1) On peut créer cette liste de deux façons : en ajoutant les couples voulus à une liste vide à l'aide de deux boucles, ou bien en utilisant une compréhension de liste.

```
Cases=[]
for i in range(8):
    for j in range(8):
        Cases.append((i,j))

# La compréhension de listes, c'est la grande classe !
Cases=[(i,j) for i in range(8) for j in range(8)]
```

2) Là aussi, on peut procéder de deux façons : voir directement si les indices sont bien entre 0 et 7, ou bien regarder si le couple proposé est dans la liste `Cases`

```
# Pas besoin de faire un tas de tests imbriqués :
# il suffit de conjuguer les 4 conditions requises
def Valide(i,j):
    return i>=0 and i<=7 and j>=0 and j<=7

# Autant utiliser la liste de la question précédente
def Valide(i,j):
    return (i,j) in Cases
```

3) L'idée est d'ajouter tous les déplacements relatifs possibles à notre position `(i,j)`, puis de ne retenir parmi les positions obtenues que celles qui font partie de la liste `Cases`.

```
def CoupSuivant(i,j):
    # Ensemble des positions obtenues
    Positions=set()

    # On crée la liste de tous les déplacements relatifs possibles
    depl=[(2,1),(2,-1),(1,2),(1,-2),(-1,2),(-1,-2),(-2,1),(-2,-1)]

    # Maintenant, on ajoute tous ces déplacements à (i,j) et
    # on regarde si la position obtenue est sur l'échiquier
    for (a,b) in depl:
        if Valide(i+a,j+b):
            Positions.add((i+a,j+b))

    return Positions
```

4) En suivant l'indication, ça marche plutôt bien. Dans la fonction suivante, `Positions` est l'ensemble des positions accessibles en n coups à partir de la position initiale.

```
def Minimum(i0,j0,i,j):
    # Visites : ensembles des positions déjà explorées
    Visites={(i0,j0)}

    # Positions : ensemble des positions accessibles en n coups
    n=0
    Positions={(i0,j0)}

    # Tant que cet ensemble ne contient pas (i,j), on passe au
    # n+1-ième coup en réunissant toutes les positions accessibles
    # en un coup à partir de chacune des positions atteintes en
    # n coups

    while (i,j) not in Positions:
        n=n+1

        # Ensemble des cases découvertes depuis celles de Positions
        PosNouv=set()

        for (a,b) in Positions:
            PosNouv=PosNouv.union(CoupSuivant(a,b))

        # Pour accélerer l'exploration, on élimine les positions
        # déjà explorées puis on met à jour l'ensemble Visites
        Positions=PosNouv.difference(Visites)
        Visites=Visites.union(Positions)

    # En fin de boucle, n contient la valeur recherchée
    return n
```

5) Là, c'est assez simple, on a fait le plus dur dans les deux questions qui précèdent.

```
from numpy import array

def Cavalier(i0,j0):
    # On imbrique deux compréhensions de listes pour créer
    # une liste de listes, que l'on transforme en matrice
    # à l'aide de la commande array()
    M=[[Minimum(i0,j0,i,j) for j in range(8)] for i in range(8)]
    return array(M)
```

6) On va reprendre la fonction `Minimum` en utilisant un dictionnaire `Antecedents` dont les clefs sont les positions atteintes et les valeurs leurs antécédents immédiats.

On le mettra à jour au cours de l'exploration, en même temps que les ensembles `Visites` et `Positions`, en procédant cette fois-ci élément par élément.

À la fin, lorsque la case `(i,j)` a été atteinte, on utilise le dictionnaire pour reconstituer le chemin recherché en partant de la fin ... d'où l'addition de liste à gauche.

```
def Chemin(i0,j0,i,j):
    Antecedents=dict()

    # Visites : ensembles des positions déjà explorées
    Visites={(i0,j0)}

    # Positions : ensemble des positions atteintes au dernier coup
    Positions={(i0,j0)}

    while (i,j) not in Positions:
        # Ensemble des cases découvertes depuis celles de Positions
        PosNouv=set()

        for (a,b) in Positions:
            # Ensemble des cases découvertes depuis (a,b)
            PosAdd=CoupSuivant(a,b).difference(Visites)

            # Mise à jour de Visites, PosNouv et Antecedents
            for (c,d) in PosAdd:
                Visites.add((c,d))
                PosNouv.add((c,d))
                Antecedents[(c,d)]=(a,b)

        Positions=PosNouv

    # La case (i,j) ayant été atteinte, on reconstitue le chemin
    chemin=[(i,j)]
    while chemin[0]!=(i0,j0):
        chemin=[Antecedents[chemin[0]]]+chemin

    return chemin
```

7) Le nombre de coups nécessaires est en fait l'indice de la case dans la liste `chemin`.

```
def Visualisation(i0,j0,i,j):
    chemin=Chemin(i0,j0,i,j)
    M=array([['X' for j in range(8)] for i in range(8)])
    for n in range(len(chemin)):
        (a,b)=chemin[n]
        M[a,b]=str(n)
    return M
```

8) Pour la première fonction, on va utiliser un dictionnaire pour associer des chiffres aux lettres A, B, ..., H. On aurait pu également rechercher leur indice dans le mot `'ABCDEFGH'` mais c'est plus long : pour ce genre d'opérations, il n'y pas mieux que les dictionnaires.

   Pour la seconde fonction, par contre, on peut facilement associer à un entier `i` la lettre d'indice `i` dans le mot `'ABCDEFGH'`.

```
dico={'A' : 0, 'B' : 1, 'C' : 2, 'D' : 3, 'E' : 4, 'F' : 5,
      'G' : 6, 'H' : 7}

def MotCase(ch):
    i=8-eval(ch[1])
    j=dico[ch[0]]
    return (i,j)

code='ABCDEFGH'

def CaseMot(i,j):
    mot=code[j]+str(8-i)
    return mot
```

9) On va se contenter de créer deux nouvelles fonctions qui utilisent les fonctions `Chemin` et `Visualisation` ainsi que les deux fonctions précédentes pour passer des couples aux mots et réciproquement.

```
def Chemin2(mot0,mot):
    # On calcule les cases associées aux mots
    (i0,j0)=MotCase(mot0)
    (i,j)=MotCase(mot)

    # On récupère le chemin via Chemin
    chemin=Chemin(i0,j0,i,j)

    # On convertit les positions en mots
    chemin2=[CaseMot(i,j) for (i,j) in chemin]

    return chemin2


def Visualisation2(mot0,mot):
    # On calcule  les cases associées aux mots
    (i0,j0)=MotCase(mot0)
    (i,j)=MotCase(mot)

    # Ensuite on lance Visualisation
    return Visualisation(i0,j0,i,j)
```