TP 01 – Types (Partie 1)

Pour s'échauffer

Exercice 1 (**Affectation de variables**) Donner l'affichage des programmes ci-dessous.

Entrée [1]:
```
s = 0
s = s + 1
s = s + 2
print(s)
```

Out [1]:

Entrée [2]:
```
a = 1
b = 2
a = a-b
b = a+b
print(a,b)
```

Out [2]:

Entrée [3]:
```
a, b, c = 1, 2, 1
a = 2*c + 3*b
c = b
b = a
print(a,b,c)
```

Out [3]:

Exercice 2 (**Échange de deux variables**) Écrire un programme qui demande à l'utilisateur deux réels `a` et `b`, qui échange ces deux valeurs et qui renvoie les nouvelles valeurs de `a` et `b`. On pourra utiliser une valeur auxiliaire. ***Vérification :*** *On vérifiera que, si l'utilisateur choisit* `a=2` *et* `a=5`*, alors le programme renvoie* `5` *et* `2`*.*

Entrée [4]:
```
a = float(input('Donner un réel a:'))
b = float(input('Donner un réel b:'))
```

I Tour d'horizon rapide des principaux types structurés

En plus des nombres (types `int`, `float`) et des booléens (type `bool` : `True` et `False`), vous serez couramment amenés à utiliser les listes, les uplets et les chaînes de caractères, et parfois les dictionnaires.

- Une **liste** (type `list`) est une suite entre crochets d'expressions de toutes sortes, séparées par des virgules. Par exemple,

Entrée [5]:
```
L=  [1, 'ptsi', 1.8 , True , 51, 'a' , 2.6 ]
```

- Un **uplet** (type `tuple`) est une suite entre parenthèses (ou non, les parenthèses sont facultatives) d'éléments séparés par des virgules. Par exemple,

Entrée [6]:
```
u = (1, 'ptsi', 1.8 , True , 51, 'a' , 2.6)
v = 1, 'ptsi', 1.8 , True , 51, 'a' , 2.6
```

- Une **chaîne de caractères** (type `string`) est une suite de lettres, de signes de ponctuation et d'espaces (les caractères) non séparés, mise entre guillemets (ou apostrophes). Par exemple,

Entrée [7]:
```
s = 'PTSI Lycée Nelson Mandela'
```

- Un **dictionnaire** est un ensemble de paires « clef-valeur », écrit sous la forme `nom_dico= {c1 : v1, c2 : v2, ... , cn : vn}`. Par exemple,

Entrée [8]:
```
dico = {"voiture": "véhicule à quatre roues", "vélo": "véhicule à
deux roues"}
```

II Zoom sur : les listes (type list)

Une **liste** est une succession d'éléments rangés dans un certain *ordre*, séparés par des virgules et délimités par des crochets. Une liste peut contenir des éléments de types différents, comme des entiers, des chaînes de caractères, des flottants, des booléens...

Voici les commandes principales, illustrées sur l'exemple ci-dessous (en reprenant pour chaque commande la liste de départ).

Entrée [9]:
```
L = ['Lundi', 12, 'Info', 2.4]
```

Attention ! Chaque élément de la liste (et plus généralement d'une séquence) est *indicé* par un entier indiquant sa position. Dans une liste de longueur n, les éléments sont **indicés à partir de 0**, le dernier élément a donc pour indice $n-1$.

Syntaxe	Explication	Exemple
`L=[]`	Créer une liste vide	
`len(L)`	Renvoie la longueur d'une liste	`len(L)` renvoie
`L[i]`	Renvoie l'élément d'indice i*	`L[2]` renvoie `L[-1]` renvoie
`L[deb:fin]`	Renvoie la liste des éléments d'indice `deb` à `fin-1`	`L[0:2]` renvoie
`L[deb:]`	Renvoie la liste des éléments d'indice compris entre `deb` et le dernier indice	`L[2:]` renvoie
`L[:fin]`	Renvoie la liste des éléments d'indice compris entre le premier indice et `fin-1`	`L[:2]` renvoie
`L[d:f:pas]`	Renvoie la liste des éléments d'indice compris entre le premier indice et `f-1` avec un certain `pas`	`L[0:3:2]` renvoie
`x in L`	Renvoie `True` si `x` est dans `L` et `False` sinon	`12 in L` renvoie
`L[i]=x`	Remplace l'élément d'indice i par `x` (pas d'affichage)	`L[2]="Maths"` modifie la liste qui devient `L =`
`L.append(x)`	Ajoute l'élément `x` à la fin de la liste (pas d'affichage)	`L.append(1)` modifie la liste qui devient `L =`
`L.insert(i,x)`	Ajoute l'élément `x` en position `i` (pas d'affichage)	`L.insert(3,1)` modifie la liste qui devient `L =`
`L.remove(x)`	Enlève la première occurence de `x` dans la liste (pas d'affichage)	`L.remove(12)` modifie la liste qui devient `L =`
`L.pop()`	Supprime le dernier élément de la liste et l'affiche	`L.pop()` modifie la liste qui devient `L =` et affiche `2.4`
`L.pop(i)`	Supprime l'élément en position `i` de la liste et l'affiche	`L.pop(2)` modifie la liste qui devient `L =` et affiche
`del L[i]`	Supprime l'élément en position i	`del L[2]` modifie la liste qui devient `L =`
`L1+L2`	Concatène les deux listes	`[1,2]+["Maths", 3]` renvoie
`L*k`	Répète k fois la liste par concaténation	`[1,2]*3` renvoie
L.copy()	Création d'une copie de la liste `L`	
tuple(L)	Transforme la liste `L` en tuple et l'affiche	`tuple(L)` renvoie `('Lundi', 12, 'Info', 2.4)`

*De plus, si $i < 0$, `L[i]` renvoie le `i`-ième élèment en partant de la fin.

III Zoom sur : les uplets (type tuple)

Un uplet est (comme en mathématiques) une liste d'objets séparés par une virgule, entourée ou non de parenthèses (facultatives) : `(1,'a',3.5)` ou `1,'a',3.5`. La grande différence avec les listes est la suivante : les uplets ne sont pas *mutaples* : on ne peut pas modifier les éléments d'un uplet, ni en ajouter ou en supprimer.

Cependant, le reste des opérations sont les mêmes que pour les listes.

- On peut récupérer la longueur d'un uplet grâce à la commande `len`.
- On peut concaténer ou répéter des uplets grâce aux opérateurs `+` et `*`.
- Les opérations d'extractions sont les mêmes que pour les listes : `u[ ]`, `u[ : : ]`,..

On peut transformer un uplet en une liste grâce à la commande `list`. Enfin, les uplets permettent des affectations multiples : la commande `(a,b,c) = (1,2,3)` donne simultanément à `a` la valeur `1`, à `b` la valeur `2` et à `c` la valeur `3`.

IV Zoom sur : les chaînes de caractères (type str)

Une chaîne de caractères (type `string`) est une suite de lettres, de signes de ponctuation et d'espaces (les caractères) non séparés, mise entre guillemets (ou apostrophes) : `'abc'` ou `''abc''`. Comme les uplets, les chaînes de caractères ne sont *pas mutables* : on *ne peut pas* modifier les éléments d'une chaîne de caractères, ni rajouter ou supprimer des éléments. Le reste des commandes reste valable. On peut transformer une chaîne de caractères en liste, en uplet, en entier ou en flottant grâce aux commandes suivantes : `list`, `uplet`, `int`, `float`.

Exercice 3 Donner l'affichage des programmes suivants.

Entrée [10]:
```
s = 'abc de'
len(s)
```

Out [10]:

Entrée [11]:
```
'PT'+'SI'
```

Out [11]:

Entrée [12]:
```
'bla'*3
```

Out [12]:

Entrée [13]:
```
s='PTSI Lycee Nelson Mandela'
print(s[5])
print(s[-3])
```

Out [13]:

Entrée [14]:
```
s='PTSI Lycee Nelson Mandela'
print(s[1:3])
print(s[2:])
print(s[1:5:2])
```

Out [14]:

V Zoom : sur les dictionnaires (type dict)

Les dictionnaires sont des listes indexées non pas par des indices, mais par des clefs alphanumériques. On accède alors à chaque élément par sa clef (et non par son indice comme dans le cas des listes). Cela fonctionne comme un dictionnaire au sens usuel, dans lequel on accède à une définition à partir d'un mot.

- Un dictionnaire est un ensemble de paires « clef-valeur », écrit sous la forme :

Entrée [15]:
```
nom_dictionnaire= { c1 : v1, c2 : v2, ... , cn : vn}
```

Par exemple,

Entrée [16]:
```
moyen_locomotion = {"voiture": "véhicule à quatre roues",
"vélo": "véhicule à deux roues"}
```

- On obtient la *longueur* d'un dictionnaire `dico` (c'est à dire le nombre de paires qu'il contient) par `len(dico)`.

Entrée [17]:
```
len(moyen_locomotion) = 2
```

- On accède à une valeur du dictionnaire en utilisant sa clef grâce à la commande suivante `nom_dictionnaire[clef]`.

Entrée [18]:
```
moyen_locomotion["voiture"]
```

Out [18]:
```
"véhicule à quatre roues"
```

- Pour ajouter un élément à un dictionnaire, il suffit d'affecter sa valeur à une nouvelle clef (cette commande modifie le dictionnaire, mais ne renvoie pas d'affichage) avec la commande `nom_dictionnaire[nvclef]=nvvaleur`.

Entrée [19]:
```
moyen_locomotion["monocyle"]="véhicule à une roue"
print(moyen_locomotion)
```

Out [19]:
```
moyen_locomotion = {"voiture": "véhicule à quatre roues",
"vélo": "véhicule à deux roues",
"monocyle":"véhicule à une roue"}
```

- Pour savoir si une clef `c` est dans un dictionnaire `dico`, on utilise la commande `c in dico`.

Entrée [20]:
```
"vélo" in moyen_locomotion
```

Out [20]:
```
True
```

Entrée [21]:
```
"moto" in moyen_locomotion
```

Out [21]:
```
False
```

VI Lecture de programme

Pour chaque programme, noter dans chaque case du tableau les valeurs successives prises par les variables et s'il y a un affichage, écrire la valeur affichée ou faire une croix sinon.

(i)

```
L1 = [5, 4, 2]
L2 = [True, "Oui", 4]
print(L1[1])
print(L2[2])
print( L1 + [-1, 0])
print(L1 * 2)
print(L1[:2])
print(4 in L1)
print(-10 in L1)
```

	L1	L2	Affichage
Ligne 1			
Ligne 2			
Ligne 3			
Ligne 4			
Ligne 5			
Ligne 6			
Ligne 7			
Ligne 8			
Ligne 9			

(ii)

```
L = [-1, 2, 8, 4, 6]
print(L[1:3])
x = L[2]
print( x * 2)
y = x * 3
L.append(y)
print(L)
L[1:]
```

	L	x	y	Affichage
Ligne 1				
Ligne 2				
Ligne 3				
Ligne 4				
Ligne 5				
Ligne 6				
Ligne 7				
Ligne 8				

(iii)

```
L = []
print(L)
L.append("Bonjour")
L = L + [1,2]
print(L)
L[1] = 3
print(L)
L.pop()
```

	L	Affichage
Ligne 1		
Ligne 2		
Ligne 3		
Ligne 4		
Ligne 5		
Ligne 6		
Ligne 7		
Ligne 8		

(iv)

```
L = ["A", "M", "T", "H"]
L[0] = "I"
print(L)
x = 0
k = x + 2
print(L[k])
L[1] = L[3]
print(L)
del L[1]
```

	L	x	k	Affichage
Ligne 1				
Ligne 2				
Ligne 3				
Ligne 4				
Ligne 5				
Ligne 6				
Ligne 7				
Ligne 8				
Ligne 9				

(v)

```
L = [-1, 100, 2.0, 3, 6]
n = len(L)
print(n)
print(L[n-1])
print(L[0]!=L[2])
x = L[0]
y = L[2]
print(x > y)
```

	L	n	x	y	Affichage
Ligne 1					
Ligne 2					
Ligne 3					
Ligne 4					
Ligne 5					
Ligne 6					
Ligne 7					
Ligne 8					

(vi)

```
L1 = [0, 3, 5]
L2 = [4]
L3 = L1 + L2
print(L3, len(L3))
a, b = L3[0], L3[3]
print(a + b)
L3 = L2 * 5
print(L3, len(L3))
```

	L1	L2	L3	a	b	Affichage
Ligne 1						
Ligne 2						
Ligne 3						
Ligne 4						
Ligne 5						
Ligne 6						
Ligne 7						
Ligne 8						

VII Exercices

Exercice 4

1. Créer une liste L1 contenant les éléments 4, 7, 12, 11 et 8 dans cet ordre.

Entrée [22]:

2. Afficher le nombre d'éléments dans la liste L1.

Entrée [23]:

Out [23]:

3. Afficher le premier élément de L1.

Entrée [24]:

Out [24]:

4. Afficher le dernier élément de L1.

Entrée [25]:

Out [25]:

5. Afficher le troisième élément de L1 (le 12).

Entrée [26]:

Out [26]:

6. Afficher la portion de liste [7,12] à partir de L1.

Entrée [27]:

Out [27]:

7. Ajouter un 13 à la fin de L1.

Entrée [28]:

8. Vérifier, avec un booléen, que 13 est dans la liste.

Entrée [29]:

Out [29]:

9. Supprimer le 13.

Entrée [30]:

10. Vérifier, avec un booléen, que 13 n'est plus dans la liste.

Entrée [31]:

Out [31]:

11. Modifier le deuxième élément (le 7) pour qu'il soit égal à -1.

Entrée [32]:

12. Afficher la liste.

Entrée [33]:

Out [33]:

Exercice 5 (**Autour de la notion de copie**) Dans cet exercice, on cherche à comparer deux manières de copier une liste.

1. Définir les listes `L=['a', 'b','c','d','e']`, `copie1=L`, et `copie2=L.copy()`.

Entrée [34]:

2. Ajouter l'élément 7 à la fin de la liste `L`. Est-ce que les listes `copie1` et `copie2` sont aussi modifiées ?

Entrée [35]:

3. Ajouter la chaîne de caractères `'PTSI'` en position 1 de la liste `L`. Est-ce que les listes `copie1` et `copie2` sont aussi modifiées ?

Entrée [36]:

4. Supprimer le dernier élément de la liste `copie1`. Est-ce que les listes `L` et `copie2` sont aussi modifiées ?

Entrée [37]:

5. Supprimer l'élément en position 1 de la liste `copie2`. Est-ce que les listes `L` et `copie1` sont aussi modifiées ?

Entrée [38]:

Exercice 6 (**Liste de liste**) On peut rencontrer des listes de listes : ce sont des listes dont les éléments sont eux-mêmes des listes.

1. Définir la liste `L=[[1,4,7],[2,-1,5],[3,9]]`.

Entrée [39]:

2. Que renvoie `len(L)` ? Que renvoie `L[0]` ? Que renvoie `L[0][1]` ?

Entrée [40]:

Out [40]:

Entrée [41]:

Out [41]:

Entrée [42]:

Out [42]:

3. Trouver la commande qui permet de modifier l'élément `2` de la liste et de le remplacer par la valeur `10` et afficher la liste `L` ;

Entrée [43]:

Out [43]: