**Planche 1**
**Agro-Véto 2024**

**Exercice sans préparation.**

1. Écrire en Python une fonction `occ` qui prend en argument un nombre flottant `x` et une liste `L` et qui renvoie le nombre d'occurrences de `x` dans `L`.

2. a. Écrire en Python une fonction qui prend en argument deux entiers `m` et `n` et un flottant `p`$\in [0,1]$ (inutile de le vérifier), et qui renvoie un tableau de taille $(1,m)$ dont les coefficients sont des réalisations de variables aléatoires indépendantes suivant la loi binomiale $\mathcal{B}(n,p)$.

   b. Écrire en Python une fonction qui prend en argument une série statistique sous forme d'une liste `L` et qui renvoie un tableau de taille $(2,n)$ dont la première ligne est l'ensemble des différentes valeurs de la série statistique et dont la deuxième ligne est composée des nombres d'occurrences de chacune des valeurs.

Corrigé

**Planche 2**
**Agro-Véto 2024**

**Exercice sans préparation.**

1. Écrire en Python une fonction sans argument qui renvoie les résultats du lancer de deux lancers équilibrés (un vert et un rouge).

2. Il y a six joueurs numérotés de 1 à 6. Au début, ils ont chacun le même nombre de jetons.

   Lors d'une manche de jeu, on lance le dé vert. Le résultat désigne le joueur portant le numéro du dé. On fera attention qu'il lui reste des jetons. On lance ensuite le dé rouge et on calcule la somme des résultats des deux dés. Si ce résultat est le numéro d'un joueur (donc inférieur ou égal à 6), alors le joueur désigné par le dé vert lui donne un jeton, sinon il enlève son jeton du jeu.

   Écrire une fonction qui prend en argument une liste avec le nombre de jetons de chaque joueur et qui renvoie la liste des jetons restants de chaque joueur après une manche.

Corrigé

**Planche 3**
**Agro-Véto 2024**

**Exercice sans préparation.**

1. Une urne contient 10 boules indiscernables au toucher. Sur chacune est écrit une lettre : il y a 3 A, 3 T, 2 C et 2 G.

   On effectue $n$ tirages avec remise. Écrire en Python une fonction qui renvoie la chaine des $n$ caractères successivement tirés.

2. Écrire en Python une fonction boolénne `contient_avec_erreur` prenant en argument deux chaines de caractères qui renvoie `True` si, et seulement si, la première chaine de caractères contient la seconde à une erreur de caractère près. On supposera que la longueur de la première chaine est supérieure ou égale à celle de la seconde.

   *Par exemple* `contient_avec_erreur("ATCGG","ATT")` *devra renvoyer* `True`.

Corrigé

**Planche 4**
**Agro-Véto 2024**

**Exercice sans préparation.**

1. Écrire une fonction `somme` qui prend en argument une liste `L` et deux indices `i<j` et qui renvoie la somme des éléments de l'indice `i` inclus à l'indice `j` exclu.

2. Écrire une fonction `sous_liste_long` qui prend en argument une liste `L` et un entier $k$ et qui renvoie la plus grande somme des éléments de sous-listes de `L` (d'éléments consécutifs) de longueur $k$.

Corrigé

**Planche 5**
**Agro-Véto 2024**

**Exercice sans préparation.**

1. Écrire en Python une fonction prenant en argument une liste de nombres deux à deux distincts et qui renvoie le maximum et le minimum de cette liste ainsi que leurs indices respectifs.

2. Écrire en Python une fonction prenant en argument une liste de nombres deux à deux distincts et qui renvoie le maximum et le deuxième maximum (la deuxième valeur la plus grande) de cette liste ainsi que leurs indices respectifs.

Corrigé

**Planche 6**
**Agro-Véto 2024**

**Exercice sans préparation.**

1. Écrire une fonction `extract` prenant en argument une liste `L` de couples $(i, j)$ et deux entiers $n$ et $p$ et qui renvoie une liste de couples $(i, j)$ - éléments de `L` - tels que $0 \leqslant i < n$ et $0 \leqslant j < p$.

   Par exemple `extract([(1,2),(5,3),(9,2)], 6, 3)` devra renvoyer `[(1,2)]`.

2. Écrire une fonction prenant en argument une matrice $M$, et deux indices de ligne et de colonne $i$ et $j$ et renvoyant la liste des éléments de $M$ voisins de celui d'indices (de ligne et colonne) $i$ et $j$.

Corrigé

**Planche 7**
**Agro-Véto 2024**

**Exercice sans préparation.**
On considère un nuage de points $((x_i, y_i))_{1\leqslant i\leqslant n}$.

1. Écrire une fonction qui prend en argument les coordonnées de deux points $A$ et $B$ du nuage et qui renvoie la distance $AB$ entre ces deux points.

2. Écrire une fonction qui prend argument la liste des coordonnées d'un nuage de points et qui renvoie les coordonnées du point moyen du nuage.

3. Écrire une fonction qui renvoie l'indice $i$ du point $(x_i, y_i)$ du nuage le plus proche du point moyen.

Corrigé

**Planche 8**
**Agro-Véto 2024**

**Exercice sans préparation.**

1. Écrire en Python une fonction d'argument une liste `L` qui teste si la liste `L` vérifie l'hypothèse $\mathcal{H}$ suivante

   $\mathcal{H}$ : les éléments de la liste `L` sont des entiers compris (au sens large) entre 0 et $n-1$, où $n$ désigne la longueur de `L`.

2. Écrire en Python une fonction d'argument une liste `L` vérifiant l'hypothèse $\mathcal{H}$ et qui teste si la liste `L` vérifie que l'hypothèse $\mathcal{H}'$ : `L` est une permutation de $[\![0, n-1]\!]$ où $n$ est la longueur de `L`, i.e. :

   $\mathcal{H}'$ : la liste `L` contient exactement une fois chaque valeur comprise entre 0 et $n-1$.

3. Combien de listes vérifient que `L[0] = 0` lorsqu'on permute les éléments de la liste $L = [0, \dots, n-1]$ ($n \in \mathbb{N}^*$) ?

Corrigé

**Planche 9**
**Agro-Véto 2024**

**Exercice sans préparation.**

Soient $a$ et $b$ deux entiers naturels tels que $a < b$.

1. Écrire en Python une fonction `verif(L,a,b)` qui a pour paramètres une liste `L` d'entiers naturels et deux entiers `a` et `b` et qui renvoie `True` si tous les éléments de `L` sont dans l'intervalle $[\![a, b]\!]$ et `False` sinon.

2. Soit `L` une liste donc chaque élément est dans l'intervalle d'entiers $[\![a, b]\!]$.

   Écrire en Python une fonction `denombre(L,a,b)` qui détermine le nombre d'apparitions de chacune des valeurs possibles (entre $a$ et $b$) de la liste `L` et qui renvoie le résultat dans une liste dont le premier élément représentera le nombre de $a$ dans `L`, le deuxième le nombre de $a+1$ dans `L`, etc.

   *Par exemple, pour* $a = 0$ *et* $b = 4$, `denombre([1,3,0,4,1,3,1],0,4)` *renverra* `[1,3,0,2,1,0,0]`.

Corrigé

**Planche 10**

**Planche 11**

**Planche 12**

**Planche 13**

**Planche 14**

**Planche 15**

**Planche 16**

**Planche 17**

**Planche 18**

**Planche 19**

**Planche 20**

**Planche 21**

**Planche 22**
**Agro-Véto 2023**

**Exercice sans préparation.**

1. Écrire une fonction qui détermine le maximum d'une liste de nombres.
2. Écrire une fonction `histo` qui prend en argument une liste $L$ de chiffres (entre 0 et 9) et qui renvoie une liste $nb$ telle que pour tout $k \in [\![0, 9]\!]$, `nb[k]` soit le nombre de fois où le chiffre $k$ apparait dans `L`.

Corrigé

**Planche 23**

**Planche 24**

**Planche 25**

**Planche 26**
**Agro-Véto 2023**

**Exercice sans préparation.**
Soient $n \in \mathbb{N}$ et $(a_0, \dots, a_n) \in \mathbb{R}^{n+1}$. On considère le polynôme $P(X) = a_0 + a_1 X + \cdots + a_n X^n$. On représente ce polynôme $P$ en Python par la liste de ses coefficients $[a_0, \dots, a_n]$.

1. Écrire une fonction `evalue` qui prend en argument un polynôme $P$ et un réel (flottant) $a$ et qui renvoie $P(a)$.
2. Écrire une fonction `deriv` qui prend en argument un polynôme $P$ et qui renvoie $P'$.
3. On considère l'application linéaire $f$ définie sur $\mathbb{R}[X]$ par $f(P)(X) = 2XP(X) - P'(X)$. Écrire une fonction $f$ qui prend en argument un polynôme $P$ et qui renvoie $f(P)$.

Corrigé

**Planche 27**

**Planche 28**

**Planche 29**
**Agro-Véto 2023**
**(sujet 0)**

**Exercice sans préparation.**

Dans cet exercice, on considère des listes de booléens, c'est-à-dire des listes ayant pour éléments uniquement des valeurs `True` et `False`. Par exemple les listes `[True, False, True]` et `[False, False]` sont des listes de booléens.

1. Écrire une fonction `nb_vrai` prenant en entrée une liste de booléens et renvoyant le nombre d'occurrences de la valeur `True`. Par exemple `nb_vrai([False, True, True])` doit renvoyer 2, car la valeur `True` apparait deux fois dans la liste `[False, True, True]`.

2. Écrire une fonction `bloc_vrai` prenant en entrée une liste de booléens, et renvoyant :

   - `True` si la valeur `True` est présente dans la liste et que tous les `True` sont côte-à-côte, dans le même bloc,
   - `False` sinon.

   Par exemple, `bloc_vrai([False, True, True, True, False, False])` doit renvoyer `True` car il y a un unique bloc de 3 `True`. Autre exemple, `bloc_vrai([False, True, True, True, False, False, True, True, False])` doit renvoyer `False` car il y a deux blocs de `True` : un premier de longueur 3, un second de longueur 2.

Corrigé

**Planche 30**

**Planche 31**

**Planche 32**

**Planche 33**

**Planche 34**

**Planche 35**

**Planche 36**

**Planche 37**

**Planche 38**

**Planche 39**
**Agro-Véto 2022**

**Exercice sans préparation.**

On suppose qu'on dispose d'une fonction `alphabet` qui renvoie une chaîne de 26 caractères composées des 26 lettres (minuscules) de l'alphabet français, dans l'ordre alphabétique.

On souhaite coder une chaîne de caractères par décalage de $n$ lettres. Par exemple, pour $n = 3$, `"a"` devient `"d"`, `"b"` devient `"e"`, ... `"w"` devient `"z"`, `"x"` devient `"a"`, `"y"` devient `"b"` et `"z"` devient `"c"`.

1. Écrire une fonction `decalage` qui prend en argument un entier `n` et qui renvoie une chaîne de caractères contenant les 26 lettres de l'alphabet (en minuscule) décalés de $n$.

2. En déduire une fonction `codage` qui prend en argument une chaîne de caractères et un entier naturel $n$ qui code par décalage la chaîne de caractères en décalant chaque caractères de $n$.

Corrigé

**Planche 40**

**Planche 41**

**Planche 42**

**Planche 43**

**Planche 44**

**Corrigé de l'exercice de la planche 1**

1.
```
def occ(x,L):
    nb_x = 0
    for y in L:
        if x==y:
            nb_x += 1
    return nb_x
```

2. a.
```
import random as rd

def simulebinom(n,p):
    s = 0
    for _ in range(n):
        if rd.random() < p:
            s += 1
    return s


def seriebinom(m,n,p):
    L = []
    for _ in range(m):
        L.append(simulebinom(n,p))
    return L
```

b. La méthode la plus esthétique (et efficace) est d'utiliser un dictionnaire :
```
import numpy as np

def occ_serie(L):
    dico = {}
    for x in L:
        if x in dico:
            dico[x] += 1
        else:
            dico[x] = 1
    tab = [[],[]]
    for x,nb_x in dico.items():
        tab[0].append(x)
        tab[1].append(nb_x)
    return np.array(tab)
```

```
def occ_serie(L):
    dico = {}
    for x in L:
        if x not in dico:
            dico[x]=1
        else:
            dico[x] += 1
    tab = [list(dico.keys()),list(dico.values())]
    return np.array(tab)
```

Retour à la planche 1

**Corrigé de l'exercice de la planche 2**

1.
```
import random as rd

def desVR():
    return (rd.randin(1,6),rd.randin(1,6))
```

2.
```
def manche(nbjetons):
    v,r = desVR()
    if nbjetons[v-1] == 0:
        return nbjetons
    elif v+r <= 6:
        nbjetons[v+r-1] += 1
    nbjetons[v-1] -= 1
    return nbjetons
```

Retour à la planche 2

**Corrigé de l'exercice de la planche 3**

1.
```
import random as rd

def chaine_tirages(n):
    urne = "A"*3 + "T"*3 + "C"*2 + "G"*2
    ch = ""
    for _ in range(n):
        ch += rd.choice(urne)
    return ch
```

2. On commence par écrire une fonction qui cherche la seconde chaine à un indice i fixé dans la première, en comptant le nombre d'erreurs de caractères. On peut alors écrire la deuxième fonction parcourant tous les indices possibles où peut commencer la deuxième chaine dans la première.

```
def contient_avec_erreur_indice_i(i,ch1,ch2):
    nb_erreurs = 0
    for j in range(len(ch2)):
        if i+j < len(ch1) and ch1[i+j] != ch2[j]:
            nb_erreurs += 1
        if nb_erreurs == 2:
            return False
    return True

def contient_avec_erreur(ch1,ch2):
    for i in range(len(ch1)-len(ch2)+1):
        if contient_avec_erreur_indice_i(i,ch1,ch2):
            return True
    return False
```

Retour à la planche 3

**Corrigé de l'exercice de la planche 4**

1.
```
def somme(L,i,j):
    s = 0
    for k in range(i,j):
        s += L[k]
    return s
```

2.
```
def sous_liste_long(L,k):
    s_max = somme(L,0,k)
    for i in range(1,len(L)-k):
        s = somme(L,i,i+k)
        if s > s_max:
            s_max = s
    return s_max
```

Retour à la planche 4

**Corrigé de l'exercice de la planche 5**

1.
```
def minmax(L):
    i_min, i_max = 0, 0
    for i in range(1,len(L)):
        if L[i] < L[i_min]:
            i_min = i
        elif L[i] > L[i_max]:
            i_max = i
     return L[i_max], i_max, L[i_min], i_min
```

2.
```
def minmax(L):
    i_max, i_2max = 0, 1
    if L[0] < L[1]:
        i_max, i_2max = 1, 0
    for i in range(2,len(L)):
        if L[i] > L[i_max]:
            i_max, i_2max = i, i_max
        elif L[i] > L[i_2max]:
            i_2max = i
    return L[i_max], i_max, L[i_2max], i_2max
```

Retour à la planche 5

**Corrigé de l'exercice de la planche 6**

1.
```
def extract(L,n,p):
    return [(i,j) for (i,j) in L if (0<=i<n and 0 <= j <p)]
```

2.
```
def voisins(M,i,j):
    L = []
    n,p = np.shape(M)
    for u in range(max(0,i-1),1+min(n-1,i+1)):
        for v in range(max(0,j-1),1+min(n-1,j+1)):
            if abs(u-i) + abs(v-j) > 0:
                L.append(M[u,v])
    return L
```

Retour à la planche 6

**Corrigé de l'exercice de la planche 7**

1.
```
def d(A,B):
    return ((A[0]—B[0])**2 + (A[1]—B[1])**2)**0.5
```

2.
```
def point_moyen(nuage):
    s_x, s_y, n = 0, 0, len(nuage)
    for k in range(n):
        x,y = nuage[k]
        s_x += x
        s_y += y
    return (s_x/n,s_y/n)
```

3.
```
def i_plus_proche_pm(nuage):
    pm = point_moyen(nuage)
    i_min = 1
    for i in range(2,len(nuage)):
        if d(nuage[i],pm) < d(nuage[i_min],pm):
            i_min = i
    return i_min
```

Retour à la planche 7

**Corrigé de l'exercice de la planche 8**

1.
```
def testeH(L):
    n = len(L)
    for x in L:
        if x != int(x) or not(0 <= x < n): # ou type(x) != int
            return False
    return True
```

2.
```
def testeHprime(L):
    n = len(L)
    nb_occ = [0]*n
    for x in L:
        if int(x) == x and 0 <= x < n and nb_occ[x] == 0: # ou type(x) = int
            nb_occ[x] = 1
        else:
            return False
    return True
```

3. Il suffit de considérer tous les permutations de la sous-liste $[\![1, n-1]\!]$ : il y en a $(n-1)!$.

Retour à la planche 8

**Corrigé de l'exercice de la planche 9**

1.
```
def verif(L,a,b):
    for x in L:
        if x != int(x) or not(a <= x <= b):
            return False
    return True
```

2.
```
def denombre(L,a,b):
    n = len(L)
    occ = [0]*n
    for x in L:
        occ[x-a] += 1
    return occ
```

Retour à la planche 9

**Planche 45**

**Planche 46**

**Planche 47**

**Planche 48**

**Planche 49**

**Planche 50**

**Planche 51**

**Planche 52**

**Planche 53**

**Planche 54**

**Planche 55**

**Planche 56**

**Planche 57**

**Corrigé de l'exercice de la planche 22**

1.
```
def maxi(L):
    m = L[0]
    for x in L[1:]:
        if x > m:
            m = x
    return m
```

2.
```
def histo(L):
    nb = [0] * 10
    for x in L:
        nb[x] += 1
    return nb
```

Retour à la planche 22

**Planche 58**

**Planche 59**

**Planche 60**

**Corrigé de l'exercice de la planche 26**

1. On propose une version itérative et une version récursive :

```
def evalue(P,a):
    s = 0
    for k in range(len(P)):
        s += P[k]*a**k
    return s

def evalue2(P,a):
    if len(P)==1:
        return P[0]
    else:
        return P[0] + a * eval2(P[1:],a)
```

2. On distingue le cas où $P$ est nul. Dans le cas contraire, si $P(X) = \sum_{i=0}^{n} a_i X^i$, alors :

$$P'(X) = \sum_{i=1}^{n} i a_i X^{i-1} = \sum_{k=0}^{n-1} (k+1) a_{k+1} X^k.$$

On en déduit le code ci-dessous :

```
def deriv(P):
    if len(P)==1:
        return [0]
    return [(k+1)*P[k+1] for k in range(len(P)-1)]
```

3.

```
def f(P):
    if P == [0]:
        return P
    else:
        n = len(P)
        res = [0]*(n+1)
        XP = [0]+P
        P_prime = deriv(P)
        for k in range(n-1):
            res[k] = 2*XP[k] - P_prime[k]
        res[n-1] = 2*XP[n-1]
        res[n] = 2*XP[n]
        return res
```

Retour à la planche 26

**Planche 61**

**Planche 62**

**Corrigé de l'exercice de la planche 29**

1. On parcourt (par élément ici) une liste de booléens et on ajoute 1 à notre compteur dès qu'on rencontre `True`.

```
def nb_vrai(liste):
    s = 0
    for x in liste:
        if x:
            s += 1
    return s
```

On peut aussi profiter du fait que les booléens sont en réalité un sous-type du type **`int`** (entiers) en Python. Les booléens `True` et `False` sont respectivement égaux à 1 et 0.

```
def nb_vrai(liste):
    s = 0
    for x in liste:
        s += x
    return s
```

2. On interromp l'algorithme pour renvoyer `False` si on trouve un `True` juste après un `False` et si on a déjà lu précédemment un `True`. On définit donc deux variables drapeau : l'une, `au_moins_un_vrai`, détectant si on a déjà lu un `True` et l'autre, `faux_apres_vrai`, détectant si on lit un `False` après un `True`.

   Enfin, il ne faut pas oublier le cas où il n'y a aucun `True` dans la liste, c'est pourquoi, si on n'a pas déjà renvoyé `False`, on renvoie `au_moins_un_vrai` en fin de programme.

```
def bloc_vrai(liste):
    au_moins_un_vrai = False
    faux_apres_vrai = False
    for x in liste:
        if x:
            if not au_moins_un_vrai:
                au_moins_un_vrai = True
            elif faux_apres_vrai:
                return False
        elif au_moins_un_vrai:
            faux_apres_vrai = True
    return au_moins_un_vrai
```

Retour à la planche 29

**Planche 63**

**Planche 64**

**Planche 65**

**Planche 66**

**Planche 67**

**Planche 68**

**Planche 69**

**Planche 70**

**Planche 71**

**Corrigé de l'exercice de la planche 39**

1.
```
def decalage(n):
    chaine = alphabet()
    return chaine[n:] + chaine[:n]
```

2.
```
def codage(chaine,n):
    alphaB = alphabet()
    alphaBdecale = decalage(n)
    crypt = {}
    for k in range(26):
        crypt[alphaB[k]] = alphaBdecale[k]
    chaine2 = ""
    for car in chaine:
        chaine2 += crypt[car]
    return chaine2
```

Retour à la planche 39

**Planche 72**

**Planche 73**

**Planche 74**

**Planche 75**

**Planche 76**