Représentation des données : types construits

Pour une image couleur en mode "RGB" c'est à dire R pour rouge, G pour Green,(Vert) et B pour bleu, à chaque pixel est associé un tuple ou n-uplet donnant la couleur du pixel sous la forme d'un triplet de nombres entiers, les intensités de Rouge, de Vert et de Bleu Par exemple le pixel repéré par le tuple (150,50) a pour couleur (205,102,103)

```
>>> from PIL import Image

>>> image = Image.open("/Users/vallon/Documents/lena.png")

>>> image.mode
'RGB'

>>> image.getpixel((150,50))
(205, 102, 103)
```

On observe donc que les informations :

1. position d'un pixel, par exemple (150,50)
2. taille de l'image ici (512,512)
3. couleur d'un pixel par exemple (205,102,103)

sont représentés en mémoire par une collection de plusieurs entiers appelée tuple

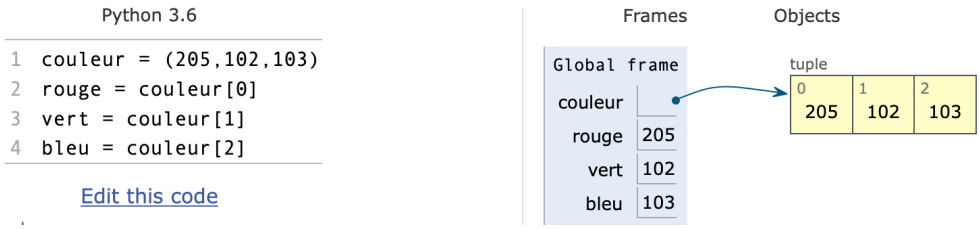
```
>>> image.size
(512, 512)

>>> taille = image.size

>>> type(taille)
<class 'tuple'>
```

1 Tuples

On va utiliser PythonTutor pour visualiser les propriétés des tuples.



1. Dans un premier temps (voir figure ci-dessus) on observe "une petite flèche" qui associe la variable **couleur** au triplet de nombres en mémoire (205,102,103). Cette "flèche" signifie *qu'à la variable couleur est associée non pas la valeur (205,102,103)* mais **une référence** c'est à dire concrètement une adresse en mémoire qui permettra de retrouver en mémoire *la valeur* du tuple (205,102,103).

   Il est important de bien comprendre cette association **variable** en tant qu'étiquette et **référence** en tant qu'adresse mémoire permettant de retrouver la **valeur** en mémoire
2. On peut avoir accès aux composantes d'un tuple, ainsi pour avoir la composante en rouge de la couleur il suffit de faire `couleur[0]` (voir ci-dessus)
3. Mais on ne peut pas changer ces valeurs on dit que le type tuple est **immuable** L'instruction `couleur[0] = 255` conduirait à une erreur
4. On veut écrire une fonction en Python qui prend en paramètres les anciennes coordonnées (x,y) d'un point et qui retourne les nouvelles coordonnées milieu de (x,y) et de (a,b) un point fixe du plan
   Une possiblité est :

```
def milieu(x,y):
    nouvellePosition = ((a + x)/2 , (b + y)/2)
    return nouvellePosition
```

   Mais en Python on peut ne pas mettre des parenthèses l'interpréteur saura interpréter un tuple

```
def milieu(x,y):
    nouvellePosition = (a + x)/2 , (b + y)/2
    return nouvellePosition
```

   que l'on peut encore simplifier

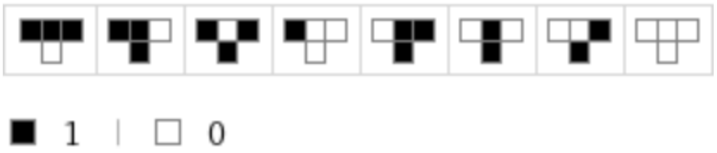
```
def milieu(x,y):
    return    (a + x)/2 , (b + y)/2
```

2 Tableaux

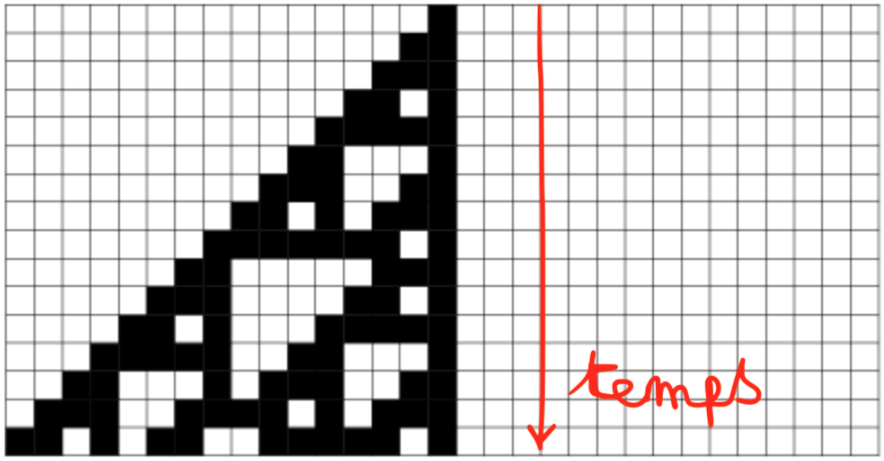
Au stade suivant **on veut à la fois avoir une collection d'entiers référencé par une seule variable mais aussi pouvoir modifier chacune des valeurs**

Ce type de structure de données est appelé généralement **tableau** en Python on appelle cette structure une **liste**

Pour bien comprendre l'intérêt des listes nous allons étudier un automate cellulaire à une dimension.
Pour commencer lire et comprendre l'article suivant `https://fr.wikipedia.org/wiki/Automate_cellulaire` (seulement la partie "les automates cellulaires les plus simples")
Nous allons nous intéresser à l'automate suivant la règle 110 schématisée ci-dessous



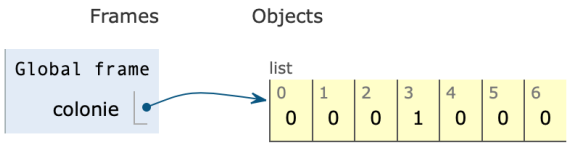
La règle s'appelle règle 110 car un état d'une cellule et de ses deux voisins correspond à un mot écrit avec trois bits donc 8 possibilités et par conséquent définir l'état futur d'une cellule correspond à énumérer toutes les possibilités sur un octet donc 256 possibilités
Voir aussi l'animation dans l'article Wikipedia suivant `https://en.wikipedia.org/wiki/Rule_110`
L'état initial de la colonie est la liste ne contenant que des 0 sauf en son "milieu"
Voici l'évolution de la colonie "au cours du temps" au bout de 15 itérations



1. **Initialisation :** On crée une liste `colonie` ne contenant que trente 0 et un 1 au "milieu"
   Comment ? S'il n' y avait eu que 7 éléments on aurait pu le faire **en extension** ainsi

```
colonie = [0,0,0,1,0,0,0]
```

   et on aurait eu



2. Mais il y a 31 éléments on risque de se tromper en procédant par extension, alors il faut procéder de manière "logique"
   Soit ainsi :

```
colonie = [] //liste vide
//on ajoute à la liste trente et un 0
for i in range(31):
    colonie.append(0)
// on affecte au seizième champ noté 15, la valeur 1
colonie[15] = 1
```

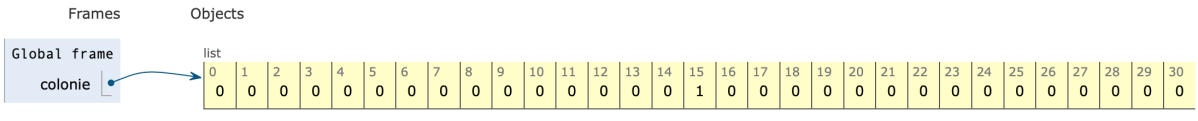
   ou encore

```
 //la liste colonie contient trente et un 0 par compréhension
colonie = [0 for i in range(31)]
// on affecte au seizième champ noté 15, la valeur 1
colonie[15] = 1
```

   ou encore

```
  //la liste colonie contient trente et un 0 par compréhension
colonie = [0] *31
// on affecte au seizième champ noté 15, la valeur 1
colonie[15] = 1
```

   on obtient alors ceci



3. **Itération**Comment créer l'état de la colonie à l'instant t + 1 à partir de l'état de la colonie à l'instant t ?
   Puisque l'état d'une cellule à l'instant t + 1 est déterminée par son état à l'instant

t ainsi que l'état de ses deux voisins à l'instant t on ne peut pas créer le nouvel état de la ccolonie uniquement avec la seule variable `colonie`
Il nous faut une deuxième variable `temp` de type liste et ayant autant de cellules que la variable `colonie` et on va procéder ainsi

Algorithme 1 : Automate cellulaire 110

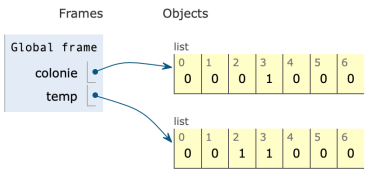
Données : La liste colonie et un nombre n
Résultat : Un entier i, la première position de la valeur dans le tableau

```
1 début
2     dessiner(colonie)
3     Répéter n fois
4       Pour chaque élément de colonie faire
5         Construire l'état suivant de élément dans temp en appliquant les règles
6       colonie ← temp
7       dessiner(colonie)
8 fin
```

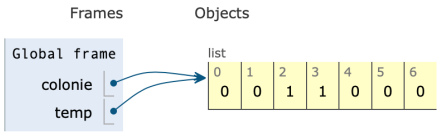
4.

5. Ce qui donne en Python

```
colonie = [0]*31
colonie[15] = 1
dessiner(colonie)
for i in range(15):
    temp = [0]*31
    for j in range(1,30):
        temp[j] = regle(colonie[j-1],colonie[j],colonie[j+1])
    colonie = temp
    dessiner(colonie)
```

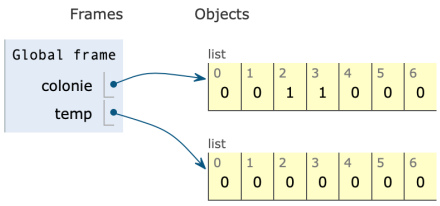


6. Voici une image de ce qui se passe à chaque tour de boucle
La variable `colonie` est associée à une référence dont la valeur contient l'état actuel de l'automate
La variable `temp` est associée à une référence dont la valeur contient l'état futur de l'automate



7. L'instruction `colonie = temp` fait que les variables `colonie` et `temp` pointent vers la même référence dont la valeur est l'état futur de la colonie

Pour faire une nouvelle génération il suffit de recréer une liste `temp` ne contenant que des 0 puis de se servir de `temp` pour construire la nouvelle génération



8. La fonction regle peut être vue comme une expression booléenne dépendant de a = colonie[j-1] , b = colonie[j] et c = colonie[j+1]
   On vérifiera en exercice que
   regle(a,b,c) est équivalente à (a et xor(b,c) ou (non a et (b ou c))

2.1 Exercices : Tuples - Listes

Ex 1

1. Ecrire en extension la liste contenant les dix chiffres de 0 à 9
2. Ecrire en compréhension la liste contenant les dix chiffres de 0 à 9
3. Ecrire en compréhension la liste contenant les carrés des entiers de 1 à 10, puis des cubes

Ex 2

Dans un jeu de 52 cartes il y a 13 hauteurs de 1 (l'as),2, 3, 4, 5, 6, 7, 8, 9, 10, 11 (valet), 12 (dame), 13 (roi)

et 4 couleurs , **H**eart (Coeur), **D**iamond (Carreau), **C**lub (Trèfle) , **S**pade (Pique)

On va représenter une carte d'un jeu de 32 cartes par un tuple par exemple la dame de Coeur sera représentée par (H,12)

Ecrire en compréhension la liste contenant les 52 tuples

Ex 3

1. Définir une fonction python `maximum(liste)` qui retourne la plus grande valeur trouvée dans la liste de nombres `liste`
   Par exemple `maximum([-2,3,4,3,1.5)` retourne 3
2. Définir une fonction python `maximum(liste)` qui retourne l'indice de la première occurence de la plus grande valeur trouvée dans la liste de nombres `liste`
   Par exemple `maximumIndice([-2,3,4,3,1.5)` retourne l'indice 1 en effet la plus grande valeur 3 se situe aux indices 1 et 3 et on retourne 1 car c'est le premier indice

Ex 4

1. Définir une fonction `occurences(sequence)` où `sequence` est une liste contenant uniquement des lettres "A", "C", "G" et "T", qui retourne la liste des occurences des lettres "A", "C", "G" et "T"
2. Définir une fonction `fréquences(sequence)` où `sequence` est une liste contenant uniquement des lettres "A", "C", "G" et "T", qui retourne la liste des fréquences des lettres "A", "C", "G" et "T"
3. La même chose mais cette fois ci `sequence` est de type `str`

Ex 5

1. Définir une fonction Python `carre(main)` qui retourne Vrai si la liste `main` de cartes d'un jeu de 52 cartes contient un carré c'est à dire 4 cartes de même hauteur (Pour cela s'aider d'une liste `occurences` ...)
2. On montre que les évènements suivants : "un carré", "un full" , "un brelan", "2 paires" et "une paire" sont classés du plus rare au moins rare
   Définir une fonction Python `valeurs(main)` qui retourne une chaîne de caractères précisant l'un des évènements ci-dessus

Ex 6

On lance un dé à six faces **tant que les chiffres de 1 à 6 ne sont pas tous sortis**

On aimerait connaître le nombre de lancers moyen pour que les six chiffres apparaissent

1. Engendrer en compréhension une liste `chiffresSortis` de 7 booléens False.
   Lorsque le 2 sort on fait chiffresSortis[2] = True
2. Définir une fonction Python `pasTousSortis(chiffresSortis)` qui retourne Vraie si tous les chiffres ne sont pas sortis et Faux sinon
3. Définir une fonction `nbLancers()` qui retourne le nombre de lancers de dé effectués pour faire apparaître tous les chiffres
4. Définir une fonction `nbLancersMoyen(nbRepetitions)` qui retourne le nombre de lancers moyen de dé effectués pour faire apparaître tous les chiffres

Ex 7

Définir une fonction python `negation(liste)` où liste est une liste de 0 et de 1 et negation(liste) retourne la liste où les 1 sont changés en 0 et les 0 en 1

Par exemple `negation([1,0,0,1,0,0])` retourne la liste [0,1,1,0,1,1]

Ex 8 : opérateur &

Le & (lire et) correspond à la fonction logique `et` mais appliqué aux représentations des nombres en binaire, bit par bit.
Ainsi par exemple 110 & 100 = 100
Tester cette fonction à la console puis définir une fonction `et(liste1,liste2)` où liste1 et liste2 sont deux listes de même longueur contenant que des 0 et des 1 et retournant la liste résultat de liste 1 & liste2

Ex 9

Définir une fonction Python `decToBin(n)` qui retourne une liste L de 0 et de 1 et telle L[i] est le chiffre de rang i de l'écriture de n en base 2

Ex 10

Définir une fonction Python `binToDec(l)` qui retourne un entier n en base 10 à partir d'une liste l de 0 et de 1 et telle l[i] est le chiffre de rang i de l'écriture de n en base 2 (Utiliser la méthode de Horner)
(la méthode de Horner consiste par exemple pour calculer $4x^3+3x^2+2x+1$ à disposer le calcul ainsi $((4\times x+3)\times x+2)\times x+1$
Sur cet exemple sans la méthode de Horner il y a 6 multiplications, avec 3 multiplications

Ex 11

Voici un programme en python

```
liste1 = [1,1]
liste2 = [0,0]
liste1[0] = 2
liste1 = liste2
print(liste1[0])
```

Qu'observe-t-on à l'écran ? (Justifier)
A :) 2 B) : 1 C) : 0

Ex 12

Voici un programme en python

```
liste1 = [0,0,0]
liste2 = [1,1,1]
liste3 = [liste1,liste2]
liste2 = liste1
print(liste3)
```

Qu'observe-t-on à l'écran ? (Justifier)

A :) [liste1,liste2] B) : [[0,0,0],[1,1,1]] C) : [[0,0,0],[0,0,0]]

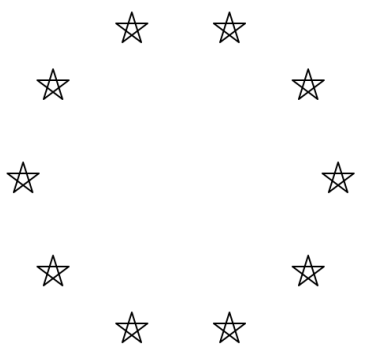
Ex 13

Etant donné une liste de points du plan par exemple `[(0,1),(1,2),(1,1),(1.2,3)]` il s'agit de trouver le point le point le "plus haut et le plus à droite possible" dans le plan

On peut comparer les tuples avec l'ordre du dictionnaire ainsi `(0,1) < (1,2)` est vrai ainsi que `(1,1) < (1,2)`

Définir une fonction `pointSupDroit(liste)` où liste est une liste de tuples correspondant à des coordonnées de points dans le plan

Ex 14

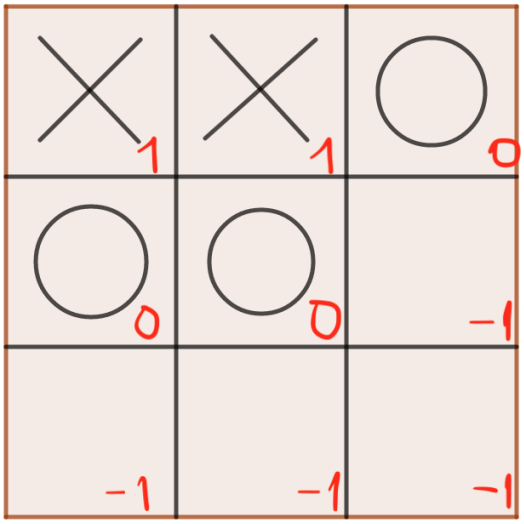


Il s'agit de faire dessiner par la Tortue dix étoiles disposées en cercle

1. Créer une liste angles en compréhension de $\frac{2k\pi}{10}$ pour k variant de 0 jusqu'à 9
2. Créer une liste de tuples en compréhension $(100\cos(a), 100\sin(a))$ pour a variant dans la liste angles
3. Finir en itérant sur la liste de tuples et dessiner une étoile

3 Tableaux à 2 dimensions

On veut simuler le jeu du Tic Tac Toe. Il faut mémoriser l'état du jeu par un tableau 2D. Que peut être un tableau 2D ? On associe 0 au rond O , 1 à la croix X et -1 pour

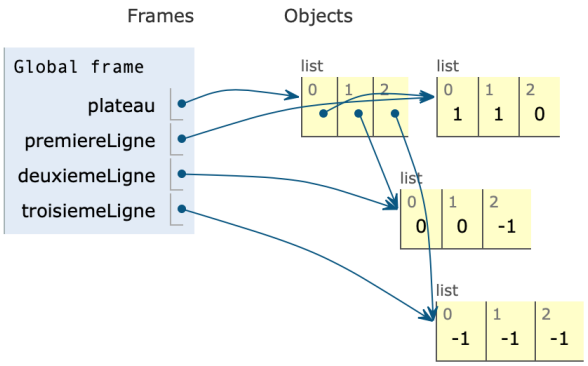


tac toe.png tac toe.png tac toe.png

une case non jouée. Une façon de faire est de "découper" le tableau ci-dessus en lignes et le tableau sera en mémoire sous la forme d'une liste de liste (en extension)

```
plateau = [[1,1,0],
           [0,0,-1],
           [-1,-1,-1]]
premiereLigne = plateau[0]
deuxiemeLigne = plateau[1]
troisiemeLigne = plateau[2]
```

Regardons avec Python tutor :



Si le joueur qui joue avec les X veut jouer sur la deuxième ligne et la troisième colonne, il va sélectionner la ligne puis la colonne, ce qui donne

```
plateau[1][2] = 1
```

3.1 Exercices : Tableaux 2D

Ex 1

Construire le tableau 2D 10 x 5 des entiers pairs de 2 à 100 **en compréhension**

Ex 2

1. Ecrire **en extension** un tableau 2D de 8 lignes et 4 colonnes pour la règle 110 de l'automate cellulaire
   `regle = [[False,False,False,False],`
   `[ [False,False,True,True],...]`
   Les quatrièmes éléments sont les résultats de la règle sur les trois premiers éléments
2. Définir une fonction `etatSuivant(numCellule,colonie,regle)` qui retourne l'état futur de la cellule `colonie[numCellule]` en suivant la règle dans le tableau `regle`

Ex 3

Voici un programme en Python :

```
regles = [[False]*4]*8
regles[0][2] = True
print(regles[1][2])
```

Qu'observe-t-on à l'écran ?

A) `True` B) `False`

Ex 4

Voici un programme en Python :

```
regles = [[False]*4 for i in range(8)]
regles[0][2] = True
print(regles[1][2])
```

Qu'observe-t-on à l'écran ?

A) `True` B) `False`

Ex 5

Ecrire une fonction `affichePlateau()` qui affiche le plateau du jeu tictactoe à partir d'un tableau 2D `plateau` contenant les caractères 'O','X', et '#' sous la forme d'un tableau 3 x 3 par exemple :

```
# X O
# X #
# # #
```

Ex 6

1. Ecrire une fonction `alignementLigne(numLigne)` qui retourne vrai s'il y a alignement sur la ligne de numéro `numLigne` du plateau du Tic Tac Toe
2. idem pour un alignement sur une des colonnes
3. idem pour un alignement sur la première et la deuxième des diagonales

4 Dictionnaires

Un dictionnaire associe à une clé **clé unique** une **valeur**
C'est une structure de données de correspondance, on parle aussi de tableau associatif
Par exemple dans le dictionnaire des variables gérées par Python , à un nom de variable est associée soit une valeur soit une référence, ainsi dans l'exemple suivant à la variable `quotient` est associée la valeur 3

```
>>> quotient = 3
>>> vars()
{'quotient': 3, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '
__package__': None, '__doc__': None, '__name__': '__main__', '__builtins__': <module 'builtins
' (built-in)>}
```

Un autre exemple :
Nous avons tous dans nos téléphones un dictionnaire où sont associés à des noms, des numéros de téléphone, par exemple :

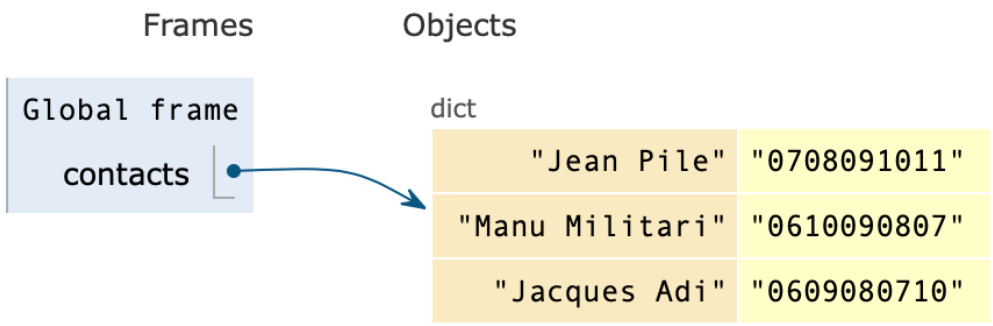
```
contacts = { "Jean Pile": "0708091011",
 "Manu Militari": "0610090807",
 "Jacques Adi ": "0609080710"}
```

Le dictionnaire `contacts` ci-dessus a été défini en extension mais on aurait pu à partir d'un dictionnaire vide ajouter des éléments au fur et à mesure, par exemple

```
contacts = { }
contacts["Jean Pile"] = "0708091011"
contacts["Manu Militari"] = "0610090807"
contacts["Jacques Adi"] = "0609080710"
```

Il n'y a pas de structure d'ordre dans un dictionnaire d'ailleurs à l'affichage les éléments n'apparaissent pas forcément dans l'ordre d'insertion dans le dictionnaire. Ainsi par exemple

```
>>> contacts = {}
>>> contacts["Jean Pile"] = "0708091011"
>>> contacts["Manu Militari"] = "0610090807"
>>> contacts["Jacques Adi"] = "0609080710"
>>> contacts
{'Jean Pile': '0708091011', 'Jacques Adi': '0609080710', 'Manu Militari': '0610090807'}
```



Quelles sont les principales opérations que l'on peut faire sur un dictionnaire ?

1. Se demander si une clé se trouve dans le dictionnaire

```
>>>"Jean Pile" in contacts
True
```

ou

```
>>>"Jean Pile" in contacts.keys()
True
```

contacts est un **objet** et `keys()` est une fonction spécifique ou méthode de l'objet `contacts` et exercer une certaine action sur un objet par l'intermédiaire d'une méthode `f(...)` s'écrit `objet.f(...)`
La méthode `keys()` appliquée à l'objet `contacts` retourne les clés du dictionnaire `contacts` sous forme de liste

2. Supprimer une clé et sa valeur

```
>>>del contacts["Jean Pile"]
```

3. Parcourir la liste des clés pour faire un traitement sur les valeurs associées, par exemple afficher les noms des numéros commençant par "06"

```
for cle in contacts.keys():
    if contacts[cle][1] == "6":
        print(cle," a pour numéro de téléphone : ",
        contacts[cle])
```

En effet `contacts[cle]` est une chaîne de caractères et `contacts[cle][1]` est le deuxième caractère de cette chaîne de caractères

4. Il existe aussi la méthode `values()` qui appliquée à l'objet `contacts` retourne la liste des valeurs. On n' aurait pu afficher que les numéros de téléphone commençant par 06 **sans leur antécédents** c'est à dire les noms

```
for val in contacts.values():
    if val[1] == "6":
        print(val)
```

5. Il existe aussi la méthode `items()` qui appliquée à l'objet `contacts` retourne la liste des tuples (clés,valeurs)

```
for couple in contacts.items():
    if couple[1][1] == "6":
        print(couple[0]," a pour numéro de téléphone : ",
        couple[1])
```

Ici couple est un tuple dont le premier élément couple[0] est une clé et le deuxième élément coulple[1] est une valeur de type chaîne de caractères.
couple[1][1] est le deuxième caractère de cette chaîne

4.1 Exercices : Dictionnaires

Ex 1

Le serveur DNS (Dynamic Name System) d'authentification confirme l'association d'un nom de domaine à une adresse IP (Internet Protocol)
On dispose du dictionnaire suivant :

```
dns = {"fr.wikipedia.org":"91.198.174.192",
"mathly.fr":"85.236.158.88",
"gallerianazionale.com":"94.130.217.148"}
```

1. Qu'affiche l'instruction `print(dns["mathly.fr"])` ?
2. Qu'affiche l'instruction `len(dns.keys())` ?

Ex 2

On dispose d'un dictionnaire `volcans` qui associe à des noms de volcans (de type str) des couples (type,date) où type est le type du volcan, le caractère "E" pour explosif ou "e" pour effusif et date la date de la dernière éruption connue.
Par exemple l'instruction `print(volcans["Piton de la Fournaise"])` a affiché `("e","01/11/2019")`
On dit que le Piton de la Fournaise est en cours d'éruption

1. Définir une fonction `volcansEffusifs(volcans)` qui à partir du dictionnaire `volcans` retourne la liste des volcans effusifs
2. Définir une fonction `volcansEnCoursEruption(volcans)` qui à partir du dictionnaire `volcans` retourne la liste des volcans en cours d'éruption

Ex 3

A chaque nom de polygone régulier (type str) on lui associe le couple (type tuple) constitué du nombre de sommets (type int) et du nombre de diagonales (type int)
On dispose d'un dictionnaire `nbDiagonales`
Par exemple l'instruction `print(nbDiagonales["carré"])` a affiché `(4,2)`

1. On aimerait trouver une loi qui relie le nombre de sommets (ou de côtés) du polygone et le nombre de diagonales
   Pour cela construire une liste `nbDiag` à partir du dictionnaire `nbDiagonales` en parcourant les valeurs de ce dictionnaire telle que
   `nbDiag[n]` est le nombre de diagonales d'un polygone à `n` sommets et
   `nbDiag[n]` $= 0$ si $0 \leqslant n \leqslant 3$
   Définir une fonction `creeListe(nbDiagonales)` qui crée la liste `nbDiag` à partir du dictionnaire `nbDiagonales`
2. Etant donnée une liste `liste1` on définit la liste des variations `liste2` telle que `liste2[n] = liste1[n] - liste1[n-1]` et `liste2[0] =0` Définir une fonction `creeListeVariations(liste)` qui retourne la liste des variations de `liste`
3. Les valeurs de la liste des variations de la liste `nbDiag` nous emmènent à penser que la loi qui relie le nombre de sommets (ou de côtés) du polygone et le nombre de diagonales est un trinôme $d(n) = an^2 + bn$ avec $d(3) = 0$ et $d(4) = 2$. Quelle est cette loi ?
4. Utiliser cette loi pour engendrer **par compréhension** le dictionnaire `nbDiagonales`

Problème :

Une expression littérale étant donnée par exemple sous cette forme '5 - 2' c'est à dire une chaîne de caractères représentant un nombre **puis un espace** puis un symbole parmi + - * / **puis un espace** puis à nouveau une chaîne de caractères représentant un nombre, il s'agit **d'évaluer** cette expression littérale c'est à dire de lui attribuer la valeur numérique correspondante, ici 3

On va utiliser la fonction `split()` qui appliquée à une chaîne de caractères contenant

```
>>> expression = '5 - 2'
>>> elt1, op, elt2 = expression.split(' ')
>>> type(expression.split(' '))
<class 'list'>
>>> print(elt1)
5
>>> type(elt1)
<class 'str'>
```

un séparateur ici l'espace, "coupe" cette chaîne selon le séparateur et les éléments sont des chaînes de caractères insérés dans une liste

On utilise un dictionnaire pour **associer** au symbole une **fonction** qui permet ensuite d'évaluer l'expression

```
def somme(x,y):
    return x + y

def difference(x,y):
    return x - y

def mult(x,y):
    return x*y

def div(x,y):
    if y == 0:
        return 'infini'
    else:
        return x/y

operateurs = {'+' : somme,
              '-' : difference,
              '*' : mult,
              '/' : div}

def evaluation(expression):
    elt1, op, elt2 = expression.split(' ')
    operateur = operateurs[op]
    return operateur(int(elt1),int(elt2))
```

```
#----------------main----------------------
expression = '6 + 7'
print(evaluation(expression))
```

Le nom d'une fonction est une **variable** associée à une **référence**, on voit cette association ci-dessous dans le dictionnaire `vars()`

```
>>> def somme(x,y):
        return x + y

>>> def difference(x,y):
        return x - y

>>> def mult(x,y):
        return x*y

>>> def div(x,y):
        if y == 0:
                return 'infini'
        else:
                return x/y
```

```
>>> vars()
{'mult': <function mult at 0x1129e3510>, '__package__': None, 'div': <function div at 0x1129e3598>, '__do
__': None, 'difference': <function difference at 0x101e4cbf8>, 'somme': <function somme at 0x10ffce510>,
__name__': '__main__', '__builtins__': <module 'builtins' (built-in)>, '__loader__': <class '_frozen_impo
tlib.BuiltinImporter'>, '__spec__': None}
```

On retrouve cette association dans le dictionnaire `operateurs` que l'on peut créer dynamiquement en créant chaque association en autant d'instructions

```
>>> operateurs = {}
>>> operateurs['+'] = somme
>>> operateurs['-'] = difference
>>> print(operateurs)
{'+': <function somme at 0x10ffce510>, '-': <function difference at 0x101e4cbf8>
>>> operateurs['+'](7,5)
12
```

```
>>> operateurs['*'] = mult
>>> operateurs['/'] = div
>>> operateurs.keys()
dict_keys(['+', '/', '*', '-'])
>>> operateurs.values()
dict_values([<function somme at 0x10ffce510>, <function div at 0x1129e3598>, <function mult at 0x1129e351
>, <function difference at 0x101e4cbf8>])
>>> operateurs.items()
dict_items([('+', <function somme at 0x10ffce510>), ('/', <function div at 0x1129e3598>), ('*', <function
mult at 0x1129e3510>), ('-', <function difference at 0x101e4cbf8>)])
```