# NOTIONS PYTHON FONDAMENTALES pour le début de la TERMINALE NSI

## I – Structure d'un programme

Partie Importation
Partie Déclaration des CONSTANTES
Partie Déclaration des fonctions
Partie Programme principal

## II – Variables

**Déclaration** : `>>> a = 50 + 10`

La lecture se fait de droite à gauche :
1) Evaluation de l'expression 50 + 10
2) Récupération de 60
3) Affectation de 60 à la variable a

**Lecture** : il suffit de taper a dans une expression et l'interpréteur va l'évaluer à 60.

```
>>> a
60
>>> a*2
120
```

**Modification** : `>>> a = 70`

On réalise une nouvelle affectation. Attention l'identifiant ou l'adresse de a n'est pas la même après la nouvelle affectation : il s'agit d'une variable qui porte le même nom mais son identifiant mémoire n'est pas le même que la variable de a de a = 50 + 10. On obtient l'identifiant d'une variable en utilisant la fonction native **`id().`**

**Incrémentation** : `>>> a = a + 1`

Consiste à modifier la valeur d'une variable en utilisant sa valeur actuelle.

## III – Types

### INTEGER :

Le type des entiers. Type **`int`** en Python.
La fonction **`type()`** permet de connaître le type d'une donnée stockée.

```
>>> a = 50
>>> type(a)
<class 'int'>
```

### FLOAT :

Le type des nombres à virgules. Type **`float`** en Python.

```
>>> b = 50.0
>>> type(b)
<class 'float'>
```

Le problème des floats : certaines valeurs (comme 0,1) ne sont stockées qu'approximativement. **Aucun informaticien sérieux ne réalise donc de tests d'égalité sur des valeurs flottantes**.

```
>>> c = 0.1
>>> 0.3 – 3 * c
-5.551115123125783e-17
```

### BOOLÉEN :

Le type des valeurs ne pouvant valoir que **`True`** ou **`False`**. Type **`bool`** en Python.
On peut en obtenir facilement en utilisant les opérateurs de comparaison :
Opérateur égalité : **`==`**
Opérateur différence :**`!=`**
Opéreur de comparaison : **`>  <  >=  <=`**

**`False`** : `[], (), {}, '', 0, 0.0, None`

```
>>> a = 20
>>> b = 10
>>> c = a == 2*b
>>> c
True
```

```
>>> d = 0.3
>>> e = 0.1
>>> f = d == 2*e
>>> f
False
```

**STRING :**

Le type des chaînes de caractères. Type **str** en Python.

On les définit avec des guillemets simples ou doubles mais il faut choisir.

```
>>> g = "Hello "
>>> type(g)
<class 'str'>

>>> h = 'World !'
>>> type(h)
<class 'str'>
```

On peut réaliser des **concaténations avec** + et des **répétitions avec *** :

```
>>> i = g + h
>>> i
'Hello World !'

>>> j = "Pom ! " * 4
>>> j
'Pom ! Pom ! Pom ! Pom ! '
```

On notera que dans certains langages, on fait la distinction entre :
* le caractère unique (guillemet simple) :

```
k = 'H'
```

* le string (guillemet double) :

```
m = "Hello World !"
```

**TABLEAU STATIQUE**

Déclaration ; avec des **crochets**.

Contenu : toutes les cases ont le même type.

Nom du type en Python : **list**

Première case : indice 0, attention.

Muable en Python : on peut modifier le contenu des cases après création.

Lecture : avec des **crochets** et l'**indice**.

```
>>> t = [10, 20, 30]
>>> t[1]
20
>>> type(t)
<class 'list'>
>>> t[1] = 15
>>> t
[10, 15, 30]
```

**Attention :** le type list permet d'avoir des cases contenant des types différents. Ce n'est pas une raison pour utiliser la fonctionnalité en NSI.

**Parcours d'un tableau** : 2 façons.

**Par indices** : complexe mais permet les modifications.

```
t = [10, 20, 30]
for i in range(len(t)):
    t[i] = t[i] * 2
    print(t[i])
```

```
20
40
60
```

Ensuite **t référence toujours [20, 40, 60]**

**Par valeurs** : plus clair mais sans modification possible.

```
t = [10, 20, 30]
for v in t:
    v = v * 2
    print(v)
```

```
20
40
60
```

Mais ici **t référence toujours [10, 20, 30]**

**TABLEAU DYNAMIQUE**

Ajout à la fin ; méthode **append()**

```
t = [10, 20, 30]
t.append(5)
```

**t référence alors [10, 20, 30, 5]**

Suppression de la fin ; méthode **pop()**

```
t = [10, 20, 30]
a = t.pop()
```

**t référence alors [10, 20]**

**a référence 30**

**N-UPLET (singleton 1-uplet, couple 2-uplet, triplet ..)**

Déclaration ; avec des **parenthèses**.

Contenu : types différents possibles.

Nom du type en Python : **tuple**

Première case : indice 0, attention.

Immuable en Python : on ne peut pas modifier le contenu des cases après création.

Lecture : avec des **crochets** et l'**indice**. Avec les boucles, s'utilise comme les tableaux sinon.

```
>>> t = (10, 20, 30)
>>> t[1]
20
>>> type(t)
<class 'tuple'>

>>> t = (10,)   ← virgule importante
```

**DICTIONNAIRE**

Déclaration ; avec des **accolades**

Contenu : types différents possibles.

Nom du type en Python : **dict**

Première case : n'existe pas ! Pas d'indice.

Muable en Python : on peut modifier le contenu des cases après création.

Lecture : avec des **crochets** et l**a clé**.

```
>>> d = {'Alice':10, 'Bob':15}
>>> d['Alice']
10
>>> type(d)
<class 'dict'>

>>> d['Alice'] = 18
>>> d
{'Alice':18, 'Bob':15}
```

En boucle : **keys(), values(), items()**

```
d = {'Alice':10, 'Bob':15}
for c in d.keys():
      print(c)

Affiche :
Alice
Bob
```

```
d = {'Alice':10, 'Bob':15}
for v in d.values():
      print(v)

Affiche :
10
15
```

```
d = {'Alice':10, 'Bob':15}
for tup in d.items():
      print(tup)

Affiche :
('Alice', 10)
('Bob', 15)
```

## IV – Fonctions

Une fonction se déclare avec le mot-clé **def**. Lors de la déclaration, on place les **paramètres** attendus. Elle renvoie sa réponse avec le mot-clé **return**.

Les variables définies à l'intérieur d'une fonction sont dites **locales** : elles n'existent que le temps de l'appel de la fonction. Elles disparaissent dès que le return a été exécuté.
Les **variables globales** sont les variables définies en dehors des fonctions : elles durent durant toutes l'éxécution du programme lui-même.

Lors de l'appel de la fonction, on lui envoie des **arguments** qui servent alors à définir le contenu des paramètres.

Une fonction correctement réalisée doit contenir sa documentation comportant à minima : une description de son utilité, des types et des **préconditions** et **postconditions**.

```
# Déclaration des fonctions

def exemple(a, b):
  '''Renvoie a*b si deux positifs, a si b est négatif
  :: param a(int) :: un entier
  :: param b(int) :: un entier
  :: return (int) :: un entier, a*b ou a en fonction de b
  '''
  r = a*b
  if b < 0:
    r = a
  return r
```

```
def exemple2(a:int, b:int) -> int:
  '''Renvoie a*b si deux positifs, a si b est négatif'''
  r = a*b
  if b < 0:
    r = a
  return r

# Programme et appels des fonctions

x = exemple(10, 20)
y = exemple2(10, -20)
```

10, 20 et -20 sont les arguments qui seront stockés temporairement dans les paramètres (variables locales temporaires) a et b.
A la fin du programme, les variables globales x et y contiendront respectivement 200 et 10.

**RAPPEL** :

| | | |
|---|---|---|
| Préconditions respectées (travail de l'utilisateur) | IMPLIQUE | Postcondition respectées (travail du concepteur) |
| Préconditions fausses | IMPLIQUE | Rien sur la postcondition... |

## V – Instructions conditionnelles

### 5.1 Principe :

Activation d'un seul bloc possible parmi les blocs **if**, **elif** et **else**. Seul le bloc **if** est indispensable, les autres sont optionnels. Lorsqu'une des expressions booléennes est **True**, l'interpréteur Python active le **premier bloc qui réagit positivement** puis sort de l'instruction conditionnelle.

### 5.2 Structure :

```
if expression1:
   les trucs à faire
elif expression2 :
   les trucs à faire
elif expression3:
    les trucs à faire
else:
    les trucs à faire
```

### 5.3 Expression non booléenne ?

Dans ce cas, Python va interpréter l'expression comme si l'utilisateur avait placé la fonction native **bool()** autour de l'expression.

```
a = 0
if a:
  print("a est VRAI !")
else:
  print(("a est FAUX !")  ← Activé
```

**False : [], (), {}, '', 0, 0.0, None**

### 5.4 Exemple de if mal conçu

```
n = 20
if n > 10:
   app = "Correct"          ← Activé
elif n > 15:
   app = "Bien"
else:
   app = "Pas bien"
```

Résultat : app contient **"Correct"** et pas **"Bien"** comme le concepteur l'aurait certainement voulu.

## VI – Boucle bornée : POUR

### 6.1 Principe :

On la codifie avec le mot-clé **for** en Python. On crée une variable de boucle qui va prendre successivement plusieurs valeurs.

```
c = 0
for a in range(3):
    c = c + a
```

Equivalent à
c = 0
c = 0 + 0 (= 0)
c = 0 + 1 ( = 1)
c = 1 + 2 ( = 3)

**6.2 Description de range** :

→ un seul argument : valeur « presque » finale.
in range(3) → 0 puis 1 puis 2

→ deux arguments :
initiale puis presque finale.
in range(5,8) → 5 puis 6 puis 7

→ trois arguments :
initiale puis presque finale puis pas.
in range(50, 75, 10) → 50 puis 60 puis 70

## VII – Boucle non bornée : TANT QUE

**7.1 Principe** :

Tant que la condition de poursuite est vraie lorsqu'on la teste, l'interpréteur réalise la boucle. Il revient alors au début de la boucle et teste à nouveau la condition de poursuite pour voir s'il continue.

**7.2 Structure** :

```
while condition_poursuite:
    les trucs à faire

while not condition_arret:
    les trucs à faire
```

## VIII – ET et OU : tables de vérité et not (a and b) et not (a or b)