1. Objets mutables, objets immuables

Les objets du type bool, int, float, complex, str, tuple sont immuables (ils ne peuvent pas être modifiés) ; les objets du type list, dict, set, array sont mutables (leur contenu peut être modifié).

L'instruction $a = a + 1$ ne modifie pas la variable *a* mais crée une nouvelle variable du même nom (l'ancienne est donc écrasée) et lui donne la valeur :
(*ancienne valeur de a*) + 1

L'instruction *a*[0] = *a*[0] + 1 modifie la variable *a*.

L'instruction *a* = *b* ne crée pas une copie de l'objet *b*, mais juste un autre nom pour le même espace mémoire. On parle d'aliasing. Si *a* et *b* sont mutables, toute modification effectuée sur *a* sera également effectuée sur *b*. La fonction **id**() renvoyant la référence d'une variable permet de vérifier ce principe.

```
In [1]: a=3
In [2]: b=a
In [3]: id(a)
Out[3]: 502119992
In [4]: id(b)
Out[4]: 502119992
```

L'instruction **return** d'une fonction n'effectue pas de copie. Si une fonction renvoie une liste prise en paramètre, toute modification effectuée sur cette liste sera effectuée sur la liste initiale.

Exercices

a.

```
a = [1,2,3]
b = a
a = [0,2,3]
```

```
a = [1,2,3]
b = a
a[0] = 0
```

Que contiennent les variables a et b après exécution des codes ci-dessus ?

b. Expliquer ces différentes instructions portant sur des listes a et b.

a = b	a = b[:]	a[:] = b	a[:] = b[:]

L'instruction a = b[:] effectue-t-elle une vraie copie de b ?

2. Fonctions, procédures, sous-programmes, méthodes

Une fonction est un bout de code qui peut être appelé par le programme principal ou par une autre fonction. Elle peut aussi s'appeler elle-même, on parle alors d'appel récursif.

L'utilisation des fonctions permet d'une part d'éviter de répéter des lignes de code et surtout de découper les gros programmes en plusieurs petits morceaux plus simples qu'il suffit ensuite d'assembler (diviser pour régner).

Certaines fonctions retournent une valeur comme les fonctions mathématiques qui renvoient une image. D'autres ne retournent rien. La présence ou non du mot clé **return** suivi d'une variable permet de les distinguer.

Afin de respecter l'analogie avec les fonctions mathématiques, on appelle parfois procédures ou sous-programmes les « fonctions » qui ne retournent rien. En programmation orientée objet (à la limite de notre programme), les fonctions et procédures encapsulées dans un objet sont appelées méthodes.

Exercice

```
def echange(lst,i,j):
    lst2 = lst[:]
    lst2[i],lst2[j] = lst2[j],lst2[i]
    return lst2
```

```
def echange2(lst,i,j):
    lst[i],lst[j] = lst[j],lst[i]
```

```
liste = list(range(10))
echange2(liste,3,5)
liste_bis = echange(liste,2,6)
liste_ter = echange2(liste,1,8)
```

Expliquer la différence entre les fonctions echange et echange2.
Décrire les variables liste, liste_bis, liste_ter après l'exécution des lignes précédentes.

Remarque : L'appel d'une vraie fonction est une expression, l'appel d'une procédure est une instruction.

3. Variables locales, variables globales

Les variables définies à l'extérieur d'une fonction (dans le corps principal du programme) peuvent être lues par les fonctions et modifiées si elles sont mutables (même si ce n'est pas conseillé), mais pas redéfinies dans des fonctions, sauf mention contraire avec le mot clé **global**, qui permet à la fonction de (re)définir une variable globale alors accessible dans tout le programme.

Les variables définies à l'intérieur d'une fonction sont des variables locales. Elles ne peuvent être utilisées qu'à l'intérieur de la fonction et sont supprimées une fois l'exécution de la fonction terminée (sauf utilisation de **global**).

Exercice : Analyser les codes suivants.

```
def f():
    a = 1
    print(a)

a = 2
f()
print(a)
```

```
def f():
    print(a)

a = 2
f()
print(a)
```

```
def f():
    print(a)
    a = 1
    print(a)

a = 2
f()
print(a)
```

```
def f():
    global a
    print(a)
    a = 1
    print(a)

a = 2
f()
print(a)
```

L'utilisation de variables globales est à éviter, sauf bonne raison. L'écriture de fonctions autonomes, utilisant uniquement des variables locales, permet de réutiliser ces fonctions dans d'autres programmes et de rechercher les erreurs en se concentrant uniquement sur la fonction concernée.

Comparer les deux programmes suivants.

```
def pluscinq(a):
    a=a+5
    return a

x = 1
print("x = ",x)
y = pluscinq(x)
print("x = ",x)
print("y = ",y)
```

```
def pluscinq(a):
    a[0]=a[0]+5
    return a

x = [1]
print("x = ",x)
y = pluscinq(x)
print("x = ",x)
print("y = ",y)
```

Dans d'autres langages de programmation, on parle de passage d'arguments

- par valeur : on transmet à la fonction la valeur de la variable *x*, qui est affectée dans une nouvelle variable *a*,
- par référence : on transmet à la fonction l'adresse mémoire de la variable *x*, qui est affectée à la variable *a* ; les variables *a* et *x* pointant vers le même objet, toute modification effectuée sur *a* est aussi effectuée sur *x*.

En Python, l'explication est différente. Chaque objet crée sur Python (nombre, chaine de caractères, liste…) est stocké en mémoire, et la variable correspondante (son nom) contient l'adresse mémoire (la référence) de l'objet. Tous les passages d'arguments s'effectuent ensuite par référence. Ainsi, lors de l'appel de la fonction pluscinq, les variables *a* et *x* pointent vers le même objet. La différence est que le code de gauche manipule des variables immuables et celui de droite des variables mutables.

Remarque : une fonction peut accéder à un objet mutable (et le modifier) sans que ce soit un argument de cette fonction.

Dans le programme de droite, la modification de l'objet mutable *a* par la fonction pluscinq modifie bien sûr l'objet *x* puisqu'il s'agit du même objet.

Dans le programme de gauche, l'objet *a* étant immuable, la ligne $a = a + 5$ ne modifie pas *a* (et donc pas *x*), mais elle crée un autre objet (local) portant le même nom *a*.

4. Booléens, tests, instructions conditionnelles et boucles

Un booléen est une variable égale à True ou False. Un booléen est un nombre déguisé comme le montre l'exemple de gauche :

```
In[1]: True + True
Out[1]: 2
```

```
In[2]: (1+1==2)
Out[2]: True
```

Un test est évalué et renvoie un booléen comme le montre l'exemple de droite.

Exercice : écrire sans utiliser de test d'égalité une expression équivalente à (bool_1 + bool_2 + bool_3 == 3).

Une instruction conditionnelle (**if**) est suivie d'un booléen. Les variables nulles ou vides sont interprétées comme False (None, 0, 0.0, 0j, [], "…), les autres comme True.
Ainsi, si a est un flottant, l'instruction **if** a: do_this() va appeler la fonction do_this() si et seulement si a est non nul.

Si une condition combine plusieurs tests successifs, ils sont exécutés en respectant la règle de priorité (and est prioritaire sur or), puis de gauche à droite.

Exercice : expliquer l'erreur dans le code suivant (liste est une liste !).

```
i = 0
while liste[i] < 100 and i < len(liste):
   i += 1
```

Pour interrompre une boucle **while**, on ajoute une condition après le **while**. Dans une boucle **for**, on peut utiliser **break**, ou **return** si la boucle est dans une fonction (on quitte alors la fonction). Une instruction **else** après une boucle va être exécutée après la boucle si celle-ci n'a pas été interrompue par **break** ou **return**.

5. Parenthèses et crochets

Les crochets [] servent à accéder au contenu d'une variable de type liste, chaine de caractères ou tableau numpy, en lecture et en écriture.

- a[0] pointe vers le premier élément de a, a[1] le second…
- a[-1] pointe vers le dernier élément de a , a[-2] l'avant-dernier…
- a[n:p] désigne la sous-liste d'éléments de a compris entre a[n] inclus et a[p] exclu. Si n est omis, la sous-liste commence à a[0] et si p est omis, la sous-liste se poursuit jusqu'au dernier élément de a.
- a[n:p:k] désigne la sous-liste d'éléments de a compris entre a[n] inclus et a[p] exclu avec un pas de k.

Exemple
Ainsi, si a = [1, 2, 3, 4, 5, 6, 7, 8], a[1:7:2] renvoie [2, 4, 6].
a[0] est un entier : a[0] est égal à 1.
a[0:1] est une liste : a[0:1] est égal à [1].

Les **parenthèses ()** servent à exécuter une fonction (appeler une fonction, call en anglais). Ainsi, le type d'erreur xx is not callable signifie souvent qu'on a mis des parenthèses à la place de crochets.

Avec une fonction f, x = f crée un alias de la fonction f, ce qui peut servir si des instructions conditionnelles permettent de sélectionner quelle fonction on va appliquer par la suite.

Dans l'instruction x = f(), les parenthèses indiquent qu'on exécute la fonction f : x recevra alors le résultat retourné par f (les fonctions sans return renvoient toutes None en Python). On doit bien sûr préciser les arguments de f entre les parenthèses s'il y en a.

Exercices

Exercice 1
Qu'affiche le code suivant ?
Expliquer.

```
def test(p):
    p = p + p
    print(p)
    return p

def test2(p):
    p[:] = p + p
    print(p)
    return p

p = [1,2,3]
test(p)
print(p)
test2(p)
print(p)
```

Exercice 2
Dans quel(s) cas la fonction permute échange-t-elle le contenu des listes l1 et l2 transmises en paramètres ?

```
def permute(l1,l2):
    l1, l2 = l2, l1

def permute(l1,l2):
    l1, l2 = l2[:], l1[:]

def permute(l1,l2):
    l1[:], l2[:] = l2, l1

def permute(l1,l2):
    l1[:], l2[:] = l2[:], l1[:]
```

Exercice 3
La fonction suivante modifie-t-elle la variable liste transmise en paramètre ?

```
def test(liste):
    l = liste   # crée une variable locale l
    l[0] = 2
```