BASES DE LA PROGRAMMATION

I Un exemple d'algorithme égyptien: Ahmès

On donne les deux nombres 50 et 64.

1. Écrire une colonne d'entiers sous le nombre 50 dont chaque nombre est la moitié, ou la plus petite des deux moitiés s'il y a le choix, du nombre situé immédiatement au-dessus de lui, et en s'arrêtant au nombre 1.
2. Écrire à la droite de la colonne précédente, une colonne commençant au nombre 64, de longueur égale à celle de la précédente colonne, et où chaque nombre est le double du nombre situé immédiatement au-dessus de lui.
3. Écrire, à la droite de la colonne précédente, une troisième colonne où, pour chaque ligne, si le contenu de la première colonne est impair, on recopie le contenu de la seconde colonne, sinon on laisse en blanc.
4. Additionner les nombres de la troisième colonne.

50	64

II Le même en « langage naturel », puis en langages de programmations.

Langage naturel :

Algorithme d'Ahmès

En Javascool :

```
int ahmes ( int a , int b) {
        int x = a ;
        int y = b ;
        int z = 0 ;
        while ( x != 0) {
                if ( x%2 == 1) {
                        z = z + y ;
                }
                x = x / 2 ;
                y = y * 2 ;
        }
        return z ;
}
void main ( ) {
print ( ahmes (50 , 64) ) ;
```

En Python :

```
        def ahmes ( a , b) :
            x = a
            y = b
            z = 0
            while x != 0 :
                    if x%2 == 1 :
                            z = z+y
                    x = int ( x /2)
                    y = y*2
            return z
        print ( ahmes ( 50 , 64 ) )
```

En C :

```
    #include <stdio.h>
    int ahmes(int a, int b) {
            int x = a ;
            int y = b ;
            int z = 0 ;
            while (x != 0) {
                    if (2*(x/2) != x)
                            z = z+y ;
                    x = x/2 ;
                    y = y*2 ;
            }
    }
    main() {
                    printf("%d", ahmes(50,64)) ;
```

En Ada :

```
with Ada.Text_Io ; use Ada.Text_Io ;
with Ada.Integer_Text_Io ; use Ada.Integer_Text_Io ;

procedure TestFctnAhmes i s
function Ahmes ( a , b : in integer ) return integer is
        x , y , z : integer ;
        begin -- Ahmes
                x := a ;
                y := b ;
                z := 0 ;
                while x/=0 loop
                        if x mod 2 = 1 then
                                z := z + y ;
                        end if ;
                        x := x / 2 ;
                        y := y * 2 ;
                end loop ;
                return z ;
        end Ahmes ;

begin
        put (Ahmes ( 5 0 , 6 4 ) ) ;
end                     TestFctnAhmes                   ;
```

Caractéristiques communes et différences :

III Rappels (en théorie) sur l'algorithmique.

Les instructions autorisées sont :

- Affectation de variables (symbolisées par ←, →, :=, voire =)
- Instruction répétitive :
  - Tant que *condition* faire… fin tant que.
  - Instructions complémentaires
    Pour *variable de valeur de début à valeur de fin* faire… fin pour
    Répéter… jusqu'à *condition*
- Instruction conditionnelle :
  - Si *condition* alors… (sinon…) fin si
  - Instruction complémentaire (aiguillage traduit en switch, case, elif):
    Si *condition1* alors … sinon si *condition2* alors… etc… (sinon…) fin aiguillage
- Entrée/sortie :
  - Lire au clavier
  - Afficher/imprimer à l'écran

Pour chaque conditionnelle, chaque répétitive, les instructions doivent être décalées, et un trait doit limiter la séquence d'instructions à effectuer à l'intérieur de cette conditionnelle/répétitive.

IV Quelques notions de base sur la programmation.

1. Qu'est-ce que programmer ?
   Programmer, c'est résoudre des problèmes, puis les traduire en une suite d'ordres donnés à l'ordinateur.
   Un ordinateur sans programme ne sait rien faire : *Windows* est un programme, les jeux sont des programmes.
2. Les différents types de langage de programmation
   Le seul langage de programmation directement utilisable par un ordinateur est le langage machine, sous forme binaire (formé uniquement de 0 et de 1).
   Personne ne programme en langage machine car c'est trop compliqué. Ainsi, les informaticiens ont inventé de nombreux langages qui utilisent des instructions au lieu d'une suite de 0 et de 1. Ces instructions, une fois écrites par le programmeur, sont traduites en langage machine, à l'aide d'un programme destiné à cet effet. Ce système de traduction s'appelle interpréteur ou bien compilateur, suivant la méthode utilisée pour effectuer la traduction.
   Il existe 2 types de langages de programmation :
   - les langages de bas niveau : très complexes à utiliser (car très éloignés du langage naturel), on dit que ce sont des langages « proches de la machine ». Ils permettent en contrepartie de faire des programmes très rapides à l'exécution.
   - Les langages de haut niveau : ils sont plus faciles à utiliser car plus proches du langage naturel.
3. Compilation et interprétation
   Le programme tel que nous l'écrivons est appelé programme source (ou code source). Comme déjà signalé plus haut, il existe deux techniques principales pour effectuer la traduction d'un programme source en langage machine : l'interprétation et la compilation.
   a *L'interprétation*
   Dans cette technique, le logiciel interpréteur doit être utilisé chaque fois que l'on veut faire fonctionner le programme. Chaque ligne du programme source analysé est traduite au fur et à mesure en quelques instructions du langage machine, qui sont ensuite directement exécutées. Python, Javascript, par exemple, sont des langages interprétés.
   b *La compilation*
   La compilation consiste à traduire la totalité du code source en une seule fois. Le logiciel compilateur lit toutes les lignes du programme source et produit une nouvelle suite de codes, écrits en langage machine, que l'on appelle programme objet (ou code objet). Celui-ci peut désormais être exécuté indépendamment du compilateur et être conservé tel quel dans un fichier appelé fichier exécutable. Le *C++*, par exemple, est un langage compilé.

c Avantages et inconvénients de l'interprétation

L'interprétation est idéale lorsque l'on est en phase d'apprentissage d'un langage, ou en cours d'expérimentation sur un projet. Avec cette technique, on peut en effet tester immédiatement toute modification apportée au programme source, sans passer par une longue phase de compilation qui peut durer des heures pour de très gros programmes.

Par contre, traduire les instructions à la volée ralentit l'exécution du programme. Les programmes en langage interprété sont toujours plus lents à l'exécution que les programmes en langage compilé.

d Choix du langage Python

Il existe un très grand nombre de langages de programmation, chacun ayant ses avantages et ses inconvénients.

Parmi les langages libres[1] et gratuits il existe des interpréteurs et des compilateurs pour toute une série de langages, mais surtout ces langages sont modernes, performants, portables (c'est-à-dire utilisables sur différents systèmes d'exploitation tels que *Windows*, *Linux*, *Mac OS* ...), et fort bien documentés.

Le langage dominant est sans conteste le *C/C++* et ses variantes. Ce langage s'impose comme une référence absolue, et tout informaticien sérieux doit s'y frotter tôt ou tard. Il est assez proche de la machine donc sa syntaxe est peu lisible (la remarque sur la lisibilité vaut aussi dans une large mesure pour le langage *Java*, très répandu également).

Pour les débuts dans l'étude de la programmation, il est préférable d'utiliser un langage de plus haut niveau, moins contraignant, à la syntaxe plus lisible. Nous avons décidé d'adopter *Python*, langage couramment utilisé, langage qui s'impose comme référence en machine learning (intelligrnce artificielle)

V Comment utiliser Python

Les outils dont vous aurez besoin peuvent se limiter à Python et un environnement de développement. JE vous conseille très fortement (très très fortement) (très très très fortement) d'installer de quoi lire les notebooks (cf 2. Jupyter)

1. Installation de Python (pour : exercices, devoirs, projet)

   La première chose à faire pour utiliser Python est de l'installer, si nécessaire…

   Si vous utilisez *Linux* sur votre machine, vous avez peut-être déjà *Python 3*. La plupart des distributions *GNU/Linux,* ainsi que Mac OsX, installent par défaut *Python 2*, de plus en plus de distributions fournissent également *Python 3*. Les autres systèmes d'exploitation comme Windows ne sont fournis avec aucune version de Python.

   Si Python est installé sur votre machine, assurez vous qu'il s'agit de la version 3. Si tel n'est pas le cas, vous pouvez downloader Python à : http://www.python.org/download/. Veillez à télécharger la dernière version (Python 3) correspondant à votre environnement (*Windows*, *Macintosh*, etc.).

2. Installation de Jupyter (pour reprendre les cours en notebook).

   Si vous souhaitez retravailler les cours interactifs distribués chez vous, il est indispensable d'utiliser un lecteur de notebook Python.

   Vous pouvez utiliser un environnement Python temporaire avec http://tmpnb.org. Inconvénient majeur : l'upload est très capricieux.

   Deuxième méthode, quasiment aussi simple, et bien plus fiable : télécharger la distribution Anaconda et son Navigator à https://www.anaconda.com/download/ . Choisir la version correspondante à votre système d'exploitation.

   Puis « upload » le fichier à utiliser, et double-clic pour le lancer.

[1] Un **logiciel libre** (*Free Software*) est avant tout un logiciel dont le code source est accessible à tous (*Open Source*). Souvent gratuit (ou presque), copiable et modifiable librement au gré de son acquéreur, il est généralement le produit de la collaboration bénévole de centaines de développeurs dispersés dans le monde entier. Des exemples de logiciels libres : le système d'exploitation *GNU/Linux*, la suite O*pen Office*. Un logiciel non libre est dit **propriétaire**. Des exemples de logiciels propriétaires : le système d'exploitation *Windows*, la suite *Microsoft Office.*

Troisième méthode, plus complexe, parce que l'on utilise des commandes en ligne, et non une interface graphique. Mais on y survit !

- Sous OsX ou Linux, c'est faisable en quelques étapes, en commençant par :
  - Ouvrir un terminal (OsX : applications>utilitaires, Linux : dépend de l'interface graphique utilisée). Sauter la partie ci-dessous « sous Windows ».
- Sous Windows, le début est un peu plus compliqué (n'ayant pas Windows8, je ne l'ai fait que sous Windows7, c'est donc à adapter).
  - Après avoir installé Python, il faut changer les variables d'environnement (la méthode provient de https://adesquared.wordpress.com/2013/07/07/setting-up-python-and-easy_install-on-windows-7/) :
    - dans le panneau de démarrage, clic droit sur ordinateur>propriétés
    - dans la fenêtre qui s'ouvre, paramètres systèmes avancés>paramètres systèmes avancés>variables d'environnement
    - chercher "path" dans variable système, rajouter à la fin ;C:\Python34;C:\Python34\Scripts (si après ça ne marche pas vérifier dans le lien ci-dessus si c'est vraiment ça)
  - Ouvrir une invite de commande en tant qu'administrateur (avec un clic droit sur invite de commande, dans menu démarrer>accessoires)
- Puis, pour tous les systèmes :
  - Taper « pip install ipython » après l'invite de commande (le premier caractère en début de ligne, un $ souvent). Le terminal/l'invite de commande vous raconte tout un tas de choses, puis :
  - Taper « pip install pyzmq jinja2 tornado mistune jsonschema pygments terminado »
  - Sur certains systèmes, il semblerait qu'il faille aussi faire « pip install notebook »
  - Remarque : vous pouvez faire un copier coller de l'instruction précédente directement à partir de https://ipython.org/ipython-doc/3/install/install.html#installation-using-pip, dans le paragraphe « dependancies for the ipython HTML notebook »

3. L'environnement de développement (IDE)

Avant de commencer à écrire des programmes *Python* dans des fichiers sources, nous avons besoin d'un éditeur de texte. Le choix de l'éditeur est très important. L'un des besoins de base est la coloration syntaxique. Ainsi les différentes parties de votre programme sont coloriées de différentes couleurs afin d'en faciliter la lecture et d'éviter les erreurs.

Vous disposez dans l'application MCNL de l'ordinateur de la région de deux éditeurs *IDLE* et *Pyzo*.

IDLE est le plus simple des IDE, il est suffisant. En plus de la coloration syntaxique, *IDLE* offre la possibilité de lancer votre programme à l'intérieur de *IDLE*. Celui-ci est installé par défaut avec les installeurs de *Python* pour *Windows* et *Mac OS X*. Il est aussi disponible à l'installation pour *Linux*.

Pyzo est plus complexe ; il présente l'avantage d'avoir le terminal et le code source dans une même fenêtre. Il a aussi de nombreuses options que vous n'utiliserez pas.

Enfin, l'iDe favori de votre professeur est Spyder, sur Ananconda Navigator. Il arrive que la version de Python utilisé dans Spyder ne soit pas la dernière. De plus Spyder possède en plus des bibliothèques scientifiques spécialisées, qui sont très pratiques d'usage. Ces bibliothèques ne seront utilisées qu'en fin d'année, et je vous donnerai le code nécessaire : vous n'aurez pas à en comprendre l'usage.

Il existe des environnements de développement bien plus complexes, comme Eclipse par exemple. Ceci au cas où vous souhaiteriez programmer le prochain Call Of Duty tout(e) seul(e).

4. Un manuel

Qui offre l'avantage d'être gratuit (en téléchargement légal), assez complet, et pas mal fichu, malgré un index parfois défaillant. Il s'agit de G.Swinnen : Apprendre à programmer avec Python 3.

Téléchargeable gratuitement à cette adresse : http://inforef.be/swi/python.htm.

5. Remarque sur les noms de fichier

Je vous conseille, lorsque vous faites un exercice, de mettre comme nom de fichier :

TD1_Ex1_VotrePrénom.py, et de mettre à nouveau en première ligne de commentaire le numéro de l'exercice, ainsi que votre prénom. Ainsi vous pourrez réviser comme vous le faites en mathématiques ou en physique.

VI Les bases du Python.

1. Commentaires.

Les commentaires sur une ligne, ou en fin de ligne, commencent par # ; ils sont indispensables pour la compréhension, la réutilisation ultérieure et la relecture d'un programme.

On peut mettre des commentaires de plusieurs lignes entre triples guillemets : """…"""

Pour des raisons de compatibilité sur les différents systèmes, évitez les accents ainsi que tous les caractères spéciaux dans les noms de variables (cf. ci dessous). Vous pouvez en mettre dans les

commentaires.

2. Expressions.
   Outre les opérations arihtmétiques standard, on dispose des opérateurs suivants:
   - //
   - %
   - **
3. Variables
   Le nom des variables suit quelques règles de base :
   - un nom de variable est une suite de lettres (a → z , A → Z) et/ou de chiffres (0 → 9), qui doit toujours commencer par une lettre ;
   - seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont à proscrire, à l'exception du caractère _ (tiret bas ou underscore) ;
   - la casse est significative (les caractères majuscules et minuscules sont distingués). Par exemple, Nombre, nombre, NOMBRE sont des variables différentes. Soyez attentifs !
   - prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans nomDeFamille[2]. Par contre, si vous utilisez le nombre π souvent, c'est une constante ; on écrira alors la variable correspondante en majuscules PI = 3.14159.
   - N'hésitez pas à donner des noms clairs à vos variables. Par exemple, nomDeFamille est plus clair que nf.
   - De plus, les variables ne peuvent pas être un des « mots réservés[3] » suivant utilisés par le langage lui-même :

```
and   as        assert    break     class     continue  def   del
elif  else  except    exec  False     finally   for   from  global
if    import    in    is    lambda    None  nonlocal   not   or    pass
print           raise     return    True  try   while     with  yield
```

4. Booléens
   Les booléens sont des variables qui ne prennent que deux valeurs : vrai ou faux (`True` et `False` respectivement en Python, avec des majuscules)
   Les variables booléennes sont utilisées pour les tests lors des boucles ou des conditionnelles. On peut les combiner à l'aide des connecteurs logiques : et, ou, non qui sont en Python `and`, `or`, `not`.
   Opérateurs de comparaison :

```
x == y          # x est égal à y
x != y          # x est différent de y
x > y      # x est plus grand que y
x < y      # x est plus petit que y
x >= y          # x est plus grand que, ou égal à y
x <= y          # x est plus petit que, ou égal à y
```

   Attention : pas d'encadrement en programmation
   Opérateurs logiques :

```
a and b         # « et » mathématique
a or b          # « ou » mathématique
not(a)          # « non » mathématique
```

   Ils correspondent aux notions vues en probabilité sur les événements.
5. Typage
   Au paragraphe précédent, on a vu les variables de type « booléen ». Toutes les variables ont un type ; voici ci-dessous les types simples :
   - `int` : nombre entier

[2] Cette convention est très utilisée. Personnellement, je trouve nom_de_famille encore plus lisible.
[3] La liste est variable suivant les versions de Python

- `float` : nombre décimal (nombre à virgule « qui se finit »). On ne peut pas réprésenter en machines des nombres réels « compliqués » comme π ou $\sqrt{2}$, on en a juste des approximations
- `str` : chaîne de caractères, comme « bonjour le monde ». Les chaînes de caractère sont entre guillemets simples ou doubles `"bonjour le monde"` ou `'bonjour le monde'`
- `bool` : booléens

Le type d'une variable est obtenu avec l'instruction type(var).
Exemple :

```
>>> a = "coucou"
>>> type(a)
<type 'str'>
```

On peut forcer une variable d'un type à devenir une variable d'un autre type ; attention cela peut causer des erreurs ! Pour changer le type d'une variable, on utilise le type que l'on veut affecter :
Exemple :

```
>>> a = 23.1
>>> type(a)
<type 'float'>
>>> a = str(a)
>>> a
'23.1'
>>> b = 23.1
>>> b = int(b)
>>>b
23
>>> c = "bonjour"
>>> int(c)
Traceback (most recent call last):
       File "<ipython-input-20-089eb950f64f>", line 1, in <module>
            int(c)
       ValueError: invalid literal for int() with base 10: 'bonjour'
```

Quelques opérations sur les chaînes de caractères données par des exemples :
Les chaînes de caractères sont un type particulier de liste, que l'on verra ultérieurement plus en détail.

- La *concaténation et la répétition* :

```
>>> a = "j'aime "
>>> b = "pas "
>>> c = "les mathématiques"
>>> print(a+c)        # concaténation
j'aime les mathématiques
>>> print(3*b)        # répétition
paspaspas
```

- Longueur d'une chaîne de caratères :

```
>>> len("j'aime ")
7        #  Comptez bien tous les caractères !
```

- Obtenir un caractère ou un morceau d'une chaîne de caractères :

```
>>> a = "bonjour"
>>> len(a)
7
>>> a[0]
'b'
>>> a[6]
'r'
>>> a[7]
Traceback (most recent call last):
  File "<ipython-input-7-9cf13ba20553>", line 1, in <module>     a[7]
IndexError: string index out of range
```

Comment fonctionne le comptage des caractères ?

```
>>> a[1 :3]
'on'
```
Quels sont les indices (numéros) des caractères affichés dans cette commande ?

6. Instructions conditionnelles

Le « si *condition* alors *instruction* (sinon *instruction*) » se traduit par :

```
if condition :
    instruction 1
(else :)
    instruction 2
suite du programme
```

Vous remarquez l'indentation, c'est à dire le fait que l'instruction est décalée vers la droite (en général de 4 espaces). Ceci traduit le fait que l'instruction 1 sera exécutée uniquement si la condition dans le « si » est vraie, l'instruction 2 étant exécutée uniquement si la condition est fausse. Bien sûr, l'instruction 1 peut être formée de plusieurs instructions élémentaires. Si l'on imbrique plusieurs « si », alors on indentera à chaque fois les instructions. N'oubliez pas les « : ».
L'ensemble if *condition* : *instruction 1* forme une instruction composée.

La condition évalue un booléen, cf. notebook nsi_1_2_controle.ipynb.

L'instruction « `if`…`elif`…`elif`…etc…(`else :`) » permet de choisir entre plusieurs alternatives (elif est la contraction de else if)

7. Instructions répétitives

Le « tant que » est traduit par :

```
while condition :
    instruction
suite du programme
```

Même remarque sur l'indentation, ainsi que sur la condition qui est un booléen, qu'au paragraphe précédent

Le « pour » est traduit par :

```
for variable in séquence :
    instruction 1
suite du programme
```

Par exemple, « `for i in range(0, 10, 3) :` » exécutera l'instruction 1 pour les valeurs de *i* respectivement égales à 0, 3, 6, 9. Si on écrit `range(0, 10)` , alors *i* prend toutes les valeurs de 0 à 9 (10 exclu).

Lorsque l'on a une condition compliquée à tester pour sortir d'une boucle par exemple, il est souvent pratique d'utiliser un booléen (exemple : un jeu que l'on peut finir de plusieurs manières à l'intérieur de la boucle, cf. exercice « deviner un nombre »). Dans ce cas, si « `continuer` » est le nom de la variable booléenne, on écrit « `while continuer` », et non « `while continuer == True` » qui est un pléonasme informatique !

Remarque : de nombreux programmes utilisent l'instruction `break` pour sortir d'une boucle, voire d'un test. Cette instruction est à éviter autant que possible, pour des raisons de bonnes pratiques de programmation. En effet elle rend les programmes moins lisibles, et est source d'erreurs difficilement corrigeables. Au lycée, sans interface graphique, il n'y a aucune utilisation nécessaire de cette instruction. En conséquence, elle vous est interdite d'usage (ou presque). Vous devez néanmoins savoir qu'elle existe. Quel que soit le niveau, l'utiliser avec un `for` ou un `if` est de la mauvaise programmation. Avec un while, on voit des exemples comme suit :

`while True :`
`  if condition :`
`  break`
`...`
	Que l'on peut remplacer par :	`var_bool = True`
		`while var_bool :`
		`  if condition :`
		`  var_bool = False`
		`  else :`
		`  ...`

8. Modules

Une bibliothèque (ou un module) est un ensemble de fonctions préprogrammées, ainsi le programmeur n'a pas à les refaire.
On inclut une bibliothèque dans un programme à l'aide de la commande « import ».
Nous utiliserons les bibliothèque « math », qui permet de calculer des cosinus, des racines carrées, des valeurs absolues… et la bibliothèque « random », qui permet d'obtenir des nombres aléatoires. Ultérieurement, nous utiliserons également des bibliothèques graphiques pour les interfaces.
On tape si nécessaire en début de programme :

```
>>> from math import *
>>> from random import *
```

9. Lisibilité des programmes et fonctions

Pour être lisible, un programme doit être structuré en fonctions, que l'on écrira au début du programme. Lorsque l'on réfléchit à un problème que l'on veut programmer, il faut le faire en termes de fonctions : « je veux d'abord une fonction qui me permet de rentrer mes données », « je veux une fonction qui teste si une variable est positive et dans ce cas calcule sa racine carrée » etc…

Les fonctions ressemblent à ce que vous utilisez en mathématiques quand vous écrivez $y = f(x)$. Vous avez en *paramètre d'entrée* un nombre *x*, une fonction qui fait « un truc », et en *paramètre de retour ou de sortie* un nombre *y*, résultat de la fonction. En informatique on peut avoir 0, 1 ou plusieurs paramètres d'entrée, et de même en sortie.
La syntaxe pour les fonctions est :

`def` *nom_de_fonction* (*variables passées en paramètre, à utiliser dans la fonction*) :
  instructions
  `return` (*variable*)

L'appel dans le programme se fait par :
variable_resultat = nom_de_fonction (*variables à passer en paramètre*)

Exemple : le programme suivant teste vos capacités en calcul mental

```
# F. Mandon
#
# Programme de test de connaissance sur les tables de multiplication

# bibliotheques
from random import *
######################
#
# Fonctions
#
######################
def alea10():
      """
      Choix d'un nombre aléatoire entre 2 et 10
      @param : aucun paramètre d'entrée
      @return n : entier n ≥ 2 et n ≤ 10
      """
      n = randint(2,10)
      return (n)


def testTable(x,y,z):
```

```
    """
    Teste si le produit de deux nombres est égal à un troisième
    @param x, y, z : trois nombres (a priori de type entier, mais ce n'est pas
    obligatoire)
    @return gagne : booleen, qui vaut Vrai si z = x*y et Faux sinon
    """


    if z == x*y:
        gagne = True
    else:
        gagne = False
    return(gagne)
#####################
#
#   Programme principal
#
#####################
a = alea10()
b = alea10()
print("Que vaut le produit de ",a," par ",b," ?")
c = int(input())
juste = testTable(a,b,c)
if juste:
    print("Bravo !")
else:
    print("Lamentable...")
```

Dans le code, on met tous les `import` en premier, puis toutes les fonctions, puis le programme principal.
Vous remarquez que les fonctions sont commentées (on dit *spécifiées*), on indique d'abord leur nom, puis les noms et types des variables passées en paramètre (avec d'éventuelles restrictions), et enfin le(s) nom(s) et type(s) de la/des variable(s) renvoyée(s). Les spécifications sont mises entre triples guillemets. On peut préciser dans les commentaires la « stratégie » de la fonction, si elle est un peu compliquée. La spécification est indispensable. Il est préférable de taper les commentaires au fur et à mesure, pendant que l'on sait ce que l'on vient de faire… Il est même fréquent de le faire avant de construire la fonction, en laissant cette dernière vide si elle est compliquée à faire !
A terme, vous devez dans les spécifications :

- Définir l'objectif de la fonction
- Identifier les paramètres de la fonction : ce qui est en entrée, ce qui est en sortie
- Choisir un nom de fonction clair
- Choisir des noms de variables clairs
- Donner les types des paramètres, et préciser s'il y a des conditions dessus (on parle de préconditions pour les paramètres d'entrée et de postconditions pour les paramètres de sortie).

Les variables crées à l'intérieur des fonctions sont *locales*, c'est-à-dire qu'elles n'existent pas en dehors des fonctions. Si dans le programme précédent donné en exemple, vous faites un `print(z)` dans le programme principal, il y aura une erreur.

Remarque : évitez les petites blagues qui consiste à donner des noms rigolos, mais ne facilitent pas la compréhension

`def ensedelephant (var icelle)`… pour une fonction qui calculerait l'écart entre deux dates ne rend pas les choses très claires !

EXERCICES PROGRAMMATION

INTRODUCTION

Ex 1. On dispose de la formule suivante pour convertir les degrés Fahrenheit en degrés Celsius : $C = 0,55556 \times (F - 32)$, où F est une température en degrés Fahrenheit et C la température correspondante en degrés Celsius.

1. Ecrire un programme qui convertit en degrés Celsius une température rentrée au clavier en degrés Fahrenheit.
2. Même question pour la conversion inverse.

Ex 2. Écrire un programme qui permute et affiche les valeurs de trois variables *a*, *b*, *c* qui sont entrées au clavier : *a* ==> *b* , *b* ==> *c* , *c* ==> *a*.

BOUCLES ET CONDITIONS

Ex 3. (pour ceux qui l'ont fait en spécialité mathématiques) Ecrire un programme qui donne le nombre de solutions réelles de l'équation du second degré, ainsi que leur valeur. Les coefficients *a*, *b* et *c* seront rentrés au clavier.

Ex 4. Ecrire un programme qui lit une valeur et affiche sa table de multiplication (on se limitera aux 12 premiers termes)
Faire une variante du programme précédent qui affiche la table de multiplication de tous les chiffres compris entre 2 et 9 (inclus).
Remarque : Pensez à laisser un espace entre deux tables de multiplication (`print()` imprime une ligne vide).

Ex 5. Écrire un programme qui affiche un triangle rempli d'étoiles (*) sur un nombre de lignes donné passé en paramètre, exemple :

- 1ère version : à l'aide de deux boucles for, en imprimant les * une par une. On remarquera que, par défaut dans l'instruction « print », figure end = '\n', qui fait passer à la ligne. print(…, end ='') ne fera donc pas passer à la ligne. De même, end = ' lol ' vous fera passer pour utilisateur standard de facebook
- 2ème version : avec une seul boucle for, et une chaîne de caractères où vous accumulerez des étoiles (pour ceux qui vont un peu plus vite, print(« machin » end= '') évite de passer à la ligne.

```
*
**
***
****
*****
******
*******
********
```

Ex 6. Ecrire un programme qui teste si un nombre *a* est divisible par un nombre *b*, les deux étant rentrés au clavier. Le programme retournera un message signalant la divisibilité ou non, et éventuellement le reste dans le cas où *a* n'est pas divisible par *b*.

Ex 7. Programmation d'un petit jeu de devinette. L'ordinateur choisit au hasard un nombre compris entre 1 et 100. Le but du jeu est de le deviner en un nombre d'essai minimal. À chaque tentative, l'ordinateur, indique « gagné », « trop petit » ou « trop grand ». L'utilisateur dispose d'un nombre d'essais limités.
Écrire l'algorithme en « langage naturel ». Programmer le jeu, et le tester.
Remarque : on utilisera la bibliothèque *random.*
Pour cela, on écrit « import random » en début de programme.
nombre = `random.randint`(*a*, *b*) renverra un nombre aléatoire tel que $a \leq Nombre \leq b$
Pour plus d'informations sur le bibliothèque random et de possibilités : https://docs.python.org/3.5/library/random.html

Ex 8. Écrire une fonction qui retourne la factorielle d'un nombre (exemple $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$).
Déduire de la solution précédente, une fonction qui permet le calcul du nombre de combinaisons de p éléments parmi q définie par : $\begin{pmatrix} n \\ p \end{pmatrix} = \frac{n!}{p!(n-p)!}$.

Ex 9. Nombres parfaits
Un nombre entier est parfait s'il est égal à la somme de ses diviseurs (sauf lui-même).
Ex : 6 = 1 + 2 + 3 est parfait.
Écrire une fonction `somme_div` qui retourne la somme des diviseurs d'un nombre passé en paramètre.
Écrire une fonction `parfait` qui teste si un nombre passé en paramètre est parfait et qui retourne `True` s'il l'est et `False` sinon. Écrire un programme principal qui affiche tous les nombres parfaits inférieurs à une certaine limite.
Dans un deuxième temps, optimiser la fonction `somme_div`

Ex 10. Nombres amicaux.
Deux nombres M et N sont appelés nombres amis si la somme des diviseurs propres (sauf M) de M est égale à N et la somme des diviseurs propres de N est égale à M
Écrire une fonction `amis` qui teste si deux nombres passés en paramètre sont amis. Cette fonction utilise la fonction `somme_div` de l'exercice 8.
Écrire un programme principal qui affiche tous les nombres amis inférieurs à une certaine limite.

TYPAGE

Ex 11. Trouver les chiffres a et b tels que le nombre $32a1b$ soit divisible par 156

Ex 12. Trouver sans calculatrice le plus petit nombre divisible par 999 qui ne contienne pas de 9 parmi ses chiffres.

Ex 13. Combien de chiffres a le plus petit nombre entier qui se termine par 2, tel que si l'on déplace le chiffre 2 en tête du nombre, on obtient un nombre deux fois plus grand ? *Remarque : écrivez le code, et laissez mijotez quelques heures si vous partez de 0… le nombre est très grand (on peut prendre comme valeur initiale pour rechercher :* 105 263 157 894 736 002)

Contact et site du professeur

email prof : frederic.mandon@ac-montpellier.fr
Site prof : http://www.maths-info-lycee.fr/