Python

Guide de base

Matteo Wei

Le but de ce document est de présenter les constructions syntaxiques, objets et fonctions de base de Python. Les sections se recouvrent un peu les unes les autres, donc il ne faut pas hésiter à sauter.

1 Syntaxe

1.1 Indentation significative

Python reconnaît les différents blocs de code à partir de leur indentation : d'une manière générale, les débuts de ligne dans un nouveau bloc doivent commencer alignées et plus à droite que la ligne qui introduit le bloc. On laisse d'une manière générale deux ou quatre espaces (facile avec la touche tabulation).
Par exemple, le premier programme ci-dessous affiche `10`, le deuxième `1`.

```
x, y = 0, 0
for i in range(10):
  y += 1
  x += 1
print(x) # Affiche x.
```

```
x, y = 0, 0
for i in range(10):
  y += 1
x += 1
print(x)
```

1.2 Commentaires

On peut ajouter des *commentaires* au code, *i.e.* du texte qui est ignoré par l'interpréteur et qui sert juste à expliquer le code, de deux manières différentes :

- Avec `#`, auquel cas tout ce qui suit sur la ligne est en commentaire.
- En encadrant du texte avec `"""`, auquel cas tout ce qui est entre les symboles est un commentaire (notamment, on peut sauter des lignes).

```
# Ceci est un commentaire.

x = 0 # On affecte 0 à la variable x.

"""
Ceci est un autre commentaire.
Noter le saut de ligne.
"""
```

2 Types, objets de base

2.1 `NoneType`

Le type contient un seul objet, `None`, qui sert généralement à représenter des valeurs défaut. Notamment, c'est ce que renvoie une fonction dont aucune valeur de retour n'a été spécifiée.

2.2 `bool`

Il contient deux objets, `True` et `False`, et représente les valeurs de vérité. C'est *a priori* le type des conditions.
Parmi les opérateurs pouvant être utilisés dessus, il y a les opérateurs logiques `and` (et), `or` (ou) et `not` (non).

2.3 `int`

Le type des entiers. Il supporte les opérations arithmétiques habituelles, `+`, `-`, `*` (multiplication), `**` (exponentation, *i.e.* `a ** b` est le nombre a^b), `//` et `%` (les deux derniers calculent le quotient et le reste dans la division euclidienne).
Pour abréger les instructions de la forme `x = x + k`, courantes quand on utilise des variables comme compteurs, on peut utiliser les symboles `+=` et `-=` : `x += k` revient à `x = x + k` ; `x` est incrémenté de `k`.

2.4 `float`

Le type des nombres réels, ou *flottants* (pour « nombre à virgule flottante »). Il supporte les opérations arithmétiques habituelles, `+`, `-`, `*`, `**`, `/`.
Les `int` sont automatiquement convertis en `float` au besoin (donc `1 / 2` ne produit pas d'erreur, par exemple) car le type `float` est plus général. Pour l'autre sens, en revanche, il faut convertir explicitement, en utilisant la fonction `int`, qui calcule la partie entière d'un flottant.
Il vaut mieux éviter d'utiliser des flottants si on en a pas besoin car ils sont sujet aux erreurs d'arrondi à cause de la manière dont ils sont représentés, contrairement aux entiers.

2.5 `list`

Ce type représente des listes d'objets (qui peuvent être de n'importe quel type).
Une *liste* peut être définie de deux manière : explicitement, ou par compréhension. Le premier cas consiste simplement à écrire les éléments de la liste, dans l'ordre, entre `[]`.
Le deuxième cas ressemble à la définition d'ensemble par compréhension en maths : `[expr for i in iter]`, où `iter` est un itérable (*c.f.* la section sur `for`), et `expr` est une expression (`i` peut apparaître dedans), retourne la liste dont les éléments sont les résultats de l'évaluation de l'expression pour les éléments de l'itérable, dans l'ordre.
Par exemple, la liste `[2 * i for i in range(4)]` est égale à la liste `[0, 2, 4, 6]`.
Une fois qu'une liste `l` est définie, on peut accéder à son élément d'indice `[i]` avec `l[i]` (en notant que son premier élément a comme indice 0). Celui-ci se comporte ensuite comme n'importe quel objet, on peut notamment le modifier avec quelque chose de la forme `l[i] = expr` par exemple.
On peut aussi rajouter un élément `x` à la fin d'une liste `l` avec `l.append(x)`. D'une manière générale, certaines fonctions appelées *méthodes* qui se réfèrent à un objet `obj` s'écrivent de la sorte : `obj.method(arg1, ..., argn)`.
On peut aussi citer les méthodes `.pop()` (élimine et retourne le dernier élément d'une liste), `.reverse()` (renverse une liste)... Voir ici pour plus de détails.
En plus de ça, la fonction `len` renvoie la longueur d'une liste, et l'opérateur `in` permet de tester l'appartenance d'un objet à une liste (`x in l` vaut `True` si, et seulement si, `x` apparaît dans `l`).
Dernière remarque, les éléments d'une liste peuvent aussi être des listes. Ainsi, si `l` est une liste de listes, on peut accéder à l'élément d'indice `j` de la liste d'indice `i` dans `l` avec `l[i][j]`.
Par exemple, le programme suivant affiche `4`.

```
matrix = [[0, 1, 2],
          [3, 4, 5],
          [6, 7, 8]]
print(matrix[1][1])
# matrix[1] est le deuxième élément de matrix, donc la deuxième ligne.
# matrix[1][1] est le deuxième élément de cette ligne, donc 4.
```

2.6 string

Il s'agit des chaînes de caractères, notées entre `'`, ou `"`. Il n'y a pas grand chose à dire de plus, on peut les voir comme des listes de lettres non modifiables (notamment, on peut itérer sur les lettres), donc on peut faire tout ce qu'on pouvait faire sur les listes sans modifier la liste.

2.7 dict

Une structure de données un peu plus complexe : le dictionnaire. Un dictionnaire est conçu pour stocker des valeurs indexées par des clés arbitraires, d'une manière qui rende

efficace la recherche de clés dans le dictionnaire, un peu comme une liste dont les indices seraient plus généraux que des entiers. Pour l'analogie avec les objets de la vie réelle, les clés sont les mots et les valeurs sont les définitions.
On peut initialiser un dictionnaire vide avec `{}`.
Ensuite, si `d` est un dictionnaire, `d[key] = value` rajoute l'association `key: value` dans le dictionnaire, ou la remplace s'il y en avait déjà une.
`d[key]` renvoie la valeur associée à `key` dans `d`, si elle existe.
Enfin, on peut vérifier si une clé est dans un dictionnaire avec le mot clé `in`.
Pour plus de détails, voir ici.

2.8 Fonctions de base

Certains opérateurs et fonctions sont compatibles avec la plupart des types qu'on a évoqué ici. C'est le cas des comparaisons (`==`, `!=`, `<`, `>`, `<=`, `>=`), mais aussi de la fonction `print`, qui permet d'afficher des objets.

3 Flot de contrôle

3.1 Tests conditionnels

Il y a quelques variantes, basées sur trois mots clés : `if` (si), `else` (sinon) et `elif` (sinon si).
La version la plus basique est le simple test :

```
if cond:
  instrs
```

Python évalue la condition `cond` (qui est de type `bool`), et exécute les instructions du bloc `instrs` si le résultat de l'évaluation est `True`.
On peut aussi avoir des tests de la forme « si ..., sinon ... » sans avoir à faire deux tests séparés, avec la syntaxe :

```
if cond:
  instrsif
else:
  instrselse
```

Plus généralement, on peut faire des choses de la forme

```
if cond0:
  instrs0
elif cond1:
  instrs1
...
else:
  instrsn
```

Python cherche le premier `if` ou `elif` tel que la condition correspondante est vraie, et exécute les instructions qui correspondent. Sinon, il exécute les instructions sous `else`.

3.2 Boucle `while`

Le mot clé `while` (tant que) permet de faire des boucles, qui tournent jusqu'à ce qu'une certaine condition soit vérifiée.
La syntaxe est :

```
while cond:
  instrs
```

Python évalue la condition `cond` (un `bool`). Si elle est vérifiée, il exécute `instrs`, puis reteste la condition, etc. Si la condition n'est pas vérifiée lors du test, on passe à la suite du code.
Par exemple, le code suivant affiche les entiers de 0 à 9.

```
x = 0
while x < 10:
  print(x)
  x += 1
```

Quand on utilise des boucles `while`, il faut faire attention que la condition finisse par ne pas être vérifiée, sous peine que le programme ne s'arrête pas (quoique cela peut être voulu, dans le cas d'un jeu vidéo par exemple, qui ne doit *a priori* pas s'arrêter tout seul).

3.3 Boucle `for`

Le mot clé `for` (pour) permet de faire des boucles particulières, qui consistent à itérer des opérations sur tous les éléments d'un objet *itérable*. La plupart des objets qui représentent des formes de collections sont des itérables : c'est le cas des listes, des chaînes de caractères (on itère sur les lettres), des dictionnaires (dans ce cas on itère sur les clés), *etc.*
La syntaxe est :

```
for i in iterable:
  instrs
```

Python exécute `instrs` pour chaque `i` dans `iterable`, dans un ordre qui dépend de la structure de `iterable`. Dans le cas des listes et des chaînes de caractères, l'ordre est l'ordre de lecture gauche à droite. Pour un dictionnaire, les clés sortent par ordre croissant.
Il est possible d'utiliser un type particulier d'itérable appelé `Range`, qui représente des progressions de nombres. Ils sont généralement créés par la fonction `range`, qui reçoit trois arguments : `range(start, stop, step)` créée un objet qui contient dans l'ordre les

nombres entre `start` (inclus) et `stop` (exclu), avec un saut de `step` entre deux nombres consécutifs.
À noter que `range` peut recevoir moins d'arguments : `range(n)` donne le même résultat que `range(0, n, 1)`, et `range(m, n)` que `range(m, n, 1)`.
Voici quelques exemples de codes faisant la même chose que dans la section précédente.

```
for x in range(10):
  print(x)

l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for x in l:
  print(x)
```

3.4 Fonctions récursives

Il y a une autre manière de gérer des boucles en Python : en utilisant des fonctions *récursive*, c'est-à-dire qui s'appellent elle-même dans leur code.
Voici un exemple typique :

```
def pow(a, n): #Calcule a^n
  if n == 0:
    return 1
  return a * pow(a, n - 1)
```

Plusieurs observations :

- L'idée de base de cette implémentation est d'exploiter le fait que pour tout n, on a $a^n = a \cdot a^{n-1}$. On peut remarquer que $n - 1 < n$: en itérant ce processus, on finira par atteindre 0, pour lequel on renvoie 1, et l'exécution s'arrête. Après cela, Python va remonter les appels en attente et retourner la bonne valeur à chaque fois, jusqu'à atteindre le premier et retourner a^n.
- On utilise dans ce code le fait que `return` stoppe l'exécution de la fonction. Cela permet d'éviter d'écrire un `else`, car dans le cas $n = 0$ la quatrième ligne n'est pas vue.
- Si n est strictement négatif, alors on atteindra jamais 0 et la fonction ne retourne jamais. Dans l'idée, on devrait avoir une boucle infinie. En pratique, ce n'est pas le cas, car Python limite la profondeur de la pile d'appel récursif.

D'une manière générale, une fonction récursive est implémentée de la manière suivante : on distingue un ou des cas simples (les *cas de base*) pour lesquels on sait ce qu'on doit renvoyer, et on renvoit cette valeur pour eux. Pour les autres cas, on fait des calculs qui peuvent impliquer un ou des appels de la fonction récursive, mais toujours sur des cas plus simples (dans un sens à préciser selon le contexte), afin qu'on atteigne toujours les cas de bases après un nombre fini d'appels.
Un autre exemple, un peu plus compliqué : une implémentation du minimax (peu efficace) pour le jeu de Nim.

```python
def minimax(p, n):
  # Retourne quel joueur a une stratégie gagnante depuis l'état
  # à n bâtons, où c'est au joueur p de jouer.
  if n == 1:
    return (3 - p) # (3 - p) vaut 1 si p = 2 et 2 si p = 1.
  for m in range(max(p - 3, 1), p):
    if minimax(3 - p, m) == p:
      return p
      # Si p a une stratégie gagnante depuis l'état (m, 3-p),
      # alors il en a une depuis (n, p).
  return (3 - p)
  # Si on n'a rien retourné avant, c'est que p est perdant depuis (n, p).
```

4 Mon code ne marche pas !

4.1 Identifier l'erreur

Il y a plusieurs cas de figure possible :

— Soit Python arrête l'exécution et renvoie un message d'erreur. Dans ce cas, lire le message d'erreur, essayer de le comprendre. Si l'étape d'avant ne marche pas, c'est souvent une bonne idée de le copier/coller dans un navigateur : Python est très utilisé donc il est très probable que quelqu'un ait déjà bloqué sur le même problème et ait demandé de l'aide sur un forum (typiquement StackOverflow). Par contre, les résultats seront probablement en anglais.

— Soit le code est correct, mais il y a une erreur d'implémentation quelque part (un décalage d'indice par exemple). Il faut passer au débuggage.

4.2 Débugger

Il y a quelques sources d'erreurs communes :

— Se tromper de symbole à un endroit (par exemple confondre `=` et `==`).

— Une erreur d'indexation, ou de condition d'arrêt (typiquement une `IndexError`). Dans ce cas, il faut se rappeller que les choses sont d'une manière générale indexées de 0 à $n-1$, pour un objet de longueur n. Le cas typique étant d'essayer de faire une opération sur l'objet en position n (qui du coup n'existe pas).

— Une disjonction de cas où un cas est oublié.

Une bonne solution pour progresser dans le débuggage consiste à intercaler des `print` un peu partout dans le code, pour essayer de cerner l'endroit où ça déraille (par exemple pour afficher la valeur d'une variable à chaque itération de la boucle).