

|            |               |
|------------|---------------|
| <b>NOM</b> | <b>PRENOM</b> |
|------------|---------------|

| <b>Titre</b>               | page | date | état |
|----------------------------|------|------|------|
| Mode interactif            | 2    |      |      |
| OuLiPo                     | 6    |      |      |
| Suite de Stern-Brocott     | 8    |      |      |
| Feu de forêt               | 10   |      |      |
| Suites de Beaty            | 12   |      |      |
| Marche de l'ivrogne        | 13   |      |      |
| Mélange de cartes/Bataille | 14   |      |      |
| Calcul mental              | 16   |      |      |
| Volterra                   | 18   |      |      |
| Produit des chiffres       | 20   |      |      |
| Défis arithmétiques        | 20   |      |      |
| Problème de Kantorovitch   | 21   |      |      |
| Récif coralien             | 24   |      |      |
| Base -10                   | 25   |      |      |
| Cube mobile                | 27   |      |      |
| Montrez Bézout             | 30   |      |      |
| 6 sur $\pi^2$              | 31   |      |      |
| Suite autoréférente        | 32   |      |      |
| Tchebitchev/Erdős          | 33   |      |      |
| Codage de Vigenere         | 33   |      |      |
| Intégrales de Wallis       | 35   |      |      |
| Markov dans le métro       | 36   |      |      |
| Exploration d'une arène    | 38   |      |      |
| Cinq cartes                | 39   |      |      |
| Puissance 4                | 40   |      |      |
| Traducteur SMS             | 41   |      |      |

Le but de ce T.D. est de familiariser avec la syntaxe de Python (*le langage informatique, pas celui des auteurs de "sit on my face" et créateurs du Ministère des démarches singulières*<sup>1</sup>).

Quand vous lancez Python<sup>2</sup>, vous vous trouvez en "mode interactif", détectable par la présence d'un prompteur marqué par `>>>`. Vous n'êtes donc pas encore en mode programmation en train de créer un programme avec des lignes et des lignes qui ne sont exécutées qu'une fois que vous lancez le programme. Tout au contraire, vous allez taper des lignes une par une qui seront exécutées chacune à leur tour. Enfin, elle seront exécutées si vous utilisez la bonne syntaxe qu'il faut apprendre peu à peu.

Vous pouvez faire de simples calculs, comme des additions, multiplications, divisions, exponentiations (*on dirait "élévation à une puissance" si on ne connaissait pas le vrai mot*).

Par exemple, tapez `2+4+5-6` et regardez ce que Python affiche alors (pour qu'il affiche, faites un retour chariot avec la classique touche "enter").

Essayez avec `2*3+5*7` et constatez qu'effectivement, Python utilise les mêmes règles de priorité que vous.

Mettez aussi des parenthèses : `(2+4)*5-6*7` et aussi des nombres non entiers : `4.6+7.2`.

Glissez aussi une variable littérale : `4+3*a+5*b`. Déception... Python n'est pas un langage de calcul formel.

Essayez une chose non autorisée : `5/0`. Soulagement.

Il vous faudra ensuite comprendre la différence entre les deux divisions possibles : `/` et `//`. La première s'applique aux nombres, entiers ou réels, la seconde s'applique uniquement aux entiers. Testez : `33/7`, `43.2/7.2` (*c'est utile ça ?*), mais aussi `33//7` et `43//7`.

Tant que vous y êtes, cherchez le reste de la division de 43 par 7 autrement que par `43-7*(43//7)` (*le symbole % aura un rôle*). Surveillez aussi les règles de priorité avec `5*42//15` et `(5*42)//15`.

Si vous avez déjà une idée de ce que sont les différents types, voyez si `42/7` est un entier, de même que `42./7.` et `42./7.`

La question des puissances se pose aussi. méfiez vous, vous avez peut être l'habitude de taper `a^b` pour calculer  $a^b$ . Mais que se passe-t-il si vous validez `2^3` par exemple ? Etrange, non ?

Allez, il faut le savoir, l'exponentiation de  $a$  à la puissance  $b$  se fait par `a**b`. Vérifiez avec `3**2`, `5**3` et autres. Et vérifiez les règles de priorité : `4**3+1`, `4** (3+1)`, `2*3**2`, `(2*3)**2`, `3**2**3`, `(3**2)**3` et `3** (2**3)`.

Vous voulez calculer  $\sin(\pi)$ . Vous tapez donc `sin(pi)`. Et rien ne va... Alors quoi ?

Alors, c'est juste que Python ne connaît a priori ni `pi` ni `sin`. Pourquoi ? Parce qu'il est destiné à un large public qui par exemple concevra des sites Web, des utilitaires pour comptabilité ou gestion de stocks... Il accepte bien de connaître les fonctions et constantes mathématiques, mais il faut pour cela lui dire d'aller les chercher dans un fichier qu'il a en réserve sur disque dur. Il existe donc un module `math` qui contient les fonctions usuelles (*trigonométrie, exponentielle, logarithme, factorielle, densité gaussienne, trigonométrie réciproque et trigonométrie hyperbolique...*), de même qu'il existe des modules dévolus aux tirages aléatoires, à l'interaction avec le système d'exploitation, à la gestion du temps, au web...

<sup>1</sup>Le nom du logiciel Python est un clin d'oeil de ses concepteurs au *Monthy Python Flying Circus*, troupe d'humoristes britanniques des années 70 dont certains sévissent encore

<sup>2</sup>icône DrPython du bureau, et une fois dans ce logiciel, tapez sur l'icône verte en forme de serpent

Pour charger des fonctions du fichier `math` et les utiliser ensuite, on peut par exemple taper `from math import pi`. Alors, le nombre  $\pi$  est connu de Python (*jusqu'à ce que vous redémarriez*) sous le nom de `pi`. Vérifiez en demandant alors `pi`.

Mais si vous devez importer aussi le sinus, le cosinus, la tangente,, et ainsi de suite, ça risque d'être long. C'est pourquoi je vous recommande `from math import *`, qui signifie "du module `math`, importe tout". Vérifiez avec `from math import *` suivi de `sin(pi)`, `tan(pi/4)`, `exp(0)`, `e/exp(1)`, `sqrt(49)`, `sqrt(5)**2`, `sqrt((-5)**2)`.

Allez, on progresse. On va créer des variables, qui porteront un nom et auront donc une valeur. Tapez `a=5` (validez...), et tapez ensuite `3*a+2`. Vous comprenez que l'affectation se fait donc par un simple signe égale (vous évitez les `:=` de certains langages de programmation que j'utilise encore souvent pour ma part).

Vous pouvez ensuite modifier la valeur de votre variable. Tapez `a=6`, puis `2*a+3`.

Vous avez une autre variable à affecter : tapez `b=7` puis `3*a+2*b` (ou d'autres formules, ne soyez pas "suivistes").

Vous avez deux variables à affecter ? Vous pouvez le faire en une seule fois. tapez `c, d=2, 3` (avec un seul signe égale, mais deux virgules), puis tapez ensuite `print(2*c+4*d, c*d, c**5//d)`.

J'en profite pour vous indiquer qu'avec la fonction `print`, vous pouvez ainsi demander à Python d'afficher plusieurs valeurs en une seule fois, ce qui vous fait gagner du temps.

Vous noterez qu'on comprend avec ces instructions d'affectations simultanées par "virgule et un seul égale" pourquoi une approximation de  $\pi$  à  $10^{-5}$  près va se noter `3.15159` et pas `3,14159`.

L'avantage de ces affectations simultanées est que vous évitez les mauvaises surprises de l'échange de deux variables.

Un problème connu en programmation est bien le piège pour l'échange de deux variables que je vais appeler *u* et *v*.

Le réflexe trop rapide/naïf, c'est de taper `u=v` puis `v=u`. Le résultat est que la variable *u* est per-

due. Vérifiez le en imaginant par exemple que *u* vaut 2 et *v* vaut 3 :

| instruction      | <i>u</i> | <i>v</i> |
|------------------|----------|----------|
|                  | 2        | 3        |
| <code>u=v</code> | 3        | 3        |
| <code>v=u</code> | 3        | 3        |

Le réflexe du programmeur "à l'ancienne" est d'utiliser une variable tampon : `tampon=u`, puis

`u=v` puis `v=tampon`. On vérifie avec un tableau :

| instruction           | <i>u</i> | <i>v</i> | tampon |
|-----------------------|----------|----------|--------|
|                       | 2        | 3        | ?      |
| <code>tampon=u</code> | 2        | 3        | 2      |
| <code>u=v</code>      | 3        | 3        | 2      |
| <code>v=tampon</code> | 3        | 2        | 2      |

Avec Python, tout va vite : `u, v=v, u` et c'est fini. Vérifiez par vous même avec cette instruction, puis `print(u, v)`.

Faites ensuite le test avec trois variables que vous mélangez.

Que se passe-t-il si vous tapez d'abord `a, b=1, 1`, puis plusieurs fois l'affectation double `a, b=b, a+b` ?

Bien sûr, vous ne devez pas être surpris par une instruction telle que `a=2*a`. Elle consiste à doubler la valeur de la variable *a*.

De même, avec `a=a+2`, vous n'écrivez pas une équation absurde, mais vous augmentez *a* de 2 unités.

Vous aurez souvent besoin d'augmenter des variables qui servent de compteur dans une boucle. L'instruction usuelle est `i=i+1`. Mais on peut la raccourcir par `i+=1` qui semble étrange les premières fois où on la croise dans un programme.

Un petit test : tapez déjà `a=2`, puis tapez `a==2` et tapez aussi `a==3`. Que signifie ce double égale ? Allez, on essaye : `a=3` puis `a**2==9`.

Oui, le double égale, c'est le test, celui que fait l'ordinateur avec réponse **True** ou **False**. En mode interactif, c'est fort peu intéressant, mais en programmation, c'est vite utile.

A ce stade, juste pour voir si vous avez compris. Lesquelles de ces instructions vont être comprises par Python, et que répondra-t-il : `a=3`, `a==6/2`, `(a==6)/2`, `t=(a==5)`, `t==a=5`, `t==a==5`, `t=a=5`, `(t=a)==5`.

On peut aussi faire des test pour une relation d'ordre : `a<10`, `a>-4`.

Je vous laisse le soin de faire vous même des tests pour comprendre ce qu'est le symbole `!=`.

Si vous avez l'envie d'approfondir, faites des tests plus poussés avec des connecteurs logiques comme **and**, **or** et **not**.

---

Heureuse surprise, Python connaît les complexes. Comment lui définit-on un complexe ? Avec le mot **complex** justement. Essayez `z=complex(real=2,imag=4)` et validez. Notez que Python parle comme nos amis physiciens et note *j* la célèbre racine de  $-1$ , et non pas *i*.

Pour simplifier, contentez vous de `zz=complex(1,-2)`. C'est quand même plus pratique.

Vérifiez ensuite que Python calcule correctement : `z+zz`, `2*z+3*zz`, `z*zz`. Essayez même `zz**2` ou `z**5`, et pourquoi pas `z/zz`.

Si vous savez déjà poser des questions à Python sur les types, vérifiez si pour lui `complex(1,1)**4` est un complexe ou un réel.

Vous voulez extraire la partie réelle ou la partie imaginaire d'un complexe ? Vous allez commencer sans le savoir à travailler sur des objets : tapez `z.real`, `zz.imag`.

Python sait il extraire des racines carrées de complexes (essayez `sqrt` et "puissance 1/2") ? Connaît il la notion de conjugué ?

---

On va maintenant utiliser les "boucles" **for** pour calculer la somme des *n* premiers entiers, la somme des carrés, ainsi que *n!* et autres petits objets du même type.

Si j'écris "boucle" avec des guillemets, c'est parce que la syntaxe Python est un peu spécifique. On n'a pas de boucle en `for i=1 to n` ou autre modèle tel qu'on le croise en C++.

La syntaxe de Python est `for i in range(n)` :. Cette instruction donne un entier qui va de 0 à *n* - 1 (ce qui fait quand même *n* indices). Les deux points juste après mon `range(n)` sont ils une erreur typographique ? Pas du tout, c'est pour marquer le début de la séquence d'instructions.

Par exemple, tapez `for k in range(10):print(k)`. Que voyez vous ? Comprenez vous pourquoi vous devez taper deux "enter" ?

Initialisez une variable *s* à 0 (oui, c'est juste `s=0`), puis tapez `for k in range(10):s=s+k print(k,s)` en pensant à mettre quelques "enter" aux bons moments.

Que se passe-t-il si vous n'initialisez pas *s* à la valeur 0 ? Essayez donc avec `for k in range(10):s=s+k print(k,s)`.

Comment pouvez vous modifier ce petit script pour calculer la somme des 120 premiers entiers ?

Comment pouvez vous modifier ce petit script pour calculer la somme des carrés des cent premiers entiers ?

---

Exercice : utilisez une "boucle" **for** pour calculer  $\prod_{k=0}^{20} (2.k + 1)$  par exemple.

---

Un élève qui pense avoir compris se propose de calculer 40!. Il tape alors `p=0` puis `for k in range(40):p=p*k` et enfin, il demande la valeur de *p* par `print(p)` par exemple. Il a ici commis plusieurs erreurs. Trouvez les, corrigez les.

Bien sûr, il y a une fonction toute prête dans le module **math**. Mais c'est quand même la base

même de la programmation que de concevoir soi-même une fonction factorielle.

Cependant, à ce stade, on n'a pas créé une fonction. Il faut retaper le petit script chaque fois qu'on change la valeur de  $n$ . Il serait temps de créer ses propres fonctions. On va donc définir (syntaxe : **def**) une fonction factorielle (ou tout autre nom idiot, mais ce serait vraiment stupide de la nommer *cosinus*). On va donc passer l'entier  $n$  comme variable.

On commence par **def factorielle(n)** : avec les deux points suivis d'un "enter" qui font passer à la ligne avec indentation<sup>3</sup>.

On continue avec **p=1** (oui, c'était ça une des erreurs élémentaires plus haut). On passe à la ligne suivante et on constate que Python maintient une belle cohérence dans les indentations.

On annonce la boucle : **for k in range(n)** : (dans laquelle la valeur de  $n$  sera celle que vous aurez passée quand vous taperez *factorielle(13)* par exemple).

On effectue la boucle : **p=p\*(k+1)** (notez la nouvelle indentation automatique de Python et vérifiez que le  $k+1$  à la place de  $k$  correspond aussi à une des erreurs détectées).

On sort de la boucle en allant bien à la ligne, mais en revenant en arrière d'une indentation (on n'est plus dans la boucle **for**) : **return p** (vous prendrez l'habitude de ce **return** qui indique à Python ce que vous souhaitez qu'il "dise à voix haute" à la fin de la procédure). On sort définitivement avec un "enter" et un "back-space".

Il est fort probable qu'une erreur se sera glissée dans tout ce que vous aurez tapé, mais je suis à pour vous aider, appelez moi (mais poliment, merci).

Si toutefois tout a été bien tapé, vérifiez avec **factorielle(3)**, **factorielle(4)** et autres. Tant que vous y êtes, pourquoi pas **for n in range(12) : print(n, factorielle(n))** ?

La question qui peut venir à l'esprit est alors : jusqu'à combien peut on aller ? Les résultats seront ils exacts ? Je vous renvoie au cours théorique sur les entiers et réels avec Python.

Malgré tout, méfiez vous du temps que va mettre Python pour effectuer des calculs de plus en plus complexes. Il sera peut être temps de prendre connaissance de la touche qui arrête un programme en cours d'exécution.

---

La syntaxe concurrente de la boucle **for** est la boucle **while**. La traduction est "tant que". Elle se rédige par **while (condition) : (instruction)**.

Si la condition n'est pas vérifiée, l'instruction n'est même pas effectuée. Sinon, on l'effectue tant que la condition n'est pas validée. Il va de soi que les variables intervenant dans la condition doivent être modifiées par l'instruction, sinon vous entrez dans une "boucle folle".

En effet, ne tapez surtout pas le script suivant : **k,n = 0,0 while k<1 : n=n+1**. Comprenez vous pourquoi ?

Vous pouvez en revanche taper **k,n=1,1 while k<10 : n = n\*k k=k+1** (avec les "enter" au bon endroit).

Je vous propose même d'analyser et comprendre le script suivant (et seulement ensuite de le taper) :

```
k,p = 1,1
while p<1000000 :
    ...p=p*k
    ...k=k+1
print(k)
```

Exercice de programmation : si  $L$  est une liste de variables (indexée de 0 à  $n-1$ ), que fait le petit script suivant :

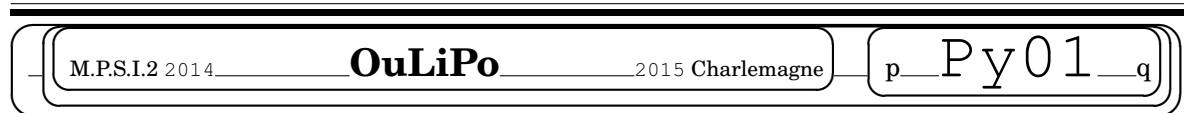
---

<sup>3</sup>l'indentation, c'est le "saut en début de ligne" de plus ou moins de caractères blancs

```

k=0
while L[k] !=a and k<n :
    .... k=k+1
if k=n :
    ....print(...)
else :
    ....print(...)

```



L'**OuLiPo** est un **Ouvroir de Litterature Potentielle**. C'est un groupe de créateurs qui sont (*ou étaient*) à la fois fous de mathématiques et de la langue française. Leur but est de créer de la littérature sous contraintes plus ou moins mathématiques. Ses créateurs en furent Raymond Queneau, Georges Perec, François LeLyonnais et quelques autres depuis disparus, mais l'équipe s'est sans cesse renouvelée. Des séances en réunissent encore les membres très régulièrement à Paris.<sup>4</sup>

Quelques exemples de créations :

- des lipogrammes (*textes où certaines lettres n'apparaissent jamais*), comme le livre "la disparition" de Perec, dans lequel il n'y a aucune lettre E <sup>5</sup>(suivi des "revenentes" où l'unique voyelle utilisée est le E)
- des palindromes (*textes qui se lisent dans les deux sens*),
- des histoires écrites sous une contrainte de théorie des groupes ou de théorie des graphes,
- des ré-écritures de textes célèbres dont les mots sont remplacés par leur définition dans le dictionnaire, par des synonymes, le mot de position  $n + 7$  dans le dictionnaire...

Je vous propose ici quelques recherches avec Python inspirées des oeuvres oulipiennes :

- vérifier qu'un texte est un palindrome
- vérifier qu'un texte est un lipogramme
- créer une machine à engendrer quelques uns des  $10^{12}$  sonnets de Queneau
- chercher le mot utilisant le plus de lettres différentes dans un texte donné,
- tester si un texte est un pangramme.

Plusieurs de ces programmes auront besoin d'accéder à des fichiers de textes déjà saisis, et c'est en cela qu'il s'agira ici d'exercices formateurs pour nos travaux pratiques de programmation.



<sup>4</sup>existent aussi l'OuLiPoPo (*Ouvroir de Litterature Policière Potentielle*), et l'OuBaPo (*Ouvroir de Bande Dessinée Potentielle*)

<sup>5</sup>lisez ce livre, et cherchez même le paragraphe ne contenant ni E ni A ; pour information, à sa sortie, des critiques littéraires n'ont même pas vu/lu que c'était un texte sans E

Un palindrome est un texte qui se lit dans les deux sens, comme “Esope reste ici et se repose”, “Elu par cette crapule” ou le très long texte de Georges Perec de plusieurs lignes (*cherchez sur internet ou dans les fichiers que je vous ai donnés*).

On vous donne un texte fait d’une très longue chaîne que nous allons appeler **Texte**.

Il vous faut/suffit a priori vérifier caractère par caractère : **Texte[k]=Texte[L-k-1]** <sup>6</sup> avec **L=len(Texte)** (*rappelons que k peut aller de 0 à L - 1*). Mais ça n’est pas si simple.

Il ne faut pas tenir compte des espaces ni de la ponctuation. Il ne faut pas tenir compte non plus de la casse (*majuscule/minuscule*) ni des accents. En effet, sans ces considérations, les seuls palindromes possibles sont Laval (*et encore, sans majuscule*), aha ou autres simples mots.

Vous allez donc parcourir la liste de **Texte** en sautant les espaces, ponctuations... et votre fonction de comparaison de deux caractères devra ne pas tenir compte de l’accentuation.

Les fonctions utiles :

- **len** pour la longueur d’une liste
- **ord** pour chercher le code ASCII d’un caractère donné
- **chr** pour revenir d’un entier (*code ASCII*) au caractère
- les majuscules ont un code de 65 à 92 et leurs minuscules associées ont un code augmenté de 32
- pour effacer les lettres accentuées, je vous propose des choses sur le modèle **if lettre in “éèêë” : lettre=“e”**
- pour effacer les ponctuations, pensez à la fonction **delete**

Struture du programme :

une boucle **for k in range(...)**

vous extrayez le caractère **Texte[k]** que vous appelez **lettre**

vous regardez si c’est une lettre accentuée auquel cas vous la remplacez (*tests indiqués plus haut*)

vous convertissez en minuscule si c’est une majuscule par un test **if ...<ord(lettre)<... : lettre=chr(ord(lettre)-...)**

si lettre est une lettre, vous la collez au bout de la chaîne que vous construisez par **chaîne=chaîne+lettre**, sinon, vous ne collez rien (*structure en **if lettre in alphabet : ...**, pour laquelle vous aurez pris soin de définir **alphabet** comme le mot de vingt six lettres allant de a à z*)

ensuite, vous comparez la chaîne obtenue à sa chaîne renversée par des tests du type **chaîne(k)=chaîne(-k)**, sachant que quand on donne un indice négatif à Python, il lit la chaîne à partir de la fin.

Attention toutefois, les bons indices ne sont pas **chaîne(k)** et **chaîne(-k)**, réfléchissez un peu.

Variante : plutôt que de faire ce test, créez tout de suite deux chaînes : une dans le sens direct et l’autre dans le sens indirect.

Il vous faudra pour cela utiliser **chaîne=chaîne+lettre** mais aussi **eniahc=lettre+eniahc** qui collera les lettres au début de la chaîne renversée. Il ne vous restera plus qu’à tester **chaîne=eniahc**

---

Je tiens à votre disposition (*uniquement avec accord parental*) un fichier contenant une liste de palindromes de mon ami Gerard Durand (*cherchez le sur internet sur le site fatrasie*), quelques peu osés hélas<sup>7</sup>. Vous devrez alors prendre une à une les phrases et en tester la palindromie (*certaines sont des palindromes phonétiques et non calligraphiques, ce qui change tout*). Evidemment, ce ne sera pas à vous de les recopier un à un.

C’est en fait l’occasion d’apprendre la lecture de fichiers sur le disque.

• avec **Durand=open(palindromes.txt, ‘r’)** vous allez chercher le fichier appelé **palindromes.txt** et vous l’ouvrez sous le nom **Durand**, en mode lecture (pas d’écriture possible sur le fichier, pas de destruction...<sup>8</sup>)

---

<sup>6</sup>remarque : avec juste **Texte[-k]**, on accède directement à l’élément d’indice **k** à partir de la fin

<sup>7</sup>hélas, vraiment ?

<sup>8</sup>vous verrez plus tard avec S.Chevalier qu’il existe des modes écriture, ajout

- avec `Durand.close()` vous fermez `Durand` à la fin de votre travail (prudence élémentaire)
- avec `Durand.readline()` vous extrayez la première ligne du fichier `Durand` (à vous de l'imprimer ou de la mettre dans `Texte`)
- en mettant en boucle `for` l'instruction `Durand.readline()`, vous lisez les lignes les unes après les autres

MPSI 2/2014

## Lipogrammes

Py01

Que doit faire votre programme? Ouvrir un fichier déjà tapé (voir ci dessus), le parcourir et vérifier que nulle part ne figure une certaine lettre passée en variable. Ce sera donc un simple parcours du type `for`.

MPSI 2/2014

## Pangrammes

Py01

Un pangramme est un texte court contenant au moins une fois chaque lettre de l'alphabet : "Portez ce vieux whisky au juge blond qui fume".

La mission de votre programme ou fonction :

vérifier que chaque lettre de l'alphabet intervient bien au moins une fois dans le texte.

Vous devrez déjà engendrer un alphabet en minuscules, que vous appellerez `ALPHABET`. Soit que vous tapez un mot les contenant toutes, soit que vous utilisez une boucle `for`, sachant que les voyelles ont un code ASCII allant de 97 à 122 et que c'est la fonction `chr` qui crée la lettre à partir du nombre.

Ensuite, vous pouvez parcourir le texte par une boucle `for` avec comme `range` la longueur du texte. Vous lisez chaque lettre et vous l'effacez de la liste `ALPHABET`. Je vous laisse voir si vous devez utiliser `liste.remove(...)` ou `liste.pop(...)` ou encore une autre fonction. Vous regardez à la fin si finalement `ALPHABET` est vide.

On peut aussi parcourir `ALPHABET` et vérifier si chaque caractère de alphabet est présent dans texte (utiliser `in`).

Un problème encore : il faut, comme pour les palindromes, remplacer les éventuelles majuscules par des minuscules, afin de ne pas faire de distinction entre A et a par exemple. Et comme vous avez bien travaillé avec les palindromes, je vous vends la mèche : avec `texte.lower()`, on convertit un texte quelconque en sa version en minuscules.

Monsieur Jack, vous dactylographiez bien mieux que votre ami Wolf.  
Buvez de ce whisky que le patron juge fameux.

M.P.S.I.2 2014 **STERN-BROCOTT** 2015 Charlemagne

p Py02 q

Il existe une suite simple qui "par miracle" parcourt de manière bijective l'ensemble  $\mathbb{Q}^{+*}$  (c'est à dire que chaque rationnel strictement positif est atteint une fois et une seule par cette suite).

Sa définition en est simple :  $b_0 = 1$  et  $b_{n+1} = \frac{1}{2 \cdot [b_n] - b_n + 1}$

Vérifiez que tous les termes de la suite seront bien rationnels et positifs <sup>9</sup>

<sup>9</sup>si je vous avais demandé "vérifiez que tous les termes de la suite seront bien des rationnels positifs", vous n'auriez

Testez “à la main” les premières valeurs.

---

Votre mission sera ici de générer les premières valeurs de cette suite, d’en vérifier l’injectivité sur les premières valeurs...

Vous devrez donc définir la fonction  $\text{stern} : x \mapsto \frac{1}{2.[x] - x + 1}$  où les crochets désignent donc la partie entière (*j’aurais du vous le dire plus tôt*).

Pour définir une fonction avec son nom et ses variables, c’est **def nom(variables)** :

S’il n’y a pas de variables, c’est **def nom()** : (*n’oubliez pas les deux points suivis du retour à la ligne avec indentation, ils font partie de la syntaxe*). S’il y en a plusieurs, c’est **def nom(variable, variable)** : dans un premier temps (*on verra plus tard qu’on pourra avoir intérêt à leur donner des noms pour pouvoir les passer dans le désordre ou n’en passer que certaines, et qu’on pourra même leur donner des valeurs par défaut si l’utilisateur ne veut les spécifier que dans certains cas*).

Ensuite, vous faites les calculs que vous voulez (*y compris des tests, des appels à d’autres fonctions...*) puis vous précisez par **return(...)** ce que la fonction doit finalement renvoyer (*là encore, il peut s’agir de plusieurs variables, si vous avez une fonction de  $E \times E'$  dans  $F \times F' \times F''$  par exemple*). Attention, c’est bien **return(...)** et pas **print(...)** qui afficherait une valeur mais ne la donnerait pas au programme pour en faire quelque chose.

Ici, comme la fonction est simple, on se limitera sans doute à

```
def stern(x) : return(1/(...))
```

sachant que vous devrez quand même rechercher sous quel nom se cache la fonction partie entière en Python (**round, trunc, floor, int, intpart...** à vous de trouver).

Ensuite, vous créez une boucle de taille prédéterminée par **for k in range(n)** :, et à chaque itération, vous remplacez *b* par *stern(b)*. Attention toutefois, si vous n’avez pas initialisé *b*, le programme va refuser de tourner.

---

Cette première version hélas ne donne pas satisfaction et “diverge” très vite par rapport à vos calculs “à la main”.

Pourquoi? A vous d’y réfléchir.

Comment remédier à ce problème?

Souvenez vous que tous les termes de la suite sont rationnels. Il vaut mieux donc les mémoriser sous la forme  $\frac{\text{numer}}{\text{denom}}$  avec *numer* et *denom* deux entiers naturels. La fonction **stern** va donc prendre deux variables (*n* et *d*) et retourner deux valeurs (**return(..., ...)**). Je vous laisse faire le calcul du numérateur et du dénominateur de la fraction **stern(numer/denom)** “à la main”. Je vous rappelle juste que le quotient d’une division euclidienne s’obtient par **dividende//diviseur**.

Ensuite, la mise en boucle par **for k in range(n)** : reste valable.

---

Il faut ensuite perfectionner ce programme pour quelques tests et visualisations.

On peut se donner un rationnel (*sous forme num/den irréductible*) et se poser la question : “au bout de combien d’étapes est il atteint?”.

C’est donc l’occasion d’expérimenter un autre modèle de boucle : la boucle conditionnelle **while**. Elle s’exécute tant qu’une condition est réalisée. Ici, la condition, c’est **b != (num, den)** en rappelant que **!=** est le symbole de test de non égalité (*l’égalité en test, c’est ==*).

On va donc avoir une structure en **while b != (num, den) : b=stern(b)**.

Un tel script va poser quelques problèmes. On pourra le voir s’arrêter quand la valeur rationnelle

---

peut-être prouvé que la moitié des choses...

est atteinte, mais il n'affichera pas alors le nombre d'étapes. Il va donc falloir à chaque boucle exécutée incrémenter un compteur d'une unité par `compt=compt+1` ou sa syntaxe allégée `compt+=1` (si vous la trouvez vraiment allégée de la sorte, faites moi signe). Il ne restera plus qu'à afficher en sortie de boucle la valeur de ce compteur par un `print(...)`.

Au fait, avez vous pensé à initialiser `compt` ?

Il reste quand même un risque sur une telle boucle conditionnelle. Si le rationnel que vous attendez est affreux (quand la suite de Stern atteint elle  $\frac{1765459789}{324}^{10}$ ) ou si vous l'avez mal saisi (numérateur ou dénominateur négatif ou non entier). Il faut donc doubler ce test d'un autre test `compt < max_compt` où `max_compt` est un entier  $N$  qui vous évitera de devoir ne finir ce T.D. que dans trois jours. On aura donc une boucle en `while ... and ... :`

Vous pourrez aussi par concaténation créer la liste des  $N$  premiers termes de la suite de Stern en ajoutant quelquepart une ligne `L=L+[b]` ou même `L+= [b]` ou encore `L.append(b)`. Il conviendra d'avoir pensé à initialiser cette liste.

---

Ensuite, vous pourrez faire divers tests :

- vérifier qu'il n'y a pas de terme en double dans `L`
- compter le pourcentage de termes plus petits que 1
- ordonner cette liste ou la trier selon d'autres critères (les entiers, les inverses d'entiers...)
- répartir ces rationnels dans le tableau à double entrée (numer, denom), les modules graphiques `turtle` et `tkinter` pourront alors vous servir
- estimer le temps qu'il faut pour passer d'un rationnel  $p/q$  au rationnel  $q/p$  suivant la complexité de l'écriture  $p/q$
- voir ce qu'il se passe quand on ne garde qu'un terme sur deux ou un terme sur trois...

---

MPSI 2/2014

## Ein Stern ist geboren

Py02

La suite de Stern est définie par  $\varphi(0) = 0$  et  $\varphi(1) = 1$  puis  $\varphi(2.n) = \varphi(n)$  et  $\varphi(2.n + 1) = \varphi(n) + \varphi(n + 1)$  pour tout  $n$ .

On donne pour comprendre les premières valeurs : (0, 1, 1, 2, 1, 3, 2, 3, 1, 2, 3, ...).

Créez une procédure Python qui pour un entier  $n$  donné calcule  $\varphi(n)$ .

Créez une procédure qui calcule les  $n$  premiers termes de la suite.

Créez une procédure qui calcule  $\varphi(2^n)$  pour tout  $n$ .

Ecrivez ensuite un petit script qui va afficher les  $N$  premiers termes de la suite  $\left(\frac{\varphi(n)}{\varphi(n+1)}\right)$  (il s'agit ici de la suite de Stern-Brocott déjà croisée en Python et dans le chapitre -1 qui énumère les rationnels positifs).

---

M.P.S.I.2 2014

## FEU DE FORET

2015 Charlemagne

p Py03 q

---

<sup>10</sup>et d'ailleurs, ce nombre est il sous forme irréductible ?

On se propose de modéliser et visualiser la propagation d'un feu de forêt dans un milieu boisé. C'est un exercice assez classique, que l'on a pu croiser dans des sujets de concours.

- 
- Le domaine sur lequel on travaille sera un tableau (*qu'on appellera **zone***) fait de cases, de taille fixée à l'avance.
  - on aura donc deux constantes à prédéfinir **imax** et **jmax** et les calculs se feront avec des **for i in range(imax) :** et **for j in range(jmax) :**, les éléments s'appelleront alors **zone[i][j]**.

Avec **[0]\*N** on crée une ligne **[0, 0, 0...0]** (*N fois*).

Avec **[[0]\*N]\*P** on pense qu'on crée *P* listes **[0, 0, 0...0]** côte à côte, sous la forme attendue **[[0, 0, 0], [0, 0, 0], [0, 0, 0]]**. Mais il y a un piège qu'il faut connaître. Créez justement **zone=[ [0]\*imax]\* jmax**. Affichez **zone**. Tout semble parfait. Modifiez un élément : **zone[2][3]=0**. Que constatez vous ?

Il faudra donc coller des lignes par **[... for j in range(jmax)]**.

- 
- Chaque cellule pourra être dans plusieurs états :
    - \* **0** pour l'état normal
    - \* **1** et plus pour "en feu", l'entier désignant depuis combien de temps la cellule est en feu
    - \* **-1** pour brûlé (*et ne pouvant donc plus être en feu*)
  - L'entier **zone[i][j]** pourra donc prendre l'une ou l'autre de ces valeurs.
  - L'état initial du domaine devra donc être 0 sauf pour quelques cellules en lesquelles le feu prendra.
  - Une solution consistera à parcourir le tableau case par case et à donner à **zone[i][j]** la valeur 1 avec une certaine probabilité *p*.  
Pour effectuer un tirage avec probabilité  $1/N$ , le plus simple est d'utiliser un test du type **if randrange(N)==0** (*on rappelle que **randrange(N)** tire un entier entre 0 et N-1 ; on teste donc le cas où l'on tombe sur 0<sup>11</sup>*).
  - A chaque étape, on étudie cellule par cellule **zone[i][j]** :
    - \* si elle brûlée, on la laisse telle quelle
    - \* si elle est en feu, on augmente sa valeur de l'entier positif pour tenir compte du temps écoulé ; de plus, si elle brûle depuis trop longtemps, elle achève de se consumer et on passe à l'état "brûlé" (*c'est à dire **zone[i][j]=-1***) ; il y aura donc un test du type **if zone[i][j]>duree** où l'entier **duree** est une constante définie dès le début du programme.
    - \* si elle est à l'état normal, on regarde si elle risque de prendre feu. Pour ce faire, on compte combien de ses voisins sont en feu ; si ce nombre de voisins dépasse une valeur critique (*disons trois voisins sur les huit*), alors la cellule prend feu (*avec une probabilité *p'* s'il le faut*).
  - On effectuera donc un parcours avec **for i in range(...)** : **for j in range(...)** : **if zone[i][j]** ... Attention toutefois, il faudra prendre garde aux cases du bord (*i ou j nul ou égal au maximum*) : ces cases n'ont pas le bon nombre de voisins pour les tests. Je vous propose donc de ne pas passer sur ces cases du bord et de vous limiter à **for i in range(1,imax-1)** ..

Attention à l'erreur classique : il ne faut pas modifier l'état des cases **zone[i][j]** du tableau au fur et à mesure de leur lecture. En effet, quand vous lirez par exemple la case **zone[i+1][j]** et que vous regarderez l'état de ses huit voisins, certains seront dans leur état de l'ancienne étape et d'autres dans l'état de la nouvelle étape. Il faudra donc travailler sur une copie du tableau. Vous devrez donc avant le parcours définir **new\_zone=zone**, affecter les nouvelles valeurs à **new\_zone** (*en calculant le nombre de voisins en feu dans **zone***) et à la fin du parcours d'affecter **zone=new\_zone**.

---

<sup>11</sup>n'oubliez pas d'importer **randrange** du module **random**

◦ Il faudra ensuite à chaque étape afficher l'état de votre territoire. Vous pourrez bien sûr le faire avec un `print(zone)` sur lequel vous devrez lire par vous même les valeurs des diverses cases. Il sera quand même plus joli de faire un affichage avec des couleurs. Pour ce faire, vous devrez ouvrir une **fenêtre** avec `tkinter`, et y ouvrir un **canevas** pour image sur lequel vous dessinerez des carrés (ou **rectangle**).

La syntaxe est un peu alourdie, mais le jeu en vaut la chandelle (*enfin, le mot n'est pas judicieux avec une forêt en feu*).

On commencera avec `from tkinter import *` pour tout importer du module graphique `tkinter` (*vérifier en fonction de la version de Python*).

On créera une fenêtre par `fenetre=Tk()`.

On y créera un canevas par `foret=Canvas(fenetre,...)` et vous passerez comme variables sa taille (**height** et **width**) et sa couleur de fond (**bg** comme *background*).

Il faudra placer ce canevas dans la fenêtre (`foret.pack()` avec des options concernant l'endroit où on le place dans la fenetre).

On dessinera des rectangles avec `can.create_rectangle(...)` dans lequel vous passerez en valeurs les coordonnées du coin supérieur (**taille\*i** et **taille\*j**) et du coin inférieur (**taille\*(i+1)** et **taille\*(j+1)**) et la couleur donnée au rectangle (*en fonction de conventions à définir par vous même à partir de l'entier `zone[i][j]`*).

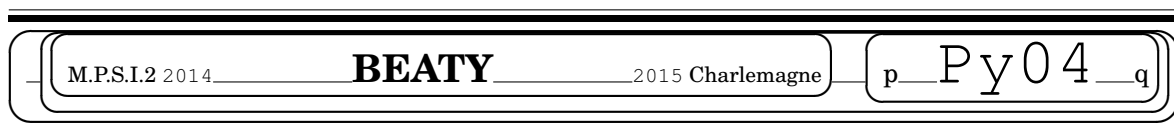
On lancera le tracé de la fenêtre par `fenetre.mainloop()`.

Vous aurez même la possibilité d'ajouter un titre à cette fenêtre (`fenetre.title("...")`) et un bouton interactif sur lequel cliquer pour passer d'une étape à la suivante.

`bouton=Button(fenetre, text="...", width=..., command=...)`

suivi d'un `bouton.pack()`, avec en option l'endroit où vous voulez le placer sur l'écran (**Top**, **Right**, **Left**, **Bottom**).

Je sais, c'est un peu lourd.



On se donne deux réels positifs  $a$  et  $b$ , irrationnels, vérifiant  $\frac{1}{a} + \frac{1}{b} = 1$ . Vérifiez que  $a$  et  $b$  doivent être plus grands que 1.

On considère alors les deux suites  $([n.a])_{n \in \mathbb{N}^*}$  et  $([n.b])_{n \in \mathbb{N}^*}$  (les crochets désignent la partie entière). Il s'agit de montrer (ou simplement de vérifier) que ces deux suites forment une partition de  $\mathbb{N}^*$  (c'est à dire que leur réunion refait  $\mathbb{N}$ , et que leur intersection est vide).

Ce que vous devez savoir faire :

- pour  $a$  donné, créer  $b$  associé
- prendre des parties entières,
- créer des listes,
- ajouter des termes au bout d'une liste (`maliste.append()`)
- les fusionner (*une addition suffit-elle ?*),
- étudier si la liste obtenue a des termes en double
- étudier si la liste obtenue recouvre bien tout un segment de  $\mathbb{N}$  (*il faudra peut être la trier, avec `liste.sort()`*).

Attention, ce n'est pas en prenant  $([a], [2.a], \dots [n.a])$  et  $([b], [2.b], \dots [n.b])$  (pour le même  $n$ ) qu'on va avoir des listes "compatibles". Ce n'est donc pas par un `for k in range(n): liste.append([k*a])`

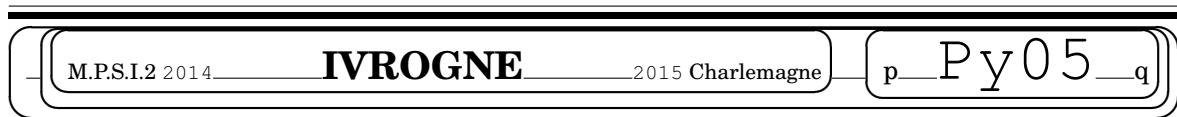
qu'on va travailler.

On va chercher à calculer les  $[k.a]$  tant que  $k.a$  est plus petit qu'un entier  $N$  donné. C'est donc avec une instruction **while** qu'on va fabriquer la liste des termes de la liste plus petits que  $N$  :

```
k=0 :
while k*a<N :
....listea.append([k*a])
....k+=1
```

On fait ensuite de même pour l'autre liste (*ou on crée une procédure dans laquelle on peut donner a ou b*).

Attention, pour  $N$  trop grand, les arrondis inhérents au format des réels dans Python produiront des erreurs.



Un classique des probabilités : la marche aléatoire d'un ivrogne dans une ville quadrillée comme la ville de New-York avec ses rues et ses avenues se coupant à angles droits. C'est en fait la modélisation "discrète" (à temps indexé par  $\mathbb{N}$  et non plus  $\mathbb{R}$ ) de ce que l'on appelle le mouvement brownien, qui intervient partout en probabilités (*et en finances*) de dimension 2.

Le personnage se déplace donc dans les rues qu'on peut assimiler aux droites parallèles à  $Oy$  (*abscisses entières*) et aux droites parallèles à  $Ox$  (*ordonnées entières*). Chaque fois qu'il arrive à un carrefour (point de  $\mathbb{Z} \times \mathbb{Z}$ ), il choisit au hasard la direction dans laquelle il continue son vagabondage. Son parcours est donc fait de segments de longueur entière, orthogonaux entre eux.

La modélisation : on part de  $[0, 0]$ , et à chaque étape, on ajoute au hasard  $[1, 0]$ ,  $[0, 1]$ ,  $[-1, 0]$  ou  $[0, -1]$ .

Pour choisir au hasard un entier entre 0 et 3 : **randrange(4)** que vous aurez chargé en début de programme par **from random import randrange**.

Question : combien d'étapes pour notre ivrogne ? Au choix : un nombre de l'ordre du millier choisi à l'avance ou "jusqu'à ce que l'ivrogne revienne à son point de départ".

Signalons que cette probabilité est égale à 1, mais cette garantie n'assure pas que votre mobile reviendra en  $[0, 0]$  en un temps raisonnable.

De plus, il est heureux que cette simulation se fasse en dimension 2, car la marche aléatoire de ce type sur  $\mathbb{Z}^3$  (*départ en  $[0, 0, 0]$  et huit vecteurs de déplacement en  $[\pm 1, 0, 0]$  et autres*) part "presque sûrement" à l'infini sans revenir au départ.

---

Une simulation, c'est gentil, mais il faut encore pouvoir en donner une vision assez claire pour l'utilisateur.

On utilisera alors le module "turtle". Pour cela : **from turtle import \*** puis

- **reset()** pour tout mettre à zéro
  - **goto(a, b)** pour se déplacer (*en écrivant ou non suivant l'usage de **up** et **down***)
  - **up()** pour lever le crayon et ne pas écrire quand on se déplace
  - **down()** pour baisser le crayon et pouvoir dessiner en bougeant
  - **forward(distance)**, **backward(distance)**, **left(angle)** et **right(angle)** qui se comprennent
  - **color(couleur)** pour changer de couleur (*quelle utilité ici ?*)
  - **width(epaisseur)** pour changer la largeur du trait (*initialement à 1*)
-

On peut autoriser le déplacement en diagonale en créant plutôt huit vecteurs de déplacement en  $[dx, dy]$  avec  $dx$  et  $dy$  dans  $\{-1, 0, 1\}$ .

On peut interdire de revenir sur ses pas.

On peut envisager un ivrogne qui se déplace sur un quadrillage hexagonal (*une abeille perdue au milieu des alvéoles à miel, qui aura trop butiné de raisin macéré*). A chaque étape, le mobile a six choix de direction possible, et les vecteurs de déplacement sont  $\begin{pmatrix} 1/2 \\ \sqrt{3}/2 \end{pmatrix}$ ,  $\begin{pmatrix} 1/2 \\ -\sqrt{3}/2 \end{pmatrix}$ ,  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  et leurs opposés.

Attention alors aux arrondis si votre test est “retour à la position  $(0, 0)$ ”.

Objectif : mélanger un paquet de cartes, ou une liste qui au départ est ordonnée.

On est loin du programme qui propose plutôt d’étudier les algorithmes de tri d’une liste de données désordonnée...

On suppose donc qu’on dispose d’un paquet de  $n$  cartes. Ce paquet peut être mémorisé pour l’instant sous la forme d’une simple liste des entiers de 1 à  $n$ . S’il le faut, il n’y aura plus qu’à en faire une liste de cartes à la fin par un petit algorithme (*voir plus loin*).

Comment va-t-on s’y prendre pour mélanger à la fois efficacement et rapidement ?

Un rappel déjà : on dispose sous Python d’une procédure de tirage aléatoire d’entiers. Il faut l’importer d’un module spécifique : `from random import randrange`. On l’utilise ensuite `k=randrange(n)` tire au hasard un entier entre 0 et  $N - 1$  (*rappelons que Python indexe tout à partir de 0, les éléments d’une liste de  $N$  termes sont donc indexés par des entiers de 0 à  $N - 1$* ).

On permute des cartes deux à deux dans la liste  $N$  fois de suite.

Pratiquement : par une boucle FOR effectuée  $N$  fois, on choisit deux indices aléatoires dans la liste  $i$  et  $j$  puis on échange  $L[i]$  et  $L[j]$ .

Attention, pour échanger deux éléments  $a$  et  $b$ , on n’effectue pas la double instruction `a=b` puis `b=a`. Avec certains langages, on doit passer par `c=a`, `a=b`, `b=c`. Avec Python, l’instruction `a, b = b, a` est parfaite.

Question ouverte : quelle valeur donner à  $N$  pour que le paquet soit “bien mélangé” ? Un  $O(n)$ ,  $O(n^2)$ ,  $O(n^a)$ ,  $O(n \times \ln(n))$ ...

On coupe le paquet  $N$  fois de suite en un point  $c$  compris entre 1 et  $n$ , tiré au hasard. Pratiquement, comme effectue-t-on une coupe sur une liste ? Avec Python, on peut

- prendre le début d’une liste jusqu’à l’indice  $k$  avec `liste[:k]`
- prendre la fin d’une liste à partir de l’indice  $k$  avec `liste[k:]`

- concaténer deux listes avec une simple addition.

Question devenant usuelle :  $N$  est elle une fonction linéaire de  $n$  ? Polynomiale ? Exponentielle ?

MPSI 2/2014

### 3-Extraction

Py06

On extrait les cartes une par une, au hasard, jusqu'à épuisement du paquet.

On doit donc  $n$  fois de suite choisir une carte au hasard dans le paquet, la mettre dans une liste en cours de construction, et l'effacer du paquet initial (*dont le cardinal va passer peu à peu de  $n$  à 0*). Avec Python, pour extraire l'élément d'indice  $k$  d'une liste, c'est l'instruction `pop(liste, k)` (*l'élément est retourné par la procédure, et de plus, il est éliminé de l'ancienne liste dont la taille se trouve alors réduite*). Et on ajoute un élément à la fin d'une liste avec `append` ou une addition.

Votre programme peut donc être très court...

MPSI 2/2014

### 4-Tri

Py06

Qua-

trième méthode : placer les cartes du paquet (*dans l'ordre où elle sortent*) sur  $p$  paquets en choisissant l'indice du paquet à chaque fois au hasard, puis poser les  $p$  paquets (*de taille moyenne  $n/p$  mais très variable de fait d'un paquet à l'autre*) l'un au dessus de l'autre.

Question : quelle valeur donner à  $p$  en fonction de  $n$  pour que le mélange soit "pertinent" ?

Autre question : ne serait il pas judicieux d'appliquer cet algorithme deux fois, puisque dans chacun des  $p$  paquets, les cartes sont quand même dans le même ordre que dans le paquet initial.

Cinquième méthode : associer à chacune des cartes une valeur tirée au hasard entre 1 et  $h$  (fonction de  $n$ ) et trier la liste en fonction de cette valeur.

Utiliser une boucle `for` pour donner les valeurs. Pour ce qui est du tri, suivant le niveau des élèves, utiliser un algorithme qui existe déjà, ou le créer de toutes pièces.

MPSI 2/2014

### Supplément

Py06

Petit algorithme de conversion.

On a une liste d'entiers de 1 à 52 et il s'agit d'en faire une liste de cartes. Chaque cartes est un couple (*couleur, hauteur*). La couleur, c'est  $\heartsuit, \clubsuit, \diamondsuit, \spadesuit$  et la hauteur c'est de 1 à 10 puis *valet, dame, roi*. Pour chaque entier  $k$  entre 1 et 52, on effectue la division euclidienne de  $k - 1$  par 13. Le quotient donne la couleur et le reste donne la hauteur. Par exemple pour  $k = 24$ , on trouve  $23 = 1.13 + 10$ . Le 1 (*entre 0 et 3*) donne  $\clubsuit$  et le 10 (*entre 0 et 12*) donne *valet*.

Vous pourrez utiliser les fonctions prédéfinies de division et congruences.

D'ailleurs, vous auriez pu dès le début créer une liste de cartes, avec deux boucles `for`.

MPSI 2/2014

### Bataille.

Py06

Pour

prolonger l'exercice : pourquoi ne pas alors couper le paquet de cartes en deux parts égales (*de taille  $n/2$* ), et jouer à la bataille. La bataille est un jeu où il est facile de faire jouer l'ordinateur contre lui-même. D'ailleurs, la seule donnée des deux listes de départ donne alors un jeu totalement "déterministe" n'ayant plus d'aléatoire ni de stratégie... D'où l'intérêt d'en faire un programme...

Pour prolonger l'exercice d'une autre façon : comparer les diverses façons de mélanger et estimer "l'entropie d'un mélange". Mais suivant quel critère(s) mesure-t-on la qualité d'un mélange ?

Mon fils est en C.E.2 et aime le calcul (*fonda*)mental. Il adore poser des petites opérations (*additions, soustraction, et même multiplications*).

Pour que je sois tranquille et puisse corriger vos copies sans qu'il ne vienne me réclamer des petits calculs à effectuer, je vais vous demander de me concevoir un petit programme interactif qui lui pose des équations sans en avoir l'air, telles que  $12 + \dots + 34 = 87$  lui demande sa réponse et lui indique si sa réponse est bonne.

Le principe sera donc la conception de formules à trou à compléter.

Votre programme devra tirer trois nombres au hasard **a**, **b** et **c** et calculer leur somme **s**, n'en afficher que deux et la somme et demander à mon fils la valeur qui manque.

Pour tirer des nombres entiers au hasard entre deux bornes min et max, on utilise **randrange** (**min**, **max**) (*faites des tests pour voir si min et max sont inclus*). Il faut penser au début du programme à importer cette procédure issue du module **random** (from random import \* ou from random import randrange).

Ensuite il faut afficher les entiers **a**, **c** et **s** avec les symboles +, + et = ainsi que le trou pour l'entier **b** à deviner. C'est **print** qui va servir, avec des virgules entre les différents termes à afficher et des espaces.

Sinon, vous pouvez rendre la visualisation plus attrayante en utilisant le module **Tkinter** si vous savez l'utiliser.

Il faut ensuite demander à mon fils la valeur qui manque, avec un **int** (**input** ("message")) (*le classique **int** est là pour que ce qui est saisi comme une chaîne de caractère soit converti en entier*).

Il ne reste plus qu'à comparer la valeur saisie avec la vraie valeur **b** par un test, afficher un message de félicitation (*ou le contraire*) et à proposer de recommencer.

Variante : au lieu de ne proposer que des additions, on peut aussi proposer de compléter des additions et soustractions :  $32 + \dots - 23 = 54$ .

Il faudra alors tirer non seulement les trois nombres, mais aussi les signes. Avec une probabilité quand même plus forte de tirer + que -. Pour avoir un événement avec probabilité 2/7 par exemple, je vous rappelle la méthode :

```
a = randrange(7)
if a < 2 :
    ....signe, val = '+', 1
else :
    ....signe, val = '-', -1
```

Reste ensuite à calculer la somme **s** en tenant compte des deux possibilités pour le signe. Je vous laisse saisir l'utilité de **val** ci dessus.

D'autre part, mon fils me pose des exercices inspirés de ce qu'il fait à l'école, où son intitutrice lui apprend la notation matricielle.

Je veux dire par là qu'il apprend à placer des points dans un tableau à double indice (*grilles de*

*mots croisés, matrice, tableur (pas) Excel(lent)...*

On lui donne un tableau à douze lignes (numérotées de 1 à 12) et douze colonnes (numérotées de A à L). Des symboles ou lettres sont inscrites dans quelques cases du tableau. Il faut alors qu'il indique les coordonnées de chacune. L'intitutrice vérifie alors si il ne s'est pas trompé. Là, ce sera au programme de le faire.

On va donc commencer par tracer avec **Tkinter** un tableau à douze lignes et douze colonnes sur un canevas (la valeur 12 sera appelée **N** et sera définie en début de programme, de sorte à ce qu'elle puisse être modifiée).

Je rappelle qu'il faut importer **tkinter** (ou **Tkinter** suivant votre version de Python). Ensuite, on crée une fenêtre **fen=Tk()**.

On crée ensuite un **Canvas** qu'on va appeler **can** et qu'on va placer dans la fenêtre **fen**. Il faut en donner la couleur de fond (**background='...'**) et la taille (**height = ..., width=...**)<sup>12</sup>. On va donc définir une échelle appelée **ech** et définie en début de programme pour pouvoir la modifier (ordre de grandeur : 20). La taille du Canvas sera donc **(N+1)\*ech** et **(N+1)\*ech**.

Pourquoi **N+1** ? Parce que l'on va aussi écrire en tête de ligne et de colonne les nombres et lettres. Pour tracer un trait sur un **Canvas** appelé **can**, c'est **can.create\_line()** dans laquelle vous passer les quatre coordonnées des deux extrémités des ladite ligne. On va donc mettre une boucle itérative un **can.create\_line(1, k\*ech, (N+1)\*ech, k\*ech)**.

On va ensuite placer les noms des lignes et colonnes sur le bord de ce tableau. Pour placer un caractère à un endroit (**a,b**) du **Canvas**, c'est **can.create\_text(a, b, text='...')**. Vous pouvez y ajouter des précisions comme la couleur, la police de caractère...

Attention, si **k** est l'indice de la ligne que vous tracez, il faut le convertir en chaîne de caractère pour l'afficher. Et n'oubliez pas que Python commence à compter à partir de 0 alors que François et Mme Bouvier comptent à partir de 1.

Pour ce qui est du choix de **a** et **b**, ne prenez pas **k\*ech** et **1**. pensez à traduire de **ech//2** pour que l'affichage se fasse au centre de la case.

Faites de même pour les colonnes en vous rappelant que **A** est le caractère 65 en code ASCII. Utilisez la fonction **chr()**.

Méfiez vous quand même, il faut réserver la première colonne et la première ligne pour l'indexation.

Il vous faudra ensuite taper la ligne **fen.mainloop()** pour que la fenêtre s'affiche.

Ensuite, avec le module **random**, vous tirez deux indices **i** et **j** au hasard entre 0 et **N** et vous placez un symbole dans la case de coordonnées (**i,j**) (on attend donc de votre part un **can.create\_text()** avec **i\*ech** et **j\*ech** plus une translation convenable).

Il ne vous reste plus qu'à interroger François en lui demandant dans quelle case se trouve chaque lettre, puis à tester la validité de sa réponse. Vous ferez donc usage de **input** pour la question, **int** pour la conversion en nombre, et des tests comparatifs **==** avec les valeurs **i** et **j**.

Pour faire joli, ajoutez

```
bou = Button(fen, text='Arret', command=fen.destroy)
bou.pack()
```

Ensuite, approfondissez cette idée pour la saisie de la réponse de mon fils.

Pour poursuivre : l'ordinateur affiche deux valeurs **i** et **j** et François doit cliquer sur la case demandée.

---

<sup>12</sup>dois je préciser que vous ne taperez pas stupidement et gentiment **background='...'** mais bien **background='blue'** ou toute autre couleur à votre guise. De même pour **height** et **width**.

L'objectif de ce T.D. est l'étude de l'évolution au fil du temps de deux populations dans un univers clos donné : des proies et des prédateurs.

Le point de départ est une étude de Vito Volterra au début du vingtième siècle<sup>13</sup>. Dans son ouvrage, il est parti des observations sur les relevés sur les populations de poissons et de requins dans une mer bordant l'Italie. Il y a donc modélisé ensuite les systèmes proies prédateurs et validé son modèle en le comparant aux observations. Il a ensuite étendu l'idée au cas de plusieurs populations en interactions mutuelles.

On note  $p_t$  le nombre de poissons à l'instant  $t$  et  $r_t$  le nombre de requins à l'instant  $t$  (*pas forcément entiers, car on travaille à une échelle macroscopique*).

S'il n'y a pas de requins, l'évolution de la populations de poissons/proies est  $p'_t = a.p_t$ . C'est la loi logistique classique : plus il y a de poissons, plus il y a de naissances.

En présence de requins, on remplace l'équation différentielle par  $p'_t = a.p_t - b.p_t.r_t$ . Ce terme négatif en  $p_t.r_t$  correspond au nombre de rencontres poissons/requins qui sont comme on s'en doute néfastes aux poissons.

En revanche, chez les requins, le terme en  $p_t.r_t$  est positif (*les requins se requignent*) et le terme malthusien en  $r_t$  est négatif (*faute de poissons, avec  $p_t$  nul, la population de requins s'éteindrait comme s'éteint un matériau radioactif*). Il n'y a évidemment pas de terme en  $\alpha.p_t$  dans  $r'_t$ .

On résume :  $p'_t = a.p_t - b.p_t.r_t$  et  $r'_t = -c.r_t + d.p_t.r_t$

On va utiliser Python ou tout logiciel de programmation et calcul pour simuler un tel système différentiel que l'on ne sait pas résoudre à l'aide des fonctions usuelles.

On va donc suivre la méthode d'Euler pour la résolution des équations, en découpant le temps normalement continu en des instants  $0, 0 + dt, 0 + 2.dt, 0 + 3.dt$  et ainsi de suite, avec  $dt$  "petit", pas trop mal choisi (si  $dt$  est trop grand, l'approximation à instants discrets est trop grossière, si  $dt$  est trop grand, la simulation est trop lente).

On se donne donc  $p_0$  et  $r_0$ . On calcule alors les deux accroissements  $dp_0$  et  $dr_0$  de valeurs  $(a.p_0 - b.p_0.r_0).dt$  et  $(-c.r_0 + d.p_0.r_0).dt$ . On incrémente alors  $p$  et  $r$  :  $p_{dt} = p_0 + (a.p_0 - b.p_0.r_0).dt$  et  $r_{dt} = r_0 + (-c.r_0 + d.p_0.r_0).dt$ .

On recalcule alors à ce nouvel instant les deux accroissements  $(a.p_{dt} - b.p_{dt}.r_{dt}).dt$  et  $(-c.r_{dt} + d.p_{dt}.r_{dt}).dt$ , puis les nouvelles valeurs.

En toute généralité :

$$p_{(n+1).dt} = p_{n.dt} + (a.p_{n.dt} - b.p_{n.dt}.q_{n.dt}).dt \quad r_{(n+1).dt} = r_{n.dt} + (-c.r_{n.dt} + d.p_{n.dt}.q_{n.dt}).dt$$

Evidemment, on ne va pas entasser toutes ces valeurs, mais on va les calculer de proche en proche par

**p = p + (a\*p-b\*p\*r)** et **r = r + (-c\*r+d\*p\*r)** (à effectuer simultanément).

Ce calcul sera mis en boucle (à vous de choisir entre une boucle `for` ou une boucle `while`).

Les valeurs pratiques que je vous propose mais que vous pourrez modifier pour voir la dépendance en les paramètres et en la méthode :

| $a$  | $b$   | $c$  | $d$   | $dt$  | $p_0$ | $r_0$ |
|------|-------|------|-------|-------|-------|-------|
| 2, 2 | 0, 01 | 0, 5 | 0, 05 | 0, 01 | 100   | 100   |

Evidemment, il va falloir visualiser ces résultats et se convaincre de la périodicité du phénomène.

<sup>13</sup>vers 1920, Volterra devant par la suite s'exiler, refusant de collaborer au régime de Mussolini

Il va donc falloir tracer les fonctions  $t \mapsto p_t$  et  $r \mapsto r_t$ .

Je n'ai aucune envie d'utiliser des modules de **numpy** et **scipy** (de même que je préfère vous faire créer par vous même les procédures de multiplication matricielle, et aussi parce que je n'ai pas ces modules chez moi pour les tester).

On va utiliser la tortue qui a un avantage : on y voit les courbes se construire progressivement car les tracés sont alors volontairement ralentis par l'ordinateur.

Il faudra donc importer le module **turtle**, et en utiliser les fonctions **goto(., .)**, **color(., .)**, plus quelques variantes que je vous laisse essayer.

On peut importer tout du module **turtle** par **from turtle import \***;

On pourra aussi préparer le terrain avec **import turtle as tort ; (as = en tant que)**

Que fait cette ligne ? Elle prévient le compilateur Python que vous allez utiliser des fonctions du module **turtle** et que vous allez l'appeler **tort** pour vous simplifier la vie et la frappe : **tort.goto(., .)**, **tort.color(., .)** et autres. Quel est l'avantage : cela évite qu'il n'y ait des problèmes de concurrence si une de vos procédures porte déjà un nom qui existe dans le module **turtle**, cela évite de charger dès le début tout le module, et cela vous permet de voir au premier coup d'oeil quelles procédures font appel au module dans votre programme.

A chaque nouveau calcul de  $p$  et  $q$  dans la boucle itérative, vous ajouterez donc un **goto(p, r)** (ou **tort.goto(p, r)**), qui est bien plus visuel qu'un **print(p, r)**.

Cette instruction vous permet de tracer un graphique sur lequel vous voyez simultanément le nombre de poissons et de requins. Le temps est alors juste la variable dynamique du temps passé à dessiner.

---

#### **Variantes :**

- on peut faire varier les coefficients  $a, b, c$  et  $d$  au fil du temps
- on peut augmenter ou diminuer la valeur de  $dt$  pour voir si elle a une grosse influence
- on peut "perturber" la méthode en effectuant les deux instructions  $p = p + (a \cdot p - b \cdot p \cdot r) \cdot dt$  et  $r = r + (-c \cdot r + d \cdot p \cdot q) \cdot dt$  de manière consécutives et non successives
- on peut, comme le fit Volterra faire intervenir trois ou quatre populations en interactions
- on peut tracer des graphes  $(n, p_{n \cdot dt})$  et  $(n, r_{n \cdot dt})$ .

---

On peut aussi modifier le système en créant une concurrence parmi les prédateurs :  $r'_t = -c \cdot r_t + d \cdot p_t \cdot r_t - e \cdot (r_t)^2$ .

On peut introduire une troisième espèce, qui serait prédateur des prédateurs par exemple (Volterra a étudié les systèmes avec  $n$  espèces en interaction et est arrivé à une conclusion dépendant du nombre d'espèces et même simplement de sa parité).

On peut envisager le cas d'une seule espèce qui est son propre prédateur :  $h'_t = c \cdot h_t - d \cdot (h_t)^2$  (essayez d'ailleurs de résoudre cette équation, c'est faisable avec des fonctions simples par "séparation de variables et décomposition en éléments simples").

---

L'idée de résolution approchée d'équations différentielles "pas à pas" par cette méthode avec des  $dt$  (dite "méthode d'Euler") peut être étendue à d'autres problèmes différentiels réputés insolubles par les méthodes usuelles.

Pourquoi ne pas traiter l'équation du pendule  $\theta'' = -\sin(\theta)$ .

A chaque étape, on connaît  $\theta_n$  et  $\theta'_n$ . On calcule alors  $\theta''_n = -\sin(\theta_n)$ , on augmente :  $\theta'_{n+1} = \theta'_n + \theta''_n \cdot dt$  et aussi  $\theta_{n+1} = \theta_n + \theta'_{n+1} \cdot dt$ .

On avance ainsi par segments de temps "petits", et on trace la solution approchée. On la compare alors avec le graphe de la solution du faux pendule  $\theta'' = -\theta$  (de solutions dans  $Vect(\cos, \sin)$ ).

---

Si vous avez réussi ce problème du pendule, attaquez vous au problème des trois corps soumis

mutuellement à l'attraction newtonnienne.

|                     |                             |      |
|---------------------|-----------------------------|------|
| M.P.S.I.2 2014-2015 | <b>PRODUIT DES CHIFFRES</b> | Py09 |
| Charlemagne         |                             |      |

Le problème est simple :

on prend un entier  $n$  (la "graine"), on l'écrit en base 10 ( $n = \sum_{k=0}^d c_k \cdot 10^k$ ), on effectue le produit de ses chiffres  $P = c_0 \times c_1 \dots c_d$ , et on recommence avec ce nouvel entier.

On s'arrête en un certain nombre d'étapes (la "longueur du vol"), quand on retombe sur le même entier ().

Exemple : on part de 273, on trouve  $2 \times 7 \times 3 = 42$ , on recommence :  $4 \times 2 = 8$ , on a fini, avec longueur de vol égale à 3.

Nouvel exemple : on part de 4327, on trouve  $4 \times 3 \times 2 \times 7 = 168$ , on recommence :  $1 \times 6 \times 8 = 48$ , on recommence :  $4 \times 8 = 32$  on termine :  $3 \times 2 = 6$ .

Votre travail de programmation : quand on vous donne un entier  $n$ , déterminer la liste de ses chiffres, puis le produit de ses chiffres  $p(n)$ .

Seconde étape : mettre ce programme en boucle jusqu'à ce que l'on ait  $p(n) = n$ . Associer alors à la graine sa longueur de vol et l'élément final.

Travail supplémentaire : trouver des entiers dont le temps de vol est le plus long possible.

|                     |                            |      |
|---------------------|----------------------------|------|
| M.P.S.I.2 2014-2015 | <b>DEFIS ARITHMETIQUES</b> | Py10 |
| Charlemagne         |                            |      |

|             |                         |      |
|-------------|-------------------------|------|
| MPSI 2/2014 | <b>Les dix entiers.</b> | Py10 |
|-------------|-------------------------|------|

**Il existe dix entiers  $n$  tels que  $n! + 1$  soit premier. Trouvez les.**

Vous devrez créer une fonction qui calcule  $n!$  à partir de  $n$  soit par une simple boucle, soit par méthode récursive.

Ensuite, pour tester si un nombre impair est premier, un test assez simple est :

def rithmayonnaise(N) :

....k = 3

....while N%k != 0 and k\*k <= N :

.....k += 2

....return(k\*k > N)

Pouvez vous expliquer pourquoi ?

|             |                                   |      |
|-------------|-----------------------------------|------|
| MPSI 2/2014 | <b>Les entiers de Moret-Blanc</b> | Py10 |
|-------------|-----------------------------------|------|

**Les nombres de Moret-Blanc sont ceux qui sont égaux à la somme des chiffres de leur cube.**

Tenez, essayez :  $8^3 = 512$  et  $8 = 5 + 1 + 2$ .

Ici, ça va être assez facile. On fait une boucle sur  $n$ . On calcule le cube. On le convertit en chaîne par `val()`. On coupe ensuite cette chaîne de caractères en caractères (*retrouvez la fonction*). On somme alors les chiffres de cette liste (*reconvertis en nombres par `int`*).

Ou alors créez une somme par des `s=s+k%10` et `k=k//10`.

MPSI 2/2014

**Les nombres de Victor Thebault**

Py10

Combien y-a-t-il d'anagrammes de 123456789 ? Facile, il y en a 9!.

Mais combien sont premiers ? Facile. Aucun, puisque la somme de leurs chiffres est un multiple de 9.

Ensuite, combien sont pairs ? Facile. Ceux qui se terminent par un chiffre pair.

Mais combien sont des carrés parfaits ?

C'est la question posée au début du vingtième siècle par Victor Thébault dans une revue mathématique.

A vous de créer la liste des 9! permutations et de tester à chaque fois si vous avez un carré parfait. Mais vous pouvez optimiser.

MPSI 2/2014

**La remarque de Edouard Lucas.**

Py10

Pour tout entier naturel  $n$  on note  $L_n$  le nombre de coefficients binomiaux  $\binom{n}{k}$  qui sont impairs.

Il faut comprendre le lien entre  $L_n$  et la décomposition de  $n$  en base 2.

Vous aurez donc besoin de calculer pour chaque valeur de  $n$  les binomiaux sur la ligne d'indice  $n$ .

Attention, n'utilisez pas la formule  $\frac{n!}{k!(n-k)!}$ . Pensez à  $\binom{n}{k} = \frac{n-k+1}{k} \binom{n}{k-1}$  pour avancer pas à pas. Le test de parité c'est par `bin%2==0`.

Ensuite, approfondissez en cherchant les congruences modulo 3 ou modulo un autre nombre premier  $p$ .

MPSI 2/2014

**La suite de Lenstra.**

Py10

$$1 + \sum_{k=0}^n (\lambda_k)^2$$

Lenstra a défini la suite  $\lambda$  par  $\lambda_0 = 1$  et  $\lambda_{n+1} = \frac{1 + \sum_{k=0}^n (\lambda_k)^2}{n+1}$ .

Il prétend que tous les termes de cette suite sont entiers. Est ce vrai ?

Attention, les divisions de Python se font en calcul approché avec la seule barre /. Vous devrez

donc tester de proche en proche si  $\left(1 + \sum_{k=0}^{n-1} (\lambda_k)^2\right) \% n$  est nul, puis définir  $\left(1 + \sum_{k=0}^{n-1} (\lambda_k)^2\right) // n$  si tout va bien.

MPSI 2/2014

**La fonction de Wilson.**

Py10

On définit :  $n \mapsto \left[ \frac{(k-1)! + 1}{k} - \left[ \frac{(k-1)!}{k} \right] \right]$ .

Que fait cette fonction ?

MPSI 2/2014

**Autant de diviseurs.**

Py10

Trouvez les entiers  $n$  tels que  $n, n+1, n+2, n+3$  et  $n+4$  ont autant de diviseurs.

M.P.S.I.2 2014

**KANTOROVITCH**

2015 Charlemagne

p Py11 q

Un classique de l'optimisation et de la recherche opérationnelle (*domaine de l'ingénierie*), posé en 1940 par LÉONID KANTOROVITCH.

Tout se passe sur un terrain/carte que l'on va supposer carré. Sur ce terrain, des mines et des usines. Le même nombre **Nb** de chaque. Réparties aléatoirement.

Votre mission : créer des routes. Chaque route devra relier (*en ligne droite*) une mine et une usine (*donc Nb routes*). L'objectif : **minimiser la somme des longueurs**<sup>14</sup>.

*On retrouve ce problème dans le domaine de la conception de circuits intégrés en électronique, dans l'optimisation de réseaux en Intranet ou sur Internet.*

Il faudra donc tirer au hasard la liste des coordonnées des mines, la liste des coordonnées des usines, visualiser le terrain avec ses mines et usines.

Ensuite, il faudra chercher le graphe le plus court, qui correspond à une permutation  $\sigma$  de la liste  $[0, 1, \dots, Nb-1]$  (*qui à la  $k^{ième}$  mine associe la  $\sigma(k)^{ième}$  usine*).

Une solution serait d'explorer toutes les permutations possibles, de calculer pour chacune la dis-

tance totale  $\sum_{k=0}^{Nb-1} dist(Mine_k, usine_{\sigma(k)})$ , et de mémoriser celle pour laquelle cette distance est la plus courte.

C'est la démarche naturelle. Mais elle nécessite d'explorer  $Nb!$  listes. Et si vous avez 20 usines, vous devez tester un peu plus de  $10^{13}$  listes<sup>15</sup>.

Il faudra donc se contenter dans un premier temps d'un algorithme qui teste quelques milliers de listes au hasard et cherche le minimum parmi ces listes.

*Il existe ensuite des algorithmes dont l'ordre de grandeur est  $Nb^3$  ce qui est un énorme progrès. Mais pour Nb de l'ordre de quelques milliers, cela redevient monstrueux.*

MPSI 2/2014

## Programme

Py11

Les premières lignes de votre programme :

- l'importation des modules (**Tkinter** pour les dessins, **random** pour les tirages "aléatoires" et **math** pour la fonction **sqrt**).

- la définition des constantes (*vous oubliez souvent cette étape, et vous avez des valeurs numériques qui trainent tout au long du programme que vous devez changer à dix sept ou dix huit endroits quand vous voulez perfectionner le programme*) :

**Nb, taille, echelle, NbEssais = 12, 40, 10, 300** par exemple

(**taille** : c'est la taille du terrain, **echelle**, c'est l'échelle à laquelle vous représenterez le terrain, car une unité de longueur sur un canvas de Tkinter fait un dixième de millimètre).

Ensuite, il faut tirer les coordonnées des usines et des mines au hasard. Vous commencez par définir deux matrices

```
mines = [[0]*2 for k in range(Nb)]
```

pour pouvoir nommer **mines[k][0]** et **mines[k][1]** l'abscisses et l'ordonnée de la  $k^{ième}$  mine.

Avec une boucle for, vous tirez pour chaque  $k$  de 0 à  $Nb-1$  deux nombres en **randrange(taille)** que vous placez dans **mines[k][0]** et **mines[k][1]**.

Vous faites de même pour les usines.

Question mathématique : quelle est la probabilité qu'il y ait deux des  $2.Nb$  mines ou usines au même point de la matrice de dimensions **taille** sur **taille** ?

Question informatique : comment faire pour qu'il n'y ait pas deux usines ou mines au même endroit ?

Pensez pour cela à créer **terrain=[[0]\*taille for k in taille]** et à mettre des **while terrain[i][j] != 0 : i, j = randrange(taille), randrange(taille)**.

<sup>14</sup>avec des variantes : minimiser la plus grande longueur, minimiser la somme des carrés des longueurs

<sup>15</sup>avec dix mille listes par seconde, vous aurez la réponse dans un peu plus de quatre ans, et je vous l'enverrai à l'école où vous serez élève ingénieur

*Je vous laisse perfectionner cette idée si vous voulez effectivement éviter les points doubles.*

Cela dit, lors de la première séance, E.R. a proposé une assez jolie méthode. On crée la liste **L** des *taille*<sup>2</sup> couples **[i, j]** possibles<sup>16</sup>, et on tire au hasard dans cette liste des couples qu'on place dans la liste des usines et des mines. Mais chaque couple tiré est effacé de la liste. C'est ce que fait la méthode **pop**.

*Pour vérifier : créez une liste **L** de huit termes et faites **print (L.pop(5))** suivi de **print (L)**.*

On va donc ici mettre en boucle (**Nb** fois) des **site=L.pop(randrange(...))** et **mines.append(site)**. Le **randrange(...)** va permettre de tirer au hasard dans la liste des sites encore disponibles et le **pop** va permettre d'effacer les termes de la liste **L**. Que fait il mettre dans le **randrange**? L'élève qui répond **taille\*taille** a perdu, car la liste **L** réduit au fur et à mesure. Pensez à la fonction **len**.

---

Passons au graphique, avec le module **Tkinter** (ou **tkinter** selon la version de Python que vous utilisez).

On crée une fenêtre **fenetre** et un canevas **can** dans ladite fenetre :

```
fenetre = Tk()
```

```
can = Canvas(fenetre, width = ..., height = ..., bg = '...')
```

Je vous laisse trouver les formules fonctions de **taille**, **echelle** à mettre dans les trois petits points qui suivent **width** et **height**. Pour ce qui est de la couleur de fond **bg**, je vous laisse choisir.

N'oubliez pas **can.pack()** qui permet d'inclure effectivement le Canvas à la fenêtre. N'oubliez pas non plus **fenetre.mainloop()** qui lance l'exécution de **fenetre**.

*A toutes fins utiles, je vous conseille aussi*

```
Button(text = 'Fin', command = fenetre.destroy).pack()
```

*qui va créer un widget sur lequel cliquer pour fermer la fenetre.*

*Sur le même mode, vous pouvez créer*

```
Button(text = 'Affichage', command = affiche).pack()
```

*si vous avez pensé à créer une procédure **affiche()** quelquepart dans le programme, de même pour tout bouton pour faciliter la vie de l'utilisateur.*

Ensuite, vous tracez sur **can** des cercles pour les mines et les usines avec deux boucles **for** et des **can.create\_rectangle(...)** ou **can.create\_oval(...)**. Cette fonction dessine un rectangle (ou carré) ou ovale (ou cercle) sur le canevas. Les quatre quantités à transmettre sont les deux coordonnées du coin supérieur gauche, puis les deux coordonnées du point inférieur droit. Vous ajoutez aussi **fill='...'** pour définir la couleur de remplissage.

Par exemple **can.create\_oval(100,200,140,240,fill='Blue')** crée un cercle dans un carré centré sur (120, 220) tangent aux côtés du carré, donc de rayon 20. De plus, ce cercle sera alors un disque bleu.

---

On passe enfin à la recherche de la longueur d'un graphe une fois choisie une liste.

Pour tirer une liste au hasard, permutation de **[0, 1, ..., Nb - 1]**, je vous propose :

```
Liste_fixe, Liste_alea = range(Nb), [] #on crée une liste ordonnée Liste_fixe et une liste vide
```

```
for k in range(Nb) : #on crée une boucle
```

```
....i = randrange(Nb-k) #on tire un indice dans la liste de ce qu'il reste
```

```
....Liste_alea.append(Liste_fixe.pop(i))
```

On (re)tire (par **pop**) l'élément d'indice **i** dans **Liste\_fixe** (qui réduit donc peu à peu), et on le colle (par **append**) au bout de **Liste\_alea**.

A la fin des boucles, on a vidé la liste **Liste\_fixe** et on a tout placé (dans le désordre dans **Liste\_alea**).

---

<sup>16</sup>**L=[[i,j] for i in range(...) for j in range(...)]**

Une fois établie une telle liste, calculez la somme des longueurs, par une boucle **for** (n'oubliez pas d'initialiser **distance** à 0 avec la boucle).

Vous pouvez en profiter pour représenter le réseau de routes que vous avez créé avec des

**can.create\_line(mines[k][0], mines[k][1], usines[L[k]][0], usines[L[k]][1])**

Ne recopiez pas cette instruction, comprenez la avant tout.

*Petit problème : à chaque fois que vous choisissez une permutation, si vous tracez ces lignes, vous allez finir par transformer votre canevas en toile d'araignée. Pensez à utiliser **can.delete(ALL)** pour tout effacer et recommencer un nouveau tracer.*

Il faut ensuite mettre ces essais en boucle, en mémorisant celui qui donne la plus courte distance.

**for k in range(NbEssais) :** #vous ferez NbEssais essais et non pas Nb!

.... votre tirage de liste **L**, votre calcul de distance

.... **if distance < min\_distance :** #vous avez détecté une solution meilleure que les précédentes

..... **Liste\_optimale = L** # vous mémorisez la bonne liste

..... **min\_distance = distance** #et vous actualisez la distance

Evidemment, ce petit script n'est pas convenable, il faut avoir initialisé **min\_distance** (mais à quelle valeur ?) et **Liste\_optimale** (ici, la valeur est elle importante ?).



Objectif : modéliser la formation d'un récif de coraux.

Mais de manière équivalente, on peut penser à la formation de structures métalliques sur des électrodes dans une solution chimique qui pourrait être une pile électrique.

Le domaine sera ici plan pour faciliter le travail et la visualisation. Il s'agira d'un rectangle de taille **tx** sur **ty** (largeur **tx** et profondeur **ty**). Pour simplifier encore le travail, on discrétisera ce domaine en **tx**×**ty** carrés.

Dans ce domaine vivent (et meurent) des cellules qui se déplacent de manière un peu cahotique et aléatoire (ce sont des coraux dans l'eau ou des ions dans la solution).

Quand ils touchent le fond, ils meurent et s'immobilisent.

Si la règle n'est que celle ci, le fond va se couvrir de cellules mortes, et ce ne sera pas très intéressant. D'autant que quand le fond sera couvert, il ne se passera plus rien.

En fait, une cellule meurt aussi si elle touche une cellule déjà morte. Ceci va avoir pour effet qu'il va commencer à se former des arbres ou stalacmites, et plus un arbre de cellules mortes sera haut, plus il pourra agripper des cellules sur ses branches.

C'est ainsi qu'on verra se former des structures qui évoqueront pour certains une forêt, pour d'autres la corrosion sur des anodes, pour d'autres des récifs, des cristaux et pour Yacine des choses que moi même je n'ose pas imaginer.

On notera que l'expérience in vivo est parfois réalisée en laboratoire avec de solutions cristallines et donne lieu alors à de jolies photographies de "jardins" visibles sur Internet.



Il faudra au début importer **Tkinter** pour les dessins et **random** pour les déplacements aléatoires.

Il faudra définir quelques constantes (faciles d'accès et modifiables à loisirs) :

**tx** et **ty** : les dimensions du terrain

**ech** : l'échelle qui fera qu'une case du terrain sera représentée sur le Canvas par un certangle de

taille ech sur ech

**Nb** : le nombre de cellules initialement présentes dans la solution

**NbEtapes** : le nombre d'étapes pendant lequel vous allez regarder tomber la neige, à moins que vous n'utilisiez une boucle **while** portant sur le nombre d'individus encore vivants.

---

Il faudra initialiser la liste des cellules vivantes.

Il vous suffira de mettre en boucle **Nb** fois une instruction du type

```
x, y = randrange(tx), randrange(ty)
```

```
ListeVivants.append([x, y])
```

Il n'est ici pas gênant dans un premier temps qu'il y ait plusieurs cellules au même endroit.

Par la suite, on pourra créer une version plus intelligente où des cellules qui se recontrent fusionnent en une seule cellule plus lourde.

On pourra aussi remplacer **randrange(y)** par **randrange(y-20)** ou **randrange(y//2)** pour qu'il n'y ait pas tout de suite des cellules au fond de l'eau.

Il faudra aussi initialiser le fait que le fond soit létal, en le considérant comme fait de cellules mortes.

Soit que vous créez une liste de cellules mortes :

```
ListeMortes=[x, ty for x in range(tx)]
```

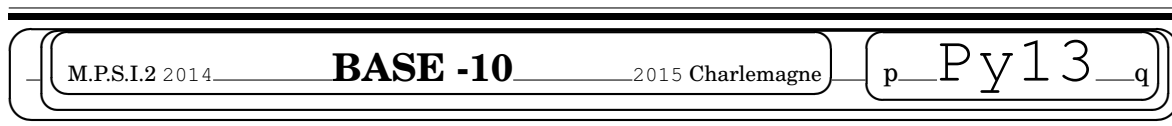
soit que vous remplissez à la valeur 1 toute la ligne **ty** d'un tableau de taille **tx** sur **ty**.

---

Ensuite, il faudra regarder les cellules se déplacer, tour après tour, une après l'autre.

A chaque tour, vous prenez une par une les cellules de la liste **ListeVivants**. Vous tirez au hasard un nombre *alea* entre 0 et 3 avec **randrange**. Suivant la valeur de ce tirage, vous incrémentez ou diminuez **ListeVivants[k][0]** ou **ListeVivants[k][1]** d'une unité (*structure en **if alea==...** **elif ... elif...** ou usage de deux tableaux prédéfinis **x=x+dx[k]** et **y=y+dy[k]***).

L'idée du tirage d'un nombre entre 0 et 3 n'est toutefois pas judicieuse. Elle laisse divager les cellules dans toutes les directions sans en favoriser alors qu'on doit tenir compte d'un tropisme/attraction des cellules en direction du fond (*gravitation, attraction électrique...*). Il faut donc tirer un nombre entre 0 et 5 par exemple. Pour les *alea* de 0 à 2 : haut, droite ou gauche et pour *alea* plus grand que 2 : bas.



On rappelle le principe de l'écriture en base  $b$  :

le nombre à  $n + 1$  chiffres s'écrivant  $\overline{a_n \dots a_2 a_1 a_0}$  est  $\sum_{k=0}^n a_k \cdot b^k$ .

On connaît l'écriture binaire (*base 2*) et décimal (*base 10*).

Mais on peut également définir la base  $-10$ . C'est ainsi que par exemple le nombre s'écrivant  $\overline{3425}$  n'est autre que  $3 \cdot (-10)^3 + 4 \cdot (-10)^2 + 2 \cdot (-10) + 5$ . On notera que cet entier est négatif. C'est l'un des avantages de la base  $-10$ . On peut représenter tous les entiers de  $\mathbb{Z}$  avec des écritures "positives" (*la parité du nombre de chiffres a son importance*).

---

Premier programme : on vous donne l'écriture d'un entier en base  $-10$  et il faut l'interpréter en décimal. Il vous faudra donc découper en chiffres et calculer la somme des  $c_k \cdot (-10)^k$ .

La meilleure solution est la fonction récursive. L'idée est que l'on doit avoir  $\overline{abcde} = e - 10 \cdot \overline{abcd}$  par exemple (*vérifiez sur un exemple ou même si possible en toute généralité*).

Quand vous définirez votre fonction par **def changebase(n)** : vous aurez sans doutes un

`return (changebase(n//...) * ... + (...))`, en rappelant que le quotient et le reste de la division euclidienne de  $a$  par  $b$  sont  $a//b$  et  $a\%b$ . Il faudra penser quand même par commencer par un test qui fait sortir de la boucle récursive quand le nombre n'a qu'un chiffre.

---

Pour la saisie, pensez à la fonction `input`.

Elle retourne la chaîne de caractères tapée au clavier et autorise l'affichage d'un message d'invitation : `a=input("entrez la valeur de a")`.

Il y a quand même un problème. L'entrée d'un `input` est une chaîne de caractères. Donc, même si l'utilisateur tape un nombre entier au clavier, le résultat passé en variable est une chaîne. Vérifiez avec un `input` suivi d'un `print`. Il vous faudra donc convertir cette chaîne en nombre entier par un `int(input(...))`.

---

Il vous faudra ensuite créer la fonction inverse. On lui donne un entier relatif (*en écriture décimale*), et on doit en donner la liste des chiffres en base  $-10$ .

Là, il faut réfléchir un peu plus. Mais une démarche assez similaire en  $n//10$  et  $n\%10$  est encore possible, de même qu'une démarche récursive. Mais entraînez vous d'abord.

---

Prolongements :

- peut on créer une fonction qui compare deux nombres écrits en base  $-10$  sans les calculer explicitement.
- un palindrome en base  $-10$  peut il être un carré parfait
- de tous les anagrammes d'un nombre donné en écriture de base  $-10$ , lequel est le plus grand, lequel est le plus petit.

---

Après, on peut aussi réfléchir à l'écriture en base factorielle :

$n = a_1.1! + a_2.2! + a_3.3! + \dots$  avec pour chaque indice  $k : a_k \leq k$  (*pour ne pas faire passer  $a_k.k!$  à  $du (k+1)!$* ).

Entraînez vous sur des exemples pour comprendre la démarche, puis pondez les deux programmes de conversion factoriel vers décimal et décimal vers factorielle.

---

|             |                 |      |
|-------------|-----------------|------|
| MPSI 2/2014 | <b>Premiers</b> | Py13 |
|-------------|-----------------|------|

---

Commençons par une remarque. Regardez l'entier 131. Il est premier. Et plutôt deux fois qu'une... Qu'est ce que je veux dire par là ? Que l'entier s'écrivant 131 en base 10 est premier, mais aussi que l'entier s'écrivant 131 en base  $-10$  l'est aussi (*c'est celui que vous appelez 71*).

Le but du jeu : trouver des écritures  $\overline{abc\dots}$  telles que l'entier ainsi décrit en base 10 soit premier, de même que l'entier ainsi décrit en base  $-10$ .

Pour cet exercice, je vous fais cadeau d'un script classique :

```
LP, MaxP = [2], 1000
```

```
i = 1
```

```
while i < MaxP :
```

```
....i += 2
```

```
....tp = 1
```

```
....for k in LP :
```

```
.....tp *= (i%k)
```

```
....if tp != 0 :
```

```
.....LP += [i]
```

```
print(LP)
```

Que fait ce script ? Testez le. Modifiez le aussi pour qu'il engendre une liste de longueur `NbP`

donnée à l'avance.

Parcourez ensuite cette liste **LP**, prenez en les éléments (*dont vous savez qu'ils ont la propriété voulue*), transcrivez les en base  $-10$ , et regardez si l'entier obtenu (*gare à la valeur absolue*) est dans **LP**. Si tel est le cas, collez le dans la liste **SuperPrem** des éléments que vous cherchez.

Variante : trouvez des écritures  $\overline{abcd\dots}$  telles que l'entier ainsi décrit soit premier dans beaucoup de bases.

Exemple : en base 4,  $\overline{131}$  décrit l'entier 29 qui est premier, en base 5 c'est 41 premier aussi, il l'est aussi en base 7, 8, 9, 12, 14, 15, 18, 19...152,153, 154, ...202,...



**Objectif : un peu de dessin sur ordinateur.**

On va représenter la chute d'un dé (*ou d'un cube*), en perspective.

Il faudra d'abord modéliser la loi physique de la chute de son centre de gravité (*chute libre ici évidemment sans frottements*). Le mouvement horizontal sera donc uniforme (*vitesse initiale et donc vitesse à définir par saisie*). Le mouvement vertical sera uniformément accéléré. On notera **xc**, **yc** et **zc** ces trois coordonnées qui varieront au fil du temps.

Exprimer **xc**, **yc** et **zc** (*lois horaires et non relation entre ces variables*).

Vérifier la cohérence de votre modèle en traçant un cercle qui chute suivant cette loi. On va donc utiliser le module Tkinter, et faire appel au module time pour que le mouvement se fasse étape par étape.

```
from Tkinter import *
```

```
fen = Tk() on crée une fenêtre
```

```
can = Canvas(fen, height=..., depth=..., background=...)
```

```
can.pack()
```

 on place un canevas dans cette fenêtre (il faut le créer, avec ses dimensions et sa couleur de fond, puis il faut l'incorporer à la fenêtre avec la méthode **pack** ou **grid**).

Ensuite, on trace des lignes, carrés, cercles sur le canevas avec **can.create\_line(...)**, **can.create\_rectangle()** ou **can.create\_oval()** dans lequel vous indiquez des coordonnées de points (extrémités du segment, coins du rectangle...), la couleur, éventuellement la largeur du trait.

Pour créer l'animation, on met en boucle l'instruction :

on affiche un dessin, on attend un peu (**sleep**), on efface tout (**can.delete(ALL)**) ou juste le dessin si il a un nom, et on recommence, avec un nouveau dessin

Attention, n'oubliez pas que le point de coordonnées **0,0** sur un canevas est le point en haut à gauche. Votre dé risque de remonter si vous n'y prenez pas garde.

Si votre carré ou cercle se déplace bien, il faut passer au cas d'un cube, en perspective.

Il faut définir la position des huit sommets. Il faut donc définir trois vecteurs de norme  $20^{17}$  orthogonaux entre eux  $\vec{u}$ ,  $\vec{v}$  et  $\vec{w}$  (*fixes pour l'instant, puis ensuite ou variables au fil du temps suivant une loi à préciser*) et créer les huit sommets  $A_1 = C + \vec{u} + \vec{v} + \vec{w}$ ,  $A_2 = C + \vec{u} + \vec{v} - \vec{w}$ ,  $A_3 = C + \vec{u} - \vec{v} + \vec{w}$ ,  $A_4 = C + \vec{u} - \vec{v} - \vec{w}$ ,  $A_5 = C - \vec{u} + \vec{v} + \vec{w}$ , et ainsi de suite (*justifiez que ce sont les sommets du cube, et précisez quel est la longueur de son arête*).

<sup>17</sup>vingt pixel à l'écran, ça fait un centimètre environ

Ces coordonnées seront stockées dans un tableau à huit colonnes et trois lignes : **sommets** ou dans huit points avec des noms si vous avez des difficultés à faire de la presque-vraie programmation.

Il faudra ensuite projeter ces points de  $\mathbb{R}^3$  sur le plan de l'écran de visualisation qui lui est de dimension 2. On fera pour cela usage de matrices à trois colonnes et deux lignes (*de  $\mathbb{R}^3$  dans  $\mathbb{R}^2$* ). Il existe deux types classiques de projections : cavalière et isométrique (*suivant l'angle des lignes de fuite*) :  $\begin{pmatrix} 1 & \sqrt{3}/2 & 0 \\ 0 & 1/2 & 1 \end{pmatrix}$  ou  $\begin{pmatrix} \sqrt{2}/2 & -\sqrt{2}/2 & 0 \\ \sqrt{2}/2 & \sqrt{2}/2 & 1 \end{pmatrix}$  (*expliquez ces valeurs*).

Bien sûr, il suffira de créer une fonction telle que

```
def ourmirouj(point) :
....X = R*point[0] - R*point[1]
....Y = R*point[0] + R*point[1] + point[3]
....return(X,Y)
```

après avoir défini dès le début du programme  $R=\sqrt{2}/2$  (*il n'est en effet pas très judicieux de lui faire recalculer cette valeur à chaque fois*).

En utilisant ensuite cette fonction, on crée un tableau à huit colonnes et deux lignes : **projec\_sommets**.

Maintenant que ces points du plan sont définis, on trace le cube “en fil de fer”.

Pour tracer un segment, on utilise **can.create\_line(...)** dans lequel on précise les coordonnées des extrémités (ici **projec\_sommets[i][0]**, **projec\_sommets[i][1]**, **projec\_sommets[j][0]**, **projec\_sommets[j][1]** si il s'agit du segment reliant les sommets **i** et **j**) et éventuellement la couleur du tracer.

Au fait, il y aura combien de segments à tracer ? faudra-t-il créer une fonction

```
def ransuxai(i, j) :
....can.create_line(...)
```

pour ne pas taper douze fois la même instruction répétitive ?

Pour que l'image bouge ? On la dessine, on remet le canevas à jour (**can.update()**), puis on l'efface (**can.delete(ALL)**) et on recommence. Vous devrez encore faire appel au module **time** et à sa fonction **sleep**.

MPSI 2/2014

## Pour aller plus loin

Py14

Maintenant que vous avez un cube qui tombe en chute libre, vous pouvez le faire tourner.

A chaque étape, on modifie  $\vec{u}$ ,  $\vec{v}$  et  $\vec{w}$  par une petite rotation. On rappelle que dans  $\mathbb{R}^3$  les matri-

ces de rotation sont de la forme  $\begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}$  ou  $\begin{pmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{pmatrix}$  ou  $\begin{pmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c \end{pmatrix}$  ou produit

de matrices de ce type. Comme vous allez utiliser une telle matrice à chaque intervalle de temps, ne prenez pas un angle trop grand (quel angle ? mais vous connaissez quand même nos notations  $c$  et  $s$ !).

Si votre dé cubique tombe bien, pourquoi ne pas faire tomber un autre objet, comme un tétraèdre, un icosaèdre, un dodécaèdre, ou plus simplement une chaise ?

Si vous avez créé une classe cube, pourquoi ne pas faire tomber plusieurs dés en même temps ?

Et si on s'intéressait à la vision en relief ?

On va tracer deux copies du cube qui tombe, de deux couleurs complémentaires (*rouge et vert*). En regardant l'écran avec des lunettes ayant un verre rouge à droite et vert à gauche je sais, les

rouges et les verts sont à gauche tous les deux, votre oeil droit ne verra que le cube vert (*en noir*) et votre oeil gauche ne verra que le cube rouge (*en noir*).

Ensuite, votre cerveau cherchera à faire coïncider les deux images et le décalage qu'il y a entre elles se traduira par une impression de relief (*visions stéréographique, utilisée depuis le début du vingtième siècle en cartographie, photographie, imagerie...*).

Si de plus le décalage entre les points associés dépend de leur éloignement y par un léger facteur perturbant, votre dé sera en relief.

MPSI 2/2014

Py14

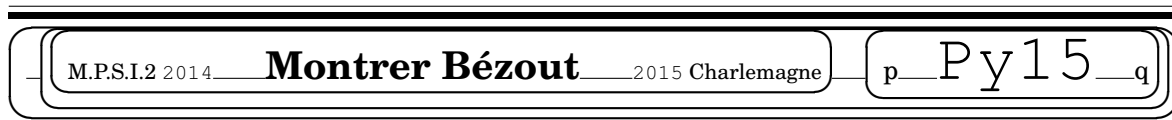
```
from Tkinter import *
from time import sleep
from math import sqrt, sin, cos, pi
from random import randrange
R = sqrt(2)/2
co = cos(pi/80)
si = sin(pi/80)
u = [40,0,0]
v = [0,40,0]
w = [0,0,40]
xc, yc, zc = 100, 100, 100
def proj(point) :
    ...X = R*point[0]-R*point[1]+200
    ...Y = R*point[0]+R*point[1]+point[2]
    ...return ([X,Y])
def cree_point(s1,s2,s3) :
    ...x = xc+s1*u[0]+s2*v[0]+s3*w[0]
    ...y = yc+s1*u[1]+s2*v[1]+s3*w[1]
    ...z = zc+s1*u[2]+s2*v[2]+s3*w[2]
    ...return ([x,y,z])
def trace_ligne(i,j) :
    ...can.create_line(cube_plan[i][0],cube_plan[i][1],cube_plan[j][0],cube_plan[j][1],fill='white')

def trace_cube() :
    ...trace_ligne(0,1)
    ...trace_ligne(1,2)
    ...trace_ligne(2,3)
    ...trace_ligne(3,0)
    ...trace_ligne(4,5)
    ...trace_ligne(5,6)
    ...trace_ligne(6,7)
    ...trace_ligne(7,4)
    ...trace_ligne(0,4)
    ...trace_ligne(1,5)
    ...trace_ligne(2,6)
    ...trace_ligne(3,7)
def definit_cube() :
    ...L = [[0,0,0] for k in range(8)]
    ...L[0] = cree_point(1,1,1)
    ...L[1] = cree_point(1,1,-1)
    ...L[2] = cree_point(1,-1,-1)
    ...L[3] = cree_point(1,-1,1)
    ...L[4] = cree_point(-1,1,1)
    ...L[5] = cree_point(-1,1,-1)
    ...L[6] = cree_point(-1,-1,-1)
```

```

....L[7] = cree_point(-1,-1,1)
....return(L)
fen = Tk()
can = Canvas(fen, height = 600, width = 600, background = 'steelblue1')
can.pack()
for n in range(400) :
....can.delete(ALL)
....xc += randrange(-2,2)
....yc += randrange(-2,2)
....zc += randrange(-2,2)
....rot = randrange(3)
....if rot == 0 :
.....un = [co*u[0]-si*u[1], si*u[0]+co*u[1], u[2]]
.....vn = [co*v[0]-si*v[1], si*v[0]+co*v[1], v[2]]
.....wn = w
....if rot == 1 :
.....un = u
.....vn = [v[0], co*v[1]-si*v[2], si*v[1]+co*v[2]]
.....wn = [w[0], co*w[1]-si*w[2], si*w[1]+co*w[2]]
....if rot == 2 :
.....un = [co*u[0]-si*u[2], u[1], si*u[0]+co*u[2]]
.....vn = v
.....wn = [co*w[0]-si*w[2], w[1], si*w[0]+co*w[2]]
....u,v,w = un, vn, wn
....cube = definit_cube()
....cube_plan = [proj(cube[k]) for k in range(8)]
....trace_cube()
....sleep(0.02)
....fen.update()
#fen.mainloop()

```



On rappelle le théorème de Bézout :

si  $a_0$  et  $a_1$  sont deux entiers naturels donnés de p.g.c.d.  $d$ , alors il existe deux entiers relatifs  $\alpha$  et  $\beta$  vérifiant  $a_0.\alpha + a_1.\beta = d$ .

On détermine  $d$  par des divisions euclidiennes successives (algorithme d'Euclide).

On détermine  $\alpha$  et  $\beta$  par remontée de l'algorithme d'Euclide, pouvant se mettre sous forme matricielle.

On crée donc une liste d'entiers qui commence par  $[a_0, a_1]$  et se prolonge au fur et à mesure, tant que le reste n'est pas nul.

Dans le même temps, on crée une liste de quotients, et même des matrices carrées de taille 2 que l'on multiplie au fur et à mesure.

A chaque étape, on prend  $a[n-1]$  qu'on divise par  $a[n]$ . Le quotient est  $q[n]$  avec pour matrice  $M_n = \begin{pmatrix} 0 & 1 \\ 1 & -q[n] \end{pmatrix}$ . Le reste est alors  $a[n+1]$  directement.

Il ne reste plus qu'à mettre en boucle tant que le reste n'est pas nul.

Le dernier reste non nul  $a[...]$  est le p.g.c.d. et le produit  $M_1.M_2 \dots M_n$  (indices ?) donne la combinaison cherchée par sa dernière ligne.

Un exemple figure dans le cours.

Rappel des fonctions utiles :

$a/b$  donne le quotient de la division de  $a$  par  $b$

$a\%b$  donne le reste qu'on affecte par un `liste.append()` en fin de liste

pour accéder aux deux derniers éléments d'une liste : `liste[-1]`, `liste[-2]` ?

Vous devrez quand même aussi inventer un module qui effectue les produits matriciels en taille 2. Il fera appel à des formules comme  $A[1, 1] * B[1, 2] + A[1, 2] * B[2, 2]$  et autres.

---

Un exemple d'algorithme d'Euclide pour saisir :  $a_0 = 2013$  et  $a_1 = 153$ .

On effectue :  $2013 = 13.153 + 24$ , puis  $153 = 6.24 + 9$ , puis  $24 = 2.9 + 6$ , puis  $9 = 1.6 + 3$  et enfin  $6 = 2.3 + 0$  (reste nul).

La liste  $a$  est alors  $[2013, 153, 24, 9, 6, 3]$ .

La liste des quotients est  $[13, 6, 2, 1]$ .

Les matrices sont alors  $\begin{pmatrix} 0 & 1 \\ 1 & -13 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & -6 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$ .

Leur produit est...

---

---

MPSI 2/2014

## De nouveaux défis arithmétiques

Py15

**Vrai ou faux :** pour tout nombre premier  $p$ , il existe un entier naturel  $d$  tel que  $q, q + d, q + 2.d, q + 3.d \dots q + (q - 1).d$  soient tous premiers. C'est une conjecture. On n'en sait rien.

Trouver  $d$  pour  $p$  égal 3, 5 et 7. Pour 11,  $p$  est de l'ordre de  $1,53 \times 10^9$  (nombre trouvé en 1986). Et pour  $p$  gal 13, une suite de neuf nombres premiers suffira.

**Nombres de Keller et Clark.** Pour tout nombre premier  $a$ , trouver les nombres premiers  $p$  tels que  $a^{p-1}$  soit gal 1 modulo  $p^2$  et pas seulement modulo  $p$ .

**Le retour de Sophie Germain.** Parait que si  $p$  est un nombre premier de Sophie Germain, alors il n'existe pas d'entiers  $x, y$  et  $z$  vérifiant  $x^p + y^p = z^p$  et qui ne soient pas des multiples de  $p$ . On ne sait pas si le nombre de nombres de Sophie Germain est fini ou non.

**Chaînes de Cunningham.** Il s'agit de suites de nombres  $(p_1, p_2, \dots p_k)$  tous premiers, tels que  $p_{(i+1)}$  soit égal  $2.p_i + 1$ . Trouver une chaîne de Cunningham de longueur la plus grande possible. Chercher des chaînes de Cunningham de seconde espèce avec  $p_{i+1} = 2.p_i - 1$

---

---

M.P.S.I.2 2014

**6 sur  $\pi^2$**

2015 Charlemagne

p Py16 q

---

Un classique des probabilités et de l'arithmétique :

**la probabilité que deux entiers tirés au hasard soient premiers entre eux est  $\frac{6}{\pi^2}$  (oui, l'inverse de  $\zeta(2)$ ).**

Mais qu'appelle-t-on tirer deux entiers au hasard, sachant qu'il y a une infinité d'entiers ?

On se donne  $N$ , on tire deux entiers au hasard uniforme entre 1 et  $N$ . On teste si ils sont premiers entre eux.

Et on fait tendre  $N$  vers l'infini.

Pratiquement, on va se donner un entier  $N$  "assez grand".

On va tirer deux entiers **a** et **b** par **randrange(N)**.

On va tester si ils sont premiers entre eux par algorithme d'Euclide.

Si c'est le cas, on incrémente un compteur.

On met ensuite ce tirage/test en boucle.

A la fin, on effectue le quotient "compteur" sur "nombre de boucles" (*attention, on veut un réel, pas un quotient entier*) et on le compare à  $\pi^2/6$ .

On rappelle que pour savoir si deux entiers  $a$  et  $b$  sont premiers entre eux, un test possible est récursif :

```
def Olidsoncor(a, b) :  
    ....if b == 0 :  
    .....print(....)  
    ....else :  
    .....Olidsoncor(b, a%b)
```

Je vous laisse indiquer dans le print le test qui doit finalement porter sur le dernier reste non nul...

MPSI 2/2014\_\_\_\_\_

### Tirages aléatoires.

\_\_\_\_\_Py16

Tiens, et si on exploitait le module aléatoire **random** pour faire des tirages du loto ?

Pour initialiser une liste des entiers de 1 à 50, c'est direct avec **range(50)**. Si toutefois vous voulez qu'ils aillent de 1 à 50 et non de 0 à 49, que faites vous ?

Ensuite, pour tirer un nombre de cette liste, c'est **k= randrange(len(L))**, **print(L[k])**.

Mais si vous utilisez plusieurs fois de suite cette instruction, vous risquez de tomber plusieurs fois sur le même nombre.

Il faut donc faire des tirages sans remises.

C'est ainsi qu'après chaque tirage, il faut retirer l'élément de la liste L. Comprenez vous alors pourquoi j'ai suggéré **randrange(len(L))** au lieu de **randrange(50)**.

M.P.S.I.2 2014\_

### SUITE AUTOREFERENTE

\_2015

Charlemagne

p\_Py17\_q

L'autre jour, ma fille, persuadée de me coller avec ce petit casse-tête me propose de compléter la suite écrite un peu plus loin. Hélas pour elle, c'est une suite classique sur laquelle déjà beaucoup de personnes se sont penchées même en mathématiques et informatique, en dépit de son caractère "simpliste". Il y a même un article de Delahaye dans un Pour La Science des dix dernières années qui lui est presque intégralement consacré :

[1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ...]

Avez vous compris le truc ?

Pour voir si vous avez compris, faites comme ce que j'ai vérifié auprès de ma fille :

- est il vrai que le nombre de chiffres est toujours pair ?
- y a-t-il toujours un 1 parmi les deux premiers chiffres ?
- qu'aurait donné la liste en commençant par 17 ? se terminera -t-elle toujours par 7 ? par 17 ?
- existe-t-il une graine (=premier terme de la suite)  $g$  telle que le terme suivant soit  $gg$  (autre [1, 11, ...], avec plus de chiffres)

Allez, on passe à l'étape de programmation :

Ecrivez un script Python qui prend une chaîne  $L$  et la transforme en "sa description". En effet, les termes de la suite sont à considérer comme des chaînes et non comme des entiers, car sinon, on dépasse vite les tailles raisonnables de manipulation de ceux ci.

Vous avez besoin de savoir

- que les termes d'une chaîne sont indexés à partir de 0,
- qu'on les appelle par  $L[k]$  pour avoir le  $k^{ième}$  terme,
- que c'est par  $len(L)$  qu'on a la longueur de la chaîne,
- 

Vous devrez prendre le premier caractère de la liste, puis avancer dans la liste tant que les nouveaux caractères sont les mêmes que celui que vous venez de lire, en les comptant et en les détruisant au fur et à mesure (*avec **pop** on extrait un caractère en l'effaçant*).

Vous collez alors sur la nouvelle liste le compteur (*convertir sa valeur numérique en chaîne*) et le caractère en question.

Vous recommencez avec le caractère suivant de la liste.

J'aimerais savoir si au fil des étapes les 1 restent bien majoritaires dans l'écriture.

---



---

|                |                          |      |
|----------------|--------------------------|------|
| M.P.S.I.2 2014 | <b>TCHEBITCHEV-ERDOS</b> | 2015 |
| Charlemagne    | p Py18 q                 |      |

---



---

Le théorème de Tchebitchev affirme : entre  $n$  et  $2.n$  il y a toujours un nombre premier (en tout cas pour  $n$  plus grand que 2).

Le résultat a été découvert et démontré d'abord par Tchebitchev, avec une démonstration assez longue et peu popularisée. En revanche Erdős (l'immense Erdős) en a trouvé une démonstration élégante qui peut faire l'objet d'un grand D.S. de Sup ou d'un joli D.M. de Spé (jamais croisé a priori en sujet de concours toutefois). Cette démonstration fait un gros usage de la décomposition en produit de facteurs premiers des entiers  $\binom{2.n}{n}$ .

Pourquoi? Reprenez la définition, et constatez :  $\binom{2.n}{n} = \frac{(n+1).(n+2) \dots (2.n)}{1.2 \dots n}$ . Au numérateur, on trouve les entiers de  $n+1$  à  $2.n$ . C'est parmi eux que l'on trouvera le(s) nombre(s) premier(s) recherché(s). Regardez par exemple pour  $n$  égal à 10 :  $\binom{20}{10} = \frac{11.12.13.14.15.16.17.18.19}{1.2.3.4.5.6.7.8.9.10} = \frac{2^2.11.13.17.19}{1}$ .

Le but de ce T.D. est donc de décomposer en produit de facteurs premiers  $\binom{2.n}{n}$ . Et déjà de calculer cet entier.

Mais finalement, est-il judicieux de le calculer pour le décomposer ensuite? Ou vaut-il mieux chercher tout de suite ses facteurs premiers?

On se permettra aussi de chercher directement le premier nombre premier entre  $n$  et  $2.n$ .

---



---

|                |                 |                  |
|----------------|-----------------|------------------|
| M.P.S.I.2 2014 | <b>VIGENERE</b> | 2015 Charlemagne |
|                | p Py19 q        |                  |

---



---

Revenons sur l'OuLiPo, avec "LA CIMAISE ET LA FRACTION", célèbre fable. Si si, vous connaissez. Simplement, c'est une fable de La Fontaine qui a subi le S+7.

Quelle fable? "LA CIGALE ET LA FOURMI".

S+7? Vous prenez le texte, et à chaque nom trouvé, vous ouvrez le dictionnaire, vous retrouvez le mot en question, et vous avancez alors sept noms communs plus loin. C'est ce qui fait passer de "CIGALE" à "CIMAISE" et de "FOURMI" à "FRACTION". Vous faites de même avec les adjectifs. Vous faites de même avec les verbes. Bref, avec les substantifs (*d'où le nom de S+7*).

---

La question est maintenant : peut-on faire ce travail avec Python?

Il faudrait simplement parcourir le texte mot à mot, identifier chaque mot dans le dictionnaire

(indice  $n$ ), et remplacer par **dictionnaire**( $n+7$ ) dans le **texte**.

Seul problème : il faut disposer d'un dictionnaire déjà saisi, et c'est un peu lourd à saisir et à manier. En plus, il faut analyser syntaxiquement pour savoir ce qui est nom, adjectif, verbe, ad-verbe... et tenir compte des conjugaisons.

---

On va donc travailler lettre à lettre sur un texte court.

De fait, on va simplement effectuer des codages et décodages de messages suivant la méthode dite de Jules Cesar.

On prend donc un texte (*chaîne de caractères nommée **texte***). On l'étudie lettre à lettre : **texte**( $k$ ) pour  $k$  dans **range**(**len**(**texte**)). On rappelle que **len** est la fonction qui lit la longueur d'une chaîne de caractères. On remplace chaque lettre par celle qui est (*par exemple*) sept indices plus loin dans l'alphabet.

Evidemment, il faut travailler avec des congruences, pour les lettres de la fin de l'alphabet (*en  $L+7$ ,  $v$  est remplacé par  $c$* ).

On crée petit à petit la nouvelle chaîne par concaténation.

Par exemple, la phrase "BONJOUR COMMENT ÇA VA" devient en  $L + 7$  : "IVUQVCY JVTTLUA JH CH".

---

On va donc utiliser une boucle **for**,

on utilisera aussi les fonctions **ord** et **chr** qui convertissent un caractère en code (*entier entre deux valeurs à retrouver par vous même*) et un code en caractère,

on créera avant la boucle **for** une chaîne vide qu'on agrandira peu à peu par **chaîne=chaîne+chr(...)**

on pensera à créer la bonne fonction ou formule pour avancer dans l'alphabet en tenant compte de la congruence modulo 26,

on pensera aussi avant toute chose à transformer la chaîne en lettres minuscules grâce à la fonction **lower**( ).

Il faudra penser aussi à créer la fonction qui part de la chaîne codée et remonte au message initial. Celle ci reposera sur  $L - 7$  dans notre exemple. Vous pourrez donc créer une seule fonction **codage**(..., ...) qui prendra deux paramètres : le texte et la valeur numérique à ajouter ou soustraire.

Il faudra aussi réfléchir à la ponctuation et aux espaces. Deux possibilités :

on efface les espaces et symboles de ponctuation (*ne rien recopier dans chaîne si le nombre **ord**(...) n'est pas dans le bon intervalle*)

on travaille sur un alphabet élargi fait d'une bonne trentaine de symboles : les lettres puis espace et quelques symboles de ponctuation ; il faut alors prendre garde à la congruence.

---

Pour le codage des messages, cet algorithme est quand même un peu faible. Si on vous donne une **chaîne** codée, il vous suffit d'exécuter **codage**(**chaîne**, **-n**) pour les vingt six valeurs de l'entier  $n$  possibles pour que le texte initial soit dans la liste.

C'est pourquoi ce code dit de Jules César a été perfectionné par la suite en ce qu'on appelle le codage de Vigenere, dit aussi code à clef périodique.

On prend encore un **texte** initial. On choisit comme **clef** un mot de quelques lettres qui devra rester un secret entre ceux qui s'envoient le message. Au lieu d'appliquer le même  $L + n$  à toutes les lettres de texte, on remplace  $n$  par l'indice des lettres du mot clef, les unes après les autres de manière répétitive.

Un exemple pour saisir. Le texte initial est encore "BONJOUR COMMENT ÇA VA". Le mot clef est "LAPIN" (*pourquoi pas*). Le tableau suivant indique ce qu'on calcule :

|            |    |    |    |    |    |    |    |    |    |    |    |   |
|------------|----|----|----|----|----|----|----|----|----|----|----|---|
| texte      | b  | o  | n  | j  | o  | u  | r  | c  | o  | m  | m  | e |
| clef       | l  | a  | p  | i  | n  | l  | a  | p  | i  | n  | l  | a |
| ord(texte) | 2  | 15 | 14 | 10 | 15 | 21 | 18 | 3  | 15 | 13 | 13 | 5 |
| ord(clef)  | 12 | 1  | 16 | 9  | 14 | 12 | 1  | 16 | 9  | 14 | 12 | 1 |
| somme      | 14 | 16 | 30 | 19 | 29 | 33 | 19 | 19 | 24 | 27 | 25 | 6 |
| chaîne     | N  | P  | D  | S  | C  | G  | S  | S  | X  | A  | Y  | F |

La chaîne obtenue est difficile à décrypter si on n'a pas la clef..

En revanche votre programme sera facile à modifier pour passer de Cesar à Vigenere.

Il suffira en effet de créer non pas le caractère `chr(ord(...)+n)` (*n étant l'entier de codage passé en paramètre de la fonction*) mais `chr(ord(...)+ord(...))` avec dans la seconde parenthèse une formule comme `clef(k modulo len(clef))` (à vous d'adapter en Python).

Il va de soi que pour la lisibilité du programme, vous serez amené à couper cette formule en tranches par `otexte=ord(texte(k))`, `oclef=ord(clef(k modulo ...))` puis `chr(otexte+oclef)`.

Le programme de codage devra donc passer deux variables et en donner une à la sortie. Peut être même trois variables à l'entrée suivant que vous voulez l'utiliser en codage (`chr(otexte+oclef)`) ou décodage (`chr(otexte-oclef)`).

Si vous le souhaitez, vous pouvez prendre pour clef non pas un mot à répéter, mais une chaîne au moins aussi longue que le texte à coder, que vous aurez choisi à l'avance avec votre correspondant (*si vous connaissez tous deux par coeur la légende des siècles de Victor Hugo, pourquoi pas*).

|                |                      |      |
|----------------|----------------------|------|
| M.P.S.I.2 2014 | INTEGRALES DE WALLIS | 2015 |
| Charlemagne    | p Py20 q             |      |

Il y a des nombres que l'on croise "assez souvent" en mathématiques (*en tout cas dans les sujets de concours d'écoles d'ingénieurs*). On les croise avec les intégrales de Wallis ( $\int_0^{\pi/2} \sin^n(t).dt$ ), les dérivées en 0 de l'arcinus et le développement limité de l'arcinus en 0 ( $\sum_{k=0}^n \frac{Arcsin^{(k)}(0)}{k!} .x^k$ ), des problèmes de dénombrement.

Ces entiers  $(a_n)$  sont définis par la relation de récurrence suivante :  $a_0 = 1$  et  $a_{n+1} = \frac{2.n+1}{2.n} .a_n$

On se propose de les calculer, et de différentes façons.

|             |               |      |
|-------------|---------------|------|
| MPSI 2/2014 | Boucle "for". | Py20 |
|-------------|---------------|------|

On montre déjà :  $a_n = \prod_{k=0}^{n-1} \frac{2.k+1}{2.k+2}$  par récurrence sur  $n$ .

On calcule alors ce produit par une boucle "FOR". On initialise un produit  $p$  à la valeur 1, puis on le multiplie par  $\frac{2.k+1}{2.k+2}$  pour  $k$  allant de 0 à  $n-1$ . Par chance, avec Python, l'instruction `range(n)` crée la liste des entiers de 0 à  $n-1$  (*habitude de ce langage de compter de 0 à  $n-1$* ). Avec l'instruction "FOR **K** IN RANGE(N) :", on a exactement ce que l'on désire.

On pense quand même à demander de retourner la valeur  $p$ .

|             |                 |      |
|-------------|-----------------|------|
| MPSI 2/2014 | Boucle "while". | Py20 |
|-------------|-----------------|------|

Le principe est le même, mais on fait décrire soi-même à  $k$  les valeurs de 0 à  $n - 1$ . On initialise  $k$  à la valeur 0 et on effectue une boucle tant que  $k$  est plus petit que  $n$ . C'est justement ce qui se fait avec **"WHILE (CONDITION) : (INSTRUCTION)"**.

Ce qu'il ne faut pas oublier : initialiser  $p$  à 1 et  $k$  à 0, augmenter  $k$  de une unité à chaque exécution de la boucle (*sinon on tourne en rond*), retourner  $p$  à la fin.

MPSI 2/2014

### Méthode "factorielle".

Py20

On exprime  $a_n$  à l'aide de  $(2.n)!$  et  $n!$ . Je vous laisse le soin de trouver la formule explicite avec la clef suivante :  $a_{n+1} = \frac{(2.n+1).(2.n+2)}{4.(n+1)^2} . a_n$  (vérifiez).

Le problème est : "Python connaît-il la fonction factorielle, et si oui, sous quel nom ?"

Une réponse est : **"FACTORIAL"**. Et l'autre est "il la connaît mais il ne la charge pas directement". En effet, seuls les mathématiciens ont vraiment besoin de cette fonction, et ça ne fait qu'une partie des utilisateurs de Python. Il faut donc lui demander d'aller la chercher dans son module **"MATH"**. Pour ce faire, il vous faudra taper **"FROM MATH IMPORT FACTORIAL"** qui va aller chercher sur le disque dur dans le module math la fonction appelée factorial. Notez qu'avec **"FROM MATH IMPORT \*"**, vous lui demandez d'importer toutes les fonctions mathématiques qu'il connaît (*regardez au passage ce qu'il y a*).

Rappel : pour le carré d'un entier  $N$ , l'instruction n'est pas **N^2** mais **N\*\*2**.

MPSI 2/2014

### Méthode récursive

Py20

C'est bien entendu celle qu'on préférera en tant qu'informaticien.

On passe la variable  $n$ . Si elle est plus petite que 1 on répond 1 et sinon, on utilise la formule encadrée en début de document pour calculer  $a_n$  à l'aide de  $a_{n-1}$ . On utilisera donc la formulation **"IF (CONDITION) : (INSTRUCTION) ELSE : (INSTRUCTION)"**.

Pourquoi un test en **N<1** et non en **N=0** ? Simplement pour éviter d'entrer dans une boucle folle si la valeur entrée par l'utilisateur n'est pas un entier.

M.P.S.I.2 2014/15

## MARKOV DANS LE METRO

2015

Charlemagne

p Py21 q

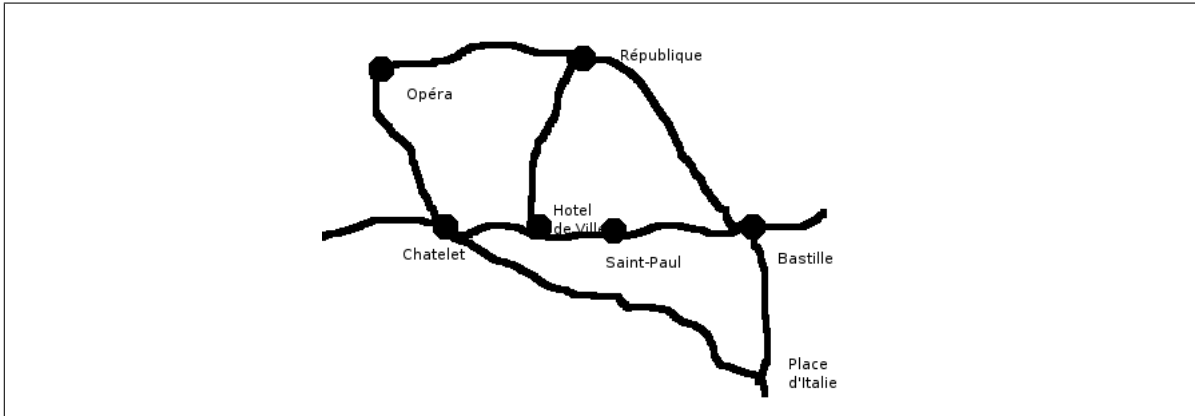
Ce T.D. vous propose de travailler sur les produits matriciels, et aussi d'étudier des suites récurrentes qui modélisent les déplacements "aléatoires" d'une population dont on connaît les probabilités de transitions.

On imagine qu'un groupe d'élèves de Charlemagne décide de se rendre tous ensemble au restaurant. Ils y vont en métro, mais leur comportement dans le métro est de type "markovien". A chaque changement, un certain pourcentage d'élèves descend et décide de prendre une nouvelle direction parmi celles qui sont disponibles, avec des probabilités connues. Peu à peu, ils se dispersent, et on regarde quel pourcentage d'élèves arrive au rendez-vous final en un nombre d'étapes fixé.

Par exemple :

- quand ils arrivent au métro Monsieur-Paul (*soyons laïc*), la moitié prend la direction La Défense, l'autre moitié prend la direction Vincennes.
- ceux qui arrivent à Hôtel de Ville se répartissent en trois groupes :  
ceux qui continuent (*soixante pour cent*)  
ceux qui prennent la ligne 11 en direction des Lilas (*trente pour cent*)  
ceux qui repartent vers Vincennes (*dix pour cent pour la cohérence, non ?*)

- à la station Bastille, cinquante pour cent restent dans le wagon, et les autres se répartissent équitablement entre les différentes directions disponibles
- à la station République (*on y trouve ceux venus par Hotel de Ville et la ligne 11 et ceux venus par Bastille et la ligne Pont de Sèvres*), soixante pour cent continuent sur la ligne où ils sont, trente-cinq pour cent se répartissent entre les diverses directions et cinq pour cent se perdent dans les couloirs (*le temps d'une étape*).
- et ainsi de suite.
- les élèves qui arrivent à la station où est le restaurant y restent. Sauf si vous décidez qu'ils sont un peu stupides.



Je vous laisse imaginer votre plan de métro très simplifié (car vous n'allez pas saisir et afficher des matrices de taille 300 sur 300...) : c'est ce qu'on appelle un graphe :

les stations en sont les sommets,  
les lignes qui les joignent sont les arêtes.

Vous numérotez vos stations.

Ensuite, il faut remplir une matrice pour dire quel pourcentage d'élèves étant dans la station  $i$  prend un métro vers la station  $j$ .

Dans notre exemple, la matrice se présentera sous la forme

| station de départ-> | Saint-Paul | Hotel de Ville | Bastille | République |
|---------------------|------------|----------------|----------|------------|
| Saint-Paul          | 0          | 1/10           |          |            |
| Hotel de Ville      | 1/2        | 0              |          |            |
| Bastille            | 1/2        | ...            | 0        |            |
| République          | 0          | 3/10           |          | 1/20       |

Quelle sera la colonne de la station où se trouve le restaurant.

Quel est le total des coefficients sur une colonne ?

Peut il exister des coefficients négatifs ?

Peut il y avoir un coefficient non nul sur la colonne Saint-Paul en ligne République ?

Si on note  $M$  la matrice carrée ainsi définie, si on note  $U_n$  la répartition des élèves dans les différentes stations à la  $n^{ième}$  étape (*qui est  $U_0$  ?*), expliquez ce que représente  $M.U_n$ .

On comprend qu'il va donc falloir calculer et afficher la matrice  $M^n$ .

Mais avant tout, même, il va falloir la créer. Ses éléments seront des lignes, qui elles mêmes seront formées de réels positifs.

Pour créer une ligne de 0, je rappelle la méthode : `[0]*n` où  $n$  est le nombre d'éléments souhaité (*ici, le nombre de stations, passé en constante au début de votre programme*).

Pour créer le tableau de telles lignes, c'est `matrice=[ [0]*n for i in range(n) ]`.

Ensuite, vous affectez des valeurs par `matrice[i][j]=...`

Vous devrez créer une procédure qui multiplie entre elles deux matrices carrées de taille  $n$  (je sais, un module fait ce type de travail à votre place, mais on est en maths, alors on recrée tout soi même).

La formule pour multiplier  $A$  par  $B$  c'est  $c_i^k = \sum_{j=1}^n a_i^j . b_j^k$ . Avec Python, ceci va devenir

```
sum(A[...][j]*B[j][...] for j in range(n)).
```

Il faudra donc effectuer une boucle sur  $i$  (**for i in range(n) :**) suivie d'une boucle sur  $j$  (**for j in...**).

Il ne vous restera plus qu'à créer un module pour afficher de manière conviviale le résultat de produits matriciels itérés.

En effet, pour élever une matrice à la puissance  $p$ , on crée la procédure de multiplication **def produit(a,b) :** et on la met en boucle **for r in range(p) : mp=produit(mp,m) .**

Après, vous pouvez de temps en temps (de manière aléatoire) remplacer **mp=produit(mp,m)** par **mp=produit(mp,incident)** où **incident** est une matrice à créer par vous même qui modélise soit un arrêt du trafic sur une ligne, soit la disparition des élèves de telle ou telle station, ou la multiplication par 2 pour une raison inconnue du nombre d'élèves présents à une station. Je vous fais confiance pour l'inventivité.



Ce T.D. sur les automates est inspiré d'un challenge de l'Epita.

La situation : une arène rectangulaire. Sur la partie centrale de cette arène, il y a des objets à ramasser. D'un point déterminé sur le bord, vous envoyez un automate/robot qui a pour mission de ramasser un certain nombre de ces objets et de les rapporter.

Cet automate sera sous la forme d'un programme Python que vous aurez construit et qui s'exécutera alors sans que vous puissiez interagir avec, ni le modifier.

A chaque instant de temps, votre robot aura le droit de regarder les cases voisines de sa position pour décider ce qu'il fait, sur quelle cas voisine il se déplace.

Les données.

Le terrain s'appelle **terrain**, et c'est une matrice de taille **N** sur **N** dont les cases sont toutes (ou presque) à la valeur 0.

Les éléments **terrain[0][k]**, **terrain[1][k]**, **terrain[N-2][k]** et **terrain[N-1][k]** ainsi que **terrain[k][0]**, **terrain[k][1]**, **terrain[k][N-2]** et **terrain[k][N-1]** pour  $k$  de 0 à 23 sont à la valeur -1 pour marquer le bord, sur lequel il vous est interdit d'aller.

Vingt éléments de la zone **terrain[i][j]** pour  $i$  et  $j$  entre 6 et 17 sont mis à la valeur 11 pour indiquer qu'il y a là un objet à ramasser.

La case **terrain[2][10]** sera votre case de départ (et d'arrivée), mise à la valeur 21.

|    |    |    |     |    |    |    |    |
|----|----|----|-----|----|----|----|----|
| -1 | -1 | -1 | ... | -1 | -1 | -1 | -1 |
| -1 | 0  | 0  | ... | 21 | 0  | 0  | -1 |
| -1 | 0  | 0  | ... | 0  | 0  | 0  | -1 |
| -1 | 0  | 11 | ... | 0  | 0  | 0  | -1 |
| ⋮  | ⋮  | ⋮  |     | ⋮  | ⋮  | ⋮  | ⋮  |
| -1 | 0  | 0  | ... | 0  | 0  | 0  | -1 |
| -1 | 0  | 0  | ... | 0  | 0  | 0  | -1 |
| -1 | -1 | -1 | ... | -1 | -1 | -1 | -1 |

(échelle non respectée).

Toute case que vous visiterez sera mise à la valeur 1.

Une fonction prédéfinie **observe**(*i*, *j*) étudie les cases voisines de la case **terrain**[*i*][*j*] et vous indique leur contenu (*qui peut donc valoir -1, 0, 1, 11 ou 21*). Votre vision se limite aux cases

très proches :

|          |            |          |            |          |
|----------|------------|----------|------------|----------|
|          |            | (i-2, j) |            |          |
|          | (i-1, j-1) | (i-1, j) | (i-1, j+1) |          |
| (i, j-2) | (i, j-1)   | (i, j)   | (i, j+1)   | (i, j+2) |
|          | (i+1, j-1) | (i+1, j) | (i+1, j+1) |          |
|          |            | (i+2, j) |            |          |

La fonction vous renvoie la liste [**terrain**[*i-2*][*j*], **terrain**[*i-1*][*j-1*], **terrain**[*i-1*][*j*] jusqu'à **terrain**[*i+2*][*j*]] (*lecture en ligne du tableau ci-dessus*).

Quand vous passez sur une case de valeur 11, votre **compteur** de ramassage devra s'incrémenter d'une unité, et la case sera mise à la valeur 1 puisque vous y serez passé.

Quand votre **compteur** de ramassage arrivera à 5 vous pourrez retourner à la case départ **terrain**[2][10]

Un compteur **temps** sera chargé de compter le nombre d'étapes nécessaires pour votre récolte.

MPSI 2/2014

**Variantes.**

Py22

Une fois que votre programme tourne correctement, je peux compliquer la règle :

•V1• vous n'avez pas le droit de passer deux fois au même endroit  
vous avez alors intérêt à créer une pile lifo qui mémorise vos déplacements précédents pour pouvoir remonter en arrière.

•V2• vous avez droit à deux automates qui fonctionnent en parallèle et sont en droit de communiquer

•V3• on met en concurrence deux programmes en même temps : le votre et celui d'un autre élève (*point de départ **terrain**[N-3][10] mis à la valeur 22*). Là où passe l'autre élève, les cases sont mises à la valeur 2. Le premier qui a tout ramassé a gagné.

En option, vous avez le droit de poser un obstacle à la valeur -1 qui pourra boucher la case arrivée de votre adversaire. Mais lui aussi a le droit de vous le faire.

En option aussi : la possibilité de détruire l'adversaire.

Un module d'affichage avec **Tkinter** doit se charger de rendre tout ceci visuel.

M.P.S.I.2 2014 **LES CINQ CARTES** 2015 Charlemagne

p Py23 q

Le point de départ de ce T.D. est un tour de cartes "de divination". Il se pratique avec un jeu de cinquante deux cartes ordinaire, et deux "magiciens" *C* et *D*.

*D* sort de la pièce. Le public tire ou choisit cinq cartes au hasard dans le paquet de cartes, faces visibles. Il les donne à *C* qui en met alors une de côté (le public voit bien laquelle, nommons la "carte mystère") et aligne les quatre autres sur la table.

On fait rentrer *D* qui regarde juste les quatre cartes alignées, réfléchit... et annonce alors qui est la "carte mystère", alors même qu'il ne l'a pas vue et qu'aucun détail supplémentaire ne lui a été

transmis.

Dans une variante, on n'aligne même pas les cartés, mais un membre du public lit même simplement la liste des quatre cartes, afin de bien convaincre que ce n'est que leurs valeurs faciales qui importent, et non leur disposition sur la table ou toute autre astuce.

Ce tour (imaginé par un matheux plutôt fou <sup>18</sup>) est en fait une affaire de codage (d'où le nom de  $C$  pour celui qui code en choisissant la "carte mystère" parmi les cinq cartes) et de décodage (d'où le nom de  $D$  pour le devin).

Première mission du programmeur : créer le "paquet de cinquante deux cartes", et créer un programme qui tire cinq cartes au hasard, ou laisse aussi la possibilité à l'utilisateur de choisir cinq cartes comme il veut.

Premier élément de ce tour : sur cinq cartes, il y en a au moins deux de la même "couleur" <sup>19</sup> par principe du pigeonier (ou " tiroirs et chassettes "). Le codeur choisit donc une des deux cartes de la couleur "en double" dont il fera la "carte mystère" (il se peut qu'il y ait deux couleurs en double, comme dans un tirage où les couleurs tirées seraient ♠, ♡, ♣, ♣, le codeur fait alors le choix d'une des couleurs en double). L'autre carte de cette couleur en double est alors la première carte posée sur la table. Il est alors aisé pour le décodeur  $D$  de connaître la couleur de la carte à deviner : la même que celle de la première carte de la liste exposée.

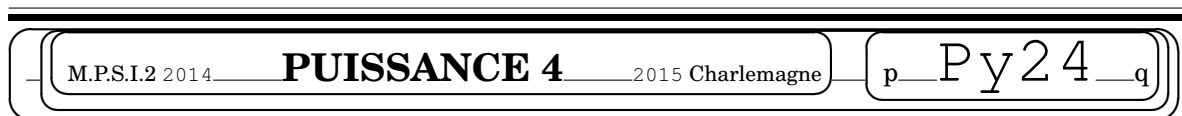
Il faut ensuite trouver la hauteur <sup>20</sup> de la carte mystère. Pour cela, on va partir de la hauteur de la première carte de la liste et avancer d'un nombre de pas codé en binaire par les trois cartes suivantes.

Problème : la différence de hauteurs peut atteindre 12 alors qu'un code binaire sur trois cartes ne peut pas dépasser 8.

On va donc avoir affaire à une petite astuce : laquelle des deux cartes de même couleur isole-t-on ?

On a deux cartes de la même couleur, mais de hauteurs différentes  $h_1$  et  $h_2$ . On les place fictivement sur un cercle  $1, 2, 3, \dots, roi$  qu'on referme en orientant par exemple dans le sens trigonométrique. On mesure alors les deux distances (dont la somme vaut 13) et on choisit celle qui est la plus petite, par exemple de  $h_1$  à  $h_2$ . On présentera alors la carte de hauteur  $h_1$  et on codera  $h_2 - h_1$ .

Pour saisir, un exemple : si les cinq cartes tirées sont (♡4), (♠Roi), (♡9), (◇Valet), (♣2), alors c'est (♡4) et (♡9) qu'on place sur le cercle "trigonométrique". La distance "4 à 9" vaut 5 et la distance "9 à 4" vaut 8. On va donc cacher (♡9), et exhiber (♡4), puis coder "+5" à l'aide de (♠Roi), (◇Valet), (♣2).



Vous connaissez tous le jeu de Puissance 4, variante du Morpion, dans lequel chaque joueur doit tenter d'aligner quatre pions de sa couleur, en ligne, colonne ou diagonale, et où les pions tombent dans chaque colonne sous l'effet de la pesanteur (*si vous vous demandez si j'ai bien décrit le jeu que vous connaissiez, ne vous inquiétez pas, c'est bien lui*).

<sup>18</sup>Cheney était capable d'écrire au tableau avec les deux mains, simultanément ; le membre de droite d'une équation avec la main droite tandis qu'il écrivait le membre de gauche avec la main gauche

<sup>19</sup>il y a quatre couleurs : ♠, ♡, ♣, ◇

<sup>20</sup>la hauteur, c'est  $1, 2, 3, \dots, roi$

J'ai d'ores et déjà tapé un programme disponible sur votre clef qui profite des possibilités de Tkinter pour proposer une version du jeu à deux. A vous de jouer un peu, mais surtout d'en comprendre la structure et le fonctionnement et de voir quels paramètres sont modifiables (largeur, hauteur, couleurs...).

Il manque (*volontairement*) une partie du programme.

Rassurez vous, il ne s'agit pas pour vous de créer un module dans lequel l'ordinateur prendrait l'initiative de jouer spontanément<sup>21</sup>.

Mais vous avez dû constater qu'il manquait une fonction d'évaluation chargée de regarder si un joueur avait ou non aligné quatre pions de sa couleur, et d'indiquer qu'il avait alors gagné. C'est donc bel et bien cela votre mission : le plateau de jeu est une matrice appelée terrain, dont les éléments valent 0 (case vide), 1 ou 2 (suivant la couleur du pion sur la case). Il faudra donc parcourir les lignes, colonnes et diagonales à la recherche de quatre valeurs identiques consécutives valant 1 ou 2.

MPSI 2/2014

## Pour aller plus loin

Py24

Si vraiment vous vous en sentez capable, perfectionnez le programme pour que ce soit l'ordinateur qui joue (tester une par une les colonnes et voir si la situation est favorable, choisir celle dont la situation est la plus favorable). La grande difficulté sera d'avoir une fonction d'évaluation de l'état du terrain.

Une année, un élève a transformé le programme pour en faire une version jouable en réseau (entrées/sorties, échanges de données, contactez Louis Lenief en Spé, ou bientôt en école).

Passez à la version en dimension 3.

M.P.S.I.2 2014

## Traducteur SMS

2015 Charlemagne

p Py25 q

Le but de ce T.D. sur les manipulations de chaînes est de prendre un texte en bonne et belle langue française et de le transformer en SMS, c'est à dire dans un langage simplifié qui ne respecte pas la syntaxe mais économise les lettres à taper/écrire. Surtout, ne me dénoncez pas ensuite auprès de Mme Seguret ou de M Ayoun.

Vous aurez donc initialement un texte à charger par les traditionnels

```
Fic = open(...txt, 'r')
Texte = Fic.read()
Fic.close()
```

Ensuite, il faudra parcourir ce texte lettre à lettre (`for k in range(lenTexte) : Texte[k]` ou mieux encore `for lettre in Texte :`), ou mot à mot pour :

- tous basculer en minuscules (*c'est `Texte.lower()`*)
- remplacer les consonnes doubles par des consonnes simples
- effacer les **s** muets en fin de mot (*c'est à dire les **s** suivis d'un espace ou d'une ponctuation*<sup>22</sup>)
- faire de même avec les **t** muets de fin de mot (*mais lesquels ne sont pas muets ? c'est là le but de ma question*)
- remplacer les **un** par **1** et les **de** par **2** (*quand il s'agit de mots*)<sup>23</sup>
- remplacer les **eau** et **au** par **o**
- supprimer les accents

<sup>21</sup>et de surcroît intelligemment, parce que sinon, choisir une colonne au hasard, c'est facile

<sup>22</sup>mais attention aux exceptions, comme le mot "plus" ou même "moins" si votre interlocuteur est du Sud

<sup>23</sup>et remplacer aussi les **deux** par **2**

- remplacer les **qu** par des **k** (*ou des c*)
- remplacer **c'est** par **C** et **j'ai** par **G**
- remplacer les **eu** par **e** tout court
- remplacer les **ç** par des **s**
- toute autre idée est la bienvenue.

Il ne vous restera plus ensuite qu'à afficher le résultat final, et vérifier si on comprend encore.

MPSI 2/2014

## Pour aller plus loin

Py25

Il existe un Python un type appelé **dictionnaire**, qui est une liste pour laquelle on accède aux éléments non pas en donnant leur index (nombre entier entre 0 et `len(dico)-1`) mais en indiquant la clef de cet élément.

C'est ainsi qu'on pourrait créer `dico = []` et définir `dico['eau']='o'`, `dico['au']='o'`, `dico['qu']='k'` et ainsi de suite.

A vous de voir si c'est pertinent.

Vous pouvez aussi tenter (*G bi1 di "tenT"*) de créer un programme qui fait le travail en sens inverse, du SMS vers le bon français. Rien n'interdit ensuite de faire des allers-retours d'un langage à l'autre pour finalement perdre tout le sens du texte initial.