2025

ABN ACADEMY

Fiche de Révision En Informatique

Muḥammad ibn Mūsā al-Khwārizmī

Sections :

Mathématiques, Sciences Expérimentales, Sciences Techniques.

Ce résumé est élaboré en stricte conformité avec les conventions algorithmiques du programme officiel tunisien. Il présente, à l'aide d'exemples en algorithme et en Python, les principaux modules étudiés, facilitant ainsi la compréhension et la révision des concepts informatiques.

Anis Ben Nacib

Les traitements récurrents et arithmétiques

Fonction qui permet de calculer **la somme des chiffres** d'un entier strictement positif x.

```
fonction somme_chiffres(x : entier) : entier
Début
    Ch ← Convch (x)
    S ← 0
    Pour i de 0 à Long(Ch)-1 faire
            S ← S + Valeur (Ch[i])
    finPour
    Retourner S
Fin
```

```
def somme_chiffres(x):
    ch = str(x)
    s = 0
    for i in range(len(ch)):
        s = s + int(ch[i])
    return s
```

Fonction qui permet de calculer **la somme des diviseurs** d'un entier strictement positif x (x non inclus).

```
fonction somme_diviseurs(x : entier) : entier
Début
    S ← 0
    Pour d de 1 à x div 2 faire
            Si x mod d = 0 alors
                  S ← S + d
            finSi
    finPour
    Retourner S
Fin
```

```
def somme_diviseurs(x):
    S = 0
    for d in range(1,x//2+1):
        if x % d == 0:
            S = S + d
    return S
```

Fonction qui permet de calculer **la factorielle** d'un entier strictement positif n.

```
fonction Factorielle (n: entier) : entier
Début
    F ← 1
    Pour i de 2 à n faire
            F ← F * i
    fin Pour
    Retourner F
Fin
```

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

non inclus

$N! = 1*2*3*....*(N-1)*N$

$5! = 1*2*3*4*5$

Fonction qui permet de calculer **la puissance x^n** de deux entiers positifs x et n.

```
fonction Puissance(x,n: entier) : entier
Début
    P ← 1
    Pour i de 1 à n faire
            P ← P * x
    fin Pour
    Retourner P
Fin
```

```
def Puissance(x,n):
    P = 1
    for i in range(1,n+1):
        P = P * x
    return P
```

non inclus

$X^n = \underbrace{X*X*X*....*X}_{\text{n fois}}$

Fonction qui permet de calculer le **PGCD** de deux entiers positifs non nuls a et b.

```
fonction PGCD(a,b: entier) : entier
Début
    tant que (a ≠ b) faire
        si a > b alors
            a ← a - b
        sinon
            b ← b - a
        finSi
    fin Tant que
    Retourner a
Fin
```

```
def PGCD(a,b):
    while a!=b:
        if a>b:
            a = a - b
        else:
            b = b - a
    return a
```

Fonction qui permet de calculer le **PGCD** de deux entiers positifs non nuls a et b. (**Méthode d'Euclide**)

```
fonction PGCD_Euclide(a, b: entier) : entier
Début
    Tant que (b ≠ 0) faire
        r ← a mod b
        a ← b
        b ← r
    fin Tant_que
    Retourner a
Fin
```

```
def PGCD_Euclide(a,b):
    while b!=0:
        r = a % b
        a = b
        b = r
    return a
```

Fonction qui permet de calculer le **PPCM** de deux entiers positifs a et b.

```
fonction PPCM(a,b: entier) : entier
Début
    P ← a
    Tant que (P mod b ≠ 0) faire
        P ← P + a
    fin Tant_que
    Retourner P
Fin
```

```
def PPCM(a,b):
    P = a
    while P % b != 0:
        P = P + a
    return P
```

Fonction qui permet de calculer le **PPCM** de deux entiers positifs a et b. (**version optimisée**)

```
fonction PPCM(a,b: entier) : entier
Début
    Si a > b alors
        max ← a
        min ← b
    sinon
        max ← b
        min ← a
    finSi
    P ← max
    Tant que (P mod min ≠ 0) faire
        P ← P + max
    fin Tant_que
    Retourner P
Fin
```

```
def PPCM(a,b):
    if a>b:
       Max = a
       Min = b
    else:
       Max = b
       Min = a
    P = Max
    while P % Min != 0:
        P = P + Max
    return P
```

Fonction qui permet de tester de tester **la primalité** d'un entier strictement positif(**Premier ou non**).

NB. Un nombre premier est un nombre dont ses seuls diviseurs sont 1 et lui-même

```
fonction Premier(x : entier) : booléen
début
    ok ← vrai
    d ← 2
    tant que (d ≤ x div 2) et (ok) faire
        si x mod d = 0 alors
            ok ← faux
        sinon
            d ← d + 1
        fin_si
    fin_tant_que
    retourner (ok) et (x>1)
fin
```

Par définition 1 n'est pas un nombre premier

```
#1ère Méthode
def Premier(x):
    ok = True
    d = 2
    while(d<= x//2) and ok:
        if x % d != 0:
            d = d + 1
        else:
            ok = False

    return ok and x>1
```

```
#2ème méthode
fonction Premier(x : entier) : booléen
début
    d ← 2
    tant que(d ≤ x div 2) et (x mod d ≠ 0 )faire
        d ← d + 1
    fin_tant_que
    retourner (d > x div 2) et (x>1)
fin
```

```
#2ème Méthode
def Premier(x):
    d = 2
    while(d<= x//2 and x % d != 0):
        d = d + 1

    return (d > x // 2) and (x>1)
```

Par définition 1 n'est pas un nombre premier

Fonction qui permet de décomposer un entier x donné en produit de ses **facteurs premiers**.

```
fonction facteurs_premiers (x : entier) : chaîne
début
    fp ← ""
    d ← 2
    Tant que (x>1) faire
        si x mod d = 0 alors
            fp ← fp + convch(d) + "*"
            x  ← x div d
        sinon
            d ← d + 1
        fin_si
    fin tant_que
    retourner sous_chaîne(fp, 0, long(fp)-1)
fin
```

non inclus

```
def facteurs_premiers(x):
    d = 2
    fp = ""
    while x>1:
        if x % d == 0:
            fp = fp + str(d) + "*"
            x = x // d
        else:
            d = d + 1
    return fp[:len(fp)-1]
```

Les traitements sur les chaînes de caractères

Fonction qui permet de vérifier si une chaîne de caractère non vide est formée **uniquement** par des **lettres majuscules**.

```
fonction verif_maj(ch : chaîne) : booléen
début
    ok ← Vrai
    i ← 0
    tant que (i<long(ch)) et ok  faire
        si ch[i] ∈ ["A".."Z"] alors
            i ← i + 1
        sinon
            ok  ← Faux
        fin_si
    fin_tant_que
    retourner ok
fin
```

```
def verif_maj(ch):
    Ok = True
    i = 0
    while (i<len(ch)) and Ok:
        if "A"<=ch[i]<="Z":
            i = i + 1
        else:
            Ok = False
    return Ok
```

```
#2ème méthode
fonction verif_maj(ch : chaîne) : booléen
début
   i ← 0
   tant que (i<long(ch)) et (ch[i]∈["A".."Z"])faire
        i ← i + 1
    fin tant_que
    retourner i = long(ch)
fin
```

```
#2ème méthode
def verif_maj(ch):
    i = 0
    while (i<len(ch) and "A"<=ch[i]<="Z"):
        i = i + 1
    return i == len(ch)
```

Fonction qui permet de vérifier si une chaîne de caractère non vide est formée **uniquement** par des **lettres**.

```
fonction verif_lettres(ch : chaîne) : booléen
début
  ch ← Majus(ch)
  i ← 0
  tant que(i<long(ch)) et (ch[i] ∈ ["A"..."Z"])faire
        i ← i + 1
  fin tant_que
  retourner i = long(ch)
fin
```

```
def verif_lettres(ch):
    ch = ch.upper()
    i = 0
    while(i<len(ch) and "A"<=ch[i]<="Z"):
         i = i + 1
     return i == len(ch)
```

Fonction qui permet de vérifier si une chaîne de caractère non vide est formée **uniquement** par des **lettres et pouvant contenir des espaces**.

```
fonction verif_lettres_espace(ch:chaîne):booléen
début
    ok ← Majus(ch)
    ok ← Vrai
    i ← 0
    tant que (i<long(ch)) et ok  faire
        si (ch[i] ∈ ["A"..."Z"]) ou (ch[i]=" ") alors
            i ← i + 1
        sinon
            ok  ← Faux
        fin_si
    fin_tant_que
    retourner ok
fin
```

```
def verif_lettres_espaces(ch):
    ch = ch.upper()
    Ok = True
    i = 0
    while (i<len(ch)) and Ok:
        if "A"<=ch[i]<="Z" or ch[i]==" ":
            i = i + 1
        else:
            Ok = False
    return Ok
```

#2ème méthode

```
fonction verif_lettres_espace(ch:chaîne):booléen
début
  ch ← Majus(ch)
  i ← 0
  tant que(i<long(ch))et(ch[i]∈["A"..."Z"] ou ch[i]=" ")faire
        i ← i + 1
  fin tant_que
  retourner i = long(ch)
fin
```

#2ème méthode

```
def verif_lettres_espaces(ch):
   ch = ch.upper()
   i = 0
   while(i<len(ch) and ("A"<=ch[i]<="Z" or ch[i]==" "):
        i = i + 1
   return i == len(ch)
```

Fonction qui permet de vérifier si une chaîne de caractère non vide est formée uniquement par des **caractères distincts**.

#1ère méthode

```
fonction verif_caracteres_distincts(ch:chaîne): booléen
début
   Ok ← vrai
   i ← 0
   tant que (i<long(ch))et (Ok) faire
      si (Pos(ch[i],ch)=i) alors
         i ← i + 1
      sinon
         Ok ← faux
      fin_si
   fin_tant_que
   retourner Ok
fin
```

#1ère méthode

```
def verif_caracteres_distincts(ch):
    Ok = True
    i = 0
    while (i<len(ch)) and Ok:
        if (ch.find(ch[i]) == i):
            i = i + 1
        else:
            Ok = False
    return Ok
```

```
#2ème méthode
fonction verif_carcteres_distincts(ch:chaîne): booléen
début
   i ← 0
   tant que (i<long(ch))et (Pos(ch[i],ch)=i)faire
        i ← i + 1
   fin_tant_que
   retourner i = long(ch)
fin
```

```
#2ème méthode
def verif_caracteres_distincts(ch):
    i = 0
    while(i<len(ch))and(ch.find(ch[i]) == i):
        i = i + 1
    return i == len(ch)
```

Fonction qui permet de calculer **le nombre de voyelles** dans une chaîne **ch**.

```
fonction nb_voyelles(ch : chaîne) : entier
début
    ch ← Majus(ch)
    nb ← 0
    pour i de 0 à long(ch)-1 faire
        si (ch[i] ∈ {"A","E","I","O","U","Y"}) alors
            nb ← nb + 1
        fin_si
    fin_pour
    retourner nb
fin
```

```
def nb_voyelles(ch):
    ch = ch.upper()
    nb = 0
    voy = "AEIOUY"
    for i in range(len(ch)):
        if voy.find(ch[i])!= -1:
            nb = nb + 1
    return nb
```

```
#Ou bien
if ch[i] in ["A","E","I","O","U","Y"]:
```

Fonction qui prend une chaîne de caractères non vide composée uniquement par **des lettres puis de la convertir en minuscule.**

```
fonction Minus(ch : chaîne) : chaîne
début
    chmin ← ""
    pour i de 0 à Long(ch)-1 faire
        si (ch[i] ∈ ["A"..."Z"] alors
            chmin ← chmin + chr(ord(ch[i])+32)
        sinon
            chmin ← chmin + ch[i]
        fin_si
    fin_pour
    retourner chmin
fin
```

Ord("A") = 65

Ord("a") = 97

32

```
#sans l'utilisation la méthode lower()
def Minus(ch):
    chmin=""
    for i in range(len(ch)):
        if "A"<=ch[i]<="Z":
            chmin = chmin+chr(ord(ch[i])+32)
        else:
            chmin = chmin + ch[i]
    return chmin
```

L'algorithme d'une procédure qui prend en paramètre une chaîne de caractères non vide composée uniquement par **des lettres majuscules puis d'afficher l'ordre alphabétique de chaque lettre.**

```
procédure rang_alphabetique(ch : chaîne)
début
    pour i de 0 à long(ch)-1 faire
        Ecrire(ch[i],"=>",ord(ch[i])-64)
    fin_pour
fin
```

Lettre	Rang	Comment ?
"A"	1	65 -64
"B"	2	66-64
"C"	3	67-64
...		
"Z"	26	90-64

Avec ord("A") = 65, ord("B")=66,....

```
def rang_alphabetique(ch):
    for i in range(len(ch)):
        print(ch[i],"=>",ord(ch[i])-64)

#Exemple d'exécution :
```

```
rang_alphabetique("OMAR")
O => 15
M => 13
A => 1
R => 18
```

Les traitements sur les tableaux

Exemples de déclarations de tableaux en Python (avec **T** un tableau de **n** entiers (2≤**n**≤20)

```
#Appel
from numpy import array
…
…
n = saisir()
T=array([int()]*n)
remplir(T,n)
```

```
#Appel
from numpy import array
T=array([int()]*20) #avec 20 la taille maximale
…
…
n = saisir()
remplir(T,n)
```

Procédure qui permet de remplir un tableau **T** par **n** entiers **positifs non nuls.**

```
procédure remplir(@T: Tab, n:entier)
début
    pour i de 0 à n-1 faire
        répéter
            écrire("T[", i, "]: ")
            lire(T[i])
        jusqu'à  (T[i] > 0)
    fin_pour
fin
```

```
def remplir(T,n):
    for i in range(n):
        T[i] = int(input("T["+str(i)+"]:"))
        while T[i]<=0:
            T[i] = int(input("T["+str(i)+"]:"))
```

Procédure qui permet de remplir d'une manière **aléatoire** un tableau **T** par **n** entiers de **trois chiffres**.

```
procédure remplir(@T: Tab, n:entier)
début
    pour i de 0 à n-1 faire
        T[i] ← aléa(100, 999)
    fin_pour
fin
```

```
#Importer randint de module random
from random import randint

def remplir(T,n):
    for i in range(n):
        T[i] = randint(100,999)
```

Procédure qui permet de remplir un tableau **T** par **n** entiers **positifs non nuls** ordonnés dans l'ordre **croissant**.

```
procédure remplir(@T: Tab, n:entier)
début
    répéter
        écrire("T[0]: ")
        lire(T[0])
    jusqu'à  T[0] > 0
    pour i de 1 à n-1 faire
        répéter
            écrire("T[", i, "]: ")
            lire(T[i])
        jusqu'à  (T[i] > T[i-1])
    fin_pour
fin
```

```
def remplir(T,n):
    T[0] = int(input("T[0]:"))
    while T[0]<0:
        T[0] = int(input("T[0]:"))

    for i in range(1,n):
       T[i] = int(input("T["+str(i)+"]:"))
        while not(T[i]>T[i-1]):
            T[i] = int(input("T["+str(i)+"]:"))
```

Procédure qui permet de remplir un tableau **T** par **n** entiers positifs non nuls **distincts**.

```
procédure remplir(@T: Tab, n:entier)
début
  pour i de 0 à n-1 faire
     répéter
         écrire("T[", i, "]: ")
         lire(T[i])
     jusqu'à (T[i]>0)et((i=0) ou (distincts(T,i)))
    fin_pour
fin
```

```
def remplir(T,n):
  for i in range(n):
    T[i] =int(input("T["+str(i)+"]:"))
    while not(T[i]>0 and (i==0 or distincts(T,i))):
        T[i] =int(input("T["+str(i)+"]:"))
```

L'appel à **distincts** se fait uniquement pour **i ≠ 0**, car pour **i = 0**, aucun élément précédent n'existe pour effectuer la comparaison

```
fonction distincts(T : Tab, i:entier) : booléen
début
    j ← 0
    tant que (j<i) et (T[j] ≠ T[i]) faire
        j ← j + 1
    fin_tant_que
    retourner j=i
fin
```

```
def distincts(T,i):
    j = 0
    while(j<i) and (T[j]!=T[i]):
        j = j + 1
    return j==i
```

```
##2ème méthode
fonction distincts(T : Tab, i:entier) : booléen
début
    ok ← vrai
    j ← 0
    tant que (j<i) et (ok) faire
        si (T[j] = T[i]) alors
          ok ← faux
        sinon
          j ← j + 1
        fin_si
    fin_tant_que
    retourner ok
fin
```

```
#2ème méthode
def distincts(T,i):
    ok = True
    j = 0
    while(j<i) and (ok):
        if T[j]==T[i]:
            ok = False
        else:
            j = j + 1
    return ok
```

Procédure qui permet de remplir un tableau **T** par **n** chaînes de caractères **non vides**, chacune contenant jusqu'à **30 lettres** au maximum.

```
procédure remplir(@T: Tab, n:entier)
début
   pour i de 1 à n-1 faire
      répéter
        écrire("T[", i, "]: ")
        lire(T[i])
      jusqu'à (0<long(T[i])≤30)et(verif_lettres(T[i]))
    fin_pour
fin
```

Voir page 4

```
def remplir(T,n):
    for i in range(1,n):
       T[i] = input("T["+str(i)+"]:")
       while not(0<len(T[i])<=30 and
verif_lettres(T[i])):
            T[i] = input("T["+str(i)+"]:")
```

Fonction qui prend en paramètres un tableau **T** de **n** réels puis de retourner la somme de ses éléments.

```
fonction somme_tab(T : Tab, n : entier) : entier
début
    s ← 0
    pour i de 0 à n-1 faire
        s ← s + T[i]
    fin_pour
    retourner s
fin
```

```
def somme_tab(T,n):
    s = 0
    for i in range(n):
        s = s + T[i]
    return s
```

Fonction qui permet de déterminer **le nombre d'occurrence** d'un entier x dans un tableau T de taille **n**.

```
fonction nb_occurrence(T:Tab, n, x:entier):entier
début
    nb ← 0
    pour i de 0 à n-1 faire
        si T[i] = x alors
            nb ← nb + 1
        fin_si
    fin_pour
    retourner nb
fin
```

```
def nb_occurrences(T,n,x):
    nb = 0
    for i in range(n):
        if T[i]==x:
            nb = nb + 1
    return nb
```

Fonction qui permet de déterminer **la valeur maximale** dans un tableau d'entiers **T** de taille **n**.

```
fonction max_tab(T : Tab, n: entier) : entier
début
    max ← T[0]
    pour i de 1 à n-1 faire
        si T[i] > max alors
            max ← T[i]
        fin_si
    fin_pour
    retourner max
fin
```

```
def max_tab(T,n):
    Max = T[0]
    for i in range(1,n):
        if T[i]> Max:
            Max = T[i]
    return Max
```

Fonction qui permet de déterminer **la position de la valeur minimale** dans un tableau d'entiers **T** de taille **n**.

```
fonction pos_min_tab(T : Tab, n: entier) : entier
début
    pmin ← 0
    pour i de 1 à n-1 faire
        si T[i] < T[pmin] alors
            pmin ← i
        fin_si
    fin_pour
    retourner pmin
fin
```

```
def pos_min_tab(T,n):
    pmin = 0
    for i in range(1,n):
        if T[i]<T[pmin]:
            pmin = i
    return pmin
```

Fonction qui implémente un algorithme de **recherche séquentielle** pour trouver un élément **x** dans un tableau **T** de taille **n**

```
fonction Recherche_sequentielle T:tab, n,x:entier):Booléen
début
    Trouve ← faux
    i  ← 0
    tant que (i<n) et (Trouve=Faux) faire
        si  (T[i] = x) alors
            Trouve ← vrai
        sinon
            i ← i + 1
        fin_si
    fin_tant_que
    Retourner Trouve
fin
#deuxième méthode
fonction Recherche_sequentielle(T:tab, n,x:entier):Booléen
début
    Trouve ← faux
    i  ← 0
    Répéter
        si  (T[i] = x) alors
            Trouve ← vrai
        sinon
            i ← i + 1
        fin_si
    Jusqu'à (Trouve) ou (i = n)
    Retourner Trouve
fin
```

```
def Recherche_sequentielle (T,n,x):
    Trouve = False
    i = 0
    while (i<n) and (Trouve==False):
        if T[i]==x:
            Trouve = True
        else:
            i = i + 1
    return Trouve
#deuxième méthode
def Recherche_sequentielle (T,n,x):
    i = 0
    while (i<n) and (T[i]!=x):
        i = i + 1
    return i<n
```

Fonction qui implémente un algorithme de **recherche dichotomique** pour trouver un élément **x** dans un tableau **T** de taille **n**

```
fonction Recherche_dichotomique (T:tab ; n,x:entier):Booléen
Début
    d ← 0
    f ← n-1
    Trouve ← faux
    Tant Que (d ≤ f) et (Trouve = faux) faire
        Milieu ← (d + f) Div 2
        Si (T[milieu] = x) Alors
            Trouve ← vrai
        Sinon Si (T[milieu] > x) Alors
                f ← milieu – 1
            Sinon
                d ← milieu+1
        Fin_si
    fin_Tant_Que
    Retourner Trouve
Fin
```

```
def Recherche_dichotomique(T, n, x):
    d = 0
    f = n - 1
    Trouve = False
    while (d <= f) and (Trouve==False):
        milieu = (d + f) // 2
        if T[milieu] == x:
            Trouve = True
        elif T[milieu] < x:
            d = milieu + 1
        else:
            f = milieu - 1
    return Trouve
```

Procédure qui implémente un algorithme de tri par **sélection** dans l'ordre croissant d'un tableau **T** de taille **n.**

```
procédure Tri_Selection (@T : Tab , n : Entier )
début
    Pour i de 0 à n-2 Faire
        posmin ← i
        Pour  j de i+1 à n-1 Faire
                si T[j] < T[posmin] Alors
                        posmin ←  j
                fin_Si

        fin_Pour
        Si posmin ≠ i Alors
            Aux  ← T[i]
            T[i] ← T[posmin]
            T[posmin] ← Aux
        Fin_Si
    Fin_Pour
Fin
```

```
def Tri_Selection(T, n):
    for i in range(n-1):
        posmin = i
        for j in range(i+1, n):
            if T[j] < T[posmin]:
                posmin = j
        if posmin != i:
            aux  ← T[i]
            T[i] ← T[posmin]
            T[posmin] ← aux
```

Procédure qui implémente un algorithme de **tri à bulle** dans l'ordre croissant d'un tableau **T** de taille **n.**

```
procédure Tri_Bulles (@T : Tab , n : Entier)
début
    Répéter
            Permut ← Faux
            Pour i de 0 à n-2 Faire
                Si (T[i] > T[i +1] ) Alors
                    Aux  ← T[i]
                    T[i] ← T[i+1]
                    T[i+1] ← Aux
                    Permut ← Vrai
                fin_Si
            fin_Pour
   Jusqu'à (Permut = Faux)
fin
```

```
def tri_bulles(T, n):
    permut = True
    while permut:
        permut = False
        for i in range(n - 1):
            if T[i] > T[i+1]:
               aux = T[i]
               T[i] = T[i+1]
               T[i+1] = aux
               permut = True
```

Procédure qui implémente un algorithme de **tri par insertion** dans l'ordre croissant d'un tableau **T** de taille **n.**

```
procédure Tri_Insertion (@T : Tab , n :entier)
début
    Pour i de 1 à n-1 faire
        aux ← T[i]
        j ← i
        Tant Que (j>0) et (T[j-1]>aux) faire
            T[j] ← T[j-1]
            j ← j-1
        fin_Tant_Que
        T[j] ← Aux
    Fin_Pour
fin
```

```
def Tri_Insertion(T, n):
    for i in range(1, n):
        aux = T[i]
        j = i
        while j > 0 and T[j-1] > aux:
            T[j] = T[j-1]
            j = j-1
        T[j] = aux
```

Annexe : Les fonctions prédéfinies

✪ Les fonctions sur les types numériques

Syntaxe		Rôle de la fonction
Algorithme	**Python**	
Abs (x)	**abs** (x)	Retourne la valeur absolue de **x**.
Aléa (Vi,Vf)	**randint** (Vi,Vf)	Retourne un entier aléatoire appartenant à l'intervalle **[Vi , Vf]**. Nb. En python elle nécessite l'importation de la bibliothèque **random.**
Arrondi (x)	**round** (x)	Retourne l'entier le plus proche de **x**.
Ent (x)	**int** (x) ou **floor**(x)	Retourne la partie entière de **x**.
RacineCarré (x)	**sqrt** (x)	Retourne la racine carrée de **x** (**x** doit être positif). **Nb.** En python elle nécessite l'importation de la bibliothèque **math**

✪ Les fonctions sur le type caractère

Syntaxe		Rôle de la fonction
Algorithme	**Python**	
Ord (C)	**ord** (C)	Retourne le code ASCII du caractère **C**.
Chr (code)	**chr** (code)	Retourne le caractère dont le code ASCII est **code**.

✪ Les fonctions sur le type chaînes de caractères

Syntaxe		Rôle de la fonction
Algorithme	**Python**	
Long (ch)	**len** (ch)	Retourne le nombre de caractères de la chaîne **ch**.
Majus (ch)	ch.**upper** ()	Retourne l'équivalent de la chaîne **ch** en majuscule.
Pos (ch1, ch2)	ch2.**find** (ch1)	Retourne la première position de la chaîne **ch1** dans la chaîne**ch2**, sinon elle retourne **-1**.
Convch (x)	**str** (x)	Retourne la conversion d'un nombre x en une chaîne de caractères.
Valeur (ch)	**int** (ch) **float** (ch)	Retourne la conversion d'une chaîne ch en une valeur numérique, si c'est possible.
Estnum (ch)	ch.**isdecimal**()	Retourne **Vrai** si la chaîne **ch** est convertible en une valeur numérique, elle retourne **Faux** sinon.
Sous_Chaine(ch , d , f)	ch **[d : f]**	Retourne une partie de la chaîne **ch** à partir de la position **d** jusqu'à la position **f** (**f exclue**).
Effacer(ch , d , f)	ch **[: d]** + ch **[f :]**	Efface des caractères de la chaîne **ch** à partir de la position **d** jusqu'à la position **f** (**f exclue**).

Annexe : Interface graphique

Nom de l'objet	Méthodes à utiliser	Rôle
Label	setText	Modifie le texte de l'étiquette.
TextEdit	toPlainText	Renvoie le texte contenu dans le champ de texte
	setText	Modifie le texte du champ par celui en argument
	clear	Efface le contenu du champ de texte.
LineEdit	text	Renvoie le texte contenu dans le champ de texte
	setText	Modifie le contenu du champ de texte par le texte fourni en argument.
	clear	Efface le contenu du champ de texte
PushButton	clicked , connect	Déclenche une action ou un événement spécifique lorsqu'il est pressé (cliqué).