# TP 7
# Révisions et consolidation

*Ce TP ne contient pas de concepts nouveaux. Le but est seulement d'appliquer ce qui a été vu avant. Éventuellement il faut faire attention à bien choisir sa méthode, boucle* `while` *ou* `for`*, itérer sur les listes ou sur les éléments, bien ajuster les bornes des boucles et des* `range`*.*

**Exercice 1.** On représente les nombres complexes par des tuples : le nombre $z = a + bi$ correspond à `(a, b)` (on n'utilise pas le type Python `complex` : `a` et `b` sont tous les deux des nombres, aussi bien `int` ou `float`).

1. Écrire une fonction `somme_complexe(z, w)` qui prend en argument deux tuples `z`, `w`, représentant des nombres complexes $z$, $w$, et qui retourne un tuple correspondant au nombre complexe $z + w$.
   *Démarrage :*
   ```
   def somme_complexe(z, w):
       (a, b) = z
       (c, d) = w
       return ...
   ```
2. Écrire une fonction `produit_complexe(z, w)` qui prend en argument deux tuples `z`, `w`, représentant des nombres complexes $z$, $w$, et qui retourne un tuple correspondant au nombre complexe $z \times w$.
3. En utilisant la fonction précédente, écrire une fonction `puissance_complexe(z, n)` qui prend en argument un tuple `z` représentant un nombre complexe $z$, et un entier $n$ (positif), et retourne un tuple correspondant au nombre complexe $z^n$.
   *Méthode : une boucle, où une variable accumulatrice* `p` *représente les puissances successives de* `z`*.*

**Exercice 2.** On souhaite vérifier que les quotients de deux termes successifs de la suite de Fibonacci se rapprochent du nombe d'or. Pour que cela ait du sens on ne peut pas démarrer la suite de Fibonacci $(F_i)_{i\in\mathbb{N}}$ avec $F_0 = 0$ et $F_1 = 1$ (sinon, $F_1/F_0$ divise par zéro !), on décide donc de démarrer directement avec les deux premiers termes $F_1 = 1$ et $F_2 = 1$.
Écrire une fonction `fibonacci_quotients(n)` qui calcule avec un tuple `(x, y)` deux termes successifs de la suite de Fibonacci et affiche les quotients successifs $F_{i+1}/F_i$ pour $1 \leqslant i < n$.
*Méthode : une boucle avec un indice* `i` *où à chaque début de boucle* `x` *doit représenter* $F_i$ *et* `y` *doit représenter* $F_{i+1}$*.*

**Exercice 3.** *Suite de Tribonacci*
C'est la suite $(T_i)_{i\in\mathbb{N}}$ définie par la relation de récurrence

$$\forall i \in \mathbb{N}, \quad T_{i+3} = T_{i+2} + T_{i+1} + T_i \tag{1}$$

avec les conditions initiales $T_0 = 0$, $T_1 = 0$, $T_2 = 1$ (c'est le minimum pour avoir une suite bien définie et qui ne soit pas la suite nulle).
Programmer cette suite de deux façons différentes :

1. C'est une fonction `tribonacci(n)` qui utilise un tuple `(x, y, z)` représentant trois termes consécutifs de la suite, et retourne le terme $T_n$.
   *Méthode : comme précédemment, dans une boucle il faut qu'en début de boucle et quand l'indice est* `i` *alors* `x` *correspond à* $T_i$*,* `y` *à* $T_{i+1}$ *et* `z` *à* $T_{i+2}$*.*
2. C'est une fonction `tribonacci_liste(n)` qui retourne une liste de $n$ éléments, les termes d'indice 0 à $n-1$.
   *Méthode : démarrer avec une liste de n éléments remplie de zéros, puis la remplir au fur et à mesure avec les termes de la suite.*

**Exercice 4.** On s'intéresse à la recherche de *triplets Pythagoriciens* : les $(a, b, c) \in \mathbb{Z}^3$ tels que

$$a^2 + b^2 = c^2 \tag{2}$$

On remarque les choses suivantes :

1. Si $(a, b, c)$ est une solution alors on en obtient une autre dès qu'on change de signe l'un des trois nombres.
2. Il y a de nombreuses solutions évidentes si $a = 0$ ($b = \pm c$) ou de même $b = 0$ ($a = \pm c$).
3. Il peut y avoir (et il y a !) une infinité de solutions : plus $a$ ou $b$ est grand, plus il faut chercher $c$ grand (ou vu dans l'autre sens, plus $c$ est grand plus il faut chercher $a$ et $b$ grands).

4. On peut échanger $a$ et $b$ ce qui donne encore une solution.

Bref, on veut énumérer toutes les solutions possibles $(a, b, c)$ et on se limite à des nombres strictement positifs, avec une borne maximale donnée (on ne s'intéresse pas forcément aux solutions qui apparaitront deux fois avec $a$ et $b$ échangés : tant pis).
Écrire une fonction `pythagore(N)` qui retourne la liste de *tous* les triplets pythagoriciens $(a, b, c)$ avec $0 < a, b, c \leqslant N$.
*Méthode : il faut une triple boucle pour énumérer toutes les solutions ; pour les accumuler dans une liste, il faut démarrer par une liste vide et ajouter les solutions qu'on trouve peu à peu avec la méthode* ***`append`****.*

**Exercice 5.**

1. Écrire une fonction `somme_inverse(n)` qui calcule la somme suivante :

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \tag{3}$$

2. Écrire une fonction `somme_inverse_carres(n)` qui calcule la somme suivante :

$$\sum_{i=1}^{n} \frac{1}{i^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2} \tag{4}$$

3. Tester ces fonctions avec des valeurs de $n$ de plus en plus grandes. Qu'observe-t-on ?

**Exercice 6.** Écrire une fonction `concatene(L)` qui prend en argument une *liste* de *chaines de caractères* et qui concatène toutes les chaines en une seule. On doit par exemple obtenir :

```
>>> concatene(["un", "deux", "trois"])
'undeuxtrois'
```

**Exercice 7.** Écrire une fonction `colle_mots(s)` qui prend en argument une chaine de caractère `s` et en retire tous les espaces (donc elle *colle* les mots) !
Exemple : on doit obtenir

```
>>> colle_mots("j'aime les TP d'informatique")
"j'aimelesTPd'informatique"
```

*Méthode : boucle avec une variable accumulatrice* `t` *qui concatène les caractères uns par uns, si ce ne sont pas des epsaces.*
Amélioration : retirer tous les caractères (points, virgules, apostrophes, ...) qui ne sont pas des lettres ou des chiffres. Ces derniers s'appellent *caractères alphanumériques* et il y a une méthode pour tester si un caractère `x` est alphanumérique, de la forme `x.is...()` et décrite dans `help(str)`.

```
>>> colle_mots_bonus("j'aime les TP d'informatique")
'jaimelesTPdinformatique'
```

**Exercice 8.** On admet qu'on peut calculer la fonction arctangente pour des nombres réels $x$ assez petits par la formule suivante :

$$\arctan(x) = \lim_{n \to +\infty} \sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{2i+1} \tag{5}$$

1. Écrire une fonction `arctan(x, n)` qui calcule la somme de droite

$$\sum_{i=0}^{n} \frac{x^{2i+1}}{2i+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \tag{6}$$

   *Méthode : boucle habituelle avec une variable accumulatrice* `s`*. Pour savoir si on additionne ou si on soustrait, on peut tester la parité de l'indice* `i`*.*

2. Sachant que $\frac{\pi}{4} = \arctan(1)$, en déduire une fonction `pi_arctan(n)` qui calcule le nombre $\pi$ avec la formule ci-dessus. La tester avec des valeurs de `n` de plus en plus grandes.
3. Sachant que $\frac{\pi}{4} = 4\arctan(\frac{1}{5}) - \arctan(\frac{1}{239})$, en déduire une autre fonction `pi_arctan_machin(n)` qui calcule le nombre $\pi$. De même, la tester avec des valeurs de `n` de plus en plus grandes. Que remarque-t-on ?

**Exercice 9.** Écrire une fonction `anagrammes(s)` qui prend en argument une chaine de caractères `s` et qui retourne la *liste* de tous les anagrammes qu'on peut faire à partir de `s`, c'est à dire les chaines obtenues en permutant les caractères de `s`.
Exemple :

```
>>> anagrammes("BIO")
['BIO', 'BOI', 'IBO', 'IOB', 'OBI', 'OIB']
```

*Méthode : **une** possibilité est d'utiliser une boucle à répéter n fois (où n est la longueur de `s`), pour $k = 1, \dots, n$, et qui calcule en même temps deux listes (de même longueur) en parallèle :*

- *`L` est la liste de tous les anagrammes qu'on peut former en prenant k lettres parmi celles de `s`,*
- *pour tout indice `i`, `M[i]` est la chaine formée de toutes les $n - k$ lettres de `s` qui ne sont pas utilisées dans `L[i]`.*

*À chaque étape on forme donc tous les anagrammes possibles avec une lettre de plus, prise parmi `M[i]` et à concaténer à `L[i]`.*
*On pourra utiliser toutes les opérations sur les chaines : concaténation, sélection d'une tranche.*
*Attention pour faire des essais, même en écrivant un programme correct, dès qu'il y a plus de 6–7 lettres il y a des milliers d'anagrammes possibles et le programme sera lent.*