Bases de python

March 21, 2021

1 Introduction

1.0.1 Un langage interprété, de haut niveau et orienté objet

Python est un langage interprété de haut niveau. Par ailleurs, c'est aussi un langage orienté objet ce qui signifie qu'il se base sur des classes qui permettent de simplifier son utilisation.

- **Un langage interprété** : Un **langage interprété** s'oppose à un **langage compilé**, qui demande une compilation du code pour rendre un programme exécutable. Le principal avantage d'un langage interprété est sa simplicité de débogage. À l'inverse, son principal défaut est sa lenteur. À la différence d'un langage compilé, les blocs de code ne sont pas optimisés afin d'être traités de manière extrêmement rapide par la machine. Dans le cas de Python, on traite le code ligne par ligne avec l'interpréteur Python.
- **Un langage de haut niveau** : un langage de haut niveau est un langage qui se rapproche le plus possible du langage naturel, c'est-à-dire qui se lit « comme il s'applique ». Il est donc extrêmement simple à mettre en œuvre mais peu optimisé. À l'inverse, un langage de bas niveau va se rapprocher le plus possible du langage de la machine sur lequel il s'applique. Le langage de bas niveau le plus connu est le langage assembleur.

En tant que langage, Python se base sur quelques principes : + Il est basé sur l'indentation. + Il est extrêmement souple. + Ses règles sont fixées par la communauté Python.

1.0.2 LES INTERPRÉTEURS : PYTHON ET IPYTHON

L'interpréteur, c'est l'outil qui traduit votre code source en action. IPython peut être vu comme une version évoluée de l'interpréteur Python classique.

L'interpréteur Python – une calculatrice évoluée L'interpréteur Python est un outil riche mais qui vous permet aussi de faire des calculs simples. Ainsi, une fois Python lancé depuis l'invite de commande, on pourra avoir :

[4]:
```
4+11
```

[4]:
```
15
```

Vous pouvez utiliser cette ligne pour soumettre du code directement ou pour soumettre des fichiers en Python (au format .py)

[5]:
```
x=3
```

[6]:
```
print(x)
```

```
3
```

L'interpréteur Python peut être utilisé comme une simple calculatrice avec des opérations mathématiques classiques :

[7]:
```
2+5
```

[7]: 7

[8]:
```
2*5
```

[8]: 10

[9]:
```
2/5
```

[9]: 0.4

[10]:
```
2-5
```

[10]: -3

[11]:
```
2**5
```

[11]: 32

[12]:
```
5%2
```

[12]: 1

On voit ici que la **puissance** est notée ** (à ne pas confondre avec ^ qui est un opérateur logique). Le **modulo** % permet d'extraire le reste de la division entière et % 2 nous permet de savoir si un nombre est pair ou impair. Comme vous pouvez le voir dans cet exemple lorsqu'on divise deux entiers, on obtient bien un nombre décimal (2/5 =0,4).

Vous pouvez bien sûr coder dans l'interpréteur Python directement mais nous avons tendance à préférer un interpréteur amélioré : IPython.

1.0.3 Les principes

Python utilise comme opérateur d'allocation le signe =. Lorsqu'on alloue une valeur à n'importe quelle structure en Python, on utilise donc l'opérateur = :

[13]:
```
var1=20
```

La variable var1 a donc comme valeur 20. Le type de cette variable est inféré automatiquement par Python, ce qui est une spécificité de ce langage. Les habitués d'autres langages comme C pourront être déroutés par cette spécificité. On ne donne jamais le type d'une variable, c'est Python qui se charge de le deviner.

Les différents types primitifs sont les suivants : + int : entier + float : nombre décimal + str : chaîne de caractère + bool : booléen (True ou False)

On peut identifier le type d'une variable en utilisant **type()**.

[14]:
```
var1=10
var2=3.5
var3="Python"
var4=True
```

[15]:
```
type(var1)
```

[15]:
```
int
```

[16]:
```
type(var2)
```

[16]:
```
float
```

[17]:
```
type(var3)
```

[17]:
```
str
```

[18]:
```
type(var4)
```

[18]:
```
bool
```

Si on change la valeur d'une variable en utilisant l'opérateur d'allocation (=), on change aussi son type :

[19]:
```
var4=44
```

L'autre principe de base de Python, c'est l'indentation. Ceci veut dire que les limites des instructions et des blocs sont définies par la mise en page et non par des symboles spécifiques. Avec Python, vous devez utiliser l'indentation et le passage à la ligne pour délimiter votre code. Ceci vous permet de ne pas vous préoccuper d'autres symboles délimiteurs de blocs. Par convention, on utilise une indentation à quatre espaces. Par exemple, si on veut mettre en place une simple condition :

[20]:
```
if var1<var4:
    var1=var4*2
```

1.0.4 Un langage tout objet

Python est un langage orienté objet. Ceci veut dire que les structures utilisées en Python sont en fait toutes des objets. Un objet représente une structure qui possède des propriétés (ce sont des caractéristiques de l'objet) et des méthodes (ce sont des fonctions qui s'appliquent sur l'objet). Il est issu d'une classe qui est définie de manière simple. En Python, tout est objet. Ainsi les variables, les fonctions, et toutes structures, sont des Python Object. Cela permet une grande souplesse.

Par exemple, une chaîne de caractères est un objet de la classe str. Ses propriétés et méthodes peuvent être identifiées en utilisant le point après le nom de l'objet (en utilisant la tabulation avec IPython, toutes les propriétés et méthodes d'un objet apparaissent).

[21]:
```
chaine1="Python"
chaine1.upper()
```

[21]:
```
'PYTHON'
```

On voit ici que la méthode .upper() de l'objet « chaîne de caractères » permet de mettre en majuscule les termes stockés dans la chaîne de caractères.

1.0.5 Les opérateurs logiques

Si on veut vérifier qu'un objet est bien de la classe attendue, le plus efficace est d'utiliser l'opérateur **is** ou **is not** :

[22]:
```
type(chaine1) is str
```

[22]:
```
True
```

Par ailleurs, les opérateurs logiques de Python sont résumés dans le tableau suivant :

LES STRUCTURES (TUPLES, LISTES, DICTIONNAIRES)

Python est basé sur trois structures de références : les tuples, les listes et les dictionnaires. Ces structures sont en fait des objets qui peuvent contenir d'autres objets. Elles ont des utilités assez différentes et vous permettent de stocker des informations de tous types.

Ces structures ont un certain nombre de points communs : + Pour extraire un ou plusieurs objets d'une structure, on utilise toujours les crochets [ ] + Pour les structures indexées numériquement (tuples et listes), les structures sont indexées à 0 (la première position est la position 0)

1.0.6 Les tuples

Il s'agit d'une structure rassemblant plusieurs objets dans un ordre indexé. Sa forme n'est pas **modifiable (immuable)** une fois créée et elle se définit en utilisant des parenthèses. Elle n'a qu'une seule dimension. On peut stocker tout type d'objets dans un tuple. Par exemple, si vous voulez créer un tuple avec différents objets, on utilise :

[23]:
```
tup1=(1, True, 7.5,9)
```

On peut aussi créer un tuple en utilisant la fonction tuple(). L'accès aux valeurs d'un tuple se fait par l'indexation classique des structures. Ainsi, si on veut accéder au troisième élément de notre tuple, on utilise :

[24]:
```
tup1[2]
```

[24]:
```
7.5
```

Les tuples peuvent être intéressants car ils requièrent peu de mémoire. D'autre part, ils sont utilisés en sorties des fonctions renvoyant plusieurs valeurs (voir plus loin sur les fonctions).

Les tuples en tant que structures sont des objets. Ils ont des méthodes qui leur sont propres. Celles-ci sont peu nombreuses pour un tuple :

[25]:
```
dir(tup1)
```

[25]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'count',
 'index']
```

[26]:
```
tup1.count(9)
```

[26]:
```
1
```

1.0.7 Les listes

La liste est la structure de référence en Python. Elle est **modifiable** et peut contenir n'importe quel objet.

Création d'une liste On crée une liste en utilisant des crochets :

[27]:
```
liste1=[3,5,6, True]
```

On peut aussi utiliser la fonction list(). La structure d'une liste est modifiable. Elle comporte de nombreuses méthodes : + .append() : ajoute une valeur en fin de liste + .insert(i,val) : insert une valeur à l'indice i + .pop(i) : extrait la valeur de l'indice i + .reverse() : inverse la liste + .extend() : étend la liste grâce à une liste de valeurs

Les listes possèdent d'autres méthodes notamment : + .index(val) : renvoie l'indice de la valeur val + .count(val) : renvoie le nombre d'occurrences de val + .remove(val) : retire la première occurrence de la valeur val de la liste

1.0.8 Extraire un élément d'une liste

Comme nous avons pu le voir plus haut, il est possible d'extraire un élément en utilisant les crochets :

[28]:
```
liste1[0]
```

[28]:
```
3
```

On est souvent intéressé par l'extraction de plusieurs éléments. On le fait en utilisant les deux points :

[29]:
```
liste1[0:2]
```

[29]:
```
[3, 5]
```

[30]:
```
liste1[:2]
```

[30]:
```
[3, 5]
```

Dans cet exemple, on voit que ce système extrait deux éléments : l'élément indexé en 0 et celui indexé en position 1. On a donc comme règle que i : j va de l'élément i inclus à l'élément j non inclus. Voici quelques autres exemples :

[31]:
```
# Extraire le dernier élément
liste1[-1]
```

[31]:
```
True
```

[32]:
```
# Extraire les 3 derniers éléments
liste1[-3 :]
```

[32]: [5, 6, True]

1.0.9 Un exemple

Créez une liste de 7 pays. Essayer d'extraire les trois premiers et les trois derniers.

[ ]:

[ ]:

[ ]:

Il existe d'autres fonctions sur les listes qui nous intéresseront plus tard dans ce chapitre.

1.0.10 Les comprehension lists

Il s'agit de listes construites de manière itérative. Elles sont souvent très utiles car elles sont plus performantes que l'utilisation de boucles pour construire des listes.

En voici un exemple simple :

[33]:
```
list_init=[4,6,7,8]
list_comp=[val**2 for val in list_init if val % 2 == 0]
```

[34]:
```
list_comp
```

[34]: [16, 36, 64]

La liste liste_comp permet donc de stocker les éléments pairs de list_init mis au carré. On aura :

[35]:
```
print(list_comp)
```

```
[16, 36, 64]
```

Cette notion de comprehension list est très efficace. Elle permet d'éviter du code inutile (boucles sur une liste) et est plus performante qu'une création de liste itérativement. Elle existe aussi sur les dictionnaires mais pas sur les tuples qui sont immuables. Nous pourrons utiliser des comprenhension lists dans le cadre de la manipulation de tableaux de données.

1.0.11 Les chaînes de caractères – des listes de caractères

Les chaînes de caractères en Python sont codées par défaut (depuis Python 3) en Unicode. On peut déclarer une chaîne de caractères de trois manières :

[36]:
```
chaine1="Python pour le data scientist"
```

[37]:
```
chaine2='Python pour le data scientist'
```

On utilisera le plus souvent la première. Une chaîne de caractères est en fait une liste de caractères et nous allons pouvoir travailler sur les éléments d'une chaîne de caractères comme sur ceux d'une liste :

[38]:
```
print(chaine1[:6])
print(chaine1[-14:])
print(chaine1[15:20])
```

```
Python
data scientist
data
```

Les chaînes de caractères peuvent être facilement transformées en listes :

[39]:
```
# on sépare les éléments en utilisant l'espace
liste1=chaine1.split()
print(liste1)
```

```
['Python', 'pour', 'le', 'data', 'scientist']
```

[40]:
```
# on joint les éléments avec l'espace
chaine1bis=" ".join(liste1)
print(chaine1bis)
```

```
Python pour le data scientist
```

Il existe de nombreuses méthodes sur les chaînes de caractères en Python. Un certain nombre sont rassemblées dans le tableau suivant :

Il existe de nombreuses autres spécificités liées aux chaînes de caractères que nous découvrirons tout au long de l'ouvrage.

1.0.12 Les dictionnaires

Les dictionnaires constituent une troisième structure centrale pour développer en Python. Ils permettent un stockage clé-valeur. Jusqu'ici nous avons utilisé des structures se basant sur une indexation numérique. Ainsi dans une liste, on accède à un élément en utilisant sa position list1[0]. Dans un dictionnaire, on va accéder à un élément en utilisant une clé définie lors de la création du dictionnaire.

On définit un dictionnaire avec les accolades :

[41]:
```
dict1={"cle1":[1,5], "cle2":True, "cle3":3}
```

Cette structure ne demande aucune homogénéité de type dans les valeurs. De ce fait, on pourra avoir une liste comme valeur1, un booléen comme valeur2 et un entier comme valeur3.

Pour accéder à un élément d'un dictionnaire, on utilise :

[42]:
```
dict1["cle2"]
```

[42]:
```
True
```

Pour afficher toutes les clés d'un dictionnaire, on utilise :

[43]:
```
dict1.keys()
```

[43]:
```
dict_keys(['cle1', 'cle2', 'cle3'])
```

Pour afficher toutes les valeurs d'un dictionnaire, on utilise :

[44]:
```
dict1.items()
```

[44]:
```
dict_items([('cle1', [1, 5]), ('cle2', True), ('cle3', 3)])
```

On peut facilement modifier ou ajouter une clé à un dictionnaire :

[45]:
```
dict1["cle4"]=4
```

On peut aussi supprimer une clé (et la valeur associée) dans un dictionnaire :

[46]:
```
del dict1["cle4"]
```

Dès que vous serez plus aguerri en Python, vous utiliserez davantage les dictionnaires. Dans un premier temps nous avons tendance à privilégier les listes aux dictionnaires car elles sont souvent plus intuitives (avec une indexation numérique). Toutefois un Pythoniste plus expert se rendra compte rapidement de l'utilité des dictionnaires. On pourra notamment y stocker les données ainsi que les paramètres d'un modèle de manière très simple. De plus, la souplesse de la boucle for de Python s'adapte très bien aux dictionnaires et les rend très efficaces lorsqu'ils sont bien construits.

1.1 LA PROGRAMMATION (CONDITIONS, BOUCLES...)

Tout d'abord, il faut clarifier un point important : Python est un langage très simple à appréhender mais il a tout de même un défaut : sa lenteur. L'utilisation d'une boucle en Python n'est pas un processus efficace en termes de rapidité de calcul, elle ne doit servir que dans les cas où le nombre d'itérations de la boucle reste faible. Par exemple, si vous avez des dizaines de milliers de documents à charger pour effectuer un traitement, on se rend très vite compte de la lourdeur d'un processus basé sur une boucle et on préfère bien souvent d'autres outils plus efficaces pour charger en groupes des données ou pour le faire de manière parallélisée.

1.1.1 Les conditions

Une condition en Python est très simple à mettre en œuvre, il s'agit d'un mot clé **if**. Comme mentionné précédemment, le langage Python est basé sur l'indentation de votre code. On utilisera pour cette indentation un décalage avec quatre espaces. Heureusement, des outils comme Spyder ou les Jupyter notebooks généreront automatiquement cette indentation. Voici notre première condition qui veut dire « si a est True alors afficher "c'est vrai" » :

[47]:
```
a=True
```

[48]:
```
if a is True :
    print("c'est vrai")
```

```
c'est vrai
```

Il n'y a pas de sortie de la condition, c'est l'indentation qui va permettre de la gérer. Généralement, on s'intéresse aussi au complément de cette condition, on utilisera pour cela else :

[49]:
```
if a is True :
    print("c'est vrai")
else :
    print("ce n'est pas vrai")
```

```
c'est vrai
```

On peut avoir un autre cas, si notre variable a n'est pas forcément un booléen, on utilise elif :

[50]:
```
if a is True :
    print("c'est vrai")
elif a is False :
    print("c'est faux")
else :
    print("ce n'est pas un booléen")
```

```
c'est vrai
```

Les opérateurs de comparaisons sont rassemblés dans le tableau suivant :

Les opérateurs de comparaisons sont assez souples sur les différents types. Ainsi, on a :

[51]:
```
# True est égal à 1
True == 1
```

[51]: True

[52]:
```
# False est égal à 0
False == 0
```

[52]: True

[53]:
```
# mais True n'est pas 1
True is 1
```

[53]: False

[54]:
```
True > False
```

[54]: True

[55]:
```
# l'ordre alphabétique prime
"Python" > "Java"
```

[55]: True

[56]:
```
"Java"< "C"
```

[56]:
```
False
```

1.2 Les boucles

Les boucles sont des éléments centraux de la plupart des langages de programmation. Python ne déroge pas à cette règle. Il faut néanmoins être très prudent avec un langage interprété tel que Python. En effet, le traitement des boucles est lent en Python et nous l'utiliserons dans le cadre de boucles à peu d'itérations. Nous éviterons de créer une boucle se répétant des milliers de fois sur les lignes d'un tableau de données. Cependant, nous pourrons utiliser une boucle sur les colonnes d'un tableau de données à quelques dizaines de colonnes.

1.2.1 La boucle for

La boucle en Python a un format un peu spécifique, il s'agit d'une boucle sur les éléments d'une structure. On écrira :

[57]:
```
for elem in [1, 2] :
    print(elem)
```

```
1
2
```

Ce morceau de code va vous permettre d'afficher 1 et 2. Ainsi l'itérateur de la boucle (elem dans notre cas) prend donc les valeurs des éléments de la structure se trouvant en seconde position (après le in). Ces éléments peuvent être dans différentes structures mais on préférera généralement des listes.

1.2.2 Les fonctions range, zip et enumerate

Ces trois fonctions sont des fonctions très utiles, elles permettent de créer des objets spécifiques qui pourront être utiles dans votre code pour vos boucles. La fonction range() permet de générer une suite de nombres, en partant d'un nombre donné ou de 0 par défaut et en allant jusqu'à un nombre non inclus :

[58]:
```
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

[59]:
```
print(list(range(2,5)))
```

```
[2, 3, 4]
```

[60]:
```
print(list(range(2,15,2)))
```

```
[2, 4, 6, 8, 10, 12, 14]
```

On voit ici que l'objet range créé peut être facilement transformé en liste avec list().

Dans une boucle, cela donne :

[61]:
```
for i in range(11) :
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
```

Les fonctions zip et enumerate sont aussi des fonctions utiles dans les boucles et elles utilisent des listes. La fonction enumerate() renvoie l'indice et l'élément d'une liste. Si nous prenons notre liste de pays utilisée plus tôt :

[62]:
```
for i, a in enumerate(liste_pays) :
    print(i, a)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-62-670c53fdeda3> in <module>
----> 1 for i, a in enumerate(liste_pays) :
      2     print(i, a)

NameError: name 'liste_pays' is not defined
```

La fonction zip va permettre de coupler de nombreuses listes et d'itérer simultanément sur les éléments de ces listes. Si par exemple, on désire incrémenter simultanément des jours et des météos, on pourra utiliser :

[63]:
```
for jour, meteo in zip(["lundi","mardi"],["beau","mauvais"]) :
    print("%s, il fera %s" %(jour.capitalize(), meteo))
```

```
Lundi, il fera beau
Mardi, il fera mauvais
```

Dans ce code, on utilise zip() pour prendre une paire de valeurs à chaque itération de la boucle. La seconde partie est une manipulation sur les chaînes de caractères. Si l'une des listes est plus longue que l'autre, la boucle s'arrêtera dès qu'elle arrivera au bout de l'une d'elles.

On pourra coupler enumerate et zip dans un seul code, par exemple :

[64]:
```
for i, (jour, meteo) in enumerate(zip(["lundi","mardi"],["beau","mauvais"])) :
    print("%i : %s, il fera %s" %(i, jour.capitalize(), meteo))
```

```
0 : Lundi, il fera beau
1 : Mardi, il fera mauvais
```

1.2.3 La boucle while

Python vous permet aussi d'utiliser une boucle while() qui est moins utilisée et ressemble beaucoup à la boucle while que l'on peut croiser dans d'autres langages. Pour sortir de cette boucle, on pourra utiliser un break avec une condition. Attention, il faut bien incrémenter l'indice dans la boucle, au risque de se trouver dans un cas de boucle infinie.

On pourra avoir :

[65]:
```
val_stop  = 50
```

[66]:
```
i=1
while i<100 :
    i+=1
    if i>val_stop :
        break
print(i)
```

```
51
```

1.2.4 Exercices à Faire

Exercice 1 Ecrivez une programme qui demande à l'utilisateur d'entrer des notes d'élèves. Si l'utilisateur entre une valeur négative, le programme s'arrête. En revanche, pour chaque note saisie, le programme construit progressivement une liste. Après chaque entrée d'une nouvelle note (et donc à chaque itération de la boucle), il affiche le nombre de notes entrées, la note la plus élevée, la note la plus basse, la moyenne de toutes les notes.

Astuce: Pour créer une liste vide, il suffit de taper la commande liste = [].

[ ]:
```

```