Quelques éléments de Python post-prépa

Laurent Signac – CC-BY-SA – 15-07-20 1853 a743b2ccfb836eedbfc9

1 Variables et types

Python est langage au typage *plutôt* fort, mais dynamique.

- typage *plutôt* fort signifie que Python ne fait pas de conversions implicites intempestives (on ne peut pas ajouter un nombre et une chaîne de caractères contenant un nombre par exemple)
- typage dynamique (≠ statique) signifie que le type des variables n'est pas annoncé lors de l'écriture du programme (on a un léger palliatif, les annotations), et qu'il peut varier au cours de la vie d'une variable.

Les noms de variable doivent être choisis intelligemment (Idiomatic Python, E. Franchi)

> Always use descriptive names, the longer the scope, the longer the name.

Les types Python peuvent avoir certaines propriétés :

- **collection** : le type permet de regrouper plusieurs objets
- **mutable** ou **non-mutable** : suivant que le contenu de l'objet peut être modifié ou non
- **hachable** : on peut calculer une sorte de valeur de contrôle de l'objet (précisément hachable = récursivement non-mutable)
- **séquence** : selon qu'on peut adresser les éléments par leur numéro d'ordre
- **itérable** : si l'on peut *épeler* les éléments de l'objet

Par exemple les listes sont des séquences mutables, itérables, non hachables. Les tuples sont des séquences non mutables, itérables et hachables…

2 Les Listes

Une liste Python est une **séquence** (donc une **collection**) **itérable** (toutes les séquences le sont) **mutable** (modifiable) et **non hachable** (les objets mutables ne sont pas hachables). Elle s'apparente à la structure de donnée abstraite : Tableau dynamique (le nom liste est plutôt mal choisi).

Conséquences :

- on peut ranger plusieurs objets dans une liste.
- on peut parcourir une liste avec une boucle `for`
- une liste est ordonnée (il y a un premier élément, un second etc…)
- on peut modifier les objets stockés à l'intérieur d'une liste
- on ne peut pas utiliser une liste comme clé d'un dictionnaire (parce qu'une liste n'est pas hachable).

2.1 Rappels

Quelques rappels sur la manipulation des listes :

```
l1 = [] # création d'une liste vide
l2 = [1, 2, [5, 6, 7], 8] # création d'une liste

l2.append(10)    # ajout d'un élément à la fin
print(l2[2])     # utilisation d'un élément
print(l2[2][1])  # liste de liste...
print(l2[-1])    # dernier élément
```

Assurez vous de maîtriser **parfaitement** les lignes qui précèdent.

3 Itérations

La syntaxe générale des boucles `for` est :

```
for <nom> in <objet iterable>:
    corps de la boucle
```

L'`<objet iterable>` peut être, par exemple : une liste, un tuple, un `range`…

`nom` vaudra tour à tour chaque élément de l'objet itérable.

Dans le cas où `nom` contient lui même plusieurs éléments, il est habituel d'utiliser le **tuple unpacking** :

```
for (c, l) in [(1, "A"), (2, "B"), (3, "C")]:
    print(c, l)
```

```
for (i, n) in enumerate([2, 3, 5, 7, 11, 13, 17]):
    print("premier", i, ":", n)
```

4 Conditionnelles

Les instructions conditionnelles sont de type : `if` ou bien `if` / `elif`…`elif`

Le nombre de `elif` est arbitraire. Dans les deux cas, on peut **ou pas** ajouter à la fin une clause `else`.

Conséquence, l'instruction suivante **n'existe pas**. Il s'agit en fait de deux instructions (`if` seul, puis `if`/`else`) mises bout à bout :

```
if ...:
  blabla
if ...:
  blabla
else:
  blabla
```

Exemple de construction (que se passe-t-il pour 15 ?)

```
l = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
for v in l:
    if v % 3 == 0:
        print(v, "est un triple")
    elif v % 5 == 0:
        print(v, "est un quintuple")
    else:
        print(v, "n'est ni un triple, ni un quintuple")
```

5 Fonctions / procédures

Ne pas confondre procédure et fonction

- Une fonction calcule. Elle **vaut** quelque chose, n'a pas d'effet de bord. Un appel de fonction est une expression.
- Une procédure n'a pas de valeur, mais elle **fait/modifie** quelque chose. Un appel de procédure est une instruction.

Conséquence : Si on vous demande d'écrire une fonction et qu'il y a des `print` dedans, mais pas de `return`, vous avez mal répondu à la question.

5.1 Portée des variables

Un nom (de variable ou de module ou de fonction) est recherché d'abord localement, puis dans la portée englobante, puis dans la portée globale, puis dans les noms prédéfinis (*built-in*).

5.2 Découpage

Il **faut** découper les problèmes en problèmes plus petits qu'on résout avec des fonctions séparées. Le résultat est un code plus simple à comprendre, plus facile à tester, et plus facile à maintenir.

6 Commentaires

Hors contextes scolaire, les commentaires sont faits pour **aider le lecteur humain** à comprendre le fonctionnement du programme. **C'est une erreur de commenter ce qui est une évidence**.

(très) Mauvais

```
# Initialise s à {1}
s = {1}
# Pour i variant de 2 à n-1
for i in range(2, n):
    # si le reste de la division de n par i est nul
    if n % i == 0:
        # ajouter i à s
        s.add(i)
```

Bon

```
# Met dans l'ensemble s les diviseurs stricts de n
s = {1}
for i in range(2,n):
    if n % i == 0:
        s.add(i)
```

7 Docstrings

Les docstrings sont des chaînes de caractères placées au début d'un objet à commenter (module, fonction…). La docstring peut ensuite être appelée avec la fonction `help`. Une docstring n'explique pas **comment fonctionne** le code (c'est le rôle des commentaires), mais **à quoi ça sert** et **comment on s'en sert**.

Par exemple :

```
import random

def melange(s):
    """ Renvoie une chaîne de caractère obtenue après
        mélange de la chaîne passée en paramètres
    """
    l = list(s)
    r = ""
    while len(l) > 0: # TODO : utiliser while l ?
        # À chaque tour, un élément est choisi, ajouté
        # à la chaîne et supprimé de la liste
        i = random.randint(0, len(l) - 1)
        r = r + l[i]
        del l[i]
    return r
```