BACCALAURÉAT GÉNÉRAL

ÉPREUVE BLANCHE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2022

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice et du dictionnaire n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.
Ce sujet comporte 12 pages numérotées de 1 à 12.

Le candidat traite au choix 3 exercices parmi les 4 exercices proposés

Exercice 1. 6 pts

Notions abordées : structures de données (dictionnaires)

Une ville souhaite gérer son parc de vélos en location partagée. L'ensemble de la flotte de vélos est stocké dans une table de données représentée en langage Python par un dictionnaire contenant des associations de type `id_velo : dict_velo` où `id_velo` est un nombre entier compris entre 1 et 199 qui correspond à l'identifiant unique du vélo et `dict_velo` est un dictionnaire dont les clés sont : `"type"`, `"etat"`, `"station"`.

Les valeurs associées aux clés `"type"`, `"etat"`, `"station"` de `dict_velo` sont de type chaînes de caractères ou nombre entier :

— `"type"` : chaîne de caractères qui peut prendre la valeur `"electrique"` ou `"classique"` ;

— `"état"` : nombre entier qui peut prendre la valeur 1 si le vélo est disponible, 0 si le vélo est en déplacement, -1 si le vélo est en panne ;

— `"station"` : chaînes de caractères qui identifie la station où est garé le vélo.

Dans le cas où le vélo est en déplacement ou en panne, "station" correspond à celle où il a été dernièrement stationné.

Voici un extrait de la table de données :

```
flotte = {
    12 : {"type" : "electrique", "etat" : 1, "station" : "Prefecture"},
    80 : {"type" : "classique", "etat" : 0, "station" : "Saint-Leu"},
    45 : {"type" : "classique", "etat" : 1, "station" : "Baraban"},
    41 : {"type" : "classique", "etat" : -1, "station" : "Citadelle"},
    26 : {"type" : "classique", "etat" : 1, "station" : "Coliseum"},
    28 : {"type" : "electrique", "etat" : 0, "station" : "Coliseum"},
    74 : {"type" : "electrique", "etat" : 1, "station" : "Jacobins"},
    13 : {"type" : "classique", "etat" : 0, "station" : "Citadelle"},
    83 : {"type" : "classique", "etat" : -1, "station" : "Saint-Leu"},
    22 : {"type" : "electrique", "etat" : -1, "station" : "Joffre"}
}
```

`flotte` étant une variable globale du programme.

Toutes les questions de cet exercice se réfèrent à l'extrait de la table `flotte` fourni ci-dessus. L'**annexe 1** présente un rappel sur les dictionnaires en langage Python.

1.

1.1. Que renvoie l'instruction `flotte[26]` ? 0,5 pts

{"type" : "classique", "etat" : 1, "station" : "Coliseum"}

1.2. Que renvoie l'instruction `flotte[80]["etat"]` ? 0,5 pts

0

1.3. Que renvoie l'instruction `flotte[99]["etat"]` ? 0,5 pts

Une erreur car il n'y a pas de clé 99.

2. Voici le script d'une fonction :

```
def proposition(choix):
    for v in flotte:
        if flotte[v]["type"] == choix and flotte[v]["etat"] == 1:
            return flotte[v]["station"]
```

2.1. Quelles sont les valeurs possibles de la variable `choix` ? 0,5 pts

La valeur `choix` doit être une valeur possible pour le type du vélo, donc : « électrique » et « classique ».

2.2. Expliquer ce que renvoie la fonction lorsque l'on choisit comme paramètre l'une des valeurs possibles de la variable `choix`. 0,5 pts

La fonction renvoie le nom de la première station à disposer d'un vélo disponible du type choisi.

3.

3.1. Écrire un script en langage Python qui affiche les identifiants `(id_velo)` de tous les vélos disponibles à la station `"Citadelle"`. 1 pt

```
for v in flotte :

    if flotte[v]["station"] == "Citadelle" and flotte[v]["etat"] == 1 :

        print(v)
```

3.2. Écrire un script en langage Python qui permet d'afficher l'identifiant `(id_velo)` et la station de tous les vélos électriques qui ne sont pas en panne. 1 pt

```
for v in flotte :

    if flotte[v]["type"] == "electrique" and flotte[v]["etat"] != -1 :

        print(v,flotte[v]["station"])
```

4. On dispose d'une table de données des positions GPS de toutes les stations, dont un extrait est donné ci-dessous. Cette table est stockée sous forme d'un dictionnaire.

Chaque élément du dictionnaire est du type : `"nom de la station" : (latitude, longitude)`.

```
stations = {
    "Prefecture" : (49.8905, 2.2967),
    "Saint-Leu" : (49.8982, 2.3017),
    "Coliseum" : (49.8942, 2.2874),
    "Jacobins" : (49.8912, 2.3016)
}
```

On admet que l'on dispose d'une fonction `distance(p1,p2)` permettant de renvoyer la distance en mètres entre deux positions données par leurs coordonnées GPS (latitude et longitude). Cette fonction prend en paramètre deux tuples représentant les coordonnées des deux positions GPS et renvoie un nombre entier représentant cette distance en mètres. Par exemple, `distance((49.8905, 2.2967), (49.8912, 2.3016))` renvoie 9591.

Écrire une fonction qui prend en paramètre les coordonnées GPS de l'utilisateur sous forme d'un tuple et qui renvoie, pour chaque station située à moins de 800 mètres de l'utilisateur :

— le nom de la station ;
— la distance entre l'utilisateur et la station ;
— les identifiants des vélos disponibles dans cette station.

Une station où aucun vélo n'est disponible ne doit pas être affichée. 1,5 pts

```
def velo_dispo(station : str) -> list :
    liste_velos_dispo = []
    for v in flotte :
        if flotte[v]["station"] == station and flotte[v]["etat"] == 1 :
            liste_velos_dispo.append(v)
    return liste_velos_dispo

def propositions(coord_utilisateur : tuple) -> list :
    liste_propositions = []
    for s in stations :
        dist_u_s = distance(coord_utilisateur, stations[s])
        if dist_u_s <= 800 :
            liste_velos_dispo = velo_dispo(s)
            if len(liste_velos_dispo) > 0 :
                liste_propositions.append([s, dist_u_s, liste_velos_dispo])
    return liste_propositions
```

Exercice 2. 6 pts

Notions abordées : manipulation de tableaux, récursivité et paradigme « diviser pour régner ».

Dans un tableau Python d'entiers `tab`, on dit que le couple d'indices `(i,j)` forme une inversion lorsque `i<j` et `tab[i] > tab[j]`. On donne ci-dessous quelques exemples.

— Dans le tableau `[1,5,3,7]`, le couple d'indices `(1,2)` forme une inversion car `5 > 3`. Par contre, le couple `(1,3)` ne forme pas d'inversion car `5 < 7`. Il n'y a qu'une inversion dans ce tableau.
— Il y a trois inversions dans le tableau `[1,6,2,7,3]`, à savoir les couples d'indices `(1,2)`, `(1,4)` et `(3,4)`.
— On peut compter six inversions dans le tableau `[7,6,5,3]` : les couples d'indices `(0,1)`, `(0,2)`, `(0,3)`, `(1,2)`, `(1,3)` et `(2,3)`.

Questions préliminaires

1. Expliquer pourquoi le couple `(1,3)` est une inversion dans le tableau `[4,8,3,7]`. 0,5 pts

Car, en notant `t` le tableau ci-dessus, on a `t[1]>t[3]`.

2. Justifier que le couple `(2,3)` n'en est pas une. 0,5 pts

Car on a `t[2]<t[3]`.

Partie A : méthode itérative

Le but de cette partie est d'écrire une fonction itérative `nombre_inversion` qui renvoie le nombre d'inversions dans un tableau. Pour cela, on commence par écrire une fonction `fonction1` qui sera ensuite utilisée pour écrire la fonction `nombre_inversion`.

1. On donne la fonction suivante.

```
def fonction1(tab, i):
    nb_elem = len(tab)
    cpt = 0
    for j in range(i+1, nb_elem):
        if tab[j] < tab[i]:
            cpt += 1
    return cpt
```

1.1. Indiquer ce que renvoie la `fonction1(tab,i)` dans les cas suivants. 0,75 pts

Cas 1 : `tab=[1,5,3,7]` et `i=0`. 0

Cas 2 : `tab=[1,5,3,7]` et `i=1`. 1

Cas 3 : `tab=[1,5,2,6,4]` et `i=1`. 2

1.2. Expliquer ce que permet de déterminer cette fonction. 0,5 pts

Elle compte le nombre d'inversions dans le tableau à partir d'un indice donné.

2. En utilisant la fonction précédente, écrire une fonction `nombre_inversion(tab)` qui prend en argument un tableau et renvoie le nombre d'inversions dans ce tableau. On donne ci-dessous les résultats attendus pour certains appels. 1 pt

```
>>> nombre_inversions([1, 5, 7])
0
>>> nombre_inversions([1, 6, 2, 7, 3])
3
>>> nombre_inversions([7, 6, 5, 3])
6
```

```
def nombre_inversions(liste : list) -> int :

    compteur = 0

    for i in range(len(liste)):

        compteur += fonction1(liste, i)

    return compteur
```

3. Quelle est l'ordre de grandeur de la complexité en temps de l'algorithme obtenu ? Aucune justification n'est attendue. 0,25 pts

Taille de la liste au carré.

Partie B : méthode récursive

Le but de cette partie est de concevoir une version récursive de la fonction `nombre_inversion`. On définit pour cela des fonctions auxiliaires.

1. Donner le nom d'un algorithme de tri ayant une complexité meilleure que quadratique. 0,25 pts

Tri fusion ou tri rapide $n\ln(n)$ où n est la taille de la liste.

Dans la suite de cet exercice, on suppose qu'on dispose d'une fonction `tri(tab)` qui prend en argument un tableau et renvoie un tableau contenant les mêmes éléments rangés dans l'ordre croissant.

2. Écrire une fonction `moitie_gauche(tab)` qui prend en argument un tableau `tab` et renvoie un nouveau tableau contenant la moitié gauche de `tab`. Si le nombre d'éléments de `tab` est impair, l'élément du centre se trouve dans cette partie gauche.

On donne ci-dessous les résultats attendus pour certains appels.

```
>>> moitie_gauche([])
[]
>>> moitie_gauche([4, 8, 3])
[4, 8]
>>> moitie_gauche ([4, 8, 3, 7])
[4, 8]
```

```
def moitie_gauche(liste : list) -> list :

    n = len(liste)

    if n % 2 == 0 :

        return liste[:n//2]

    else :

        liste[:n//2+1]
```

Dans la suite, on suppose qu'on dispose de la fonction `moitie_droite(tab)` qui renvoie la moitié droite sans l'élément du milieu.

3. On suppose qu'une fonction `nb_inv_tab(tab1,tab2)` a été écrite. Cette fonction renvoie le nombre d'inversions du tableau obtenu en mettant bout à bout les tableaux `tab1` et `tab2`, à condition que `tab1` et `tab2` soient tr_velos_dispo.append(v)iés dans l'ordre croissant.

On donne ci-dessous deux exemples d'appel de cette fonction :

```
>>> nb_inv_tab([3, 7, 9], [2, 10])
3
>>> nb_inv_tab([7, 9, 13], [7, 10, 14])
3
```

En utilisant la fonction `nb_inv_velos_dispo.append(v)_tab` et les questions précédentes, écrire une fonction récursive `nb_inversions_rec(tab)` qui permet de calculer le nombre d'inversions dans un tableau. Cette fonction renverra le même nombre que `nombre_inversions(tab)` de la partie A. On procédera de la façon suivante :

— Séparer le tableau en deux tableaux de tailles égales (à une unité près).

— Appeler récursivement la fonction `nb_inversions_rec` pour compter le nombre d'inversions dans chacun des deux tableaux.
— Trier les deux tableaux (on rappelle qu'une fonction de tri est déjà définie).
— Ajouter au nombre d'inversions précédemment comptées le nombre renvoyé par la fonction `nb_inv_tab` avec pour arguments les deux tableaux triés.

```
def nb_inversions_rec(liste : list) -> int :

    if len(liste) == 1 :

        return 0

    else :

        gauche = moitie_gauche(liste)
        droite = moitie_droite(liste)

        nb_inv_g = nb_inversions_rec(gauche)
        nb_inv_d = nb_inversions_rec(droite)

        gauche = trier(gauche)
        droite = trier_droite(droite)

        return nb_inv_g + nb_inv_d + nb_inversions_tab(gauche,droite)
```

Exercice 3. 6 pts

Notions abordées : structure de données (programmation objet) et langages et programmation (spécification).

Une entreprise fabrique des yaourts qui peuvent être soit nature (sans arôme), soit aromatisés (fraise, abricot ou vanille). Pour pouvoir traiter informatiquement les spécificités de ce produit, on va donc créer une classe Yaourt qui possédera un certain nombre d'attributs :

— Son genre : nature ou aromatisé ;
— Son arôme : fraise, abricot, vanille ou aucun ;
— Sa date de durabilité minimale (DDM) exprimée par un entier compris entre 1 et 365 (on ne gère pas les années bissextiles). Par exemple, si la DDM est égale à 15, la date de durabilité minimale est le 15 janvier.

On va créer également des méthodes permettant d'interagir avec l'objet Yaourt pour attribuer un arôme ou récupérer un genre par exemple. On peut représenter cette classe par le tableau de spécifications ci-dessous :

Yaourt	
Attributs	**Méthodes**
`genre`	`construire(arome,duree)`
`arome`	`obtenir_arome()`
`duree`	`obtenir_genre()`
	`obtenir_duree()`
	`attribuer_arome()`
	`attribuer_genre()`
	`attribuer_duree()`

1. La classe `Yaourt` est déclarée en Python à l'aide du mot-clé `class` :

```
class Yaourt:
    """ Classe définissant un yaourt caractérisé par :
    - son arome
    - son genre
    - sa durée de durabilité minimale"""
```

L'**annexe 2** donne le code existant et l'endroit des codes à produire dans les questions suivantes.

1.1. Quelles sont les assertions à prévoir pour vérifier que l'arôme et la durée correspondent bien à des valeurs acceptables ? Il faudra aussi expliciter les commentaires qui seront renvoyés. 1 pt

Pour rappel :

— L'arôme doit prendre comme valeur « fraise », « abricot », « vanille » ou « aucun ».
— Sa date de durabilité minimale (DDM) est une valeur positive.

```
assert arome in ["fraise", "abricot", "vanille", "aucun"], "L'arôme entré n'est pas
valide."

assert type(duree) == int, "La durée (DDM) entrée n'est pas un nombre entier."
assert 1 <= duree <= 365, "La durée (DDM) entrée n'est pas un nombre compris entre
1 et 365 (inclus)."
```

1.2. Pour créer un yaourt, on exécutera la commande suivante :

```
mon_yaourt = Yaourt('fraise',24)
```

Quelle valeur sera affectée à l'attribut genre associé à `mon_yaourt` ? 0,5 pts

L'attribut arome du yaourt aura la valeur "fraise" donc le genre aura la valeur "aromatise".

1.3. Écrire en python une fonction `get_arome(self)`, renvoyant l'arôme du yaourt créé. 0,5 pts

```
def get_arome(self) :

    return self.__arome
```

2. On appelle mutateur une méthode permettant de modifier un ou plusieurs attributs d'un objet. Sur votre copie, écrire en Python le mutateur `set_arome(self,arome)` permettant de modifier l'arôme du yaourt. 1 pt

On veillera à garder une cohérence entre l'arôme et le genre.

```
def set_arome(self, arome) :

    self.__arome = arome

    if arome == "aucun" :

        self.__genre = "aucun"

    else :

        self.__genre = "aromatise"
```

3. On veut créer une pile contenant le stock de yaourts. Pour cela il faut tout d'abord créer une pile vide :

```
def creer_pile():
    pile = []
    return pile
```

3.1. Créer une fonction `empiler(p,Yaourt)` qui renvoie la pile `p` après avoir ajouté un objet de type `Yaourt` à la fin. 1 pt

```
def empiler(p : list, Yaourt : class) -> list :
    p.append(Yaourt)
    return p
```

3.2. Créer une fonction `depiler(p)` qui renvoie l'objet à dépiler. 1 pt

```
def depiler(p : list) -> list, class :
    if est_vide(p) : # On utilise la fonction définie à la question suivante
        return None
    else :
        return p.pop()
```

3.3. Créer une fonction `est_vide(p)` qui renvoie `True` si la pile est vide et `False` sinon. 0,5 pts

```
def est_vide(p : list) -> bool :
    if p == [] :
        return True
    else :
        return False
```

3.4. Qu'affiche le bloc de commandes suivantes ci-dessous ? 0,5 pts

```
mon_yaourt1 = Yaourt('aucun',18)
mon_yaourt2 = Yaourt('fraise',24)
ma_pile = creer_pile()
empiler(ma_pile, mon_yaourt1)
empiler(ma_pile, mon_yaourt2)
print(depiler(ma_pile).get_duree())
print(est_vide(ma_pile))
```

Ces commandes renvoient la DDM du second yaourt (qui était au sommet de la pile et qui a été dépilé) donc 24 puis le résultat de la fonction `est_vide()`, i.e. `False`. En effet, la pile n'est pas vide puisqu'il reste le premier yaourt.

Exercice 4.

Notions abordées : bases de données (modèle relationnel, base de données relationnelle et langage SQL).

Dans notre monde, l'information a de plus en plus de valeur et d'importance mais nous sommes de plus en plus confrontés à l'infobésité.

Considérons l'utilisation des données issues de la table de Mendeleïev (tableau périodique des éléments). Il est contraignant de faire des recherches sur des moteurs dédiés à chaque fois qu'une valeur est nécessaire (masse volumique, rayon de covalence, point de fusion...).

Les lignes 3, 4 et 5 de cette table Mendeleïev ont permis de construire, en **annexe 3**, une base de données des différents atomes correspondants.

1. Donner le nom du langage informatique utilisé pour accéder aux données dans une base de données. 0,5 pts

Le SQL.

2.

2.1. Lister les différents attributs des tables `ATOMES` et `VALENCE` en précisant le type du domaine de chacun. 0,5 pts

ATOMES
Z : int
Nom : str
Sym : str
L : int
C : str
Masse_Atom : float

VALENCE
Col : int
Couche : str

2.2. Déterminer si des attributs de la table `ATOMES` peuvent avoir un rôle de clé primaire et / ou de clé étrangère. Justifier. 0,5 pts

Z, Nom et Sym peuvent être des clés primaires car ces attributs identifient de façon unique chaque enregistrement de la table.

C est une clé étrangère car cet attribut fait référence à la clé primaire Col de la table VALENCE.

2.3. Donner le schéma relationnel pour les deux tables `ATOMES` et `VALENCE`. 0,5 pts

3. Donner les réponses des deux requêtes suivantes :

3.1. `SELECT Nom FROM ATOMES WHERE L='3' ORDER BY Sym` 0,5 pts

Aluminium, Argon, Chlore, Magnésium, Sodium, Phosphore, Souffre, Silicium. Attention, ici on ordonne par le symbole.

3.2. `SELECT DISTINCT C FROM ATOMES` 0,5 pts

1, 2 (IIA), 13 (IIIA), 14, 15 (VA), 16 (VIA), 17 (VII), 18 (VIIIA).

4. Donner la requête SQL :

4.1. Pour afficher le nom et la masse atomique des atomes. 1 pt

```
SELECT Nom, Mass_Atom FROM ATOMES
```

4.2. Pour afficher le symbole des atomes dont la couche de valence est s. 1 pt

```
SELECT Sym
FROM ATOMES
INNER JOIN VALENCE
ON ATOMES.C = VALENCE.Col
WHERE Couche = s
```

5. On a remarqué une erreur de saisie dans la table `ATOMES`, la masse atomique de l'argon (Ar) n'est pas 29,948 g.mol^{-1} mais 39,948 g.mol^{-1}. Écrire la requête SQL pour corriger cette erreur de saisie. 1 pt

```
UPDATE ATOMES
SET Mass_Atom = 39,948
WHERE Sym = "Ar"
```

Annexe 1
(à ne pas rendre avec la copie)

Action	**Instruction et syntaxe**
Créer un dictionnaire vide	`dico={}`
Obtenir un élément d'un dictionnaire existant à partir de sa clé	`dico[cle]`
Modifier la valeur d'un élément d'un dictionnaire à partir de sa clé	`dico[cle]=nouvelle_valeur`
Ajouter un élément dans un dictionnaire existant	`dico[nouvelle_cle]=valeur`
Supprimer et obtenir un élément d'un dictionnaire à partir de sa clé	`dico.pop(cle)`
Tester l'appartenance d'un élément à un dictionnaire (renvoie un booléen)	`cle in dico`
Objet itérable contenant les clés (ce n'est pas un objet de type list)	`dico.keys()`
Objet itérable contenant les valeurs (ce n'est pas un objet de type list)	`dico.values()`
Objet itérable contenant les couples (clé,valeur)	`dico.items()`
Afficher les associations clé : valeur du dictionnaire `dico`	`for cle in dico :` `  print(cle, dico[cle])`

Annexe 2
(à ne pas rendre avec la copie)

```
class Yaourt:
def __init__(self,arome,duree):

     # **** Assertions : question 1.1. à compléter sur votre copie

     self.__arome = arome
     self.__duree = duree

     if arome == 'aucun':
          self.__genre = 'nature'
     else:
          self.__genre = 'aromatise'


# **** Méthode get_arome(self) : question 1.3. à compléter sur votre copie

     def get_duree(self):
          return(self.__duree)

     def get_genre(self):
          return ( self.__genre)


     def set_duree(self,duree):

          if duree > 0 :
               self.__duree = duree

# **** Mutateur d'arôme set_arome(self,arome) : question 2. à compléter sur votre copie

     def set_genre(self,arome):
          if arome == 'aucun':
               self.__genre = 'nature'
          else:
               self.__genre = 'aromatise'
```

Annexe 3
(à ne pas rendre avec la copie)

ATOMES					
Z	Nom	Sym	L	C	Masse_Atom
11	Sodium	Na	3	1	22.98976928
12	Magnésium	Mg	3	2 (IIA)	24.305
13	Aluminium	Al	3	13 (IIIA)	26.9815386
14	Silicium	Si	3	14	28.0855
15	Phosphore	P	3	15 (VA)	30.973762
16	Soufre	S	3	16 (VIA)	32.065
17	Chlore	Cl	3	17 (VII)	35.453
18	Argon	Ar	3	18 (VIIIA)	39.948
19	Potassium	K	4	1 (IA)	39.0983
20	Calcium	Ca	4	2 (IIA)	40.078
21	Scandium	Sc	4	3	44.955912
22	Titane	Ti	4	4	47.867
23	Vanadium	V	4	5	50.9415
24	Chrome	Cr	4	6	51.9961
25	Manganèse	Mn	4	7	54.938045
26	Fer	Fe	4	8	55.845
27	Cobalt	Co	4	9	58.933195
28	Nickel	Ni	4	10	58.6934
29	Cuivre	Cu	4	11	63.546
30	Zinc	Zn	4	12	65.409
31	Gallium	Ga	4	13 (IIIA)	69.723
32	Germanium	Ge	4	14	72.64
33	Arsenic	As	4	15	74.9216
34	Sélénium	Se	4	16 (VIA)	78.96
35	Brome	Br	4	17 (VII)	79.904
36	Krypton	Kr	4	18 (VIIIA)	83.798
37	Rubidium	Rb	5	1 (IA)	85.4678
38	Strontium	Sr	5	2 (IIA)	87.62
39	Yttrium	Y	5	3	88.90585
40	Zirconium	Zr	5	4	91.224
41	Niobium	Nb	5	5	92.90638
42	Molybdène	Mo	5	6	95.94
43	Technétium	Tc	5	7	98
44	Ruthénium	Ru	5	8	101.07
45	Rhodium	Rh	5	9	102.9055
46	Palladium	Pd	5	10	106.42
47	Argent	Ag	5	11	107.8682
48	Cadmium	Cd	5	12	112.411
49	Indium	In	5	13 (IIIA)	114.818
50	Étain	Sn	5	14	118.71
51	Antimoine	Sb	5	15	121.76
52	Tellure	Te	5	16	127.6
53	Iode	I	5	17 (VII)	126.90447
54	Xénon	Xe	5	18 (VIIIA)	131.293

VALENCE	
Col	Couche
1	s
2	s
3	d
4	d
5	d
6	d
7	d
8	d
9	d
10	d
11	d
12	d
13	p
14	p
15	p
16	p
17	p
18	p

Z : numéro atomique ;

Sym : symbole ;

L : numéro de ligne dans le tableau périodique ;

C ou Col : numéro de colonne dans le tableau périodique ;

Couche : couche de valence.