Les notions de base de Python

Lien vers la documentation exhaustive: https://docs.python.org/3/

Les données

Les commentaires en Python se font à l'aide de **#**.

Documentation à consulter.

Les 5 types de base

- Chaînes de caractères
- Entiers
- Réels
- Complexes (la notation utilise j à la place de i)
- Booléens

Afin de connaître le type d'un objet on peut utiliser la fonction type:

```
>>> type(42)
<type 'int'>
```

On peut changer le type d'un objet avec des fonctions telles que:

- str()
- float()
- int()
- complex()
- bool()

Variables

Une variable est un nom auquel est associée un objet. Pour associer un nom à un objet on utilise l'opérateur d'affectation = tel que:

```
nom = objet
```

A gauche de l'opérateur = se trouve un nom (en gros, quelque chose ne pouvant commencer par un nombre) et à droite un objet.

Un nom n'est PAS une chaîne de caractères. Une chaîne de caractère est un objet alors qu'un nom n'est qu'un alias vers un objet.

Il est important de comprendre que l'opérateur d'affectation = n'est pas symétrique. A gauche des noms.

Attardons nous un moment sur ces notions car elles seront cruciales plus tard pour appréhender les possibilités offertes par les objets.

Considérons le programme suivant:

```
x = 1
y = 1
x = y
```

La figure montre le résultat après chaque instruction. On voit qu'un même objet peut parfaitement avoir plusieurs noms. Cependant, à un nom correspond un unique objet. Les objets qui n'ont plus de noms sont supprimés à intervalles réguliers (c'est ce qu'on appelle le garbage collector) puisque l'on ne peut plus y accéder.

Le mécanisme décrit précédemment (remplacement des noms par les objets référencés avant exécution de l'instruction) montre que l'on peut affecter plusieurs noms en même temps, comme le montre l'exemple suivant:

```
i = 2
j = 3
i, j = j, i
```

Les structures de données

Les listes

https://docs.python.org/3.4/tutorial/datastructures.html#more-on-lists

Création Directe

On peut créer une liste directement:

- Soit en créant une liste vide puis en ajoutant des éléments un à un.

```
l = []
```

- Soit en créant la liste déjà pré-remplie.

```
l = [1, 2, True, "Hello World"]
```

. Cette liste contient 4 éléments et est **indexée à 0**.

La fonction len() permet d'obtenir la longueur de la liste. Sur le dernier exemple,

```
len(l)
```

rend 4. On peut alors accéder aux éléments de la liste à l'aide d'un indice variant entre 0 et len(l) - 1. Ainsi avec

```
l[3]
```

on obtient la chaîne de caractère "Hello World".

Création à l'aide de range()

La commande range permet de créer des listes de nombres.


https://docs.python.org/3/library/stdtypes.html#range

Ajout, suppression d'éléments d'une liste


Regarder la documentation indiquée au début de la partie.

Copie d'une sous-liste

On peut copier une partie d'une liste. Pour **copier la liste l à partir de l'indice i jusqu'à l'indice j avec un pas de k** par exemple:

```
l[i:j:k]
```

Il n'est pas nécessaire de renseigner tous les champs.


Essayez `l[::3]` ou `l[1::5]` etc... (il faut bien évidemment des listes assez longues).

Les dictionnaires

Documentation:
https://docs.python.org/3.4/library/stdtypes.html?highlight=dict#mapping-types-dict

Un dictionnaire (ou tableau associatif, voir http://fr.wikipedia.org/wiki/Tableau_associatif) permet d'associer des clés à des valeurs, ces clés pouvant être des chaines de caractères ou des nombres. C'est en gros comme une 'liste' où l'on remplace les indices par à peu près que l'on veut.

```
d = {} #on crée un dictionnaire vide
d["quarante deux"] = 42 #on associe à la clé "quarante deux" la valeur 42
d[3.14] = "pi" #on associe à la clé 3.14 la valeur "pi"
print("quarante deux" in d)
print(42 in d)
for cle in d:
    print("cle :", cle, "valeur :", d[cle])
```

Un dictionnaire n'est pas ordonné

Les ensembles : set

https://docs.python.org/3.4/library/stdtypes.html?highlight=dict#set

Un ensemble permet de garder des données en mémoire de manière non indexée. Contrairement aux listes, où l'on rangeait les éléments dans des cases distinctes, on ne peut **pas** accéder aux éléments d'un ensemble d avec d[i].

Notion d'objets mutables

Les objets que nous avons rencontrés sont mutables, c'est à dire que lorsque on crée une liste `l = [1, 2, 3]`, il est toujours possible de changer la valeur d'un indice, ou d'ajouter un élément.

Cela n'est toutefois pas possible avec les `tuples` par exemple.

https://docs.python.org/3.4/library/stdtypes.html?highlight=dict#tuples

Un tuple peut se créer de la manière suivante:

```
t = (1, 2, 3)
```

Essayez maintenant des commandes telles que:

```
t[0] = 10
t.append(42)
```

Cela nous renvoie alors des erreurs.

Pour ajouter un élément, il faut créer un autre tuple:

```
t2 = t + (1, )
```

Le **frozenset** est un set (ensemble), mais cette fois non mutable.

Structures de contrôle

Comparaisons

https://docs.python.org/3.4/library/stdtypes.html#comparisons

Blocs

On a souvent besoin de n'exécuter un certain bout de code que si une comparaison particulière est vraie(True). Ce bout de code est appelé **bloc**

Tous les blocs sont définis de la même manière:

1. Ce qui va identifier le bloc pour son exécution (une condition, son nombre d'exécution, son nom).
2. Les instructions le constituant.

Pour séparer les blocs les un des autres, et savoir ce qui le défini , le langage Python utilise l'indentation(4 espaces): un bloc est donc une suite d'instructions ayant la même indentation.

```
question = "A quelle heure on mange ?"
print("Question :")
print(question)
print("Reponse :")
if question == "la vie, l'univers et le reste":
    reponse_partielle = 1 + 2 + 3 + 4 + 5 + 6
    reponse = reponse_partielle * 2
    print(reponse)
else:
    print("je ne sais pas.")
```

Ce code contient 3 blocs: le bloc principal, puis les 2 blocs de conditions (respectivement `if` et `else`).

L'indentation est donc **primordiale** en python.

Nous allons utiliser la convention classique : chaque bloc sera indenté de 4 espaces supplémentaire par rapport au bloc parent. Pour ne pas passer son temps à compter les espaces à ajouter pour chaque bloc, on pourra utiliser la la **touche tabulation** en modifiant son fonctionnement (remplacement du caractère tabulation par des espaces, cela est déjà prédéfini avec PyCharm) et est disponible dans tout bon éditeur.

En python, toute ligne définissant un nouveau bloc doit être terminée par le caractère **:**

Conditions si/sinon si/sinon (if/elif/else)

https://docs.python.org/3/reference/compound_stmts.html#the-if-statement

Exemple:

```
x = 7
if x < 0:
    print("Strictement negatif")
elif x == 0:
    print("vaut Zero")
elif x % 2 == 0:
    print("pair strictement positif")
else:
    print("impair strictement positif")
```

Il est à noter que elif et else sont optionnels.

Boucle while

https://docs.python.org/3/reference/compound_stmts.html#the-while-statement

```
b = 6
while b > 0:
    print(b)
    b = b - 1
```

On peut aussi inclure des blocs dans des blocs comme le montre le programme suivant :

```
b = 6
while b > 0:
```

```
    print(b)
    b = b - 1
    if b == 2:
        print("b vaut 2")
```

Boucle for

Les itérateurs

Pour faire simple, les littérateurs sont des objets qui permettent de créer des suites de données. Prenons un exemple connu: range()

range permet de créer des itérateurs : range(10) est un itérateur qui va renvoyer les valeurs de 0 à 9.

Pour utiliser `for`, il faut un itérateur tel que: `for x in mon_iterateur` est la syntaxe.

Exemple:

```
mon_iterateur = range(10)
for y in mon_iterateur:
    print(x)
```

Essayez ce code et comprenez le : les itérateurs sont de puissants objets python.

Vous pouvez créer votre propre itérateur à l'aide de l'instruction `yield`

```
def mon_iterateur(valeur):
    for x in range(valeur):
        yield valeur * x

for x in mon_iterateur(5):
    print(x)
```

Ce qui va s'afficher sera:

```
0
5
10
15
20
```

On peut également boucler sur une liste, qui est un **objet itérable**:

```
l = ["Jet fuel", "can't", "melt", "steel beams"]
for mot in l:
    print(mot)
```

Méthodes, fonctions et modules

Les fonctions

Motivations

https://docs.python.org/3/reference/compound_stmts.html#function-definitions

Il n'est jamais bon de copier/coller un bout de programme qui se répète plusieurs fois (corriger un problème dans ce bout de code reviendrait à le corriger autant de fois qu'il a été dupliqué...). Il est de plus souvent utile de séparer les éléments logiques d'un programme en unités autonomes, ceci rend le programme plus facile à relire.

Pour cela, on utilise des `fonctions`.

Une fonction est un bloc (revoir si nécessaire) auquel on donne un nom (le nom de la fonction) qui peut être exécuté lorsqu'on l'invoque par son nom.

La partie de programme suivant défini une fonction:

```
def bonjour():
    print("Salutations")
```

La première ligne est la définition du bloc fonction. Il contient:

- un mot clé spécial précisant que l'on s'apprête à définir une fonction: `def`
- le nom de la fonction. Ici `bonjour`
- des parenthèses qui pourront contenir des paramètres (on verra ça plus tard)
- le **:** qui indique que la ligne d'après va commencer le bloc proprement dit

Ensuite vient le bloc fonction en lui même qui ne contient qu'une seule ligne.

Si on exécute le bloc précédent, il ne se passe rien. En effet on n'a fait que définir la fonction. Pour l'utiliser, ajoutez

```
bonjour()
```

à la suite du bloc.

Une **fonction** s'utilise toujours en faisant suivre son nom d'une parenthèse contenant ses paramètres séparés par une virgule (notre fonction n'a pour l'instant pas de paramètres). Donner juste son nom ne suffit pas à l'invoquer.

Paramètres d'une fonction

```
def plus_moins(nombre):
    if nombre > 42:
        print("Supérieur à 42")
    else:
        print("Inférieur à 42")
```

Cette fonction nécessite donc un paramètre pour être invoquée. Testez alors

```
plus_moins(17)
```

La variable nombre sera associée à l'objet entier de valeur 17 dans la fonction. La variable nombre n'existe que dans la fonction.

Les *paramètres* d'une fonction sont des **NOMS** de variables qui ne seront connus qu'à l'intérieur de la fonction. À l'exécution de la fonction, le nom de chaque paramètre est associé à l'objet correspondant.

Retour d'une fonction

Toute fonction rend une valeur. On utilise le mot-clef `return` suivi de la valeur à rendre pour cela. Le fonction suivante rend le double de la valeur de l'objet passé en paramètre:

```
def double(valeur):
    x = valeur * 2
    return x
```

Il ne sert à rien de mettre des instructions après une instruction `return` car dès qu'une fonction exécute cette instruction, elle s'arrête en rendant l'objet en paramètre. Le retour d'une fonction est pratique pour calculer des chose et peut ainsi être affecté à une variable.

Ainsi, avec la fonction double précédemment définie, testez:

```
x = double(21)
print(x)
```

Le code précédent exécute la fonction de nom `double` avec comme paramètre un entier de valeur 21. La fonction commence par associer à une variable nommée `valeur` l'objet passé en paramètre (ici un entier de valeur 21), puis crée une variable de nom `x` à laquelle est associée un entier de valeur 42 et enfin se termine en retournant comme valeur l'objet de nom `x`. Les variables `valeur` et `x` définies à l'intérieur de la fonction sont ensuite effacés (pas les objets, seulement les noms).

Cette valeur retournée est utilisée par la commande `print` pour être affichée à l'écran.

Les noms de paramètres d'une fonction et les variables déclarée à l'intérieur de la

fonction n'existent qu'à l'intérieur de celle-ci. En dehors de ce blocs, ces variables n'existent plus.

Fonctions v.s. méthodes

Python vient avec de nombreuses fonctions que l'on peut utiliser. Vous en connaissez déjà comme `range`, `len`, ou encore `type`.

Ne confondez pas fonctions et méthodes. Une fonction s'exécute toute seule alors qu'une méthode a besoin d'un objet sur lequel elle s'applique (celui avant le .). Vous pouvez voir ça comme un 1er paramètre indispensable à l'exécution d'une méthode. Considérez le micro-programme suivant:

```
ma_liste = range(5)
ma_liste.append(10)
```

La première ligne exécute une *fonction* (range) avec un paramètre qui rend une liste. La seconde instruction est une *méthode* (append) qui s'applique à l'objet de nom `ma_liste` et qui a un paramètre (ici un entier valant 10).

Le point un peu délicat est que certaines méthodes ne rendent rien et modifient l'objet sur lequel elle est appliquée, c'est le cas de la méthode `append`, `insert` ou encore `reverse`, alors que d'autres rendent des objets, c'est le cas de `index` par exemple.

```
ma_liste = range(5)
ma_liste.insert(2, "coucou")
un_indice = ma_liste.index("coucou")
print(un_indice)
print(ma_liste[un_indice])
```

Visibilité d'un objet

Les noms des objets sont accessibles à l'intérieur du `bloc unitaire` dans lequel ils sont déclarés ainsi que dans les blocs unitaires contenus dans celui-ci. Les blocs unitaires sont~:

- les fonctions,
- les modules (nous verrons cela),
- les classes (que nous ne verrons pas).

Les variables définies dans une fonction cachent les variables définis dans des blocs supérieurs. Ainsi, le code suivant imprime 42 puisque la variable x déclarée dans le bloc unitaire de la fonction n'existe plus dans son bloc parent. La variable x valant 42 est masquée dans la fonction par une nouvelle variable de nom x valant 24.

```
def f():
    x = 24

x = 42
f()
```

```
print(x)
```

De la même manière, que donne le programme suivant ?:

```
def f(parametre):
    parametre = 24

f(2)
print(parametre)
```

Les noms déclarés dans une fonction, y compris ses paramètres, restent dans la fonction.

Récursion

Modification d'objets dans une fonction

Dans un programme récursif, on a souvent besoin de modifier le même objet plusieurs fois. Même si la fonction récursive ne rend rien. Pour cela, on doit modifier les objets passés en paramètres. Pour comprendre comment cela marche, considérez la fonction suivante~:

```
def ajoute_max(liste_en_parametre):
    maximum_liste = max(liste_en_parametre)
    liste_en_parametre.append(maximum_liste)
```

Cette fonction ajoute à la fin d'une liste passée en paramètre son maximum (au passage, on a apprit une nouvelle fonction, `max`. regardons le programme suivant qui utilise cette fonction:

```
x = list(range(1, 6, 2))
ajoute_max(x)
print(x)
```

La figure suivante montre ce qu'il s'est passé dans le monde des noms et des objets. Il reste un objet sans nom après l'exécution de la fonction (un entier valant 9), il est détruit. On a pu ainsi modifier un objet sans utiliser de retour de fonction. C'est une technique puissante mais à n'utiliser qu'à bon essient.

Modules

Un *module* (aussi appelé *bibliothèque* ou *library*) est un ensemble de fonctions utiles, utilisable dans de nombreux programmes. Plutôt que de refaire à chaque fois ces fonctions ou (c'est pire) de les copier/coller dans chaque programme on les *importe* directement pour les utiliser.

Il existe de nombreux modules, réalisant une foultitude d'opérations. Avant de se

mettre à coder quelque chose, commencez toujours par vérifier *google* est votre ami) s'il n'existe pas un module tout fait. Vous gagnerez du temps.

Python fournit déjà de nombreux modules, les plus courants sont décrits là : https://docs.python.org/3.4/library/index.html.

Pour utiliser un module, il faut commencer par l'importer avec la commande `import`. Par exemple avec le module `math`.

- Importation directe du module. On mets le nom complet avant chaque appel :

```
import math
pi_sur_deux = math.pi / 2 #PI est défini dans le module math
x = math.cos(pi_sur_deux) #on utilise la fonction cosinus du module math
```

- Importation d'une méthode particulière. Ceci peut être dangereux si des fonctions différentes possèdent le même nom.

```
from math import cos, pi #importation directe de cos et de pi
x = cos(pi / 2)
```

- Importation de toutes les fonctions du modules.

```
from math import *
y = log(e)
```

Cette page vous en explique plus en prenant le module `random` comme exemple.

Retour sur les objets

Comme on l'a vu les objets sont partout en python, qu'ils soient int, str, float, ou même des fonctions. Si vous avec bien compris l'exemple de la récursion et du la modification d'un objet passé en paramètre, alors vous vous demandez peut-être "pourquoi ne pas envoyer une fonction en paramètre d'une autre fonction ?"

Eh bien cela est tout à fait possible, exemple:

```
def produit(x, y):
    return x * y

def calcul(fonction, z):
    return z + fonction(2, 17)

print(calcul(produit, 8)) #On envoie l'objet associé au nom 'produit' à la
fonction 'calcul'
```

Ce programme affichera alors 42 ! Essayez-le pour vous en persuader.

Les fichiers : lecture, écriture

Consulter https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

Lecture

Pour lire le fichier ligne par ligne

```
f = open('fichier.txt', 'r')

for ligne in f:
    print(l)
```

On peut aussi compter le nombre de mots dans le texte, par exemple:

```
f = open('fichier.txt', 'r')
nombre_mots = 0

for ligne in f:
    ligne = ligne.strip("\n")
    mots = ligne.split(' ')

    nombre_mots += len(mots)

print(nombre_mots)
f.close()
```

Essayez ce code avec un fichier .txt de votre choix.

Au lieu de juste compter les mots, vous pouvez même les garder dans un ensemble !

```
f = open('fichier.txt', 'r')
ensemble_mots = set()

for ligne in f:
    ligne = ligne.strip("\n")
    mots = ligne.split(' ')

    ensemble_mots.update(mots)  # Regarder les opérations sur les ensembles
!
```

```
print(ensemble_mots)
print(len(ensemble_mots))
f.close()
```

Ce code n'est pas parfait, par exemple, s'il y a des mots en fin de phrase par exemple, ils seront comptés en double à cause du '.'. Ici, ce qui est compté est la longueur de l'ensemble, ie le nombre de mots différents utilisés.

Écriture

Pour lire-écrire, ouvrez le fichier avec 'r+' au lieu de 'r'. Pour l'écriture seule, 'w'.

Ouvrir en écriture un fichier existant, **l'efface**. Pour ajouter des choses à la fin d'un fichier on utilise 'a' (pour append)

Utilisez ensuite la méthode write():

```
f = open('fichier', 'w')
f.write('For the night is dark and full of terrors')
f.close()
```

Bonnes pratiques

Revenons un moment sur les bonnes pratiques, et notamment de la PEP8.

PEP8 https://www.python.org/dev/peps/pep-0008/

Voici quelques conventions de bonne pratique à mettre en place:

- Les indentations se font avec 4 espaces par niveau.
- Les imports se font librairie par librairie: www.python.org/dev/peps/pep-0008/#imports
- Pour les espaces blancs: https://www.python.org/dev/peps/pep-0008/#whitespace-in-expressions-and-statements
- Nommer variables et fonctions: https://www.python.org/dev/peps/pep-0008/#descriptive-naming-styles

Il faut se souvenir qu'un nom doit être **clair et informatif** cela permet une relecture aisée !

Rédacteurs

- Augustin Agbo-Kpati
- François Brucker

- Pascal Préa