

## Point de cours :

### A. Procédures et fonctions

Devant un problème complexe, par exemple « construire une maison », il est nécessaire de décomposer la tâche à réaliser en sous-tâches : réaliser des plans, obtenir un permis de construire, préparer le terrain, réaliser les fondations, le gros œuvre (maçonnerie, menuiseries, charpente, couverture), le second œuvre (plomberie, électricité, raccordement aux réseaux, carrelage, peinture), les finitions, la décoration... Certaines de ces tâches doivent être effectuées dans un ordre précis. D'autres peuvent être réalisées simultanément ou dans un ordre quelconque.

**CE QU'IL FAUT SAVOIR**

En algorithmique, on cherche à décomposer la tâche à réaliser en sous-tâches plus simples. On utilise pour cela des **sous-programmes** auxquels on fait appel dans un **programme principal**.

Cette façon de concevoir un algorithme a donc de nombreux avantages :

- l'algorithme est plus clair, car la structure de l'algorithme principal est minimale, chaque sous-tâche est séparée dans un sous-programme,
- l'algorithme est plus compact, car on évite toutes les répétitions,
- l'algorithme est plus simple à contrôler (on peut tester, un par un, les sous-programmes),
- l'algorithme est plus simple à modifier (on ne modifie souvent qu'un sous-programme),
- les sous-programmes peuvent être réutilisés dans un autre algorithme qui utilise des données similaires ou réalise des tâches voisines,
- on peut concevoir des parties de l'algorithme et les tester sans attendre d'avoir tout terminé.

Ces sous-programmes existent sous deux formes : les **procédures** et les **fonctions**. Il en existe des préprogrammés mais il est possible d'en créer de nouveaux.

#### Procédures

**CE QU'IL FAUT SAVOIR**

Une **procédure** ne retourne pas de valeur.

En Python, il existe certaines procédures préprogrammées, par exemple :

- la procédure `input()`,
- la procédure `print(« bonjour »)` qui prend en **paramètre d'entrée** une chaîne de caractères.

Pseudo-code	Python
<b>Procédure</b> <code>ma_procedure(paramètre)</code> <i>instructions</i>	<code>def ma_procedure(parametre):</code> <i>instructions</i>

#### Exemple 1

La procédure suivante prend en paramètre d'entrée un entier naturel *n* et affiche *n* lignes d'étoiles en triangle, comme ci-dessous.

```
*  
**  
***  
**** ...
```

### Algorithme : affichage d'étoile

```
Procédure étoiles(n)
Pour i allant de 1 à n
    Pour j allant de 1 à i
        Afficher « * »
    Fin Pour
Fin Pour
```

### Python : affichage d'étoile

```
def etoiles(n):
    for i in range(n):
        mot=''
        for j in range(i+1):
            mot=mot+'*'
        print(mot)
```

Comme son nom l'indique, un sous-programme n'a pas vocation à être utilisé seul, mais à être appelé dans un programme principal, de la façon suivante :

### Algorithme : affichage d'étoile

```
# Sous-programme
Procédure étoiles(n)
Pour i allant de 1 à n
    Pour j allant de 1 à i
        Afficher « * »
    Fin Pour
    Retour à la ligne

Fin Pour
# Programme principal
DEBUT
    Afficher « Veuillez saisir le
    nombre de lignes souhaité »
    Saisir taille
    étoiles(taille)
FIN
```

### Python : affichage d'étoile

```
# Procédure
def etoiles(n):
    for i in range(n):
        mot=''
        for j in range(i+1):
            mot=mot+'*'
        print(mot)

# Programme principal
taille=int(input("Veuillez saisir le
nombre de lignes souhaité"))
etoiles(taille)
```

Il est à noter que lorsqu'on fait appel à une procédure dans un programme principal les paramètres d'entrée peuvent porter un nom différent que ceux utilisés dans la définition de la procédure. Ici la **variable locale** *n* (voir plus loin pour la définition d'une variable locale) correspond à la variable *taille* du programme principal.

#### Remarque :

En Python, on peut faire directement appel à la procédure dans l'interpréteur pour l'exécuter, notamment pour la tester. Pour cela, on écrit :

```
>>> etoiles(2)
```

Affichage obtenu :

```
*  
**
```

## Fonctions

### CE QU'IL FAUT SAVOIR

Une fonction est un sous-programme qui retourne une valeur.

On l'utilise donc avec une affectation, par exemple :  
*mon\_resultat* ← *ma\_fonction()*

```

Python
mot=input("Veuillez saisir un mot")
nouveau_mot=""
for i in range (len(mot)-1):
    nouveau_mot=nouveau_mot+mot[i]+"\"
nouveau_mot=nouveau_mot+mot[len(mot)-1]
print(nouveau_mot)

```

### Exemple 2 : Nombre de mots dans une chaîne de caractères

On souhaite écrire un algorithme qui, à partir d'une chaîne de caractères saisie au clavier par l'utilisateur, affiche le nombre de mots de la phrase. On considère que l'utilisateur saisit des chaînes de caractères sans ponctuation.

**Idée :** compter le nombre d'espaces dans la phrase saisie.

<b>Algorithme : nombre de mots</b>
<b>Variables numériques :</b> <i>i, cpt</i> <b>Variables alphanumériques :</b> <i>chaine</i> <b>DÉBUT</b> <b>Afficher</b> "Veuillez saisir une phrase sans ponctuation, commençant et se terminant par une lettre" <b>Saisir</b> <i>chaine</i> <i>cpt</i> $\leftarrow$ 1 <b>Pour</b> <i>i</i> allant de 0 à <i>longueur(chaine)</i> - 1 <b>Si</b> <i>chaine[i]</i> = " " <b>alors</b> <i>cpt</i> $\leftarrow$ <i>cpt</i> + 1 <b>Fin Pour</b> <b>Afficher</b> <i>cpt</i> <b>FIN</b>

```

Python
chaine=input("Veuillez saisir une phrase sans ponctuation, commençant et se terminant par une lettre")
cpt=1
for i in range (len(chaine)):
    if chaine[i]=="":
        cpt=cpt+1
print(cpt)

```

### Transtypage

Il arrive que l'on ait besoin de transformer une chaîne de caractères en nombre et réciproquement. Cette action s'appelle une opération de transtypage.

En Python :

- On peut transformer un nombre en chaîne de caractères en utilisant la fonction de transtypage **str(chaine)**.
- On peut transformer une chaîne de caractères en nombre avec la commande **int(chaine)** ou **float(chaine)**.

<b>Python</b>
>>> str(52) # transforme le nombre 52 en la chaîne de caractères '52'. >>> int('52') #transforme la chaîne de caractères '52' en le nombre entier 52. >>> float("3.14")#transforme la chaîne de caractères "3.14" en le nombre réel 3.14.

Par exemple :

- `random()` est une fonction ne prenant pas de paramètre d'entrée
- `floor(a)` est une fonction qui prend en paramètre d'entrée un nombre réel
- `randint(a,b)` est une fonction qui prend en paramètres d'entrée deux entiers naturels

Pseudo-code	Python
<b>Fonction ma_fonction(paramètre)</b> instructions <b>retourner(valeur)</b>	<b>def ma_fonction(parametre):</b> <i>instructions</i> <b>return(valeur)</b>

### Exemple 2

- La fonction suivante prend en paramètre d'entrée un tableau et retourne sa valeur maximale.

Algorithme : recherche maximum	Python : recherche maximum
Variables numériques : $T, i, \text{maximum}$ <b>Fonction maximum(<math>T</math>)</b> $\text{maximum} \leftarrow T[0]$ <b>Pour <math>i</math> allant de 0 à longueur(<math>T</math>)-1</b> <b>Si <math>T[i] &gt; \text{maximum}</math></b> $\text{maximum} \leftarrow T[i]$ <b>Fin Si</b> <b>Fin Pour</b> <b>retourner(maximum)</b>	<b>def maximum(<math>T</math>):</b> $\text{maximum}=T[0]$ for $i$ in range(len( $T$ )) : if $T[i]>\text{maximum}$ : $\text{maximum}=T[i]$ <b>return(maximum)</b>

- La fonction suivante prend en paramètre d'entrée un tableau et retourne sa valeur minimale.

Algorithme : recherche minimum	Python : recherche minimum
Variables numériques : $T, i, \text{minimum}$ <b>Fonction minimum(<math>T</math>)</b> $\text{minimum} \leftarrow T[0]$ <b>Pour <math>i</math> allant de 0 à longueur(<math>T</math>)-1</b> <b>Si <math>T[i] &lt; \text{minimum}</math></b> $\text{minimum} \leftarrow T[i]$ <b>Fin Si</b> <b>Fin Pour</b> <b>retourner(minimum)</b>	<b>def minimum(<math>T</math>):</b> $\text{minimum}=T[0]$ for $i$ in range(len( $T$ )) : if $T[i]<\text{minimum}$ : $\text{minimum}=T[i]$ <b>return(minimum)</b>

- La fonction suivante prend en paramètre d'entrée un tableau et retourne l'étendue de ses valeurs, c'est-à-dire la différence entre la valeur maximale et la valeur minimale.

**Idée :** Faire appel aux fonctions `maximum` et `minimum` déjà écrites.

Algorithme : étendue	Python : étendue
<b>Fonction étendue(<math>T</math>)</b> $m \leftarrow \text{minimum}(T)$ $M \leftarrow \text{maximum}(T)$ <b>retourner(<math>M-m</math>)</b>	<b>def etendue(<math>T</math>):</b> $m = \text{minimum}(T)$ $M = \text{maximum}(T)$ <b>return(<math>M-m</math>)</b>

## Variables locales

### CE QU'IL FAUT SAVOIR

Lorsque des variables sont définies à l'intérieur d'une procédure ou d'une fonction, elles ne sont accessibles que pour cette dernière. On dit que ce sont des **variables locales**.

Une variable locale définie dans une procédure ou une fonction sera supprimée de la mémoire après son exécution.

En Python, les variables locales sont les paramètres d'entrée et les variables affectées à l'intérieur du sous-programme.

### Exemple 3

On teste, dans l'interpréteur Python, l'exécution de cette procédure :

```
Python : affichage
def affichage():
    k=15
    print(k)
```

```
>>> affichage()
15
```

Mais, si on essaie d'afficher le contenu de la variable *k* dans l'interpréteur, un message d'erreur apparaît :

```
>>> print(k)
Traceback (most recent call last):
File "<interactive input>", line 1, in
<module>
NameError: name 'k' is not defined
```

La variable *k* définie précédemment n'existe qu'à l'intérieur de la procédure **affichage**. La variable *k* est une variable locale à la procédure **affichage**.

## Variables globales

### CE QU'IL FAUT SAVOIR

Les variables définies à l'extérieur d'une procédure ou d'une fonction sont appelées **variables globales**. Elles sont accessibles à l'intérieur du sous-programme mais ce dernier ne peut pas les modifier.

En Python, les variables globales sont les variables utilisées dans un sous-programme qui ne sont pas des variables locales, c'est-à-dire les variables qui ne sont pas des paramètres d'entrée ou qui ne sont pas affectées à l'intérieur du sous-programme.

### Exemple 4

```
Python : affichage2
def affichage_bis():
    print("La valeur de k est :",k)
    k=2
    affichage_bis()
    k=k+15
    affichage_bis()
```

À l'exécution de ce programme, on obtient :

```
>>>
La valeur de k est : 2
La valeur de k est : 17
```

Dans cet exemple, on peut accéder à la valeur de la variable *k* à l'intérieur de la procédure **affichage\_bis** mais pour la modifier, on écrit les instructions dans le programme principal. La variable *k* est une variable globale.

À l'intérieur d'un sous-programme, ce sont les variables définies localement qui ont la priorité. Pour modifier cela, on peut demander au sous-programme de traiter une variable comme une variable globale à l'aide de l'instruction `global`.

### Exemple 5

Python : affichage3_1	Python : affichage3_2
<pre>def affichage_3():     g=17     print("Dans la procedure : g=",g)      g=2     print("Avant la procedure : g=",g)     affichage_3()     print("Apres la procedure : g=",g)</pre>	<pre>def affichage_ter():     global g     g=g+15     print("Dans la procedure : g=",g)      g=2     print("Avant la procedure : g=",g)     affichage_ter()     print("Apres la procedure : g=",g)</pre>

```
>>>
Avant la procedure : g= 2
Dans la procedure : g= 17
Apres la procedure : g= 2
```

```
>>>
Avant la procedure : g= 2
Dans la procedure : g= 17
Apres la procedure : g= 17
```

Dans le programme Affichage3\_1, il y a deux variables *g*, l'une locale à la procédure **affichage\_3** et l'autre de portée globale dans le programme principal. À l'issue de l'exécution de la procédure, le contenu de la variable *g* du programme principal n'a pas été modifié.

Dans le programme Affichage3\_2, la variable *g* est une variable globale, la modification effectuée dans la procédure **affichage\_ter** est prise en compte dans le programme principal.

### Exemple 6

Python : affichage4
<pre>def affichage_quater():     g[0]=g[0]+15     print("Dans la procedure : g=",g)      g=[0]*2 # initialisation du tableau     g[0]=2     print("Avant la procedure : g=",g)     affichage_quater()     print("Apres la procedure : g=",g)</pre>

On obtient :

```
>>>
Avant la procedure : g=[2, 0]
Dans la procedure : g=[17, 0]
Apres la procedure : g=[17, 0]
```

## Applications :

### Application 1 : produit

Écrire une fonction prenant en paramètres d'entrée deux nombres entiers naturels et qui retourne leur produit.

### Application 2 : affichage message

Écrire une procédure qui prend en paramètres d'entrée un nombre entier naturel *n* et une chaîne de caractères *message* et affiche *n* fois la variable *message*.

### Application 3 : calcul de moyenne

Écrire une fonction prenant en paramètre d'entrée un tableau de nombres et qui retourne la moyenne des valeurs contenues dans le tableau.

## TP : Liste des diviseurs d'un nombre entier naturel

### A. Travail par écrit

On considère la fonction suivante écrite en Python :

Python : fonction F	
def F(n)	
1.    div = 1	
2.    Liste_div = []	
3.    while div <= n :	
4.     if n%div == 0 :	
5.       Liste_div = Liste_div + [div]	
6. <i>div = div + 1</i>	
7.    return (Liste_div)	

1. Lister les variables ainsi que leur type.
2. Quelle instruction manque-t-il à la ligne 6 ? Que se passe-t-il si on ne la rajoute pas ?
3. Que retourne cette fonction si l'utilisateur saisit n=60 ?
4. Quel est le rôle de cette fonction ?

### B. Travail sur poste informatique

On souhaite écrire une fonction qui prend en paramètres d'entrée deux nombres entiers naturels a et b et qui retourne le PGCD de ces deux nombres. Pour cela on compare les listes des diviseurs de chacun de ces nombres : le PGCD est le plus grand élément commun à ces deux listes.

1. Que doit retourner cette fonction pour a=60 et b=24 ?
2. À l'aide de la fonction déjà implémentée, écrire la fonction PGCD décrite ci-dessus.
3. **Pour aller plus loin** : écrire une fonction qui prend en paramètres d'entrée trois entiers naturels a, b et c qui retourne le plus grand diviseur commun à ces trois nombres.