Programmation

Exercice 1. *Exercice théorique sur la portée*

1. Quelles sont les variables locales et globales de `f` ? Qu'affiche le programme suivant ?

```
def f():
    global a
    a = a + 1
    c = 2 * a
    return a + b + c
a, b, c = 3, 4, 5
print(f(), a, b, c)
```

2. Qu'affiche le programme suivant ? Justifier.

```
def f(x):
    global b
    b = 19
    return x + 1
a, b = 3, 4
print(f(a) + b)
```

3. Qu'affiche le programme suivant ? Justifier.

```
def g(x):
    global v
    v = 1000
    return 2 * x
def f(x):
    v = 1
    return g(x) + v
a = 3
print(f(6) + a)
```

Exercice 2. *Manipulation de fonctions*

1. Définir des fonctions anonymes effectuant les tâches suivantes :
   - **a.** fonction qui ajoute 1 à son argument ;
   - **b.** fonction qui prend x et y et renvoie x^y ;
   - **c.** fonction qui divise par deux son argument s'il est pair, et le multiplie par 3 et ajoute 1 sinon.
2. Écrire une fonction `derive` qui prend en argument une fonction f et un flottant $\epsilon > 0$ et renvoie la fonction $x \mapsto \frac{1}{\epsilon}(f(x+\epsilon) - f(x))$. Tester la fonction avec quelques fonctions usuelles.
3. Définir les fonctions suivantes [1] :
   - **a.** `map(f, L)` renvoie la liste constituée des images des éléments de `L` par la fonction `f` ;
   - **b.** `filter(p, L)` renvoie la sous-liste constituée des éléments `x` de `L` tels que `p(x)` renvoie `True` ;
   - **c.** `reduce(f, L, e)` renvoie la valeur obtenue en appliquant de manière cumulative la fonction de deux paramètres `f` aux éléments de `L` en partant de `e` : par exemple, la somme des éléments d'une liste pourra s'obtenir avec `reduce(lambda x,y : x+y, L, 0)`.
4. Écrire les fonctions décrites ci-dessous, à l'aide des fonctions de la question précédente :
   - **a.** une fonction qui prend en entrée une liste de liste d'entiers, et renvoie la liste obtenue en ajoutant 1 à chacun de ces entiers ;

1. Ces fonctions existent en Python, parfois dans des bibliothèques spécialisées, et avec des *itérateurs* comme types de retour.

b. une fonction qui transforme une liste d'entiers en une liste de listes à un seul élément chacune (par exemple `[1,2,3]` devient `[[1],[2],[3]]`) ;
c. une fonction qui prend en entrée une liste et un entier n et renvoie la sous-liste des éléments de `L` qui ne sont pas multiples de n.

Exercice 3. *Classes et héritage*
On considère le code suivant, où le mot-clef `pass` sert à définir la classe `D` avec aucune fonction (seulement celles dont elle hérite).

```
class A:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def f(self):
        return self._x + self._y

class B(A):
    def f(self):
        return self._x - self._y

class C(A):
    def g(self):
        return self._x * self._y
    def h(self):
        return (self.f(), self.g())

class D(B, C):
    pass
```

Soit `a = A(5, 10), b = B(5, 10), c = C(5, 10)` et `d = D(5, 10)`.
Que renvoient (valeur de retour ou erreur) les appels suivants :

1. `a.f(), b.f(), c.f(), d.f()` ?
2. `a.g(), b.g(), c.g(), d.g()` ?
3. `a.h(), b.h(), c.h(), d.h()` ?

Exercice 4. *Fibonacci*
La suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ est définie par $F_0 = F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$ pour tout $n \in \mathbb{N}$.

✎ Programmer une fonction `fibonacci` pour répondre aux questions suivantes[2] :

a. Que vaut F_{10} ?
b. Que vaut F_{100} ?
c. F_{10^7} est-il divisible par 1515 ?
d. *(difficile)* Que vaut $F_{10^{12}} \mod 2042$?

Exercice 5. *Programmation objet*
On souhaite écrire des classes `Rationnel` et `Gaussien` pour représenter et manipuler des rationnels et des entiers de Gauss (de la forme $a + ib$ où $a, b \in \mathbb{Z}$ et $i^2 = -1$).

1. Écrire une classe `Vecteur` de vecteurs entiers, avec représentation (méthode `__repr__`) sous la forme `(a,b)`, et une méthode `norme` qui calcule la norme $\sqrt{a^2+b^2}$. La création d'un vecteur (a, b) sera effectuée par `Vecteur(a,b)`.
2. Ajouter des méthodes d'addition (`__add__`) et de test d'égalité (`__eq__`) à la classe vecteur.
3. Écrire une classe `Gaussien` qui hérite de `Vecteur` pour les entiers de Gauss. On devra pouvoir additionner, multiplier, soustraire et calculer le conjugué (avec une méthode `conjugue`).

2. Vous pouvez commencer par coder une version naïve, et l'améliorer progressivement si besoin. L'objectif est de ne pas dépasser un temps de calcul de l'ordre d'une seconde par question.

4. Écrire une classe `Rationnel` qui hérite de `Vecteur` pour les nombres rationnels, représentés comme quotients de deux entiers. *Attention à ce que le test d'égalité reste cohérent chez les rationnels, en le réécrivant si nécessaire.*

On souhaite maintenant écrire une classe `Polynome` pour manipuler des polynômes à coefficients rationnels ou Gaussien. La représentation interne des polynômes sera un dictionnaire qui à un entier k associe le coefficient du monôme de degré k du polynôme. Par exemple, le polynôme $2X^3 - X + 5$ est représenté par le dictionnaire `{0:5, 1:-1, 3:2}`.

5. Écrire la classe `Polynome` avec l'initialisation (à partir d'un dictionnaire), la représentation, l'addition et la multiplication. Tester l'implantation avec des polynômes à coefficients entiers (`int`), `Rationnels` et `Gaussiens`.
6. Ajouter une méthode d'évaluation des polynômes, pour que `p(5)` renvoie l'évaluation de `p` en la valeur 5. Il faut pour cela définir une méthode `__call__`. *La fonction ne doit pas effectuer plus de d multiplications et $d-1$ additions, où d est le degré du polynôme.*
7. Ajouter une méthode qui renvoie le coefficient du monôme de degré i lorsque l'utilisateur demande `p[i]`. Il faut pour cela définir une méthode `__getitem__`. *Cette méthode doit lever une exception `ValueError` si et seulement si $i < 0$.*

Exercice 6. *Compression de données*

La compression de type RLE[3] permet de compresser des images en noir et blanc. L'idée est de remplacer plusieurs pixels consécutifs ayant la même couleur par leur nombre suivi de la couleur commune. Par exemple, supposons qu'on note `N` et `B` les pixels noir et blanc respectivement. Alors l'*image* représentée par la chaîne `BBBBBBBBBBNNBBBBNBNNBBBBBB` sera compressé en `10B2N4B1N1B2N2N6B`.

1. Écrire les algorithmes d'encodage et de décodage pour le code décrit.

On remarque qu'écrire `1B` ou `1N` fait perdre de la place. On définit un nouveau code qui fonctionne comme l'ancien mais qui écrit `B` à la place de `1B`, et `N` à la place de `1N`.

2. Montrer que le code défini ci-dessus peut se décoder de manière unique.
3. Écrire les algorithmes d'encodage et de décodage correspondant.

Exercice 7. *Chiffrements historiques*

Dans cet exercice, les messages clairs et chiffrés seront des mots écrits sur les 26 lettres de l'alphabet latin, en majuscule.

1. Écrire les algorithmes de chiffrement et de déchiffrement du système de César.
2. Même question pour le chiffrement de Vigenère.

Dans la suite de l'exercice, on s'intéresse à la cryptanalyse de ces systèmes. On effectue, pour le chiffrement de César, une attaque par fréquence : on suppose que le texte chiffré est un texte écrit en français (sans espace ni ponctuation) et on utilise le fait que la lettre `E` est la plus fréquente en français. On cherche donc la lettre la plus fréquente du texte pour en déduire le décalage.

3. Écrire un algorithme de cryptanalyse du chiffrement de César par analyse de fréquence.
4. Déchiffrer le texte suivant (*bonus : d'où est-il tiré ?*) :
`TYVQCRAFCZVIFJVKKVRLTRWVULTRERCJFLJCVKIFETULKZCCVLCHLZFDSIRXVRZKCVSRCFEGFLMRZKCZIVJ`
`FLJUVLOTFVLIJVEKIVCRTVJZTZFEGVLKRGGFIKVIJVJSRZJVIJ`

On peut utiliser la même analyse de fréquence pour le cas du chiffrement de Vigenère, mais il faut d'abord trouver la longueur de la clef. Pour cela, on utilise le fait que certains triplets de lettres consécutives sont fréquents en français (comme `ION` ou `QUE`). Pour trouver la longueur de la clef, on calcule donc tous les triplets de lettres apparaissant au moins deux fois dans le texte, avec les positions auxquelles ils apparaissent. Le PGCD de leurs distances est (probablement) la longueur de la clef.

5. Écrire les fonctions nécessaires pour retrouver la longueur de la clef dans le chiffrement de Vigenère.
6. En déduire un algorithme de cryptanalyse du chiffrement de Vigenère.

3. *Run length encoding*, ou codage par plage.

7. Déchiffrer le texte suivant[4] (*même bonus !*) :

```
FACWDCPAEAPEGEKGSDYKNACYAVSPRKGTDMDGSSYFGOQWUWRXXWGTSYFGEEEAUSTYJFECGJGUCLGOMTWMKVP
MLUEJPDCGGEFFEGSMVESIECRRLAGNCIKCMDRLUOJHAZKXPGOEIVWUDTTSXERSMRACXLQUIHJQIIELTAKIJU
LTWUJABTKFEQILVEGENGSSINCNIPMKIARWXONEAVMTQWRAHPWUOARGKRTXANNPZSKTAEKGNHELKOCHWNIBQ
WPSTLGTIOSFRLPXIWEEEJNEHWGWFUPWUDJZWPTSIECRHHWURPJSNEHPSTGTWUQMBIKWRJRWOEGKDCCTIKFA
KSATBPPSAESIKNITYWUDTQSTAXWWVDTXWTRTWFWEHEMEUCIGOBGIVCRQVWPEIEUJAXXDGCXIDNEEENGSTHW
TOJPSKTPZWELPVWETXXMFESYFGJTXWGAJQANITYVGLTQTTUCENGUVPSPTSIKVECITTEHPZQMBIWVAXXHCRI
MVGMPVUJITRFGSKIJUDTYPJEJVWUIAQSTCWEAVDJRHCSPPDQNVIYTEASLVACXKQUHPWEOISFCMXRUKDTWSX
EHXWGTSIKQNEEFVAASFFEKIDQUGWMPPTXAVPPUMGTCSMGDPRKWNBSMEHDMJCCPVJGAJBDGGTRSKTQISWCDY
HGTXPDGSTVJCIIGGPTGIKGSUPSPCHXSPTDXVWNRSMFEIEFVOIHWNAJXJGPDYJILXWKGRPYXQNSHWUEHTGEH
TWDGSSIMZMPMFUAAEXQIHHWUMPMFUGDYJFEHUMGLTWDCNXIJGSSYNGNIHWUTUEAUAXIFVSPMYPEGYFGSTYD
GISIWQCRYHCIIWSVEIINKDTHGWVGMWTSPRKVRPZSKLTXKCNHKAVEAIKROXVIWEAIXTOXHKGRPMLOOXRKXIU
EHTEHPWNEKIJFUYSMTDTTMKSJRWJEJVWKLPZSPCPMLCICWANOGWIWEHYJNAVEMEHTEVGUMOANOBILTEHHWO
OCXKQUXPSREGGMVDTWXGUMVGWGTWLTOXWTTAHMWTSQVMNACXSWPAIAPAXVWVCDQEGSJWHGNSYKFAQSJFIAL
WUIIEHTIHHWERPMFVEEYAUIARWRUIVWUIHXWTAJFWUOXRVQUASMTEJBVGSTGZCUUJWTUCMFUTPRLNEHQSKNH
```

Exercice 8. *Cryptosystème RSA*

Pour effectuer la génération des clefs privés et publiques dans RSA il faut savoir faire des calculs *modulo* un entier t. En particulier, il faut savoir inverser *modulo* $\varphi(N)$. Pour cela, on peut utiliser l'algorithme d'Euclide étendu, qui calcule le pgcd de deux entiers a et b, et les coefficients de Bézout associés u et v tels que $au + bv = \text{pgcd}(a, b)$. Il peut être défini de manière récursive en utilisant le fait que $\text{pgcd}(a, b) = \text{pgcd}(b, a \mod b)$.

1. Écrire, d'abord sur papier, l'algorithme d'Euclide étendu vaguement esquissé[5] ci-dessus. L'algorithme prend en entrée deux entiers `a` et `b` et renvoie un triplet `(g, u, v)` où $g = au + bv$ est le pgcd de a et b.
2. Démontrer que a est inversible *modulo* n si et seulement si $\text{pgcd}(a, n) = 1$. Comment trouver dans ce cas l'inverse de a à partir des coefficients de Bézout ?
3. En déduire un algorithme qui prend en entrée deux entiers a et n et renvoie l'inverse de a *modulo* n s'il existe, et soulève une exception `ValueError : a n'est pas inversible modulo n` sinon.
4. Écrire un algorithme d'exponentiation modulaire, qui prend en entrée a, e, et m et renvoie $a^e \mod n$. *Attention, votre algorithme ne doit pas calculer d'abord a^e puis le reste modulo n, mais bien faire les calculs au fur et à mesure (on n'utilisera donc pas `**`).*
5. À partir des briques de base écrites précédemment, écrire un algorithme de génération de clefs RSA. L'algorithme prend en entrée deux nombres premiers p et q et renvoie le couple `(pk, sk)` où `pk` est la clef publique constituée de $N = pq$ et d'un entier e, et `sk` est la clef secrète constituée d'un entier d, avec les caractéristiques vues en cours. *Note : le module `random` contient la fonction `randint(a, b)` qui renvoie un entier aléatoire entre `a` et `b`, inclus.*
6. Écrire les algorithmes d'encodage et de décodage pour RSA.
7. Comment tester si un nombre est premier ? Et comment générer des nombres premiers aléatoires ? *Question de recherche bibliographique.*

4. Il est probable qu'il faille un peu travailler *à la main* pour retrouver le texte…
5. Il reste du boulot !