Quelques éléments de programmation Python

Laurent Signac – CC-BY-SA – 23-09-21 0713 481f84ddd640d524db06

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 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 Collections

Les collections : listes, tuples, objets itérables etc. font la force et l'efficacité des langages interprétés modernes. Savoir les manipuler permet de concevoir des programmes concis, élégants et efficaces (car les algorithmes sous-jacents utilisés par les concepteurs de l'interpréteur sont généralement bien choisis).

- Hiérarchie des types standard[1]
- Types natif[2]
- **Collection** : objet destiné à contenir d'autres objets. Une collection peut être ordonnée (c'est alors une séquence) ou non. Les conteneurs standards sont : `list` `tuple` `str` `dict` (on mettra `str` aussi…)
- **Séquence** : collection ordonnée d'éléments indicés par des entiers. Les séquences Python sont par exemple : `list`, `tuple`, `str`. On peut accéder à un élément d'une séquence en précisant entre crochets le numéro d'ordre de l'enregistrement. La numérotation commence à 0.
- **Mutable** ou **Non-mutable** : se dit d'un objet selon que le contenu de l'objet peut être modifié ou non.
- **Hachable** : on peut calculer une sorte de valeur de contrôle de l'objet qui reste constante tout au long de la vie de l'objet (précisément, poulres types prédéfinis en Python, un objet est hachable s'il est récursivement non-mutable).
- **Itérable** : se dit d'un objet donc on peut épeler le contenu
- Type `list` : **séquence modifiable** (**mutable**) d'éléments éventuellement hétérogènes
- Type `tuple` : **séquence non modifiable** (**non mutable**) d'éléments éventuellement hétérogènes.
- Type `dict` (dictionnaire ou tableau associatif) : **collection** non ordonnée **modifiable** d'éléments éventuellement hétérogènes. Un tableau associatif est un type de données permettant de stocker des couples clé/valeur, avec un accès très rapide à la valeur à partir de la clé. Celle-ci ne peut bien sûr

[1] https://docs.python.org/fr/3/reference/datamodel.html#the-standard-type-hierarchy
[2] https://docs.python.org/fr/3/library/stdtypes.html

être présente qu'une seule fois dans le tableau. La clé doit être **hachable**. Le type `dict` fait partie des collections de type **Mapping** qui associent un objet à un autre.

- Type `set` : collection non ordonnée d'éléments hachables distincts.

3 Listes

Une liste Python est une **collection**, une **séquence 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 `list` est plutôt mal choisi à mon avis).

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 ou élément d'un ensemble (parce qu'une liste n'est pas hachable).

3.1 Création d'une liste ajout d'éléments et accès

```
>>> 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

>>> 3 + l2[1]     # utilisation d'un élément
---> 5
>>> l2[2][1]  # liste de liste...
---> 6

>>> l2.insert(1, 42) # Ajout en position 1
---> [1, 42, 2, [5, 6, 7], 8]

>>> l2.extend([1, 2, 3]) # Ajout depuis un itérable (ne pas confondre avec append)
---> [1, 42, 2, [5, 6, 7], 8, 1, 2, 3]
```

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

3.2 Slices

Ce qui est indiqué dans les crochets d'une séquence (liste, tuple, chaîne) s'appelle un `slice`. Il y a quelques règles à connaître :

- Forme générale : `[start:stop:step]`
- Seconde forme : `[start:stop]` équivalente à `[start:stop:1]`

Conséquence : si on veut préciser le pas, il **faut** qu'il y ait deux fois `:` dans le slice.

Les éléments compris dans le slice commencent sur `start` inclu, et finissent sur `stop` **non inclus**. Si `start` ou `stop` est strictement négatif, cela correspond aux éléments numérotés à partir de la fin (`-1` est le dernier, `-2` est l'avant dernier…, `-n` est le premier s'il y a `n` éléments dans la séquence).

Les slices permettent de déborder. Par exemple, si `stop` est strictement plus grand que le nombre d'éléments, le slice ira jusqu'à la fin de la séquence, et il n'y aura pas d'erreur.

Pour une séquence de `n` éléments :

- Si le pas est strictement positif :

  - et que `start` est omis (ou vaut `None`), on prend `start=0` (depuis le début inclus)
  - et que `stop` est omis (ou vaut `None`), on prend `stop=n` (jusqu'à la fin incluse)

- Si le pas est strictement négatif :
  - et que `start` est omis (ou vaut `None`), on prend `start=-1` (depuis la fin incluse)
  - et que `stop` est omis (ou vaut `None`), on prend `stop=-(n + 1)` (jusqu'au début inclus)

  Voici quelques exemples d'utilisation sur la chaîne `s = 'ABCDEFGHIJKLM'` de longueur `n = 13`

slice	résultat	Signification
`s[4]`	`'E'`	Élément 4 (le cinquième)
`s[-2]`	`'L'`	Élément `n-2` (avant dernier)
`s[2:6]`	`'CDEF'`	Éléments 2 à 5
`s[10:]`	`'KLM'`	Éléments 10 jusqu'à la fin
`s[:10]`	`'ABCDEFGHIJ'`	Les 10 premiers
`s[:-5]`	`'ABCDEFGH'`	Tout sauf les 5 derniers
`s[-5:]`	`'IJKLM'`	Les 5 derniers
`s[:6:2]`	`'ACE'`	Un sur deux dans les 6 permiers
`s[1::2]`	`'BDFHJL'`	Élements en position impaire
`s[0::2]`	`'ACEGIKM'`	Élements en position paire
`s[::-1]`	`'MLKJIHGFEDCBA'`	Chaîne à l'envers
`s[2:5:-1]`	`''`	Chaîne vide car `start <= stop` et `step < 0`
`s[-1::-2]`	`'MKIGECA'`	Du dernier jusqu'au début en en prenant 1 sur 2

3.3 Listes en compréhension

Vous devez savoir utiliser les écritures en compréhension (au moins pour les listes, il y a la même chose pour les dictionnaires et les ensembles)

```python
# Liste des entiers de 1 à 19
l1 = [i for i in range(1, 20)]
```

```python
# Liste des carrés des entiers de 1 à 19
l2 = [i**2 for i in range(1, 20)]
```

```python
# Liste des carrés de nombres impairs de 1 à 19
l3 = [i**2 for i in range(1, 20) if i % 2 == 1]
```

4 Littéraux

Attention aux confusions parfois subtiles entre des variables et des valeurs littérales.

Les valeurs littérales peuvent être :

- des chaînes de caractères : `"ceci est une chaîne"`
- des entiers exprimés dans plusieurs bases : `42`, `0x2a`, `0b101010`, `0o52`
- des flottants : `4.2`, `42e-1`
- des booléens (il n'y a que deux possibilités) : `True` et `False`
- la seule valeur de type `NoneType` : `None`
- Types composés :
  - tuple : `(4, 2, "quarante")`
  - liste : `[13, 21, 34, 55, 89]`
  - …

Les guillemets servent à marquer les **littéraux** chaînes de caractères (et pas les variables).

```
# a est une chaîne, mais ce n'est pas un littéral
# mais "quarante-deux" est un littéral chaîne de caractères
>>> a = "quarante-deux"

# guillemets autour du littéral
>>> print("quarante-deux")
'quarante-deux'

# pas de guillemets, puisque a n'est pas un littéral
>>> print(a)
'quarante-deux'

# enfin... on a le droit, de mettre des guillemets,
# mais ça ne fait peut-être pas ce qui est attendu...
>>> print("a")
'a'
```

5 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)
```

6 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`.

L'instruction `if`…`if`…`else` **n'existe pas**. Si on l'écrit, on a en fait deux instructions. Un `if` seul, puis une instruction `if`….`else`.

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")
```

7 Fonctions

7.1 Fonctions vs 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.

7.2 Découpage en fonctions

Le découpage en fonctions a beaucoup d'avantages :

- concision dans le code (factorisation) ;
- facilité de vérification (possibilité de tester la fonction à part, sans effet de bord) ;
- clarté du code.

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.