
16
17 #programme principal
18 n = int(input('Entrez le niveau de récursion : '))
19 up()
20 c = (window_width()+window_height())/4
21 goto(-c/2,-c/2)
22 down()
23 #on construit un grand triangle de coté c remplit d'une certaine couleur
24 colormode(255)
25 r,g,b = 175,239,111
26 fillcolor((r,g,b))
27 begin_fill()
28 for i in range(3):
29     forward(c)
30     left(120)
31 end_fill()
32 sierpinski(c,n)
33 sauve('sierpinski_rec%s'%n)
34 done()
```

5 D'autres structures de données

5.1 Les ensembles

Méthode *Opérations sur les ensembles*

En Python, un **ensemble**, de type **set** est une **collection non ordonnée d'objets uniques**.

```
>>> s1 = set() #création d'ensemble vide, {} est un dictionnaire vide, pas un ensemble
>>> type(s1)
<class 'set'>
>>> s1 = s1 | {5} #opérateur réunion
>>> s1
{5}
>>> s1 |= {2,3} #opérateur update avec réunion (comme + = pour les variables de type int
     )
>>> s1
{2, 3, 5}
>>> s2 = {5,3,4} #création d'ensemble non vide ou s2 = set((5,3,4))
>>> s1-s2 #opérateur de différence
{2}
>>> s1&s2 #opérateur d'intersection
{3, 5}
>>> s1^s2 #opérateur de différence symétrique
{2, 4}
>>> s3 = {2,3}
>>> s3<s1 #opérateur de comparaison (est inclus dans)
True
>>> s2>s3 #opérateur de comparaison (contient)
False
>>> s4, s5 = set([1,2,3,4]), set('chaine')
 #transformation d'une liste ou d'une chaine de caractères en ensemble
>>> s4, s5
{1, 2, 3, 4}, {'a', 'c', 'e', 'i', 'h', 'n'}
```

Exercice 33

1. Utiliser des ensembles pour écrire une fonction qui détermine si un mot comporte des voyelles, puis une fonction qui détermine si un nombre entier comporte le chiffre 5 dans son écriture décimale.
2. Peut-on utiliser des ensembles pour déterminer si deux chaînes de caractères sont des anagrammes ? pour déterminer si deux listes d'entiers comportent les mêmes éléments, éventuellement dans le désordre ?
3. Utiliser des ensembles pour écrire une fonction `doublon(liste)` qui détermine le nombre de doublons dans une liste.

5.2 Les dictionnaires

Méthode *Manipulation de dictionnaires*

Un dictionnaire (type `'dict'`)est une collection non ordonnée de relations entre clefs et valeurs. Un dictionnaire est un type de données modifiable (comme les listes).
Ci-dessous on définit puis manipule une variable `options` de type `'dict'` qui contient les couples (clef,valeur)=(spécialité,effectif) de la répartition des spécialités dans une classe de Terminale S :

```
>>> options = {} #définition du dictionnaire ou options = dict()
>>> options['ISN'] = 18
>>> options['Maths'] = 9
>>> options['Physique'] = 5
>>> options['SVT'] = 1
>>> print(options)
{'Maths': 9, 'SVT': 1, 'Physique': 5, 'ISN': 18}
>>> print(options['ISN'])
18
>>> del options['SVT'] #suppression d'une entrée
>>> print(options)
{'Maths': 9, 'Physique': 5, 'ISN': 18}
>>> print(len(dico)) #len retourne la longueur du dictionnaire
3
>>> 'SVT' in options #Pour tester l'appartenance d'une clef au dictionnaire
False
>>> for clef in options.keys(): #vue des clefs avec la méthode keys
...    print(clef)
...
Maths
Physique
ISN
>>> for clef,valeur in options.items(): #vue des couples (clef,valeur) avec la méthode
      items
...    print(clef,valeur)
...
Maths 9
Physique 5
ISN 18
>>> options['SVT']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'SVT'
>>> dico.get('SVT','Cette spécialité n\'est pas disponible')
#la méthode get ne déclenche pas d'exception si clef n'existe pas
"Cette spécialité n'est pas disponible"
```

```
>>> options.update({'Maths': 7, 'Physique': 14, 'ISN': 3,'SVT' : 15})
#modification sans changer l'identifiant mémoire de la variable options avec la méthode
      update
>>> dico = {'C' : 'vélo', 'A' : 'bateau', 'B': 'moto'}
>>> sorted(dico) #la primitive sorted ne classe que les clefs
['A', 'B', 'C']
>>> l = list(dico.items()) ; print(l) #on crée une liste des entrées
[('A', 'bateau'), ('C', 'vélo'), ('B', 'moto')]
>>> sorted(l) #on peut classer cette liste avec sorted
[('A', 'bateau'), ('B', 'moto'), ('C', 'vélo')]
```

Exercice 34 *Mini-Projet : Répertoire téléphonique*

Paul veut créer un répertoire avec les numéros de téléphone de ses amis stockées sous la forme (clef,valeur) =('Nom Prénom','numéro de téléphone') par exemple ('Pierre','0645464748') dans un dictionnaire. Dans le programme ci-dessous, la variable `répertoire` est une variable globale et on utilise un dictionnaire de fonctions `choix` pour rediriger le flux d'instruction vers l'utilisation de la fonction souhaitée. Compléter le programme avec l'écriture :

1. d'une procédure `saisie` qui permet d'ajouter de nouvelles entrées au répertoire tant qu'on le souhaite ;
2. d'une procédure `recherche_numero` qui prend en entrée un 'Prénom' et retourne en sortie le numéro de téléphone correspondant ;
3. d'une procédure `tri_alphabetique` qui retourne une liste triée par ordre alphabétique des 'Noms' des entrées ('Nom Prénom','numéro de téléphone') du répertoire ;
4. d'une procédure `repertoire_inverse` qui crée un répertoire inversé d'entrées (clef,valeur) =('numéro de téléphone','Nom Prénom'). On suppose dans ce cas que la relation (clef,valeur) est bijective et que deux personnes différentes n'ont pas le même numéro ;
5. d'une procédure `recherche_personne` qui prend en entrée un numéro de téléphone et retourne en sortie le 'Nom Prénom' correspondant ;

```
#-*-coding:Utf-8-*
import pickle

"""
le module pickle permet de sauvegarder des données complexes comme
des dictionnaires, des listes dans des fichiers grace à la sérialisation
des données en une chaine de caractères.
La fonction pickle.dump permet de sauvegarder une variable dans un fichier
et la fonction pickle.load permet de charger une variable stockée dans un
fichier. Voici un exemple :

>>> import pickle
>>> options = {'ISN': 7, 'Maths': 8, 'SVT': 10, 'Physique': 10}
>>> fic = open('dico','wb') #options 'wb' pour écriture en mode binaire
>>> pickle.dump(options,fic)
>>> fic.close()
>>> fic = open('dico','rb') #options 'rb' pour lecture en mode binaire
>>> recup = pickle.load(fic)
>>> fic.close()
>>> print(recup)
{'Maths': 8, 'SVT': 10, 'Physique': 10, 'ISN': 7}
"""
```

Code à compléter

```
def charger(repertoire):
    """charge dans une variable un répertoire stocké dans un fichier
    binaire avec la fonction load du module pickle.
     Le paramètre fichier est un nom de fichier"""
    fichier = input('Entrez le nom du fichier')
    with open(fichier,'rb') as fic:
        repertoire.update(pickle.load(fic))

def enregistrer(repertoire):
    """enregistre la variable répertoire dans un fichier
    binaire avec la fonction dump du module pickle.
     Le paramètre fichier est un nom de fichier"""
    fichier = input('Entrez le nom du fichier')
    with open(fichier,'wb') as fic:
        pickle.dump(repertoire,fic)

def saisir(repertoire):
    .......

def recherche_numero(repertoire):
    .......

def tri_alphabetique(repertoire):
    .......

def repertoire_inverse(repertoire):
    .......

def recherche_personne(repertoire):
    .......


if __name__ == "__main__":
    repertoire = {}
    print(id(repertoire))
    menu = ['C pour Charger','E pour Enregistrer','S pour Saisir',
    'P pour Rechercher d\'un numéro à partir d\'une personne ',
    'T pour Rechercher d\'une personne à partir d\'un numéro ','Un autre choix met fin au
         programme']
    choix = { 'C': charger , 'E' : enregistrer ,'S' : saisir,
    'P' : recherche_numero, 'T' :recherche_personne}
    print('Voici les choix possibles : \n')
    for i in menu:
        print(i)
    rep = input('Entrez votre choix : \n')
    rep.upper().strip()
    while rep in choix:
        choix[rep](repertoire)
        print(repertoire)
        rep = input('Entrez votre choix')
        rep.upper().strip()
```

Code à compléter

Exercice 35 *Mini-Projet : Le mot le plus long*

Le principe du jeu est le suivant : l'utilisateur saisit une série de lettres et l'ordinateur retourne la liste des plus longs mots écrits avec ces lettres (la même lettre peut être répétée) qu'il peut trouver dans son dictionnaire.

1. Le dictionnaire des mots connus par l'ordinateur se trouve dans le fichier `dico.txt`. Pour constituer un dictionnaire de mots classés par longueur, écrire une fonction `dicolongueur(fichier)` qui ouvre un fichier texte et qui retourne un dictionnaire

Python dont les clefs sont des longueurs de mots et les valeurs, les listes de mots du fichier de longueur donnée, classés dans l'ordre alphabétique.

2. Ecrire une fonction `tirage(n)` qui retourne une chaîne de caractères de longueur `n`, composée de lettres minuscules choisies au hasard.

3. Ecrire une fonction `histo(mot)` qui retourne un dictionnaire contenant l'histogramme de répartition des lettres de `mot` (les clefs sont les lettres et les valeurs leurs occurences dans `mot`).

4. Ecrire une fonction `pluslongmot(tirage,dico)` qui retourne la liste des plus longs mots de `dico` que l'on peut écrire avec les lettres de la chaîne de caractères `tirage`.

5. Avec les fonctions précédentes, écrire un programme qui effectue le tirage aléatoire d'une chaîne de caractères de longueur choisie et qui retourne la liste des plus longs mots du dictionnaire `dico.txt` qu'on peut écrire avec les lettres de cette chaîne.

Table des matières