J'adore quand un plan se déroule sans accroc !

Hannibal Smith

Music non stop

Kraftwerk

1

Les différentes structures algorithmiques

Extrait du programme

Thème : Algorithmique

Thème : Langage et programmation

Contenus :

Constructions élémentaires

Capacités attendus :

Mettre en évidence un corpus de constructions élémentaires.

Commentaires :

Séquences, affectation, conditionnelles, boucles bornées, boucles non bornées, appels de fonction.

Vous pouvez retrouver le cours complet à l'adresse suivante :
`https://github.com/NaturelEtChaud/NSI-Premiere/tree/main/01LesDifferentesStructuresAlgorithmiques`

Avertissement pour mon petit scarabée [1] :
Donald Knuth, qui ne manque pas d'humour, a écrit un article, publié en avril 1984, dans lequel il étudie les chansons des algorithmes à part entière : *The Complexity of Songs*.
Aussi, ai-je décidé dans ce chapitre de faire référence le plus possible à des chansons.
Si l'article de Donald Knuth vous intéresse, vous pouvez lire le résumé ici :
`https://en.m.wikipedia.org/wiki/The_Complexity_of_Songs`
ou lire l'article en entier là
`https://dl.acm.org/doi/10.1145/358027.358042`.

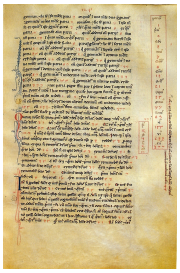
1. Si tu ne l'as pas encore compris, mon *petit scarabée*, c'est toi, ami lecteur.

Les exposés en lien avec ce cours

Muhammad Ibn Musa al-Khuwarizmi (dans les années 780 - vers 850)

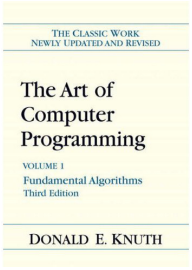
Première page du *Kitab al-mukhtasar fi hisab al-jabr wa-l-muqabala*

Leonardo Fibonacci (vers 1175 - vers 1250)



un extrait du *Liber Abaci*

Donald Knuth (10 janvier 1938 - ...)



la couverture du *Art Of Computer Programming.*

Guido van Rossum (31 janvier 1956 - ...)

Rebecca Allen (1954 - ...)

une image de synthèse.

Sources images :
`https://fr.wikipedia.org/wiki/Al-Khwârizmî`
`https://fr.wikipedia.org/wiki/Leonardo_Fibonacci`
`https://fr.wikipedia.org/wiki/Donald_Knuth`
`https://fr.wikipedia.org/wiki/Guido_van_Rossum`
`https://en.wikipedia.org/wiki/Rebecca_Allen_(artist)`
`http://www.rebeccaallen.com/` `https://www.futura-sciences.com/tech/dossiers/technologie-cinema-animation-tec page/4/`

I. Mais monsieur, c'est quoi la différence entre un algorithme et un programme ?

En regardant sur Wikipédia [2], on trouve entre autres la définition suivante du mot **algorithme** :

Donald Knuth (1938-) liste, comme prérequis d'un algorithme, cinq propriétés :

- finitude : Un **algorithme** doit toujours se terminer après un nombre fini d'étapes.
- définition précise : Chaque étape d'un **algorithme** doit être définie précisément, les actions à transposer doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas.
- entrées : quantités qui lui sont données avant qu'un **algorithme** ne commence. Ces entrées sont prises dans un ensemble d'objets spécifié.
- sorties : quantités ayant une relation spécifiée avec les entrées.
- rendement : toutes les opérations que l'**algorithme** doit accomplir doivent être suffisamment basiques pour pouvoir être en principe réalisées dans une durée finie par un homme utilisant un papier et un crayon.

Un algorithme est donc un ensemble d'instructions avec un but précis, écrit dans un langage « naturel », indépendamment de tout langage de programmation.
La plupart du temps, on peut écrire directement un programme dans un langage choisi. Parfois, on cherche à obtenir une grande efficacité et il est bon de réfléchir en amont à la meilleure façon d'y accéder ; c'est dans ce cas que l'on va imaginer le meilleur algorithme.

II. Variables

a) Définition

Une **variable** est un objet pouvant prendre différentes valeurs lors de l'execution d'un algorithme. Une variable est caractérisée par :

- son identificateur (son nom)
- son type
- son contenu

b) Affectation d'une variable

Cette opération consiste à donner un contenu à une variable.

Exemple
En pseudo-code :

```
1 a ← 5 ;
2 moyenne ← 15.2 ;
3 Nom ← "Alan" ;
4 initiale ← 'T' ;
5 fini ← Vrai ;
```

En scratch :

En Python :

```python
a = 5
moyenne = 15.2
Nom = "Alan"
initiale = 'T'
fini = True
```

2. https://fr.wikipedia.org/wiki/Algorithme

c) Les types de bases

Les types élémentaires d'une variable sont

- les **entiers** (**integer** en anglais ou **int** pour les intimes) ;
- les nombres décimaux ou **flottants**[3] (**float** en anglais). La partie entière est séparée de la partie décimale par un point ;
- les **caractères** (**character** en anglais ou **char** pour les intimes). Dans la plupart des langages, un caractère est encadré par des apostrophes ;
- les **chaînes de caractères** (**character string** en anglais ou **String** pour les intimes). Dans la plupart des langages, une chaîne de caractères est encadré par des guillemets ;
- et enfin un type un peu particulier, les **booléens**[4] (**boolean** en anglais). Ces variables ne peuvent avoir que deux états, **Vrai** ou **Faux**, `True` ou `False` en Python.

Remarque
Dans la plupart des langages de programmation, on doit préciser le type d'une variable au moment de sa création. Par exemple en Java, on écrirait ainsi :

```
int a;
a = 5;
```

ou plus rapidement

```
int a = 5;
```

En Python, au moment de l'affectation, comme le contenu 5 est un entier, la variable sera automatiquement de type entier. C'est ce qu'on appelle le **typage dynamique**.
A ce propos, il faut faire particulièrement attention car ainsi, on peut modifier le type d'une variable dans un même programme (ce qui est évidemment à éviter !).

```
a = 5
# a est de type entier
a = "bonjour"
# a devient de type chaîne de caractères
```

Remarque
Une remarque dans la remarque. Dans le dernier code écrit en Python, on a utilisé le symbole dièse `#` qui signale simplement que tout ce qui suit sur la ligne est un commentaire. L'interpréteur Python n'essayera donc pas de le lire.

Exercice 1
Remplir le tableau suivant concernant les variables de l'exemple :

Identificateur	Contenu	Type
a		
moyenne		
Nom		
initiale		
fini		

Exercice 2 (Python)
La fonction `type()` permet de déterminer le type d'une variable. Par exemple, en tapant `type(a)` dans la console, on obtient la réponse `int`.
Tester le type des 4 autres variables. Que remarquez-vous ?

. . . . . .
. . . . . .
. . . . . .

3.
–Mais Monsieur ! C'est vraiment bizarre comme nom !
– Oui, je sais petit scarabée. Nous verrons ça plus tard, dans le détail dans un prochain chapitre. Il faut suivre le plan !
4.
–Mais Monsieur ! C'est vraiment bizarre comme ...
– Oui, oui, je sais petit scarabée. Nous verrons ça plus tard, dans le détail dans un prochain chapitre. Il faut suivre le ...
– Plan !
– Très bien.

d) Opérations de bases

Les opérations de bases s'effectuent en général sur des variables de même type.

Opération	Symbole en Python	Types des variables	Type du résultat
Addition	+	entiers	entier
Addition	+	flottants	flottant
Soustraction	-	entiers	entier
Soustraction	-	flottants	flottant
Puissance	**	entiers ou flottants	entier ou flottant
Division décimale	/	entiers ou flottants	flottant
Division entière	//	entiers	entier
Reste de la division entière	%	entiers	entier
Concaténation	+	chaînes de caractères	chaîne de caractères

Exercice 3 (Python)

Essayer de deviner le résultat des opérations suivantes écrites en Python.

```
1  a = 11
2  b = 2
3  c = 2.0
4  prenom = "Ada"
5  nom = "Lovelace"
6
7  d = a + b
8  e = a + c
9  f = a * b
10 g = a * c
11 h = a ** b
12 i = a / b
13 j = a // b
14 k = a % b
15 l = prenom + nom
16 m = b * prenom
17 n = c * prenom
18 o = prenom / b
19 p = prenom // b
```

Identificateur	Contenu	Type
d		
e		
f		
g		
h		
i		
j		
k		
l		
m		
n		
o		
p		

Exercice 4

1. Quelles seront les valeurs des variables `a` et `b` après exécution des affectations suivantes ?

```
1 a ← 2 ;
2 b ← a − 3 ;
3 a ← 3 ;
```

`a` .....

`b` .....

2. Quelles seront les valeurs des variables `a`, `b` et `c` après exécution des affectations suivantes ?

```
1 a ← 15 ;
2 b ← −8 ;
3 c ← a + b ;
4 a ← 6 ;
5 c ← b − a ;
```

`a` .....

`b` .....

`c` .....

3. Quelles seront les valeurs des variables `a` et `b` après exécution des affectations suivantes ?

```
1 a ← 5 ;
2 b ← 2 ;
3 a ← b ;
4 b ← a ;
```

`a` .....

`b` .....

Exercice 5
Écrire un algorithme qui permet d'échanger les valeurs des variables `a` et `b`.
Autrement dit, `a` vaut 5 et `b` vaut 2. A la fin de l'algorithme ce sera le contraire.

```
1 a ← 5 ;
2 b ← 2 ;
3 ..... ;
4 ..... ;
5 ..... ;
```

III. Structures conditionnelles

Jusqu'à présent les instructions que vous avez étudiées sont toutes exécutées, l'une après l'autre. Nous allons voir maintenant comment exécutés des instructions uniquement si les conditions choisies par la codeuse ou le codeur sont respectées.

a) Structure conditionnelle simple

Exemple
En pseudo-code :

```
1 si condition alors
2 |  traitement1 ;
3 |  traitement2 ;
4 |  traitement3 ;
5 |  ...
6 fin si
```

En scratch :

En Python :

```
1 if condition :
2     traitement1
3     traitement2
4     traitement3
5     ...
```

Remarque
En Java facilement lisible :

```
1 if (condition){
2     traitement1;
3     traitement2;
4     traitement3;
5     ...
6     }
```

En Java peu lisible mais parfaitement exécutable :

```
1 if (condition){traitement1;
      traitement2; traitement3; ...}
```

Dans la plupart des langages (C++, Java, JavaScript, ...) l'écriture du code n'est structurée que pour faciliter sa lecture pour la codeuse ou le codeur. Par exemple, la fin d'une phrase se termine par un point virgule, un bloc est encadré par des accolades { et }. On pourrait si on le souhaite (mais c'est très vivement déconseillé) écrire tout le code sur une seule ligne. En **Python**, c'est tout à fait différent :

- à chaque fois que l'on revient à la ligne c'est que l'on est sur une phrase différente. Ainsi `traitement1` et `traitement2` sont deux phrases différentes ;
- un bloc commence toujours par une instruction suivi de deux points (ligne 1) et ensuite l'ensemble des instructions dans le bloc sont toutes indentés avec le même espace [5].

5. Attention, il ne faut jamais créer cet espace en tapant plusieurs fois la touche espace. Vous devez utiliser uniquement la touche **Tabulation** (**Tab** pour les intimes). D'ailleurs sur votre éditeur, après les deux points, si vous appuyez sur la touche **Entrée** (ou **Return** en anglais), vous irez à la ligne avec automatiquement la bonne indentation.

b) Mais Monsieur ! Qu'est-ce qu'une condition ?

Une condition est une expression qui est soit vraie, soit fausse. On appelle cela une expression **booléenne**[6]. Le plus souvent cette condition se présente sous la forme d'une comparaison. Voici les différents symboles de comparaison les plus utilisées en Python.

==	est égal à
!=	n'est pas égal à
<	est strictement inférieur à
<=	est inférieur ou égal à
>	est strictement supérieur à
>=	est supérieur ou égal à

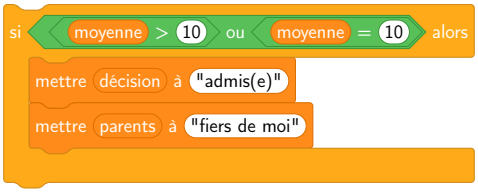
Voici un exemple, concernant votre prochaine moyenne au bac :

Exemple

En pseudo-code :

```
1 si moyenne ≥ 10 alors
2 |   décision ← "admis(e)" ;
3 |   parents ← "fiers de moi" ;
4 fin si
```

En scratch :



En Python :

```
1 if moyenne >= 10 :
2     decision = "admis(e)"
3     parents = "fiers de
      moi"
```

Comme vous pouvez le constater, l'opérateur `>=` n'existe pas dans le langage Scratch. On a donc dû utiliser deux conditions `moyenne > 10` et `moyenne == 10` ainsi que l'opérateur logique `ou`.
Les opérateurs logiques de bases sont :

Opérateur	en Python	
et	and	est vrai si les deux conditions testées sont vraies
ou	or	est vrai si l'une au moins des deux conditions testées est vraie
non	not	est vrai si la condition testé est fausse

On pourrait réécrire notre programme Python ainsi[7] :

```
1 if not (moyenne < 10) or (moyenne == 10) :
2     decision = "admis(e)"
3     parents = "fiers de moi"
```

6. – *Oui, oui, j'ai compris. Il faut suivre le plan.*
– *Exactement mon petit scarabée, nous reviendrons sur ce mot un peu bizarre « **booléen** ». Tu as raison de vouloir respecter le plan.*

7. Mais c'est très fortement déconseillé !

c) Structure conditionnelle complète

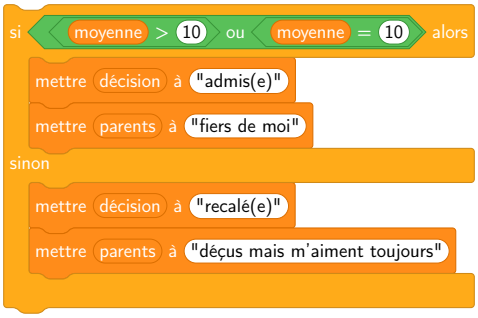
Dans notre exemple précédent nous avons traité le cas où la moyenne de l'élève était bien supérieure à 10. Mais que se passe-t-il si elle est inférieure ? Nous pourrions proposer l'algorithme ci-contre.
On peut remarque néanmoins que le test de la **ligne 5** n'est pas nécessaire puisque cela correspond exactement au contraire du test de la **ligne 1**.
Nous pouvons simplifier notre algorithme avec le mot-clef **sinon**.

```
1 si moyenne ≥ 10 alors
2 |  décision ← "admis(e)" ;
3 |  parents ← "fiers de moi" ;
4 fin si
5 si moyenne < 10 alors
6 |  décision ← "recalé(e)" ;
7 |  parents ← "déçus mais m'aiment toujours" ;
8 fin si
```

Exemple
En pseudo-code :

```
1 si moyenne ≥ 10 alors
2 |  décision ← "admis(e)" ;
3 |  parents ← "fiers de moi" ;
4 sinon
5 |  décision ← "recalé(e)" ;
6 |  parents ← "déçus mais
  |   m'aiment toujours" ;
7 fin si
```

En scratch :



En Python :

```
1 if moyenne >= 10 :
2     decision = "admis(e)"
3     parents = "fiers de
      moi"
4 else :
5     decision = "recalé(e)"
6     parents = "déçus mais
      m'aiment toujours"
```

Remarque
On notera qu'en Python, c'est le mot-clef `else :` accompagné de ses deux petits points.
Il faudra bien faire attention à respecter l'indentation pour bien avoir visuellement les deux blocs.

Exercice 6 (Python)
On dispose de deux variables `nb` qui contient un nombre entier et `reponse` qui contient un booléen. Votre programme devra affecter la valeur `True` si `nb` est pair et `False` sinon à la variable `reponse`.

```
1 nb = 5 #tester d'autres valeurs de nb
2 ....................
3 ....................
4 ....................
5 ....................
```

Et si on le faisait avec une seule ligne ? ? ?
(pas forcément conseillé)

```
1 nb = 5 #tester d'autres valeurs de nb
2 ....................
```

Exercice 7 (Python)
On dispose de deux variables `a` et `b` qui contiennent chacune un nombre entier. Votre programme devra créer la variable `plusgrand` et lui affecter la plus grande valeur entre celle de `a` et celle de `b`.

```
1 a = 5 #tester d'autres valeurs
2 b = 3 #tester d'autres valeurs
3 ....................
4 ....................
5 ....................
6 ....................
```

d) Structure conditionnelle généralisée

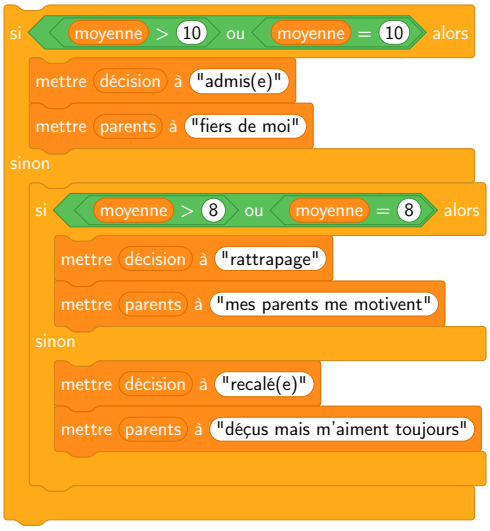
Le monde n'est pas si binaire et souvent il y a plus de 2 cas à tester. Avec un algorithme, Scratch ou certains langages on devra encahîner les structures conditionnelles les unes dans les autres. Avec Python, on pourra enchaîner les tests avec le mot-clef `elif` [8].

Exemple

En pseudo-code :

```
1  si moyenne ≥ 10 alors
2  |  décision ← "admis(e)" ;
3  |  parents ← "fiers de moi" ;
4  sinon
5  |  si moyenne ≥ 8 alors
6  |  |  décision ← "rattrapage" ;
7  |  |  parents ← "mes parents
   |  |   me motivent" ;
8  |  sinon
9  |  |  décision ← "recalé(e)" ;
10 |  |  parents ← "déçus mais
   |  |   m'aiment toujours" ;
11 |  fin si
12 fin si
```

En scratch :



En Python :

```
1 if moyenne >= 10 :
2     decision = "admis(e)"
3     parents = "fiers de
      moi"
4 elif moyenne >= 8 :
5     decision = "rattrapage
      "
6     parents = "mes parents
       me motivent"
7 else :
8     decision = "recalé(e)"
9     parents = "déçus mais
      m'aiment toujours"
```

Exercice 8

Pourquoi pour la deuxième condition (ligne 5 du pseudo-code ou ligne 4 en Python), on ne vérifie pas que la moyenne est inférieure strictement à 10 ?

.....

.....

Exercice 9 (Python)

On dispose de la variable `moyenne` qui contient un nombre flottant. Votre programme devra créer la variable `decision` et lui affecter la valeur `"recalé(e)"` pour une moyenne inférieure strictement à 8, `"rattrapage"` pour une moyenne supérieure à 8 et strictement inférieure à 10, `"admis(e)"` pour une moyenne supérieure à 10 et strictement inférieure à 12, `"mention assez bien"` pour une moyenne supérieure à 12 et strictement inférieure à 14, `"mention bien"` pour une moyenne supérieure à 14 et strictement inférieure à 16, et enfin, `"mention très bien"` pour une moyenne supérieure à 16.

```
1  moyenne = 15.3 #tester d'autres valeurs
2  ....................
3  ....................
4  ....................
5  ....................
6  ....................
7  ....................
8  ....................
9  ....................
10 ....................
11 ....................
12 ....................
13 ....................
```

8. C'est la contraction de **else if** ; c'est ce que nous écririons en Java.

Exercice 10 (Python (*))

On dispose de la variable `mois` qui contient un nombre entier entre 1 et 12 représentant le mois de l'année (par exemple 5 pour le mois de mai).
Votre programme devra créer la variable `saison` et lui affecter la valeur `"printemps"`, `"été"`, `"automne"` et `"hiver"` correspondant à la saison.

```
mois = 5 #tester d'autres valeurs
.....
.....
.....
.....
.....
.....
.....
.....
```

Exercice 11 (Python (**))

On dispose de la variable `annee` qui contient un nombre entier représentant l'année (par exemple 2021 pour cette année).
Votre programme devra créer la variable booléenne `bissextile` et lui affecter la valeur `True` ou `False` pour savoir si on pourra fêter l'anniversaire de Superman [9].

```
annee = 2021 #tester d'autres valeurs
.....
.....
.....
.....
```

9. https://blog.francetvinfo.fr/case-a-part/2013/04/11/les-10-secrets-de-superman.html

IV. La structure répétitive POUR

a) La boucle POUR

Vous connaissez tous la chanson :

1 km à pieds ça use, ça use
1 km à pieds ça use les souliers
2 km à pieds ça use, ça use
2 km à pieds ça use les souliers
3 km à pieds ça use, ça use
...

Si on doit parcourir un distance de 10 km, il suffit de continuer ainsi jusqu'à :

10 km à pieds ça use, ça use
10 km à pieds ça use les souliers

On peut aussi remarquer, que mise à part le nombre de km parcouru, la phrase est la même. Ainsi pour le i-ème km on chanterait :

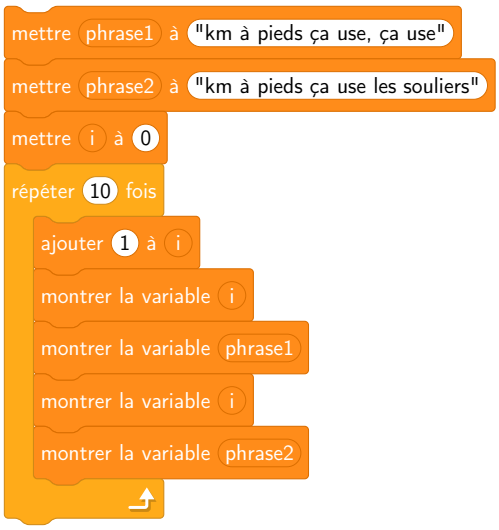
i km à pieds ça use, ça use
i km à pieds ça use les souliers

Il suffirait donc juste de demander à répéter ces 2 phrases **pour** une variable i qui à chaque étape serait modifier en l'**incrémentant** [10] de 1.

Exemple
En pseudo-code :

```
pour i allant de 1 à 10 faire
    Chante : "i km à pieds ça use, ça use" ;
    Chante : "i km à pieds ça use les souliers" ;
fin pour
```

En scratch :



En Python :

```
for i in range(1,11):
    sing(str(i) + " km à pieds ça use, ça use")
    sing(str(i) + " km à pieds ça use les souliers")
```

10. Autrement dit en lui rajoutant le nombre 1 à chaque étape.

Remarque
On constate que l'écriture de ce petit programme est bien compliqué en Scratch, mais c'est bien le langage Python qui nous intéresse. Voici quelques remarques autour du programme proposé :

- La fonction `range(a,b)` (ligne 1) permet d'incrémenter de 1 en 1 en partant du nombre `a` et en s'arrêtant juste avant le nombre `b`. C'est une petite bizarrerie de Python. Autrement dit `range(1,11)` permet d'incrémenter `i` de 1 inclus à 11 exclu, au encore de 1 à 10.
- La fonction `sing()` (lignes 2 et 3) n'existe pas, mais on pourrait l'inventer. Ce pourrait d'ailleurs être l'objet d'un projet de fin d'année.
- En supposant que la fonction `sing()` chante une chaîne de caractère il faut pouvoir écrire `"i km à pieds ça use, ça use"`, or la variable `i` est du type entier et on ne peut pas additionner ou concaténer un entier avec une chaîne de caractères.
  La fonction `str()` va donc convertir l'entier contenu dans `i` en une chaîne de caractères avec la valeur de `i`. Ainsi par exemple `str(10)` renvoie `"10"`.
  Ensuite on utilise la concaténation pour coller le nombre à la phrase. Ainsi `str(i) + " km à pieds ça use, ça use"` crée bien la chaîne de caractères `"10 km à pieds ça use, ça use"` si c'est l'entier 10 qui est contenu dans la variable `i`.

Exercice 12 (Python)
Écrire un programme qui affiche la table de multiplication par 3. On utilisera la fonction `print("un joli message")` qui affiche les chaînes de caractères.

```
1 ..........................................
2 ..........................................
```

Exercice 13 (Python (*))
Écrire un programme qui affiche tous les nombres multiple de 13 compris entre 0 et 666. On utilisera à nouveau la fonction `print(i)` qui affiche aussi le contenu d'une variable.

```
1 ....................
2 ....................
3 ....................
```

Exercice 14 (Python (**))
Écrire un programme qui affiche tous les nombres cubiques de 3 chiffres. Un nombre est dit cubique s'il est égal à la somme des cubes de ses chiffres.
Par exemple, 153 est un nombre cubique de 3 chiffres car :

- il est composé de 3 chiffres
- $153 = 1^3 + 5^3 + 3^3$

On pourra utiliser les variables suivantes `c` le chiffre des centaines, `d` le chiffre des dizaines et `u` le chiffre des unités.

```
1 ....................
2 ....................
3 ....................
4 ....................
5 ....................
6 ....................
```

b) Quelques détails autour de la fonction `range()`

La fonction `range()` accepte 1, 2 ou 3 arguments.

- `range(a)` permet d'incrémenter de 1 en 1, en partant de 0 inclus jusqu'à `a` exclu.
  Autrement dit `range(10)` permet d'obtenir les incréments 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
  Nous avons bien une liste de 10 nombres ce qui nous donne 10 répétitions dans la boucle `for`.
- `range(a,b)` permet d'incrémenter de 1 en 1, en partant de `a` inclus jusqu'à `b` exclu.
  Autrement dit `range(2,10)` permet d'obtenir les incréments 2, 3, 4, 5, 6, 7, 8, 9.
- `range(a,b,c)` permet d'incrémenter de `c` en `c`, en partant de `a` inclus jusqu'à `b` exclu.
  Autrement dit `range(2,10,3)` permet d'obtenir les incréments 2, 5, 8.

Exercice 15 (Fonction `range()`)
Quelles sont les listes de nombres obtenues dans chacun des cas suivants :

1. `range(5)` : .....
2. `range(4,10)` : .....
3. `range(10,4)` : .....
4. `range(4,10,2)` : .....
5. `range(10,4,-1)` : .....

Exercice 16 (The Residents - Commercial Album)
En 1980, le groupe d'avant-garde américain `The Residents` part du constat qu'une chanson pop est composée uniquement de deux parties : un couplet et un refrain. Ces deux parties d'une durée approximative d'une minute sont répétées trois fois ce qui permet d'obtenir une chanson d'une durée total de 3 minutes. En supprimant les répétitions, une chanson pop ne dure donc que 1 minute.
`The Residents` propose alors dans son album `Commercial Album`[11] 40 chansons d'une minute chacune. Pour en faire des chansons pop, il « suffit » de les répéter trois fois chacune.

Source Image :
`https://en.wikipedia.org/wiki/Commercial_Album`

1. On suppose que l'on a deux variables `couplet` et `refrain` de type chaîne de caractères et une fonction `sing()` qui chante le texte écrit dans une chaîne de caractères.
Écrire un programme qui permet d'obtenir une chanson pop selon le constat de `The Residents`.

```
1 .....
2 .....
3 .....
```

2. La structure proposée par `The Residents` est plutôt simpliste[12]. Une chanson pop a plutôt la structure suivante : `intro`, `couplet`, `refrain`, `couplet`, `refrain`, `pont`, `refrain`, `couplet`, `refrain`, `refrain`[13].
Compléter le programme ci-contre :

```
1 sing(intro)
2 for i in range(4) :
3 .....
4 .....
5 .....
6 .....
7 .....
8 sing(refrain)
```

Exercice 17 (Retour de l'exercice 13 (*))
Écrire un programme qui affiche tous les nombres multiple de 13 compris entre 0 et 666.

```
1 .....
2 .....
```

11. `https://www.deezer.com/fr/album/84154472`
12. Et c'était le but !
13. Ce qui permet d'obtenir une chanson d'un peu plus de 4 min !!!!

V. La structure répétitive TANT QUE

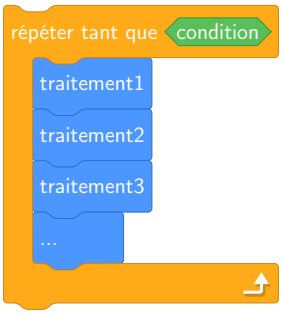
Dans certains cas, dans une boucle répétitive, le nombre d'itération est imprévisible. On doit répéter les instructions **tant que** les conditions voulues sont réunis.

Exemple

En pseudo-code :

```
1 tant que condition faire
2 |   traitement1 ;
3 |   traitement2 ;
4 |   traitement3 ;
5 |   ...
6 fin tq
```

En scratch :



En Python :

```
1 while condition :
2     traitement1
3     traitement2
4     traitement3
5     ...
```

Remarque

Toute boucle **POUR** peut être réécrite avec une boucle **TANT QUE**. Voici deux versions de notre chanson vue au paragraphe précédent :

Version boucle **POUR** :

```
1 for i in range(1,11):
2     sing(str(i) + " km à pieds ça use, ça use")
3     sing(str(i) + " km à pieds ça use les souliers")
```

Version boucle **TANT QUE** :

```
1 i = 1
2 while i<11:
3     sing(str(i) + " km à pieds ça use, ça use")
4     sing(str(i) + " km à pieds ça use les souliers")
5     i = i + 1
```

On constate dans cette dernière version que l'on est obligé d'affecter une première valeur `i = 1` à la variable `i` et surtout que l'on doit incrémenter de 1 cette variable `i = i + 1` dans la boucle.

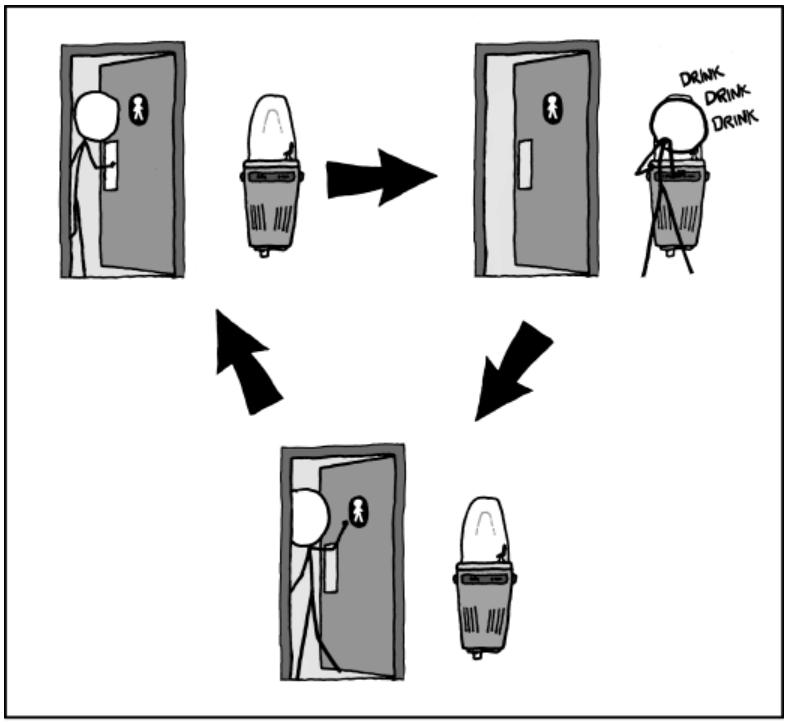
Exercice 18
Expliquer ce qui se passe dans les deux cas suivants :

```
i = 1
while i<11:
    sing(str(i) + " km à pieds ça use, ça use")
    sing(str(i) + " km à pieds ça use les souliers")
```

.....
.....
.....
.....
.....

```
i = 1
while i<11:
    sing(str(i) + " km à pieds ça use, ça use")
    sing(str(i) + " km à pieds ça use les souliers")
i = i + 1
```

.....
.....
.....
.....
.....



Source XKCD[14] *:* `https://xkcd.com/986/`

14. Et si vous alliez voir celui-là ? `https://xkcd.com/537/`

Exercice 19 (Kraftwerk - Music non stop)

`Kraftwerk` est un groupe allemand pionnier de la musique électronique. En 1986, il propose `Music non stop` une chanson minimaliste, les paroles étant simplement une répétition de la phrase `"music non stop"`.
A ce propos la vidéo musicale qui l'accompagne est une des toutes premières effectuée totalement en image de synthèse. On peut la trouver par exemple ici : `https://www.youtube.com/watch?v=O0lIlROWro8`. Elle a été créé par Rebecca Allen, une pionnière du **digital art** [15].

Source image :
`https://en.wikipedia.org/wiki/Musique_Non-Stop`

Écrire le plus simplement possible un programme qui répète à la l'infini `'music non stop'`.

```
1 ......................
2 ......................
```

Exercice 20 (Jeu de dé (**))

On peut importer de nouvelles fonctionnalités qui ne sont pas présente de base dans le langage Python. Il existe pour cela de nombreuses bibliothèques [16] (que l'on nomme **module** [17] en Python).
Par exemple le module `random` contient de nombreuses fonctions pour obtenir des nombres au hasard. Nous allons ici utiliser la fonction `randint(a,b)` qui permet d'obtenir un nombre entier au hasard [18] entre `a` et `b`.
Voici un exemple :

```
1 # on importe la fonction randint présente dans le module random
2 from random import randint
3
4 # on peut utiliser la fonction randint dans tout le reste du programme
5 de = randint(1,6)
```

1. Écrire un programme qui simule le lancer d'un dé, jusqu'à ce qu'on obtienne enfin un 6.
Les résultats de chaque lancer seront affichés avec la fonction `print()`.

```
1 from random import randint
2 ......................
3 ......................
4 ......................
5 ......................
```

2. (*) Modifier le programme précédent pour afficher à la fin le nombre total de lancer nécessaire pour enfin obtenir un 6.

```
1 from random import randint
2 ......................
3 ......................
4 ......................
5 ......................
6 ......................
7 ......................
8 ......................
```

15. `https://en.wikipedia.org/wiki/Rebecca_Allen_(artist)`
16. Attention Attention ! On parle en anglais de *Library* qui est un faux ami. Il s'agit bien de « bibliothèque ». En effet en anglais, « librairie » se dit tout simplement *bookshop* .
17. *– Alors y dit quoi le plan, sur ce sujet ?*
– Je te trouve bien impertinent petit scarabée. Tu devrais aussi revoir la construction de ta phrase. Néanmoins, je te répondrai. Le plan a prévu de parler plus en détail des modules au chapitre n°6.
18. Presque au hasard ! On ne peut pas obtenir de tirage totalement au hasard avec un ordinateur. Voir l'excellent article wikipédia sur les générateurs de nombres pseudo-aléatoires : `https://fr.wikipedia.org/wiki/Générateur_de_nombres_pseudo-aléatoires`.

3. (**) On décide de déterminer combien, en moyenne, il est nécessaire de lancer un dé pour obtenir un 6. On décide d'effectuer 10 fois des séries de lancers de dé jusqu'à obtenir 6, puis de calculer la moyenne des 10 nombres de lancers nécessaire.
Pour une plus grande clarté, nous n'afficherons plus les résultats des lancers du dé.

```
from random import randint
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

VI. Les sous-programmes

Pour résoudre des problèmes complexes, il est préférable de les décomposer en sous problèmes plus simples. On associe alors à chacun de ses problèmes simples un **sous-programme** ou **module**[19]. Il existe deux types de sous-programmes :

- les **procédures**, ce sont des sous-programmes dont le but est d'effectuer une action (afficher une image, commander une action à un robot, ...)
- les **fonctions**, ce sont des sous-programmes dont le but est d'effectuer un calcul et de renvoyer un résultat (sous la forme d'un entier, d'un flottant, d'un booléen, d'un caractère, d'une chaîne de caractères, ...)

La classification des sous-programmes n'est pas aussi binaire car on peut imaginer un sous-programme qui effectue une action et qui renvoie un résultat. De plus, en Python, tout est fonction, car même si on ne le voit pas, même si on ne demande rien, un sous-programme renverra toujours un résultat (qui au pire sera vide !)

– *C'est carrément bizarre votre truc !*

– *Tout d'abord, petit scarabée, ce n'est pas* ***mon*** *truc. Mais je te l'accorde, c'est bizarre mais c'est comme ça.*

Vous avez déjà rencontré trois fonctions : `print()`, `range()` et une fonction qui n'est pas installée de base `randint()`. Vous constaterez que l'on rajoute toujours le couple de parenthèses `()` à la fin de la fonction. Ainsi, on peut faire la différence entre une variable `print` et la fonction `print()`.

– *ATTENTION !!! Petit scarabée ! En Python on peut faire des choses très dangereuses et je t'interdis expressément de donner le même nom à une variable et à une fonction ! De même, on peut très bien nommer une fonction avec un nom déjà existant ! Je t'interdit formellement de le faire !*

– *Oui chef ! Bien chef !*

Nous allons voir ici comment nous pouvons créer notre propre fonction.

Exemple

La fonction ci-contre est une fonction qui permet de vérifier si un nombre est un nombre pair.

Pour définir une fonction on commence par le mot-clef `def` puis le nom choisi par la codeuse ou le codeur. On termine toujours cette première ligne par les deux points `:`. L'ensemble du code correspondant à cette fonction s'affiche alors sous la forme d'un bloc grâce à l'indentation.

Après le nom de la fonction, entre parenthèses, on écrit les arguments nécessaire à l'utilisation de la fonction. Dans notre cas, nous avons besoin d'un nombre `nb`.

Pour terminer notre fonction, si on doit renvoyer un résultat, on le fait avec le mot-clef `return`. Il ne peut plus y avoir d'action après.

```
1  def pair(nb) :
2      '''
3      entrée : est un entier
4      sortie : vrai ou faux si nb est pair
5      '''
6      if nb%2 == 0 :
7          reponse = True
8      else :
9          reponse = False
10     return reponse
```

On remarquera qu'aux lignes 2 à 5 on a inséré un petit commentaires entre un couple de 3 apostrophes `''' ... '''`. Ceci n'est pas obligatoire, mais fortement conseillé. Elle permet de documenter la fonction.

Si on ne se souvient plus comment on utilise une fonction, il suffit d'utiliser la fonction `help()` pour avoir accès à sa documentation.

```
1  >>> help(pair)
2  Help on function pair in module __main__:
3
4  pair(nb)
5      entrée : est un entier
6      sortie : vrai ou faux si nb est pair
```

19. Ici dans une définition plus large que celle de Python.

Une fois la fonction enregistrée et interprétée, on peut l'utiliser dans la console en modifiant simplement le nombre que l'on souhaite tester.

```
>>> pair(666)
True

>>> pair(13)
False
```

Remarque
Pour les adeptes du code cours, nous pouvions aussi définir la fonction `pair()` ainsi :

```
def pair(nb) :
    '''
    entrée : est un entier
    sortie : vrai ou faux si nb est pair
    '''
    return nb%2 == 0 :
```

Exercice 21 (Retour sur l'exercice 12)
Écrire une fonction qui permet d'afficher la table de multiplication choisie.

```
def table_multi(nb) :
    '''
    entrée : nb est un nombre entier
    rôle : affiche la table de
    multiplication de nb de 1 à 10
    '''
.....
.....
```

Exercice 22 (Retour sur l'exercice 13)
Écrire une fonction qui correspond à sa documentation.

```
def multi(nb,a,b) :
    '''
    entrées : nb, a, b sont des nombres
    entiers
    rôle : affiche la liste des multiples
     de nb compris entre a et b (inclus)
    '''
.....
.....
.....
```

Exercice 23 (1 2 3 BOUM ! (*))
La fonction `boum(maxi,div)` affiche la liste des nombres entiers de 1 jusqu'à `maxi` avec une légère modification. Tous les nombres divisibles par le nombre `div` sont à écarter de la liste et sont remplacer par la chaîne de caractères `"BOUM !"`.
Par exemple `boum(14,3)` affichera 1, 2, BOUM !, 4, 5, BOUM !, 7, 8, BOUM !, 10, 11, BOUM !, 13, 14.

```
def boum(maxi,div) :
    '''
    entrées : maxi, div sont des nombres
    entiers
    rôle : affiche la liste modifiée des
    nombres entre 1 et maxi (inclus)
    '''
.....
.....
.....
.....
.....
```

Exercice 24 (1 2 3 BOUM ! (***))

La fonction `boum_plus(maxi,div)` affiche la liste des nombres entiers de 1 jusqu'à `maxi` avec une légère modification. Tous les nombres divisibles par le nombre `div` sont à écarter de la liste, ainsi que les nombres qui contiennent la séquence de chiffres de `div` et sont remplacer par la chaîne de caractères `"BOUM !"`.

Par exemple `boum_plus(14,3)` affichera 1, 2, BOUM !, 4, 5, BOUM !, 7, 8, BOUM !, 10, 11, BOUM !, BOUM !, 14.

(le nombre 13 n'est pas affiché car il contient le chiffre 3)

```
def boum_plus(maxi,div) :
    '''
    entrées : maxi, div sont des nombres entiers
    rôle : affiche la liste modifiée des nombres entre 1 et maxi (inclus)
    '''
.....
.....
.....
.....
.....
```

Exercice 25 (Retour sur l'exercice 20 (**))

La théorie probabiliste nous apprend qu'en **moyenne** il faut lancer 6 fois un dé pour enfin obtenir le nombre 6. A l'exercice 20, nous n'avons la moyenne que sur 10 lancers ce qui nous donne des résultats très fluctuants.

Écrire une fonction `moyenne_de(faces,series)` qui retourne sous la forme d'un flottant le nombre moyen de lancers nécessaire pour enfin obtenir le résultat `faces` pour un dé qui aurait un nombre de faces donné par la variable `faces` pour un nombre total de séries de lancers donné par la variable `series`.

Par exemple `moyenne_de(30,1000)` affichera le nombre moyen de lancers pour enfin obtenir 30 avec un dé à 30 faces sur un total de 1000 séries de lancers.

(Comme vous vous en doutez, ce nombre est en théorie égale à 30)

```
from random import randint

def moyenne_de(faces,series) :
    '''
    entrées : faces, series sont des nombres entiers
    sortie : un nombre flottant
    rôle : calcul la moyenne sur series séries de lancers jusqu'à obtenir le nombre faces
    '''
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

VII. Corrections des exercices

b) Variables

Correction de l'exercice 1

Remplir le tableau suivant :

Identificateur	Contenu	Type
a	5	entier
moyenne	15.2	flottant
Nom	"Alan"	chaîne de caractères
initiale	'T'	caractère
fini	Vrai	booléen

Correction de l'exercice 2 (Python)

La fonction `type()` permet de déterminer le type d'une variable. Par exemple, en tapant `type(a)` dans la console, on obtient la réponse `int`.

Tester le type des 4 autres variables. Que remarquez-vous ?

En Python, on ne fait pas la distinction entre caractère et chaîne de caractères. Dans les deux cas ce sont des chaînes de caractères (`str`). D'ailleurs, on peut aussi bien définir une chaîne de caractère entre guillemets ou entre apostrophe.

Correction de l'exercice 3 (Python)

Essayer de deviner le résultat des opérations suivantes écrites en Python.

```
a = 11
b = 2
c = 2.0
prenom = "Ada"
nom = "Lovelace"

d = a + b
e = a + c
f = a * b
g = a * c
h = a ** b
i = a / b
j = a // b
k = a % b
l = prenom + nom
m = b * prenom
n = c * prenom
o = prenom / b
p = prenom // b
```

Identificateur	Contenu	Type
d	13	entier
e	13.0	flottant
f	22	entier
g	22.0	flottant
h	121	entier
i	5.5	flottant
j	5	entier
k	1	entier
l	"AdaLovelace"	chaîne de caractères
m	"AdaAda"	chaîne de caractères
n	`TypeError`	`can't multiply sequence by non-int of type 'float'`
o	`TypeError`	`unsupported operand type(s) for /: 'str' and 'int'`
p	`TypeError`	`unsupported operand type(s) for //: 'str' and 'int'`

Correction de l'exercice 4

1. Quelles seront les valeurs des variables `a` et `b` après exécution des affectations suivantes ?

```
1 a ← 2 ;
2 b ← a − 3 ;
3 a ← 3 ;
```

`a` contient 3
`b` contient -1

2. Quelles seront les valeurs des variables `a`, `b` et `c` après exécution des affectations suivantes ?

```
1 a ← 15 ;
2 b ← −8 ;
3 c ← a + b ;
4 a ← 6 ;
5 c ← b − a ;
```

`a` contient 6
`b` contient -8
`c` contient -14

3. Quelles seront les valeurs des variables `a` et `b` après exécution des affectations suivantes ?

```
1 a ← 5 ;
2 b ← 2 ;
3 a ← b ;
4 b ← a ;
```

`a` contient 2
`b` contient 2

Correction de l'exercice 5

Écrire un algorithme qui permet d'échanger les valeurs des variables `a` et `b`.
Autrement dit, `a` vaut 5 et `b` vaut 2. A la fin de l'algorithme ce sera le contraire.

```
1 a ← 5 ;
2 b ← 2 ;
3 temp ← a ;
4 a ← b ;
5 b ← temp ;
```

Il est à noter que le langage Python permet d'écrire cet échange rapidement :

```
1 a = 5
2 b = 2
3 a, b = b, a
```

C'est ce qu'on appelle une **pythonnerie**, une spécificité du langage Python. Enfin pour ceux que ça intéresse, Python utilise pour faire cet échange la notion de tuple [20].

c) Structures conditionnelles

Correction de l'exercice 6 (Python)

On dispose de deux variables `nb` qui contient un nombre entier et `reponse` qui contient un booléen. Votre programme devra affecter la valeur `True` si `nb` est pair et `False` sinon.

```
1 nb = 5 #tester d'autres valeurs de nb
2 if nb%2 == 0 :
3     reponse = True
4 else :
5     reponse = False
```

Quelques commentaires :

- `nb%2` signifie « le reste de la division par 2 ». Si ce reste est bien égal à 0 c'est que la division par 2 tombe juste. Autrement dit, `nb` est pair ;
- bien faire attention à l'utilisation du symbole `==` ligne 3 et du symbole `=` ligne 4 et 6 qui ont une signification totalement différente.

20. *–Mais Monsieur ! C'est vraiment bizarre comme truc ! Vous dites comment ? Un tuple ?*
– Oui, je sais petit scarabée. Nous verrons ça plus tard, dans le détail au chapitre n°7. Il FAUT suivre le plan !

Correction de l'exercice 7 (Python)

On dispose de deux variables `a` et `b` qui contiennent chacune un nombre entier. Votre programme devra créer la variable `plusgrand` et lui affecter la plus grande valeur entre celle de `a` et celle de `b`.

```
1 a = 5 #tester d'autres valeurs
2 b = 3 #tester d'autres valeurs
3 if a>b :
4     plusgrand = a
5 else :
6     plusgrand = b
```

Correction de l'exercice 8

Pourquoi pour la deuxième condition (ligne 5 du pseudo-code ou ligne 4 en Python), on ne vérifie pas que la moyenne est inférieure strictement à 10 ?

Cette ligne 5 (dans le pseudo-code) ou 4 (dans le code en Python) n'est lue que si la première condition `moyenne >= 10` n'est pas respectée. Autrement dit la moyenne est avec certitude inférieure strictement à 10.

Correction de l'exercice 9 (Python)

On dispose de la variable `moyenne` qui contient un nombre flottant. Votre programme devra créer la variable `decision` et lui affecter la valeur `"recalé(e)"` pour une moyenne inférieure strictement à 8, `"rattrapage"` pour une moyenne supérieure à 8 et strictement inférieure à 10, `"admis(e)"` pour une moyenne supérieure à 10 et strictement inférieure à 12, `"mention assez bien"` pour une moyenne supérieure à 12 et strictement inférieure à 14, `"mention bien"` pour une moyenne supérieure à 14 et strictement inférieure à 16, et enfin, `"mention très bien"` pour une moyenne supérieure à 16.

```
1  moyenne = 15.3 #tester d'autres valeurs
2  if moyenne < 8:
3      decision = "recalé(e)"
4  elif < 10 :
5      decision = "rattrapage"
6  elif < 12 :
7      decision = "admis(e)"
8  elif < 14 :
9      decision = "mention assez bien"
10 elif < 16 :
11     decision = "mention bien"
12 else :
13     decision = "mention très bien"
```

Correction de l'exercice 10 (Python (*))

On dispose de la variable `mois` qui contient un nombre entier entre 1 et 12 représentant le mois de l'année (par exemple 5 pour le mois de mai).

Votre programme devra créer la variable `saison` et lui affecter la valeur `"printemps"`, `"été"`, `"automne"` et `"hiver"` correspondant à la saison.

```
1 mois = 5 #tester d'autres valeurs
2 if 3<= mois <= 5:
3     saison = "printemps"
4 elif 6<= mois <= 8:
5     saison = "été"
6 elif 9<= mois <= 11:
7     saison = "automne"
8 else :
9     saison = "hiver"
```

Remarque

Dans la plupart des langages la condition de la ligne 2 s'écrirait `(mois >= 3) and (mois <=5)`. Il s'agit à nouveau d'une petite **pythonnerie** bien pratique qui nous permet de contracter ces deux conditions en un encadrement, comme on le fait naturellement en mathématiques.

Correction de l'exercice 11 (Python (**))
On dispose de la variable `annee` qui contient un nombre entier représentant l'année (par exemple 2020 pour cette année). Votre programme devra créer la variable booléenne `bissextile` et lui affecter la valeur `True` ou `False` pour savoir si on pourra fêter l'anniversaire de Superman [21].

```
annee = 2020 #tester d'autres valeurs
if (annee%4 == 0) and ((annee%100 != 0) or (annee%400 ==0)) :
    bissextile = True
else :
    bissextile = False
```

On aurait aussi pu imaginer le code suivant, moins synthétique mais peut-être plus clair :

```
annee = 2020 #tester d'autres valeurs
if annee%4 == 0 :
    if annee%100 != 0 :
        bissextile = True
    elif annee%400 ==0 :
        bissextile = True
    else :
        bissextile = False
```

Remarque
Une année est bissextile si elle un multiple de 4 sauf pour les multiples de 100 sauf pour les multiples de 400...
Un exemple, l'année 2000 est un multiple de 4 donc ce devrait être une année bissextile, mais comme c'est aussi un multiple de 100 ce n'est pas plus année bissextile, mais comme il s'agit tout de même d'un multiple de de 400 l'année redevient un multiple de 400.
Pour plus d'information sur le sujet :
`https://fr.wikipedia.org/wiki/Année_bissextile`

d) La structure répétitive POUR

Correction de l'exercice 12 (Python)
Écrire un programme qui affiche la table de multiplication par 3. On utilisera la fonction `print("un joli message")` qui affiche les chaînes de caractères.

```
for i in range(1,11) :
    print("3 fois "+str(i)+" égale "+str(3*i))
```

Correction de l'exercice 13 (Python (*))
Écrire un programme qui affiche tous les nombres multiple de 13 compris entre 0 et 666. On utilisera à nouveau la fonction `print(i)` qui affiche aussi le contenu d'une variable.

```
for i in range(0,667) :
    if i%13 == 0 :
        print(i)
```

Correction de l'exercice 14 (Python (**))
Écrire un programme qui affiche tous les nombres cubiques de 3 chiffres. Un nombre est dit cubique s'il est égal à la somme des cubes de ses chiffres.

Par exemple, 153 est un nombre cubique de 3 chiffres car :

- il est composé de 3 chiffres
- $153 = 1^3 + 5^3 + 3^3$

On pourra utiliser les variables suivantes `c` le chiffre des cen-

21. `https://blog.francetvinfo.fr/case-a-part/2013/04/11/les-10-secrets-de-superman.html`

taines, `d` le chiffre des dizaines et `u` le chiffre des unités.

```
for i in range(100, 1000):
    c = i//100
    d = i//10 - c*10
    u = i - c*100 - d*10
    if c**3 + d**3 + u**3 == i :
        print(i)
```

Correction de l'exercice 15 (Fonction `range()`)

Quelles sont les listes de nombres obtenues dans chacun des cas suivants :

1. `range(5)` : 0, 1, 2, 3, 4
2. `range(4,10)` : 4, 5, 6, 7, 8, 9
3. `range(10,4)` : rien
4. `range(4,10,2)` : 4, 6, 8
5. `range(10,4,-1)` : 10, 9, 8, 7, 6, 5

Correction de l'exercice 16 (The Residents - Commercial Album)

1. On suppose que l'on a deux variables `couplet` et `refrain` de type chaîne de caractères et une fonction `sing()` qui chante le texte écrit dans une chaîne de caractères.
Écrire un programme qui permet d'obtenir une chanson pop selon le constat de `The Residents`.

```
for i in range(3) :
    sing(couplet)
    sing(refrain)
```

2. La structure proposée par `The Residents` est plutôt simpliste[22]. Une chanson pop a plutôt la structure suivante : `intro`, `couplet`, `refrain`, `couplet`, `refrain`, `pont`, `refrain`, `couplet`, `refrain`, `refrain`[23].
Compléter le programme ci-contre :

```
sing(intro)
for i in range(4) :
    if i!=2 :
        sing(couplet)
    else :
        sing(pont)
    sing(refrain)
sing(refrain)
```

Correction de l'exercice 17 (Retour de l'exercice 13 (*))

Écrire un programme qui affiche tous les nombres multiple de 13 compris entre 0 et 666.

```
for i in range(0,667,13) :
    print(i)
```

e) La structure répétitive TANT QUE

Correction de l'exercice 18

Expliquer ce qui se passe dans les deux cas suivants :

```
i = 1
while i<11:
    sing(str(i) + " km à pieds ça use, ça use")
    sing(str(i) + " km à pieds ça use les souliers")
```

Comme la variable `i` n'est pas modifiée, elle vaut toujours 1. La condition `i<11` est toujours vraie. Le programme bouclera indéfiniment.

22. Et c'était le but !
23. Ce qui permet d'obtenir une chanson d'un peu plus de 4 min !!!!

```
1 i = 1
2 while i<11:
3     sing(str(i) + " km à pieds ça use, ça
      use")
4     sing(str(i) + " km à pieds ça use les
      souliers")
5 i = i + 1
```

La modification de la variable `i`, ligne 5, ne s'effectue qu'une fois la boucle **TANT QUE** terminée. On peut le voir car il n'y a pas d'indentation et donc la ligne 5 ne fait pas partie du bloc. La variable `i` n'est donc pas modifiée dans le bloc, elle vaut toujours 1. La condition `i<11` est toujours vraie. Le programme bouclera indéfiniment (et la ligne 5 ne sera jamais lu).

Correction de l'exercice 19 (Kraftwerk - Music non stop)

Écrire le plus simplement possible un programme qui répète à la l'infini `music non stop`.

```
1 while True :
2     sing("music non strop")
```

Correction de l'exercice 20 (Jeu de dé (**))

1. Écrire un programme qui simule le lancer d'un dé, jusqu'à ce qu'on obtienne enfin un 6.
Les résultats de chaque lancer seront affichés avec la fonction `print()`.

```
1 from random import randint
2 de = 7 #où autre nombre différent de 6
3 while de != 6 :
4     de = randint(1,6)
5     print(de)
```

2. (*) Modifier le programme précédent pour afficher à la fin le nombre total de lancer nécessaire pour enfin obtenir un 6.

```
1 from random import randint
2 de = 7 #où autre nombre différent de 6
3 nb = 0
4 while de != 6 :
5     de = randint(1,6)
6     print(de)
7     nb = nb + 1
8 print("Il y a eu " +str(nb)+ " lancers.")
```

3. (**) On décide de déterminer combien, en moyenne, il est nécessaire de lancer un dé pour obtenir un 6. On décide d'effectuer 10 fois des séries de lancers de dé jusqu'à obtenir 6, puis de calculer la moyenne des 10 nombre de lancers nécessaire.
Pour une plus grande clarté, nous n'afficherons plus les résultats des lancers du dé.

```
1 from random import randint
2 somme = 0
3 for i in range(10):
4     de = 7 #où autre nombre différent de
      6
5     nb = 0
6     while de != 6 :
7         de = randint(1,6)
8         nb = nb + 1
9     print("Il y a eu ",nb," lancers.")
10    somme = somme + nb
11 moyenne = somme / 10
12 print("La moyenne est de",moyenne,"
      lancers")
```

f) Les sous-programmes

Correction de l'exercice 21 (Retour sur l'exercice 12)

Écrire une fonction qui permet d'afficher la table de multiplication choisie.

```
def table_multi(nb) :
    '''
    entrée : nb est un nombre entier
    affiche la table de multiplication de nb de 1 à 10
    '''
    for i in range(1,11) :
        print(nb,"fois",i,"égale",nb*i)
```

Correction de l'exercice 22 (Retour sur l'exercice 13)
Écrire une fonction qui correspond à sa documentation.

```
def multi(nb,a,b) :
    '''
    entrées : nb, a, b sont des nombres entiers
    affiche la liste des multiples de nb compris entre a et b (inclus)
    '''
    for i in range(a,b+1) :
        if i%nb == 0 :
            print(i)
```

Correction de l'exercice 23 (1 2 3 BOUM ! (*))
La fonction `boum(max,div)` affiche la liste des nombres entiers de 1 jusqu'à `max` avec une légère modification. Tous les nombres divisibles par le nombre `div` sont à écarter de la liste et sont remplacer par la chaîne de caractères `"BOUM !"`.
Par exemple `boum(14,3)` affichera 1, 2, BOUM !, 4, 5, BOUM !, 7, 8, BOUM !, 10, 11, BOUM !, 13, 14.

```
def boum(max,div) :
    '''
    entrées : max, div sont des nombres entiers
    affiche la liste modifiée des nombres entre 1 et max (inclus)
    '''
    for i in range(1,max+1) :
        if i%div == 0 :
            print("BOUM !")
        else :
            print(i)
```

Correction de l'exercice 24 (1 2 3 BOUM ! (***))
La fonction `boum_plus(max,div)` affiche la liste des nombres entiers de 1 jusqu'à `max` avec une légère modification. Tous les nombres divisibles par le nombre `div` sont à écarter de la liste, ainsi que les nombres qui contiennent la séquence de chiffres de `div` et sont remplacer par la chaîne de caractères `"BOUM !"`.
Par exemple `boum_plus(14,3)` affichera 1, 2, BOUM !, 4, 5, BOUM !, 7, 8, BOUM !, 10, 11, BOUM !, BOUM !, 14.
(le nombre 13 n'est pas affiché car il contient le chiffre 3)

```
def boum_plus(max,div) :
    '''
    entrées : max, div sont des nombres entiers
    affiche la liste modifiée des nombres entre 1 et max (inclus)
    '''
    for i in range(1,max+1) :
        if (i%div == 0) or (str(div) in str(i)) :
            print("BOUM !")
        else :
            print(i)
```

Correction de l'exercice 25 (Retour sur l'exercice 20 (**))

La théorie probabiliste nous apprend qu'en **moyenne** il faut lancer 6 fois un dé pour enfin obtenir le nombre 6. A l'exercice 20, nous n'avons la moyenne que sur 10 lancers ce qui nous donne des résultats très fluctuants.
Écrire une fonction `moyenne_de(faces,series)` qui retourne sous la forme d'un flottant le nombre moyen de lancers nécessaire pour enfin obtenir le résultat `faces` pour un dé qui aurait `faces` faces sur un total de `series` séries de lancers.
Par exemple `moyenne_de(30,1000)` affichera le nombre moyen de lancers pour enfin obtenir 30 avec un dé à 30 faces sur un total de 1000 séries de lancers.
(Comme vous vous en doutez, ce nombre est en théorie égale à 30)

```
from random import randint

def moyenne_de(faces,series) :
    '''
    entrées : faces, series sont des nombres entiers
    sortie : un nombre flottant
    calcul la moyenne sur series séries de lancers jusqu'à obtenir le nombre faces
    '''
    somme = 0
    for i in range(series):
        essais = -1 #pour être égal à faces
        nb = 0
        while essais != faces :
            essais = randint(1,faces)
            nb = nb + 1
        somme = somme + nb
    moyenne = somme / series
    return moyenne
```