DEVOIR SURVEILLÉ N°1

Samedi 10 octobre 2020 — 2 heures

On a le droit d'utiliser, dans une question, les programmes des questions antérieures, *même si* l'on n'a pas réussi à les écrire. Les trois exercices sont indépendants et peuvent être traités dans n'importe quel ordre. Ils ne sont pas rangés par ordre de difficulté.

Les calculatrices sont interdites pour cette épreuve.

Exercice I — Quelques entiers naturels

Dans cet exercice, on interdit d'utiliser les chaînes de caractères. Les seules opérations autorisées sont les six opérations arithmétiques sur les entiers.

QUESTION 1.1 — *Écrire le programme* `PleinDeUns(`n`)` *qui renvoie le nombre* $11111\ldots1$ *avec* n *fois le chiffre* 1.

```
>>> PleinDeUns(8)
11111111
```

QUESTION 1.2 — *Écrire le programme* `PleinDePuissancesDeDix(`n`)` *qui calcule et renvoie le nombre* $110100100010000\ldots$ *qui se termine par un* 1 *suivi de* n *zéros.*

```
>>> PleinDePuissancesDeDix(8)
1101001000100001000001000000100000000100000000
```

QUESTION 1.3 — *Écrire le programme* `AvecTousLesNombres(`n`)` *qui renvoie le nombre* $1234567891011\ldots$ *obtenu en mettant bout à bout tous les nombres de* 1 *jusqu'à* n.

```
>>> AvecTousLesNombres(20)
12345678910111213141516171819 20
```

Exercice II — Sommes avec des coefficients binomiaux

On rappelle que les coefficients binomiaux sont définis, pour $n \geqslant 0$ et $k \in \{0\,;\,1\,;\,\ldots\,;\,n\}$, par la formule

$$\binom{n}{k} = \frac{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1},$$

et qu'on a la relation

$$\binom{n}{n-k} = \binom{n}{k}.$$

QUESTION 2.1 — *Expliquer l'intérêt de cette relation, si l'on doit par exemple calculer « 80 parmi 82 ».*

QUESTION 2.2 — *Écrire un programme* `Produit(a, b)` *qui calcule et renvoie* $a \times (a+1) \times (a+2) \times \cdots \times (b-1) \times b$, *en convenant que ce produit vaut* 1 *dès lors que* $a > b$.

```
>>> Produit(3, 5)
60
>>> Produit(5, 3)
1
```

QUESTION 2.3 — *Écrire le programme* `Binomial(n, k)` *qui calcule les coefficients binomiaux.*

```
>>> Binomial(10, 3)
120
```

On veut maintenant calculer des sommes du type

$$\mathrm{S}(f,n,a,b) = \sum_{k=a}^{b} \binom{n}{k} \times f(n,k).$$

QUESTION 2.4 — *On donne ci-dessous deux programmes pour calculer la somme de tous les coefficients binomiaux (faisons semblant d'ignorer qu'elle vaut 2^n). Lequel des deux est le meilleur et pourquoi ?*

```
def SommeDesCoefficientsBinomiaux(n) :
    s = 0 ; cnk = 1     # Première version, dite de la grenouille
    for k in range(0, n + 1) :
        s += cnk
        cnk *= n - k
        cnk //= k + 1
    return s
def SommeDesCoefficientsBinomiaux(n) :
    s = 0               # Deuxième version, dite du silure
    for k in range(0, n + 1) :
        s += Binomial(n, k)
    return s
```

QUESTION 2.5 — *On suppose avoir écrit le programme* `Somme(f, n, a, b)` *qui calcule* $\mathrm{S}(f,n,a,b)$ *(on le fera dans la question juste après). Comment définir la fonction f et compléter le code ci-dessous pour retrouver le programme précédent ?*

```
def f(...

def SommeDesCoefficientsBinomiaux(n, k) :
    # Troisième version, dite de l'anguille-sous-roche
    return Somme(...
```

QUESTION 2.6 — *Écrire le programme* `Somme(f, n, a, b)`*. Si $a > b$ on convient de renvoyer zéro.*

Exercice III — Un rond dans un truc bizarre

Pour chaque nombre complexe c, on considère la suite $(u_n)_{n\geqslant 0}$ de premier terme $u_0 = 0$ et vérifiant la relation de récurrence $u_{n+1} = u_n^2 + c$.

QUESTION 3.1 — *Écrire un programme* `Terme(n, c)` *qui calcule et renvoie u_n.*

Pour chaque nombre $c \in \mathbf{C}$, on définit

$$\mathrm{T}(c) = \min\{n \geqslant 0 \mid |u_n| > 2\} \in \mathbf{N} \cup \{\infty\}.$$

C'est le temps (en nombre d'itérations) que met la suite pour sortir du disque de rayon 2 et de centre 0. Le programme

```
def TempsDeSortie(c) :
    u = 0
    n = 0
```

```
    while abs(u) <= 2 :
        u = u ** 2 + c
        n += 1
    return n
```

calcule $\mathrm{T}(c)$, mais seulement dans le cas où il est fini : il se peut que la suite ne sorte jamais du disque et que le programme tourne indéfiniment.

Pour empêcher ce fâcheux cas, on introduit une variable globale

```
N_MAX = 100
```

et on interdit que le programme précédent effectue plus de `N_MAX` itérations.

QUESTION 3.2 — *Réécrire le programme* `TempsDeSortie(c)` *pour qu'il termine quoi qu'il arrive, et renvoie* `N_MAX` *lorsque* $\mathrm{T}(c) \geqslant$ `N_MAX`.

On utilise dans ce qui suit `Aléa()`, qui renvoie un réel au hasard dans $[0\,;1[$, et on définit le programme suivant.

```
from random import random as Aléa
from numpy import pi as π, exp
def ComplexeAléa() :
    θ = 2 * π * Aléa()
    ρ = 2 * Aléa()
    return ρ * exp(1j * θ)
```

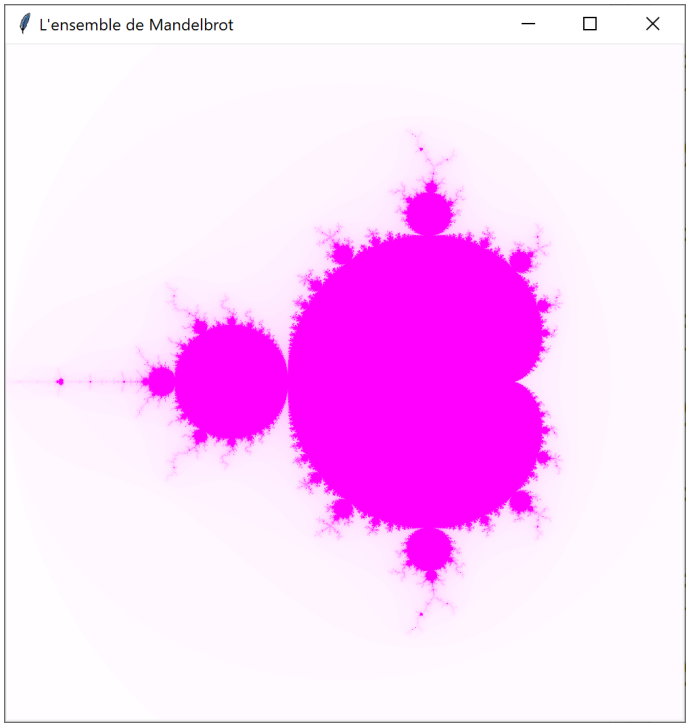
QUESTION 3.3 — *Que renvoie précisément le programme* `ComplexeAléa()` *?*

QUESTION 3.4 — *En déduire un programme* `Rechercher(n)` *qui cherche, en essayant des valeurs au hasard, un nombre c tel que $\mathrm{T}(c) = n$, et qui renvoie celui-ci lorsqu'il en a trouvé un.*

L'*ensemble de Mandelbrot* $\mathscr{M}$ est défini de la manière suivante : c'est l'ensemble des nombres complexes c pour lesquels le temps de sortie est au moins égal à `N_MAX`, c'est-à-dire

$$\mathscr{M} = \{c \in \mathbf{C} \mid \mathrm{T}(c) \geqslant \texttt{N\_MAX}\}.$$

Voici cet ensemble, dans une fenêtre $[-2\,;1] \times [-1{,}5\,;1{,}5]$.



QUESTION 3.5 — *Écrire un programme* `Mandelbrot(c)` *qui teste si* $c \in \mathbf{C}$ *appartient à* $\mathscr{M}$*, et qui renvoie* `True` *ou* `False` *en conséquence.*

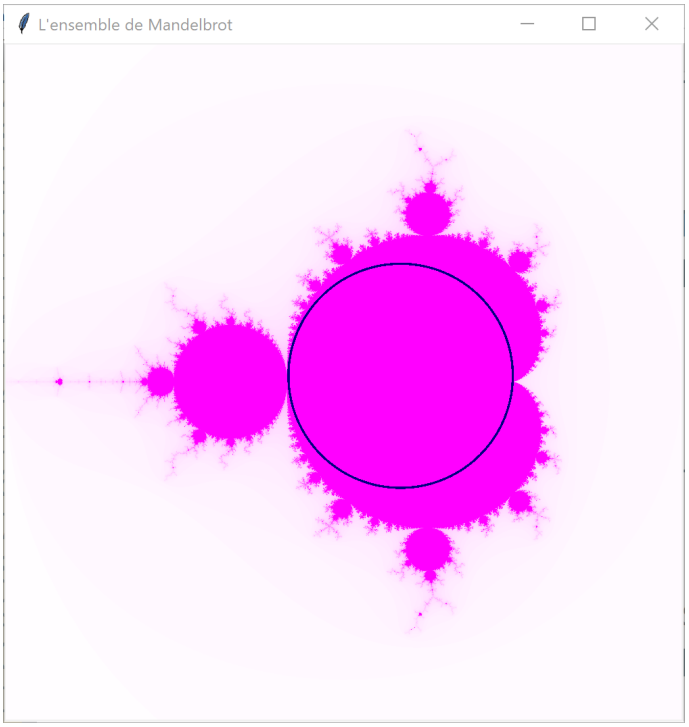
On cherche maintenant le plus grand cercle inscrit dans l'ensemble $\mathscr{M}$. Cela consiste à trouver son rayon r et l'abscisse de son centre a.

QUESTION 3.6 — *Écrire un programme* `CercleEstInclus(a, r, NB_ESSAIS = 1000)` *qui choisit un certain nombre de points au hasard sur le cercle de centre* a *et de rayon* r*, et renvoie* `True` *s'ils sont tous dans* $\mathscr{M}$*, ou bien* `False` *si au moins l'un d'eux ne l'est pas.*

QUESTION 3.7 — *En déduire un programme* `PlusGrandRayon(a)` *qui étant donné une affixe* a *détermine le plus grand nombre* r *tel que le cercle de centre* a *et de rayon* r *est contenu dans* $\mathscr{M}$*.*

QUESTION 3.8 — *Écrire finalement un programme* `PlusGrandCercle()` *qui résout le problème annoncé.*

```
>>> PlusGrandCercle()
((-0.24572953547624338+0.02641646363917898j), 0.5029296875)
```



FIN

Solutions & commentaires

Exercice I — Quelques entiers naturels

1.1) Si l'on note u_n ce nombre, on constate que $u_1 = 1$ (et même $u_0 = 0$) et que $u_{i+1} = 10u_i + 1$. D'où le programme ci-dessous.

```
def PleinDeUns(n) :
    x = 0
    for i in range(n) :
        x = 10 * x + 1
    return x
```

Variante plus astucieuse : le nombre qui s'écrit avec n fois le chiffre 9 est évidemment $10^n - 1$. Pour obtenir u_n il suffit de le diviser par 9.

```
def PleinDeUns(n) :
    return (10 ** n - 1) // 9
```

1.2) Même idée : notons v_n ce nombre. Alors $v_1 = 1$ (et même $v_0 = 0$) et on a la relation de récurrence $v_{i+1} = 10^{i+1} \times x + 10^i$. Rangeons 10^i dans une variable `p`, on aura alors $10^{i+1} = 10p$ et $v_{i+1} = (10p)x + p$.

```
def PleinDePuissancesDeDix(n) :
    x = 0
    for i in range(0, n + 1) :
        p = 10 ** i
        x = (10 * p) * x + p
    return x
```

1.3) On généralise encore la même idée. Notons x le nombre à construire : à chaque étape il faut le multiplier par une certaine puissance de 10, disons p, de sorte à décaler ses chiffres ; avant de lui ajouter le nombre i. Si i s'écrit sur k chiffres, il faut prendre $p = 10^r$. Mais on n'a pas besoin de calculer r : on sait que pour i allant de 1 à 9 il faut prendre $p = 10$, puis pour i allant de 10 à 99 il faut prendre $p = 100$, etc. et ainsi, on change p en $10p$ lorsque i se retrouve égal à p.

```
def AvecTousLesNombres(n) :
    x = 0 ; p = 10
    for i in range(1, n + 1) :
        if i == p :
            p *= 10
        x = p * x + i
    return x
```

Exercice II — Sommes avec des coefficients binomiaux

2.1) Si l'on applique brutalement la définition, il faut faire 80 multiplications au numérateur et autant au dénominateur. Alors qu'avec la relation, on a plus

$$\binom{82}{80} = \binom{82}{2} = \frac{82 \times 81}{2 \times 1} = 3\,321,$$

qui peut presque se calculer de tête, mais qui en tout cas demande considérablement moins de calculs.

2.2) La seule difficulté de cette question est de ne pas se tromper dans les bornes du `range`.

```
def Produit(a, b) :
    p = 1
```

```
    for i in range(a, b + 1) :
        p *= i
    return p
```

2.3) L'énoncé exclut les cas où $k < 0$ ou $k > n$, mais traitons-les quand même (en renvoyant zéro s'ils surviennent).

```
def Binomial(n, k) :
    if k < 0 or k > n :
        return 0
    else :
        if k <= n // 2 :
            return Produit(n - k + 1, n) // Produit(2, k)
        else :
            return Produit(k + 1, n) // Produit(2, n - k)
```

2.4) Le silure calcule $2 \times (1 + 2 + 3 + 4 + 5 + \cdots + n)$ multiplications sur la totalité des utilisations du programme `Binomial` (il faut recalculer le produit à chaque fois à partir du début).

Alors que si l'on est plus calme et attentif (comme la grenouille, voir en librairie), on remarque que l'on peut déduire simplement chaque coefficient binomial à partir du précédent, avec la relation

$$\binom{n}{k+1} = \frac{n \times (n-1) \times \cdots \times (n-(k+1)+1}{(k+1) \times k \times \cdots \times 1} = \frac{n-k}{k+1} \times \binom{n}{k}.$$

Autrement dit, et c'est ce que fait la grenouille, on peut s'en sortir en effectuant seulement une multiplication et une division à chaque étape. C'est donc bien plus efficace.

2.5) Voilà voilà, on pense bien à mettre tous les arguments de chaque programme.

```
def f(n, k) :
    return 1
def SommeDesCoefficientsBinomiaux(n, k) :
    return Somme(f, n, 0, n)
```

2.6) Pour ré-exploiter la version de la grenouille, il faut quoi qu'il arrive calculer les termes de la somme à partir de l'indice $k = 0$, mais on ne les ajoute à la variable `s` qu'à partir de $k = a$.

```
def Somme(f, n, a, b) :
    s = 0 ; cnk = 1
    for k in range(0, b + 1) :
        if k >= a :
            s += f(k) * cnk
        cnk *= n - k
        cnk //= k + 1
    return s
```

Exercice III — Un rond dans un truc bizarre

3.1) On itère n fois (avec une boucle `for`) la relation de récurrence.

```
def Terme(n, c) :
    u = 0
    for i in range(n) :
        u = u ** 2 + c
    return u
```

3.2) Ajoutons un deuxième test dans la condition de continuation, à l'aide du mot-clé `and`. Remarquons que si c'est la condition $n <$ `N_MAX` qui échoue et fait sortir de la boucle, alors c'est que $n =$ `N_MAX`, et c'est alors ce qu'on renvoie, comme voulu.

```
N_MAX = 100
def TempsDeSortie(c) :
    u = 0
    n = 0
    while n < N_MAX and abs(u) <= 2 :
        u = u ** 2 + c
        n += 1
    return n
```

Attention ici à ne pas recalculer *tous* les termes à partir de u_0 à chaque fois, en écrivant quelque chose du genre

```
def TempsDeSortie(c) :
    n = 0
    while abs(Terme(n, c)) <= 2 :
        n += 1
    return n
```

C'est évidemment beaucoup moins efficace : on ferait ici $1+2+3+\cdots+n = n(n+1)/2$ opérations, contre seulement n avec l'algorithme précédent. Pour se convaincre que c'est mal, on regarde avec $n = 1000$.

3.3) Le programme `ComplexeAléa()` commence par choisir un nombre θ au hasard entre 0 et 2π (parce qu'en prenant un nombre entre 0 et 1 et en le multipliant par 2π, on obtient un nombre entre 0 et 2π), puis un nombre ρ entre 0 et 2, et renvoie $\rho e^{i\theta}$. Donc un nombre complexe aléatoire, choisi dans le disque de centre l'origine et de rayon 2.

3.4) On fait naïvement ce qui est dit dans le sujet. De manière étonnante, ça répond toujours très vite.

```
def Rechercher(n) :
    c = ComplexeAléa()
    while not VitesseDeSortie(c) == n :
        c = ComplexeAléa()
    return c
```

3.5) Comme on a bridé à `N_MAX` les valeurs renvoyées par le programme `TempsDeSortie`, il suffit de tester s'il renvoie `N_MAX`. Mais si dans le programme ci-dessous on met un `>=` au lieu du `==`, ça marche aussi.

```
def Mandelbrot(c) :
    return TempsDeSortie(c) == N_MAX
```

3.6) La aussi, on fait ce que dit l'énoncé. On peut évidemment arrêter le programme (avec l'instruction `return`) dès qu'on a trouvé un point qui n'est pas dans $\mathscr{M}$.

```
def CercleEstInclus(a, r, NB_ESSAIS = 1000) :
    for k in range(NB_ESSAIS) :
        θ = 2 * π * Aléa()
        c = a + r * exp(1j * θ)
        if not Mandelbrot(c) :
            return False
    return True
```

3.7) C'est le moment de caser l'algorithme de dichotomie. On sait que le rayon cherché est entre 0 et 2, et que pour les valeurs de r inférieures à celle qu'on recherche, le cercle est inclus dans $\mathscr{M}$, alors que pour les valeurs supérieures, il ne l'est pas.

On va donc encadrer de plus en plus finement le rayon cherché, disons jusqu'à une précision de 10^{-3}.

```
def PlusGrandRayon(a) :
    r_min = 0 ; r_max = 2
    while r_max - r_min > 1e-3 :
```

```
        r = (r_min + r_max) / 2
        if CercleEstInclus(a, r) :
            r_min = r
        else :
            r_max = r
    return r
```

3.8) Première idée, naïve : on choisit des a au hasard, on calcule les plus grands rayons qui leur sont associés, et on renvoie le couple correspondant au plus grand rayon qu'on a observé.

```
def PlusGrandCercle() :
    NB_ESSAIS = 100
    meilleur_a = 0
    meilleur_r = PlusGrandRayon(0)
    for k in range(NB_ESSAIS) :
        a = ComplexeAléa()
        r = PlusGrandRayon(a)
        if r > meilleur_r :
            meilleur_a = a
            meilleur_r = r
    return (meilleur_a, meilleur_r)
```

C'était la solution attendue dans le sujet. Alors en pratique, ça ne marche pas très bien. Si on veut faire beaucoup mieux, on procède par amélioration sucessives. Voyons comment faire, en s'autorisant à utiliser une liste pour simplifier.

On suppose qu'on a trouvé un a et un r assez bons, rangés dans des variables `ma` et `mr`. On se donne un $h > 0$ assez petit et on essaie de « déplacer » le centre a dans chaque des quatre directions principales : $\texttt{ma} + h$, $\texttt{ma} + hi$, $\texttt{ma} - h$ et $\texttt{ma} - hi$. On renvoie, parmi ces quatre nouveaux centres plus le point de départ, le couple correspondant à celui des cinq qui a le plus grand rayon.

```
def Améliorer(ma, mr, h) :
    meilleur_a = ma ; meilleur_r = mr
    for d in [1, 1j, -1, -1j] :
        a = ma + h * d
        r = PlusGrandRayon(a)
        if r > meilleur_r :
            meilleur_a = a
            meilleur_r = r
    return (meilleur_a, meilleur_r)
```

Ensuite on va itérer cette étape. On part d'un `ma` choisi au hasard, et de $h = 0{,}1$ (par exemple). Puis on essaie d'améliorer plusieurs fois, avec l'algorithme précédent. Lorsqu'on y arrive plus, on recommence d'où l'on est arrivé avec un h dix fois plus petit, puis cent fois plus petit, puis mille fois plus petit.

```
def PlusGrandCercle() :
    meilleur_a = ComplexeAléa()
    while not(Mandelbrot(meilleur_a)) :
        meilleur_a = ComplexeAléa()
    meilleur_r = PlusGrandRayon(0)
    h = 0.1
    for k in range(3) :
        (a, r) = Améliorer(meilleur_a, meilleur_r, h)
        while a != meilleur_a :
            meilleur_a = a ; meilleur_r = r
            (a, r) = Améliorer(meilleur_a, meilleur_r, h)
        h /= 10
    return (meilleur_a, meilleur_r)
```