Types et erreurs

I. Typage des données

1. Notion de type en Python

Le langage Python est un langage **fortement typé** : toutes les données manipulées ont un **type** qui indique comment interpréter et utiliser ces données (qui ne sont, à la base, qu'une suite de chiffres binaires) :[1]

```
>>> type(3)
<class 'int'>
>>> type([1, 2, 3])
<class 'list'>
>>> type('Bonjour')
<class 'str'>
```

Python est, de plus, un langage typé **dynamiquement** : le type est vérifié lors de l'exécution, *au dernier moment*. Si, au moment de l'exécution, on lui demande de faire une opération qui n'a pas de sens, il le signale en interrompant son exécution et en affichant une erreur :

```
>>> 1 + "Bonjour"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

L'aspect *dynamique* se traduit par le fait que Python ne détecte à l'avance, lorsque l'on écrit la fonction, s'il va y avoir des problèmes. Ainsi, Python accepte sans rien dire la définition de la fonction suivante :

```
def erreur_en_vue(n):
    x = n + 1 # n doit être un nombre
    y = "Bonjour" + n # n doit être une chaîne de caractère
```

Mais on ne peut pas l'utiliser avec un entier :

```
>>> erreur_en_vue(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in erreur_en_vue
TypeError: cannot concatenate 'str' and 'int' objects
```

[1] C'est indiqué `class` car en Python, toute donnée est un *objet*.

ou avec une chaîne de caractères :

```
>>> erreur_en_vue("tout le monde")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in erreur_en_vue
TypeError: cannot concatenate 'str' and 'int' objects
```

On notera que selon les cas, l'erreur est déclenchée à la ligne 2 ou la ligne 3.

2. Indication de types

Pour éviter cette sorte d'erreur, voici quelques règles à respecter :

Quelques règles

- ⋆ **Chaque variable** doit contenir des données d'un **unique type**.
- ⋆ Lors que l'on manipule un tableau, il faut distinguer les **indices** et les **valeurs**.
- ⋆ Concernant les **fonctions**, chaque **argument** doit avoir un **type précis** et si la fonction renvoie un résultat, celui-ci doit avoir un **type précis**. En particulier, tous les exécutions doivent se terminer par un `return`, et tous les `return` doivent renvoyer le même type de donnée.

Pour les fonctions, on s'efforcera de toujours **spécifier sa signature**, c'est-à-dire indiquer quels sont les types des arguments et quel type est retourné par la fonction.

Il est possible d'inclure cette signature directement dans la définition de la fonction, à l'aide d'une syntaxe dont voici un exemple, tiré d'un sujet de Centrale :

```
def uneFonction(n:int, X:[float], c:str, u) -> (float, int):
```

Ici, on définit une fonction `uneFonction` dont les arguments sont `n` de type `int` donc un entier, `X` de type `[float]` autrement dit un tableau de flottants (les crochets rappelant la notation des tableaux), `c` une chaîne de caractères, `u` de type non précisé (quelle drôle d'idée) et qui renvoit un couple constitué d'un flottant et d'un entier.

Remarque Cette notation est indicative, puisque comme indiqué précédemment, Python ne vérifie pas les types à l'avance. Ainsi, la définition de fonction suivante est acceptée sans broncher :

```
def erreur_en_vue(n:int) -> str:
    return "Bonjour" + n
```

De plus, certains programmes d'aide à l'édition de textes (des linters[2]) permettent d'analyser les indications de type, comme mypy.

II. Gestion des erreurs

Les erreurs dans l'exécution d'un programme entraîne normalement l'interruption de celui-ci (même si, mais c'est en dehors des limites du programme, une erreur peut être *récupérée* et traîtée plutôt que d'interrompt)

Pour vérifier qu'une condition est bien remplie, et que le signaler au besoin, on utilise une instruction spéciale : une **assertion**. La syntaxe de base est d'écrire `assert` suivi d'une condition booléenne. Par exemple :

```
assert x == 1
```

Le comportement du programme est le suivant :

1. Le programme teste la condition ;
2. si la condition s'évalue à `True` (si la condition est vraie), alors le programme continue ;
3. sinon le programme s'interrompt en faisant une erreur `AssertionError`.

Exemple Considérons par exemple la définition de la factorielle, avec une assertion vérifiant que l'entier passé en argument est positif (on a utilisé ici des indications de type) :

```
def factorielle(n : int) -> int:
    assert n >= 0
    f = 1
    for i in range(2, n + 1):
        f = f * i
    return f
```

Si on appelle la fonction avec un argument positif (et un entier car c'est ce qui est spécifié), alors tout se passe comme normalement :

```
>>> factorielle(10)
3628800
```

Sinon, l'exécution s'interrompt avec une erreur. Notons que l'on peut retrouver où l'erreur a été déclenchée, et donc dans quelle fonction l'assertion fausse a été trouvée :

```
>>> factorielle(-5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in factorielle
AssertionError
```

Exemples de conditions où une assertion est utile

- ⋆ on veut un entier positif ;
- ⋆ on veut un tableau non vide ;
- ⋆ on veut deux tableaux de même longueur ;
- ⋆ on veut qu'un entier soit un indice valide dans un tableau.

Remarques

- ⋆ On pourrait inclure dans les assertions la vérification des types de données. Mais il s'agit un problème différent, puisque l'usage d'une fonction avec les bons types (qui doivent être spécifiés d'une manière ou d'une autre) est de la responsabilité du programmeur. Les assertions permettent de tester d'autres conditions, en particulier la **validation des données**, qui ne peuvent être exprimées à l'aide d'indications de types.
- ⋆ Les assertions sont uniquement un instrument de débuggage. Ils servent à s'assurer que lors de la mise au point d'une fonction, de l'implémentation d'un algorithme, d'un programme, etc., les fonctions sont utilisées correctement. Une fois la mise au point terminée, il est possible de désactiver les assertions (qui, après tout, sont des tests et ont un coût à l'exécution).

Il est possible d'ajouter une chaîne de caractères à un `assert`, qui sera affiché en cas de condition non vérifiée (les « \ » en fin de ligne permettent de passer à la ligne pour des instructions trop longues) :

```
def min_diff(t1, t2):
    assert len(t1) != len(t2), \
        "les deux tableaux doivent avoir la même taille"
    assert len(t1) > 0, \
        "les tableaux ne doivent pas être vides"
    m = abs(t2[0] - t1[0])
    for i in range(1, len(t1)):
        v = abs(t2[i] - t1[i])
        if v < m:
            m = v
    return m
```

[2] Le lien pointe vers la page en anglais, car la page en français ne parle que du programme qui, originellement, donne son nom à ce type d'analyseur.