



# Python pour les Blondes

## Quelques éléments de programmation et modélisation de systèmes chimiques

---

Alain

# Activité 1

---

## On commence en douceur ?

---

Vous trouverez, si nécessaire, en annexe 5.1 de ce document, une présentation de l'environnement de développement que j'utilise.

### 1 Quelques rappels « théoriques »...

#### 1.1 Types de données

Le langage de programmation python utilise plusieurs *types de données*. Les plus courants sont les suivants :

Types de données	Nom en python
Nombre entier	<code>int</code>
Nombre « à virgule »	<code>float</code>
Variable booléenne : vrai ou faux (True/False)	<code>bool</code>
Liste (ordonnée)	<code>list</code>
Chaîne de caractères (du texte...)	<code>str</code>

Ponctuellement, on pourra en rencontrer d'autres... comme par exemple le type tuple correspondant à un couple de valeurs (de n'importe quel type) séparées par une virgule.

Lors de la conception d'un programme, il est important de savoir le type de données de chacune des variables utilisées. Par exemple, si on tente de faire des opérations mathématiques sur une variable de type chaîne de caractères, on obtient une erreur (même si le texte est constitué de chiffres). Pour savoir le type de données d'une variable, on peut utiliser la fonction ***type()***.

#### 1.2 Un premier mot sur les variables

En python, comme dans tout autre langage, l'affectation d'une valeur (issue d'une initialisation, d'un calcul, du résultat de l'appel d'une fonction...) se fait dans une variable.

Un nom de variable, valide en python, est constitué de lettres majuscules ou minuscules (non accentuées si possible!), de chiffres et de tirets bas « `_` ». Elle ne doit pas commencer par un chiffre et ne doit pas contenir d'espace ni de caractères comme `+`, `-`, `*`...

Le langage python est sensible à la casse : une différence est faite entre majuscules et minuscules.

Chaque fois qu'une affectation est exécutée :

- une variable est créée dans la « liste » des variables (si elle ne l'était pas déjà) ;

- cette variable « pointe » vers une zone mémoire dans laquelle est stockée le contenu de la variable : entier, flottant, chaîne de caractères, liste...



Aucun nom de variable ne peut apparaître à droite du signe = si elle n'a pas été affectée au préalable !

## a

## Stockage des variables

### a.1 Codage des entiers

L'information élémentaire est stockée sous forme de bit qui peut valoir soit **0** soit **1**. Ces bits sont regroupés par octet : un octet = 8 bits.

Ainsi, par exemple, le nombre  $43 = 32 + 8 + 4 + 1 = 2^5 + 2^3 + 2^1 + 2^0$  peut être codé, en base 2 par 101011 et être stocké dans un octet : 00101011. Si cette représentation est fort pratique au niveau machine, pour communiquer la représentation en hexadécimal est plus facile : et le nombre s'écrira finalement : 2B en hexadécimal.

Voici, pour la petite histoire, un petit bug à 370 millions d'euros !

*Dans le système de pilotage de la fusée Ariane 4, une variable était allouée à l'accélération horizontale. Sa valeur décimale étant d'environ 64 (dans leur système d'unités), on décida de la stocker dans un registre mémoire de 8 bits. Lors du passage à Ariane 5, « on » ne prit pas garde de réviser cette partie du système qui fonctionnait à la perfection. Mais... Ariane 5 était bien plus puissante et l'accélération pouvait atteindre une valeur de 300...*

La plupart des systèmes traitent maintenant les entiers sur 32 bits ou 64 bits. En python, la taille n'est limitée que par la mémoire puisque les entiers sont stockés sous forme continue d'octets.

### a.2 Codage des nombres à virgule flottante

Ouh, là, là... ça devient trop compliqué pour nous.

Par exemple que pour coder un nombre flottant en simple précision sur 32 bits on écrit ce nombre sous la forme  $(-1)^S \times M \times 2^{E-127}$

- S est le bit de signe ;
- E est l'exposant codé sur 8 bits
- M est la mantisse : c'est un nombre réel compris entre 1 et 2 :  $1 \leq M < 2$  dont on code les chiffres après la virgule sur 23 bits (comme  $2^{23} = 8388608$  on a environ 7 chiffres significatifs)

On se fiche un peu de cette salade interne mais, il faudra parfois être vigilant. Certains nombres réels, même très simple (comme par exemple 0,1) n'est pas stocké sous forme exacte. Et donc, si on teste l'égalité entre les nombre 0.3 et  $3*0.1$  ; le résultat sera faux. Ainsi, si le test de l'égalité entre deux nombres réels est critique au niveau du programme, il sera préférable de vérifier si la valeur absolue de la différence entre les nombres n'est pas plus petite qu'une valeur fixée arbitrairement ( $10^{-16}$  conviendrait dans le cas de  $3*0.1 - 0.3$ ).

## b

## Codage d'un caractère

On ne citera que le code ASCII qui, depuis les années 1960, permet de coder les caractères sur un octet. Ce système ne permet pas, en particulier, la gestion des accents !

En python, on accède au code ASCII d'un caractère grâce à l'instruction **ord(caractere)** et, inversement, au caractère à l'aide de **chr(code)**. Par exemple `chr(65)` retourne la lettre "A" et `ord("A")` retourne l'entier 65. Le code ASCII de la lettre "a" vaut 97.

Ces instructions, qui vous seront rappelées si besoin, sont fort utiles pour tout ce qui concerne le codage de message (code de CÉSAR, VIGÉNÈRE ...).

## 1.3 Entrées/sorties élémentaires

### a Demander une information à l'utilisateur

C'est l'instruction `reponse = input("texte affiché à l'écran")` qui permet de stocker dans la variable `reponse` la réponse de l'utilisateur.



`reponse` est une variable de type chaîne de caractères, même si on demande d'introduire un nombre ! Si l'on souhaite obtenir un nombre entier ou réel, une conversion explicite est nécessaire : elle se fera à l'aide, respectivement, des instructions `entier = int(reponse)` ou `reel = float(reponse)`.



Si vous utilisez Spyder, la saisie se fait dans la console ! Si vous écrivez `ch = input("machin")`, `machin` apparaît dans la console, il faut cliquer à droite de cette ligne et valider.

### b Imprimer un résultat

C'est, bien sûr l'instruction `print(chaine)` qui permet d'imprimer la chaîne de caractère `chaine`. Pour imprimer un nombre (entier ou réel), on utilisera l'instruction `chaine = str(nombre)` si la conversion implicite n'est pas possible.

Une instruction comme `print("R = ", R, " J.K-1.mol-1")` est valable (pour peu que l'on ait écrit la variable `R = 8.314` au préalable).

## 1.4 Avec des si...

### a Objectif

Il s'agit de permettre l'exécution d'une série d'instruction uniquement si une condition est vérifiée.

```
INITIALISATION de paramètres
si condition(s) sur ces paramètres alors
| ... suite d'instruction à effectuer
fin
... instructions effectuées systématiquement
```

On peut, parfois souhaiter exécuter certaines instructions si une condition est vérifiée, et d'autres instructions dans le cas contraire :

```
INITIALISATION de paramètres
si condition(s) sur ces paramètres alors
| ... suite d'instruction à effectuer si la condition est vérifiée
fin
sinon
| ... suite d'instruction à effectuer si la condition n'est pas vérifiée
fin
... instructions effectuées systématiquement
```

Les conditions peuvent être multiples... il faudra prendre soin de s'assurer s'il s'agit d'un *et* ou d'un *ou* entre les différents tests mis en jeu.

**b Implémentation**

En python, `si` se traduit par `if` et `sinon` se traduit par `else` .  
La structure générale est la suivante :

```
1 initialisations
2 if condition à vérifier :
3     ... suite d'instruction à effectuer
4 suite du programme
```

ou

```
1 initialisations
2 if condition à vérifier :
3     ... suite d'instruction à effectuer si la condition est
      vérifiée\\
4 else :
5     ... suite d'instruction à effectuer si la condition n'est
      vérifiée\\
6 suite du programme
```

ou encore

```
1 initialisations
2 if condition1 :
3     ... suite d'instruction à effectuer si la condition1 est
      vérifiée\\
4 elif condition2 :
5     ... suite d'instruction à effectuer si la condition2 n'est
      vérifiée\\
6 else :
7     ... à effectuer si ni la condition1 ni la condition2 n'est
      vérifiée
8
9 suite du programme
```

La condition mise en jeu dans l'instruction *if* ou *elif* résulte la plupart du temps du résultat :

- du test d'égalité entre deux nombres : on utilise alors l'opérateur `==`
- du test d'une différence entre deux nombres à l'aide de l'opérateur `!=`
- d'une comparaison de deux nombres à l'aide des opérateurs : `<`, `<=`, `>`, `>=`

**c Pièges et difficultés**

L'instruction pour tester une égalité n'est pas « `if a = b :` » mais « `if a == b :` » . Le symbole « `=` » est réservé aux affectations de valeurs ; l'égalité se teste à l'aide de l'instruction `==`

Si deux conditions sont imbriquées, on n'écrit pas « `else if :` » mais `elif :` .



Dès que les tests à effectuer se compliquent un peu, il faudra veiller à bien gérer les indentations ! Par exemple, on passe par ici, par là, ou là ?

```
1 for a in [True, False]:
2     for b in [True, False]:
```

```

3         if a :
4             print("a : ", a, " b : ", b, " ici")
5         elif b :
6             print("a : ", a, " b : ", b, " là")
7         else :
8             print("a : ", a, " b : ", b, " ou là")
9
10    for a in [True, False]:
11        for b in [True, False]:
12            if a :
13                print("a : ", a, " b : ", b, " ici")
14                if b :
15                    print("a : ", a, " b : ", b, " là")
16            else :
17                print("a : ", a, " b : ", b, " ou là")
18
19    for a in [True, False]:
20        for b in [True, False]:
21            if a :
22                print("a : ", a, " b : ", b, " ici")
23                if b :
24                    print("a : ", a, " b : ", b, " là")
25            else :
26                print("a : ", a, " b : ", b, " ou là")

```

## d Opérations booléennes

**d.1 Opérations booléennes de base** Supposons que  $a$  et  $b$  soient deux variables booléennes. On peut envisager, quelques opérations booléennes de base.

**Et :** `and` La proposition  $a$  et  $b$  ne peut être vraie que si les deux propositions sont vraies en même temps.  
La structure de test en python s'écrira : `if a and b :` .

**Ou :** `or` La proposition  $a$  ou  $b$  est vraie dès que l'une des deux propositions est vraie.  
La structure de test en python s'écrira : `if a or b :` .

**Ou exclusif :** `^` Le ou exclusif entre  $a$  et  $b$  peut se traduire par soit  $a$ , soit  $b$ . La proposition est vraie si une seule des deux conditions soit  $a$ , soit  $b$  est vraie.  
La structure de test en python s'écrira : `if a ^ b :` .

**Négation :** `not` Pour prendre la négation d'une proposition, on utilise, en python, l'instruction `not` .

**d.2 Combinaison d'opérateurs de base** Toute opération logique entre ces opérateurs est alors possible... même si le résultat n'est pas toujours simple à prévoir !

```

1    for a in [True, False]:
2        for b in [True, False]:

```

```
3 | print(a, " ", b, " ", (a ^ (not b)) and (not (a or b)))
```

## 1.5 Retour sur... les boucles

L'utilisation des instructions séquentielles est fondamentale en informatique !

### a Tant que...

#### a.1 Objectif

Répéter certaines instructions tant qu'une condition n'est pas réalisée.

```

valeur_test ← valeur numérique
compteur ← valeur numérique
tant que compteur ≤ valeur_test faire
| ... suite d'instruction à effectuer
| modification de la valeur de compteur
fin

```

Plus généralement, la syntaxe devra obligatoirement obéir à la structure suivante :

```

INITIALISATION de paramètres
tant que condition(s) sur ces paramètres faire
| ... suite d'instruction à effectuer
| MODIFICATION des paramètres
fin

```

Les conditions peuvent être multiples... il faudra prendre soin de s'assurer s'il s'agit d'un *et* ou d'un *ou* entre les différents tests mis en jeu.

**a.2 Implémentation** En python, `tant que` se traduit par `while` (en minuscule).

La structure générale de la boucle *while* est la suivante :

```

1 | initialisations
2 | while condition à vérifier :
3 |     ...
4 |     suite d'instruction à effectuer
5 |     modification de la condition
6 | suite du programme

```

Les premiers exemples d'utilisation de la boucle *while* peuvent se mettre sous la forme suivante :

## Utilisation d'un compteur de boucle

```
i = debut
while i <= fin :
    ...
    ... instructions à exécuter
    ...
    i = i + pas
... suite du programme
```

## Utilisation d'un test

```
flag = True
while flag :
    ...
    ... instructions à exécuter
    ...
    if (condition)
        flag = False
... suite du programme
```

Une utilisation courante de la boucle *while* consiste à remplir un tableau de valeur (afin de tracer une courbe par exemple). Dans l'exemple suivant, on remplit le tableau *T* avec toutes les valeurs réelles comprises entre *debut* et *fin* espacées de *increment*.

```
1 debut = 0
2 fin = 1
3 increment = 0.1
4 compteur = debut
5 T = []
6 while compteur <= fin :
7     T.append(compteur)
8     compteur = compteur + increment
9 print (T)
```

## b

## Pièges et difficultés



Le plus grand piège est de ne pas modifier les paramètres dans la boucle *while*. ... on obtient alors une boucle infinie (ou jamais parcourue) puisque la condition est toujours (ou jamais) vérifiée. Si vous utilisez Spyder, il y a un carré rouge dans la barre d'entête de la console IPython pour mettre fin à l'infini !

Exemple : remplir le tableau *T* avec des nombres réels compris entre 0 et 1 inclus espacés de 0,01.

```
1 T=[]
2 x=0
3 while x <= 1 :
4     T.append(x)
5     x = x+0.01
6 print("T : ", T)
```

Bien sûr, si on initialise mal les paramètres ou si on code mal la condition d'arrêt, le programme ne fera pas ce que l'on souhaite.

## c

## Pour...

**c.1 Objectif** La principale utilité de ce type de boucle est de répéter certaines instructions un certain nombre de fois.

Une autre application intéressante en python est de pouvoir parcourir une liste sans avoir à gérer les indices.

### c.2 Implémentation

En réalité, l'implémentation d'origine de la boucle Pour en python, est justement le parcours



```
pour i variant de début à fin par pas de pas faire  
| ... suite d'instruction à effectuer  
| ... on peut accéder à différentes éléments par l'intermédiaire de la variable i  
fin
```

```
TableauT ← valeurs  
pour x dans T faire  
| ... suite d'instruction à effectuer  
| ... on accède directement à la variable x (on ne dispose pas de son indice dans le tableau)  
fin
```

d'éléments d'une collection.  
L'instruction :

```
for x in T :
```

permet de stocker successivement dans la variable *x* tous les éléments initialement contenus dans le tableau *T*.  
Par exemple le script suivant permet d'imprimer dans la console les différents jours de la semaine.

```
1 semaine = ["dimanche", "lundi", "mardi", "mercredi", "jeudi",  
             "vendredi", "samedi"]  
2 for jour in semaine :  
3     print(jour)
```

Le problème, ici, est que l'on ne peut pas accéder à l'indice du jour dans le tableau semaine.

L'instruction **in range(*debut*, *fin*, *pas*)** permet de générer une liste de valeurs entières : [*debut*, *debut* + *pas*, *debut* + 2*pas*, ... dernière valeur strictement inférieure à *fin*].  
En combinant les deux instructions on va pouvoir récupérer soit une liste d'indices, soit une liste de valeurs.  
La boucle **Pour** se construit alors à partir de l'instruction :

```
for i in range(début, fin (valeur exclue), pas (= 1 par défaut)) :
```



ATTENTION : les variables *début*, *fin* et *pas* sont obligatoirement des ENTIERS.



ATTENTION : La valeur *fin* n'est pas atteinte!  
Si l'incrément (ou le pas) vaut 1, on n'est pas obligé de le spécifier.<sup>1</sup>  
Par exemple :

Instruction	Valeurs successives de <i>i</i>
for <i>i</i> in range(0, 4)	0 ; 1 ; 2 ; 3
for <i>i</i> in range(-10, 10, 2)	-10 ; -8 ; -6 ; -4 ; -2 ; 0 ; 2 ; 4 ; 6 ; 8
for <i>i</i> in range(10, 5, -1)	10 ; 9 ; 8 ; 7 ; 6

1. On rencontre également une instruction du type for *i* in range(5). Dans ce cas, le début vaut forcément 0, on ne précise que la *fin* (non atteinte) et le pas vaut 1. Dans le cas présent, *i* vaut successivement 0, 1, 2, 3, 4.

### c.3 Pièges et difficultés



Attention aux deux erreurs principales dans l'utilisation de `for i in range(...)` :

- la valeur *fin* n'est pas atteinte ;
- les valeurs sont des entiers (cette erreur est facilement détectée lors de l'exécution mais à l'écrit, c'est une autre histoire !)

Comme pour toutes les structures de contrôle, il ne faut pas oublier les « : » à la fin de la ligne ; et bien sûr, comme ailleurs en python, seules les instructions indentées convenablement seront exécutées !

Signalons enfin qu'il ne faut pas modifier la valeur du « compteur de boucle » à l'intérieur de la boucle.

## 1.6 While ou for ?

A priori, la réponse est assez simple :

- on utilise la boucle **for** :
  - pour le parcours des éléments d'une liste : `for i in range(len(liste)) ;`
  - si on connaît facilement le nombre d'itérations
- on utilise la boucle **while** : dans tous les autres cas !

Parfois, les deux solutions sont possibles sans pouvoir privilégier l'une ou l'autre. A titre purement personnel, je privilégie la boucle **while** !

*Si vous avez un peu de mal avec l'écriture des boucles while :*

- commencez par écrire l'initialisation de la boucle ;
- passez à la ligne *while* en codant la condition
- laissez un peu de place et codez la modification de la condition
- terminez par le corps de la boucle

## 2 A vous de jouer...

Comme vous le dites à vos élèves... « jouez le jeu ! » ; je vous donne quelques pistes de résolution et la solution par la suite... essayez de résoudre sans aide !

N'hésitez pas à ajouter des commentaires dans votre code. Pour mettre tous les exercices sur le même fichier python et ne pas être embêté (pour être poli) avec tous les `input` vous pouvez mettre tout un bloc de code en commentaire :

- soit en le sélectionnant et en utilisant (sous spyder) le menu Edit puis Comment/UnComment ;
- soit en mettant 3 guillemets avant et 3 guillemets après le bloc de code correspondant

```

1  a = 1 #et le commentaire pour une seule ligne
2
3  #a = 2
4  #b = 3
5  #print(a+b)
6
7  """
8  a = 3
9  b = 4
10 print(a+b)
11 """
```

```

12 |
13 | b = 2
14 | print(a+b)

```

L'importation de fonctions de la bibliothèque mathématique (cf Annexe 5.2) sera nécessaire pour certains exercices.

## 2.1 Blondes

.1 Écrire à l'aide d'une boucle **for** un programme qui calcule la somme de nombres pairs entre 1 et 100 (inclus) et affiche le résultat.

.2 Écrire un programme qui demande un nombre  $x$ , calcule  $\frac{x^2}{2}$  si  $x < -2$  ou  $x > 4$ ;  $4 - \frac{x^2}{2}$  sinon, puis affiche le résultat.

.3 Écrire un programme qui affiche les nombres réels compris entre 0 et 1 (inclus) espacés de 0,1 :

- a) à l'aide d'une boucle **while**;
- b) à l'aide d'une boucle **for**

## 2.2 Châtain clair...

.1 Écrire à l'aide de deux boucles **for** imbriquées un programme qui calcule la somme :  $S = \sum_{i=1}^n \sum_{j=1}^n ij$ . Que vaut cette somme pour  $n = 10$  ?

.2 Écrire un programme qui calcule la somme des entiers  $k$  positifs tels que  $k + k^2 + k^3 \leq n$ ,  $n$  étant un nombre introduit dans le programme. Que vaut cette somme pour  $n = 1000$  ?

.3 Écrire un programme qui calcule la limite de la suite  $S_n = \sum_{i=1}^n \frac{1}{i^2}$  sachant que l'on arrête l'exécution lorsque  $|S_{n+1} - S_n| < \epsilon$  avec  $\epsilon$  proche de zéro.

Vous vérifierez que la limite de la suite est une bonne approximation de  $\frac{\pi^2}{6}$  et en déduirez une valeur approchée de  $\pi$ .

.4 On introduit un entier positif  $n$ . Écrire un programme qui calcule la somme  $S_n$  des termes suivants :

$$\begin{array}{ccccccc}
 1 \times 1 & + & 1 \times 2 & + & \cdots & + & 1 \times n \\
 & & + & 2 \times 2 & + & \cdots & + & 2 \times n \\
 & & & & & \ddots & & \vdots \\
 & & & & & & \ddots & \vdots \\
 & & & & & & & + & n \times n
 \end{array}$$

Que vaut  $S_8$  ?

.5 Écrire un programme qui demande les nombres  $a$ ,  $b$  et  $c$  dans l'équation  $ax^2 + bx + c = 0$  et qui donne en retour la(les) solution(s) de cette équation (ou affiche un message si elles n'existent pas).

## 2.3 Brunes ou chauves !

L'objectif est d'écrire un programme de **chifoumi** (pierre - feuille - ciseaux). Le cahier des charges est le suivant :

- une partie se joue en 3 manches gagnantes ;
- pour chaque manche, pour éviter de tricher, on fait une proposition ; l'ordinateur (qui ne triche pas non plus) tire au hasard sa proposition, affiche la comparaison des deux propositions ainsi que le résultat ;
- à la fin d'une partie on affiche "Tapez q ou Q pour quitter". On quitte le jeu si l'utilisateur tape *q* ou *Q* et on rejoue sinon.

Pour choisir aléatoirement ce que joue l'ordinateur, on peut utiliser la bibliothèque **random** :

```
1 import random as rd
2 n = rd.randint(0,100) # génère un entier aléatoire entre 0 et
    100 (inclus)
```

## 3 Quelques pistes...

### 3.1 Blondes

.1

- une boucle `for i in range` fera l'affaire (attention à la borne supérieure!);
- pour sommer, on initialise une variable que l'on incrémente!

.2

- la gestion du `if/else` ne devrait pas poser de problème;
- pensez à convertir la chaîne de caractères (obtenue grâce à `input`) en entier!

.3

- de classiques parcours de boucles...;
- que l'on peut rater en oubliant l'incrémentation (boucle `while`) ou en se trompant dans les bornes (boucle `for`).

### 3.2 Châtain clair...

.1

- si l'on sait faire une boucle pour faire une somme... ce n'est pas plus difficile d'en imbriquer deux!
- vérifiez que pour  $n = 10$  la somme vaut 3025.

.2

- on ne connaît pas, *a priori*, la plus grande valeur de  $k$  telle que  $k + k^2 + k^3 \leq n$ ; il faut donc utiliser une boucle `while`!
- il faudra alors gérer la variable  $k$  d'une part, la variable de test  $k + k^2 + k^3$  à la fois dans l'initialisation et dans la boucle `while`.
- vérifiez que pour  $n = 1000$  la somme vaut 45.

.3

- cette fois encore, on ne sait pas immédiatement quand on va devoir s'arrêter donc... on fait une boucle tant que l'incrément est plus grand qu'une limite fixée à l'avance.
- la fonction racine carrée est codée par `sqrt`; il faut l'importer ou importer la bibliothèque `math`:

.4

- la principale difficulté, ici, est la gestion des indices : on a un indice  $l$  pour la ligne qui doit aller de 1 à  $n$  inclus; et un indice  $c$  pour la colonne qui doit aller de  $l$  (inclus) à  $n$  inclus<sup>2</sup>.
- pour  $n = 8$ , je trouve 750.

---

2. J'ai toujours eu du mal avec les indices  $i$  et  $j$  pour savoir qui était la ligne et qui était la colonne; un jour, j'ai eu l'idée d'appeler  $l$  l'indice pour la ligne et  $c$  l'indice pour la colonne... c'est con mais c'est efficace!

### .5 Equation du second degré

- cette fois encore il faut convertir les chaînes de caractère en réel ;
- la fonction racine carrée est codée par `sqrt` ; il faut l'importer ou importer la bibliothèque `math` (c'est la dernière fois que je le rappelle!).

## 3.3

### Brunes ou chauves !

Bon, cette fois, ça se complique un peu. Comme on ne dispose pas encore des listes et des fonctions, il va falloir se débrouiller avec ce que l'on vient de réviser !

*A priori*, 4 problèmes se posent :

- problème de structure de données : va-t-on traiter des chaînes de caractères (ou des caractères) : "P", "F", "C" ou des entiers 0, 1, 2 ? C'est LE problème auquel vous serez très souvent confronté, quelle structure de données allez vous utiliser pour modéliser votre système ? Le choix n'est pas forcément évident !
- sinon il va falloir gérer une manche : comparer les propositions du joueur et de l'ordinateur pour savoir qui va marquer le point (s'il n'y a pas égalité) ;
- puis aussi la partie, il faut 2 points d'avance pour gagner !
- et enfin, savoir si on doit rejouer ou pas.

Pour la gestion de la structure de données, le plus simple est peut-être de demander une réponse du joueur sous forme de nombre entier 0, 1 ou 2 plutôt que sous forme de caractère "P", "F", "C".

Pour le programme proprement dit, il est illusoire, j'espère, d'envisager l'écriture linéaire ! Plusieurs solutions s'offrent à vous, sans qu'il y en ait une de meilleure... pour les débutants, le plus sage serait peut-être :

- d'écrire d'un côté le cœur du programme :
  - on commence par écrire la gestion d'une manche et l'affichage du résultat de la comparaison du choix (si on ne veut toujours avoir à répondre aux input, on peut très bien coder le choix du joueur et tester les différents cas de figure) ;
  - on modifie ce programme pour tenir compte maintenant de la gestion d'une partie.
- d'écrire à part la gestion nouvelle partie
- de rabouter tout ça !

Bon, ce qui est pénible, c'est que l'on va passer son temps à changer les indentations et qu'on risque d'en oublier une. Avec un peu plus d'expérience, le plus simple est peut-être la stratégie « qu'y a-t-il à l'intérieur d'une noix ? »

- on commence par programmer la boucle : tant que l'on ne quitte pas on joue !
- quand on joue, c'est pour commencer une nouvelle partie ; il faut donc initialiser les points
- et tant que la différence entre ces points est plus petite que 2
- il faut faire une manche... et on termine par la programmation d'une manche !

## 4 Solutions !

En tête du fichier j'introduis les bibliothèques nécessaires. Je ne supprime pas la ligne avec utf-8, elle permet d'avoir (avec spyder) des caractères accentués.

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Mon Aug 28 21:28:24 2017
4
5  @author: Alain
6  """
7
8  from math import sqrt
9  import random as rd

```

### 4.1 Blondes

```

.1
1  S = 0
2  for i in range(0,102,2):
3      S = S + i
4  print("Somme des nombres pairs de 1 à 100 : ", S)

```

Bien sûr, si la valeur 100 doit être incluse, il faut aller jusqu'à 102.

```

.2
1  ch = input("Entrez un nombre !")
2  x = float(ch)
3  if x < -2 or x > 4 :
4      y = x*x/2
5  else :
6      y = 4 - x*x/2
7  print("résultat = ", y)

```

On pourrait aussi écrire `x = float(input("Entrez un nombre !"))`.

```

.3
1  print("Avec boucle while")
2  x = 0
3  while x <= 1 :
4      print(x)
5      x = x + 0.1
6
7  print("Avec boucle for")
8  for i in range(0,11):
9      print(i/10)

```

Comme les matheux semblent préférer la boucle `for` à la boucle `while`, vous verrez bon nombre de vos élèves gérer des listes de réels de la sorte !

### 4.2 Châtain clair...

```
.1
1 n = 10
2 S = 0
3 for i in range(1,n+1):
4     for j in range(1, n+1):
5         S = S + i*j
6 print(S)
```



il faut bien écrire `in range(1,n+1)` pour atteindre la valeur `n` ; bon OK je ne vous le redis plus !

```
.2
1 n = 1000
2 S = 0
3 k = 1
4 test = 3
5 while test < n :
6     S = S + k
7     k = k + 1
8     test = k + k**2 + k**3
9 print (S)
```

C'est l'écriture typique d'une boucle `while` sur un test logique, il faut prendre garde à bien initialiser le test avant d'entrer dans la boucle... et de le modifier ensuite.

```
.3
1 epsilon = 1e-12
2 S = 0
3 i = 1
4 increment = 1
5 while increment > epsilon :
6     S = S + increment
7     i = i + 1
8     increment = 1/(i*i)
9 print("Limite = ", S)
10 print("pi = ", sqrt(6*S))
```

La division est évaluée avant la multiplication : si on écrit `1/i*i` il comprend  $(1/i)*i$ . Par contre la puissance est évaluée avant toute autre opération (sauf le moins unitaire) donc si on écrit `1/i**2` ; c'est bon. Bon, si on écrit `1/i^2`, autant dire que l'on a une erreur de syntaxe !

```
.4
1 n = 8
2 S = 0
3 for l in range (1,n+1):
4     for c in range(1, n+1):
5         S = S + l*c
6 print ("Somme = ", S)
```

Il me semble avoir dis que je ne vous disais plus de faire attention aux indices...

```
.5 Equation du second degré
1 cha = input("a = ?")
```



```

2 chb = input("b = ?")
3 chc = input("c = ?")
4 a = float(cha)
5 b = float(chb)
6 c = float(chc)
7 if a == 0 :
8     print("Solution unique : ", -c/b)
9 else :
10     delta = b*b - 4*a*c
11     if delta > 0 :
12         print("Deux racines : ", (-b + sqrt(delta))/(2*a), "
13             et ", (-b - sqrt(delta))/(2*a))
14     elif delta == 0 :
15         print("Solution unique : ", - b/(2*a))
16     else :
17         print("Aucune racine")

```

### 4.3 Chifoumi

Voici les différentes étapes de ma progression :

- on joue ou on joue plus ?

```

1 while onjoue :
2
3     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
4     if ch == "q" or ch == "Q" :
5         onjoue = False
6 print ("Merci")

```

- si on joue, il faut initialiser le nombre de points du joueur et de l'ordinateur et faire une boucle tant que la différence est plus petite que 2.

```

1 while onjoue :
2     pointJoueur = 0
3     pointOrdi = 0
4     while abs(pointJoueur - pointOrdi) < 2 :
5
6         ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
7         if ch == "q" or ch == "Q" :
8             onjoue = False
9 print ("Merci")

```

- tant qu'on y est dans la gestion des points, autant dire qui a gagné!

```

1 while onjoue :
2     pointJoueur = 0
3     pointOrdi = 0
4     while abs(pointJoueur - pointOrdi) < 2 :
5
6
7         if pointJoueur > pointOrdi :
8             print("Bravo, vous avez gagné ", pointJoueur, " à ",
9                 pointOrdi)

```

```

9     else:
10         print("Pas de chance pour vous, j'ai gagné",
                pointOrdi, " à ", pointJoueur)
11     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
12     if ch == "q" or ch == "Q" :
13         onjoue = False
14     print ("Merci")

```

- dans une manche, maintenant il faut demander au joueur de faire un choix (pour me simplifier la tâche j'ai demandé d'écrire un nombre). L'ordinateur tire un nombre au hasard entre 0 et 2.

```

1  while onjoue :
2      pointJoueur = 0
3      pointOrdi = 0
4      while abs(pointJoueur - pointOrdi) < 2 :
5          choixJoueur = int(input("Quel est votre choix : 0 pour
                                   pierre, 1 pour feuille, 2 pour ciseaux ?"))
6          choixOrdi = rd.randint(0,2)
7
8      if pointJoueur > pointOrdi :
9          print("Bravo, vous avez gagné ", pointJoueur, " à ",
                pointOrdi)
10     else:
11         print("Pas de chance pour vous, j'ai gagné",
                pointOrdi, " à ", pointJoueur)
12     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
13     if ch == "q" or ch == "Q" :
14         onjoue = False
15     print ("Merci")

```

- il nous reste alors à gérer la comparaison entre les deux choix. Petite ruse de sioux, pour limiter tout un tas de else j'ai fixé un choix arbitraire au début. Au lieu d'un elif contenant tous les cas de figure, on aurait pu les traiter un par un!

```

1  while onjoue :
2      pointJoueur = 0
3      pointOrdi = 0
4      while abs(pointJoueur - pointOrdi) < 2 :
5          choixJoueur = int(input("Quel est votre choix : 0 pour
                                   pierre, 1 pour feuille, 2 pour ciseaux ?"))
6          choixOrdi = rd.randint(0,2)
7          resultat = - 1 # reste à détecter l'égalité ou le
                        joueur vainqueur
8          if choixJoueur == choixOrdi :
9              resultat = 0
10         elif (choixJoueur == 0 and choixOrdi == 2) or
                (choixJoueur == 1 and choixOrdi == 0) or
                (choixJoueur == 2 and choixOrdi == 1) :
11             resultat = 1
12         if resultat == 0 :
13             print("J'ai joué ", choixOrdi, " - égalité -
                    Joueur : ", pointJoueur, " ordinateur : ",
                    pointOrdi)

```

```
14         elif resultat == 1 :
15             pointJoueur = pointJoueur + 1
16             print("J'ai joué ", choixOrdi, " - vous avez gagné
              - Joueur : ", pointJoueur, " ordinateur : ",
              pointOrdi)
17         else :
18             pointOrdi = pointOrdi + 1
19             print("J'ai joué ", choixOrdi, " - j'ai gagné -
              Joueur : ", pointJoueur, " ordinateur : ",
              pointOrdi)
20     if pointJoueur > pointOrdi :
21         print("Bravo, vous avez gagné ", pointJoueur, " à ",
              pointOrdi)
22     else:
23         print("Pas de chance pour vous, j'ai gagné",
              pointOrdi, " à ", pointJoueur)
24     ch = input("Tapez q ou Q pour quitter, sinon on rejoue !")
25     if ch == "q" or ch == "Q" :
26         onjoue = False
27     print ("Merci")
```

Et en principe, ça marche !

## 5

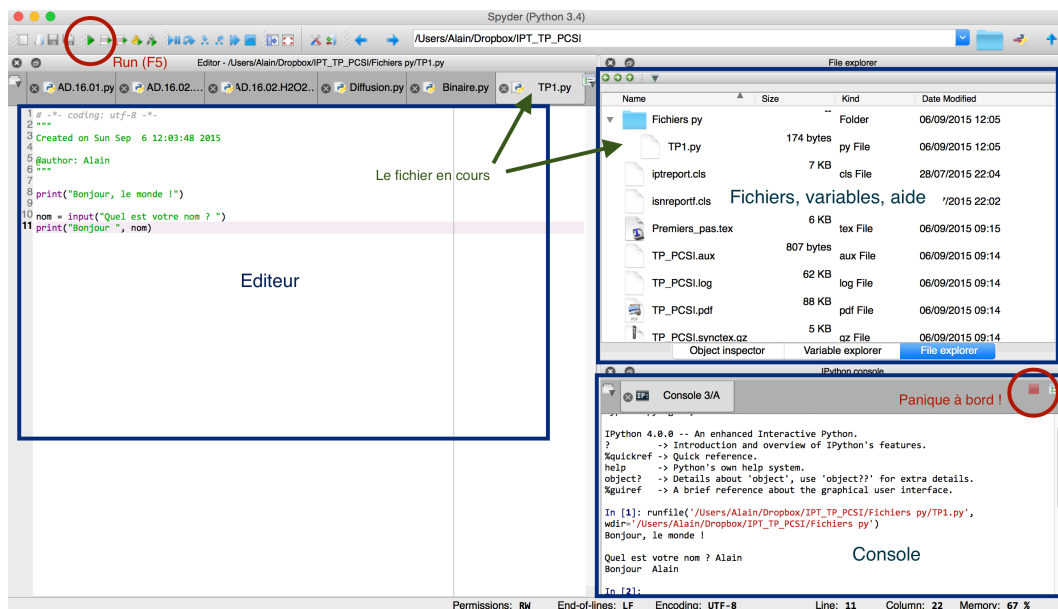
## Annexes

## 5.1 Environnement de développement Synder

Le langage python est un langage simple, puissant et gratuit. Il en existe de nombreuses distributions, plus ou moins conviviales et plus ou moins « volumineuses ». Moi, j'aime bien l'environnement *Spyder*. Vous pouvez en trouver une version pour tout type d'ordinateur à l'adresse : <http://http://continuum.io/downloads> (il suffit de taper Anaconda python sous Google). Vous choisissez votre système d'exploitation et, surtout, vous choisissez une version 3.x

Pour celles et ceux qui ont des machines peu performantes, vous pouvez utiliser un environnement plus « léger » comme la distribution pyzo ([www.pyzo.org/downloads.html](http://www.pyzo.org/downloads.html)) ou Edupython (<http://edupython.tuxfamily.org/>).

Si vous avez téléchargé Spyder, vous vous retrouvez avec l'écran suivant ; les autres distributions se présentent de façon similaire.



Que doit-on repérer en priorité ?

- Une zone **Editeur** qui sert... à éditer le programme dont vous souhaitez l'exécution ; ce sera votre terrain de jeu !
- Une zone **Console** dans laquelle sera affichés les résultats de vos calculs, les graphes, très souvent... les messages d'erreur ! C'est également là qu'il faudra introduire éventuellement certaines données demandées par le programme. C'est donc une zone d'interface entre le programme et l'utilisateur<sup>3</sup>. On peut, si on veut se servir de la console comme d'une calculatrice ou effectuer quelques tests simples de programmation.
- Le bouton **Run** (équivalent clavier F5) vous permet de lancer l'exécution du programme (parfois, des problèmes de connexion à la console interviennent et un message d'erreur du type « Open an IPython console » apparaît, il faut alors ouvrir une telle console à partir du menu Console).
- Parfois votre programme va se bloquer (suite à une erreur de programmation !) ; on peut arrêter son exécution à l'aide du bouton « carré rouge » dans la console.

3. Parfois, aucune console ne se trouve associée au programme que l'on souhaite exécuter. Le logiciel va vous demander d'ouvrir une console ; un menu *Console* est prévu à cet effet, choisir une « IPython Console »

- Une fenêtre en haut à droite avec différents onglets qui permet d'accéder, entre autre : à l'arborescence des fichiers sur votre disque dur, aux valeurs des différentes variables que vous allez utiliser, à l'aide en ligne sur certaines fonctions. . .

## 5.2 Importation de bibliothèques

### a Opérations arithmétiques de base

Un certain nombre d'opération de base sont disponible. Elles sont représentées, sans surprise, par les symboles :  $+$ ,  $-$ ,  $*$ ,  $/$ . On peut également utiliser des parenthèses.

Ajoutons **\*\*** pour une puissance ( $a**b$  revient à  $a^b$ ) ou encore **%** pour un modulo, **abs** pour la valeur absolue.

Attention aux ordres de priorité des opérateurs. Exécuter le programme suivant et, s'il ne fait pas ce que vous souhaitez, ajoutez des parenthèses !

```
1 a = 3
2 b = 2
3 print(a + b/a - b)
4 print(a/b*b)
5 print(a/b**2)
```

Mais, la plupart des opérations usuelles (fonctions trigonométriques,  $\pi$ , logarithme ou exponentielle. . .) ne sont pas disponibles et il faudra quasiment systématiquement importer la bibliothèque mathématique.

### b Utilisation de bibliothèques

On souhaite maintenant calculer la racine carrée ou le sinus d'un nombre. Le langage python ne dispose pas, de base, de la fonction racine carrée. Pour pouvoir en bénéficier, il faut utiliser une bibliothèque de fonctions (que l'on appelle aussi module). Dans notre cas, nous allons importer la bibliothèque **math**. Pour cela, il faut écrire, en tête du programme, une des lignes suivantes :

- « `from math import *` » ; on importe alors la totalité de la bibliothèque mathématique ;
- « `from math import sqrt, sin, pi` » ; on importe alors uniquement les deux fonctions souhaitées ;
- « `import math as m` » et l'on écrira alors<sup>4</sup> `m.sqrt(3)` ou `m.sin(pi/4)`

Voici quelques fonctions de la bibliothèque math :

---

4. Cette méthode peut sembler un peu lourde de prime abord. Elle permet toutefois d'éviter des conflits si une même fonction est définie dans plusieurs bibliothèques. Elle permet également de disposer d'une aide contextuelle lors de la frappe ; il suffit de taper `m.` et ensuite un menu déroulant apparaît avec toutes les fonctions disponibles

<code>sqrt()</code>	Racine carrée d'un nombre
<code>pow(x ,y)</code> ou <code>x**y</code>	Calcule x à la puissance y
<code>sin()</code>	Sinus d'un angle (en radian)
<code>cos()</code>	Cosinus d'un angle (en radian)
<code>tan()</code>	Tangente d'un angle (toujours en radian)
<code>pi</code>	Une valeur approchée précise du nombre $\pi$
<code>log</code>	Logarithme népérien
<code>log10</code>	Logarithme décimal
<code>exp</code>	Exponentielle