

# Seconde

## Cours Python

*Préambule important :*

Apprendre l'informatique est un exercice délicat. En seconde, cela fait presque quinze ans que vous faites de maths (si si, on commence dès son plus jeune age à compter les frites dans son assiette) mais vous êtes totalement novice en Python.

Le moyen le plus efficace d'apprendre l'informatique est d'essayer, par soi même de comprendre ce qui se passe. Il existe de nombreuses ressources pour y parvenir. Vous trouverez en bas de cette page de quoi débiter par vous même dans un cadre amusant.

## Introduction à Python pour les élèves de seconde

Qu'est-ce-que Python ?

Python est un langage de programmation crée en 1991. Un *langage de programmation* est un programme informatique dans lequel on peut décrire ce qu'on veut faire faire à l'ordinateur à l'aide d'une syntaxe particulière. Python lit ces programmes et les *interprète*. Il exécute une par une les commandes qu'il reçoit.

Python est un *vrai* langage de programmation : il est utilisé par des millions de professionnels pour réaliser à peu près n'importe quoi (l'application entière utilisée par les employés d'une grande banque, des calculs scientifiques élaborés, un jeu vidéo etc.).

Nous allons simplement nous initier à Python.

## Comment utiliser Python ?

Votre calculatrice Numworks contient une version de Python particulière appelée *microPython* dans laquelle on peut :

- écrire un programme,
- exécuter un programme,
- entrer des commandes une par une comme dans une calculatrice.

```
deg PYTHON
1 def syracuse(n=14):
2     maxi=0
3     compte=1
4     while n>1:
5         print(n, end=" ")
6         maxi=max(n,maxi)
7         compte+ 1
8         if n%2==0:
9             n/ 2
10        else:
11            n=3*n+1
12        print(1)
```

## Déroulé des instructions

Comme dans tous les langages de programmation, les commandes sont écrites ligne par ligne et sont exécutées dans le même ordre.

Certaines commandes appelées *structures de contrôle* permettent de sauter d'une ligne à une autre.

## Fonctions

Les fonctions se comportent "grosso modo" comme les fonctions habituelles en maths.

Elles sont constituées de deux parties :

1. La définition de la fonction : c'est le code qui décrit ce qu'elle fait.
2. L'appel de la fonction : c'est le code qui demande une exécution de cette fonction.

Par exemple

```
def addition(a, b):
    return a + b

cinq = addition(2, 3)
```

Les deux premières lignes de ce programme sont la définition.

Le *nom* de cette fonction est **addition**

Elle a deux *paramètres* : **a** et **b**

Le mot clé **return**, pour *renvoyer*, nous indique ce que la fonction va produire comme valeur.

Ici, la fonction **addition** renvoie la somme de **a** et de **b**.

La dernière ligne de ce programme est un *appel* à la fonction **addition**

La *variable* **cinq** reçoit la valeur de **addition(2, 3)**, c'est-à-dire 2+3... qui valent justement 5.

## Interpréteur

Lorsqu'on exécute un programme python sur la calculatrice, il est *interprété* ligne par ligne.

Une fois ce programme terminé, la calculatrice *rend la main* et on peut taper des commandes à nouveau.

On sait qu'on *a la main* lorsque la dernière ligne est précédée de trois chevrons >>>

Ainsi dans l'exemple suivant, j'ai la main.

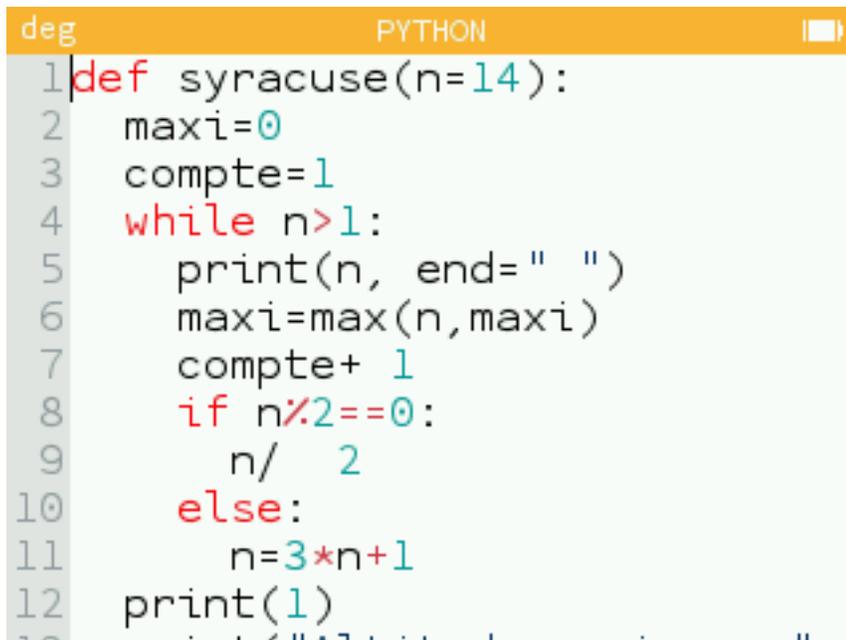
```
>>> a = 2
>>> a = a + 4
>>> a
6
>>>
```

## la coloration syntaxique

On utilise généralement des couleurs pour symboliser la nature des éléments. Les mots clés, le texte, les nombres prennent généralement des couleurs différentes pour faciliter la lecture.

```
def addition(a, b):
    return a + b

cinq = addition(2, 3)
ami = "Marcel"
```



```
1 def syracuse(n=14):
2     maxi=0
3     compte=1
4     while n>1:
5         print(n, end=" ")
6         maxi=max(n,maxi)
7         compte+ 1
8         if n%2==0:
9             n/ 2
10        else:
11            n=3*n+1
12        print(1)
```

Dans l'éditeur de code, certains mots sont en rouge, d'autres en noir.

- Les mots en rouge sont des mots clés qui sont réservés.
- Les mots en noir sont les noms des autres objets, les nombres, les autres symboles.

## Lire un extrait de code :

Considérons l'extrait de code suivant :

```
a = 8
b = 14
c = a + b
a = a + c
```

Beaucoup de choses à dire !

Allons-y ligne par ligne :

1. `a = 8` : on affecte à la variable `a` la valeur 8.

2. `b = 14` : on affecte à la variable `b` la valeur 14.
3. `c = a + b` : on affecte à la variable `c` le résultat de `a + b` donc 22.
4. `a = a + c` : **attention** : on affecte à la variable `a` une nouvelle valeur, celle de `a + c` donc  $8 + 22 = 30$

Remarquons bien ce qu'on a fait à la dernière ligne `a` valait 8 et maintenant `a` vaut 30.

Ce principe, celui de l'affectation, n'existe qu'en informatique. Il est absent des mathématiques usuelles où, une fois un objet nommé (*Soit  $K$ , le milieu de  $[AB]$* ), il ne change plus.

En informatique, un nom, comme `a` peut désigner des valeurs différentes au cours de l'exécution.

Donc, en python : `a = 1` est une affectation : on donne à `a` la valeur 1.

## Structure de contrôle

Nous allons étudier quatre structures de contrôles :

1. les fonctions avec `def`
2. les conditions avec `if`
3. les boucles bornées avec `for`
4. les boucles non bornées avec `while`

De manière générale, une structure de contrôle permet à un programme de *sauter* d'une ligne à l'autre. Plutôt que d'être exécuté de haut en bas (1, 2, 3, 4) le programme peut passer d'une ligne à l'autre comme on le souhaite.

## Syntaxe commune : l'indentation

Toutes les structures de contrôle ont trois points commun :

1. Elles commencent par un *mot clé* (`def`, `if`, `for`, `while`)
2. La première ligne se termine par le symbole : ("deux points")
3. Leur contenu est décalé de quelques espaces vers la droite (*c'est l'indentation*)

Lorsqu'on revient "plus à gauche", alors on quitte cette structure de contrôle.

Exemple :

```
def ma_fonction(parametre):
    return 3

a = 5
```

1. On a bien à faire à une *structure de contrôle*, une *fonction* (mot clé `def`)
2. La première ligne se termine par :
3. Le *corps* de la fonction est la ligne `return 3` (elle renvoie toujours 3)
4. La ligne `a = 5` n'est pas indentée, elle est en dehors de la fonction.

## Fonctions

On a déjà rencontré plusieurs fonctions, considérons un autre exemple.

Cette fonction calcule les coordonnées du milieu d'un segment :

```
def milieu(xa, ya, xb, yb):
    return (xa + xb) / 2, (ya + yb) / 2
```

- Cette fonction s'appelle `milieu`
- Elle a quatre paramètres `xa`, `ya`, `xb`, `yb`
- Elle renvoie deux nombres (notez la virgule à la dernière ligne)

Pour déterminer le milieu du segment  $[AB]$  avec  $A(3, 4)$  et  $B(7, 2)$ , on l'utilise comme ça :

```
>>> milieu(3, 4, 7, 2)
(5.0, 3.0)
```

Les coordonnées du milieu de  $[AB]$  sont  $(5, 3)$ .

## Charger un module

Changeons un peu de sujet avant d'attaquer une autre structure de contrôle.

Lorsque Python est lancé, il ne charge pas tout ce qu'il peut faire. Comme un technicien, il n'a pas toujours besoin d'avoir avec lui chacun de ses outils.

Certaines fonctions sont définies séparément, en particulier les fonctions mathématiques.

C'est pour ça que vous rencontrerez souvent la ligne :

```
from math import *
```

Qui se lit :

- depuis le module `math`
- importe tout (l'étoile `*` signifie souvent "tout" en informatique)

Donc on a chargé beaucoup de fonctions mathématiques :

- `cos`, `sin`, `pi` etc.

Mais aussi `sqrt` pour (*square root*, la racine carrée).

Et donc, pour calculer la racine carré de 7, on peut faire :

```
>>> from math import *
>>> sqrt(7)
2.64575
```

Lorsqu'on exécute un de nos scripts sur la numworks, Python charge le module en question. Cela explique la première ligne qu'on rencontre dans l'interpréteur.

## Lire un message d'erreur

Lorsque Python rencontre un problème, il s'arrête et affiche une erreur.

### Il est fondamental d'essayer de lire et de comprendre les erreurs

Vous en rencontrerez beaucoup, elles sont de nombreux types.

Par exemple :

```
>>> 1 / 0
Last command
ZeroDivisionError: divide by zero
```

Cette opération n'est pas définie, Python affiche une erreur et rend la main.

Autre exemple :

```
>>> (2 + 3
Last command
SyntaxError: invalid syntax
```

Python rencontre un erreur de syntaxe, on a oublié de fermer une parenthèse.

Lorsqu'il affiche une erreur, Python indique toujours d'où elle vient, ici de `Last command`, la dernière commande l'interpréteur.

## Les conditions avec if

Imaginons la situation suivante :

Après une longue partie de jeux-vidéo, on calcule son score. Si le score est supérieur à 10, on affiche “bravo” Sinon, on affiche “t’es nul” (pas facile, la vie.)

En Python, cela se traduit ainsi :

```
score = 30

if score > 10:
    print("bravo")
else:
    print("t'es nul")

print("game over")
```

Lorsque ce script est exécuté, on voit apparaître l’affichage :

```
bravo
game over
```

Pourquoi ?

1. On affecte à `score` la valeur 10
2. `if score > 10:` si score dépasse 10... cette *condition* est vraie !
3. `print("bravo")` : afficher “bravo”... cette ligne sera exécutée
4. `else:` (*sinon*) cette ligne est zappée
5. `print("t'es nul")` : afficher t’es nul... cette ligne est zappée aussi
6. `print("game over")` : cette ligne est EN DEHORS de la structure `if...` donc elle est toujours exécutée.

Reprenons le même script mais avec `score = 5`.

Qu’obtient-on ?

```
t'es nul
game over
```

Cette fois, la condition `score > 10` est fausse et le bloc `if...` est zappé jusqu’à `else:...` qui est exécuté.

### Exo

```
age = 12
if age >= 18:
    majeur = True
else:
    majeur = False
```

(`True` signifie *vrai* et `False` signifie *faux*)

1. Quelle est la valeur de `majeur` après l’exécution du script précédent ?
2. Aux états-unis, la majorité complète est atteinte à 21 ans. Comment modifier le script pour l’adapter à cette situation ?

## Les boucles bornées avec for

## Les boucles non bornées avec hile

Les listes : [1, 2, 3, 4]