Chapitre 1 : Premières instructions

Pour résoudre un problème algorithmique :

1. Analyse (dialogue avec le client…)
2. Méthode de résolution : algorithmique
3. Traduction en programme informatique
4. Exécution

Définitions :

Algorithme = une suite finie d'opération (d'instruction), prenant en entrée une ou plusieurs données, et fournissant en sortie une ou plusieurs données ; qui répond à la résolution d'un problème (exemple : une recette).

Instruction = opération élémentaire qu'un programme demande à un processeur d'effectuer, ordre le plus basique que comprend un ordinateur

Processeur = lit une séquence d'instruction (programme), et exécute pour chaque instruction l'opération correspondante (exemple : calcul arithmétique, transfert de données). Pas intelligent, ne détecte pas les incohérences d'un programme

Mémoire (= RAM) = sert à stocker les instructions et les données, disparition du contenu quand l'alimentation est coupée.

Mémoire secondaire (exemple : disque dur, clé USB…) = support permanant.

Pour décrire un algorithme → **pseudo code,** langage universel ≠ langage informatique.

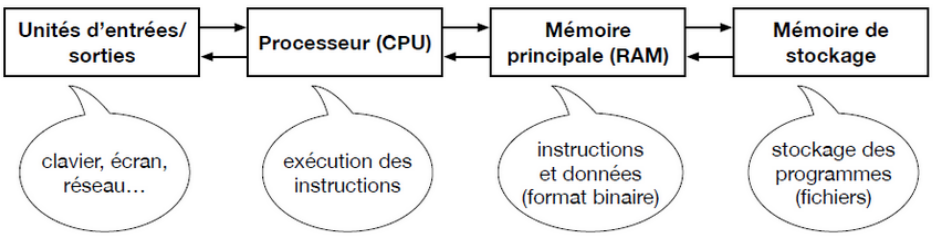
Ecriture d'un algorithme en pseudo code :

ALGO
VARIABLES
…… pour décrire les données et variables
DEBUT
…… pour décrire les opérations élémentaires
FIN

Traduction du pseudo code en programme informatique en utilisant un langage informatique dont la syntaxe est précise (vocabulaire + règles de grammaires)
→ Exécution d'un programme sur un ordinateur en code binaire

Programme = traduction d'un algorithme en un langage compréhensible par un ordinateur.

MODELE D'ORDINATEUR :



Un programme manipule des **valeurs** (codées par l'ordinateur par des 0 et des 1 = code binaire) → pour interpréter correctement une valeur, on lui associe un type.
Exemple : 01001111 = 79_{10} ou O en ASCII.

Type : détermine un ensemble de valeurs et les opérations que l'on peut effectuer sur ces valeurs.
Il y a 2 types au départ :

- NOMBRE :
  - ENTIER = *int*
  - DECMAL = *float*
- CHAINE (en mettant des « »). *Exemple : « quelle est votre ville de départ ? »*

Les valeurs sont stockées dans des zones mémoire de l'ordinateur. Chaque zone a une adresse, mais on y a accès non pas en donnant l'adresse en mémoire, mais en référençant une variable. Une variable possède un identificateur et un type.

Identificateur est une chaine de caractère (sans « » pour le différencier d'une valeur de type CHAINE).
Choix de l'identificateur important pour la compréhension/lisibilité du programme.

Affectation = instruction qui permet d'associer une valeur à une variable
En pseudo code : ← pour « prend la valeur »
En Python : **=**
Evaluation de la variable renvoie la valeur contenue dans la zone mémoire associée.

Variable : permet de stocker une valeur.

Expression : formule combinant des variables, des valeurs, des opérateurs… - évaluée, puis renvoie un résultat unique et typé. On peut mettre *expression* à la place de *valeur.*
Utilisation des propriétés usuelles de l'arithmétique. **UTILISER DES ()**

identificateur_variables ← expression

Exemple d'affectation de variables en pseudo code :
ALGO
VARIABLES
x, y TYPE NOMBRE (déclaration des variables, 2 variables de type NOMBRE sont les identifiants sont x et y)

DEBUT

x ← 6

y ← 2*x (← = affectation, 2*x = expression)

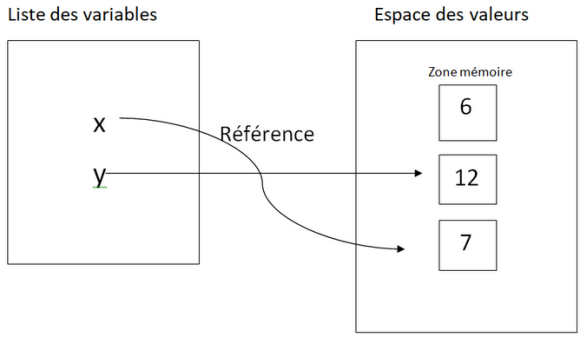
FIN

En Python : *identificateur_variable* = *expression*

Programme Python correspondant à l'algorithme précédent :

```
x = 6
y = 2*x
```

Dessin :



x = 6

y = 2*x

x = 7

En Python :

- A gauche du = : toujours une variable
- ATTENTION : « = » est une variable et pas une égalité !
  (exemple : a = a + 1 existe en informatique. C'est *ancienne_valeur+1*)
- Pas de partie explicite de déclaration de variable
- Déclaration et initialisation en même temps
- Type de la variable inféré automatiquement par la valeur initiale
- Utiliser uniquement lettres, nombres ou caractères
- Ne pas débuter un identificateur par un nombre ni par _
- Attention : nb, nB sont des identificateurs différents
- Pas d'espace, pas d'accent, ni mot clé du langage (false, and…)

Opération :

- *, +, -, /, // (=division entière), ** (= puissance), % (= reste de la division entière)
- *int* = entier, *float* = décimale (attention : 4.0 est un *float*)
- Fonction type () sert à rappeler le type d'une expression
- Conversion explicite de type Python :

```
x = 2.1
y = int (x)
```

- a, b = 4, 8.33 (pour un couple)

- Opérateurs particuliers :
  - a+ = 2 → a=a+2
  - a- = 2 → a=a-2
  - a* = 2 → a=a*2
  - a/ = 2 → a=a/2

Lecture :

Permet de lire une valeur provenant de l'extérieur (clavier) et l'affecter à la variable X.

En pseudo code : LIRE X

En Python : x = input () (=demander)→ interruption de l'exécution du programme pour attendre que l'utilisateur saisisse quelque chose au clavier. La valeur saisie est « lue » comme une chaîne de caractères et est affectée à la variable x.

x = int(input()) pour lire une valeur de type entier

X= int(input(« entrer une valeur »)) pour lire une valeur de type entier avec message

Ecrire :

Permet d'écrire une valeur pour la délivrer à l'extérieur (affichage à l'écran).

En pseudo code : ECRIRE X

En Python : *print (x)*, (=afficher)

print(x) : la chaine de caractères référencée par x est affichée.

Quand on écrit une expression, l'expression est d'abord évaluée et la valeur retournée est affichée.

pi = 3.14

*print (''la valeur affichée pour 2 pi est'' , 2*pi)*

''\n'' → pour valeurs séparer par un passage à la ligne

''\t'' → tabulation

- Fonction `print()` avec affichage par défaut :

```
>>> pi=3.14
>>> print("2*pi=", 2*pi)
```

- Modification de l'affichage par défaut avec le paramètre optionnel `sep` :

```
>>> pi=3.14
>>> print(pi, 2*pi, 4*pi)
                          3.14  6.28  12.56
>>> print(pi, 2*pi, 4*pi, sep=",")
                                  3.14,6.28,12.56
>>> print(pi, 2*pi, 4*pi, sep="\n")
```

- Modification de l'affichage par défaut avec le paramètre optionnel `end` :

```
>>> print(pi, 2*pi, 4*pi, end="Fin")
    3.14,6.28,12.56  FIN
```

```
3.14
6.28
12.56
```

<u>Opérateurs de comparaisons :</u>

<, >, == (égal, à ne pas confondre avec affectation en Python), != (différent), ≤, ≥

Instruction conditionnelles :

On peut vouloir exécuter certaines instructions uniquement si une condition est vérifiée.
Instruction conditionnelle : SI…ALORS…

En pseudo code :

```
SI (Condition) ALORS
        DEBUT
                BlocInsctructions
        FIN
```

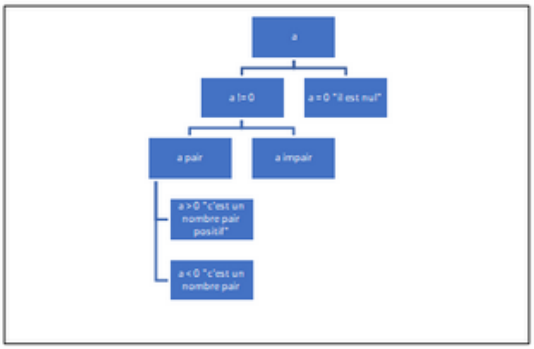
BlocInstructions est une suite d'instructions exécutée uniquement si la condition est vérifiée.
Si condition non vérifiée, il ne se passe rien.
Condition : expression logique retournant un résultat de type BOOLEEN
Type BOOLEEN : 2 valeurs VRAI ou FAUX

Pour traiter le cas où la condition n'est pas vérifiée :

```
SI (Condition) ALORS
        DEBUT
                BlocInsctructions1
        FIN
SINON
        DEBUT
                BlocInstructions2
        FIN
```

Imbrications d'instructions conditionnelles : exemple

```
ALGO
VARIABLES
a TYPE NOMBRE
DEBUT
    ECRIRE "Quel est le parametre a?"
    LIRE a
    SI (a!=0) ALORS
        DEBUT
            SI (a%2==0) ALORS
                DEBUT
                    SI (a>0) ALORS
                        DEBUT
                            ECRIRE "C'est un nombre pair positif"
                        FIN
                    ECRIRE "C'est un nombre pair"
                FIN
        FIN
    SINON
        DEBUT
            ECRIRE "Il est nul"
        FIN
FIN
```



Chapitre 2 : Boucles répétitives

1. Boucle répétitive TANT_QUE en pseudo-code

Exemple 2 : Que fait cet algorithme ?

```
ALGO
VARIABLES
n TYPE NOMBRE
DEBUT
  n <- 0
  TANT_QUE (n<=10) FAIRE
    DEBUT
      ECRIRE n
      n <- n+2
    FIN
FIN
```

Cet algorithme affiche tous les nombres pairs de 0 à 10 compris.
Si la place des deux instructions était échangée, l'algo aurait affiché les valeurs paires de 2 à 12.

Exemple 3 : On veut connaître le plus petit entier n tel que $2^n \geq 100$.

Compléter les lignes 6 et 8 de l'algorithme ci-dessous pour qu'il réponde au problème.

```
1   ALGO
2   VARIABLES
3   n TYPE NOMBRE
4   DEBUT
5     n <- 1
6     TANT_QUE (2**n  ???? ) FAIRE
7       DEBUT
8         n <- ????
9       FIN
10     ECRIRE n
11  FIN
```

```
6       TANT_QUE (2**n < 100) FAIRE
8           n <- n+1
```

Reprise de l'exemple 3 sans utiliser l'opérateur puissance :

```
ALGO
VARIABLES
CPT, p TYPE NOMBRE
DEBUT
   CPT <- 0
   P <- 2
   TANT_QUE (p < 100) FAIRE
      DEBUT
```

```
            P <- P*2
            CPT <- CPT + 1
        FIN
    ECRIRE CPT + 1
FIN
```

En Python : TANT_QUE devient WHILE

Exemple 4 : Programme Python affichant les entiers de 1 à 10

```
n=1
while(n<=10):
    print(n)
    n=n+1
```

Exemple 5 : que se passe-t-il

```
n=1
while(n<=10):
    print(n)
n=n+1
```

C'est une boucle infinie car la condition se porte sur n alors que n n'est jamais modifié.

Exemple 6 : que se passe-t-il ?

```
n=1
while True:
    print(n)
```

C'est une boucle infinie car l'expression qu'on a écrite est toujours vraie.

Problème : lire 10 valeurs > 0 et afficher leur somme, avec 2 stratégies de gestion des erreurs de saisie.

- Stratégie 1 : interruption de la lecture (BREAK)
- Stratégie 2 : poursuite de la lecture jusqu'à obtention de 10 valeurs > 0 (CONTINUE)

```
Cpt=1
Som=0
While cpt<=10:
    X=int(input("x?"))
    If x <= 0:
        break
    Cpt=cpt+1
    Som = som + x
Print ("somme =", som)
```

Ou :

Stratégie 1 pour gérer les erreurs de saisie : interrompre la lecture en utilisant l'instruction `break`

```
cpt=0
somme=0
while cpt < 10:
    x=int(input())
    if x<=0: break
    somme+=x
    cpt+=1
if cpt<10:
    print("Erreur de saisie")
else:
    print("La somme des 10 valeurs lues est :", somme)
```

Réécriture de la stratégie 1 p. 24 sans instruction `break`

```
cpt=0
somme=0
while cpt < 10:
    x=int(input())
    if x<=0: cpt=10
    else :
        somme+=x
        cpt+=1
if x<=0:
    print("Erreur de saisie")
else:
    print("La somme des 10 valeurs lues est :", somme)
```

=Pour imiter BREAK en Pseudo-Code

```
Cpt=1
Som=0
While cpt<=10:
    X=int(input("x?"))
    If x <= 0:
        Print ("non +")
        Continue
    Cpt=cpt+1
    Som = som + x
Print ("somme =", som)
```

Stratégie 2 pour gérer les erreurs de saisie : poursuivre la lecture jusqu'à obtenir 10 valeurs >0 en utilisant l'instruction `continue`

```
cpt=0
somme=0
while cpt < 10:
    x=int(input())
    if x<=0:
        continue
    somme+=x
    cpt+=1
print("La somme des 10 valeurs lues est :", somme)
```

Instruction non autorisée en algorithmique

```
Cpt=1
Som=0
While cpt<=10:
```

```
X=int(input("x?"))
If x <= 0:
   Print ("non non non")
Else:
  Cpt=cpt+1
  Som = som + x
Print ("somme =", som)
```

= Pour imiter CONTINUE en Pseudo-Code

Chapitre 3 : Fonctions

1. Introduction, Motivation

Fonction = sous algorithme réalisant une tâche, pouvant prendre des données en entrées (arguments) et pouvant renvoyer une valeur en sortie.
Exemple de fonctions déjà utilisées en Python : PRINT(), INPUT(), SQRT()
"utiliser une fonction" = "appeler une fonction" → Exécuter la fonction
Si une fonction restitue une valeur, elle doit la renvoyer → une fois exécutée, la valeur est récupérée, remplacement de l'appel de la fonction par la valeur renvoyée.

Et si on veut avoir une fonction non définie, nous pouvons en créer nous-mêmes.

2. Fonction en pseudo code

Algorithme listant tous les nombres parfaits allant de 1 à *n*, *n* étant donné par l'utilisateur

```
ALGO
VARIABLES
  n TYPE NOMBRE
DEBUT
  LIRE n
  TANT_QUE (n>1) FAIRE
    DEBUT
      SI (sommeDiviseurs(n)==n) ALORS
          DEBUT
              ECRIRE n
          FIN
      n <- n-1
    FIN
FIN
```

ou en Python :

```
n=int(input())
while (n>1) :
    if(sommeDiviseurs(n)==n):
        print(n)
    n-=1
```

Structuration d'un algo modulaire :

- Partie 1 : définition de la fonction = déclaration de la fonction

- Partie 2 : utilisation de la fonction = appel de la fonction, c'est-à-dire, instruction correspondant au nom de la fonction dans un autre algorithme.

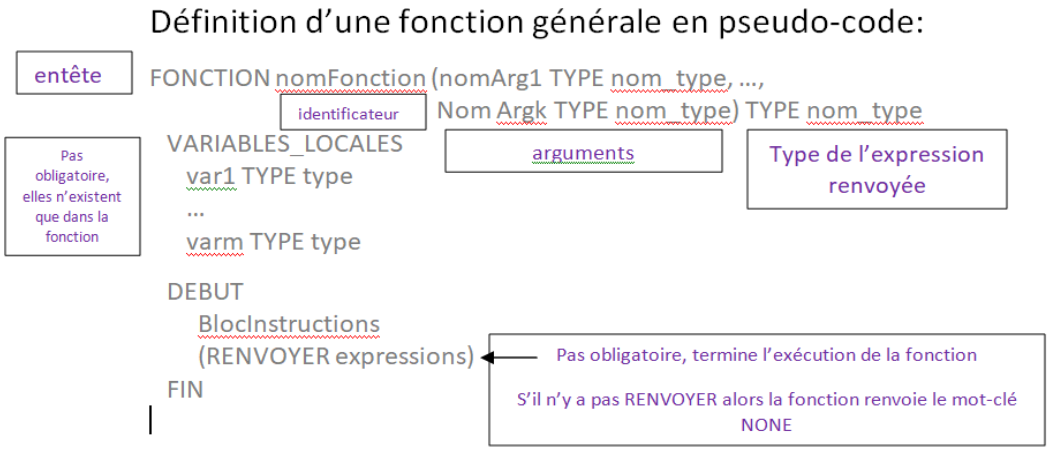
Exemple de fonction écrite en Pseudo-Code:

Le but : écrire une fonction qui prend en argument un nombre et qui renvoie le cube de ce nombre.

```
FONCTION Cube (n TYPE NOMBRE) TYPE NOMBRE
VARIABLES_LOCALES
y TYPE NOMBRE
DEBUT
    y<- n*n*n
    RENVOYER y
FIN
```

Utilisons la fonction :

```
ALGO
VARIABLES
x, t TYPE NOMBRE
DEBUT
    LIRE x
    t=Cube(x)
    ECRIRE t
FIN
```



VARIABLES LOCALES : Déclaration des variables nécessaires pour les instructions de la fonction. N'existent que dans la fonction dans laquelle elles ont été définies et ne peuvent pas être utilisées ailleurs.

Les *k* arguments : des variables locales à la fonction au même titre que les variables déclarées dans la section des variables locales.

Toujours définir les fonctions au début du programme.
Ne jamais lire les variables locales.

Fonction calculant la somme des diviseurs d'un entier passé en paramètre

```
FONCTION sommeDiviseurs(e TYPE NOMBRE) TYPE NOMBRE
  VARIABLES_LOCALES
    somme, d TYPE NOMBRE
  DEBUT
    d <- 1
    somme <- 0
    TANT_QUE (d<e) FAIRE
      DEBUT
        SI ((e%d)==0) ALORS
          DEBUT
            somme <- somme+d
          FIN
        d <- d+1
      FIN
    RENVOYER somme
  FIN
```

Exercice : écrire une fonction prenant un nombre en argument et teste si ce nombre est pair : la fonction doit renvoyer une valeur booléenne VRAI si ce nombre est pair et FAUX sinon.

```
Fonction pair(n type nombre) type booléenne
Variables locales
Saisi_n type booleen
DEBUT
   SI n%2==0 alors
      DEBUT
         Saisi_n=VRAI
      FIN
   SINON
      DEBUT
         Saisi_n=FAUX
      FIN
    RENVOYER Saisi_n
FIN
```

Ou :

```
Fonction Pair Bis (n type nombre) type boolean
   Début
      Renvoyer n%2==0
   Fin
```

Syntaxe de l'appel :

- Autant de valeurs passées en paramètres que d'arguments (si k arguments dans la définition alors k valeurs dans l'appel)
- Coïncidence des types entre valeurs passée en paramètre et argument
- Nécessité que le type retour de la fonction et la variable qui stocke la valeur retournée coïncident.

Syntaxe de Python pour la définition d'une fonction :

```
Def nom_fonction (nomArg1, …, nomArgi, …, nomArgk):
    BlocInstructions
    (return expressions)
```

3. Fonctions en Python

Exemple d'un algorithme en Python qui affiche la plus grande valeur entre 2 valeurs :

Définition de la fonction:

```
Def Maxi (x, y) :
     If x>y :
          Return x
     Else :
          Return y
```

Utilisation:

```
a=int(input("1ère valeur"))
b=int(input("2ème valeur"))
c=Maxi(a, b)
Print(c)
```

- Ne pas utiliser de mot-clé de python ou de nom de fonctions existantes pour le nom d'une fonction
- Pas de caractères spécial ou accentué (caractère _ autorisé)
- Pas besoin de préciser le type des arguments ou le type retour de la fonction en Python.
- *Return* → on quitte la fonction, pas obligé de l'avoir si on n'a pas une expression à retourner.
- Possibilité d'avoir plusieurs instructions *return* dans le corps d'une fonction mais toutes, devront se retrouver dans des instructions conditionnelles.

Exemple :

```
Def resteEtQuotient (num, denom) :
      Return num%denom, num//denom
X=12
Y=2
ResteEtQuotient(12,2)
(a, b) = resteEtQuotient(12,2)
Print ("le reste est", a)
Print("le résultat est", b)
```

Appel et passage des paramètres par valeur

- Pour les arguments de type simple (int, float, str, bool): passage des arguments par valeur.
- Pas de modification de la valeur des variables passées en paramètre par la fonction

Fonction en Pseudocode :

```
ALGO
FONCTION fonction(a, b TYPE NOMBRE) TYPE NOMBRE/TABLEAU
```

```
VARIABLES LOCALES
c VARIABLE TYPE NOMBRE
DEBUT
…
RENVOYER a
…
```

Fonction en Python :

```
Import...
def fonction(a, b) :
    …
    Return a
    …
```

Renvoyer :

- Restituer la fonction par le résultat
- Arrêt de l'exécution
- Si pas renvoyer, le résultat est none

Comment se servir d'une fonction enregistrée sur un autre fichier ?

- Imaginons que nous avons un fichier "func.py" dans lequel on met toutes nos fonctions
- Imaginons que nous travaillons maintenant dans un autre ficher "exos.py" où nous voulons nous servir d'une fonction du fichier "func.py".

Chapitre 4 : Séquences, Tableaux et Instruction POUR

1. Introduction, Motivation

Les types vus jusqu'à présent sont les "types élémentaires".
Si je veux regrouper plusieurs données dans un groupe homogène? Si j'ai bcp de données et que je ne connais pas leur nombre au préalable ?
Nous avons la possibilité de définir des "**types structurées**" = assemblage de types élémentaires.

2. Séquences

Séquence = ensemble fini d'éléments pouvant être de différents types. Leur structure est dynamique : le nombre d'éléments n'est pas fixé a priori et peut varier au cours de l'exécution du programme. N'importe quel élément est accessible à tout moment.
Notation : suite d'éléments, représentée par $S = a_1, a_2, \ldots, a_n$ avec a_i, un des éléments.
Longueur de la séquence : nombre n d'éléments dans la séquence (peut être = à 0)
Position des éléments dans la séquence : a_i à la $i^{ième}$ position, a_i précède a_{i+1}
La longueur compte le nombre de positions et non pas d'éléments ou de valeurs distinctes.
Opérations élémentaires sur les séquences : Concaténation de séquences

3. Type de données Tableau

Tableau = nombre fixé d'éléments, les éléments peuvent être repérés par un indice simple. Tous les éléments sont du même type.

T[i] : élément d'indice i dans le tableau

Attention : décalage entre l'indice du tableau et la position correspondante dans la liste, ça commence à 0 : Élément de position p = T[p-1]

Tableau en pseudo-code :

Déclarer une variable T de type tableau à une dimension :

```
T TYPE TABLEAU DE TypeElement
```

Où T peut être globale à l'algo ou locale à une fonction et TypeElement : type des éléments du tableau

Exemple :

```
T TYPE TABLEAU DE NOMBRE
N TYPE NOMBRE
LIRE N
T ← CREER_TABLEAU (N)
```

Tableaux à 2 dimensions :

Initialiser T par une boucle

Par exemple par saisi au clavier :

```
i ← 0
TANT_QUE i<N FAIRE
     DEBUT
          ECRIRE ("Valeurs en position", i+1)
          LIRE T[i]
          i←i+1
     FIN
```

Tableau à deux dimensions n et m : T[i][j] : élément de la ligne i et de la colonne j

Définir un tableau à 2 dimensions en pseudo-code :

```
T TYPE TABLEAU DE TypeElement
     …
T ← CRÉER_TABLEAU (N, M)
```

4. Boucle POUR

Instruction permettant de répéter un nombre de fois, décidé à l'avance, un bloc d'instructions : Boucle `POUR` en pseudo-code.

La valeur de i ne doit pas être modifiée dans le BlocInsruction.

T=[1,2,4,8,16] :

```
ALGO
VARIABLES
i, n TYPE NOMBRE
T TYPE TABLEAU DE NOMBRE
```

```
DEBUT
   T← CREER TABLEAU
   TANT_QUE i<5 FAIRE
        DEBUT
```

```
        LIRE n
        T[i] ←n
        i←i+1
        FIN
FIN
```

Ou :

```
ALGO
VARIABLES
i, n TYPE NOMBRE
T TYPE TABLEAU DE NOMBRE
DEBUT
   POUR i ALLANT DE 0 A 4
        DEBUT
             T[i]←2**i
        FIN
FIN
```

Algorithme affichant le carré de tous les entiers de 1 à 10 :

```
ALGO
VARIABLES
i TYPE NOMBRE
carre TYPE NOMBRE
DEBUT
    POUR i ALLANT_DE 1 A 10
        DEBUT
            carre <- i*i
            ECRIRE carre
        FIN
FIN
```

Algorithme permettant de remplacer chaque nombre par son cube :

```
ALGO
VARIABLES
i, n TYPE NOMBRE
cube TYPE NOMBRE
T TYPE TABLEAU DE NOMBRE
DEBUT
    LIRE n
    T <- CREER_TABLEAU(n)
    POUR i ALLANT DE 0 A (n-1)
        DEBUT
            LIRE T[i]
        FIN
    POUR i ALLANT_DE 0 A (n-1)
        DEBUT
            cube <- T[i]*T[i]*T[i]
            T[i] <- cube
        FIN
FIN
```

Exercice : écrire en pseudo code un algo qui demande à l'utilisateur de remplir un tableau de chaine de caractère (en lui demandant au préalable sa taille), lui demande ensuite un élément x, puis détermine la dernière position (entre 1 et n) de cet élément dans le tableau.

```
ALGO
VARIABLE
i, n TYPE NOMBRE
T TYPE TABLEAU DE CHAINE
DEBUT
    LIRE n
    T←CREER TABLEAU(n)
    POUR i ALLANT DE 1 A n-1
        DEBUT
            LIRE T[i]
```

```
          FIN
     LIRE x
     Derniere position ← -1
     POUR i ALLANT DE 0 A n-1
     DEBUT
          SI T[i]==x ALORS
          DEBUT
               Derniere_Position←i+1
          FIN
     FIN
SI derniere_position==-1 ALORS
DEBUT
     ECRIRE x ‘’n’est pas dans le tableau’’
FIN
SINON
DEBUT
     ECRIRE    "La    derniere    position    rencontree    est    ",
derniere_position)
FIN
```

Pour la première position :

```
  j←0
  trouve←Faux
  TANT_QUE j<n et trouve==Faux
     DEBUT
          SI T[j]==x ALORS
               DEBUT
               PrePos ←Vrai
               FIN
          j←j+1
          FIN
  SI trouve
     Debut
          ECRIRE x, « est à la », PrePos
     FIN
  SINON
     DEBUT
          ECRIRE « pas », x
     FIN
  FIN
```

A retenir :

Séquence d’éléments de **taille fixée** d’éléments **du même type.**

Exemple : T=[2, 5, 7, 9, 11]

Taille(T) : 5

Ecrire T[3] : 7

<u>Remplir et afficher 1 tableau et lire ses éléments :</u>

Pour i allant de 0 à Taille(T)-1

Chapitre 5 : Chaine de caractère en Python

1. Type Python *str*

- Toujours noté entre «» ou ‘ ’
- 1 valeur de type *str* = séquence de caractère
- Longueur de caractères inclus dans la séquence.
- Ch=’bonjour’
  Print (ch) → bonjour
  Ch[2] = n (3ième caractère car ça commence à 0)
- Impossible de modifier un élément d’une chaine de caractère en Python.

2. Opérateurs sur les chaines de caractères

- **Extraction** :

Permet d’extraire une sous chaine d’une chaine.
ch[n : m]= la sous chaine de ch formée par les caractères indicés de n à m-1 inclus.
Exemple : ch=’Python’
print(ch[1 :3]) → yt (de type str)
ch[n :] → de n jusqu’à la fin
ch[:m] → du début jusqu’à m
ch[::-1] → du début jusqu’à la fin en sens inverse

- **Concaténation** :

- Ch 1= « c’est »
  Ch 2 = « lundi »
  Ch = ch1+ch2 → « c’est lundi »
- Ch = ‘bonjour’ (moi jveux afficher Bonjour)
  Ch = ‘B’ + ch[1 :]

- **Duplication** :

Ch=’Hello’
Ch*2 → HelloHello

- **Test d’appartenance in :**

Ch1 in Ch2 : renvoie un booléen si la sous chaine Ch1 existe dans Ch2

3. Fonctions et méthodes prédéfinies en Python

Ch est une variable de type str.
Les fonctions suivantes renvoient toutes une nouvelle chaine et ne modifient pas la chaine originale.

- **Len(ch)** renvoie la longueur de la ch, le nombre de caractère dans ch.
  Ecrire le dernier caractère d’une chaine : print(ch[len(ch)-1])
- **Chr (i)** → renvoie le caractère dont la valeur numérique de codage est i.
- **Ord(car)** renvoie la valeur numérique du codage de car
  Car : nom d’une variable référencant 1 seul caractère
  Exemple : ord(‘a’) → 97, Chr(97) → a

- **Eval (ch)** évalue l'expression qu'on lui donne en argument qui est une chaine
  Eval ('5+7') → 12

Synthaxe pour appeler une méthode : Chaine.methode()

- **Ch.find(ch1, p)** → renvoie l'indice de la première occurrence de ch1 dans ch[p :], et renvoie -1 si ch1 n'appartient pas à ch

Exemple : ch= 'Python'
Ch.find('h') → renvoie 3

- **Ch.replace(ch1,ch2**) renvoie une chaine qui est obtenue en remplaçant dans ch toutes les apparition de ch1 par ch2.
- **Ch.count(ch1) :** renvoie le nombre d'occurrence de ch1 dans ch. Renvoie 0 si ch1 n'est pas dans ch.
- **Ch.lower()** : tout en minuscule, **ch.upper()** : tout en majuscule
- Pour connaitre les autres, taper **« dir(str) »** sur Python

Chapitre 6 : Les Listes (Python)

I. Type list

Séquence d'éléments **ordonnés** de **taille non fixe** qui **peut être de type différent.**

Exemple :

```
L=[1,2, « a », True, « cd »]
Print (L)
```

`Len(L)` : taille (nombre d'éléments) de L

```
X=len(L)
Print x      → 5
```

II. Fonction Range

`Range (n)` → créé une séquence de nombre de 0 à n-1 de type range.

Exemple :

`Range(5) (0, 1, 2, 3, 4)`

Si je veux une liste [0, 1, 2, 3, 4]

`L=list(range(5))` convertit le *range* en une liste.

Créer une liste vide :

`L=list()` ou `L=[]`

Forme générale : `range(m, n, k)`

`m` : début
`n` : fin
`k` : pas

Exemple :

`range(2, 10, 2)` affiche `2, 4, 6, 8`

```
L=list(range(13,1,-3))
Print(L) → [13,10,7,4]

L=list(range(1,13,-3))
Print(L)→ []
```

III. Accès aux éléments d'une liste

```
  L=[1, 2, True, « Bob », 5]
Indices+: 0  1   2       3     4
Indices-:-5  -4  -3      -2    -1
```

La somme des indices doit faire la taille de la liste.
Les listes sont **modifiables** (contrairement aux chaînes de caractères).

Exemple :

```
Print(L)→[1, 2, True, « Bob », 5]
Print(L[2])→ True
L[3]="Henri"
Print(L) →[1, 2, True, "Henri", 5]
Print(L[5]) → erreur "out of range"
```

`L[5]= « toto » :` ce n'est pas comme ça qu'on étend une liste, on utilise une fonction spécifique : *append*.

IV. Parcours d'une liste

1. Boucle `while` :

Exemple :

```
L=[1, 3 , True, « Bob »]
```

Je veux l'affichage suivant :

```
L[0]=1
L[1]=3
L[2]=True
L[3]= « Bob »
```

→

```
i=0
While i<len(L):
    Print ("L[", i, "]=", L[i])
    i=i+1
```

2. Boucle `For` :

Itération sur les indices :

```
For i in range(len(L)):
    Print("L[", i, "]=", L[i])
```

Exemple :

```
M=list(range(10))
For k in range(3,10,3):
     Print(L[k])
```

➔ Il va afficher le M[3], M[6] et M[9]

```
3
6
9
```

Exemple avec chaîne de caractères :

```
Ch="Hello"
For k in range(5):
     Print("letter", ch[k])
```

Itération sur les éléments de la liste :

```
L=[1, 2, True]
i=0
For x in L :
     Print("L[", i, "]=", x)
     i=i+1
```

On parcourt un par un les éléments de la liste.

```
For ele in L :
     Print(« type d'élément », ele, « est », type(ele))
```

➔ ele=1 type d'élément 1 est int

ele=2 type d'élément 2 est int

ele=True type d'élément True est bool

On peut faire la même chose avec les chaînes de caractères.

```
Mot= « coucou »
For x in mot :
     Print(« lettre », x)
```

V. Méthode des listes

```
Nom_liste.Nom_méthode([argument])
```

Les listes sont modifiables.

Toutes les méthodes sur les listes où on fait des modifications sur la liste originale, modifie la liste originales d'une manière permanente.

Méthodes renvoyant `None`, modifiant la liste :

- `L.append(objet)` → Ajoute l'objet à la fin de `L`
- `L.remove(el)` → Enlève la première occurrence de l'élément el
- `L.insert(ind, el)` → Ajoute à l'indice ind, l'élément el
- `L.reverse()` → Ecrit la liste à l'envers

- `L.sorted()` → Ordonne en ordre croissant (pour les éléments du même type)
- `L.extend(L2)` → Fait une concaténation

Méthodes renvoyant une valeur :

- `L.pop(indice)` → Enlève l'élément de l'indice indiqué et renvoie cet élément
- `L.count(el)` → Renvoie le nombre d'occurrence de l'élément
- `L.index(el)` → Renvoie l'indice de l'élément el
- `Sum(L)` → additionne tout les éléments de la liste

Chapitre 7 : Notions avancées sur les listes

I. Opérateurs de liste

1. Extractions (slices)

L[m :n :p] → **crée une nouvelle liste** qui est une partie de la liste originale.
L[::-1] → inverse la liste

Exemple : L=[2, 4, 6, 8, 10]
M=L[1 :5 :2]
Print(M)→[4, 8]

Si p≥0, il faut m<n, sinon résultat []
Si p ≤0, il faut que m>n, sinon résultat []
Exemple : L[4 : 2 : 2] → []
L [4 :4] → []
L[4 :1 :-1] → [6, 8, 10]
L[2 : 4 : -1] →[]

2. Opérateur in

x in L → True si x est dans L, False sinon
Exemple : L= liste(range(5))
print (1 in L) → True
print (5 in L) → False
print ([0, 1, 2] in L) → False (True si L=[[0, 1, 2], 3])

3. Operateurs *, +

Comme avec les str.
Exemple * : L=list(range(3))*2
Print (L) → [0, 1, 2, 0, 1, 2]
L=[0]*3
Print (L) → [0, 0, 0]
Exemple + : L =[1, 2, 3]

M=L+[4, 5]
Print(M) → [1, 2, 3, 4, 5]
Avec +, <u>on crée une nouvelle liste</u>. L d'origine n'est pas modifiée.

Action	Code	Exécution (s = "ABCDEFGHIJKL")
Extraction	`s[2:7]`	`CDEFG`
Les 4 premiers	`s[:4]`	`ABCD`
Les 4 derniers	`s[-4:]`	`IJKL`
Tous sauf les 4 premiers	`s[4:]`	`EFGHIJKL`
Tous sauf les 4 derniers	`s[:-4]`	`ABCDEFGH`
Partitionner	`s[:3], s[3:7], s[7:]`	`('ABC', 'DEFG', 'HIJKL')`
De 3 en 3	`s[::3]`	`ADGJ`
De 3 en 3 à partir de la fin	`s[::-3]`	`LIFC`
Les indices pairs	`s[::2]`	`ACEGIK`
Les indices impairs	`s[1::2]`	`BDFHJL`
Copie superficielle	`s[::]`	`ABCDEFGHIJKL`
Copie à l'envers	`s[::-1]`	`LKJIHGFEDCBA`

II. Type list : Type modifiable

Affectation d'une variable à une autre variable crée un autre nom qui partage la même valeur en mémoire. Pas de problème avec les types non modifiables (int, float, str…).

Exemple :
L1=[1, 2, 3]
L2=L1
L1.append(10)
Print("L1:", L1, "L2 : ", L2) → L1 : [1, 2, 3 10] L2 : [1, 2, 3, 10]

<u>En revanche :</u>
L1=[1, 2, 3]
L2=L1
L1=L1+[10]
Print("L1:", L1, "L2 : ", L2) → L1 : [1, 2, 3 10] L2 : [1, 2, 3]

Exemple :
L1=[1, 2, 3]
L2=L1
L1[0]=4
Print("L1:", L1, "L2 : ", L2) → L1 : [4, 2, 3] L2 : [4, 2, 3]

<u>En revanche :</u>

L1=[1, 2, 3]
L2=L1
L1= [4] +L1[1 :]
Print("L1:", L1, "L2 : ", L2) → L1 : [4, 2, 3] L2 : [1, 2, 3]

Exemple :
L1=[1, 2]
L2=L1
L1=[3, 4]
Print(L1, L2) → L1 : [3, 4] L2 : [1, 2]

En revanche :
L1=[1, 2]
L2=L1
L1[0]=3
L1[1]=4
Print(L1, L2) → L1 : [3, 4] L2 : [3, 4]

Astuce : si on ne veut pas récupérer directement les modifications sur L2 ?
→ Recopier la liste

- L1=[1, 2, 3]
  L2=list(L1) → on crée une nouvelle liste, on l'affecte à L2, L1 et L2 sont indépendantes.
  L1[0]=4
  Print(L1, L2) → L1=[4, 2, 3] L2=[1, 2, 3]
- Slice :
  L1=[1, 2, 3]
  L2=L1[:] → on crée une nouvelle liste, on l'affecte à L2, L1 et L2 sont indépendantes.
  L1.append(10)
  Print(L1, L2) → L1=[1, 2, 3, 10] L2=[1, 2, 3]

Remarque : L.sort() renvoie none, modifie L en l'ordonnant.
Sorted(Liste) : prend en argument une liste, **renvoie** une nouvelle liste L ordonnée.
Exemple : Lbis=L[:]
Lbis.sort() ⇔ Lbis=sorted(L)

Attention ! Lorsqu'une liste est parcourue par une boucle for (bornes liés aux paramètres de la liste comme la taille), il faut éviter de modifier cette liste à l'intérieure de la boucle.
Exemple :

```
L=[10, 20, -3, -4, 5]
For x in L :
        If x<0 : L.remove(x)
Print(L)
```

→ [10, 20, -4, 5] car il ne va pas lire le -4 quand il parcourt les x.

III. Passage d'argument d'une fonction

Les types int, float, str, bool : passage par valeur dans les fonctions.

Exemple :

```
Def F(ch) :
Ch=ch+ " ! "
Print(ch)
Ch = "bonjour"
F(ch)→"Bonjour!"
Print(ch)→"Bonjour"
```

Mais les listes → passage par référence → **toute modification de L, argument d'une fonction, dans cette fonction est définitive.**

Exemple :

```
Def Fonction(L, x) :
        L.append(x)
        Print(L)
L=[« a », « b »]
Fonction(L, « x ») → [« a », « b », « x »]
Print(L) → [« a », « b », « x »]
```

Si je ne veux pas de modif sur L :

```
Def Fonction(L, x) :
        Lbis=[L :]
        Lbis.append(x)
        Print(Lbis)
```

Exemple :

```
def Fct1(M) :
        M=[4]+M[1 :]
def fct2(M) :
        M[0]=4
L=[1, 2, 3]
Fct1(L)
Print(L) → [1, 2, 3]
Fct2(L)
Print(L) →[4, 2, 3]
```

2 méthodes de chaines qui permettent de passer d'une chaine à une liste et inversement :

- Chaine → liste
  Ch1.split(ch2) :
  Ch2 est par défaut un espace ou une tabulation ou un passage à la ligne.
  Méthode de chaine qui renvoie une liste dont les éléments sont obtenus en découpant les éléments de ch1 en fonction de ch2.

Exemple : ch = « aujourd'hui c'est lundi »
L= ch.split(« »)
Print(L)
[« aujourd'hui », « c'est », « lundi »]

- Liste → chaine
  Ch.join(L)
  Méthode de chaine qui prend en argument une liste et qui crée et renvoie une nouvelle chaine où les éléments de L sont concaténés par l'intermédiaire de ch.
  L=[« ohlala ! », « il », « pleut »]
  Chbis= « ». join(L)
  Print(chbis) → « ohlala ! il pleut »

Chapitre 8 : Portée des variables en Python : variable locales et globales

I. Variables locales d'une fonction

- Les arguments de la fonction
- Variables définies par une affectation dans une fonction

Elles sont accessibles depuis cette fonction mais de nulle part ailleurs (ni le programme principal, ni par une autre fonction).
La **portée** d'une variable locale est la fonction où la variable est définie.
Tout ce qui est définie au sein d'une fonction avec une affectation dedans est une variable locale.

Exemple :

```
def f() :
    Vloc=1 #variable locale, définie par une affectation dans la fonction
    Return vloc+1
Def g() :
    Vloc=10 #création d'une nouvelle variable locale nommée vloc
    Return vloc+1
```

Print(f()) → affichage 2
Print(vloc) → erreur car vloc est morte
Print(g()) → 11

Exemple :

```
Def f1() :
    Vloc=1
    Vloc=vloc+1
    Return vloc
```

Print(f1()) #il recrée la variable locale → 2

II. Variable globales

a) Lecture

Variable globale est défini e en dehors de toute fonction (donc directement dans le corps du programme ou dans l'interpréteur). Elle est accessible depuis ce même corps de programme ainsi que depuis le corps des fonctions mais **par défaut**, **uniquement en lecture dans les fonctions**, c'est-à-dire on peut récupérer sa valeur et travailler avec mais on ne peut pas la modifier dans la fonction.

Exemple :

```
Def f() :
        Vloc=vg+1
        Return vloc+1
Print (f()) → erreur car vg pas encore connu
Vg=1 #création d'une variable globale
Print(vg) → 1
Print(f()) → 3
Print(vg) → 1
```

b) Ecriture

Par défaut, une fonction ne peut pas modifier une variable globale.

Exemple :

```
Def f() :
        Vg=vg+1
        Return vg
Vg=10
```

Print (f()) → erreur car vg est une variable globale et on ne peut pas modifier sa valeur dans une fonction

Remarque : On peut autoriser l'accès en **modification** à une variable globale en indiquant explicitement à l'interpréteur son existence via le mot clé ***global***.

Exemple :

```
Def f() :
        Global vg #les modifications sur la variable globale vg seront définitives.
        Vg=vg+1
        Return vg
Vg=1
```

Print (f()) → 2
Print (vg) → 2

c) Conflit de nom

Pas d'erreur si on n'essaie pas de récupérer la valeur d'une variable globale mais simplement de l'initialiser dans une fonction.

Exemple :

```
Def f() :
        Global vg
```

Vg=2 #comme Python ne lit pas vg dans la fonction, il crée une nouvelle variable locale vg qui masque la variable globale lors de l'exécution de la fonction. Cette variable locale disparait après l'exécution de la fonction.

Return vg+1

Vg=1

Print(vg) → 1 1

Print(f()) → 3 3

Print(vg) → 1 2 #modification sur vg devient définitive à cause du mot clé globale

Conseil d'utilisation : il est déconseillé d'accéder à des variables globales directement dans des fonctions via le mot clé global. Cela réduit la lisibilité du code, et la modification d'une variable globale peut entrainer des modifications sur d'autres fonctions.

Chapitre 9 : récursivité

I. Définition

Une **fonction récursive** est une fonction contenant au moins un appel à elle-même dans son bloc d'instruction. Pour parvenir au résultat voulu elle se réemploie elle-même.

Exemple : ça ressemble aux suites

$U_0=3$

$U_n = 5U_{n-1}+2$

- Présence d'un cas particulier : U_n ne faisant pas appel à la définition mais donnant un résultat
- Un appel récursif mais avec un indice plus petit

Exemple :

```
def suite(n) :
    If n==0 : return 3
    Else: return 5*suite(n-1)+2
```

2 phases :

1/ Descente

Suite (2) → → suite (0)

2/ Montée

Suite (2) ← ← suite (0)

II. Principes généraux d'une fonction récursive

Quand écrire une fonction récursive ?

Si la résolution du problème initial s'exprime facilement en passant par la résolution d'un ou plusieurs problèmes similaires de taille inférieure.

Comment l'écrire ?

- Commencer par le cas simple (comme U_0)

- Trouver ensuite la formule récursive

Attention ! Il faut toujours avoir une instruction permettant un teste (if, boucle) et avec ce teste, on arrête (return ou ne fait rien)

Exemple : fonction prenant n comme argument et renvoyant n !

- De manière itérative

```
Def facto(n) :
        Facto = 1
        For i in range(n) :
                Facto = facto*i
        Return facto
```

- De manière récursive

```
Def facto(n) :
        If n==0 ou n==1 :
                Return (1)
        Else :
                Return (n*facto(n-1))
```

Exemple : écrire une fonction récursive qui prend en argument une chaine de caractère (ch) et qui renvoie sa longueur.

```
Def longueur(ch) :
        If ch== « » : return 0
        Else : return 1+longueur(ch[1 : ])
```

Exemple : écrire une fonction récursive qui prend en argument une chaine de caractère ch et un caractère x, qui retourne Vrai si x appartient à ch et faux sinon (interdit d'utiliser in et les méthodes des str)

```
Def test(ch, x) :
        If ch== “” : return False
        Elif ch[0]==x : return True
        Else : return (test(ch[1:], x))
```

Test (“elsa”, “l”) → test (“lsa”, “l”) → True

Exemple : écrire une fonction récursive qui prend en argument une chaine de caractère(ch) et un caractère x, qui retourne e nombre d'occurrence de x dans ch.

```
Def test(ch, x) :
        If ch== “”: return 0
        Elif ch[0]==x : return 1+test(ch[1:], x)
        Else: return (test(ch[1:], x)
```

Exemple : écrire une fonction récursive qui prend en argument une liste et un entier x, qui renvoie une liste d'entiers où on a enlevé tous les occurrences de x dans L.

```
Def elim(L, x) :
```

```
If L== [] : return L
Elif : L[0]==x :  return elim(L[1:], x)
Else: return [L[0]]+elim(L[1:], x)
```

Exemple : écrire une fonction récursive qui prend en argument une liste L et un entier x, qui renvoie -1 si x n'appartient pas à L, l'indice de la dernière occurrence de x sinon.

```
Def f(L, x) :
    If L==[] : return -1
    Elif L[len(L)-1]==x : return len(L)-1
    Else: return f(L[:len(L)-1], x)
```

III. Différents types de récursivité (simple, multiple)

Multiple : quand une fonction, lors de l'appel exécute plusieurs « sous appels » successifs à elle même.

Exemple :

$U_0=2$

$U_1=5$

$U_n=3_{Un-1}-U_{n-2}$

```
Def suite(n) :
If n==0 : return 2
Elif n==1 : return 5
Else: return(3suite(n-1)-suite(n-2)
```

La complexité algorithmique est exponentielle, faut trouver une autre façon en temps linéaire :

```
v=2
tmp=5
if n==0: return 2
elif n==1: return 5
else:
for i in range(2,n) :
    u=3v-tmp
    tmp=v
    v=u
return(u)
```

La complexité algorithmique est linéaire en nombre de n.

Récursivité terminale :

Appel récursif est la dernière instruction à être évaluée, il est seul sans autre opération → on peut faire que la descente sans la monter.

Exemple :

```
def f(n)
If n==0 : print(« partez »)
Else : print(n)
```

f(n-1)
Même sans récursivité avec une boucle :

```
Def f(n) :
    i=0
    while i<n :
        print(n-i)
        i=i+1
    print(« partez »)
```

Récursivité non terminale → récursivité terminale : avec accumulateur

Exemple : def factorielRecursivitéNonTerminale(n) :
If n<=1 : return 1
Else: return n*factoriel(n-1)

```
Def facItérative(n):
    Prod=1
    For i in range(1, n+1)
        Prod=prod*i ← accumulateur
    Return prod
```

```
Def facTerminale(n, a):
    If n==0 or n==1: return 1
    Else: return facTerminale(n-1, n*a)
```

Pour trouver factoriel (5) → facTer(5,1)

Chapitre 10 : Tuples

Tuples = nouveau type = ensemble d'éléments ordonnés pouvant être de différents types comme les listes mais **NON MODIFIABLES**, (c'est-à-dire, pas changer, ajouter, supprimer un élément).
Représentation : avec ou sans (), éléments séparés par **,**

Exemple : x=1, 2 (tuple avec 2 entiers 1 et 2)
y=(1, 2, ‘a’)
X=1 (un tuple avec 1 seul valeur)

Opérateurs et utilisation :

- **Concaténation**

X=1, 2
Y=’a’, ‘b’
Z=x+y
Print(z) ← (1, 2, ‘a’, ‘b’)

- **In**

X=1, 2
Print(1 in x) ← True
Print(‘2’ in x) ← Flase

- **Extraction (slice)**

X= 1, 2, ‘a’, ‘b’, True
Print(x[1 :4])←(2, ‘a’, ‘b’)

- **Affectation multiple :**

a, b=4,6
z=(a, b)
print(z)←(4, 6)
z=(5,6)
print(z) → (5, 6
)print(a, b) → 4, 6
x=(10, ‘toto’, 3)
u, v, w=x
print(‘u=’, u, ‘v=’, v, ‘w=’, w) → u=10 v=toto w=3

- **Echange des valeurs :**

a=5
b=10
a, b= b, a ou (a, b)=b,a ou (a,b)=(b,a)

- **Fonction renvoyant plusieurs valeurs :**

```
Def f() :
        A=10
        B=20
        Return a, b
U, v=f()
Print(u)→10
Print(v)→20
```

- **Tuples ⇔ listes**

Tuple → liste

t=(1, 2, 3)
T=list(t)

Print(T) ← [1, 2, 3]
Liste → Tuple
L=[‘a’, ‘b’, 20]
t=tuple(L)
print(l)←(‘a’, ‘b, 20)

Intérêt des tuples
- Accès plus rapide que les listes
- Renvoie plusieurs valeurs
- Pratique si on a une liste que l’on ne veut pas modifier
- Remarque : on utilise les tuples comme clés des dictionnaires si on veut avoir des clés composées.

Chapitre 11 : Dictionnaire en Python

I. Def : nouveau type

Ensemble énumérable de couples comme une liste **mais il n’y a pas d’ordre**.
Accès aux valeurs des éléments à l’aide des clés (donc il n’y a pas d’indice).
On associe une valeur à une clé et on accède à cette valeur à l’aide de sa clé.
Exemple : dico = {« pomme » : « apple », « école » : « school »} ←on a créé un dico
clé Valeur

Création de dictionnaire vide
Dico = {} ou dico = dict()

On peut initialiser directement un dictionnaire avec des éléments dedans ou bien on peut créer un dictionnaire vide et ajouter 1 par 1 les éléments.
Dico= {‘clé : ‘valeur’, ‘Zoe’ : ‘0606’, ‘tom’ : 0707}
Dico[‘Tom’]=’0909’ ← il écrase ‘0707’ et le remplace par ‘0909’

Quelques remarques sur les clés :
- Pas 2 clés identiques : s’il y’en a plusieurs, c’est la dernière qui porte
- Dans un même dictionnaire, on peut avoir des clés de différents types mais ça doit être un type non modifiable (int, float, str, tuple), (donc pas de liste).
  Par exemple, dans notre dico on aurait pu mettre comme clé, un tuple (nom, prénom), comme ça on peut avoir le numéro de 2 Zoé différentes.
- Accès à la valeur d’une clé : nom_dict[nomclé]
  Exemple : x= dico[‘Tom’]
  Print(x) ← ‘0909’
- Supprimer une clé : del nom_dict[nom_clé] ou nom_dict.pop(nom_clé) → il enlève l’élément et renvoie la valeur.
- Chaque clé doit avoir une valeur, et différentes clés peuvent avoir la même valeur.

a. Parcours d'un dictionnaire

- Parcours de l'ensemble des clés :
  Méthode `keys ()` : renvoie toutes les clés
  Cette séquence de clés est de type `dict_keys`
  Exemple:
```
cles = dico.keys()
      Cles
      Dict_keys(['Zoe', 'Tom'])
```

```
Def affichage(dico) :
     For cle in dico.keys():
          Print(cle, dico[cle])
Dico= {'Zoe' : '0606', 'Tom' :'0707' }
Affichage (dico) ←'Zoe' '0606'
                  'Tom' '0707'
```

→ Même chose sur les valeurs : méthode `values()`

- Parcours des couples (clé, valeur) :
  Méthode `items()` : renvoie une séquence de couple (clé, valeur) de type `dict_items`
  Exemple:
```
def affichage(dico) :
     For x, y in dico.items():
          Print(x,y)
```

b. Créer une liste de dictionnaire

Exemple:
```
ListeEtudiants=[]
ListeEtudiants.append({'nom':'Durand', 'prénom' : 'Alain', 'age' :
18})
ListeEtudiants.append({'nom' : 'Einstein', 'prenom' : 'Albert',
'age' : '?' })
```

c. Réduire la complexité d'une récursivité multiple

```
Connu={0 : 1, 1 :1}
Def Fibo(n) :
     If n in connu.keys():
          Return connu[n]
     Else:
          Resultat=Fibo(n-1)+Fibo(n-2)
          Connu[n]=resultat
          Return resultat
```

Gestion de fichier

Travailler avec des fichiers :
Fichier texte (info en lettres chiffres…) : nomfichier.txt
Pour désigner les fichiers par leur nom :
From os import chdir

Chdir (“users/mimoel/Documents/Python/TPGestiondeFichier”)

Ecriture dans un fichier :

- fEcriture=open(‘nomfichier.txt’, ‘w’) : crée une variable fEcriture qui permet d’accéder au nouveau fichier nomfichier.txt en mode écriture.
  Ou : fRajouter=open(‘nomfichier.txt’, ‘a’) : ça écrit à la fin sans écraser ce qu’il y a déjà
- fEcriture.write(ch) : écrit ch dans le fichier fEcriture, si on veut écrire des valeurs numériques, écrire str(34). Pas de ,
- f.close() : pour sauvegarder et fermer

Lecture depuis un fichier :

- fLecture=open(‘nomfichier.txt’, ‘r’)
- ch=fLecture.read(): envoie une chaine de caractère avec tout ce qu’il y a dans le fichier
  Ou : ch=fLecture.readline() : pour lire 1 ligne
  Ou : l=fLecture.readlines() : renvoie les données sous forme d’une liste de chaines, chaque ligne correspondant à un élément de la liste.
- fLecture.close()
- print(ch)

Exemple création fichier :

```
from os import chdir
chdir("/users/mimoel18/Documents/Python/TP gestion de fichiers")
f=open("premier.txt","w")
ch="Voici un fichier texte cree en"+"\n"+"2017"+"\n"+"a l'universite Paris-Dauphine"
f.write(ch)
f.close()
```