# Les bases

## *Le traditionnel Hello World*

Hello world (traduit littéralement en français par « Bonjour le monde ») sont les mots traditionnellement écrits par un programme informatique simple dont le but est de faire la démonstration rapide de son exécution sans erreur. (wikipédia)
Voici le code en Python:

```
print("Hello World")
```

## *Les variables...*

`print(...)` est une fonction qui permet l'affichage en console.
`print("Hello World")` permet l'affichage en console de la **chaîne de caractères**: `Hello World`
Une chaîne de caractères s'écrit entre **guillemets ou quotes** : Les codes suivants donnent les mêmes affichages:

```
print("Ceci est une chaîne de caractères")
print('Ceci est une chaîne de caractères')
```

**Une variable comme son nom l'indique c'est variable...**
Par exemple : Le code suivant stocke dans la variable maphrase la phrase précédente et l'affiche.
**L'opérateur (=)** permet **L'affectation** d'une valeur dans une variable

```
maphrase = 'Ceci est une chaîne de caractères'
print(maphrase)
```

### ? Question 1:

Que se passe t-il si on inverse les deux lignes de codes ? Et pourquoi ?

```
print(maphrase)
maphrase = 'Ceci est une chaîne de caractères'
```

On peut modifier une variable à tout moment: Tester ce programme

```
maphrase = 'Ceci est une chaîne de caractères'
print(maphrase)
maphrase = 'Ceci est une autre chaîne de caractères'
print(maphrase)
maphrase = 23
print(maphrase)
```

*Remarque : Si au final la variable maphrase doit contenir le nombre 23, le choix de son nom n'est pas très judicieux. Il faudra par la suite choisir des noms de variables appropriés...*

## *Les types simples...*

En informatique on manipule **des données**, elles peuvent être de **types simples** (nombres, mots, booléens(Vrai ou Faux)) ou comme nous le verrons plus tard de **types construits** (listes, tuples, dictionnaires) et enfin **structurées** (tableaux, base de données).

**Les nombres** : Il y en a de deux types

- Les entiers (positifs ou négatifs), leurs types est **int** (pour integer).
- Les nombres à virgule, leurs types est **float** (pour nombre à virgule flottante).
  *Remarque : Du fait de la limitation en mémoire, en informatique tous les réels sont des décimaux. ( cela s'arrête à un moment...)*

**Les chaînes de caractères**
Le type d'une chaîne de caractère est **str** ( pour String )

**Les booléens**
Le type d'un booléen est **bool** ( pour boolean )
Un booléen ne peut avoir que deux valeurs : **True** ou **False**.

**En Python, le typage d'une variable est implicite** : c'est à dire que le type d'une variable est le type de ce qu'elle contient.
Dans le code suivant , on a affecté à des variables des valeurs et on affiche leur type.

```
n=12
x=1.23
maphrase='Ceci est une chaîne de caractères'
test=True

print(type(n))
print(type(x))
print(type(maphrase))
print(type(test))
```

On obtient :
<class 'int'> signifiant que n est du type :
<class 'float'> signifiant que x est du type :
<class 'str'> signifiant que maphrase est du type :
<class 'bool'> signifiant que test est du type :
*Remarque : Le type d'une variable peut évoluer au cours d'un programme au gré des affectations qu'elle subit...*

**La fonction input()** : permet à l'utilisateur de rentrer une donnée dans un programme. L'entrée faite par l'utilisateur sera toujours de type **str**, si on souhaite que ce soit d'un autre type, il faudra obliger le programme à lire l'entrée comme étant du type voulu.

```
n=input('Entrer un nombre')
print(n, type(n))
```

```
n=int(input('Entrer un nombre'))
print(n,type(n))
```

Comparer les affichages de ces deux programmes. Les fonctions **int(..)**, **float(..)** et **str(..)** permettent de transformer le type.

## *Les opérations de bases...*

Chaque type de donnée possède son jeu d'opérations.

### les opérations arithmétiques en Python

- l'addition : +
- la soustraction : –
- la multiplication : *
- la division : /
- le quotient dans une division euclidienne : //
- le reste dans une division euclidienne: %
- la puissance : **

Pour avoir accès aux fonctions mathématiques comme la racine carrée(`sqrt(5)`), le sinus(`sin(45)`), le cosinus(`cos(45)`), l'exponentielle(`exp(0)`) etc.
Il faut au préalable importer la bibliothèque math avec l'instruction: `from math import*`

**Exercice 1 :**

On considère les variables $a = 51$, $b = 7$, $c = 21.4$, faites afficher les résultats des opérations: $a+b$, $b-c$, $a/b$, $a \times b$, $a//b$, $c\%b$, $b^4$ et $\sqrt{\frac{a+c}{b^2}}$.

### les opérations sur les chaînes de caractères

**La concaténation :** est l'addition de deux chaînes de caractères.
Tester le programme suivant :

```
ch1='Ceci est une'
ch2='chaîne de caractères'
print(ch1+ch2)
```

**Question 2:**

Que faut-il faire pour avoir l'affichage : `Ceci est une chaîne de caractères`

**Question 3:**

Quel affichage produit le code : `print('Hello'*3)`?

*Remarque : Il n'y a pas lieu de faire des divisions et autres opérations arithmétiques sur les chaînes de caractères...*

Une chaîne de caractères est **itérable**, cela signifie que l'on peut accéder à chacun des caractères (ou groupe de caractères) qui la compose.

**La longueur** d'une chaîne de caractères est le nombre de caractères qui la compose, on y accède avec le code : `len(maphrase)`

Tester le code suivant

```
maphrase='Ceci est une chaîne de caractères'
print(len(maphrase))
```

Quel est le résultat affiché ?

On accède à chaque caractère par son **indice**. Considérons une chaîne de caractères de longueur $n$, le premier caractère est à l'indice 0 et le dernier à l'indice $n-1$.

Quel est l'indice du caractère 'h' dans la variable maphrase ?

On peut faire afficher un caractère en particulier avec le code `print(maphrase[indice])`.
Par exemple `print(maphrase[3])` affiche le caractère 'i'.

Qu'affiche le code suivant: `print(maphrase[19])` ?

On peut faire afficher tous les caractères individuellement avec une **boucle**.
La syntaxe est la suivante:

```
# déclaration de la variable maphrase
maphrase='Ceci est une chaîne de caractères'
# pour chaque(for) caractère(car) dans(in) maphrase
for car in maphrase:
    print(car)
```

*Remarque : Il faut **indenter**(une tabulation) les instructions concernées par la boucle.*
*Le texte écrit après le symbole # est du commentaire, c'est très pratique pour expliquer ce que fait du code...*
Par défaut chaque appel de la fonction `print()` crée un affichage après un passage à la ligne.
Tester le programme précédent avec l'instruction `print(car, end='-')`.

Quel est l'affichage obtenu ?

On peut également utiliser les indices pour faire une **boucle** sur une chaîne de caractères:

```
# déclaration de la variable maphrase
maphrase='Ceci est une chaîne de caractères'
# pour chaque(for) indice(i) dans l'intervalle[0,len(maphrase)[  (range(len(maphrase))
for i in range(len(maphrase)):
    print(maphrase[i])
```

**Afficher une sous-chaîne** se fait en utilisant les instructions :

- `print(maphrase[p:q])` pour la sous chaîne de l'indice $p$ à l'indice $q-1$
- `print(maphrase[p:])` pour la sous-chaîne depuis l'indice $p$ jusqu'à la fin.
- `print(maphrase[:q])` pour la sous-chaîne depuis l'indice 0 jusqu'à l'indice $q-1$.

## Exercice 2 :

Écrire l'instruction qui permet d'afficher le mot 'chaîne' de la variable maphrase.

Une chaîne de caractères est **immuable(non mutable)**, cela signifie que l'on ne peut pas changer un caractère une fois qu'elle est déclarée.
Le code suivant provoquera une erreur

```
maphrase='Ceci est une chaîne de caractères'
maphrase[2]='C'
```

---

*Les instructions conditionnelles...*

---

Nous aurons besoins de **comparaisons** pour réaliser des structures conditionnelles.

**les opérations de comparaisons en Python**

- l'égalité : ==
- la non égalité : =!
- plus grand ou égal : >=
- plus petit ou égal : <=
- strictement plus grand: >
- strictement plus petit: <

Si par exemple $x = 3$ et $y = 7$ , l'instruction `print(x == y)` affichera `False` tandis que `print(x < y)` affichera `True`
Qu'affiche l'instruction `3*7%3 == 0` ?

En algorithmique la structure conditionnelle se représente comme ci-dessous:

**Si** test1 est vrai **alors**
  instruction1
  instruction2
  ...
**Si** test2 est vrai **alors**
  instruction3
  ...
**Sinon**
  instruction4
  ...

On peut mettre autant de **Si** ou **Sinon Si** que l'on souhaite et les **Sinon** ne sont pas obligatoires
On peut aussi les imbriquer..

**Si** test1 est vrai **alors**
  instruction1
  **Si** test2 est vrai
    **alors**
    ...
  ...

**Si** test1 est vrai **alors**
  instruction1
  instruction2
  ...
**Sinon si** test2 est vrai **:**
  instruction3
  ...
**Sinon**
  instruction3
  ...

On utilise également des opérations de logique : En Python le **et**→ (**and**), le **ou**→ (**or**)

**Si** test1 est vrai et test 2 est vrai **alors**
  instruction1

**Si** test1 est vrai ou test2 est vrai **alors**
  instruction1

Chaque langage de programmation a défini une stratégie pour délimiter les instructions à réaliser si une condition est réalisée.

En Java et JavaScript on utilise des accolades '{ ...}'.

En Python on utilise les ':' pour signifier le début et un retour à la ligne avec un décalage 'spatial' que l'on nomme **indentation** ( une tabulation vers la droite ).

```
if test1 == True:
    instruction1
    if test2 == False:
        instruction2
    else:
        instruction3
else:
    instruction4
instruction5
```

**? QUESTION 4:**

Pour chacune des instructions, préciser quels tests doivent être Vrais ou Faux.

| instruction | test1 | test2 |
|---|---|---|
| instruction1 | | |
| instruction2 | | |
| instruction3 | | |
| instruction4 | | |
| instruction5 | | |

---

## *Les boucles...*

**Les structures itératives**

En algorithmique les structures itératives (boucles) se représentent comme ci-dessous:

Boucle **for** :

**Pour** i allant de 0 à n
**faire**
  instruction1
  ...

⟵ Les boucles **for** ont un itérateur intégré, on dit qu'elles sont bornées.

⟶ Les boucles **while** dépendent d'un test et n'ont donc pas d'itérateurs (elles sont dites non bornées), **il faut s'assurer qu'elles se terminent.**

Boucle **while**:

**Tant que** test est vrai
**faire**
  instruction1
  ...

En Python on écrit une boucle comme suit :

```
# i varie de 0 à 9
for i in range(10):
    instructions
# i varie de 1 à 9
for i in range(1,10):
    instructions
```

```
# i varie de 0 à 9 par pas de 2
for i in range(0,10,2):
    instructions
# i varie de 10 à 1 par pas de -1
for i in range(10,0,-1):
    instructions
```

**Là aussi il faut indenter les instructions...**

L'itérateur d'une boucle **for** est toujours un entier (positif ou négatif), par défaut il vaut 1 sinon on spécifie le pas.

Pour la boucle **while** :

```
i=0
pas=1
while i<=10 :
    instructions
    i=i+pas
```

```
#pgm1
x = 0
for i in range(5):
    x = x + i
#pgm2
x = 20
for i in range(1,6):
    x = x - i
#pgm3
x = 1
i=0
while i < 1:
    x = x + 1
    i = i + 0.2
```

**Question 5:**

Pour chacun des programmes ci-contre, donner la valeurs de x avant et après exécution.

| programme | x - Avant | x - Après |
|---|---|---|
| pgm1 | | |
| pgm2 | | |
| pgm3 | | |

---

*Des exemples...*

---

**Exemple 1 :** Étant donné une chaîne de caractères, combien de 'e' contient-elle ?
L'algorithme est assez simple : On crée un compteur initialisé à 0 et pour chaque caractère de la chaîne si c'est un 'e', on incrémente le compteur de 1.

```
maphrase ← chaîne
compteur ← 0
Pour chaque car dans maphrase:
faire
    Si car = 'e' alors
        compteur=compteur+1
afficher compteur
```

```
# en Python
maphrase = "Ceci est une chaîne de
caractères"
compteur=0
for car in maphrase:
    if car == 'e':
        compteur=compteur+1
print(compteur)
```

**Exercice 3 :**

Modifier le programme précédent pour qu'il tienne compte également des 'e' accentués.

### Exercice 4 :

Modifier le programme précédent pour qu'il affiche le nombre de 'c' ou 'C' de la chaîne.

### Exercice 5 :

Modifier le programme précédent pour qu'il affiche le nombre de voyelles accentuées ou pas de la chaîne.

### Exercice 6 :

Écrire un programme qui demande à l'utilisateur de rentrer un texte, puis qui affiche le pourcentage de 'e' contenu dans le texte.

**Exemple 2 :** Le jeu du nombre mystère
L'ordinateur choisit aléatoirement un entier compris entre 1 et 100, et l'utilisateur doit le trouver.
Nous aurons besoin de la bibliothèque `random` pour utiliser un choix aléatoire:

```
from random import * #import de la bibliothèque
# choix aléatoire d'un entier entre 1 et 100 (compris)
nombre_a_trouver = randint(1,100)
```

L'algorithme là aussi est assez simple...On initialise une variable `nombre_a_tester` à 0, puis tant qu'on a pas trouvé le nombre, on invite l'utilisateur à entrer un nombre en lui indiquant si le nombre cherché est plus grand ou plus petit.
Voilà le programme:

```
from random import *
nombre_a_trouver = randint(1,100)
nombre_a_tester = 0
print(nombre_a_trouver)
while nombre_a_tester != nombre_a_trouver:
    nombre_a_tester = int(input("veuillez entrer un entier entre 1 et 100"))
    if nombre_a_tester > nombre_a_trouver :
        print("C'est moins")
    if nombre_a_tester < nombre_a_trouver :
        print("C'est plus")
print("bravo vous avez trouvé :",nombre_a_tester)
```

*Remarque: le programme se termine si l'utilisateur trouve le nombre...*

## Exercice 7 :

Modifier le programme pour qu'il affiche en plus le nombre d'essais qu'a effectué l'utilisateur.

## Exercice 8 :

Écrire un programme qui simule le lancé d'un dé et qui affiche le nombre de fois où le 6 est apparu sur 1000 lancés.

## Exercice 9 :

Voici un programme, que fait-il ?

```
from random import *
lettres = "abcdefghijklmnopqrstuvwxyz"
lettre_a_trouver = lettres[randint(0,len(lettres)-1)]
lettre_a_tester = "0"
compteur=0
while lettre_a_tester != lettre_a_trouver:
    lettre_a_tester = input("veuillez entrer une lettre en minuscule
    (non accentuée)"
    compteur=compteur+1
    if lettre_a_tester > lettre_a_trouver :
        print("C'est moins")
    if lettre_a_tester < lettre_a_trouver :
        print("C'est plus")
print("bravo vous avez trouvé ! ",lettre_a_tester, " en " , compteur ,
" essais")
```

## Exercice 10 :

Écrire un programme qui affiche parmi les 1000 premiers entiers naturels ceux qui sont à la fois divisibles par 3 et 37. *(On fera une boucle :* `for i in range(1,1001):` *)*
*Rappel : a est divisible par b si :* `a%b == 0`

## Les fonctions

Une fonction permet de délocaliser des instructions afin de rendre le programme plus lisible et éventuellement de pouvoir l'utiliser dans d'autres programmes.

Les fonctions en Python se déclarent avec le mot clé **def** suivie du nom de la fonction (que des caractères alpha numériques) auquel ont 'colle' deux parenthèses '( )' qui peuvent contenir des paramètres(séparés par une virgule) ou pas.

Ce qui donne : `def mafonction(paramètre1,paramètre2,...):`

Les instructions de la fonctions commencent après les deux points et sont aussi indentées après un retour à la ligne.

Le mot clé **return** permet de retourner un résultat lors de l'appel de la fonction.

```
# fonction avec 1 paramètre
# et un retour
def f(x):
    return x**2+1
# on appelle la fonction
#par son nom
# ici 5 fois...
for i in range(5):
    print(f(i))
```

```
# fonction sans paramètres
# et sans retour
def mafonction():
    for i in range(5):
        print(i*'*')
# On appelle la fonction
mafonction()
```

```
# fonction avec 2 paramètres
# et un retour
def max(a,b):
    if a >= b:
        return a
    else:
        return b

print(max(25,13))
```

**? QUESTION 6:**

Que produit l'instruction : `print(max(2,7),max(max(54,10),58))`

## La spécification des fonctions

Pour bien faire, toute fonction se doit d'être spécifiée.
Il s'agit d'écrire en début de fonction le type des paramètres d'entrées et les conditions sur ces paramètres, ce que fait la fonction et enfin le type de retour. Le tout entre des triples guillemets. Par exemple pour notre fonction `max(a,b)` cela pourrait donner :

```
def max(a,b):
    """ a un entier,b un entier
    compare les valeurs de a et b
    renvoie le plus grand """
    if a >= b:
        return a
    else:
        return b
```

L'instruction `print(help(max))` affichera dans la console les spécifications de la fonction:

```
Help on function max in module __main__:

max(a, b)
    a un entier,b un entier
    compare les valeurs de a et b
    renvoie le plus grand
```

## Exercice 11 :

Faites afficher l'aide de la fonction **print**.
Que signifie le fait que la valeur par défaut du paramètre end est $\backslash n$?

Tester le paramètre sep et décrire son utilisation.

## Exercice 12 :

Écrire la spécification de ce programme:

```
def table_mult(a):
    for i in range(12):
        print(a, " x ", i, " = ", a*i)
```

## Exercice 13 :

Écrire la spécification de ce programme:

```
def est_premier(n):
    if n==2:
        return True
    if n%2 == 0 or n == 1:
        return False
    for i in range(3,n,2):
        if n%i == 0:
            return False
    return True
```

## Exercice 14 :

Que fait ce programme et écrire sa spécification :

```
def palindrome(mot):
    l=len(mot)
    for i in range(l):
        if mot[i] != mot[l-i-1]:
            return False
        else:
            return True
```

## Exercice 15 :

Écrire une fonction `inverse` qui prend en paramètre un mot et qui renvoie le mot écrit à l'envers.
Exemple : bonjour → roujonb

## Exercice 16 :

La distance de Hamming entre deux mots de même longueur est le nombre d'endroits où les lettres sont différentes.
Par exemple : japon - savon
La première lettre de japon est différente de la première lettre de savon, les troisièmes aussi sont différentes.
La distance de Hamming entre japon et savon vaut donc 2.
Écrire une fonction qui calcule la distance de Hamming entre deux mots entrés par l'utilisateur

## Exercice 17 :

Le latin-cochon :
On transforme un mot commençant par une consonne selon la recette suivante :

- on déplace la première lettre à la fin du mot
- on rajoute le suffixe UM

Par exemple : VITRE devient ITREVUM

Écrire une fonction qui transforme un mot en latin-cochon

## Exercice 18 :

Écrire une fonction qui affiche les diviseurs d'un entier.
Exemple : 126 → 1-2-3-6-7-9-14-18-21-42-63-126

## Exercice 19 :

Un nombre de Armstrong est un entier positif égal à la somme des cubes de ses chiffres.

Exemple : 153 = 1 +125 +27

Écrire une fonction qui prend en paramètre un entier n qui retourne les nombres de Armstrong inférieur à n.

## Exercice 20 :

La suite de Fibonacci est définie par $f_0 = 0$, $f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$

Écrire une fonction qui prend en paramètres un entier n et qui affiche les n premiers termes de la suite de Fibonacci.