Langage Python 3

Florent Bouchez Tichadou — Gwenaël Delaval

19 avril 2017

Ce document n'est pas un document de référence du langage Python (version 3), ni un cours complet sur ce langage. Le but est de fournir des éléments en Python, en regard des éléments concernant le langage algorithmique utilisé dans le cours. Ce document suit donc autant que possible la même structure que le document intitulé « Langage algorithmique/Pseudo-code/Langage pivot ».

0.1 Informations générales sur Python

Python est un langage interprété. On peut soit interpréter directement un programme écrit en Python (ci-dessous à gauche), soit lancer l'interpréteur en ligne de commande pour taper interactivement des commandes (ci-dessous à droite).

```
% python3 mon_fichier.py
Hello World!
```

```
% python3
>>> 5+3  # commentaire sur l'addition
8
```

Dans la suite du document, les lignes qui commencent par >>> sont entrées dans l'interpréteur python, et le résultat est affiché sur les lignes suivantes.

Tout ce qui suit le caractère '#' jusqu'à la fin de la ligne, est considéré comme un commentaire et ignoré par Python.

La structure d'un programme, notamment le regroupement d'instructions dans les fonctions ou les structures de contrôle, est définie par **l'indentation des instructions**, i.e., le nombre d'espaces en début de ligne. Une séquence d'instructions d'indentation identique groupe ces instructions dans une structure de contrôle.

1 Déclarations de variables et types

Il n'y a pas à proprement parler de déclaration de variable en Python. Les variables existent et peuvent être utilisées, lues, et modifiées, à partir du moment où elles ont été *définies*. Cependant, toutes les variables ont un *type* déterminé automatiquement par l'*inférence de type*.

En revanche, toute variable doit avoir été définie (initialisée) avant d'être utilisée.

```
>>> a = 3
>>> type(a)
<class 'int'>

>>> a = a + xy
NameError: name 'xy' is not
    defined
```

```
b = True
c = 'x'
str = "Hello␣World!"
tab1 = [3, 8, 5]
tab2 = [0] * 20 # tableau
    # de 20 cases initialisées à 0
```

```
cpl = (12, 7)
(a, b) = cpl

LMAX = 1000

None # rien / valeur non
    existante
```

Il n'y a pas de notion de constante en Python. Il faut utiliser des variables globales, et par convention des lettres majuscules (cf. LMAX ci-dessus).

1.1 Utilisation de bibliothèques

Certaines fonctions ou valeurs sont définies dans des bibliothèques qui doivent être chargées avant d'être utilisables. C'est le cas par exemple de la bibliothèque mathématiques qui définit sinus, cosinus, π, etc.

```
import math
perimetre = 2 * rayon * math.pi
```

```
from math import pi
perimetre = 2 * rayon * pi
```

1.2 Types structurés

Les types structurés permettent de regrouper plusieurs valeurs.
En Python, les types complexes qui nous intéressent sont les suivants :

— les listes (**list**) en Python sont très proches des *tableaux* des autres langages impératifs. Le premier élément est d'indice 0, et on peut accéder et modifier les éléments par leur indice. Une différence majeures et que ces listes sont dynamiques, i.e., leur taille est variable même après définition (on peut ajouter ou supprimer des éléments) ; on peut également connaitre la longueur d'une liste grâce à la fonction **len**.
— les tuples (**tuple**) sont des n-uplets non mutables ;
— les dictionnaires (**dict**), qui associent à chaque *clé* une valeur (la clé est une valeur de type standard).

Une valeur de type structuré peut contenir des valeurs de n'importe quel type (dont des valeurs de types structurés : il est possible de construire des listes de dictionnaires de tuples d'entiers...).

Le langage Python ne permet pas d'exprimer les notions de « listes d'entiers », « listes de flottants », ou plutôt ne permet pas de les distinguer : une liste contenant des entiers et une liste contenant des flottants auront le même type **list** .

NB : une même structure peut contenir des valeurs de types différents : une même liste peut contenir à la fois des entiers, des booléens, des listes de listes,... **Sauf exception bien comprise, de telles pratiques doivent être évitées : elles rendent difficiles la relecture, la compréhension et le débogage des programmes.**

```
tab = [0] * 20
tab[0] = 12
tab[1] = 34
tab[-1] = 5 # modifie le dernier élément
```

```
point = (0.2, 1.1)
cercle = (point, 2.5)
(x, y) = point
```

1.3 Opérations courantes

Add., mul., sous.

```
2 + 3
v * 5
12 - x
```

Divisions

```
7.0 / 3.0   # 2.333...
7 / 3       # 2.333...
7 // 3      # 2
7.0 // 3.0  # 2.0
7 % 3       # 1
```

Comparaisons

```
3 == 5      a != 4
v < 0       r <= 2.5
r > PI      y >= 1
```

Booléens

```
b1 and b2
b1 or b2
not b
```

1.4 Accès et assignations

```
v = 12
i = i + 1
i += 1
```

```
b = True
b = b or False
r = 2.718
```

```
# c est une chaîne de caractères
c = 'A'
ch = "Hello world!"
```

```
# ou bien
ch = 'Hello world!'


cpl = (3, 8)
upl = (3, -1, 12, 5, 7)

(i,j) = cpl
```

```
p = {} # dictionnaire vide
p['x'] = 3.1
p['y'] = r

c = { 'pos':p, 'diam':10.5 }


nb = 42   # nb est un int
nb = 2.1  # changé en float
```

```
t = [1, 5, 9]
t[0] = 0


S = [] # liste vide
S.append(9)
S.append(3)
S.append(-5)
S.append(0)
# S vaut maintenant [9,3,-5,0]
```

1.5 Types construits

Il est également possible de construire un objet regroupant plusieurs « propriétés » (champs), de manière plus propre qu'avec un dictionnaire. On définit pour cela une *classe* (un nouveau « type ») et la façon de créer une instance de ce type *via* une fonction spéciale (__init__) appelée lors de la création de l'objet. Notez que dans la classe, on fait référence à l'objet par la variable spéciale self (« soi-même »).

```
class Ensemble:
    def __init__(self):
        self.tab = [0] * LMAX
        self.longueur = 0
```

```
>>> e = Ensemble() # création d'une instance
>>> e.longueur = 1
>>> f = Ensemble() # nouvelle instance
>>> f.tab[0] = 12
```

Il est également possible d'ajouter des *méthodes* (fonctions) qui opèrent directement sur nos objets.

```
    def augmente (self, x):
        self.longueur += x
```

```
>>> e.augmente (3)
>>> e.longueur
4
```

1.6 Opérations d'entrées/sorties

Permettent d'« afficher à l'écran » et de « lire au clavier ».

La fonction **input** lit une ligne de texte sur l'entrée standard, et renvoie la chaîne de caractères représentant cette ligne. Il est possible de convertir la ligne lue soit en utilisant directement le type attendu (par exemple **int**), soit en utilisant **eval** qui évalue la ligne avec l'interpréteur python. Dans ce dernier cas, le type de la valeur renvoyée dépend donc de la valeur entrée : **int** si c'est un entier, **float** si c'est un flottant, etc.

```
print ("Appuyer sur une touche pour continuer.")
print ("Valeur de x:", x)
print ("Cercle de diamètre", c['diam'], "aux coordonnées", c['pos'])

print ("Quel est votre nom? ", end="") # pas de retour à la ligne
nom = input()

print ("Entrez une valeur entière au clavier: ", end="")
x = int(input())
```

1.7 Opérations conditionnelles

```
if x > 0:
    print (" positif ")

if e.longueur == 0:
    pass
else:
    print (E)
```

```
if s.longueur == n:
    print ("Séq. pleine")
elif s.longueur == 0:
    print ("Séq. vide")
else:
    print ("mi-plein?")
```

```
if type(nb) == int:
    print ("entier:", nb)
else:
    print ("réel:", nb)
```

1.8 Structures de répétition

```
# Parcours de haut niveau
for e in E:
    print (e, "est dans l'ensemble")

print "Séquence dans l'ordre:"
for x in S:
    print (" ", x, end="")
print () # retour à la ligne

# Parcours de bas niveau
for i in range(n) # de 0 à n-1 :
    print ("Indice", i, ":", t[i])
```

```
print ("Séquence en décroissant:")
for i in range (S.longueur-1, -1, -1):
  print(" ", S.tab[i], end="");

x = input()
while x != 0:
  print ("Erreur: vous devez entrer zéro")
  x = input()
```

1.9 Multiples conditions

Il n'y a pas d'équivalent du «switch» (en C) ou du «selon» (notation algorithmique) en Python, il faut utiliser la structure conditionnelle **if** ... **elif** ... **else**.

```
if direction == 'nord':
    y = y + 1
elif direction == 'ouest':
    x = x - 1
else:
    print ("Direction interdite !")
```

```
if r < 0: r = -r
elif (r >= 0) and (r < 1): r = r * r
elif r >= 1: pass
else: print ("Heu... ?!?")
```

1.10 Fonctions

```
def debug(message):
    print ("Debug: " + message)

def addition (x,y):
    return x+y

z = addition(2, 8)

def gps ():
    x = longitude()
    y = latitude()
    return (x,y)
```

```
def deux_valeurs ():
    a = int(input())
    b = int(input())
    return (a,b)

(x, y) = deux_valeurs()
```

Il n'existe pas à proprement parler de mécanisme général pour modifier un paramètre d'une fonction (et que cette modification soit visible depuis l'appelant), car Python gère de manière particulière les références (voir section suivante). Nous décrivons plus loin plusieurs façons de contourner ce problème.

1.11 Copies et références

Pour comprendre les références en Python, il faut d'abord savoir que les valeurs en Python sont soit *mutables* (i.e., modifiables), soit *non-mutables* (non modifiables). Le programmeur **ne peut pas** choisir ce qui est mutable ou non, c'est le langage qui l'impose.

Non-mutables

int
float
str
tuple

Mutables

list
dict
class

Il n'est donc pas possible de modifier une chaine de caractère (ex : `chaine[3] = 'a'`). De même si vous « modifiez » une variable entière, en réalité l'ancienne variable est détruite, et une nouvelle est créée (avec le même nom) !

En revanche, les tableaux par exemple sont bien modifiables (ex : `tab[3] = 12`), on n'a pas créé de nouveau tableau mais modifié l'ancien.

Lors de l'assignation ou le passage en paramètre d'un type mutable, la valeur est **copiée**, alors que lors de l'assignation (ou le passage en paramètre) d'un type non-mutable, c'est la **référence** qui est copiée.

Copies à l'assignation

```
>>> a = 12
>>> b = a
>>> b = 18
>>> print (a, b)
12 18

>>> def inc(x):
...         x+=1
...         return x
...
>>> print (inc(a))
13
>>> print (a)
12
```

Références à l'assignation

```
>>> t = [1, 5, 9]
>>> u = t
>>> t[1] = 12
>>> print (u, t)
[1,  12,  9]  [1,  12,  9]

>>> p = { 'nom':"Bécassine", 'age':68 }
>>> q = p
>>> q['nom'] = "Germaine"
>>> print (p, q)
{'nom': 'Germaine', 'age':  68} {'nom': '
     Germaine', 'age':  68}
```

Dans le cas d'un type mutable, si l'on souhaite tout de même effectuer une copie à l'assignation, il faut le faire explicitement, soit « à la main » (en copiant les champs un par un), ou en utilisant le constructeur du type (**list** ou **dict** par exemple), ou *via* le module `copy`.

Dans le cas inverse, il faut ruser en encapsulant notre non-mutable dans un mutable.

```
>>> v = list(t) # crée une copie
>>> t[0] = 55 # v n'est pas affecté (mais u si)
>>> print (u, t, v)
[55,  12,  9]  [55,  12,  9]  [1,  12,  9]
```

```
>>> a = [12]
>>> b = a
>>> b[0] = 18
>>> print (a, b)
[18] [18]
```

1.11.1 Références et fonctions

Dans le cas d'un appel de fonction, les conventions restent les mêmes : les mutables sont des copies et les non-mutables sont des références. Dans l'exemple suivant, `a` n'est pas modifiée par l'appel tandis que `t` l'est.

```
>>> def test(x, t):
...         x = x+1
...         t[2] = 99
...
>>> a = 5
```

```
>>> t = [0, 8, 3]
>>> test (a, t)
>>> print (a, t)
5 [0, 8, 99]
```

Si on a besoin qu'une fonction modifie un non-mutable, il existe plusieurs possibilités.

1. Renvoyer un tuple comme résultat : souvent la contrainte de passer un argument par référence vient des langages qui n'ont qu'une seule valeur résultat d'une fonction. Python n'a pas cette contrainte ;
2. Utiliser une variables globale (**fortement déconseillé**) ;
3. Encapsuler dans un mutable : liste, dictionnaire, ou objet.

Le plus souvent, la première solution est la bonne...

```
>>> def modif (a, b):
...         a += 1
...         b = 12
...         return (a, b)
...
>>> x = 3
>>> y = 5
>>> (x, y) = modif (x, y)
>>> print (x, y)
4 12

>>> compteur = 0
>>> def compte():
...         global compteur
...         compteur += 1
...
>>> compte()
>>> print (compteur)
1

>>> def dicts(args):
...         args['a'] = 'nouvelle␣valeur'
...         args['b'] = args['b'] + 1
...
>>> d = {'a': 'ancienne␣valeur', 'b': 99}
>>> dicts(d)
>>> print(d)
{'a': 'nouvelle␣valeur', 'b': 100}
```

1.12 Listes chaînées et autres

À venir...