1ère NSI - Python avancé #1 : Variables et effets de bord

1) Références et Valeurs : surprises ⚠️

a) Cas des types simples : tout va bien !

In [1]:
```
a = 5
b = a
print("a=", a, "\tb=", b)
```

```
a= 5    b= 5
```

a et b ont la **même valeur**.

In [2]:
```
b += 4
print("a =", a, "\tb =", b)
```

```
a = 5   b = 9
```

Si on **change la valeur** de b , a reste **inchangé**. Tout va bien.

b) Cas des Listes : ⚠️ Danger ! ⚠️

Réalisons le même code avec des **listes**

In [3]:
```
liste1 = [1, 2, 3]
liste2 = liste1
print("liste1 =", liste1, "\tliste2 =", liste2)
```

```
liste1 = [1, 2, 3]      liste2 = [1, 2, 3]
```

`liste1` et `liste2` ont la **même valeur**. Mais **modifions** `liste2` ???

In [4]:
```
liste2 += [4]
print("liste1 =", liste1, "\tliste2 =", liste2)
```

```
liste1 = [1, 2, 3, 4]   liste2 = [1, 2, 3, 4]
```

⚠️ Si on **modifie** `liste2` , `liste1` est **modifiée** aussi ! ⚠️ Que s'est-il passé ???

Pour comprendre, écrivons une **fonction** qui retourne **l'adresse** (ou **référence**) d'une **valeur en mémoire** :

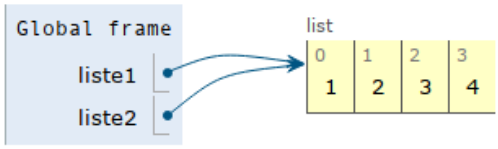
In [5]:
```
def adr(variable):
    return hex(id(variable))
```

Regardons les **adresses** des données dans `liste1` et `liste2` :

In [6]:
```
print("Adresse des données de liste1:",adr(liste1))
print("Adresse des données de liste2:",adr(liste2))
```

```
Adresse des données de liste1: 0x7fb7206355f0
Adresse des données de liste2: 0x7fb7206355f0
```

Les deux variables `liste1` et `liste2` **pointent vers la même adresse en mémoire** : leur contenu est donc toujours **identique**. En effet, si je modifie l'une, je modifie l'autre puisqu'elles **pointent sur le même contenu en mémoire**.



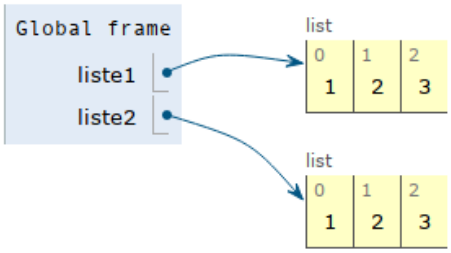
Python **ne recopie pas** le contenu de `liste1` dans `liste2` , il se contente de **faire pointer** la variable `liste2` sur **l'emplacement mémoire** de `liste1` .

Si on souhaite **recopier** le contenu de `liste1` dans `liste2` et que les 2 variables **restent indépendantes** avec des **adresses différentes**, il faut indiquer **explicitement la copie**, par exemple comme ceci :

In [7]:
```
liste1 = [1, 2, 3]
liste2 = liste1.copy()
print("liste1 =", liste1, "\tliste2 =", liste2)
print("Adresse des données de liste1:", adr(liste1))
print("Adresse des données de liste2:", adr(liste2))
```

```
liste1 = [1, 2, 3]      liste2 = [1, 2, 3]
Adresse des données de liste1: 0x7fb720635be0
Adresse des données de liste2: 0x7fb720635af0
```

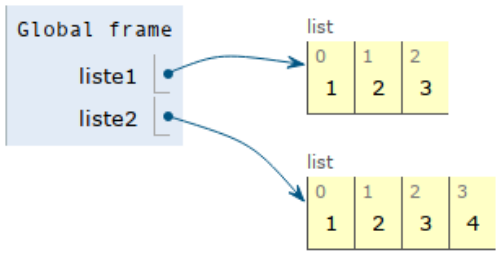
Cette fois on a bien **2 variables qui pointent vers des objets différents** :



In [8]:
```
liste2 += [4]
print("liste1 =", liste1, "\tliste2 =", liste2)
```

```
liste1 = [1, 2, 3]      liste2 = [1, 2, 3, 4]
```

Les deux variables sont bien **distinctes** et pointent vers **2 places en mémoire différentes**. Elles sont donc **indépendantes**. Donc quand on modifie le contenu de `liste2` , `liste1` reste inchangé :



2) Fonctionnement des variables en Python

Avant de passer aux explications suivantes, nous devons comprendre comment **Python** gère les **variables** et les **valeurs** qu'elles *contiennent*.

Les valeurs

En Python chaque **valeur** est en réalité un **objet** en mémoire. Un **objet** est une **structure en mémoire** qui contient des **données** comme : son **type**, sa **valeur**, des **liens sur des fonctions** qui peuvent s'appliquer à l'objet, etc.

Par exemple, la **valeur** `5`, dès qu'on y **accède** ou qu'on **l'évalue**, cela engendre la création d'un **objet**. Vérifions cela en demandant à Python **l'adresse de l'objet** `5` :

```
In [9]: adr(5)
Out[9]: '0x7fb72661d460'
```

Le nombre `5` est donc un **objet en mémoire** avec sa **propre adresse**.

Je peux aussi **interroger** cet objet et lui demander son **type** :

```
In [10]: print(type(5))
         <class 'int'>
```

Mais je peux aussi lui demander la **liste des fonctions** que je peux **appliquer** à cet objet :

```
In [11]: print(dir(5))
         ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__del
         attr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor_
         _', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs
         __', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__i
         nt__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
         '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '
         __rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloo
         rdiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rp
         ow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '
         __setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__true
         div__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'fr
         om_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Essayons une de ces fonctions sur notre **objet** `5`, par exemple `bit_length()` qui retourne le nombre de bits nécessaires pour représenter la valeur en binaire (5 -> 101)

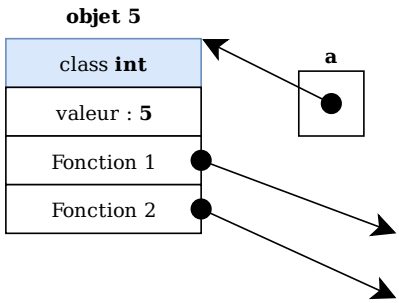
```
In [12]: (5).bit_length()
Out[12]: 3
```

⚠️ **Conclusion** : toute **valeur** en python est un **objet** en mémoire avec sa **propre adresse** ⚠️

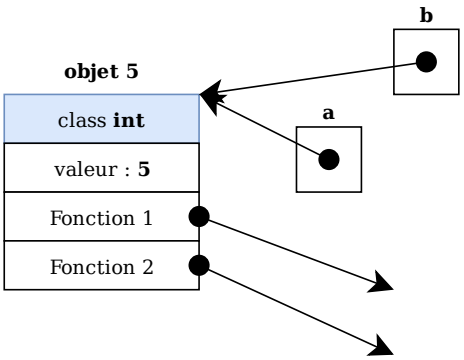
Les variables

Quand on écrit `a = 5` , Python ne range pas la **valeur** `5` dans la **case mémoire** `a` comme le font d'autres langages. C'est une bonne image pour commencer à programmer mais le fonctionnement exact est un peu plus complexe.

En réalité, Python **mémorise dans la variable l'adresse de l'objet qui contient la valeur affectée à la variable**. On dit que `a` **pointe** sur l'objet qu'elle *contient*. Voici une représentation de `a = 5` :



De même si on exécute `b = a` , Python fait **pointer** la variable `b` sur la **même valeur (objet)** que `a` :

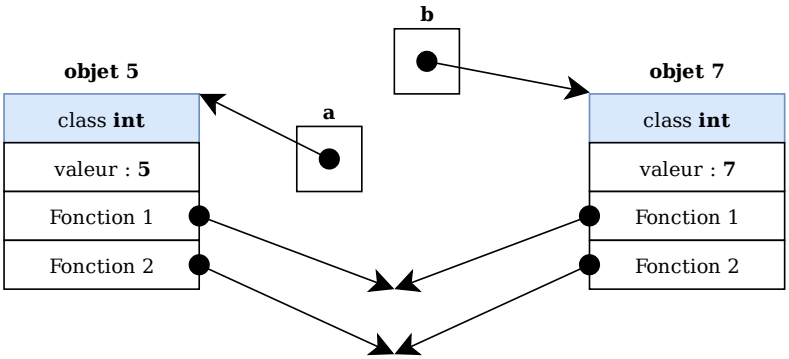


Vérifions que `a` et `b` **pointent** vers le **même objet** en mémoire :

In [13]:
```
print(adr(5))
a = 5
print(adr(a))
b = a
print(adr(b))
```

```
0x7fb72661d460
0x7fb72661d460
0x7fb72661d460
```

Puis quand on fait `b = 7`, la variable `b` va **pointer** sur la **valeur (objet)** `7` :



Ainsi, quand une **variable change de valeur**, Python **change l'adresse** dans la variable qui **pointe vers la nouvelle valeur** (objet).

3) Valeur Immuable (*immutable*) et Muable (*mutable*)

a) Immuables

Les **valeurs** de type `int, float, bool, str` sont **immuables** c'est-à-dire **pas modifiables**. En effet, nous l'avons vu précédemment, **l'objet** `5` est mémorisé à une adresse et **cet objet ne peut pas changer**. Ainsi quand une variable qui contient une **valeur immuable** change de valeur, elle **pointe** vers un **nouvel objet** puisque l'objet ne peut pas être modifié.

Vérifions cela :

In [14]:
```
a = 5
print("Valeur de a:", a, "\t\tAdresse de la variable a: ", adr(a))
a += 4
print("Valeur de a:", a, "\t\tAdresse de la variable a: ", adr(a))
```

```
Valeur de a: 5          Adresse de la variable a:  0x7fb72661d460
Valeur de a: 9          Adresse de la variable a:  0x7fb72661d4e0
```

La variable `a` est bien **modifiée, elle pointe vers l'adresse d'un autre objet ailleurs en mémoire !**

b) Muables

Les **valeurs** de type `list, set, dict` sont **muables** c'est-à-dire qu'elles sont **modifiables sans changer d'adresse en mémoire**.

Vérifions cela aussi :

In [15]:
```
liste1 = [1, 2, 3]
print("Valeur de liste:", liste1, "\t\tAdresse de la variable liste: ", adr(l
iste1))
liste1 += [4]
print("Valeur de liste:", liste1, "\t\tAdresse de la variable liste: ", adr(l
iste1))
```

```
Valeur de liste: [1, 2, 3]              Adresse de la variable liste:  0x7fb7
205ce910
Valeur de liste: [1, 2, 3, 4]           Adresse de la variable liste:  0x7fb7
205ce910
```

L'objet pointé par la variable `liste1` est **muable** il est donc bien **modifiée** directement **en gardant la même adresse en mémoire** !

Les valeurs **muables** engendrent parfois des **bugs car elles peuvent être modifiées par erreur de programmation**. En effet, ici on peut **modifier un objet en mémoire** (car il est muable) en agissant sur le contenu une variable. Mais si une **autre variable pointe sur ce même objet**, son contenu sera **modifié** aussi. Ceci explique le comportement vu précedemment et peut **engendrer des bugs**.

Ce choix s'explique car il serait **trop coûteux** de **recopier la liste ailleurs en mémoire** à chaque changement d'un élément. On permet donc aux objets de se genre d'être **modifiés**, il sont donc **muables**.

Maintenant vous comprenez pourquoi, en introduction à ce cours, `liste1` est modifiée quand on modifie `liste2` : les **2 listes pointent vers le même objet en mémoire qui est modifié par** `liste2 += [4]` :

In [16]:
```
liste1 = [1, 2, 3]
liste2 = liste1
print("liste1 =", liste1, "\tliste2 =", liste2)
print("Adresse du contenu de liste1 :", adr(liste1))
print("Adresse du contenu de liste2 :", adr(liste2))
liste2 += [4]
print("liste1 =", liste1, "\tliste2 =", liste2)
```

```
liste1 = [1, 2, 3]      liste2 = [1, 2, 3]
Adresse du contenu de liste1 : 0x7fb7205ce8c0
Adresse du contenu de liste2 : 0x7fb7205ce8c0
liste1 = [1, 2, 3, 4]   liste2 = [1, 2, 3, 4]
```

3) Paramètres des fonctions et ⚠️ effet de bord ⚠️

Il existe le même problème quand on donne des paramètres (ou arguments) aux fonctions :

- Pour les variables qui contiennent des **valeurs de type** `int, float, bool, str` , il n'y a **pas d'effet de bord** car les **objets** de ces types sont **immuables**. La **valeur** de la variable indiquée en paramètre est passée à la fonction. Si cette valeur **change** dans la fonction, Python utilise un **nouvel objet** à un **autre emplacement mémoire**. Par conséquent, la modification **ne peut pas modifier** le contenu des variables données en paramètres à la fonction.
- Pour les variables qui contiennent des **valeurs de type** `list, set, dict` , il y a des **effets de bord** car les objets de ces types sont **muables**. La **fonction** reçoit **l'objet pointé** par la variable indiquée en argument et **peut** donc modifier cet objet car il est **muable**. Si l'objet est **modifié dans la fonction** et que la variable indiquée en paramètre pointe toujours sur cet objet, le *contenu* de la variable peut être **modifié par la fonction** : c'est un **effet de bord** !

Effet de bord : quand une fonction modifie le contenu d'une variable qui appartient au contexte appelant.

Illustrons cela avec 2 exemples. Nous créons une fonction `test` qui va ajouter `4` à la variable donnée en argument.

a) Valeur immuable en argument : Pas d'effet de bord !

In [17]:
```
def test(param):
    print("Adresse de la valeur de param : ", adr(param))
    param += 4
    print("Adresse de la valeur de param une fois modifiée: ", adr(param))

a = 5
print("Adresse de la valeur de a: ", adr(a))
test(a)
print("Valeur de a après l'exécution de la fonction :",a)
```

```
Adresse de la valeur de a:  0x7fb72661d460
Adresse de la valeur de param :  0x7fb72661d460
Adresse de la valeur de param une fois modifiée:  0x7fb72661d4e0
Valeur de a après l'exécution de la fonction : 5
```

⚠️ La fonction `test` reçoit la **valeur** de la variable `a` **dans sa variable** `param` . Quand la fonction **modifie** la valeur de `param` , cette variable pointe vers un **nouvel objet à une autre adresse** : la variable `a` **ne peut pas être modifiée** ! **Ici, pas d'effet de bord !**

b) Valeur muable en argument : ⚠️ Effet de bord ! ⚠️

In [18]:
```
def test(param):
    print("Adresse du contenu de param : ", adr(param))
    param.append(4)
    print("Adresse du contenu de param une fois modifiée: ", adr(param))

liste = [1,2,3]
print("Adresse du contenu de liste: ", adr(liste))
test(liste)
print("Valeur de liste après l'exécution de la fonction :",liste)
```

```
Adresse du contenu de liste:  0x7fb720635a00
Adresse du contenu de param :  0x7fb720635a00
Adresse du contenu de param une fois modifiée:  0x7fb720635a00
Valeur de liste après l'exécution de la fonction : [1, 2, 3, 4]
```

⚠️ La fonction `test` reçoit dans `param` **l'objet contenu dans la variable** `liste` et peut **modifier** cet objet car il est **muable** ! **Ici il y a effet de bord** car le **contenu** de `liste` est **modifié** par les traitements de la fonction `test` ! ⚠️