Jean-Luc Éveno ©

Apprendre à Programmer

Version du 6 juin 2023

Table des matières

I Apprentissage de la programmation **2**

1 Qu'est ce que programmer ? **3**
1.1 Les algorithmes et leur structure . . . 3
1.2 Les données . . . 4
1.3 Les types des variables . . . 6
1.4 Les fonctions d'entrée et de sortie . . . 9
1.5 Exercices . . . 10
1.6 Correction des exercices . . . 12

2 Les structures de contrôle **16**
2.1 Les instructions conditionnelles . . . 16
2.2 Instructions de répétition . . . 19
2.3 L'instruction *for* en Python . . . 21
2.4 Exercices . . . 23
2.5 Correction des exercices . . . 27

3 Les Fonctions en Python **36**
3.1 Les fonctions . . . 36
3.2 Modules . . . 42
3.3 La récursivité en Python . . . 47
3.4 Correction de quelques exercices . . . 51

4 Les proverbes du programmeur **58**
4.1 Surtout, mais surtout pas d'astuces ! . . . 58
4.2 Une seule chose à la fois . . . 59
4.3 Nommez ce que vous ne connaissez pas encore . . . 59
4.4 Demain sera mieux, après demain, encore mieux ! . . . 59
4.5 On n'exécute jamais un ordre avant de le donner . . . 59
4.6 Documentez aujourd'hui pour ne pas pleurer demain . . . 59
4.7 Le discours de la méthode de Descartes . . . 60

II Mathématiques et programmation **61**

Première partie

Apprentissage de la programmation

Chapitre 1

Qu'est ce que programmer ?

1.1 Les algorithmes et leur structure

1.1.1 Définition et propriété d'un algorithme

1. Un algorithme est une suite d'opérations élémentaires permettant s'obtenir le résultat final lié à un problème déterminé.
2. Un algorithme, dans des conditions d'exécution similaires, avec des données identiques, fournit toujours le même résultat

Remarque 1 :

1. Les algorithmes ont pour fonction de nous faire réfléchir, mais, à priori, pas de s'exécuter sur un ordinateur. L'algorithme décrit, sur papier, un traitement. Il est donc nécessaire d'en simuler le traitement.
2. Un algorithme peut très bien être décrit en langage ordinaire. Dans la plupart des cas, on utilise un langage plus précis, mieux adapté à la nature des instructions ultérieures ; on appele cela un **langage de programmation**, mieux adapté à la nature des instructions utilisées. Nous utiliserons, sur `mathinfovannes` le langage `Python`. Ce n'est, à proprement parler, pas un langage d'apprentissage. Cependant, il est massivement utilisé ; nous nous adaptons donc
3. On peut assimiler les algorithmes à des méthodes, des procédures ou encore des recettes de cuisine ou un mode d'emploi pour monter des meubles
4. Un **algorithme** prend des données en entrée, exprime un traitement particulier et fournit des données en sortie.
5. Un algorithme ne dépend pas du langage dans lequel il est implanté, ni de la machine qui exécutera le programme correspondant
6. **Les besoins pour créer un algorithme** :
   - ▷ Savoir expliciter son raisonnement
   - ▷ Savoir formaliser son raisonnement
7. Un **programme** est une série d'instructions pouvant s'exécuter en séquence, ou en parallèle *(parallélisme matériel)* qui réalise *(implémente)* un algorithme

Remarque 2 :

Structure d'un algorithme

1. Il y a souvent des règles de programmation qu'il faut respecter ; l'apprentissage de la programmation propose souvent 2 parties :
   (a) Une première partie qui est un « bloc de déclarations »
   (b) Une seconde partie qui est un « bloc d'instructions »

2. Que contiennent ces différents blocs ?
   (a) Le **bloc de déclarations** contient le nom du programme, les définitions de variables sous forme de commentaires
   (b) Le **bloc d'instructions** est une partie de traitement d'un algorithme, constituée d'opérations élémentaires.
3. En `Python`, il est très possible de s'affranchir de ces **blocs**. Même si nous nous en affranchissons *(on le verra dans la suite du cours)* il faudra bien savoir sur quoi nous travaillons, sur quelles données le programme devra agir.
4. **Un programme se doit d'être commenté.** Les commentaires sont utiles aux programmeurs qui veulent comprendre ou modifier l'algorithme. La complexité des algorithmes impose de les commenter judicieusement : ni trop, ni trop peu. En langage `Python`, un commentaire est toujours précédé de `#`

Exemple 1 :

Ecrivons l'algorithme qui permet d'afficher « Bonjour tout le monde » ; c'est l'algorithme « Hello World » lequel est d'un classicisme béat.

```
#Algorithme "Bonjour tout le monde"
#Cet algorithme écrit à l'écran "Bonjour tout le monde"
>>> print("Bonjour tout le monde")
Bonjour tout le monde
>>>
```

Remarque 3 :

Pour la clarté de sa lecture, il est bon **d'indenter le programme**. L'indentation est essentielle et naturelle en langage `Python`

1.2 Les données

La plupart des problèmes nécessitent le traitement de valeurs : certaines sont des données de l'énoncé, d'autres issues des calculs réalisés par l'algorithme.
Pour pouvoir être manipulées, les valeurs sont stockées dans des variables.

1.2.1 Définition de variable

Une variable désigne un emplacement mémoire qui permet de stocker une valeur.
Une variable est définie par :

1. Un nom unique qui la désigne
2. Un type de définition unique
3. Une valeur attribuée et modifiée au cours du déroulement de l'algorithme

1.2.2 Nom d'une variable

Le nom ou l'identifiant d'une variable permet d'identifier cette variable de manière unique au cours de l'algorithme.

Remarque 4 :

Pour faciliter la lecture des algorithmes, il faut respecter quelques règles de nommage de ces variables

1. Le nom d'une variable commence par une minuscule
2. Le nom d'une variable ne comporte pas d'espace

3. Si le nom de la variable est composé de plusieurs mots, il faut faire commencer chacun d'eux par une majuscule
4. Donner aux variables un nom explicite

Exemple 2 :

1. Exemple de nommage de variable : `maVariableAMoi` est un bon nommage alors que `Ma Variable A Moi` ne l'est pas du tout
2. Il m'arrive d'utiliser un mix de ces recommandations. J'écris souvent `Ma_Variable_A_Moi` [1]

1.2.3 Type d'une variable

Le type ou le domaine de définition d'une variable indique l'ensemble des valeurs que la variable peut prendre.

Remarque 5 :

1. Les variables peuvent appartenir à différents domaines : **entier**, **réel**, **caractère** ou **booléen.**
2. La valeur de la variable est la seule qui soit modifée au cours de l'algorithme. Au début de l'algorithme *(au moment de la déclaration)*, toutes les variables ont des valeurs inconnues.

1.2.4 Affectation d'une variable

L'affectation est une opération qui donne une valeur à une variable

Remarque 6 :

La syntaxe pour l'affectation diffère suivant les langages ou les constructeurs

1. Sur des calculatrices, c'est souvent : $\leftarrow$
2. En `Python`, nous utiliserons `=`

Exemple 3 :

Nous cherchons à calculer le double d'un réel donné. On introduit donc deux variables associées respectivement à la donnée et au résultat.

```
#Algorithme "Double"
#Cet algorithme donne le double d'un nombre donné

>>> nombre=7          #On affecte le nombre 7 à la variable nombre
>>> nombre=nombre*2   #On multiplie la variable nombre par 2
>>> nombre            #On demande d'afficher la nouvelle valeur de la variable nombre
14                    #C'est la valeur donnée par la variable nombre
```

Remarque 7 :

Pour éviter bien des erreurs, suivez ces recommandations :

1. Une variable n'est déclarée qu'une seule fois dans un algorithme.
2. Avant de pouvoir utiliser la valeur d'une variable, une valeur doit lui être attribuée.
3. Attention : l'égalité mathématique diffère totalement de l'affectation informatique.

1. J'ai horreur de commencer une phrase par une minuscule

Exercice 1 :

On considère le programme suivant :

```
#algo mystere
>>> A=0
>>> B=4*A+3
>>> C=4*B+3
>>> A=B
>>> B=C
```

Quelles sont les différentes valeurs des variables en fin d'exécution du programme ? *(Bien entendu qu'il est tout à fait possible de connaître le résultat en utilisant IDLE, mais, pour progresser il serait bon de le faire sur papier)*

1.3 Les types des variables

1.3.1 Les variables numériques : le type réel et le type entier

Les variables de type numérique utilisées dans les algorithmes, ont comme domaine usuel, ceux fournis par les mathématiques : entiers ou réels

Remarque 8 :

1. Jamais les réels ne sont présents dans les algorithmes informatiques ; ce sont **toujours des approximations décimales.** [2]
2. On démontre que l'ensemble des décimaux et l'ensemble des dyadiques sont des ensembles denses dans $\mathbb{R}$; l'utilisation de ces nombres n'est donc pas injustifiée.
3. Les opérateurs utilisables sur ces éléments sont les opérateurs mathématiques classiques, ainsi que les opérateurs de comparaison $<,>,\leqslant,\geqslant,\neq$
4. Pourquoi 2 types de nombres ? Parce qu'en informatique, les opérations entre entiers et les opérations entre réels suivent des processus différents
5. Les réels, en informatique, sont aussi appelés **flottants**
6. En `Python`, en plus de l'addition et de la multiplication, d'autres opérations sur les entiers sont possibles :
   ⇒ **La division entière**, c'est à dire la recherche du quotient de m par n dans la division euclidienne ; elle s'exprime par un double slash `//`

   ```
   >>> A=15
   >>> B=9
   >>> A//B
   1
   ```

   ⇒ **Le modulo** ou la recherche du reste de la division euclidienne de m par n ; elle s'exprime par un pourcentage `%`

   ```
   >>> A=15
   >>> B=9
   >>> A % B
   6
   ```

   On peut remarquer que nous avons :

   ```
   >>> A=(A//B)*B+(A % B)
   >>> A
   15
   ```

   C'est donc une « reconstitution »de A

2. Aller voir l'annexe « Structure des nombres en machine » dans le cours de L_0

Exemple 4 :

```
>>> nombre1=1.2
>>> nombre2=15
>>> var1= 2
>>> type(nombre1)
<class 'float'>
>>> type(nombre2)
<class 'int'>
>>> type(var1)
<class 'int'>
```

Nous avons donc affecté à des variables `nombre1`, `nombre2` et `var1` des nombres et avons appelé la fonction `type` qui nous précise le type de ces variables ; le type `float` désigne les flottants ou réels et le type `int` désigne les entiers *(Integer en anglais)*
Faisons, maintenant, des opérations avec ces nombres

```
>>> nombre1=12
>>> nombre2=15
>>> var1= 2
>>> resultat=(nombre1/nombre2)*var1
>>> print(resultat)
1.6
>>> type(nombre1)
<class 'int'>
>>> type(resultat)
<class 'float'>
```

Remarque 9 :

Une remarque sur les conversions :

1. On peut convertir un naturel en réel : cette opération ne fait pas perdre d'information : ainsi, 15 deviendra 15.0

   ```
   >>> nombre1=12
   >>> type(nombre1)
    <class 'int'>
   >>> nombre1=float(nombre1)
   >>> nombre1
    12.0
   ```

2. Par contre convertir un réel en entier fait perdre de l'information : 15.75 deviendra 15

   ```
   >>> Z=15.75
   >>> Z=int(Z)
   >>> Z
   15
   ```

3. D'autre part, en `Python`, si nous additionnons un réel à un entier, nous obtenons un réel. Exemple :

   ```
   >>> Z=15.75
   >>> type(Z)
   <class 'float'>
   >>> Y=225
   >>> type(Y)
   <class 'int'>
   >>> X=Y+Z
   >>> X
   240.75
   >>> type(X)
    <class 'float'>
   ```

Exercice 2 :

Petit exercice : prévoyez le résultat affiché dans les opérations réalisées en `Python`

1. $\dfrac{2\times 3}{2+1}$
2. $2\times\dfrac{3}{2}+1$
3. $2^{(3^4)}$

Exercice 3 :

Que fait cet algorithme si nous lui demandons d'imprimer B ?

```
>>> A = input('Donner une valeur à A: ')
>>> A = int(A)
>>> B=A*A
>>> B=B*B
>>> B=B*A
>>> B=B*B
>>> print("Le résultat donne ",B)
```

Remarque 10 :

En `Python`, il est possible d'utiliser des opérateurs pour une variable de type caractère ; ce type s'appelle, en `Python`, `str` *(De l'anglais « String » caractère).*

1. **Longueur**
   `len` est la fonction qui donne la longueur d'une chaîne de caractères.

   ```
   >>> Caractere="Chaine de caractères"
   >>> Caractere
   'Chaine de caractères'
   >>> type(Caractere)
   <class 'str'>
   >>> len(Caractere)
   20
   ```

2. **Concaténation**
   + est l'opérateur de concaténation

   ```
   >>> S1="Bonjour"
   >>> S2=" tout le monde"
   >>> S3=S1+S2
   >>> S3
   'Bonjour tout le monde'
   ```

3. **Répétition**
   * est l'opérateur de répétition

   ```
   >>> S4=S3*3
   >>> S4
   'Bonjour tout le mondeBonjour tout le mondeBonjour tout le monde'
   ```

Remarque 11 :

On ne peut pas faire n'importe quoi avec les types ! Par exemple, il est impossible d'additionner ou concaténer des variables de différents types.

1. Dans le premier exemple, nous additionnons un entier `Nombre` à une variable de type caractère `"Les résidents"`, et nous obtenons un message d'erreur

   ```
   >>> Nombre= 6
   >>> type(Nombre)
   <class 'int'>
   >>> Expression = "Les résidents" + Nombre
   ```

```
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    Expression = "Les résidents" + Nombre
TypeError: can only concatenate str (not "int") to str
```

2. Que faire ? ? ...On peut transformer la variable entière `Nombre` en une variable caractère `str`

```
>>> Nombre= 6
>>> Nombre=str(Nombre)
>>> Expression = "Les résidents" + Nombre
>>> Expression
'Les résidents6'
```

1.3.2 Le type booléen

Les variables de type booléen utilisées dans les algorithmes ont, comme domaine usuel un ensemble formé de deux seules valeurs : {**VRAI**, **FAUX**}

Remarque 12 :

1. Les opérateurs admissibles sur les variables de ce type, sont celles que l'on trouve dans la logique : **ET**, **OU**, **NON**, dont il faut aller voir les tables de vérité dans le cours de mathématiques
2. En `Python` les valeurs affichées pour les valeurs prises par les variables booléennes sont {**TRUE**, **FALSE**}

Exemple 5 :

Il est intéressant de faire cet algorithme, pas à pas, sur le papier.

```
>>> booleen1=6<5
>>> booleen1
False
>>> booleen2 = not( 3>8)
>>> booleen2
True
>>> booleen3 =(5<6) or (3>8)
>>> booleen3
True
>>> booleen4 = (5<6) and (3>8)
>>> booleen4
False
```

1.4 Les fonctions d'entrée et de sortie

Rappelons ici, quelques fonctions que nous trouvons dans `Python`

1.4.1 La fonction de saisie

L'instruction de saisie de données par l'utilisateur est input()
L'exécution de cette instruction consiste à :

1. Demander à l'utilisateur de saisir une valeur sur le périphérique d'entrée ;
2. Modifier la valeur passée entre parenthèses

Remarque 13 :

Il s'agit de réaliser une saisie au clavier : la fonction standard `input()` interrompt le programme, affiche une éventuelle invite à l'écran et attend que l'utilisateur entre une donnée au clavier *(affichée à l'écran)* et la valide par ENTRÉE
La fonction `input()` effectue **toujours une saisie en mode texte** *(la valeur retournée est une chaîne)* dont on peut ensuite changer le type (on dit aussi « transtyper » ou « cast » en anglais) :

1. Nous voulons, ici, que la variable soit entière :

```
>>> x = int(input('Entrer un entier : '))
Entrer un entier : 26.3
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
  x = int(input('Entrer un entier : '))
ValueError: invalid literal for int() with base 10: '26.3'
>>> x = int(input('Entrer un entier : '))
Entrer un entier : 12
>>> x
12
```

2. Nous souhaitons, ici, cette fois-ci que la variable soit réelle

```
>>> y = float(input('Entrer un réel : '))
Entrer un réel : 25
>>> y
25.0
```

Quelque part, nous forçons la variable à être entière ou flottante

1.4.2 La fonction Afficher

L'instruction d'affichage de données par l'utilisateur est print

Exemple 6 :

1. Afficher une chaîne de caractères :

```
>>>print("chaîne de caractères")
chaîne de caractères
```

2. Afficher le contenu d'une variable

```
>>> A=5
>>> B=2.5
>>> Chaine="Bonjour"
>>> print("A= ",A,"B= ",B, "Chaine= ",Chaine)
A=  5 B=  2.5 Chaine=  Bonjour
```

1.5 Exercices

RÉSOUDRE CES EXERCICES NÉCESSITE UNE **PHASE D'ANALYSE** QUI CONSISTE À EXTRAIRE DE L'ÉNONCÉ DU PROBLÈME DES ÉLÉMENTS DE MODÉLISATION.
TECHNIQUE À UTILISER
DISTINGUER EN SOULIGNANT DE DIFFÉRENTES COULEURS :

- ⋆ QUEL EST LE BUT DU PROGRAMME *(traitement à réaliser)* ?
- ⋆ QUELLES SONT LES DONNÉES EN ENTRÉE DU PROBLÈME ?
- ⋆ OÙ VONT SE SITUER LES RÉSULTATS EN SORTIE ?

Exercice 4 :

Ecrire un programme qui donne la moyenne de 3 réels (*algorithme* `moyenne`)

Exercice 5 :

On considère 2 variables `var1` et `var2` ; échanger leur contenu (*algorithme* `echange`)

Il y a une particularité à Python, fort agréable qui est développée dans le corrigé

Exercice 6 :

On considère 3 variables `var1` et `var2` et `var3` ; recopier le contenu de façon circulaire, c'est à dire que le contenu de `var1` soit dans `var2`, celui de `var2` dans `var3`, et celui de `var3` dans `var1` (*algorithme* `circulaire`)

Exercice 7 :

Quel est le résultat de l'algorithme suivant :

```
# algo mystere_1

  >>>A=float(input("Donner A "))
  >>>B=float(input("Donner B ",))
  >>>A=A+B
  >>>  B=A-B
  >>>A=A-B
  >>>print("La valeur de A est ",A)
  >>>print("La valeur de B est ",B)
```

Exercice 8 :

Calculer le périmètre et la surface d'un rectangle sachant que la largeur notée « pc » et la longueur notée « gc » sont rentrées par l'utilisateur (*algorithme* `rectangle`)

Exercice 9 :

On souhaite calculer et afficher, à partir d'un prix hors taxe saisi, la TVA ainsi que le prix TTC d'un produit. Le montant TTC dépend du prix Hors Taxe et du taux de TVA de 20,6%

Exercice 10 :

On appelle **Quotient Familial** *(QF)* le rapport du total des revenus d'un ménage sur son nombre de parts, sachant que :

- Un adulte c'est 1 part
- Un enfant, une demie part
- Et les animaux 0 part

Calculez et afficher le quotient familial, sachant que l'utilisateur rentre le nombre d'adultes, le nombre d'enfants et le revenu total (*algorithme* `quotient familial`)

1.6 Correction des exercices

La correction des programmes se fait uniquement par les interpréteurs

Exercice 4 :

Ecrire un programme qui donne la moyenne de 3 réels

Il n'y a rien de particulier à dire sur cet exercice ; c'est l'utilisation basique des commandes vues dans l'exposé

```
>>> # Algorithme moyenne
       #cet algorithme calcule la moyenne de 3 nombres entrés au clavier
>>> Nb1=float(input("Entrer un premier nombre "))
Entrer un premier nombre 25
>>> Nb2=float(input("Entrer un second nombre "))
Entrer un second nombre 32.569
>>> Nb3=float(input("Entrer un troisième nombre "))
Entrer un troisième nombre -26.3
>>> moyenne = (Nb1 +Nb2+Nb3)/3
>>> moyenne
10.423
```

Exercice 5 :

On considère 2 variables `var1` *et* `var2` *échanger leur contenu*

Voici un exercice des plus classiques ; dans le langage `Python`, c'est très simplifié ; nous le verrons ci-après. L'intérêt de cet exercice est de le faire clairement, avec toutes les étapes possibles.

1. **En version classique**
   Classiquement, pour ne pas perdre de données, nous sommes obligés de passer par une « variable tampon »

   ```
   >>> var1 = float(input("Première variable "))
   Première variable 25
   >>> var2 = float(input("Seconde variable "))
   Seconde variable 69
   #Nous créons donc la variable tampon qui protège la seconde variable
   >>> temp = var2
   #On met la variable1 dans la variable 2
   >>> var2 =var1
   #On met la variable temporaire dans la variable 1
   >>> var1= temp
   #Et voilà le travail
   >>> var1
   69.0
   >>> var2
   25.0
   ```

   Nous aurions tout aussi bien pu échanger des variables de type caractère `str`

2. **En Python**
   En `python`, il existe une syntaxe très simple pour l'échange des variables : $\mathbf{A}, \mathbf{B} = \mathbf{B}, \mathbf{A}$, ce qui nous économise le passage à une variable temporaire. Nous le voyons sur un exemple :

   ```
   >>> var2=256
   >>> var1=162
   >>> var2,var1=var1,var2
   >>> var1
   256
   >>> var2
   162
   ```

Et un autre exemple, en utilisant des variables de types caratères :

```
>>> var1 ="D'amour"
>>> var2 = "Vos beaux yeux me font mourir"
>>> var2,var1 = var1,var2
>>> var1
'Vos beaux yeux me font mourir'
>>> var2
"D'amour"
```

Exercice 6 :

On considère 3 variables var1, var2 *et* var3

Recopier le contenu de façon circulaire, c'est à dire que le contenu de var1 *soit dans* var2, *celui de* var2 *dans* var3, *et celui de* var3 *dans* var1

1. **En version classique**

```
#algo circulaire
  #cet algorithme échange, de manière circulaire, le contenu de trois variables
            # le contenu des variables est fixé, ici, réel;
        #il pourrait être, de la même manière, entier ou caractère

  >>> var1 = float(input("Donner la première variable: "))
  Donner la première variable: 23.5
  >>> var2 = float(input("Donner la seconde variable: "))
  Donner la seconde variable: 120
  >>> var3 = float(input("Donner la troisième variable: "))
  Donner la troisième variable: 223
      temp:=var3;
        #on met var3 dans une variable "tampon",
        #car nous souhaitons la conserver.
      var3:=var2;
        #Le contenu de var2 est affecté à var3,
        #ce qui efface le contenu précédent de var3
      var2:=var1;
        #Le contenu de var1 est affecté à var2,
        #ce qui efface le contenu précédent de var2
      var1 := temp;
        #Le contenu de temp qui est le contenu précédent de var3 est affecté à var1,
        #ce qui efface le contenu précédent de var1
      >>> temp = var3
  >>> var3=var2
  >>> var2= var1
  >>> var1=temp
  >>> print("Le nouveau contenu de var1 est: ", var1)
  Le nouveau contenu de var1 est:  223.0
  >>> print("Le nouveau contenu de var2 est: ", var2)
  Le nouveau contenu de var2 est:  23.5
  >>> print("Le nouveau contenu de var3 est: ", var3)
  Le nouveau contenu de var3 est:  120.0
```

Cet algorithme apparaît comme une simple généralisation de l'algorithme échange

2. **En langage Python**

En version Python, nous généralisons l'exercice précédent

```
>>> var1=100
>>> var2=200
>>> var3=300
```

```
>>> var3,var1,var2=var2,var3,var1
>>> var1
300
>>> var2
100
>>> var3
200
```

Exercice 8 :

Calculer le périmètre et la surface d'un rectangle sachant que la largeur notée « pc » et la longueur notée « gc » sont rentrées par l'utilisateur

Voilà un exercice qui ne devrait poser aucune difficulté

```
#algo rectangle
  >>> pc=float(input("Donner la largeur du rectangle: "))
  Donner la largeur du rectangle: 25
  >>> gc=float(input("Donner la longueur du rectangle: "))
  Donner la longueur du rectangle: 35
  >>> p=2*(pc+gc)
  >>> s=pc*gc
  >>> print("Le périmètre du rectangle est ",p)
  Le périmètre du rectangle est  120.0
  >>> print("La surface du rectangle est ",s)
  La surface du rectangle est  875.0
```

Exercice 9 :

On souhaite calculer et afficher, à partir d'un prix hors taxe saisi, la TVA ainsi que le prix TTC d'un produit. Le montant TTC dépend du prix Hors Taxe et du taux de TVA de 20,6%

Voilà un nouvel exercice qui ne devrait poser aucune difficulté

```
#algo TVA
  >>> PrixHT=float(input("Quel est le prix Hors-Taxes? "))
  Quel est le prix Hors-Taxes? 500
  >>> PrixTTC=PrixHT*1.206
  >>> TVA=PrixHT*.206
  >>> print("Le prix TTC est ",PrixTTC)
  Le prix TTC est  603.0
  >>> print("La TVA est de ",TVA)
  La TVA est de  103.0
```

Exercice 10 :

On appelle Quotient Familial le rapport du total des revenus d'un ménage sur son nombre de parts, sachant que :

- *Un adulte c'est 1 part*
- *Un enfant, une demie part*
- *Et les animaux 0 part*

Calculez et afficher le quotient familial, sachant que l'utilisateur rentre le nombre d'adultes, le nombre d'enfants et le revenu total

```
# algo quotient_familial

  >>> adulte= float(input("Donner le nombre d'adultes: "))
  >>> enfant=float(input("Donner le nombre d'enfants: "))
  >>> revenu=float(input("Donner le revenu du foyer: "))
  >>> part=adulte+(enfant/2)
  >>> qf=(revenu/part)
```

```
>>> print("Le nombre de part du foyer est  ",part)
>>> print("Le quotient familial du foyer est ",qf)
```

Chapitre 2

Les structures de contrôle

Structures de contrôles ? ? Quel gros mot ! ! Disons que nous allons voir dans ce chapitre, tout ce qui concerne les tests... On avance ! !

2.1 Les instructions conditionnelles

2.1.1 Définition

L'instruction conditionnelle détermine si le bloc d'instructions suivant est exécuté ou non. La condition est une expression booléenne dont la valeur détermine le bloc d'instructions exécuté.

Remarque 1 :

1. La **syntaxe** de cette instruction est :

```
Si (condition) alors
  Bloc d'instruction N°1;
    #ce bloc est exécuté si (condition) est vrai
sinon
  Bloc d'instructions N°2;
    #ce bloc est exécuté si (condition) est faux
```

2. L'un des deux blocs est obligatoirement exécuté, l'autre ne l'est pas
3. En Python, nous utiliserons **if....else...** en utilisant les 2 points **:** et **l'indentation**

```
If (condition):
  Bloc d'instruction N°1;
    #ce bloc est exécuté si (condition) est vrai
else:
  Bloc d'instructions N°2;
    #ce bloc est exécuté si (condition) est faux
```

Exemple 1 :

1. **Exemple simple**

```
>>> a=-2
>>> if a>0:
        print("a est positif")
    else:
        print("a est négatif")

a est négatif
```

2. Ecrivons l'algorithme qui lit deux entiers et affiche le plus grand des deux

```
#algo MaxDeDeuxEntiers

  n1= float(input("Donner le premier entier "))
  n2= float(input("Donner le second entier "));
  if n1 >n2:
        max=n1
     else :
        max:=n2;
  print("Le plus grand des deux nombres est " ,max)
```

Exercice 1 :

Reprendre l'algorithme `MaxDeDeuxEntiers` en affichant les résultats dans l'ordre décroissant (*algorithme* `MaxDeDeuxEntiers_1`)

Remarque 2 :

Une version plus simple peut être utilisée si l'alternative n'a pas lieu.

Exemple 2 :

Donner la valeur absolue d'un réel.

```
# algo ValeurAbsolue

      x=float(input("Donner le réel: "))
      if x<0:
        x=-1*x
      print("La valeur absolue est ",x)
```

Attention à l'indentation!!

2.1.2 Instructions conditionnelles imbriquées

Il est possible d'imbriquer des blocs d'instructions conditionnelles les uns dans les autres.

Remarque 3 :

C'est à dire que nous pouvons faire un **if** dans un autre **if**

Exemple 3 :

1. ```
   a = 0
   if a > 0:
      print('positif')
   else:
      if a < 0:
         print('négatif') # deux tabulations !
      else:
         print('zéro')
   ```

2. Ecrivons, par exemple, un algorithme qui associe, à la note, un commentaire.

   ```
   #algo NotesCommentaires

         note=float(input("Donner la note: "))
   if note <= 8:
         print("la note ",note,"est très insuffisante")
   else:
         if note <=10:
   ```
   ```

```
                print("Lanote",note,"est très moyenne")
        else:
                if note <=12:
                        print("La note",note," est moyenne")
                else:
                        if note <= 16:
                                print("La note", note,"est bien")
                        else:
                                print("la note",note,"est très bien")
```

2.1.3 Important en Python

En python, « **else + if** » peut se simplifier en « **if + elif +else** »

Remarque 4 :

Remarque importante

1. La condition **if.....elif .... else** est une structure conditionnelle encore plus complète que la condition **if ..... else** qui va nous permettre cette fois-ci d'effectuer autant de tests que l'on souhaite et ainsi de prendre en compte le nombre de cas souhaité.

   En effet, nous allons pouvoir ajouter autant de **elif** que l'on souhaite entre le **if** de départ et le **else** de fin et chaque **elif** va pouvoir posséder son propre test ce qui va nous permettre d'apporter des réponses très précises à différentes situations.

2. Il faut cependant faire attention à un point en particulier lorsqu'on utilise une structure Python **if.....elif .... else** : le cas où plusieurs **elif** possèdent un test évalué à **True** par Python. Dans ce cas là, vous devez savoir que seul le code du premier **elif** (ou du **if** si celui-ci est évalué à **True**) va être exécuté. En effet, Python sort de la structure conditionnelle dans son ensemble sans même lire ni tester la fin de celle-ci dès qu'un cas de réussite à été rencontré et que son code a été exécuté.

3. **elif..** pourrait être interprété en langage courant comme « sinon si.. »

Exemple 4 :

Nous allons simplifier les exemples précédents

1. ```
   #-*-coding:Latin-1 -*
   #algo If_Elif_Simple_01

   a=float(input("Donner une valeur à  a "))
   if a > 0:
     print('positif')
   elif a < 0:
       print('négatif') # deux tabulations !
    else:
    print('zéro')

   #-*-coding:Latin-1 -*
   #algo NotesCommentaires

   note=float(input("Donner la note: "))
   if note <= 8:
      print("la note ",note,"est très insuffisante")
   elif note <=10:
      print("Lanote",note,"est très moyenne")
   elif note <=12:
      print("La note",note," est moyenne")
   elif note <= 16:
   ```
   ```

```
        print("La note", note,"est bien")
    else:
        print("la note",note,"est très bien")
```

Exercice 2 :

1. Ecrire un algorithme qui trie trois nombres réels n_1, n_2, n_3 et qui donne ce tri par ordre décroissant.(*algorithme* `TriDe3Nombres`)
2. Ecrire un algorithme qui résoud une équation du second degré à coefficients dans $\mathbb{R}$ (*algorithme* `SecondDegre`)

2.2 Instructions de répétition

2.2.1 La boucle while *(Tant que)*

L'instruction de répétition appelée boucle permet d'exécuter plusieurs fois consécutives un même bloc d'instructions. La répétition s'effectue tant que la valeur de l'expression booléenne reste vraie (ou est égale à TRUE)

Remarque 5 :

Syntaxe de la boucle `while`

```
tantque (condition booléenne ):
        instructions
```

La `condition booléenne` indique la condition de sortie de cette boucle qui est, en fait, la négation de l'expression booléenne, c'est à dire `non(condition booléenne)`

Exemple 5 :

L'algorithme suivant affiche les 5 premiers entiers, en utilisant la boucle `while`

```
#-*-coding:Latin-1 -*

#algo affichage_des_premiers_entiers

compteur=1 #initialisation du compteur
while compteur<=5 :
  print(compteur)
  compteur=1+compteur
```

Commentaires :
La variable `compteur` est initialisée **avant** la boucle.

Exercice 3 :

Ecrire un algorithme qui affiche les n premiers entiers, n étant choisi par l'opérateur. (*Algorithme* `affichageDesNPremiersEntiers1`)

Remarque 6 :

1. Il y a plusieurs algorithmes qui peuvent être équivalents. Par exemple, la condition `(compteur<=5)` est équivalente `(compteur<6)` ; il y a donc plusieurs algorithmes possibles.
2. Pour écrire une boucle, il faut :
   (a) **Chercher la condition d'arrêt**

(b) **Ecrire la négation de cette condition d'arrêt**, qui peut être tout aussi intéressante à utiliser.

3. Pour écrire une boucle, 3 étapes sont obligatoires :
   (a) Initialiser le compteur avant d'entrer dans la boucle
   (b) Etablir la condition de poursuite : il y a toujours des conditions de poursuite différentes, mais équivalentes
   (c) La modification d'une valeur dans la boucle *(celle que l'on a initialisée)*, pour que la répétition exprime une évolution des calculs

Remarque 7 :

Remarque importante
En `Python`, il y a une version originale de l'incrémentation qu'il est plus simple d'utiliser :

1. L'instruction

```
compteur = compteur +1
```

   c'est à dire qu'on incrémente 1 à la variable `compteur` est équivalente à

```
compteur += 1
```

2. De la même manière, l'instruction

```
compteur = compteur *5
```

   c'est à dire qu'on multiplie la variable `compteur` par 5 est équivalente à

```
compteur *= 5
```

Intéressant, non ?

2.2.2 Boucle et conditionnelle ; boucles imbriquées

On peut mélanger les différentes instructions de conditionelle et les boucles tout comme on peut faire des boucles imbriquées.

Exemple 6 :

1. Ce programme affiche le maximum de 5 nombres donnés par l'utilisateur

```
#-*-coding:Latin-1 -*
#algo MaxDe_5_Entiers

#Cet algorithme fait entrer 5 nombres en file et en affiche le plus grand des 5

valeur=float(input("Donner la première valeur: "))
compteur=1
max = valeur
while compteur<5 :
    valeur=float(input("Donner une autre valeur: "))
    if valeur>max :
                    max=valeur
    compteur+=1

print("Le plus grand des 5 nombres est",max);
```

2. L'algorithme suivant affiche la meilleure des 5 notes comprises entre 0 et 20 ; pour la programmation, attention à l'indentation ! !

```
#-*-coding:Latin-1 -*

#algo Note Max Entre 0 Et 20
#Cet algorithme donne la meilleure des 5 notes comprises entre 0 et 20



note= int(input("Donner une note entière entre 0 et 20 "))
while note<0 or note>20 :
  print("Vous avez fait une erreur")
  note= int(input("On recommence: donner une note entière entre 0 et 20 "))
maximum =note
compteur = 1
while compteur < 5 :
  note= int(input("Donner une note "))
  while note<0 or note>20 :
    print("Vous avez fait une erreur")
    note= int(input("Donner une note "))
  if note>=maximum :
    maximum=note
  compteur = 1 + compteur

print("La note maximale est ",maximum)
```

Exercice 4 :

Dans l'exemple précédent, nous avons l'instruction

```
while note<0 or note>20 :
```

Exprimer le test en utilisant la négation de cette condition

2.3 L'instruction *for* en Python

2.3.1 La boucle itérative for *(Pour)*

En Python, la boucle « for » ne fonctionne pas comme dans d'autres langages C ou Pascal.
Plutôt que de toujours itérer en suivant une progression arithmétique de nombres, c'est plus une **boucle de parcours** de listes ou de caractères qu'une boucle itérative.
Nous allons la travailler dans quelques exemples

Remarque 8 :

La syntaxe de la boucle `for` est donnée par :

```
for i in E:
    instructions
```

Exemple 7 :

```
words = ['cat', 'window', 'defenestrate']
for w in words:
        print(w)

cat
window
defenestrate
```

Exercice 5 :

Que fait ce petit programme ?

```
#-*-coding:Latin-1 -*

#Algorithme Mystère

sequence = "Bonjour les étudiants"

for lettre in sequence:
    if lettre in "AEIOUYaeéiyou":
        print (lettre)
    else:
        print ("*")
```

2.3.2 La fonction *range()*

La fonction *range*() permet à l'utilisateur de générer une série de nombres dans une plage donnée.
En fonction du nombre d'arguments que l'utilisateur transmet à la fonction, l'utilisateur peut décider où cette série de nombres commencera et se terminera ainsi que l'ampleur de la différence entre un nombre et le suivant.
range() prend principalement trois arguments :

1. **start** : c'est l'entier à partir duquel la séquence d'entiers doit être retournée ; c'est l'entier de départ
2. **stop** : entier avant lequel la séquence d'entiers doit être renvoyée. La plage d'entiers se termine à $stop - 1$.
3. **step** : c'est une valeur entière qui détermine l'incrément ou le pas entre chaque entier de la séquence.

Remarque 9 :

1. La fonction `range()` est utilisée pour générer une séquence de nombres.
2. `range()` est couramment utilisée pour la boucle, par conséquent, sa connaissance est un aspect clé lors du traitement de tout type de code `Python`. L'utilisation la plus courante de la fonction `range()` en Python est d'itérer le type de séquence *(Liste, chaîne, etc.)* avec les boucles `for` et `while`
3. **La syntaxe** est donnée par `range(start,stop,step)`, sachant que :
   (a) Il est possible de n'appeler qu'un seul argument `range(stop)` et nous avons comme entier de départ $n = 0$ (nous nous arrêtons à $stop - 1$; *(cf exemple 1 ci-dessous)*
   (b) Si nous avons 2 arguments `range(start,stop)` la liste part de l'entier start pour s'arrêter à $stop - 1$ *(exemple 2)*
   (c) Et pour 3 arguments `range(start,stop, step)` la liste part de l'entier start pour s'arrêter à $stop - 1$ en faisant des pas de longueur `step`*(exemple 3)*

Exemple 8 :

Des exemples

1.
```
for i in range(10):
    print(i)
```

Et nous avons comme affichage :

0 1 2 3 4 5 6 7 8 9

2.
```
for i in range(4,10):
    print(i)
```

Et nous avons comme affichage :

4 5 6 7 8 9

3.
```
for i in range(2,10,3):
    print(i)
```

Et nous avons comme affichage :

2 5 8

4.
```
somme = 0
for i in range(1, 11):
    somme += i
print("La somme des 10 premiers naturels est", somme)
```

Et nous trouvons **55**

5. Nous savons que $\lim\limits_{n\to+\infty}\sum\limits_{k=1}^{+\infty}\dfrac{1}{k^2}=\dfrac{\pi^2}{6}$, et, sans faire d'approximation de la limite, nous allons calculer $\sum\limits_{k=1}^{n}\dfrac{1}{k^2}$ jusque l'entier n choisi par l'opérateur.

```
#-*-coding:Latin-1 -*
#algo Somme_Inverse_Carres
  #Cet algorithme calcule la somme des inverses des carrés jusqu'à un ordre n, arbitrair

n=int(input("Jusqu'à quel entier souhaitez vous aller? "))

Somme = 0
for i in range(1,n+1): #Nous allons de 1 à (n+1)-1=n
    Somme += 1/(i**2)
print("Au rang ", n, "la somme est ",Somme)
```

Exercice 6 :

Ecrire un programme qui affiche tous les nombres pairs entre 0 et n, dans l'ordre croissant, le nombre n étant donné par l'opérateur.

Exercice 7 :

Ecrire un script qui calcule $p!$ pour un $p \in \mathbb{N}$ donné par l'utilisateur

Exercice 8 :

Ecrire un script qui calcule les n premiers termes de la suite :

$$\begin{cases} u_0 \in \mathbb{R}^* \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{a}{u_n}\right) \text{ avec } a \in \mathbb{R}^{*+} \end{cases}$$

On admettra, pour cet exercice que pour tout $n \in \mathbb{N}$, $u_n \neq 0$. *(Cette suite a été étudiée en L_0)*

2.4 Exercices

Exercice 9 :

Une assurance propose 3 tarifs selon l'âge et le nombre d'accidents des automobilistes :

1. Le tarif Vert
2. Le tarif Orange
3. Le tarif Rouge

	Moins de 25 ans	**25 ans et plus**
0 accident	Orange	vert
1 ou 2 accidents	Rouge	Orange
3 à 6 accidents	Pas assuré	Rouge
7 accidents ou plus	Pas assuré	Pas assuré

Ecrire un programme qui affiche le tarif après avoir saisi l'âge et le nombre d'accidents de l'automobiliste

Exercice 10 :

Vous désirez comparer 2 offres d'abonnement téléphonique. La facture est déterminée avec une somme fixe à payer obligatoirement tous plus mois, plus une partie proportionnelle au temps passé à téléphoner *(indiqué en minutes)*

Offre	**Fixe**	**Prix à la minute**
Opérateur N° 1	10,00€	0,50 €
Opérateur N° 2	15,00 €	0,42 €

Ecrire l'algorithme qui indique l'opérateur le plus intéressant après avoir saisi la consommation individuelle mensuelle en minutes

Exercice 11 :

Compléter le tableau suivant représentant la correspondance entre les conditions de continuité et les conditions d'arrêt

Condition d'arrêt	**Condition de continuité**
$(nb = 4)$ **ET** $(age < 25)$	
$(de = 6)$ **ET** $(nbDeCoup > 5)$	
$((de1 = 6)$ ET $(de2 = 5))$**OU** $(nbDeCoup > 5)$	
$(de1 = 6)$ **OU** $(de2 = 6)$	

Exercice 12 :

Quelles sont les valeurs affichées par le programme `mystere_2` suivant ?

```
#-*-coding:Latin-1 -*


#algo Mystere_2

I=5
S=2
while(I<=31):
  print("I=",I)
  print ("S=",S)
  I=I+S
  S=6-S
```

Exercice 13 :

Le programme `mystere_3` imprime une suite d'entiers naturels. Définir cette suite.

```
#-*-coding:Latin-1 -*
```

```
#algo mystere_3

N=1
S=0
while (N<=19):
        S=S+N
        print("S= ",S)
        N=2+N
```

Exercice 14 :

Le programme `mystere_4` imprime une suite de nombres réels. Définir cette suite.

```
#-*-coding:Latin-1 -*


#algo mystere_4

N=0
C=0
A=1
B=1
D=6
while(N<=10):
  print("La valeur de A est ",A)
  N=N+1
  C=C+D
  B=B+C
  A=A+B
```

Exercice 15 :

Une file d'entiers est toujours terminée par -1. Elaborer l'algorithme qui compte le nombre d'occurences du premier entier de la file.

Exercice 16 :

Calculer le minimum et le maximum d'une suite de nombres terminée par -1

Exercice 17 :

Calculez la somme et la moyenne d'une suite de notes terminées par -1 ; ces notes devant être comprises entre 0 et 20.(*algorithme* `SommeMoyenne`)

Exercice 18 :

Ecrire le programme complet d'une petite application qui affiche à l'écran un triangle rectangle isocèle rempli d'étoiles et dont les côtés de ce triangle est de longueur N étoiles (N étant saisi par l'utilisateur).(*algorithme* `Triangle`)

Exercice 19 :

La conjecture de Syracuse
Ecrire le programme qui calcule les valeurs successives de cette suite

1. Prendre comme valeur initiale un naturel A.
2. Si $A = 1$ alors **STOP**

3. Si A est pair, remplacer A par $\dfrac{A}{2}$ et aller en 2).
4. Si Aest impair, remplacer A par $3A+1$ et aller en 2)

Exercice 20 :

Voici un algorithme définissant une suite :

1. Choisir un entier naturel A quelconque comme valeur initiale
2. Si $A = 4$, alors **STOP**
3. Si A se termine par le chiffre 4, barrer ce chiffre et aller en 2
4. Si A se termine par le chiffre 0, barrer ce chiffre et aller en 2
5. Mutiplier A par 2 et aller en 2

Ecrire le programme en `Python` de calcul des termes de la suite. Essayer différentes valeurs de A ; la valeur $A = 1249$ est particulièrement intéressante.

2.5 Correction des exercices

Exercice 1 :

Reprendre l'algorithme `MaxDeDeuxEntiers` *en affichant les résultats dans l'ordre décroissant*

```
#-*-coding:Latin-1 -*
#algo MaxDeDeuxEntiers_1

n1=float(input("Donner le premier entier "))
n2=float(input("Donner le second entier "))
if n1>n2:
  max=n1
  min=n2
else:
  max=n2
  min=n1
print("Le plus grand des deux est ",max)
print("Le plus petit des deux est ",min)
```

Il y a une façon plus « Pythonnique » d'écrire ce programme :

```
#-*-coding:Latin-1 -*
#algo MaxDeDeuxEntiers_1

n1=float(input("Donner le premier entier "))
n2=float(input("Donner le second entier "))
if n1>n2:
  max, min=n1, n2
else:
  max , min =n2,n1

print("Le plus grand des deux est ",max)
print("Le plus petit des deux est ",min)
```

Exercice 2 :

1. *Ecrire un algorithme qui trie trois nombres réels n_1, n_2, n_3 et qui donne ce tri par ordre décroissant.*

```
#-*-coding:Latin-1 -*
#algo TriDe3Nombres
  #Cet algorithme trie 3 nombres réels et les présente sous forme décroissante


x1= float(input("Donner le premier réel: "))
x2= float(input("Donner le second réel: "))
x3= float(input("Donner le troisième réel: "))
      # On commence le traitement des données
if x1>=x2:
    if x1>=x3:
        print("Le plus grand des 3 nombres est x1 ",x1)
        if x2>=x3:
            print("Le nombre moyen est x2 ",x2)
            print("Le petit nombre est x3 ",x3)
        else:
            print("Le nombre moyen est x3 ",x3)
            print("Le petit nombre est x2 ",x2)
    else:
        print("Le plus grand des 3 nombres est x3 ",x3)
        print("Le nombre moyen est x1 ",x1)
        print("Le plus petit des 3 nombres est x2 ",x2)
```

```
elif x2>=x3:
    print("Le plus grand des 3 nombres est x2 ",x2)
    if x1>=x3:
        print("Le nombre moyen est x1 ",x1)
    else:
        print("Le plus petit nombre est x3 ",x3)
else:
    print("Le plus grand nombre est x3 " , x3)
    print("Le nombre moyen est x2 ",x2)
    print("Le plus petit nombre est x1 ",x1)
```

2. *Ecrire un algorithme qui résoud une équation du second degré à coefficients dans* $\mathbb{R}$

   Nous allons, dans un premier temps, analyser le problème en réalisant un « arbre de décision » :

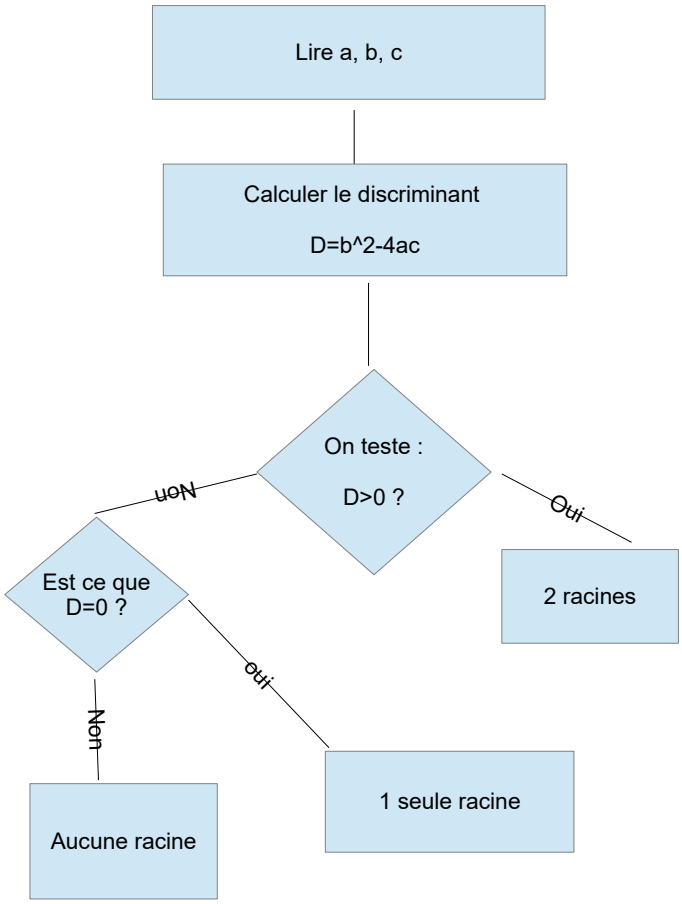

FIGURE 2.1 – L'organigramme de résolution d'une équation du second degré

```
#-*-coding:Latin-1 -*
# Algorithme de résolution d'une équation du second degré


import math #nécessité d'importer le module math:
            #nous en aurons besoin pour calculer la racine carrée
```

```
a=float(input ("Saisissez la valeur de \'a\'  "))
# l'utilisateur entre la valeur de a, le coefficient du second degré
b=float(input("saisissez la valeur de \'b \' "))
# l'utilisateur entre la valeur de b, le coefficient du terme du premier degré
c=float(input("saisissez la valeur de \'c \' "))
# l'utilisateur entre la valeur de c, le terme constant
if a==0 and b==0 and c==0:          #dans le cas où a=b=c=0
  print ("a, b, c sont égaux à 0;polynôme nul, tous les réels sont racines de l'équation
elif a==0 and b==0 and not(c==0):
  print("ce n'est pas un polynôme du second degré")
elif a==0 and not(b==0):
  S=-(c/b)
  print("Nous sommes devant un polynôme du premier degré la solution est S=",S)
elif not(a==0) and b==0 and c==0:
  print("L'équation est du type ax^2 =0; la seule solution est S=0")
elif not(a==0) and not(b==0):
  D=(b**2)-4*(a*c)
  if D<0:
    print("Le discriminant est négatif: pas de racine")
  elif D==0:
    S=-(b/(2*a))
    print("Le discriminant est nul; Il y a donc une seule solution S=",S)
  else:
    S1=(-b+math.sqrt(D))/(2*a)
    S2=(-b-math.sqrt(D))/(2*a)
    print("Le discrimant est positif; il y a 2 solutions \n")
    print("La première solution est S1= ",S1)
    print("La seconde solution est S2= ",S2)
```

Nous avons importé, ici, le module "math". Nous reviendrons, un peu plus loin sur les imports de modules

Exercice 3 :

Ecrire un algorithme qui affiche les n premiers entiers, n étant choisi par l'opérateur.

Exercice pas très difficile ; juste une adaptation de celui vu dans le cours!!!!

```
#-*-coding:Latin-1 -*

#algo affichage_des_N_premiers_entiers

n=int(input("Quel est le plus grand entier à afficher "))
compteur=1 #initialisation du compteur
while compteur<=n :
  print(compteur)
  compteur=1+compteur
```

Exercice 6 :

Ecrire un programme qui affiche tous les nombres pairs entre 0 et n, dans l'ordre croissant, le nombre n étant donné par l'opérateur.

```
#-*-coding:Latin-1 -*



#algo nombres pairs

n = int(input("Donner  le nombre jusqu'où vous souhaitez aller "))
```

```
for i in range(0,n+1,2):
    print(i)
```

Exercice 7 :

Ecrire un script qui calcule $p!$ pour un $p \in \mathbb{N}$ donné par l'utilisateur

```
#-*-coding:Latin-1 -*


#algo Factorielle

n = int(input("Donner l'entier dont on veut connaître la factorielle "))
p=1
for i in range(1,n+1):
  p=p*i

print(p)
```

Ce petit script se rapproche d'une notion de récursivité que nous verrons plus loin

Exercice 8 :

Ecrire un script qui calcule les n premiers termes de la suite :

$$\begin{cases} u_0 \in \mathbb{R}^* \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{a}{u_n}\right) \text{ avec } a \in \mathbb{R}^* \end{cases}$$

C'est le calcul de la racine carrée par la méthode des babyloniens. On remarque que si $u_0 < 0$, la suite converge vers $-\sqrt{a}$

```
#-*-coding:Latin-1 -*


#algo Racine_Carree

n = int(input("Donner le nombre de termes de la suite que vous souhaitez imprimer "))
a = float(input("Donner le réel strictement positif dont vous souhaitez calculer la racine ")
u0 = float(input("Donner le premier terme de la suite "))
while a<0:
  a = float(input("On a dit un réel strictement positif!!. "))
for i in range(1,n+1):
  u0 = (u0 + (a/u0))/2
  print("Pour n= ",i,"u= ",u0)
```

Exercice 8 :

Une assurance propose 3 tarifs selon l'âge et le nombre d'accidents des automobilistes :

1. *Le tarif Vert*
2. *Le tarif Orange*
3. *Le tarif Rouge*

	Moins de 25 ans	25 ans et plus
0 accident	Orange	vert
1 ou 2 accidents	Rouge	Orange
3 à 6 accidents	Pas assuré	Rouge
7 accidents ou plus	Pas assuré	Pas assuré

Ecrire un programme qui affiche le tarif après avoir saisi l'âge et le nombre d'accidents de l'automobiliste

Si je vous disais que ce n'est pas très difficile....Me croiriez vous ?

```
#-*-coding:Latin-1 -*


#algo Assurance_Accident


age= int(input("Quel est votre âge?  "))
accident= int(input("Combien avez vous eu d'accidents?  "))

if age <= 25:
   if accident == 0:
      print("Vous aurez un tarif Orange ")
   elif (accident == 1) or (accident == 2):
      print("Vous aurez un tarif rouge")
   else:
      print ("On n'assure pas")
else:
   if accident == 0:
      print("Vous aurez un tarif vert ")
   elif (accident == 1) or (accident == 2):
      print("Vous aurez un tarif orange")
   elif (accident >=3) and (accident <=6):
      print("Vous aurez un tarif rouge")
   else:
      print ("On n'assure pas")
```

Exercice 9 :

Vous désirez comparer 2 offres d'abonnement téléphonique. La facture est déterminée avec une somme fixe à payer obligatoirement tous plus mois, plus une partie proportionnelle au temps passé à téléphoner (indiqué en minutes)

Offre	Fixe	Prix à la minute
Opérateur N° 1	*10,00€*	*0,50 €*
Opérateur N° 2	*15,00 €*	*0,42 €*

Ecrire l'algorithme qui indique l'opérateur le plus intéressant après avoir saisi la consommation individuelle mensuelle en minutes

Cet exercice, bien que d'énoncé qui peut paraître long est de résolution très facile

```
#-*-coding:Latin-1 -*


#algo Operateur


minute= float(input("Combien de minutes?  "))
operateur1=10+0.5*minute
operateur2=15+0.42*minute
if operateur1 < operateur2 :
   print("C'est l'opérateur 1 qui est le plus intéressant")
elif operateur2 < operateur1 :
   print("C'est l'opérateur 2 qui est le plus intéressant")
else:
   print ("L'un ou l'autre, à votre choix")
```

Question : quand donc pouvons nous prendre l'un ou l'autre des opérateurs ?

Exercice 15 :

Une file d'entiers est toujours terminée par -1. Elaborer l'algorithme qui compte le nombre d'occurences du premier entier de la file.

```
#-*-coding:Latin-1 -*


#algo Occurences
#Cet algo compte le nombre d'occurence du premier entier dans une file


integ = int(input("Donner le premier entier "))

while integ == -1:
     print("Vous avez fait une erreur")
     integ = int(input("Donner le premier entier "))

premier = integ
compteur = 1
while integ != -1:
     integ = int(input("Donner l'entier suivant "))
     if integ == premier:
          compteur +=1


print("Le nombre d'occurences du premier entier ", premier," est ", compteur)
```

Exercice 16 :

Calculer le minimum et le maximum d'une suite de nombres terminée par -1

Cet algorithme est un raffinement du précédent

```
#-*-coding:Latin-1 -*



#algo OccurencesMaxMin
#Cet algo donne le maximum et le minimum d'une suite de nombres rÃ©els terminÃ©e par -1


integ = int(input("Donner le premier entier "))

while integ == -1:
     print("Vous avez fait une erreur")
     integ = int(input("Donner le premier entier "))
minimum = integ
maximum = integ
while integ !=-1:
    integ = int(input("Donner l'entier suivant "))
    if (integ >= maximum) and (integ !=-1):
        maximum = integ
    if (integ <= minimum) and (integ !=-1):
        minimum = integ



print("Le plus petit nombre de la file est:  " , minimum)
```

```
print("Le plus grand nombre de la file est:  " ,maximum)
```

Exercice 17 :

Calculez la somme et la moyenne d'une suite de notes terminées par -1 ; ces notes devant être comprises entre 0 et 20.

Pas exactement difficile!!.....Quoique!!

```
#-*-coding:Latin-1 -*

#algo SommeMoyenne
#Cet algo donne le maximum et le minimum d'une suite de nombres réels
#compris entre 0 et 20 terminée par -1


nombre= int(input("Donner le premier nombre "))#C'est l'initialisation
while nombre==-1 or nombre >20 or nombre<0:
    print("Vous avez fait une erreur, recommencez")
    nombre= int(input("Donner le premier nombre "))
somme = nombre
compteur = 1
#On continue la liste
while nombre != -1:
    nombre= int(input("Donner le nombre suivant "))
    while (nombre >20 or nombre<0)and nombre !=-1:
        nombre= int(input("Erreur: donner nombre entre 0 et 20 "))
    somme += nombre
    compteur+= 1


print("La somme est ",somme +1)
print("le compteur est ", compteur-1)
print("La moyenne est ", (somme+1)/(compteur-1))
```

Exercice 18 :

Ecrire le programme complet d'une petite application qui affiche à l'écran un triangle rectangle isocèle rempli d'étoiles et dont les côtés de ce triangle est de longueur N étoiles (N étant saisi par l'utilisateur).

```
#-*-coding:Latin-1 -*



#algo Triangle
n = int(input("Quel est la longueur du côté? "))
z = input("De quel caractère souhaitez vous remplir le triangle? ")
#rappel: la fonction input appelle un caractère de type string
for i in range(n+1):
    for j in range(i+1):
        print(z , end=""),
    print("\n")
```

Dans ce listing, on remarquera la syntaxe de retour à la ligne

Exercice 19 :

Ecrire le programme qui calcule les valeurs successives de cette suite

1. *Prendre comme valeur initiale un naturel A.*
2. *Si $A = 1$ alors* **STOP**

3. *Si A est pair, remplacer A par $\frac{A}{2}$ et aller en 2).*
4. *Si Aest impair, remplacer A par $3A+1$ et aller en 2)*

Le programme de calcul des termes de cette suite

```
#-*-coding:Latin-1 -*
#algo Distance Euclidienne

A=int(input("Donner le premier entier "))
print (A ,"\n")
i=1
while A !=1:
    if A%2==0:
        i+=1
        A=(A/2)
        print( A,"\n" )
    else:
        A=3*A+1
        i+=1
        print (A,"\n")
print("Le nombre de termes est ",i)
```

On remarquera que la suite s'arrête toujours par une suuite de termes appelée « cycle trivial » : 5, 16, 8, 4, 2 1.
Si le terme initial est 27, il faut 117 itérations pour terminer le programme et si le premier terme est 31 466 383, il en faut 706, la suite se terminant toujours par le cycle trivial.

La conjecture de Syracuse

La conjecture de Syracuse est une conjecture mathématique qui reste improuvée à ce jour. Jusqu'à présent, la conjecture de Syracuse, selon laquelle depuis n'importe quel entier positif la suite de Syracuse atteint 1, n'a pas été mise en défaut.
Par exemple, les premiers éléments de la suite de Syracuse pour un entier de départ de 10 sont : 10, 5, 16, 8, 4, 2, 1 . . .

Pour en savoir plus sur la suite de Syracuse, consultez la page Wikipedia de la suite de Syracuse

Le programme ci-après donne seulement le nombre d'itérations nécessaires pour que la suite arrive à 1

```
#-*-coding:Latin-1 -*
#algo Distance Euclidienne

A=int(input("Donner le premier entier "))
print (A)
i=1
while A !=1:
    if A%2==0:
        i+=1
        A=(A/2)
    else:
        A=3*A+1
        i+=1

print("Le nombre d'itérations est ",i)
```

Exercice 20 :

Voici un algorithme définissant une suite :

1. *Choisir un entier naturel A quelconque comme valeur initiale*
2. *Si $A = 4$, alors* **STOP**

3. *Si A se termine par le chiffre 4, barrer ce chiffre et aller en 2*
4. *Si A se termine par le chiffre 0, barrer ce chiffre et aller en 2*
5. *Mutiplier A par 2 et aller en 2*

Ecrire le programme de calcul des termes de la suite. Essayer différentes valeurs de A ; la valeur $A = 1249$ est particulièrement intéressante.

Cet exercice ne pose pas de difficultés

```
#-*-coding:Latin-1 -*
#algo Suite Amelioré

A=int(input("Donner le premier entier A "))
print (A )
i=1
while A !=4:
    if A%4==0 or A%10==0:
        i+=1
        A=(A-(A%10))/10
        print( A)
    else:
        A=2*A
        i+=1
        print (A)
print("Le nombre de termes est ",i)
```

Chapitre 3

Les Fonctions en Python

3.1 Les fonctions

3.1.1 Définition de fonction

Une fonction est un algorithme indépendant.
L'appel (avec ou sans paramètre) de la fonction déclenche l'exécution de son bloc d'instructions.
Une fonction se termine en retournant ou non une valeur

Remarque 1 :

1. Une fonction a aussi été appelée **sous-programme**
2. Nous avons déjà rencontré des fonctions comme, par exemple `print()`, `type()` ou `input()`
3. La structure d'une fonction est la suivante :

```
def Nom_De_La_Fonction(parametrel, parametre2, parametre3,....,parametreN):
        Bloc d'instructiors
```

Si nous décortiquons la définition de la fonction, on trouve dans l'ordre :
→ `def`, mot-clé qui est l'abréviation du « define » *(définir, en anglais)* et qui est le prélude à toute construction de fonction.
→ Le nom de Ia fonction, qui se nomme exactement comme une variable *(nous verrons par la suite que ce n'est pas par hasard)*. Par conséquent, n'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
→ La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses fermante *(là encore, les espaces sont optionnels mais améliorent la lisibilité)*
Les deux points, encore et toujours, qui clôturent la ligne.
→ Les **parenthèses sont obligatoires,** quand bien même votre fonction n'attendrait aucun paramètre.

4. Trois étapes sont toujours nécessaires à l'exécution d'une fonction :
   (a) Le programme appelant interrompt son exécution
   (b) La fonction appelée effectue son bloc d'instructions.
   (c) Une fois le bloc d'instructions de la fonction terminé,le programme appelant reprend alors son exécution

Remarque 2 :

1. Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement quelque chose.
   L'algorithme utilisé au sein de la fonction n'intéresse pas directement l'utilisateur.

Par exemple, il est inutile de savoir comment la fonction `math.cos()` calcule un cosinus. On a juste besoin de savoir qu'il faut lui passer en argument un angle en radian et qu'elle renvoie le cosinus de cet angle. Ce qui se passe à l'intérieur de la fonction ne regarde que le programmeur.

2. Chaque fonction effectue, en général, une tâche unique et précise. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions *(qui peuvent éventuellement s'appeler les unes les autres)*. Cette **modularité** améliore la qualité générale et la lisibilité du code. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité.
3. Le programmeur doit penser à concevoir et écrire des fonctions pour améliorer son programme. Il y gagnera sur plusieurs points :
   — **Le code des algorithmes est plus simple, plus clair et plus court.** Dans un algorithme, appeler une fonction se fait en une ligne et la fonction peut être appelée à plusieurs reprises
   — Une seule modification dans la fonction sera automatiquement répercutée sur tous les algorithmes qui utilisent cette fonction.
   — L'utilisation de fonctions génériques dans des algorithmes différents permet de réutiliser son travail et de gagner du temps

Exemple 1 :

Le premier exemple d'utilisation de fonction : somme et maximum de deux réels : une fonction ramène la somme de deux réels, une autre le plus grand des 2

1. **Fonction somme de 2 réels**

```
#-*-coding:Latin-1 -*



#algo Fonction somme de 2 réels

def somme(x,y):
    x=float(x)
    y=float(y)
    s=x+y
    print("La somme de ",x,"et ",y ," est ", s)
```

Et à l'appel de cette fonction, nous avons :

```
>>> somme(15,16)
La somme de  15.0 et  16.0  est  31.0
>>> somme(265,-256)
La somme de  265.0 et  -256.0  est  9.0
>>>
```

2. **Fonction maximum de 2 réels**

```
#-*-coding:Latin-1 -*



#algo Fonction maximum de 2 réels

def maximum(x,y):
    if (x>y):
        grd = x
    else:
        grd = y
    print("Le plus grand des 2 entiers ",x," et ",y," est", grd)
```

Et, comme tout à l'heure, à l'appel de cette fonction, nous avons :

```
>>> maximum(25,63)
Le plus grand des 2 entiers  25  et  63  est 63
>>> maximum(-96,-100)
Le plus grand des 2 entiers  -96  et  -100  est -96
>>>
```

3.1.2 Définir une fonction

La signature d'une fonction décrit les éléments permettant de l'appeler correctement
En Python, la signature d'une fonction est seulement donnée par le nom de la fonction

Remarque 3 :

1. **Attention**, dans d'autres langages comme C++ ou Java la signature est beaucoup plus importante ; elle est donnée par :
   — Le nom de la fonction
   — Le type des paramètres
   — Le type de la valeur retournée
   Ainsi, dans ces langages, deux fonctions différentes peuvent avoir le même nom, mais des types de variables différents ; ce seront donc des fonctions différentes.
2. En Python, il est impossible de définir 2 fonctions avec le même nom : si vous le faites, l'ancienne fonction est écrasée par la nouvelle

Exemple 2 :

Nous vous proposons, ici, un exemple où une fonction écrase une autre. Nous vérifions, en plus que l'adresse de stockage est identique

```
>>> def exemple():
        print("fonction à écraser")


>>> exemple
<function exemple at 0x00000265F90F2708>
#Adresse de stockage de la fonction exemple()
>>> exemple()
fonction à écraser
>>> def exemple():
        print("Fonction qui écrase")


>>> exemple()
Fonction qui écrase
>>> exemple
<function exemple at 0x00000265F90F25E8>
#C'est bien la même adresse de stockage de la fonction exemple()
>>>
```

Exercice 1 :

Ecrire un programme qui affiche tous les nombres impairs entre 0 et n, dans l'ordre croissant, le nombre n étant donné par l'opérateur

3.1.3 Renvoi de résultats : la fonction « *return* »

L'instruction return signifie qu'on va renvoyer une valeur, pour pouvoir la récupérer ensuite et la stocker.
Cette instruction arrête le déroulement de la fonction, le code situé après le return ne s'exécutera pas.

Remarque 4 :

Jusqu'à présent, nos fonctions n'ont fait qu'afficher leur résultat après qu'on les ait appelées.
En pratique, cette façon de procéder est rarement utilisée et ceci pour deux raisons :

→ D'une part, nous n'avons aucun contrôle sur le résultat affiché puisque celui ci est affiché dès que la fonction a fini de s'exécuter

→ Ensuite car nous ne pouvons pas utiliser ce résultat pour effectuer de nouvelles opérations dans la suite

Or, en programmation, nous voudrons souvent récupérer le résultat d'une fonction afin de le ré-utiliser dans le reste de notre script. Pour cela, il va falloir qu'on demande à notre fonction de retourner *(renvoyer)* le résultat de ses opérations. Nous allons pouvoir faire cela en `Python` grâce à l'instruction `return`.
Attention cependant : l'instruction `return` va terminer l'exécution d'une fonction, ce qui signifie qu'on placera généralement cette instruction en fin de fonction puisque le code suivant une instruction `return` dans une fonction ne sera jamais lu ni exécuté.

Exemple 3 :

```
>>> def carre_cube(x):
        return x**2, x**3
>>> carre_cube(2)
(4, 8)
# Affectation multiple
>>> z1,z2 = carre_cube(7)
>>> z1
49
>>> z2
343
>>>
```

Exercice 2 :

Ecrire la fonction qui prend en entrée la température en degrés Celsius et la convertit en degré Farheneit.
La formule de conversion est donnée par : $C° = (F° - 32) \times \frac{5}{9}$

3.1.4 Fonction et arguments positionnels

Lorsqu'on définit une fonction def fct(x, y) : les arguments x et y sont appelés arguments positionnels *(en anglais positional arguments)*.
Il est strictement obligatoire de les préciser lors de l'appel de la fonction.
De plus, il est nécessaire de respecter le même ordre lors de l'appel que dans la définition de la fonction.

Exemple 4 :

Cet exemple montre l'importance du positionnement. Nous faisons ici le calcul de la fonction $f(x, y) = xy^2$

```
  def fois(x,y):
    return x*(y**2)

fois(2,1)
2
fois(1,2)
4
```

```
  def fois(x,y):
    return x*(y**2)

fois(2)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    fois(2)
TypeError: fois() missing 1 required
       positional argument: 'y'
```

On constate que passer un seul argument à une fonction qui en attend deux conduit à une erreur.

3.1.5 Fonction et arguments par mots clefs

Un argument défini avec une syntaxe def fct(arg=val) : est appelé argument par mot-clé *(en anglais keyword argument)*.
Le passage d'un tel argument lors de l'appel de la fonction est facultatif.

Exemple 5 :

1. Les arguments par mot-clé sont pris dans l'ordre dans lesquels on les passe lors de l'appel. Comment pourrions-nous faire si on souhaitait préciser l'argument par mot-clé z et garder les valeurs de x et y par défaut ? Simplement en précisant le nom de l'argument lors de l'appel :

```
def fct(x=0, y=0, z=0):
    return x, y, z

fct()
(0, 0, 0)
fct(15,23,5)
(15, 23, 5)
fct(z=5)
(0,0,5)
def fct(x=0, y=0, z=0):
    return x, y+x, z+y+x

fct()
(0, 0, 0)
fct(15,23,5)
(15, 38, 43)
fct(z=12,y=-5,x=5)
(5, 0, 12)
```

2. Que se passe-t-il lorsque nous avons un mélange d'arguments positionnels et par mot-clé ? Et bien les arguments positionnels doivent toujours être placés avant les arguments par mot-clé :

```
def fct(a, b, x=0, y=0, z=0):
    return a, b, a+x, b+y, a-z

fct(2,3,z=3,x=4,y=5)
(2, 3, 6, 8, -1)
fct(1,5)
(1, 5, 1, 5, 1)
fct(z=1)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    fct(z=1)
TypeError: fct() missing 2 required positional arguments: 'a' and 'b'
```

   On peut toujours passer les arguments par mot-clé dans un ordre arbitraire à partir du moment où on précise leur nom. Par contre, si les deux arguments positionnels a et b ne sont pas passés à la fonction, `Python` renvoie une erreur.

Exercice 3 :

Ecrire une fonction qui nous donne $n!$ *(sans utiliser la récursivité)*

Exercice 4 :

Calculez et affichez à l'écran 2^k avec k variant de 0 à n inclus.

Exercice 5 :

Créez une fonction qui calcule la distance euclidienne dans $\mathbb{R}^3$ entre 2 points $A\,(x_A, y_A, z_A)$ et $B\,(x_B, y_B, z_B)$

Exercice 6 :

Écrire une fonction `suite` qui, étant donnés deux arguments n et a renvoie la valeur de u_n, où la suite $(u_k)_{k\in\mathbb{N}}$ est la suite définie par :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{a}{u_n}\right) \end{cases}$$

C'est la suite des babyloniens.

3.1.6 Variables locales, variables globales

1. Une variable déclarée à la racine du module est une variable globale ; elle est donc dite globale lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme.
2. Une variable déclarée à l'intérieur d'une fonction est une variable locale ; elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.

Exemple 6 :

1. Un premier exemple très *(trop !)* simple

```
#-*-coding:Latin-1 -*
#algo Variable_Locale_Variable_Globale

x="bonjour"
def test(y):
        z=" tout le monde"
        print(y+z)
```

   Dans cet exemple, `x` est une variable globale et `y` est une variable locale

2. Par défaut, tout identificateur utilisé dans le corps d'une fonction est local à celle-ci.
   Par exemple :

```
#-*-coding:Latin-1 -*
#algo Variable_Locale_Variable_Globale_02


def fonc(y) : # y, x et z sont affectés dans fonc => locaux
    x = 3 # ce nouvel x est local et masque le x global
    z = x + y
    return z

x=26
fonc(x)
29
x #on demande d'imprimer la variable globale x
26
```

3. Autre exemple :

```
import math
def volume_boule(r):
    volume = 4/3 * math.pi * r**3
    return volume

volume_boule(3)
113.09733552923254

v=volume_boule(5)
print("Le volume de la boule de rayon 5 est ",v)
```

```
Le volume de la boule de rayon 5 est  523.5987755982989
print(volume)
#Réponse de l'ordinateur:
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print(volume)
NameError: name 'volume' is not defined
```

La variable `volume` est une variable locale, uniquement liée à la fonction. Elle n'existe pas dans le programme général

Exercice 7 :

Un cycliste parcourt 40 km la semaine 0. Il décide que, chaque semaine, il parcourra 5kms de plus que la distance parcourue lors de la semaine précédente. La fonction `seuil` présentée ci-dessous, écrite en langage `Python`, permet de déterminer le numéro de la semaine où la distance totale qu'il aura parcourue sera supérieure à un nombre n donné.

```
#-*-coding:Latin-1 -*
#algo Enonce CAPES 2023

def seuil(n) :
  k=0
  u=40
  S=40
  while S<n :
    k=k+1
    S=S+u+5
  return k
```

Que pensez vous de cette fonction ?

3.2 Modules

Dans cette nouvelle partie, nous allons découvrir une autre facette du langage Python qui en fait un langage à la fois très puissant, modulable et évolutif : l'utilisation de modules. Nous allons notamment étudier le fonctionnement de quelques modules prédéfinis qu'il convient de savoir manipuler.

3.2.1 Qu'est ce qu'un module ?

On appelle module tout fichier constitué de code Python (c'est-à-dire tout fichier avec l'extension .py) importé dans un autre fichier ou script.

Remarque 5 :

1. Les modules sont des programmes `Python` qui contiennent des fonctions que l'on est amené à réutiliser souvent *(on les appelle aussi **bibliothèques** ou **librairies**)*. Ce sont des « boîtes à outils » qui vont nous être très utiles.
2. Nous connaissons déjà un module que nous avons importé lors de la résolution d'une équation du second degré : le module `math`
3. Les modules permettent la séparation et donc une meilleure organisation du code. En effet, il est courant, dans un projet, de découper son code en différents fichiers qui vont contenir des parties cohérentes du programme final pour faciliter la compréhension générale du code, la maintenance et le travail d'équipe si on travaille à plusieurs sur le projet.

   Ainsi le module `math` contient-il les fonctions mathématiques
4. Il y a beaucoup de modules en `Python`. En `Python`, on peut distinguer trois grandes catégories de module en les classant selon leur éditeur :

⇒ Les modules standards qui ne font pas partie du langage en soi mais sont intégrés automatiquement par Python, comme le module `math` ;
⇒ Les modules développés par des développeurs externes qu'on va pouvoir utiliser, comme le module `matplotlib` ;
⇒ Les modules qu'on va développer nous mêmes.

5. Lorsque nous ouvrons l'interpréteur `Python`, les fonctionnalités du module `math` ne sont pas incluses ; il faudra les importer. Dans tous les cas, la procédure à suivre pour utiliser un mode sera la même.
   Nous allons commencer par décrire cette procédure puis nous étudierons dans la suite de cette partie quelques modules standards qu'il convient de connaitre.

3.2.2 Importer un module : la méthode *import*

Pour importer un module, on utilise la syntaxe import nom-de-mon-module.
Pour utiliser les éléments du module dans notre script, il faudra préfixer le nom de ces éléments par le nom du module et un point.

Remarque 6 :

Préfixer la fonction par le nom du module permet d'éviter les conflits dans le cas où on aurait défini des éléments de même nom que ceux disponibles dans le module. On parle **d'espace de nommage**

Exemple 7 :

Nous allons procéder à des exemples

1. On importe le module `math`

   ```
   import math
   ```

   La syntaxe est facile à retenir : le mot-clé `import`, qui signifie « importer » en anglais, suivi du nom du module, ici, `math`

2. Après l'exécution de cette instruction, rien ne se passe....*en apparence*. En réalité, `Python` vient d'importer tout le module `math`. Toutes les fonctions mathématiques contenues dans le module sont maintenant accessibles.
   Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point, puis du nom de la fonction. Voyons un exemple :

   ```
   >>> math.sqrt(16)
   4.0
   >>> math.sqrt(20)
   4.47213595499958
   >>>
   ```

   La fonction `sqrt` du module `math` renvoie la racine carrée du nombre inséré en argument

Remarque 7 :

1. Comment connaître toutes les fonctions existantes dans le module `math` ?
   C'est une fonction, la fonction `help` qui nous donnera l'information

   ```
   help (math)
   ```

   Ou encore aller chercher la documentation sur le net dans la documentation Python
2. Il est aussi possible de se documenter sur une fonction bien précise en utilisant la fonction `help`

   ```
   >>> help(math.sqrt)
   Help on built-in function sqrt in module math:

   sqrt(x, /)
   ```

```
        Return the square root of x.

    >>>
```

3.2.3 Espace de nom

Un espace de noms est un système permettant de s'assurer que tous les noms d'un programme sont uniques et peuvent être utilisés sans aucun conflit.
Il s'agit de regrouper certaines fonctions ou variables sous un préfixe spécifique. Les espaces de noms assurent que chaque nom utilisé dans un programme est unique.

Exemple 8 :

1. En vérité, lorsque vous tapez `import math`, cela crée **un espace de noms dénommé « math »**, contenant les variables et fonctions du module `math`.
   Quand vous tapez `math.sqrt(25)`, vous précisez à `Python` que vous souhaitez exécuter la fonction `sqrt` corrtenue dans 1'espace do noms `math`.
   Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonction `sqrt` que vous avez définie vous-même.
   Il n'y aura donc pas de conflit entre, d'une part, la fonction `sqrt` que nous avons créée et que vous appellerez grâce à l'instruction `sqrt` et, d'autre part, la fonction `sqrt` du module `math` que vous apellerez grâce à l'instruction `math.sqrt`
2. **Un exemple concret**
   On considère le (tout) petit script `Python` ci-après

   ```
   import math
   a = 5
   b= 3
   ```

   Dans l'espace de nom principal, celui que l'on utilise depuis le début et qui ne nécessite pas de préfixe, on trouve :
   ⇒ La variable `a`
   ⇒ La variable `b`
   ⇒ Le module `math` qui se trouve dans un espace de noms également appelé `math` dans lequel on trouve
   - ⋆ La fonction `sqrt`
   - ⋆ La variable `pi`
   - ⋆ Et bien d'autres fonctions et variables
3. L'intérêt des modules est de stocker à part variables et fonctions, dans un espace de nom, sans risque de conflit avec les variables ou fonctions que nous avons créées.
   Dans certains cas, nous pourrons modifier le nom de l'espace dans lequel le module importé sera stocké.

3.2.4 Créer un alias de nom pour un module

On peut utiliser le mot clef as pour renommer un module et créer un alias de nom.

Exemple 9 :

1. `import math as mathematiques`

   En faisant cet alias, nous importons le module `math` en spécifiant à `Python` de **l'héberger dans un espace de noms** appelé `mathematiques` au lieu de `math`. On peut penser que cela pourra permettre de mieux contrôler les espaces de noms des modules que nous importons.

2.
```
>>> import random as rand
>>> rand.randint(1, 10)
6
>>> rand.uniform(1, 3)
2.643472616544236
```

   Dans cet exemple, les fonctions du module `random` sont accessibles via l'alias `rand`.

Remarque 8 :

1. Cela peut permettre d'obtenir des scripts plus courts et plus clairs dans le cas où le nom du module est inutilement long ou s'il est peu descriptif.
2. On ne peut importer un module qu'une fois dans un script. Si vous testez le code ci-dessus et si vous aviez déjà importé le module précédemment, il faudra que vous quittiez l'interpréteur et que vous le relanciez pour que tout fonctionne.

3.2.5 Importer uniquement certains éléments d'un module

Si nous ne voulons n'importer que certains éléments dans un module, nous utilisons, pour cela l'instruction from nom-du-module import un-element.

Exemple 10 :

```
>>> from math import fabs
>>> fabs(-10.5)
10.5
>>> fabs(45)
45.0
>>>
```

Dans cet exemple, nous n'avons importé que la fonction `fabs` qui est la fonction valeur absolue.

Remarque 9 :

1. Lorsque nous importons uniquement une fonction, vous remarquez que nous ne mettons pas le préfixe `math.`
   En fait, par cette méthode, on met la fonction `fabs` dans l'espace de nom principal, au même niveau que la fonction `print` ou les variables que nous aurons créées.
2. Par la méthode :

   ```
   >>> from math import *
   >>> fabs(-10.5)
   10.5
   >>> sqrt(49)
   7.0
   >>>
   ```

   nous importons **toutes les fonctions** du module `math` dans l'espace de nommage principal

Exemple 11 :

Dans le script ci-après, nous importons 2 modules : les modules `math` et `random`.
Nous faisons calculer, dans ce script l'aire d'un cercle de rayon R où R est un entier défini de manière aléatoire entre 1 et 10

```
#-*-coding:Latin-1 -*


#algo Fonction Aire d'un cercle
#Exemple d'importation de modules
```

```
import math
import random

R = random.randint(1,10) #Fonction randint du domaine de nommage random
                         #Nous prenons un cercle de rayon un nombre entier aléatoire
                         #pris entre 1 et 10
S = math.pi *(R**2)

print("Le rayon est R= ",R,"et la surface est S= ",S)
```

Nous obtenons comme résultat :

```
Le rayon est R=  10 et la surface est S=  314.1592653589793
>>>
Le rayon est R=  3 et la surface est S=  28.274333882308138
>>>
Le rayon est R=  6 et la surface est S=  113.09733552923255
>>>
```

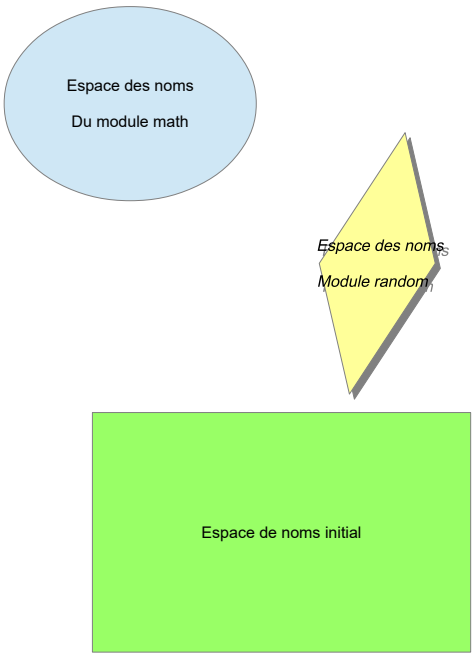

FIGURE 3.1 – Une modélisation des espaces de noms

3.2.6 Créer son propre module

Il est tout à fait possible de créer ses propres modules. Comment ?

1. On crée un fichier, par exemple `module.py` dans lequel nous mettons une ou deux fonctions
2. Dans un autre fichier exécutable, par exemple `test.py`, nous importons, ou tout le module, ou un fonction issue du module

Exemple 12 :

Exemple par un exercice résolu

Ecrire l'algorithme de calcul de maximum d'un ensemble de Max notes, le nombre de notes étant donné par l'opérateur. Le programme principal appelle une fonction qui se charge de vérifier la validité de la note (note comprise entre 0 et 20).

1. Nous commençons par créer une fonction et « l'enfermer » dans un module, c'est à dire un fichier que nous avons appelé `Essai_Module`

```
#-*-coding:Latin-1 -*

#algo controle de valeur

def  controle(x):
    f= (x>=0) and (x<= 20)
    return f
```

`f` est une variable booléenne qui n'a que 2 valeurs `True` ou `False`
Il est clair que nous pourrions réutiliser ce module pour un autre script `Python`. On appelle cela une **factorisation** du code.

2. Dans un second temps, nous écrivons le programme principal dans lequel nous importons le module nouvellement créé en lui donnant un alias.

```
#-*-coding:Latin-1 -*

#algo Note Maximale Entre 0 et 20
#Cet algorithme donne la meilleure des Max notes comprises entre 0 et 20

import Essai_Module as controle

Max =int(input("Combien de notes? "))
note= int(input("Donner une note entière entre 0 et 20 "))
while controle.controle(note) == False : #on utilise une fonction du module
  print("Vous avez fait une erreur")
  note= int(input("On recommence: donner une note entière entre 0 et 20 "))
maximum =note
compteur = 1
while compteur < Max :
  note= int(input("Donner une note "))
  while controle.controle(note) == False : #on utilise une nouvelle fois
                                           # une fonction du module
    print("Vous avez fait une erreur")
    note= int(input("Donner une note entre 0 et 20 "))
  if note>=maximum :
    maximum=note
  compteur += 1

print("La note maximale est ",maximum)
```

3.2.7 Les packages

Les packages sont des dossiers
Dans ces dossiers, on peut y trouver des fichiers (ou modules) ou d'autres dossiers (des packages)
C'est donc un mode de regroupement de modules

3.3 La récursivité en Python

La récursivité est un notion importante en programmation ; elle n'est pas réservée qu'au langage Python. Nous l'étudions donc ici avec le langage Python

3.3.1 Définition de fonction récursive

Une fonction est dite récursive si elle s'appelle elle même

3.3.2 Etude d'un exemple : la factorielle

Un premier exemple, très classique, et facile est la définition de factorielle qui est donnée par :

$$\begin{cases} 0! = 1 \\ 1! = 1 \\ n! = n \times (n-1)! \end{cases}$$

L'algorithme donne :

```
#-*-coding:Latin-1 -*


#algo Fonction factorielle avec recursivité


def factoriel(n):
    if  (n==0) or (n==1):
        return 1
    else:
        f= n*factoriel(n-1)
    return f
```

Et nous avons comme résultats :

```
>>> factoriel(0)
1
>>> factoriel(1)
1
>>> factoriel(10)
3628800
>>> factoriel(30)
265252859812191058636308480000000
>>>
```

Dans le script, nous aurions aussi pu écrire :

```
return n*factoriel(n-1)
```

Plutôt que

```
f= n*factoriel(n-1)
return f
```

Remarque 10 :

1. La récursivité est une technique de programmation très puissante : elle permet de trouver des solutions élégantes à des problèmes compliqués. Certains domaines comme les mathématiques sont plus propices à des solutions récursives simples. La difficulté est de penser à cette technique de programmation pour imaginer un algorithme.
2. Deux conditions sont nécessaires pour être en mesure d'utiliser la récursivité :
   ⇒ Il faut pouvoir exprimer un algorithme sous forme d'une fonction de telle manière que sa valeur à un certain rang ne dépende que de sa valeur aux rangs inférieurs
   ⇒ On doit aussi connaître sa valeur aux rangs initiaux.
3. **Technique pour écrire une fonction récursive :** il suffit d'utiliser la fonction qui n'a pas encore été écrite, en supposant qu'elle donne déjà un résultat
4. **Fonctionnement d'une fonction récursive**
   La récursivité, c'est une utilisation dynamique de la mémoire ; chaque appel à la fonction provoque l'obtention d'un nouvel espace mémoire

3.3.3 Suite de Fibonnacci

La suite de Fibonacci $(u_n)_{n\in\mathbb{N}}$ est une suite définie par ses deux premiers termes et le terme d'ordre n est défini par la somme des deux termes qui le précèdent ; autrement dit :

$$\begin{cases} u_0 \in \mathbb{R} \\ u_1 \in \mathbb{R} \\ u_n = u_{n-1} + u_{n-2} \end{cases}$$

La résolution algorithmique est celle ci :

```
#-*-coding:Latin-1 -*

#algo Fonction Suite de Fibonacci récursive


def Fibonacci(a,b,n):
    if  (n==0):
        return a
    elif (n==1):
        return b
    else:
        return Fibonacci(a,b,n-1) + Fibonacci(a,b,n-2)
```

Le résultat de l'exécution du script donne :

```
>>> Fibonacci(1,1,3)
3
>>> Fibonacci(1,1,5)
8
>>> Fibonacci(2,6,30)
6020698
>>>
```

Remarque 11 :

La version récursive effectue **un nombre exponentiel** de multiplication, car elle ne mémorise pas les calculs déjà effectués. La solution récursive n'est pas toujours la meilleure ; elle est même, parfois, très très lente.

3.3.4 Exercices

Exercice 8 :

Ecrire une fonction `Python` récursive $U\,(n)$ qui retourne U_n avec n un entier positif passé en paramètre :

$$\begin{cases} U_0 = 5 \\ U_n = \sqrt{1 + U_{n-1}} \end{cases}$$

Exercice 9 :

En supposant que la fonction puissance ne soit pas native, coder un programme, utilisant une fonction récursive pour calculer la fonction puissance entière. Algorithme `calcul_de_puissance`

Exercice 10 :

Ecrire une fonction récursive `Binomial(n,p)` permettant de calculer le coefficient binomial C_n^p, où n et p sont des entiers naturels passés en paramètres.
On rappelle que :

$$\mathrm{C}_n^p = \frac{n!}{p!\,(n-p)!} \text{ et que } \mathrm{C}_n^p = \mathrm{C}_{n-1}^p + \mathrm{C}_{n-1}^{p-1}$$

Exercice 11 :

Ecrire une fonction récursive `sommation(n)` qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls, n est passé en paramètre : $S = \sum_{k=1}^{n} \frac{1}{k^2}$

3.4 Correction de quelques exercices

Vous pouvez remarquer que, souvent, nous reprenons des algorithmes des chapitres précédents que nous retraitons sous forme de fonctions.

Exercice 1 :

Ecrire un programme qui affiche tous les nombres impairs entre 0 et n, dans l'ordre croissant, le nombre n étant donné par l'opérateur

La résolution de cette question ne pose pas de difficultés :

```
#-*-coding:Latin-1 -*


#algo Fonction nombres pairs

def Pair(n):
    for i in range(0,n+1,2):
        print(i)
```

Et nous obtenons :

```
>>> Pair(12)
0
2
4
6
8
10
12
>>> Pair(15)
0
2
4
6
8
10
12
14
>>>
```

Question : Comment afficher les nombres impairs ?

Exercice 2 :

Ecrire la fonction qui prend en entrée la température en degrés Celsius et la convertit en degré Farheneit.
La formule de conversion est donnée par : $C° = (F° - 32) \times \frac{5}{9}$

Cet exercice ne pose pas de difficultés.
Tout d'abord, en transformant un peu la formule $C° = (F° - 32) \times \frac{5}{9}$, nous avons $F° = \frac{9}{5}C° + 32$, d'où les fonctions :

```
 #Transformation de degré
    #Celsius en degré Farenheit
def degre(d):
    return 9/5*d+32


degre(25)
77.0
degre(180)
356.0
degre(100)
212.0
degre(0)
32.0
```

```
 #Transformation de degré
    #Farenheit en degré Celsius
def conversion(t):
    return (t-32)*5/9

conversion(-56)
-48.888888888888886
conversion(56)
13.333333333333334
```

Et je ne résiste pas :

```
conversion(degre(25))
25.0
degre(conversion(25))
25.0
```

Exercice 3 :

Ecrire une fonction qui nous donne $n!$ (sans utiliser la récursivité)

Là, non plus, ce n'est pas extraordinaire.....

```
def factorielle(n):
    fact = 1
    for i in range(2, n+1):
        fact = fact * i
    return fact

factorielle(5)
120
factorielle(12)
479001600
factorielle(30)
265252859812191058636308480000000
factorielle(50)
30414093201713378043612608166064768844377641568960512000000000000
```

Exercice 4 :

Calculez et affichez à l'écran 2^k avec k variant de 0 à n inclus.

```
#-*-coding:Latin-1 -*
#algo Puissance_de_2


def puissance_Deux(k):
    return 2**k

n=int(input("Jusque quelle valeur souhaitez vous la puissance de 2?"))
for i in range(0,n+1):
    print(" Pour n=",i, "la puissance de 2 est ", puissance_Deux(i))
```

Le résultat est donné par :

```
Jusque quelle valeur souhaitez vous la puissance de 2?25
 Pour n= 0 la puissance de 2 est  1
 Pour n= 1 la puissance de 2 est  2
 Pour n= 2 la puissance de 2 est  4
 Pour n= 3 la puissance de 2 est  8
 Pour n= 4 la puissance de 2 est  16
 Pour n= 5 la puissance de 2 est  32
 Pour n= 6 la puissance de 2 est  64
 Pour n= 7 la puissance de 2 est  128
 Pour n= 8 la puissance de 2 est  256
 Pour n= 9 la puissance de 2 est  512
 Pour n= 10 la puissance de 2 est  1024
 Pour n= 11 la puissance de 2 est  2048
 Pour n= 12 la puissance de 2 est  4096
 Pour n= 13 la puissance de 2 est  8192
 Pour n= 14 la puissance de 2 est  16384
 Pour n= 15 la puissance de 2 est  32768
 Pour n= 16 la puissance de 2 est  65536
 Pour n= 17 la puissance de 2 est  131072
 Pour n= 18 la puissance de 2 est  262144
 Pour n= 19 la puissance de 2 est  524288
 Pour n= 20 la puissance de 2 est  1048576
 Pour n= 21 la puissance de 2 est  2097152
 Pour n= 22 la puissance de 2 est  4194304
 Pour n= 23 la puissance de 2 est  8388608
 Pour n= 24 la puissance de 2 est  16777216
 Pour n= 25 la puissance de 2 est  33554432
```

Exercice 5 :

Créez une fonction qui calcule la distance euclidienne dans $\mathbb{R}^3$ entre 2 points $A\left(x_A, y_A, z_A\right)$ et $B\left(x_B, y_B, z_B\right)$

Cette distance $d\left(A, B\right)$ est donnée par : $d\left(A, B\right) = \sqrt{\left(x_A - x_B\right)^2 + \left(y_A - y_B\right)^2 + \left(z_A - z_C\right)^2}$ D'où

```
#-*-coding:Latin-1 -*
#algo Distance Euclidienne

import math

def distance(xa,ya,za,xb,yb,zb):
    return math.sqrt((xa-xb)**2+(ya-yb)**2+(za-zb)**2)

xa=float(input("donner l'abscisse de A "))
ya=float(input("donner l'ordonnée de A "))
za=float(input("donner la cote de A "))
xb=float(input("donner l'abscisse de B "))
yb=float(input("donner l'ordonnée de B "))
zb=float(input("donner la cote de B "))
print("La distance entre les points A et B est ", distance(xa,ya,za,xb,yb,zb))
```

Exercice 6 :

Écrire une fonction suite qui, étant donnés deux arguments n et a renvoie la valeur de u_n, où la suite $(u_k)_{k\in\mathbb{N}}$ est la suite définie par :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{a}{u_n}\right) \end{cases}$$

Enonce qui ne pose pas de difficultés majeures... Toujours faire attention à l'indentation

```
#-*-coding:Latin-1 -*



#algo Fonction racine carrée
def suite(n,a) :
    u = 1
    for k in range(n+1) :
        u=(u+a/u)/2
    return u
```

Nous obtenons comme résultats :

```
>>> suite(38,5)
2.23606797749979
>>> suite(100,5)
2.23606797749979
>>> suite(100,9)
3.0
>>>
```

Exercice 7 :

Cette question a été posée au Capes de mathématiques en 2023

Un cycliste parcourt 40 km la semaine 0. Il décide que, chaque semaine, il parcourra 5kms de plus que la distance parcourue lors de la semaine précédente. La fonction seuil *présentée ci-dessous, écrite en langage* Python, *permet de déterminer le numéro de la semaine où la distance totale qu'il aura parcourue sera supérieure à un nombre n donné. Que pensez vous de cette fonction ?*

```
#-*-coding:Latin-1 -*
#algo Enonce CAPES 2023

def seuil(n) :
  k=0
  u=40
  S=40
  while S<n :
    k=k+1
    S=S+u+5
  return k
```

Pour être honnête, je n'en pense pas que du bien !!

1. Pour commencer, nous allons faire « tourner » l'algorithme sur papier.
   En Recherchant, par exemple, `seuil(130)`, nous recherchons le nombre de jours k tel que la distance parcourue pendant ces k jours soit supérieure à 130. Nous trouvons :

Fonction `seuil(..)`	**Retour de la fonction** `seuil(..)`
`seuil(130)`	$k = 2$
`seuil(135)`	$k = 3$
`seuil(240)`	$k = 5$
`seuil(300)`	$k = 6$
`seuil(314)`	$k = 7$

2. Pour quoi ne pas être content de ce programme ?
   Parce qu'il n'est pas exact !!...
   Jamais la variable `u` ne change de valeur : nous avons toujours `u=40` de telle sorte que nous construisons une suite $(S_k)_{k\in\mathbb{N}}$ telle que $S_k = S_{k-1} + 40 + 5 = S_{k-1} + 45$
   C'est une suite arithmétique de raison 45 et de premier terme $S_0 = 40$, de telle sorte que $S_k = 40 + k \times 45$ et la fonction `seuil(n)` revient à chercher l'entier $k =$`seuil(n)` tel que $S_k \geqslant n$, c'est à dire l'entier k tel que $40 + k \times 45 \geqslant n \iff k \geqslant \dfrac{n-40}{45}$

3. Comment modifier ce programme ?
   Il faut modifier la valeur de la variable u dans la boucle en posant $u = u + 5$ et aussi modifier S en lui ajoutant la variable u modifiée $S = S + u$. D'où ce nouveau programme :

```
#-*-coding:Latin-1 -*
#algo Enonce CAPES 2023 Modifié

def seuilmodifie(n) :
  k=0
  u=40
  S=40
  while S<n :
    k=k+1
    u=u+5  #On modifie, ici, la valeur de la variable u
    S=S+u  #Et nous modifions S
  return k
```

4. Quelle est la réalité des choses ?
   La réalité des choses, c'est que notre cycliste, en pédalant, grâce à son entraînement quotidien, 5 kilomètres tous les jours, réalise une suite $(u_k)_{k\in\mathbb{N}}$ arithmétique de raispn 5 et de premier terme $u_0 = 40$, de telle sorte que, pour tout $k \in \mathbb{N}$, $u_k = u_0 + kr = 40 + 5k$.
   La somme des kilomètres parcourus est donnée par $\mathcal{S}_k = \dfrac{k+1}{2}(2u_0 + kr) = \dfrac{k+1}{2}(80 + 5k)$ et nous avons $\mathcal{S}_k \geqslant n \iff k \geqslant \dfrac{-85 + \sqrt{5625 + 40n}}{10}$ et on démontre facilement, par calcul que $\dfrac{-85 + \sqrt{5625 + 40n}}{10} \leqslant \dfrac{n - 40}{45}$
   Ainsi, la fonction `seuil(..)` est une majoration très large du nombre de jours à partir duquel la distance n aura été parcourue

Fonction `seuilmodifie(..)`	**Retour de la fonction** `seuilmodifie(..)`
`seuilmodifie(130)`	$k = 2$
`seuilmodifie(135)`	$k = 2$
`seuilmodifie(240)`	$k = 4$
`seuilmodifie(300)`	$k = 5$
`seuilmodifie(314)`	$k = 5$
`seuilmodifie(400)`	$k = 7$

Exercice 8 :

Ecrire une fonction récursive $U(n)$ qui retourne U_n avec n un entier positif passé en paramètre :

$$\begin{cases} U_0 = 5 \\ U_n = \sqrt{1 + U_{n-1}} \end{cases}$$

```
#-*-coding:Latin-1 -*
#algo calcul_de_suite

import math
def U(n):
  if (n==0):
    return 1
  else:
    r=math.sqrt(1+U(n-1))
    return r
```

Il est possible de ré-écrire les lignes :

```
r=math.sqrt(1+U(n-1))
return r
```

En

```
return math.sqrt(1+U(n-1))
```

Exercice 9 :

En supposant que la fonction puissance ne soit pas native, coder un programme, utilisant une fonction récursive pour calculer la fonction puissance entière.

```
#-*-coding:Latin-1 -*
#algo calcul_de_puissance

def puissance( x, n):
  x=float(x)
  if (n==0):
    r=1
    return r
  else:
    r=x*puissance(x,n-1)
    return r
```

Quelques exemples de résultats :

```
>>> puissance(2,3)
8.0
>>> puissance(32,6)
1073741824.0
>>> puissance(2,10)
1024.0
>>> puissance(1.4242,2)
2.02834564
>>> puissance(1.4142,2)
1.9999616399999998
>>> puissance (100,0)
1
>>>
```

Exercice 10 :

Ecrire une fonction récursive Binomial(n,p) permettant de calculer le coefficient binomial C_n^p, où n et p sont des entiers naturels passés en paramètres. On rappelle que :

$$C_n^p = \frac{n!}{p!\,(n-p)!} \text{ et que } C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$$

```
#-*-coding:Latin-1 -*
#algo calcul_coefficients_Binomiaux


def Binomial(n,p):
  if (n==0) or (p==n):
    return 1
  elif 0<p<n:
    return Binomial(n-1,p-1)+ Binomial(n-1,p)
  else:
    return 0
```

Nous obtenons comme résultat :

```
>>> Binomial(0,0)
1
>>> Binomial(100,100)
1
>>> Binomial(12,5)
330
>>>
```

Le calcul de

```
>>> Binomial(230,26)
```

prend un temps faramineux....à éviter!!*(je n'ai pas hésité à tuer le programme en fermant la fenêtre)*

Exercice 11 :

Ecrire une fonction récursive sommation(n) qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls, n est passé en paramètre : $S = \sum_{k=1}^{n} \frac{1}{k^2}$

```
#-*-coding:Latin-1 -*
#algo calcul_Sommation_Recursive


def Sommation(n):
  if (n==0):
    print("Division par zéro impossible")
  elif n==1:
    return 1
  else:
    return 1/(n**2)+Sommation(n-1)
```

Chapitre 4

Les proverbes du programmeur

Cette partie a été empruntée au livre de Raymond Séroul :
Informatique pour les mathématiciens

4.1 Surtout, mais surtout pas d'astuces !

Ce proverbe - *peut-être le plus important de tous* - choque toujours le mathématicien qui n'a jamais programmé ; il choque aussi les **programmeurs débutants qui ont toujours tendance à *bidouiller*.** N'en déduisez surtout pas que les informaticiens sont des imbéciles ! Vous comprendrez la profondeur de ce conseil le jour où vous commencerez à écrire des programmes importants.
Ce qui comptera alors, ce sera un programme clair, qui fonctionne tout de suite (ou presque...) et **facile à maintenir**.
Un informaticien passe beaucoup *(trop)* de temps à maintenir des programmes, ce qui veut dire modifier un programme qui a été écrit -*en général*- par une personne qui n'est plus là.
Un programme professionnel est très long : plusieurs centaines de pages. Lire un programme est un exercice très difficile *(c'est aussi atroce qu'essayer de comprendre une démonstration réduite à de simples calculs et dépourvue d'explications : **Commenter un programme,** c'est comme rédiger un devoir de maths et réciproquement).*
En outre, le temps est compté ; le programmeur qui modifie un programme ne peut pas se permettre de passer des heures à se demander ce que signifie le code qu'il a sous les yeux. *Plus le code est bête* **c'est-à-dire limpide, sans détours**, plus il est sûr et plus il est facile à modifier.
Le gros reproche que l'on peut faire aux astuces est qu'elles sont la plupart du temps inutiles.
On estime qu'en général un programmeur passe plus de 80 % de son temps dans moins de 20 % du code. Il en résulte qu'une astuce a de très fortes chances d'intervenir dans une partie du code où la machine ne fait qu'attendre *(quand on entre des données au clavier par exemple)*. Il est stupide et suicidaire de fragiliser un programme en utilisant une astuce pour faire gagner quelques millisecondes à un programme qui tourne mille fois ou cent mille fois moins vite parce qu'il vous attend
Faites-vous violence : **choisissez toujours le code le plus bête,** même si cela exige quelques lignes supplémentaires.
La modestie paye toujours à la longue. Ceci dit, le recours à une astuce est parfois indispensable, par exemple dans une boucle sollicitée en permanence dans un programme trop lent. Si c'est le cas **documentez !** Avertissez votre lecteur, expliquez-lui en détail ce que vous faites et pourquoi vous le faites *(la machine est trop lente, elle n'a pas assez de mémoire, etc.)*. N'oubliez pas que ce lecteur, ce sera peut-être vous dans quelques mois avec une machine et un point de vue différents...
Que préférez-vous ? :

- Un programme facile à mettre au point et que vous aurez plaisir à optimiser ensuite,
- Ou un programme illisible, plein de fautes et qui ne marchera qu'après des heures et des heures de déverminage ?

4.2 On ne mâche pas son chewing gum en montant un escalier

Ce proverbe -*tiré d'une campagne électorale américaine*- exprime une idée de bon sens :

on ne fait bien qu'une seule chose à la fois.

C'est pour cela qu'on

découpe un programme en procédures ou fonctions et en modules indépendants.

4.3 Nommez ce que vous ne connaissez pas encore

Ce proverbe s'applique chaque fois que vous rencontrez un sous-problème à l'intérieur d'un problème. Refusez de résoudre le sous-problème *(proverbe précédent)*, laissez-le de côté et avancez. Comment ? En donnant un nom à ce que vous ne connaissez pas encore *(du code, une fonction)*.
Cela vous permettra de terminer le travail *(i.e. le problème)* en cours. Pour le sous-problème, voyez le proverbe suivant.

4.4 Demain sera mieux, après demain, encore mieux !

Non, ce n'est pas un éloge de la paresse ! Il s'agit au contraire d'une technique extraordinaire pour être efficace.
Appliquez ce proverbe chaque fois que vous constatez un blocage provoqué par l'apparition d'un nouveau problème à l'intérieur du problème que vous essayez de résoudre : appliquez le proverbe précédent en remettant la solution du nouveau problème à demain en lui donnant un nom sous la forme d'une procédure ou d'une fonction dont vous écrirez le code plus tard.
Séparez toujours ce qui est urgent de ce qui ne l'est pas ;, apprenez à distinguer l'essentiel de l'accessoire ; ne vous noyez pas prématurément dans les détails. Les détails, vous vous en occuperez demain, ce qui signifie quelques minutes ou quelques heures en général.
Il ne s'agit pas de reprendre le travail 24 heures plus tard comme aimeraient le croire certains paresseux ! Cette technique permet d'avancer petit à petit. Vous l'avez certainement pratiquée en mathématique :

Je vais d'abord prouver mon théorème en admettant provisoirement les lemmes 1, 2 et 3.

Ce proverbe nous met aussi en garde contre un défaut de débutant : vouloir agir tout de suite en écrivant prématurément un code très technique. La bonne attitude est à l'opposé : il faut cultiver une certaine nonchalance en donnant des ordres aujourd'hui ; le reste se traitera demain. « Hâtez-vous lentement ! » dit un autre proverbe.

4.5 On n'exécute jamais un ordre avant de le donner

Un débutant est toujours trop pressé : il écrit, il écrit... Le résultat est un code trop riche, trop technique, trop long et donc forcément incompréhensible. Allez trouver ensuite la faute dans ce fatras !
Ce défaut est très facile à mettre en évidence. Si pour comprendre un code vous devez entourer certaines parties d'un rectangle auquel vous attachez une explication, vous pouvez être certain qu'il manque une procédure *(l'ordre, si vous préférez)* à cet endroit.
Remplacez cette partie de code par un appel de procédure. Vous « exécuterez » cet ordre plus tard, lorsque vous écrirez le code de la procédure. Sachez donc vous retenir...

4.6 Documentez aujourd'hui pour ne pas pleurer demain

Imaginez une démonstration réduite à des calculs et quelques symboles de logique : elle est illisible, donc inutile. [1] Quand vous programmez, pensez à expliquer très précisément ce que vous faites.

- Tout d'abord cela vous oblige à comprendre ce que vous voulez entreprendre :

1. Grand défaut habituel des étudiants

ce qui se conçoit bien s'énonce clairement

dit la sagesse populaire. Si vous n'arrivez pas à expliquer votre code à un camarade, vous pouvez être certain que vos idées sont approximatives et que votre programme est vraisemblablement incorrect.

Pour prendre conscience, pour éclaircir vos idées, dialoguez avec vous-même. Le meilleur moyen d'y parvenir est de vous contraindre à écrire les commentaires au fur et à mesure de votre progression ; n'attendez pas que votre programme soit terminé, ce serait trop tard.

Les mathématiciens ont compris ce mécanisme de prise de conscience depuis longtemps : ils rédigent soigneusement leurs démonstrations avant d'y croire vraiment.

- Si votre programme est faux, ou si vous devez le reprendre six mois plus tard pour le transformer ou recycler une procédure, vous serez bien content de trouver les explications qui vous indiqueront comment le programme a été conçu.

4.7 Le discours de la méthode de Descartes

Le Discours de la méthode de Descartes a été publié en 1637. C'est un texte fascinant,important, car les méthodes de programmation moderne s'en inspirent !

Comme la multitude des lois fournit souvent des excuses aux vices, en sorte qu'un état est bien mieux réglé lorsque, n'en ayant que fort peu, elles y sont fort étroitement observées, ainsi, au lieu de ce grand nombre de préceptes dont la logique est composée, je crus que j'aurais assez des quatre principes suivants, pourvu que je prisse une ferme et constante résolution de ne manquer pas une seule fois à les observer.
Le premier était de ne recevoir jamais aucune chose pour vraie que je ne la connusse évidemment être telle ; c'est-à-dire **d'éviter soigneusement la précipitation et la prévention,** et de ne comprendre rien de plus en mes jugements que ce qui se présenterait si clairement et si distinctement à mon esprit que je n'eusse aucune occasion de le mettre en doute.
Le second, de diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre.
Le troisième, de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu comme par degrés jusques à la connaissance des plus composés, et supposant même de l'ordre entre ceux qui ne se précèdent point naturellement les uns les autres.
Et le dernier, de faire partout des dénombrements si entiers et des revues si générales, que je fusse assuré de ne rien omettre.

Deuxième partie

Mathématiques et programmation

Index

Afficher, 10
Algorithme, 3

Données, 4

Fonction, 36
    Fonction récursive, 48
    Signature d'une fonction, 38
    Sous-programme, 36

Globales
    Variables globales, 41

Instructions
    L'instruction `For`, 21
Instructions conditionnelles, 16
    Instructions conditionnelles imbriquées, 17
    Si..alors, 17
    Si..alors..sinon, 16
Instructions de répétition, 19
    La boucle `while`, 19
    La boucle itérative `for`, 21

Locales
    Variables locales, 41

Module, 42
    `import`, 43
Modules
    Packages, 47

Packages, 47
Programme, 3

Récursivité, 47
    Factorielle, 48
    Suite de Fibonnacci, 49

Type
    Type des variables, 6
Type des variables
    Le type Booléen, 9
    Type réel et type entier, 6

Variables, 4
    Affectation d'une variable, 5
    Nom d'une variable, 4
    Type d'une variable, 5
    Type des variables, 6
    Variables globales, 41
    Variables locales, 41