Python-Initiation

October 15, 2021

1 Initiation à Python

1.1 Le mode interactif

Python propose un mode **interactif**, c'est-à-dire un mode dans lequel python est aux ordres. L'utilisateur tape des choses et l'environnement python essaie de les interpréter immédiatement. Cela diffère du mode **script** dans lequel l'utilisateur écrit un ensemble d'instructions (un programme) puis donne ce programme à python pour l'exécuter. Selon les environnements l'utilisateur est invité à taper quelque chose par un affichage (l'affichage du *prompt* ou de l'*invite*), pour python de base c'est `>>>`, pour pylab c'est `In[n°]`.

1.2 Les valeurs

Python permet de manipuler différentes valeurs. Pour désigner les différentes natures des valeurs, on parle de **type**. Les deux types de base sont les nombres et les chaînes de caractères.

1.2.1 Les nombres

Voici un nombre :

[1]:
```
88
```

[1]:
```
88
```

L'utilisateur tape un nombre (suivi du caractère d'entrée) et python l'interprète puis affiche ce qu'il comprend. Ici on a tapé deux caracatères représentant 88, python décode le tout et comprend qu'il s'agit du nombre 88 et l'affiche.

Python peut comprendre de nombreuses choses et en particulier les expressions arithmétiques (les calculs numériques à l'aide d'opérateurs mathématiques comme la division, etc). Ainsi :

[2]:
```
88*2
```

[2]:
```
176
```

Ou encore :

[3]:
```
(5+6)*9
```

[3]:
```
99
```

[4]:
```
9/2
```

[4]:
```
4.5
```

1.2.2 Les chaînes de caractères

On appelle chaîne de caractères toute suite de caractères (quelconques). Pour délimiter le début et la fin d'une chaîne de caractères on peut utiliser soit `"`, soit `'` :

[5]:
```
"Bonjour les amis"
```

[5]:
```
'Bonjour les amis'
```

Python affiche `'Bonjour les amis'` pour indiquer qu'il a bien compris qu'il s'agit d'une chaîne de caractères. L'affiche des `'` en début et en fin indique que python a **typé** l'entrée comme chaîne de caractères. Ainsi on distingue nombres et chaînes…

[6]:
```
'Bonjour les amis'
```

[6]:
```
'Bonjour les amis'
```

Attention il faut être cohérent, le marqueur de début doit être le même que le marqeur de fin, sinon on a droit à une erreur (notre première erreur de python :

[7]:
```
"bonjour'
```

```
  File "/var/folders/qs/fncjygp159x86yqvb7dmkqvc0000gn/T/ipykernel_71628/
↪208921406.py", line 1
    "bonjour'
            ^
SyntaxError: EOL while scanning string literal
```

Les erreurs possibles sont très nombreuses mais le message correspondant ressemble toujours au précédent. Il n'est pas toujours facile de les comprendre, mais ici il est indiqué tout de même que c'est une erreur de syntaxe, le caractère `^` marque l'endroit où python comprend que c'est une erreur (attention ce n'est pas forcément de là que vient vraiment l'erreur) et le message indique que la fin de ligne (`EOL`, *end of line*) a été détectée alors qu'on était dans une chaîne de caractères…

Note : les langages de programmations ont une syntaxe. On ne la décrit généralement pas intégralement (mais il existe des grammaires formelles pour cela…), car elle est relativement naturelle et suit généralement quelque principes simples. Dans notre cas : les chaînes sont délimitées par une paire soit de `"` soit de `'`.

1.3 Les variables

Dans les exemples précédents, python n'a rien retenu… Une fois l'interprétation de la ligne effectuée et le résultat effectué, il ne subsiste plus rien. On a souvent besoin de stocker des valeurs afin de

les retrouver plus tard. C'est l'objet des variables que de le permettre. Voici comment créer une variable :

```
[ ]: nombreDePersonnes = 34
```

Python ne renvoie rien en retour mais il se passe tout de même des choses dans la machine. En particulier, une variable est créée (donc un espace de stockage) dans laquelle est rangée la valeur 34. L'opération de rangement/stockage est représentée par le symbole `=` dit opérateur d'**affectation**. Python est silencieux car tout a bien fonctionné. Pour pouvoir retrouver plus tard, cette valeur on a utilisé un **identificateur** de variable, ici c'est `nombreDePersonnes`.

Un identificateur de variable doit respecter certaines contraintes : ne contenir que des caractères alphabétiques (majuscules ou minuscules), des chiffres et eventuellement le caractère `_`. Attention, il ne doit jamais commencer par un chiffre. Par exemple `machine`est un identificateur valide ou `toto_23` mais pas `2toto`.

La (dernière) valeur stockée peut être retrouvée à tout instant en utilisant simplement l'identificateur :

```
[ ]: nombreDePersonnes
```

```
[ ]: nombreDePersonnes / 2
```

```
[ ]: print("Il y a", nombreDePersonnes, "personnes")
```

1.4 La fonction `print()`

On a vu qu'une expression produit, dans le mode interactif, en sortie un affichage de valeur mais sous un forme particulière (les chaînes sont typées par exemple). Évidemment on peut avoir besoin dans un programme d'afficher de choses à l'utilisateur, c'est l'objet de la fonction `print` :

```
[ ]: print(88)
```

```
[ ]: print("Bonjour")
```

```
[ ]: print("Il y a",nombreDePersonnes,"personnes dans la salle.")
```

L'instruction précédente correspond à un **appel de fonction**. La fonction print agit sur ses **arguments**, c'est-à-dire les valeurs transmises à la fonction. La fonction `print` prend ces arguments un par un et affiche leur contenu, chacun séparé de l'autre par un espace, donc d'abord le contenu de la chaîne `"Il y a"`, suivi d'un espace, suivi du contenu de la variable `nombreDePersonnes`, suivi d'un espace, suivi du contenu de la châine `personnes dans la salle.`". Ainsi le message affiché est dépendant de la valeur de la variable `nombreDePersonnes` à l'instant de l'appel.

1.5 Retour sur les valeurs et les variables...

Il existe d'autres types que les nombres et les chaînes de caractères. Python possède aussi des types permettant de représenter des collections. Parmi ceux-ci on trouve la **liste**. C'est une collection de valeurs rangées en séquence. Par exemple :

[ ]:
```
["Haha",666,"Hihi"]
```

est une liste syntaxiquement délimitée par `[` et `]` et dont les valeurs sont séparées par des `,`. Cette liste contient trois éléments, le premier est une chaîne `'Haha'`, le second un nombre `666`, le troisième une chaîne `'Hihi'`. Bien entendu la possibilité de stocker plusieurs éléments en séquence dans une variable doit permettre de lire chaque élément individuellement, ainsi :

[ ]:
```
maListe = ["Joe", "Jack", "William", "Averell"]
print(maListe[0])
```

permet d'obtenir la valeur du premier élément de la liste. À la suite immédiate de l'identificateur désignant la liste est ajouté l'indexation, à l'aide de l'opérateur d'indexation `[]` contenant l'indice de l'élément que l'on souhaite extraire de la liste.

> Si je veux le premier pourquoi est-il indiqué 0 ? Les informaticiens ont l'habitude de numéroter les séquences en partant de 0, la logique étant que l'indice correspond, partant du début de la liste, au nombre d'emplacements à sauter pour arriver à celui qu'on souhaite; donc partant du début il n'y a en aucun à « sauter » pour arriver au premier…et pour arriver au second il faut en sauter 1, etc.

[ ]:
```
print(maListe[3])
```

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

Évidemment la dernière tentative conduit à un échec puisqu'on tente d'accéder au septième élément d'une liste qui n'en contient que 4… Python est sévère sur ce genre de choses.

1.6 Les méthodes

Comment rajoute t'on des éléments à une liste ? Par exemple, supposons que *Ma Dalton* donne naissance à un cinquième fils comment pourrait-on le rajouter à la liste précédente ? **Attention**, on ne souhaite pas fabriquer une nouvelle liste on souhaite juste modifier la liste existante.

Il suffit de comprendre que certaines fonctions sont contextualisées, c'est à dire que non seulement aux types correspondent des valeurs mais aussi des opérations qui leurs sont applicables. Ainsi l'ajout d'un élément à une liste est une opération de la liste elle-même. Ainsi :

[ ]:
```
maListe
maListe.append("Emmett")
maListe
```

La syntaxe permet d'utiliser la fonction (on doit dire **la méthode**) `append` de la liste `maListe` de sorte qu'elle recoive la valeur `'Emmett'` à y insérer en toute fin. Il existe de nombreuses autres opérations applicables aux listes (vous en découvrirez quelques-uns plus tard sans doute).

À ce propos, il est temps de dire qu'il existe une documentation de Python et qu'il est nécessaire de consulter à un moment où à un autre. Réportez-vous à https://docs.python.org/fr/3/ par exemple. **Attention** la lecture en est ardue, malgré tout il est possible de la lire à plusieurs niveaux, par exemple en ne s'attachant pas à essayer de comprendre tous les détails, parfois l'existence d'un nom de méthode suffit à comprendre l'effet produit. Ainsi et par exemple saurez-vous deviner ce que fait `maListe.clear()` ? Ou :

```
[ ]: maListe.remove("Emmett")
     maListe
```

Il n'est pas possible d'échapper un jour ou l'autre à la consultation de la documentation, il faut donc apprendre à s'y référer rapidement, quitte à demander de l'aide pour comprendre; votre expertise Python ne fera qu'augmenter, lentement au début et rapidement ensuite.

Les chaînes de caractères aussi possèdent des méthodes. Par exemple :

```
[ ]: titre = "initiation à python"
     print(titre)
     titreCap = titre.capitalize()
     print(titreCap)
```

Une autre méthode utile (parmi de très nombreuses) des chaînes est :

```
[ ]: resultat = titre.split()
     print(resultat)
```

qui renvoie la liste des mots d'une chaîne donnée… `split()` découpe une chaîne en mots. La documentation vous permettra de découvrir que `split` peut recevoir des paramètres permettant d'obtenir des découpages de toute sorte.

1.7 La fonction `len()`

Si l'on souhaite déterminer le nombre d'éléments d'une séquence on peut utiliser la fonction `len`:

```
[ ]: len(maListe)
```

Cette fonction peut aussi être employée sur une chaîne de caractères :

```
[ ]: print(maListe[0],"s'écrit avec",len(maListe[0]),"caractères")
```

1.8 Les itérations

Python est un langage de programmation, cela signifie qu'on peut donc demander à faire faire certaines choses particulières. Par exemple, répéter des tâches. Parmi les variantes de ces répétitions il y a l'itération.

L'itération (dite boucle `for`) permet de répéter une série d'actions sur divers éléments. Songez par exemple à un publipostage à l'ancienne (*via* la poste) : on veut pour toutes les adresses du carnet, rédiger un mail, le timbrer et le poster. On veut donc répéter les mêmes actions un certain nombre de fois. C'est la boucle `for` qui est adéquate. Ainsi :

```
[ ]: for enfant in maListe:
         print("Ma Dalton a pour fils",enfant)
```

permet d'obtenir 4 messages personnalisés pour chacun des fils…

La syntaxe de la boucle `for` est particulière :

`for enfant in Maliste:`

exprime le fait que l'on souhaite itérer sur l'ensemble des éléments de la liste (du premier au dernier). `enfant` est une variable créée pour la boucle `for` et qui capte les valeurs des éléents de la liste, l'une après l'autre. Autrement dit, au cours du temps, `enfant` reçoit d'abord la valeur du premier élément, puis une fois le corps de la boucle exécuté (voir ci-après) prend la valeur du second, etc. **Notez** que les éléments `for`, `in` et `:` sont des mots-clés et donc réservés à cet usage.

Le **corps** de la boucle est ici réduit à un simple affichage :

```
print("Ma Dalton a pour fils",enfant)
```

donc à chaque fois que la variable prend une nouvelle valeur depuis la liste, cette instruction est exécutée. Et on affiche les différentes messages successivement. **Attention** d'un point de vue syntaxique, l'indication que cette instruction fait partie du corps de la boucle est représentée par un décalage vers la droite. Ainsi :

[ ]:
```
for enfant in maListe:
    print("Ma Dalton a pour fils",enfant)
print("C'est fini.")
```

indique que seul le premier `print` fait partie de la boucle, le second étant calé sur la même colonne que le `for`, il n'en fait donc pas partie. Et donc, après avoir terminé la boucle `for`, on exécute ce qui suit, c'est-à-dire l'affiche de `C'est fini.`. Et :

[ ]:
```
for enfant in maListe:
    print("------")
    print("Ma Dalton a pour fils",enfant)
print("C'est fini.")
```

montre que les deux premiers `print` font partie de la boucle…

> **Exercice** Transformez le programme qui affiche le nom des enfants de Ma Dalton de sorte que l'affichage produise des sorties comme `Ma Dalton a pour fils Joe, et le prénom Joe s'écrit avec 3 lettres` (et ce pour tous les fils).

Les itérations aussi sont applicables aux chaînes de caractères (et oui après tout une chaîne de caractères c'est une séquence de caractères). Ainsi on peut écrire :

[ ]:
```
print("Le nom",maListe[0],"s'écrit avec")
for c in maListe[0]:
    print("le caractère",c)
print("et puis c'est tout...")
```

> **Exercice** Sauriez-vous écrire un programme qui décompose le prénom de chacun des fils de Ma Dalton ? Par exemple :
>
> ```
> Le nom Joe s'écrit avec
> le caractère J
> le caractère o
> le caractère e
> Le nom Jack s'écrit avec
> ...
> ```

Si vous avez résolu l'exercice, c'est que vous avez découvert qu'une itération peut contenir dans son corps une autre itération… Mais c'est naturel! En effet, si on vous demande de faire le ménage dans toutes le bâtiment Olympe de Gouges, vous devez : pour chaque étage, pour chaque pièce de l'étage, pour chaque bureau de la pièce, nettoyer le bureau :

```
for etage in etages:
    for piece in etage:
        for bureau in piece:
            nettoyer(bureau)
```

1.9 Exploiter le contenu d'un fichier

Un programme, en général, prend ses entrées de l'extérieur, par exemple depuis un fichier (si on souhaite compter le nombre de mots contenus dans un fichier). L'utilisation des fichiers contenant du texte est assez simple en Python car il suffit de savoir qu'un fichier texte est essentiellement une séquence de lignes (alors on va utiliser `for` ? Oui!). Une autre chose, qui peut surprendre, est la façon d'utiliser un fichier. Un fichier est une resource externe (au programe qui souhaite l'utiliser) par conséquent il est nécessaire de demander l'autorisation d'utiliser la resource et lorsqu'on a terminé de l'utiliser de la rendre (un peu comme si on souhaite entrer dans une salle fermée, on demande la clé, si on l'obtient on utiliser la salle et lorsqu'on a fini on rend la clé. La clé représente la possibilité d'utiliser la resource. C'est pareil avec toutes les resources, donc les fichiers. L'opération d'acquisition de la resource est `open` (c'est pour ça qu'on dit *ouvrir un fichier*) et pour relâcher la resource c'est `close`(on dit *fermer le fichier*). Le schema général est donc :

```
fichier = open(nom_du_fichier,mode_d'ouverture)
... # utilisation de la resource
fichier.close()
```

Une autre écriture, simplifiée et plus sûre est :

```
with open(nom_du_fichier,mode_d'ouverture) as fichier:
    ... # utilisation de la resource
... # suite
```

qui a l'avantage de libérer la resource automatiquement!

Dans la suite on supposera qu'il existe un fichier correspondant au nom utilisé et qui contient le texte suivant :

```
Bonjour les amis,

Est-ce qu'on peut se voir dimanche prochain ?
Je préparerais un barbecue, vous êtes les bienvenus!

Amicalement, Jean-Baptiste
```

[ ]:
```
fichier = open('/tmp/toto.txt','r')
for ligne in fichier:
    print("Voici une ligne:",ligne)
fichier.close()
```

1.9.1 La fonction `open`

Analysons d'abord la ligne `fichier = open('/tmp/toto.txt','r')`. Il s'agit de demander l'ouverture d'un fichier dont le nom est `/tmp/toto.txt`. Dans cet exemple, le nom est un nom pour les systèmes de la famille Unix/Linux. Pour un nom Windows on aurait utilisé quelque chose comme `C:/Users/bidule/truc/toto.txt`. On ira pas plus loin dans la description des chemins d'accès aux fichiers (consultez vos camarades informaticiens pour plus de détails). L'autre point important est le `'r'` donné comme second argument à la fonction, il s'agit de préciser quelle utilisation nous souhaitons faire du fichier, on peut vouloir simplement lire, simplement écrire, lire et écrire, etc. Ici `'r'` signifie *read* (lecture). Devinez ce qu'il faut préciser pour écrire ?

1.9.2 Le fichier comme séquence

Comme on l'avait indiqué, un fichier texte est une séquence de lignes, alors l'emploi de la boucle `for` permet d'itérer sur toutes les lignes du fichier. Et pour chaque ligne on affiche un message.

1.9.3 La méthode `close`

Une fois la resource employée, si l'on souhaite la rendre (ce qui est fairplay) on doit utiliser la méthode `close()`, point à partir duquel la resource est rendue et devenue inemployable. Si on souhaite manipuler le fichier à nouveau, il faut l'ouvrir à nouveau.

1.9.4 Un exemple

Supposons que l'on souhaite afficher séparément tous les mots d'un fichier, alors le programme pourrait être :

[ ]:
```
with open('/tmp/toto.txt','r') as flec:
    for ligne in flec:
        liste_mots = ligne.split()
        for w in liste_mots:
            print("J'ai trouvé le mot",w)
```

Exercice Sauriez-vous écrire un programme qui décompose le fichier lettre par lettre ?

```
J'ai trouvé le caractère B
J'ai trouvé le caractère o
J'ai trouvé le caractère n
J'ai trouvé le caractère j
...
```