Les types de base en Python

Les types de base

Pour qu'un programme puisse manipuler une valeur, il faut qu'il connaisse le **type** de cette valeur. Par exemple `a + b` ne fera pas la même chose si `a` et `b` sont des nombres ou des textes. Certains langages ont un **typage statique**, c'est-à-dire qu'il faut indiquer le type de chaque variable avant de l'utiliser. C'est le cas en Java ou en C++ : `int a = 5;`. D'autres langages, comme Python, ont un **typage dynamique**. Le type de chaque variable est déterminé au fur et à mesure de l'exécution.
Il est possible de connaître le type d'une valeur en Python :

```
>>> type(1)
<class 'int'>
```

```
>>> type(1.0)
<class 'float'>
```

```
>>> type(True)
<class 'bool'>
```

```
>>> type("1")
<class 'str'>
```

Les types `int`, `float` et `bool` sont des **types simples**. Il servent de base pour ce qu'on appelle les **types construits**, comme les listes. Le type `str` est un type construit un peu particulier puisqu'il sert également de type de base pour construire d'autres types.
Il est possible de tester si une valeur est d'un type particulier :

```
>>> isinstance(5,int)
True
```

```
>>> isinstance(5,str)
False
```

```
>>> isinstance(1.0,int)
False
```

Les types construits

Les types simples permettent de stocker une valeur en mémoire. Pour stocker plusieurs valeurs, on peut utiliser plusieurs variables. Mais cela peut vite devenir fastidieux et particulièrement compliqué si on a un grand nombre de valeurs ou si leur nombre est arbitraire. Pour cela, on utilise des types construits, comme les listes. Python dispose de plusieurs types construits, pour des usages différents. Les principaux sont :

- Les **listes** (`list`), qui servent à stocker une liste de valeurs ou un tableau de valeur, où l'ordre des éléments est important.
- Les **p-uplets**, ou tuples, (`tuple`), qui servent également à stocker une liste de valeurs, mais contrairement aux listes, une fois définit, un p-uplet ne peut pas être modifié.
- Les **ensembles** (`set`), qui servent à stocker un ensemble de valeurs, sans répétition et sans ordre particulier.
- Les **dictionnaires** (`dict`), qui servent à associer des valeurs à des clefs, comme des définitions à des mots dans un dictionnaire.

Il est important de savoir identifier le bon type à utiliser en fonction des besoins.

Les p-uplets

Les p-uplets permettent de regrouper une liste de valeur, comme des coordonnées, des informations sur une personne… La différence avec une liste c'est qu'il est impossible de modifier ou d'ajouter une valeur. Il faut recréer un nouveau p-uplet avec la nouvelle valeur. En Python, ils se notent avec des parenthèses au lieu de crochets. Il est possible d'accéder aux éléments comme pour des listes, d'obtenir la longueur ou de les concaténer pour faire de nouveaux p-uplets.

```
>>> ()      # p-uplet vide
()
>>> ("Alan", "Turing", 1912, 1954)
('Alan', 'Turing', 1912, 1954)
>>> (5, 7)
(5, 7)
>>> coord = (3, 6)
>>> coord
(3, 6)
>>> coord[0]
3
```

```
>>> coord[1]
6
>>> debut = (1,) # pour un p-uplet
>>> fin = (3,)    # à 1 élément
>>> debut + fin
(1, 3)
>>> coord = coord + (4, 3)
>>> coord
(3, 6, 4, 3)
>>> len(coord)
4
```

Il est également possible de dépiler un p-uplet en plusieurs variables. Les parenthèses ne sont pas obligatoires s'il y a plusieurs éléments.

```
>>> (x, y) = (2, 3)
```

```
>>> x, y = (2, 3)
```

```
>>> (x, y) = 2, 3
```

```
>>> x, y = 2, 3
```

```
>>> x
2
>>> y
3
>>> x, y
(2, 3)
```

Cela permet d'effectuer plusieurs affectations sur une seule ligne. Cela permet également d'échanger les valeurs de 2 variables :

```
>>> a, b = 0, 5
>>> a, b = b, a
>>> a, b
(5, 0)
```

Exercice 1 :

1) Quelles sont les valeurs de a et b après cette suite d'instructions ?
2) Proposer un moyen d'échanger les valeurs de deux variables sans utiliser des p-uplets.

```
a = 1
b = 2
a = b
b = a
```

Une fonction peut également renvoyer plusieurs valeurs en utilisant un p-uplet :

```
def milieu(ptA, ptB):
    xA, yA = ptA
    xB, yB = ptB
    return ((xA+xB)/2, (yA+yB)/2)
```

```
>>> milieu((2, 4), (1, 7))
(1.5, 5.5)
>>> milieu((0, 1), (8, -5))
(4.0, -2.0)
```

Lorsqu'on manipule une liste de p-uplets, il est possible de les dépiler en parcourant une liste.

```
for x, y in [(1, 3), (2, 6)]:
    ...
```

La fonction spéciale **enumerate**(iterable) permet de parcourir à la fois les indices et les valeurs d'un itérable.

```
for i, v in enumerate(liste):
    ...
```

Conversions entre types

Il est possible de transformer une valeur en une valeur d'un autre type en effectuant une **conversion**. Par exemple, en faisant `1 + 2.1`, la valeur 1 est convertie en valeur décimale pour que l'addition soit effectuée. Par contre pour d'autres conversions, il faut l'indiquer explicitement à Python.

```
>>> int(3.1)
3
>>> int(3.9)
3
```

```
>>> str(1)
'1'
>>> str([1, 3,5])
'[1, 3, 5]'
```

```
>>> list((2, 3))
[2, 3]
>>> tuple([5, 3, 1])
(5, 3, 1)
```

N'importe quelle valeur peut être convertie en booléen. Une valeur nulle sera convertie en **`False`** et toute autre valeur sera convertie en **`True`**.

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool(-1)
True
```

```
>>> bool([])
False
>>> bool([1, 4])
True
>>> bool(())
False
```

```
>>> bool((1,))
True
>>> bool("")
False
>>> bool("bonjour")
True
```

Exercice 2 : Déterminer la valeur de chacune des variables.

```
a = int("123") + 5
b = "123" + str(5)
c = bool("False")
```

Il est également possible de convertir une chaîne de caractères en liste des symboles qui la composent.

```
>>> list("bonjour")
['b', 'o', 'n', 'j', 'o', 'u', 'r']
```

Exercice 3 : Quelle est la valeur de `[int(x) for x in str(123)]` ?

Les conversions sont souvent nécessaires lorsqu'on demande des valeurs à l'utilisateur :

```
un_texte = input("Quel est ton nom ? ")
un_entier = int(input("Donner un entier : "))
un_reel = float(input("Donner un nombre reel : "))
```

Enfin, la commande "magique" **`eval(TEXTE)`** permet d'évaluer une expression donnée sous forme de texte et d'obtenir le résultat correspondant, avec le bon type.

```
>>> eval("2+3")
5
>>> eval("[1, 2, 5]")
[1, 2, 5]
>>> eval("False")
False
>>> eval("3.5*2.6")
9.1
```

Exercices sur les types

Exercice 4 : On considère les différents types suivants : **int**, **float**, **str**, **list**, **dict** et **tuple**. Quels sont les couples de types pour `a` et `b` pour que l'expression `a + b` puisse être évaluée en Python.

Exercice 5 : On considère deux versions de la fonction `compte` qui prend deux paramètres de type **str**.

```
def compte(symbole, texte):
    compteur = 0
    for s in texte:
        if s == symbole:
            compteur = compteur + 1
    return compteur
```

```
def compte(symbole, texte):
    compteur = 0
    for i in range(len(texte)):
        if texte[i] == symbole:
            compteur = compteur + 1
    return compteur
```

Déterminer le type des variables dans chacune des versions.

Exercice 6 : On considère la fonction suivante :

```
def f(a):
    b = []
    for c in a:
        if c >= 0:
            b.append(c)
    return b
```

1) Essayer d'identifier les types possibles pour les différentes variables.
2) Déterminer le but de cette fonction.
3) Écrire le code d'une fonction faisant la même chose, mais avec des noms, pour la fonction et les variables, plus parlants.

Exercice 7 : On considère la fonction suivante :

```
def donne_antecedents(val, associations):
    antecedents = []
    for a in associations:
        if associations[a] == val:
            antecedents.append(a)
    return antecedents
```

1) Parmi les appels suivants, déterminer ceux qui sont valides.
   a) `donne_antecedents(4, {0: 0, 1: 1, 2: 4, 3: 9, -2: 4, -1: 1})`
   b) `donne_antecedents(4, [0, 1, 4, 9, 4, 1])`
   c) `donne_antecedents('chien', {'A': 'chat', 'B': 'chien', 'C': 'chien', 'D': 'poisson'})`
   d) `donne_antecedents('chat', {'chat': 'A', 'chien': 'B', 'chien': 'C', 'poisson': 'D'})`
   e) `donne_antecedents('chat', ['Alice', 'Bob', 'Charlie', 'Daniel'])`
   f) `donne_antecedents('chien', ['chat', 'chien', 'chien', 'poisson'])`
2) Déterminer le résultat obtenu pour les appels valides.
3) Expliquer le but de cette fonction.