Introduction à Python

Cours 1 : notions fondamentales

Justin Cano

Polytechnique Montréal
Département de Génie Électrique.

5 mai 2021

Plan de la section 1

Qu'est-ce que Python ?

Quelques propriétés du serpent

- Python est un langage informatique interprété, ce qui lui permet d'être indépendant de la plateforme qui l'exécute.
- C'est un langage de type script, utilisant la hiérarchisation par indentation.
- Il s'agit aussi d'un langage non-typé (c'est-à-dire sans déclaration *a priori*, chose qui peut être un *Pharmakôn*[a] : salvatrice car rapide mais trompeuse car simpliste).

a. Antidote et poison, notion platonicienne.

Depuis quand existe-il ?

Il fut mis au point en 1991, par l'ingénieur néerlandais *Guido van Rossum* (né en 1956) et nommé ainsi en l'hommage aux Monty Pythons, dont Rossum était un inconditionnel.

Plan de la section 2

Nota bene

Dans cette section, nous exécuterons des opérations élémentaires directement dans **la console Python**. Ce qui est rapide (on n'a pas à sauvegarder un fichier et on peut voir étape par étape ce qu'il se passe).

Sur Linux lancer la commande `python3` dans un terminal : la console apparaîtra.

Sur Windows exécutez ceci dans *IDLE*, la console qui vient avec le téléchargement de `Python`.

Mémoire et allocation I

D'un point de vue sémantique coder c'est **jouer avec des variables** : l'étymologie du terme *algorithmique*[1] en atteste.
Cependant, en pratique, il s'agit de placer des données quantifiées en bits dans la mémoire vive. Python gère cela de manière automatisée, à la manière de `Java` mais à l'inverse de `C/C++` (voir la notion de *pointeur*).

Définition

*On appelle **adresse** l'endroit où se trouve stockée la **valeur** d'une **variable**.*

Exemple

On affecte dans python la variable n à la valeur 2 :

```
>>> n = 2
```

où se trouve cette dernière ?

Mémoire et allocation II

Pour répondre à la question précédente on va utiliser la routine `id()` :

```
>>> id(n)
94539112154496
```

dans **mon ordinateur**[2], cette variable voit son **premier octet** occuper la 94539112154496-ème case de la mémoire vive.

Rassurez-vous : `Python` gère ceci tout seul et sait retrouver les variables.

1. Signifiant calcul en arabe.
2. Et pas nécessairement le votre, le mien a pris cette décision aujourd'hui car il y'avait de la place dans ces octets-ci.

Copie profonde et copie superficielle I

Disons qu'on veuille maintenant jouer avec une autre variable *m* qu'on va, par paresse, affecter à la même valeur que *n* :

```
>>> m = n
```

Question

Où est-située m ?

```
>>> id(m)
94539112154496
```

Ce nombre me dit quelque chose... je vais **comparer les deux adresse** pour en être sûr (== est un opérateur de test d'égalité)...

```
>>> id(m)==id(n)
True
```

Copie profonde et copie superficielle II

Ce sont les mêmes adresses !
Toutefois... on peut remarquer

```
>>> n=2
>>> m=n
>>> id(m)==id(n)
True
>>> m=4222
>>> id(m)==id(n)
False
>>> m
4222
```

Conclusion ? Python va déplacer *m* car il «voit» qu'on a choisi une autre valeur. Mais il peut ne pas le faire...

Copie profonde et copie superficielle III

En particulier, lorsqu'on manipule des objets (entités composées de plusieurs variables, dont nous reparlerons), on peut avoir quelques surprises.

Exemple

Supposons un tableau contenant deux nombres :

```
>>> tableau = [1958,1994]
```

Je veux faire une copie de ce dernier :

```
>>> nouveau_tableau = tableau
```

Je change d'avis... je veux modifier le premier élément du nouveau tableau :

```
>>> nouveau_tableau[0]=2021
```

Copie profonde et copie superficielle IV

Et là c'est le drame...

```
>>> nouveau_tableau
  [2021, 1994]
>>> tableau
  [2021, 1994]
```

Les deux tableaux sont identiques ! Modifier l'un entraîne la modification de l'autre...

Solution

Procéder à la copie profonde des objets.

Copie profonde et copie superficielle V

Exemple

Retour à l'exemple :

On importe le module **copie** :

```
>>> import copy
>>> tableau = [1958,1994]
```

on fait deux copies d'objets :

```
>>> copie_superficielle = copy.copy(tableau)
>>> copie_profonde = copy.deepcopy(tableau)
```

Copie profonde et copie superficielle VI

On remarque que les deux copies ont donné un bon résultat ce coup-ci (seul «tableau» est modifié) :

```
>>> tableau[0]=2021
>>> tableau
[2021, 1994]
>>> copie_superficielle
[1958, 1994]
>>> copie_profonde
[1958, 1994]
```

La différence entre les deux est que si on avait des **objets imbriqués** (tableaux de tableaux par exemple) on n'aurait pas eu de copie des objets en question.

Copie profonde et copie superficielle VII

Exemple

Soit le tableau contenant un tableau contenant l'élément de valeur 500. On procède aux deux copies :

```
>>> tableau = [500]
>>> tableau_du_tableau = [tableau]
>>> copie_superficielle = copy.copy(tableau_du_tableau)
>>> copie_profonde = copy.deepcopy(tableau_du_tableau)
```

Modifions l'unique case (case zéro du zéroième tableau) en lui donnant une valeur qui est un texte «*hello*» :

```
tableau_du_tableau[0][0]="hello"
```

Résultat de la manœuvre :

Copie profonde et copie superficielle VIII

```
>>> copie_superficielle
[['hello']]
>>> copie_profonde
[[500]]
```

CQFD : on voit tout de suite la différence... la copie superficielle voit son élément (le tableau) pointer vers la même adresse...

Les types I

Définition

Il s'agit de structures de données. C'est-à-dire une matérialisation standardisée des ***variables*** *dans la* ***mémoire*** *composée d'une série de* 0/1 *(bits).*

Les types II

Liste : (non exhaustive, types principaux)

Booléens types ne pouvant que prendre les valeurs `True` ou `False`. Nominativement, ils ne prennent qu'un bit de mémoire, mais cela est erroné : les «cases mémoire» sont en réalité souvent regroupées par paquet de huit bits, qu'on appelle **octets**. L'opérateur de conversion est la fonction `bool()` qui permet de transformer toute variable en booléen.

Entiers En `Python 3` il n'y a qu'un type d'entiers (signés) qui va sélectionner automatiquement le nombre de case mémoire dont il a besoin (les entiers codés sur N bits peuvent appartenir à l'intervalle $[-2^{N-1}+1, 2^{N-1}]$). L'opérateur de conversion est la fonction `int()`.

Les types III

Flotants les nombres **flotants** sont des nombres à virgule, généralement encodés sur huit octets [3]. Le fait est qu'ils diffèrent des entiers de par leur structure à point flottant, contrairement aux premiers à points fixes.
Sans entrer dans les détails, disons que les flottants sont comme de l'écriture scientifique.

Exemple

Soit deux nombres entiers $s = 314$ (chiffres significatifs) et $e = -2$ (exposants), soit un nombre à virgule $v = \pi$. On peut (avec une certaine précision dépendant de s) l'encoder dans le flottant $f = [a, e]$ qui sera lu ainsi par la machine :

$$v \approx s \times 10^e \Leftrightarrow \pi \approx 314 \times 10^{-2} = 3,14.$$

L'opérateur de conversion des floats est la fonction `float()`.

Les types IV

Astuce : mettre un .0 à la fin d'une déclaration de nombre entier va créer un flottant.

Exemple

`f = 25.0` est reconnu comme un flottant tandis que `i = 25` est reconnu comme un nombre entier.

String Appelées chaînes de caractères qui encodent les caractères alphanumériques. Contrairement aux langages de plus bas niveau que `Python`, tels le `C`, les notions de caractère (élément) et de chaîne (tableaux de caractère) sont liées. Ainsi `a = "y"` et `b = "yaourt"` sont des `string`. L'opérateur de conversion des string (**très utile pour afficher du texte sur la console**) est la fonction `str()`.

3. Appelés `double` en `C` car ils comportent 64 bits ($64 = 8 \times 8$), soit le **double** des traditionnelles architectures 32 bits, populaires lorsqu'ils furent instaurées. En effet, en `C`, le `float` de base est de 32-bits, convention reprise par d'autres langages (`C++`,`Java`...).

Opérateurs mathématiques de base

Soient `x` et `y` deux variables, voici les syntaxes de base des opérations élémentaires :

- Addition : `x+y`;
- Soustraction : `x-y`;
- Multiplication : `x*y`;
- Division : `x/y`;
- Division entière : `x//y` (condensé de `int(x/y)`);
- Modulo : `x%y`, c'est-à-dire x $[y]$;
- Puissance : `x**y`, c'est-à-dire x^y.

Note : L'addition marche aussi sur les «string», cela les concatène : `"J'aime "+"les yaourts"` donnera `"J'aime les yaourts"`.

Opérateurs mathématiques : notation condensée

Soient `x` et `y` deux variables, voici les syntaxes des opérations condensées (x est impliqué deux fois dans l'équation)

Incrémentation : `x += y` $\Leftrightarrow x = x + y$;

Soustraction : `x -= y` $\Leftrightarrow x = x - y$;

Gain[4] : `x *= y` $\Leftrightarrow x = x * y$;

Normalisation : `x /= y` $\Leftrightarrow x = x/y$;

4. Ou facteur d'échelle.

Opérateurs de comparaison

Soient `x` et `y` deux variables numériques (flottantes ou entières), les opérateurs de comparaison produisent un **Booléen** qui est le résultat du test :

- Égalité : `x == y` ;
- Différence : `x != y` ;
- Inégalité : `x < y` (stricte) `x <= y` (large) ;
- Inégalité(bis) : `x > y` (stricte) `x >= y` (large) ;

Opérateurs logiques

Soient `x` et `y` deux variables booléennes (qui peuvent être des résultats de test)

Égalité : `x == y` ;

Différence : `x != y` ;

et : `x&&y` ou bien `x and y` (résultat de «si x et y sont vrais») ;

ou : `x||y` ou bien `x or y` (résultat de «si x ou y sont vrais») ;

ou exclusif : `x^y` (résultat de «si **soit** x **soit** y est vrai, mais **pas les deux à la fois**»).

Impression de variables I

Car la console c'est bien mais... si on veut un jour écrire des fichiers de code plus complexe (et les sauvegarder) on doit s'en passer.

Question

Comment afficher alors les résultats des programmes ?

On peut demander une impression des résultats dudit script dans la console, et ceci en utilisant la routine `print()` qui prend en argument un **string**.

Impression de variables II

Exemple

On veut afficher un nombre n avec un peu de texte autour.

Le code proposé est :

```
n = 2
string = "Le nombre que je veux afficher est " + str(n)
string+=" !"
print(string)
```

on l'enregistre dans un fichier `1_print_example.py` (l'extension `.py` est celle du python) et on l'exécute, on obtient le résultat suivant dans la console :

```
Le nombre que je veux afficher est 2 !
```

Impression de variables III

Remarquer la conversion (cast) d'entier (`int`) vers `string` avec `str()` et l'incrémentation `+=` qui rajoute le point d'exclamation dans la chaîne de caractère. En effet, le code suivant donne une erreur :

```
>>  string = "Le nombre que je veux afficher est " + n
TypeError: must be str, not int
```

une erreur qui est de type. On ne peut ajouter deux types différents. Toutefois... `print(n)` fonctionne et retourne 2. Les développeurs de `Python` ont rendu flexible ce dernier, rigoureusement c'est `print(str(n))` qu'il faudrait écrire.

Plan de la section 3

Listes I

Définition

Une liste est un ensemble ordonné d'éléments de même nature.

On peut voir ces dernières comme un tableau. Rigoureusement, il s'agit d'une **classe** au sens **programmation objet du terme**. C'est pour cela que les listes ont quelques fonctions (méthodes) qui leur sont propres.

Exemple

Définissons la liste de chaîne de caractères suivante :

```
personnes = ["Jose","Axel","Justin"]
```

Listes II

Les listes sont caractérisées par leurs éléments, on appelle leur **nombre** la **longueur de cette dernière** qu'on peut extraire grâce à `len()`

Exemple

```
print(len(personnes))
```

la précédente commande donnera 3 sur la console.

Pour accéder au i-eme élément d'une liste `l` on doit écrire `l[i]`. Ainsi, dans notre exemple `personnes[1]`, le deuxième élément, contient la valeur `"Axel"`.
Note : Remarquer que les listes de longueur l sont indexées de 0 à $l-1$.
Attention : dans notre exemple, `personnes[3]` donne une erreur : il n'**existe pas** de quatrième élément.

Listes III

Astuce : le dernier élément d'une liste peut être indexé par -1 : `personnes[-1] = personnes[2] = "Justin"`. Généralisable aux avant-derniers -2 et etc...

- On affecte une valeur à un élément de liste ainsi : `liste[index] = valeur`
- On lit une case d'une liste ainsi : `valeur=liste[index]`

Listes IV

Définition (Dépilage et empilage)

On peut retirer le dernier élément d'une liste (dépilage, `pop()`) ou au contraire en ajouter un (empilage, `append()`)

Exemple

```
>>> personnes = ["Jose","Axel","Justin"]
>>> personnes.pop()
'Justin'
>>> personnes
['Jose', 'Axel']
>>> personnes.append("Agathe")
>>> personnes
['Jose', 'Axel', 'Agathe']
```

Listes V

Note : il existe également les routines `liste.remove(index)` pour supprimer des éléments et `liste.insert(index)` pour en insérer.

Définition (Concaténation)

Déjà vue pour les `string`, elle est généralisable à toutes les listes ***pourvu qu'elles contiennent le même type de données****.*

Exemple

```
>>> notes_axel = [20.0,15.0,13.0]
>>> notes_jose = [12.0,22.0,3.1459]
>>> notes_classe = notes_axel + notes_jose
>>> notes_classe
[20.0, 15.0, 13.0, 12.0, 22.0, 3.1459]
```

Listes VI

Astuce : si par hasard vous voulez initialiser une liste vite, faites `liste= [ ]` . Cela peut être utile.

Définition (Tri)

*On peut également trier une liste **numérique** par valeurs croissantes ou décroissante facilement avec `sort()` et `reverse()`.*

Exemple

```
>>> notes_classe.sort()
[3.1459, 12.0, 13.0, 15.0, 20.0, 22.0]
>>> notes_classe.reverse()
>>> notes_classe
[22.0, 20.0, 15.0, 13.0, 12.0, 3.1459]
```

Remarque : les fonctions présentées ci-haut sont des fonctions qui sont **propres** à chacun des objets considérés. On écrira `liste.sort()` et non pas `liste(sort)` .

Itérer sur les éléments d'une liste

Dans le cas où l'on souhaite faire ceci, on peut invoquer une boucle `for`. C'est une boucle à laquelle on fournit une liste sur laquelle elle ira itérer. En voici la syntaxe :

```
for i in liste:
  instructions
```

Exemple

Calcul de moyenne de classe :

```
notes_classe = [12.0,16.3,5.5,10.0,15.0,14.0,2.5,14.0]
total=0
for note in notes_classe:
  total+=note
moyenne=total/len(notes_classe)
print(moyenne)
```

la réponse est 11.16... pas fameux.

Hiérarchie en Python

Remarque importante : L'indentation est **la clef** en `Python`. Une boucle (qui est un test «tant que»), un autre test ou une déclaration de fonction (voir plus tard) voit nécessairement du code lui être subordonnée.
L'instruction qui lance ce code est marquée d'un « :» tandis que le code qu'elle subordonne est indenté (tabulation).
Ainsi, on a la syntaxe suivante :

```
test ou declaration de procedure:
  lignes de code subordonnees (niveau 1)
  test:
    lignes de codes subordonnees (niveau 2)
  lignes de code subordonnees (niveau 1)
```

Ainsi **une mauvaise indentation** peut faire **échouer un programme**.

Que faire si on veut itérer juste N fois (sans liste) ?

En créant... une liste. La fonction `range(N)` crée une liste l de N éléments entiers commençant par 0 :

$$l = [0, 1 \dots N-1] \in \mathbb{N}^N$$

on peut également spécifier d'autres options à range : ne pas commencer la liste en zéro 0, sortir N nombres pairs... etc...

Exemple

Dénombrer et afficher les nombres entiers multiples de 7 entre 1 et 200

```
list = []
for i in range(1,200): # pour tous les entiers
  if i%7==0: #si i modulo 7 est nul
    list.append(i) # Ajout a la liste
print(list) # On s'amuse a les aficher
print(len(list)) # On les compte
```

on trouve 28 nombres dans ce cas.

Boucles conditionnelles I

Cette classe de boucle va faire tourner un code **tant que** (*while*) la condition spécifiée est vraie. Voici la syntaxe d'une telle boucle :

```
while condition:
  instructions
```

Boucles conditionnelles II

Exemple

Approximer par une fraction entière à 0.01 près :

```
valeur = 3.14159
erreur = 9999 # valeur aberrante
numerateur=1
denominateur=1
while abs(erreur)>0.01: #Tant que la valeur absolue de l'
                        erreur est mauvaise
  if erreur<-0.01:
    numerateur+=1 # On est trop bas!!!
  else:
    denominateur+=1 # On est trop haut!!!
  erreur = float(numerateur)/float(denominateur) - valeur
# Attention aux types la division doit etre un flottant!

print("Pi = " + str(numerateur) + "/" + str(denominateur))
```

Réponse... les fameux 22/7 de l'école primaire.

Plan de la section 4

Les fonctions

Parfois, lorsqu'on construit du code... on se répète. Ou on se répéte à quelques paramètres près. Voyons comment faire pour se simplifier la tâche.

Définition (Fonction)

Une fonction est une procédure appelable avec des paramètres en argument (entrée) ainsi que possiblement des résultats en sortie.

La syntaxe de **déclaration** est la suivante :

```
def ma_fonction(param1,param2):
  instructions
  resultat1 = ....
  resultat2 =
  return resultat1,resultat2
```

lorsqu'on voudra l'invoquer, on écrira
`resultat_1,resultat2 =ma_fonction(param1,param2)`.

Exemple d'usage I

Exemple

On veut coder rapidement un programme qui comporte les fonctionnalités suivantes :

1. Afficher les prénoms d'une liste d'élève d'une classe ;
2. Précisant leur âge respectif ;
3. Précisant la moyenne d'âge de la classe.

Exemple d'usage II

```
eleves = ["Jose","Axel","Agathe"]
ages = [62,26,3]
# Declaration des fonctions
def print_eleve(nom,age):
  string = nom + " qui a " + str(age) + " ans"
  print(string)
def moyenne_age(tab_ages):
  s=0
  for age in tab_ages:
    s+=age # somme des ages
  s/=len(tab_ages) # moyenne (flottant)
  return str(int(s)) # partie entiere et string
# Execution du code
print("Dans la classe, il y a :")
for i in range(len(eleves)):
  print_eleve(eleves[i],ages[i])
print("Moyenne d'age du groupe "+moyenne_age(ages)+" ans")
```

Exemple d'usage III

Qui donne le résultat suivant :

```
Dans la classe, il y a :
Jose qui a 62 ans
Axel qui a 26 ans
Agathe qui a 3 ans
Moyenne d'age du groupe 30 ans
```

et ce avec un code assez compact.

Exemple d'opérations récursives I

Exemple

Supposons qu'on veuille savoir la date à laquelle une épargne de 10000$ aura doublé. Son taux initial est de 2%, pour chaque année passée dans la banque, ce taux augmente lui-même de 3%. Coder une routine faisant le calcul annuel de l'argent dans le compte, sachant que vous y versez 16$ annuels.

Exemple d'opérations récursives II

```
#Routine de recursion
def compte(somme,taux):
  somme = somme*taux + 16.0 # Recursion sur l'argent
  taux = taux*1.03 #Recursion sur le taux
  return somme, taux

somme = 10000
taux = 1.02
annee = 0
while somme<20000: #Tant que cela n'a pas double
  somme,taux = compte(somme,taux)
  annee+=1
print("Rentable a partir de "+str(annee)+" ans")
```

La réponse est 7 ans !

The end

Merci pour votre attention.

Exercices suggérés :

- Refaire les exemples ;
- S'aider du code-compagnon (voir archive en `.zip`) ;
- Inventez-vous quelques *scenarii* pratiques à partir de ces derniers.