Leçon 1 : Notions de variables, d'objets et de typage dynamique

Nous allons découvrir les notions d'objet, de variable et de typage dynamique. En Python, tout est un objet. Pour manipuler les objets dans Python, il faut leur donner un nom par l'intermédiaire de variables. Les variables référencent les objets.

La notion d'objet

Dans un programme informatique, **un objet** est un morceau de code qui va contenir des données. Mais il va également contenir un ensemble de mécanismes qui permettent de manipuler ces données qu'on appelle **méthodes**.

Les objets ont tous **un type**. Le type est le comportement par défaut qui va être défini pour les objets. Par conséquent, le type permet de définir les données et les méthodes qui vont être associées à cet objet.

Exemple : L'analogie d'une chaîne de montage de voitures. Lorsqu'on a une usine, l'usine va construire des voitures et c'est cette usine qui va définir un ensemble de comportements que toutes les voitures qui vont sortir de la chaine de montage vont avoir. Par exemple, la puissance du moteur, le fait que la voiture va avoir des clignotants, des accélérateurs, va être déterminé par la chaine de montage. On peut donc dire que la chaine de montage détermine le type de voiture qui va sortir de cette chaine et que la voiture est l'objet qui sort de cette chaine de montage.

Un objet chaine de caractère :

=> 'spam'

Une fois créé avec l'interpréteur Python, on voit que l'objet a des données associées : le mot spam. Et cet objet a également un ensemble de méthodes. Comme par exemple la méthode **()**. D'où viennent les méthodes de la chaine de caractère ? En fait toutes les méthodes de la chaine de caractère viennent grâce à son type : le type « chaine de caractère ».

On peut appeler les méthodes sur ces objets. Il suffit d'utiliser l'annotation **point** :

=> 'spam'.upper()
La fonction upper() met spam en majuscule.

Nommer / Référencer les objets

Pour référencer un objet, il faut affecter un objet à un nom de variable avec une notation que l'on appelle **la notation d'affectation**.

=> note = 1

Un nom de variable en python peut être défini en lettres minuscules, en majuscules, avec les entiers de 0 à 9 et le caractère _ (underscore). Un nom de variable peut commencer par une lettre, par un underscore mais pas par un chiffre.

base OK

_id OK

6id NON

Un nom de variable prend en compte la casse : ID ≠ id. Il est important de donner des noms explicites car cela participe à la documentation automatique du code.

Le typage dynamique

L'espace de variable, s'appelle un espace de nommage.

Pour a = 3, Python va réaliser 3 opérations :

1. Python crée l'entier 3 dans l'espace des objets
2. Python crée une variable a dans l'espace des variables
3. Python crée une référence entre la variable a et l'entier 3

Si je crée une autre variable a = 'spam', Python fait les mêmes opérations mais il déréférence en plus l'objet 3 pour que la variable a référence désormais l'objet 'spam'.

Le tapage dynamique veut dire qu'en Python, le type n'est pas lié à la variable qui référence l'objet mais à l'objet. Python est un langage à tapage fort. Ce qui veut dire que le tapage est lié aux objets et que l'objet va garder le même type durant toute l'exécution du programme. Par contre la variable peut référence des objets qui vont être de type différent en cours d'exécution.

La commande : **del a** permet de supprimer la variable a de l'espace des variables. Si l'objet n'a plus de référence, un mécanisme en python qui s'appelle **mécanisme de garbage collector** libère la mémoire de l'ordinateur une fois que les objets ne sont plus référencés.

Notebooks de la leçon

Le signe **==** permet de tester si deux variables ont la même valeur.

Pour a = 1, b = 2 et c = 2 :

- a == b renvoie FALSE
- a == a renvoie TRUE
- b == c renvoie TRUE

En général, les noms de variable simple sont uniquement en minuscules. Les majuscules sont réservées en principe pour d'autres sortes de variables comme les noms de classe.

ATTENTION ! Il est déconseillé d'utiliser _**variable**_ qui sont réservés au langage.

En Python 3, il est possible d'utiliser des caractères Unicode dans les identificateurs (accents, alphabet grec). Mais tous les caractères unicode ne sont pas permis.

Il est impossible d'utiliser les mots clés du langage (ci-dessous) pour nommer des variables.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	**nonlocal**	while
assert	del	global	not	with
async	elif	if	or	yield

Nous avons indiqué **en gras** les nouveautés **par rapport à Python 2** (sachant que réciproquement `exec` et `print` ont perdu leur statut de mot-clé depuis Python 2, ce sont maintenant des fonctions).

La fonction **type()** est une fonction built-in, c'est-à-dire prédéfinie dans Python, qui donne le type d'une variable ou d'un objet.

La fonction **isinstance()** permet de savoir si un objet est bien d'un type donné.

=> isinstance(23, int) renvoie **TRUE**.

Leçon 2 : Les types numériques

L'objectif de cette leçon est de passer en revue les 4 types numériques qui existent en Python.

int -> entier

float -> nombre décimaux / réels

complex -> nombres complexes

bool -> les booléens (qui sont un sous ensemble des **int**)

Le type int

Pour créer un entier dans Python, il suffit de taper un entier dans l'interpréteur.

=> 1

Mais tel quel, on ne peut pas l'appeler puisqu'il n'est pas référencé. On va donc affecter l'objet 1 au nom de variable i.

=> i = 1

On peut vérifier le type avec la fonction built-in **type()**. On peut multiplier, diviser, soustraire les entiers. On peut aussi réaffecter une variable avec le résultat d'une opération.

=> i = i + 5

=> print(i) renvoie 6

Les entiers en Python sont des objets de précision illimitée. Python n'a aucun problème à faire des opérations sur des entiers extrêmement grands.

Le type Float

=> f = 4.3

La virgule se représente par un point. Les float en Python ont une précision limité. Ils sont généralement codés sur 15 chiffres significatifs et encodés sur 53 bits. Cela peut dépendre de la plateforme sur laquelle ont fait tourner notre interpréteur Python.

Le type complexe

=> c = 4 + 5j
*La constante complexe que nous notons **i** en français, se note **j** en Python*.

Un nombre complexe, c'est 2 nombres Float mis l'un à côté de l'autre.

On peut additionner n'importe quel type numérique. Python se charge de faire la conversion pour nous. Lors de la conversion, on peut avoir une perte de précision.

=> i (int) **+ f** (float) renvoie un float (moins précis qu'un entier)

=> i (int) **+ f** (float) **+ c** (complex) renvoie un nombre complexe avec là encore perte de préicsion au niveau de l'entier.

On peut convertir des nombres à l'aide de fonctions built-in :

- **int()** convertit en entier (l'opération s'appelle **la troncation**)
- **float()** convertit en float
- **complex()** convertit en complex
- **str()** convertir en chaine de caractère

En Python, il est possible de faire des additions, des soustractions, des multiplications, des divisions (qui ont la caractéristique d'être des divisions naturelles). Ainsi pour les divisions, si l'on souhaite obtenir **le quotient** il faut utiliser un **//** (**3//6**), pour obtenir **le reste** il faut utiliser un **%** (**3%6**). Élever un nombre à une puissance se fait avec ****** (**3**2 = 3^2**). **abs()** renvoie la valeur absolue d'un nombre.

Le type booléen

Le booléen n'est pas un nombre. Il est représenté soit par **TRUE**, soit par **FALSE**. Ce type booléen est cependant un sous-ensemble des entiers. Il est d'ailleurs codé comme un entier. **FALSE** étant l'entier 0 et **TRUE** étant l'entier 1.

Notebooks de la leçon

La bibliothèque mathématique de Python :

Pour les valeurs spéciales comme **π**, il est possible d'utiliser les valeurs prédéfinies par la bibliothèque mathématique de Python.

```
from math import e, pi
```

Et ainsi imprimer les racines troisièmes de l'unité par la formule :

$r_n = e^{2i\pi\frac{n}{3}}$, pour $n \in \{0, 1, 2\}$

```
n = 0
print("n=", n, "racine = ", e**((2.j*pi*n)/3))
n = 1
print("n=", n, "racine = ", e**((2.j*pi*n)/3))
n = 2
print("n=", n, "racine = ", e**((2.j*pi*n)/3))
```

```
n= 0 racine =  (1+0j)
n= 1 racine =  (-0.4999999999999998+0.8660254037844387j)
n= 2 racine =  (-0.5000000000000004-0.8660254037844384j)
```

Remarque : bien entendu il sera possible de faire ceci plus simplement lorsque nous aurons vu les boucles `for` .

La fonction input :

On peut demander à l'utilisateur d'entrer une valeur grâce à la fonction input.

=> reponse = input("quel est votre âge ?")

La réponse est par défaut de type chaine de caractère. Pour l'interpréteur en entier, il faudrait la convertir **int(reponse)**.

Lisibilité des longs nombres :

Pour améliorer la lisibilité des longs nombres, on peut utiliser l'underscore. Ainsi **123_465 = 123465**. L'underscore fonctionne avec les entiers et les flottants.

Entiers et systèmes de numération :

En Python, les entiers peuvent être entrés sous forme binaire, octale (base 8) ou encore hexadécimale (base 16). Pour d'autres bases, on peut utiliser la fonction de conversion **int()** avec un argument supplémentaire :

=> deux_cents = int ('3020',4)
*Le **4** désigne la base 4*

Affectations et opérations :

Il existe des opérateurs dérivés de l'affectation qui permettent de faire tout à la fois une opération et une affectation :

=> entier =10

=> entier += 2 renvoie 12 (10 + 2)

Même chose pour :

- **entier -= 4** (soustraction)
- **entier *= 2** (multiplication)
- **entier /= 2** (division)
- **entier **=10** (puissance)
- **entier %=5** (modulo)

Cette construction est disponible sur tous les modèles numériques. Il est ainsi possible d'additionner les listes entre elles :

=> liste = [0,3,5]

=> liste += ['a','b'] renvoie **[0,3,5,'a','b']**

Opérateurs de décalage :

var << n décale les bits vers la gauche (équivaut à **var*2^n**)

var >> n décale les bits vers la droite (équivaut à **var//2^n**)

Avec les opérateurs de décalage aussi la double opération est possible :

entier >>= 4 (**entier//2^4**)

entier <<= 4 (**entier*2^4**)

Flottants et précision :

Les flottants peuvent poser un problème de précision. Ainsi lorsque l'on écrit un nombre flottant en valeur décimale, la valeur utilisée en mémoire pour représenter ce nombre, parce que cette valeur est codée en binaire, ne représente pas toujours exactement le nombre entré.
Ainsi, **0.3 - 0.1 == 0.2** renvoie **FALSE**.

Une solution consiste à penser en terme de **nombres rationnels**. Alors qu'il n'est pas possible d'écrire 3/10 en base 2, on peut représenter exactement ces nombres dès lors qu'on les représente comme des fractions et qu'on les encode avec deux nombres entiers. Python fournit en standard le module **fractions**.

=> from fractions import Fraction

=> Fraction(3,10) - Fraction(1,10) == Fraction (2,10) renvoie **TRUE**

Cela peut aussi s'écrire :

=> Fraction('0.3') - Fraction('0.1') == Fraction ('2/10') renvoie également **TRUE**

Si les nombres manipulés ne sont pas rationnels, il est alors possible d'utiliser le module standard **decimal**.

=> from decimal import Decimal

=> Decimal('0.3') - Decimal('0.1') == Decimal('0.2') renvoie **TRUE**.

BONUS : Numération et bases de numération *http://www.courstechinfo.be/MathInfo/sommaire.html*

Une base de numération est un nombre dont on utilise les puissances successives pour former d'autres nombres plus importants. Ainsi, en base 10, les puissances successives sont Un (1=100), Dix (10=101), Cent (100=102), Mille (1000=103), Dix mille (10.000=104), etc.

Il existe différentes bases de comptage :

Base 60	Système sexagésimal utilisé en Mésopotamie. Il nous en reste 60 minutes, 60 secondes. Le nombre 60 a de nombreux diviseurs : 2, 3, 4, 5, 6, 10, 12, 15 et 30
Base 20	Le système vigésimal aurait été utilisé par nos ancêtres gaulois et était commun au moyen âge, il nous reste le "quatre-vingts". Quatre-vingt-dix, Soixante quinze se basent sur des multiples de 20.
Base 12	Système duodécimal pour compter les mois, les heures et les œufs par douzaines
Base 10	Celle que nous utilisons tous les jours (maintenant que nous portons des chaussures et que nous ne pouvons plus compter aussi sur nos orteils)
Base 5	Système quinaire que l'on retrouve partiellement dans la numération romaine et pour les calculs avec certains Japonais (système biquinaire)
Base 2	Incontournable en informatique. Elle vient du fait que les ordinateurs sont construits à partir de composants qui comme les contacts électriques n'ont que deux états possibles : ouvert ou fermé, bloqué ou passant, 0 ou 1.
Bases 8 et 16	Fort proches du binaire les bases hexadécimale et octale sont plus concises et plus facile à transcrire pour nous « humain » que le binaire fait de longues chaînes de 0 et de 1

La numération de position :

Prenons le nombre 1975 écrit en base 10 comme nous en avons l'habitude.

La valeur que l'on attribue à chaque chiffre dépend du chiffre en lui-même (la valeur qu'il représente quand il est seul) mais aussi, de sa position.

Ainsi, si on prend le nombre 1975 (écrit en base 10)
le 5 vaut 5 unités = 5 x 1
le 7 représente des dizaines, il vaut 7 x 10
le 9 qui suit représente des centaines, il vaut 9 x 100
le 1 compte pour un millier = 1 x 1000

Le chiffre le plus à droite représente des unités. En décimal, les chiffres suivants représentent les dizaines, puis les centaines etc. Nous numéroterons les positions en allant de droite à gauche. Ainsi les unités auront toujours la même et première position, la position 0, quelle que soit la taille du nombre.

3	2	1	0
1	9	7	5

La règle qui permet de déterminer le poids d'un chiffre est la suivante :

$$\text{Poids d'un chiffre} = \text{base}^{\text{position}}$$

Ainsi en base 10 :

En position 0 se trouvent les unités. Leur poids est 10^0

En position 1 se trouvent les dizaines dont le poids est 10^1

En position 2 : les centaines dont le poids est 10^2

En position 3 : les milliers dont le poids est 10^3

...

En position n : les milliers dont le poids est 10^n

Positions	3	2	1	0
Chiffres	**1**	**9**	**7**	**5**
Base Position = **Poids**	**10** 3 = **1000**	**10** 2 = **100**	**10** 1 = **10**	**10** 0 = **1**
Valeur de chaque chiffre	**1** * 1000	**9** * 100	**7** * 10	**5** * 1
	1000	**900**	**70**	**5**
Valeur totale	**1000 + 900 + 70 + 5 = 1975**			

*D'une manière plus théorique, on peut dire que la valeur d'un nombre **N** représenté par **n** chiffres en base **B** est la valeur numérique d'un polynôme du n-1 ième degré où **B** est la base et dont les coefficients sont entiers et inférieurs à **B***

$$N = c_{n-1} B^{n-1} + ... + c_i B^i + ... + c_2 B^2 + c_1 B + c_0$$

$$= \sum_{i=0}^{i=n-1} c_i B^i$$

Exemple : En base 10, B=10 et les coefficients C_{n-1}, C_{n-2}, C_i, , C_1, C_0 ont tous une valeur inférieure à 10.

La suite de ces coefficients $c_{n-1} c_{n-2} ... c_1 c_0$ n'est autre que la suite des chiffres qui forment le nombre.

La valeur d'un nombre est la somme des valeurs de chaque chiffres multipliés par leurs poids respectifs.

1011011(2)

= 1 x 2^6 + 0 x 2^5 + 1 x 2^4 + 1 x 2^3 + 0 x 2^2 + 1 x 2^1 + 1 x 2^0

= 64 + 0 + 16 + 8 + 0 + 2 + 1

= 91

175(8)

= 1 x 8^2 + 7 x 8^1 + 5 x 8^0

= 1 x 64 + 7 x 8 + 5 x 1

= 125

Numération binaire :

La numération binaire est une numération en base 2. En binaire il n'y a donc que deux chiffres 0 et 1. On remarque qu'en base 2, le chiffre 2 n'existe pas, tout comme le chiffre 10 n'existe pas en base 10.

Exemple : **10110 en binaire** → **Poids** de chaque bit = **2** position					
Positions :	4	3	2	1	0
Chiffres binaires	**1**	**0**	**1**	**1**	**0**
Poids	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Valeur de chaque chiffre	1 * 16	0 * 8	1 * 4	1 * 2	0 * 1
	16	**0**	**4**	**2**	**0**
Valeur totale (comptée en décimal)	**16 + 4 + 2 = 22**				

Bit, Byte, Octet :

Les codes binaires sont incontournables en informatique car l'information la plus élémentaire y est **le bit**. Ce mot « bit » est formé par la fusion des mots **Binary digit**. Ce qui en français se traduit par : Chiffre binaire. Un mot de 8 bits est appelé **Octet** (en français) et **Byte** (en anglais).

Les mots de 8, 16, 32 ou 64 bits sont courants. Ecrits en binaire, ils sont plus lisibles si on laisse un espace entre les groupes de quatre bits comme ceci : 0100 0001. Un groupe de 4 bits est parfois appelé **Quartet** ou **nibble** mais ces termes sont peu utilisés.

Numération hexadécimale :

Il faut 16 chiffes pour écrire les nombres en base 16. Aux 10 chiffres du système décimal (0 à 9) ajoutons les 6 caractères A, B, C, D, E et F pour représenter ce que nous considèrerons ici comme étant les "chiffres" de 10 à 15.

Exemple : **1A2F hexadécimal** → **Poids** de chaque chiffre = **16** position				
Positions :	3	2	1	0
Chiffres binaires	**1**	**A**	**2**	**F**
Poids	$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
Valeur de chaque chiffre	1 × 4096	10 × 256	2 × 16	15 × 1
	4096	**2560**	**32**	**15**
Valeur totale (comptée en décimal)	**4096 + 2560 + 32 + 15 = 6703**			

Pourquoi utiliser l'hexadécimal ?

Les codes hexadécimaux sont bien pratiques en informatique. Ils représentent les codes binaires de manière compacte et nous évitent de devoir lire de longues enfilades de 0 et de 1 qui conviennent mieux aux ordinateurs qu'aux humains. Un groupe de quatre bits permet de former 16 combinaisons différentes. On peut faire correspondre un chiffre hexadécimal à chacune de ces combinaisons de 4 bits. L'hexadécimal est en quelque sorte du binaire condensé. Le code hexadécimal 1A2F est bien plus lisible que 0001 1010 0010 1111 en binaire.

Tableau de conversion décimal / binaire / hexadécimal :

Décimal	Binaire	Hexa
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Méthode de conversion d'un nombre entier en base quelconque :

1° Convertir $1830_{(10)}$ en binaire ⇒ divisions successives par 2

1830 : 2 = 915	reste **0**	0×2^0	= 0 unité
915 : 2 = 457	reste **1**	1×2^1	
457 : 2 = 228	reste **1**	1×2^2	
228 : 2 = 114	reste **0**	0×2^3	
114 : 2 = 57	reste **0**	0×2^4	
57 : 2 = 28	reste **1**	1×2^5	
28 : 2 = 14	reste **0**	0×2^6	
14 : 2 = 7	reste **0**	0×2^7	
7 : 2 = 3	reste **1**	1×2^8	
3 : 2 = 1	reste **1**	1×2^9	
1 : 2 = 0	reste **1**	1×2^{10}	
0	C'est fini, il ne reste plus rien à diviser		

Le résultat est : **111 0010 0110**$_{(2)}$

2° Convertir $1830_{(10)}$ en hexadécimal ⇒ divisions successives par 16

1830 : 16 = 114	reste **6**	6×16^0 = six unités
114 : 16 = 7	reste **2**	2×16^1 = 2 seizaines
7 : 16 = 0	reste **7**	$7 \times 16^2 = 7 \times 256$
0	C'est fini, il ne reste plus rien à diviser	

Le résultat est $726_{(16)}$ ce qui concorde bien avec la valeur trouvée en binaire **111 0010 0110**$_{(2)}$

Leçon 3 : Codage, jeux de caractères et Unicode

Dans cette leçon, nous allons parler des notions fondamentales de codage, décodage, jeux de caractère et unicode.

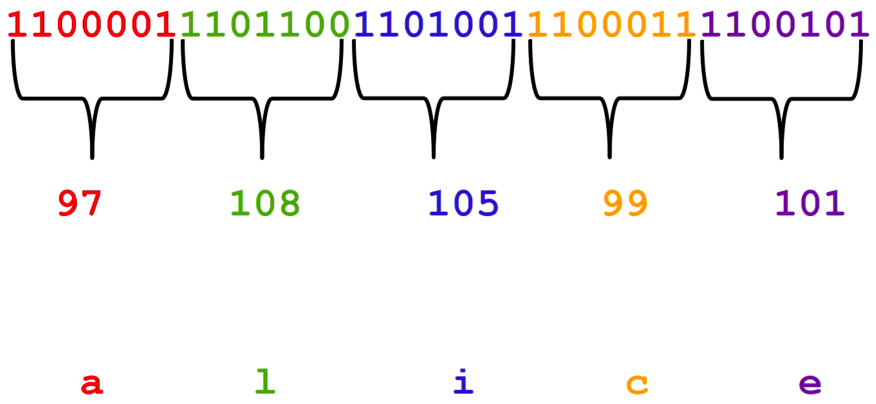
Codage et jeux de caractères

Prenons une chaine de caractère : JEFAISUNMOOCSURPYTHON. Il faut quelques secondes pour discerner les mots qui forment la phrase : JE FAIS UN MOOC SUR PYTHON. Notre cerveau a fait une opération que l'on appelle **une opération de décodage**. Il a pris une suite de lettres et il a été capable d'identifier la suite de mots contenus dans la suite de lettres.

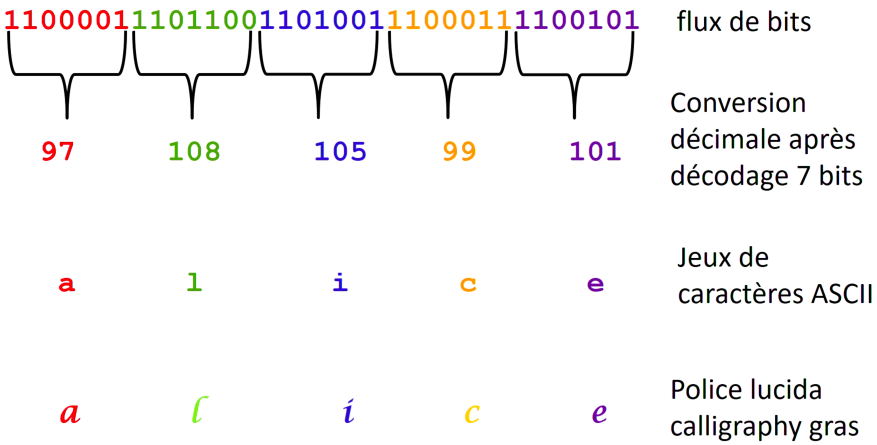
En informatique, nous n'avons pas des lettres. Nous manipulons des bits. La question que l'on peut se poser c'est comment passer d'**un flux de bits** à la notion de lettre. On fait une opération de décodage. Le décodage consiste à définir une convention qui va dire : je découpe mon flux de bits en blocs d'une certaine taille.

Nous allons ici prendre des blocs de 7 bits. C'est ce que l'on appelle le codage **ASCII**. Nous découpons notre flux de bits en blocs de 7 bits et nous regardons à quel nombre correspondent ces blocs de 7 bits. Maintenant que nous avons nos blocs, il faut être capable de dire à quelle lettre correspond chaque bloc.

Nous utilisons pour ça ce qu'on appelle **un jeu de caractères**. Un jeu de caractère c'est une table qui va donner un code qui correspond à un caractère. Nous prenons le jeu de caractères ASCII dans lequel 97(10) correspond à la lettre a, 108(10) à l, etc.



Une fois qu'on a déterminés, les caractères restent des notions abstraites. Ce ne sont pas encore des dessins que l'on va être capable d'afficher sur notre écran. Pour être capable d'afficher un dessin sur un écran, il faut utiliser **une police de caractère**. Ici, nous représentons les différentes lettres du prénom alice avec la police *Lucida calligraphie gras*. La police de caractère définit un glyphe pour chaque caractère, c'est-à-dire un dessin qui correspond à chaque caractère que l'on veut afficher.



En résumé, lorsqu'on lit un contenu sur Internet ou sur un disque dur, nous avons un flux de bits et nous faisons des opérations de décodage pour obtenir les caractères. De manière inverse,

lorsque l'on écrit quelque chose sur un disque dur ou que l'on envoie des informations sur Internet, nous allons faire une opération de codage pour retransformer nos lettres en flux de bits.

Le projet Unicode

Nous avons utilisé un codage ASCII qui est sur 7 bits. Or sur 7 bits, on ne peut coder que 128 caractères différents. Nous voyons bien que nous ne pouvons pas coder tous les caractères du monde sur simplement 128 caractères. Nous devons donc utiliser des codages que l'on appelle **des codages étendus**. Il existe des codages ASCII étendus qui sont codés sur 8 bits. Sur 8 bits, on peut coder 256 caractères. Encore une fois, ce n'est pas suffisant pour pouvoir coder tous les caractères du monde.

Or pendant des années, nous avons utilisé un grand nombre de codages ASCII étendus et d'autres types de codage qui ont permis par exemple de coder les caractères utilisés en français ou les caractères utilisés en allemand. Cependant, ces jeux de caractères sont incompatibles. Ce qui a produit pendant des années les erreurs du type : recevoir des mails avec des caractères bizarres. Ce type d'erreur s'explique par le fait que notre ordinateur ne contient pas le jeu de caractères qui a été utilisé pour encoder le texte publié.

Pour résoudre ce problème, un projet a été démarré : **le projet Unicode**. L'objectif de ce projet est de coder l'intégralité des caractères du monde. Dans Unicode, nous avons actuellement plus de 120 000 caractères qui ont été codés. Ainsi, en utilisant Unicode, nous avons la certitude de pouvoir coder et décoder l'intégralité des caractères du monde entier. Unicode utilise différents types de codage : **UTF-8**, **UTF-16** et **UTF-32**. Ces encodages participent à un compromis entre compacité du codage et vitesse de décodage.

En pratique, nous utilisons régulièrement le codage UTF-8 qui a la caractéristique importante d'être totalement compatible avec le codage ASCII standard. Python est totalement compatible avec le codage Unicode. Mais il faut toujours contrôler l'encodage. Malgré l'arrivée d'Unicode, il existe encore de nombreux autres jeux de caractères utilisés dans d'anciens documents. De telle sorte, que si on ne contrôle pas finement notre codage et décodage, on ne pourra pas dire à Python quel type de jeu de caractère utiliser.

La règle est donc simple : lorsqu'on crée des documents ou lorsqu'on lit des documents, il faut toujours utiliser UTF-8, sauf s'il nous est dit explicitement d'utiliser un autre jeu de caractères.

Leçon 4 : Les chaînes de caractères

Dans ce cours, nous allons aborder les chaînes de caractères et notamment comment gérer l'encodage avec les chaînes de caractères.

Chaînes de caractères et méthodes

Pour créer une chaîne de caractère en Python, il faut l'entourer soit par des apostrophes ' ', soit par des guillemets " ". Les chaînes de caractères sont des objets immuables. Une fois qu'ils ont été créés, on ne peut plus les modifier.

Les chaînes de caractères contiennent également de nombreuses méthodes qui permettent de manipuler ces chaînes de caractères. Toutes les chaînes de caractères étant immuables, les fonctions qui manipulent les chaînes de caractères retournent un nouvel objet chaîne de caractères.

Python a l'avantage d'être un langage auto-documenté. Ça veut dire que l'on peut très facilement interroger l'aide Python depuis un interpréteur. Ainsi, par exemple, les chaînes de caractères étant de type **str**, nous pouvons appeler la documentation en tapant **str?**. ATTENTION : cette opération avec le **?** n'est possible que depuis un interpréteur ipython ou depuis un notebook.

On peut également accéder à l'intégralité des méthodes qui existent sur un objet particulier, en utilisant la fonction built-in **dir()**. Avec **dir(str)** on obtient la liste de toutes les méthodes associés aux chaînes de caractères.

La méthode title() :

=> 'un mooc sur python'.title()
renvoie 'Un Mooc Sur Python'

Cette méthode **title()** met la première lettre de chaque mot en majuscule.

La méthode replace() :

=> s = "le poulet c'est bon"
=> s.replace("poulet","spam")
renvoie "le spam c'est bon"

Comme on obtient une nouvelle chaîne de caractères, on peut réaffecter la nouvelle chaîne de caractères à la variable s.

=> s = s.replace("poulet","spam")
=> s
renvoie "le spam c'est bon"

La méthode isdecimal() :

Un certain nombre de méthodes de chaînes de caractères permettent également de faire des comparaisons ou de faire des tests. Ainsi, si on prend la chaîne **'123'** et que l'on souhaite la convertir en entier, on pourrait vouloir s'assurer que cette chaîne de caractères représente bien un nombre décimal.

=> "123".isdecimal()
renvoie **TRUE**

Formater une chaîne de caractères :

=> n = "sonia"
=> age = 30

Nous voulons écrire Sonia a 30 ans. Pour faire cela en Python, il faut utiliser la fonction **format()** qui permet de formater une chaîne de caractères en fonction de certaines variables.

=> "{} {}".format(n,age)
renvoie "sonia 30"

L'instruction **format()** substitue le premier argument n à la première {} et le deuxième argument age à la deuxième {}.

Depuis Python 3.6, il est possible de créer des **f-string** qui permettent de mettre directement les variables entre les {}.

=> f“{n} à {age} ans“

Le support d’Unicode en Python 3

ATTENTION! Le support natif d’Unicode n’existe qu’en Python 3 et pas en Python 2.

On peut taper :

=> “noël, été“
renvoie “noël, été“

Cette chaîne de caractère est parfaitement importée par Python. On peut aussi directement rentrer le caractère unicode avec l’annotation **\u**.

=> “\u03a6“
renvoie la lettre grecque phi en alphabet grec

Il peut arriver que Python ne renvoie pas le caractère attendu. Le problème vient alors de la police utilisée qui ne supporte pas le caractère demandé. Il faut alors aller dans l’interpréteur et choisir une police qui supporte l’alphabet recherché.

Les notions de codage et de décodage

Sur un ordinateur, on a un flux de bits que l’on peut avoir à lire. Lorsqu’on lit ce flux de bits, on doit décoder ce flux de bits pour obtenir les lettres. À l’inverse, lorsque l’on veut écrire ces lettres sur un disque dur ou sur Internet, on doit les encoder.

Cette opération de décodage/encodage est extrêmement simple en Python. Dès qu’on manipule des chaînes de caractères en Unicode, on utilise le type str. Dès qu’on manipule des flux de bits, on utilise le type bytes. Et Python permet de passer très facilement de l’un à l’autre.

Exemple pour **s = “un noël en été“**, si on veut encoder cette chaîne de caractères en bytes, il faut utiliser la fonction **encode()**

```
In [9]: s.encode('utf8')
Out[9]: b'un no\xc3\xabl en \xc3\xa9t\xc3\xa9'
```

On obtient un nouvel objet : une chaîne de caractères qui commence avec un b et ensuite qui contient des caractères particuliers. En fait, le type bytes va contenir uniquement des octets. Et Python nous offre la possibilité lorsque l’octet représente un caractère ASCII qui a un affichage sous forme de lettre ou de chiffre, d’afficher cette lettre ou ce chiffre tel quel. Cependant il ne s’agit que d’une facilité et Python va bien coder les bytes sous forme d’octets et non sous forme de caractères.

On peut ensuite décoder chaîne de caractère pour repasser au type str avec la fonction **decode()**.

=> en = s.encode(‘utf8’)
=> en.decode(‘utf8’)
renvoie ‘un noël en été’

On peut évidemment utiliser d’autres types d’encodages comme **latin1** pour encoder et décoder notre chaîne de caractère. Ce qui est très important, c’est de ne jamais mélanger les encodages et les décodages. C’est pour ça qu’il est recommandé de toujours utilisé UTF-8.

En résumé, les principaux problèmes qui existent aujourd’hui avec l’encodage et le décodage ne sont pas liés aux jeux de caractères puisqu’on a Unicode. Ils sont quasiment toujours liés au fait que la personne qui développe le code ne communique par le bon encodage utilisé.

C’est pour cela qu’il est recommandé de toujours utiliser UTF-8, de toujours exactement contrôler l’encodage, donc de bien encoder et décoder les chaînes de caractères. Et de toujours dire à la personne ou aux personnes qui vont utiliser notre code quel type d’encodage nous utilisons.

Leçon 5 : Les séquences

Dans cette leçon nous allons parler d'un ensemble de types qu'on appelle les séquences. Les séquences en Python regroupent notamment les **list**, les **tuple**, les **str** et les **bytes**.

Définition de séquence :

Une séquence en Python est un ensemble fini et ordonné d'éléments indicés de 0 à n-1 si on a n éléments.

Opérations de base sur une séquence "chaîne de caractère"

Commençons avec une chaîne de caractère : **s = 'egg, bacon'**. L'opération **s[0]** permet d'accéder au premier élément de la séquence, c'est-à-dire 'e'. **s[9]** permet d'accéder au dernier élément de la séquence, puisque la séquence fait 10 éléments.

Pour connaître le nombre d'éléments d'une séquence, il faut utiliser la fonction built-in **len()**. **len(s)** renvoie 10. Nous avons effectivement 10 éléments dans la séquence.

Toutes les séquences supportent également **le test d'appartenance**, qui est une opération très puissante en Python. **'egg' in s** veut dire est-ce que 'egg' est dans s. L'opération renvoie True. De même on peut faire **'egg' not in s** qui est **le test de non-appartenance** et qui retourne ici False.

Ensuite on peut faire de **la concaténation** de séquence. **s + ', and beans'** rajoute ', and beans' à notre séquence et renvoie donc 'egg, bacon, and, beans'. Cette concaténation produit une nouvelle chaîne de caractères puisqu'une chaîne de caractère n'est pas mutable.

L'opération **index()** permet de trouver la première occurence par exemple de la lettre g : **s.index('g')** renvoie 1 puisque la première occurence de la lettre g c'est la deuxième lettre à la place 1.

L'opération **count()** permet de compter le nombre d'éléments, ici le nombre de 'g' que l'on a dans la séquence. **s.count('g')** renvoie 2, puisqu'on a deux fois la lettre g.

On peut également appliquer les fonctions built-in **min()** et **max()** qui retournent le minimum et le maximum d'une séquence. **min(s)** renvoie ' ' et **max(s)** renvoie 'o'. Comme notre séquence est une chaîne de caractère, le min et le max utilisent l'ordre lexicographique : l'espace pour le minimum et la lettre o pour le maximum.

L'opération de **Shalow copy** permet de copier un certain nombre de fois un élément : **'x'*30** produit 30 fois 'x'. Nous verrons par la suite que cette opération produit en fait une shalow copy qui peut avoir des effets de bord lorsque la séquence multipliée n'est pas un immuable mais un objet de type mutable.

L'opération de slicing

L'opération de slicing est valable pour toutes les séquences (chaînes de caractères aussi bien que list). Pour commencer nous créons une chaîne de caractère **s = 'egg, bacon'**.

Une opération de sclicing se note : **s[0:3]** avec 0 la borne de gauche inclue et 3 la borne de droite exclue. **s[0:3]** va ainsi retourner tous les éléments qui vont de 0 inclus à 3 exclu, donc ici la chaîne de caractère de 0 à 2 : 'egg'.

Si on ne précise pas la borne de gauche **s[:3]** on va parcourir tous les éléments depuis le début jusqu'au 3 exclu, donc ici la chaîne de caractères de 0 à 2 : 'egg'. À l'inverse, si on ne précise pas la borne de droite, l'opération va retourner tous les éléments allant de la borne de gauche (inclue) jusqu'à la fin. Avec **s[5:]** on obtient la chaîne de caractères de 5 jusqu'à la fin, donc 'bacon'. Enfin, si on ne spécifie ni la borne de gauche, ni la borne de droite, l'opération retourne toute la chaîne de caractères. **s[:]** renvoie 'egg, bacon'. **ATTENTION!** Cette opération de slice ne va pas retourner la chaîne de caractère elle-même mais une copie de la chaîne de caractère. Nous verrons par la suite qu'il s'agit d'une opération dite de shalow copy.

Nous allons maintenant rajouter à notre slice **la notion de pas**. Dans un sclice, on peut indiquer une borne de gauche, une borne de droite et un pas : **s[0:10:2]**. Avec un pas de 2, l'opération parcourt les éléments 1 sur 2. **s[0:10:2]** renvoie donc 'eg ao'. **s[::2]** parcourt tous les éléments allant du début à la fin par pas de 2 et renvoie donc ici 'eg ao'.

Avec **s[100]**, on utilise un indice qui est en dehors de la chaîne de caractères. L'opération renvoie donc une erreur qu'on appelle **une exception**.

```
>>> s[100]
Traceback (most recent call last):
  File "<pyshell#109>", line 1, in
<module>
    s[100]
IndexError: string index out of range
```

L'exception comme toujours en Python est explicite. Elle s'appelle *IndexError* et elle écrit le message d'erreur *string index out of range*. Cela veut dire que l'indice est en dehors des indices couverts par la chaîne de caractères.

Par contre avec **s[5:100]**, on obtient tous les éléments qui vont depuis 5 (inclus) jusqu'à la fin de la chaîne de caractère : 'bacon'. Pourquoi n'avons-nous pas d'erreur ? Parce que le slice est un objet. Python prend donc tous les indices couverts par le slice, ici les indices allant de 5 à 100, puis il cherche l'intersection avec les indices disponibles dans l'objet chaîne de caractère. Et Python retourne uniquement les éléments qui sont à l'intersection. Ici l'intersection est non-nulle, c'est 'bacon'. Python retourne donc 'bacon'.

Avec **s[100:200]**, les indices représentés par le slice et les indices représentés par la chaîne de caractères n'ont pas d'intersection. Le slice retourne donc un objet de type chaîne de caractère mais qui est vide : ' '.

Avec **s[-10:-7]**, les indices négatifs sont numérotés à partir de la fin. C'est donc un moyen très commode d'accéder aux derniers éléments d'une séquence lorsque l'on connaît leur position à la fin de la séquence.

0	1	2	3	4	5	6	7	8	9
e	g	g	,		b	a	c	o	n
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

s[-10:-7] va retourner tous les éléments allant de -10 inclus jusqu'à -7 exclu, donc ici l'opération va aller de -10 à -8 : 'egg'. Les indices négatifs ne changent pas l'ordre de l'opération de slice qui continue de s'effectuer de gauche à droite. Avec **s[:-3]** l'opération retourne tous les éléments allant du début jusqu'à -3 : 'egg, ba'.

Avec **s[::-1]** le pas étant négatif, me slice va parcourir la séquence dans le sens inverse de droite à gauche. Cette notation est donc très souvent utilisée pour renverser une séquence. **s[::-1]** renvoie donc ici : 'nocab ,gge'. Avec **s[2:0:-1]** l'opération inverse la séquence et prend un sous-ensemble allant de droite à gauche. Donc allant de 2 inclus à 0 exclu, c'est-à-dire ici : 'gg'. Avec **s[2::-1]** la séquence est retournée de 2 jusqu'au début. C'est-à-dire ici : 'gge'.

Leçon 6 : Les listes

La liste représente un type extrêmement souple et puissant. Une liste est une séquence d'objets hétérogènes. Donc une liste peut comporter n'importe quel type d'objet. Mais il est important de comprendre que la liste ne stocke pas les objets. Elle ne stocke que des références vers les objets. Par conséquent, la taille de l'objet liste est indépendante du type d'objets qui sont référencés. Une liste peut augmenter en tailler, réduire. On peut l'écarter au milieu, rajouter des éléments à l'intérieur. On peut complètement la manipuler.

La liste est malléable parce que c'est un objet mutable. Cette notion de mutabilité est importante. Un objet mutable est un objet que l'on peut modifier en place. Cela veut dire qu'on peut le modifier là où il est stocké. L'avantage de cette mutabilité c'est qu'on n'a pas besoin de faire une copie de l'objet pour le modifier. C'est donc extrêmement efficace au niveau mémoire.

Liste et séquence, mêmes opérations

Une liste est une séquence. Par conséquent, toutes les opérations applicables aux séquences sont applicables aux listes. Ainsi du test d'appartenance, de la concaténation, de la fonction built-in len(), la fonction count(), etc. Toutes ces opérations sont disponibles pour toutes les séquences, donc pour les listes.

Prenons une liste mêlant une variable **i = 4**, une chaîne de caractère **'spam'**, un flottant **3.2** et un. booléen **True**. **A = [i,'spam',3.2,True]** Pour rappel, la liste ne stocke que des références vers les objets qu'elle renferme. Elle ne copie jamais les objets qu'elle référence.

Comme notre liste est une séquence, on peut accéder à chaque élément de la liste. Ainsi **a[0]** renvoie la valeur de **i** c'est-à-dire **4**. La liste étant mutable, on peut modifier ses éléments. On peut ainsi rentrer **a[0] = 6** et la liste contient désormais **[6,'spam',3.2,True]**. On peut également faire directement une opération sur un élément d'une liste. **a[0] = a[0] + 10** ajoute **10** au premier élément de la liste qui devient : **[16,'spam',3.2,True]**.

Opérations de slicing

On peut également réaliser des opérations de slicing sur une liste. **a[1:3]** prend tous les éléments de 1 inclus à 3 exclu et retourne donc **['spam',3.2]**. On peut aussi faire des opérations d'affectation sur des slice : **a[1:3] = [1,2,3]** renvoie **[16,1,2,3,True]**. L'opération sur un slice effectue deux opérations indépendantes : 1) d'abord avec **a[1:3]**, tous les éléments de 1 inclus à 3 exclu sont enlevés ; 2) puis tous les éléments de la séquence **[1,2,3]** sont insérés à la place des éléments qui ont été effacés.

Cette opération d'affectation sur un slice est un moyen très simple d'effacer des éléments dans une liste. Pour **a =[16,1,2,3,True]**, si on entre **a[1:3] = []**, les éléments entre **1** inclus et **3** exclu sont effacés. L'opération renvoie donc ici : **[16,3,True]**. On peut également utiliser l'instruction **del** pour enlever des éléments dans un slice. Avec **del a[1:2]**, l'élément à l'indice **1** est effacé. Pour **a = [16,3,True]** l'instruction renvoie **[16,True]**.

Opérations .append() et .extend()

L'opération **list.append()** ajoute un objet à la fin d'une liste. Avec **a = [16, True]**, si on entre **a.append('18')**, on obtient **a = [16,True,'18']**.

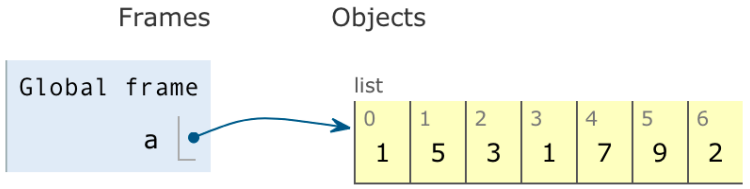
L'opération **list.extend()** prend une séquence et ajoute chaque élément de cette séquence à la fin de la liste. C'est l'équivalent de **list.append()** sur chaque élément de la séquence. Avec **a = [16,True,'18']**, si on entre **a.extend([1,2,3])**, on obtient **a = [16,True,'18',1,2,3]**.

Opération de tri .sort()

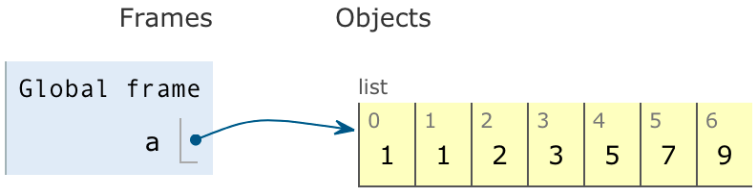
Pour une liste **a = [1,5,3,1,7,9,2]**, on peut appeler la méthode **.sort()** qui va trier les éléments de la liste. ATTENTION! **.sort()** fonctionne en place, c'est-à-dire que la liste est triée en place, sans faire de copie temporaire et la méthode **.sort()** ne retourne rien puisque l'objet a été trié en place. **a.sort()** renvoie **a = [1,2,3,5,7,9]**.

Il ne faut jamais faire d'opération d'affectation sur la méthode **.sort()** parce qu'elle va retourner l'objet **None**. Ainsi, si on fait **a = a.sort()**, **a** ne référencera pas l'objet trié mais simplement la valeur de retour de **.sort()** qui ne sert à rien, qui est juste l'objet **None**, donc l'objet vide.

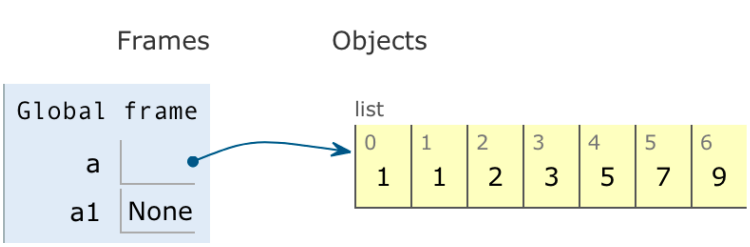
a = [1,2,3,5,7,9]



a.sort()



a1 = a.sort()



Transformer une chaîne de caractère en liste et vice versa

Il existe des opérations permettant de passer d'une chaîne de caractère à une liste et d'une liste à une chaîne de caractère. C'est une opération fréquemment utilisée pour accéder à des fichiers que l'on veut traiter avec Python.

Pour commencer, on crée une chaîne de caractères **s = 'spam egg beans'**. L'objectif est de séparer cette chaîne de caractère en colonnes. Pour cela, on utilise la fonction built-in **split()** qui est une fonction des chaînes de caractères. En tapant **a = s.split( )** on obtient **a = ['spam','egg','beans']**. On peut passer n'importe quel caractère de séparation à **.split()**. Par exemple, si nous avions eu des virgules, **.split()** aurait utilisé les virgules comme séparateur.

Une fois qu'on a notre chaîne de caractères, la liste étant mutable, on peut entrer **a[0] = a[0].upper()** qui renvoie **a = ['SPAM','egg','beans']**.

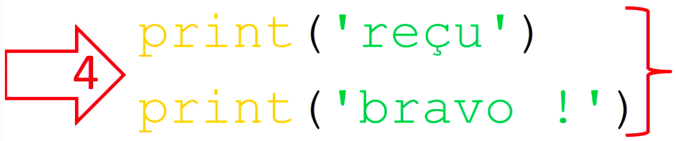
On peut retransformer la liste en chaîne de caractères avec la syntaxe : **" ".join(a)** qui renvoie **'SPAM egg beans'**. L'espace entre guillemets avant **.join()** désigne le séparateur que **.join()** va mettre entre chaque élément.

Leçon 7 : Introduction aux tests if et à la syntaxe

Dans cette leçon, nous allons parler de l'instruction **if else** qui permet de faire de l'exécution conditionnelle. C'est-à-dire qu'un morceau de code va s'exécuter en fonction du fait qu'un test soit vrai ou faux.

Prenons l'exemple d'un message à afficher en fonction d'une notation supérieure ou inférieure 10/20.

```
note = 8
if note > 10:
    print('reçu')
    print('bravo !')
else:
    print('recalé')
```



If est l'instruction. **note > 10** est l'expression. **print ('reçu')** et **print ('bravo !')** forment le bloc d'instructions. Le **:** est systématique avant un bloc d'instruction. Un bloc d'instructions est un ensemble d'instructions qui sont toutes indentées du même nombre de caractères vers la droite. La convention étant d'indenter tous les blocs d'instructions de 4 caractères vers la droite.

Si le test **if** est vrai, Python exécute les instructions du bloc d'instructions. Si le test est faux, on entre dans la clause **else** et Python exécute le bloc d'instructions dans la clause **else**. Là encore, le bloc d'instructions, ce sont toutes les instructions décalées de 4 caractères vers la droite.

Python est un langage conçu autour de cette notion de bloc d'instructions. Dans un certain nombre de langages, il n'y a pas cette notion de bloc d'instruction. Les blocs d'instruction sont délimités par exemple par des accolades, comme ci-dessous avec une syntaxe à la java.

```
note = 8;
if note > 10{
    print('reçu');
    print('bravo !');}
else
    {
    print('recalé');
     }
```

Toutes les instructions finissent par un point virgule pour déterminer la fin de l'instruction. Et les blocs d'instructions sont séparés par des accolades ouvrantes et des accolades fermantes. Cela constitue un problème connu en programmation. Puisque pour être capable de savoir où placer les accolades, on définit ce qu'on appelle des conventions de codage. La convention de codage n'a absolument aucun impact sur l'exécution du code. Or ces conventions de codage font partie d'écoles de programmation. Et certaines personnes préfèrent certaines conventions de codage.

Par exemple : mettre les accolades en fin de ligne et en début de ligne ou mettre les accolades alignées avec les instructions. En fait, lire un code avec une convention de codage qui n'est pas la notre rend extrêmement difficile l'interprétation ou la lecture de ce code.

En Python, on n'a pas ce problème puisque la convention de codage fait partie de la syntaxe. Si on ne respecte pas la convention de codage, on aura une erreur de syntaxe, le code ne s'exécutera donc pas. L'avantage c'est qu'avec Python, il n'y a qu'une seule manière de présenter le code.

En python pas de **;** . La fin d'une instruction c'est le retour chariot. Pas d'accolades. Les blocs d'instruction sont décalés de 4 caractères vers la droite. Seul reste le symbole **:** . On peut se demander pourquoi garder le **:** . En fait, on le garde parce qu'à l'issue de tests utilisateurs, on s'est rendu compte qu'il est plus facile de détecter les blocs d'instructions quand ils sont précédés d'un **:** .

Le fait que Python utilise l'indentation comme base de sa syntaxe n'a presque que des avantages. Ça permet d'avoir un code écrit toujours de la même manière, bien présenté, facile à lire et écrire.

L'inconvénient c'est que l'indentation ne supporte pas bien le copier-coller. Il faut donc toujours vérifier attentivement l'indentation quand on copie-colle du code sous Python. La convention de codage a par ailleurs tendance à faire des lignes un peu grandes. En Python, il est recommandé de ne pas dépasser 79 caractères sur une seule ligne. Python permet assez facilement de retourner à la ligne. Tout ce qui est entre parenthèses, entre crochets ou entre accolades supporte le retour chariot sans créer de problème de syntaxe.

Leçon 8 : Introduction aux boucles for et aux fonctions

Un principe fondateur de la programmation est la factorisation du code. La factorisation du code c'est ce qui permet de ne pas réécrire plusieurs fois un code qui fait la même chose. L'intérêt c'est de faciliter la maintenance et surtout la qualité du code. Dans cette leçon, seront abordées les notions de boucle **for** et de **fonction** qui sont deux techniques de factorisation de code.

Les boucles for

Nous voulons afficher les carrés des nombres allant de 0 à 9. On peut taper **print(1**2)**, puis **print(2**2)**, etc. C'est fastidieux. On peut automatiser cette tâche avec une boucle **for**. L'intérêt d'une boucle **for** c'est d'automatiser une tâche qui se répète.

Ci-contre : On a une boucle **for** qui parcourt tous les entiers de 0 jusqu'à 9. La variable **i** référence chacun de ces entiers. Et le **print(i**2)** affiche le carré de l'entier référencé par **i** au moment du tour de boucle. On peut mettre n'importe quelle séquence dans une boucle **for**, comme par exemple une liste :
for i in ['a',3.5,True]:
print(i2)**
renvoie **a**, **3.5** et **True**

```
In [4]: for i in range(10):
   ...:     print(i**2)
   ...:
0
1
4
9
16
25
36
49
64
81
```

Les fonctions

Prenons une liste **a = [1,2,5,8,9]**. On souhaite parcourir les carrés de ces entiers. On crée donc une boucle **for**.

for i in a:
print(i2)**
renvoie **1 4 25 64 81**

Prenons maintenant une autre liste **b = [3.6, 18, 12, 25]**. Pour parcourir les carrés de cette liste, on refait une boucle **for**.

for i in b:
print(i2)**
renvoie **12.96 324 144 625**

Les deux boucles fonctionnent de la même manière. Il serait donc utile de factoriser ce code. Une manière de le faire est d'utiliser une **fonction**. Une **fonction** est un morceau de code que l'on peut rappeler n'importe quand.

Ci-contre : L'instruction **def** définit une fonction à laquelle on donne un nom, ici **carre**. Entre parenthèses, dans la définition, on définit un argument, ici **a**. C'est l'argument que l'on va être capable de passer à la fonction lorsqu'on va l'appeler. Donc **carre(b)** fait passer la liste **b** à la fonction **carre**. Ce traitement se fait par référence. Il n'y a pas de copie d'objet. La fonction **carre()** exécute un **for i in** l'objet qui est passé par référence à la fonction.

```
In [10]: def carre(a):
    ...:     for i in a:
    ...:         print(i**2)
    ...:

In [11]: carre(a)
1
4
25
64
81

In [12]: carre(b)
12.96
324
144
625
```

En fait, dans une **fonction**, on fait très rarement des affichages avec **print()**. Ce qui est utile c'est plutôt de faire des retour avec l'instruction **return**.

Ci-contre : On cherche toujours à retourner le carré des nombres des listes **a** et **b**. Mais cette fois-ci, on veut que les résultats soient retournés sous forme de liste. On définit donc une liste **L =[]** et à chaque tour de la boucle on va exécuter **L.append(i**2)** et **return** retourne la liste **L** avec les nombres au carré. L'instruction **return** ne s'exécute que lorsque la boucle **for** a terminé tous ses tours.

```
In [14]: def carre(a):
    ...:     L =[]
    ...:     for i in a:
    ...:         L.append(i**2)
    ...:     return L
    ...:
    ...:

In [15]: carre(b)
Out[15]: [12.96, 324, 144, 625]
```

On peut même affecter le résultat à la variable **b** pour que **b** référence maintenant la liste des carrés : **b = carre(b)**.

Leçon 9 : Introduction aux compréhensions de listes

Les listes sont au coeur de tous les programmes en Python. La liste est un objet extrêmement flexible qui peut référencer n'importe quel type d'objet. Et une manière simple de parcourir une liste, c'est d'implémenter une boucle for. Ainsi, on peut appliquer une opération à chaque élément de notre liste.

Cette opération est tellement commune que Python a inventé une nouvelle expression qu'on appelle **compréhension de liste** qui permet de manière extrêmement simple et intuitive d'appliquer une opération à chaque élément d'une liste et éventuellement d'ajouter une condition de filtre.

Supposons que l'on souhaite prendre les logarithmes d'une liste d'entiers **a = [1, 4, 18, 29, 13]**. On importe le module **math** qui contient la fonction logarithme : **import math**. Notre objectif est de calculer le logarithme de chaque entier de la liste **a** et de placer les résultats à l'intérieur d'une nouvelle liste **b**.

Ci-contre : La boucle **for** parcourt les éléments de **a** et ajoute à la liste **b** le résultat du logarithme de chaque nombre de la liste **a**.

```
In [1]: a = [1, 4, 18, 29, 13]

In [2]: import math

In [3]: b = []

In [4]: for i in a:
   ...:     b.append(math.log(i))
   ...:

In [5]: b
Out[5]:
[0.0,
 1.3862943611198906,
 2.8903717578961645,
 3.367295829986474,
 2.5649493574615367]
```

Cette opération est extrêmement courante en Python. C'est pour ça qu'on a **la compréhension de liste**. La compréhension permet d'appliquer une opération à chaque élément d'une liste et de grouper les résultats dans un nouvel objet liste.

Ci-contre : La même opération que précédemment avec **la compréhension de liste**.

```
In [6]: b = [math.log(i) for i in a]

In [7]: b
Out[7]:
[0.0,
 1.3862943611198906,
 2.8903717578961645,
 3.367295829986474,
 2.5649493574615367]
```

Maintenant ajoutons un nombre négatif dans la liste **a = [1, 4, 18, 29, 13, -1]**. Comme nous avons un nombre négatif, nous ne pouvons plus calculer le logarithme de **-1**. Pour s'en sortir dans une compréhension de liste on peut rajouter un test.

Ci-contre : Le résultat est un nouvel objet liste qui contient le logarithme de chaque élément de **a** uniquement si i est positif **if i > 0**.

```
In [11]: b = [math.log(i) for i in a if i >0]

In [12]: b
Out[12]:
[0.0,
 1.3862943611198906,
 2.8903717578961645,
 3.367295829986474,
 2.5649493574615367]
```

Une compréhension de liste permet d'appliquer n'importe quelle opération, n'importe quelle expression, n'importe quelle fonction à n'importe quel type de séquence.

Leçon 10 : Introduction aux modules

Un module est un fichier python qui finit en **.py**. Lorsqu'on importe ce fichier, cela crée un objet **module**. Un module contient un certain nombre de fonctions, d'opérations à effectuer. L'idée des modules, c'est de mettre des opérations similaires dans le même fichier. Un module est une sorte de boîte à outil que l'on importe quand on a besoin de l'ouvrir.

Pour importer un module, on utilise l'instruction **import** suivie du nom du module. Exemple avec le module **random** :

import random

On peut accéder à tous les objets d'un module avec la fonction **dir( )**. En entrant **dir(random)**, Python affiche tous les attributs du module random. Avec **help(random)**, Python affiche toute l'aide liée à random. **help()** est très pratique pour regarder le fonctionnement d'une méthode particulière issue d'un module : **help(random.randint)** nous indique que la méthode **randint()** renvoie un entier au hasard dans [a, b] en incluant a et b.

Python est livré avec un grand nombre de modules. C'est ce qu'on appelle **la librairie standard**. Il y en a autour d'une centaine. Ces modules permettent de faire un grand nombre d'opérations courantes. L'avantage de la librairie standard c'est qu'elle est disponible par défaut avec Python.

Avec la librairie standard on peut :

- Faire de la programmation parallèle ou de la programmation asynchrone
- Faire de la persistance de données, une opération de serialisation pour garder une copie des objets sur le disque dur
- Faire de la communication sur Internet. Python supporte quasiment tous les protocoles classiques sur Internet
- Formater les données et lire des formats spécifiques de données sur Internet
- Manipuler des fractions ou des nombres décimaux
- Écrire des expressions régulières
- Gérer des dates et des calendriers
- Interagir avec le système de fichiers, créer des fichiers, des répertoires, parcourir des répertoires
- Faire de la compression de fichiers
- Écrire des interfaces graphiques

En plus de sa librairie, Python a également des centaines de milliers de modules écrits par des groupes de développement ou par des individus que l'on peut charger et importer notre programme. Nous verrons notamment dans ce Mooc une partie de la librairie utilisée pour la programmation scientifique, qu'on appelle le **data-science**. Notamment les modules **numpy** et **pandas**.

Leçon 11 : Les fichiers

Les fichiers sont, comme souvent en Python, simples et faciles à utiliser. Il faut cependant maîtriser 3 notions : **l'encodage**, **l'itération** et la notion de **context manager**. Nous avons vu que qu'il est très important de maîtriser l'encodage pour les chaînes de caractères. Avec les fichiers, cette gestion de l'encodage est simple car c'est l'objet fichier qui se charge d'encoder et de décoder les chaînes de caractères pour nous.

Création et ouverture d'un fichier

La création d'un fichier se fait avec l'instruction built-in **open()**.

f = open(r 'c:\temp\spam.txt', 'w', encoding = 'utf8')

Les chaînes de caractères avec des backslash **** interprètent ces backslash. Pour qu'il n'y ait pas d'interprétation on a deux solutions :

- doubler tous les backslash ****
- transformer la chaîne de caractères en **raw string** avec **r**, ce qui désactive tous les caractères backslash. (*méthode recommandée*)

Ouvrir un fichier implique de définir un mode d'ouverture. Les modes d'ouverture les plus courants sont le mode **'w'** pour ouvrir le fichier en écriture ou **'r'** pour ouvrir le fichier en lecture.

Notre fichier étant ici un fichier **.txt**, il faut également spécifier l'encodage avec **encoding = 'utf8'**.

La commande **f = open(r 'c:\temp\spam.txt', 'w', encoding = 'utf8')** permet d'ouvrir un fichier en mode écriture. Si le fichier n'existe pas, il vient d'être créé sur le disque dur avec un encodage utf8. ATTENTION ! Avec le mode d'ouverture en lecture **r**, si le le fichier n'existe pas, Python renvoie une erreur, le fichier n'est pas créé.

Manipulation d'un fichier

Maintenant nous pouvons manipuler le fichier de manière très simple. Nous n'avons plus à nous préoccuper de l'encodage. Nous allons écrire dans notre fichier des chaînes de caractères Python de type **str**. Et après nous pourrons lire des chaînes de caractères de notre fichier. L'objet fichier va se charger pour nous de faire l'encodage et le décodage.

Pour écrire dans un fichier, on fait une boucle :

```
f = open( r 'c:\temp\spam.txt', 'w', encoding = 'utf8' )
for i in range(100):
    f.write( f "ligne { i + 1 } \n")
f.close()
```

Ici, une boucle **for** parcourt tous les entiers de 0 à 99 et avec l'instruction **.write()**, on écrit une chaîne de caractères dans le fichier. Il est très important de ne pas oublier les **\n** (retour chariot) sans quoi tous les **.write()** vont écrire à la suite de la ligne.

Pour finir, il ne faut pas oublier de fermer le fichier avec un **f.close()**.

Dans un interpréteur Python, on peut facilement lancer des commandes **shell** avec **!**. On utilise **type** avec Windows et **less** ou **cat** sous Linux ou Mac OS, pour regarder un fichier.

!type c:\temp\spam.txt

Le fichier apparaît avec à chaque ligne le mot ligne et le numéro de la ligne, de 1 à 100.

Lecture d'un fichier et réécriture dans un nouveau fichier

En Python, le parcours des fichiers est également extrêmement simple puisqu'en Python, les fichiers sont des **itérateurs**. Cela veut dire que l'on peut directement les mettre dans une boucle **for**. Rouvrons notre fichier spam.txt en mode lecture et créons un deuxième fichier (forcément en mode écriture).

```
f = open( r 'c:\temp\spam.txt', 'r', encoding = 'utf8' )
f2 = open( r 'c:\temp\spam2.txt', 'w', encoding = 'utf8' )
```

On va maintenant parcourir notre premier fichier **f**, faire une manipulation des lignes et écrire dans notre fichier **f2**.

```
f = open( r 'c:\temp\spam.txt', 'r', encoding = 'utf8' )
f2 = open( r 'c:\temp\spam2.txt', 'w', encoding = 'utf8' )

for line in f
   line = line.split()
   line[0] = line[0].upper()
   f2.write( ",".join(line) + "\n" )

f.close()
f2.close()
```

On commence par une boucle **for** sur le fichier **f**. Lorsqu'un fichier est ouvert, une boucle **for** le parcourt ligne par ligne. Python va donc être capable automatiquement de détecter les sauts de ligne et il va retourner à chaque tour de la boucle **for** une nouvelle ligne.

Avec **line = line.split()** on transforme chaque ligne chaîne de caractère en objet liste dans lequel ici, le premier élément va être le mot ligne et le deuxième élément le numéro de la ligne.

Dès lors on peut transformer chaque colonne ligne en majuscule avec l'instruction **line[0] = line[0].upper()**. L'objet ligne étant désormais une liste, il est mutable, donc modifiable en place.

Ensuite, on souhaite retransformer nos objets liste en objets chaîne de caractères que l'on veut écrire dans notre fichier **f2**. Et on souhaite au passage remplacer les espaces qui séparent nos deux colonnes par des virgules. Pour cela on va écrire dans le fichier **f2** avec l'instruction **.write()** dans laquelle on fait un **.join()**, en oubliant pas de rajouter un **"\n"** pour avoir un retour chariot à chaque ligne.

Enfin on ferme les fichiers **f** et **f2** avec l'instruction **.close()**. Quand on regarde le fichier **f2**, on constate que le fichier a bien été modifié avec à chaque ligne le mot ligne en majuscule et avec une virgule entre les colonnes : **LIGNE,81**.

Pourquoi doit-on fermer les fichiers ?

On peut se demander pourquoi est-ce tellement important de fermer les fichiers. La raison est simple. Le programme Python discute avec le système d'exploitation et c'est le système d'exploitation qui va effectivement ouvrir, fermer et écrire dans les fichiers. Lorsqu'on fait un **open()**, on dit au système d'exploitation : « attention! Je veux ouvrir ce fichier ». Et lorsqu'on fait **.close()** on dit au système d'exploitation que l'on veut fermer le fichier.

Comme le nombre de fichiers simultanément ouverts est limité dans un système d'exploitation. Si on ne dit pas que l'on fait un **.close()**, le système d'exploitation va croire que le fichier est toujours ouvert. Et on peut se retrouver dans une situation dans laquelle, le système d'exploitation n'est plus capable d'ouvrir de nouveaux fichiers.

Le protocole de context manager

On peut continuer à s'interroger sur le fait que c'est au programmeur de penser systématiquement à fermer le fichier, alors qu'en Python tout est un objet et qu'un fichier est également un objet. On pourrait se dire : qu'on pourrait faire en sorte que l'objet soit suffisamment intelligent pour savoir que lorsqu'on n'en a plus besoin, il fait toutes les opérations nécessaires à sa fermeture.

En fait, ce mécanisme existe. C'est quelque chose qu'on appelle un protocole : **le protocole de context manager**. Un objet fichier implémente ce protocole context manager. Pour y accéder, il faut utiliser l'instruction **with** puis ouvrir le fichier avec **open()** et en lui donner un nom avec **as f**.

```
with open( r 'c:\temp\spam.txt', 'r', encoding = 'utf8' ) as f:
```

Un bloc d'instruction va ensuite être exécuté. Et lorsque l'on sortira de ce bloc d'instruction, Python appellera automatiquement une méthode exit sur le context manager de l'objet fichier qui va avoir pour effet de fermer automatiquement le fichier. Même si on a une exception, le fichier sera quand même automatiquement fermé.

Comment écrire dans un fichier binaire

with open(r ‘c:\temp\spam.txt’, ‘wb’) as f:

Pour ouvrir en mode binaire, il faut ajouter un **b** dans le mode d’ouverture :

- **’wb’** pour ouvrir en mode binaire et écriture
- **‘rb’** pour ouvrir en mode binaire et lecture

Comme le fichier est ouvert en mode binaire, il n’y a pas besoin de préciser l’encodage. On ne peut écrire dans ce fichier que des objets de type **bytes**. Pour écrire un objet de type **bytes**, il suffit de placer un **b** avant une chaîne de caractère.

```
with open( r ‘c:\temp\spam.bin’, ‘wb’ ) as f:
    for l in range(100):
    f.write( b ’\x3d’ )
```

À la sortie du bloc de code, le **context manager** a fermé le fichier automatiquement. Nous avons maintenant sur notre disque dur un fichier spam.bin qui contient le caractère **3d** écrit 100 fois.

Leçon 12 : Les tuples

Dans cette leçon nous allons parler d'un nouveau type built-in qui s'appelle **le tuple**. Le tuple est très proche de la liste. Comme une liste est une séquence, on peut donc appliquer les opérations comme le test d'appartenance avec **in**, accéder aux différents éléments avec un crochet, faire du slicing dessus. Et également, un tuple peut référencer des objets complètement hétérogènes. C'est donc très proche de la liste.

Mais il y a une différence fondamentale entre la liste et le tuple. C'est que le tuple est un objet immuable. Ça veut dire qu'une fois qu'on a créé le tuple, on ne peut plus le modifier.

Nous verrons la raison de l'existence du tuple lorsque nous parlerons des dictionnaires dans une prochaine vidéo.

La syntaxe des tuples

Commençons par créer un tuple vide. On crée un tuple en utilisant simplement des parenthèses ouvrantes et fermantes : **t = ( )**. Comme le tuple est immuable, lorsqu'on crée un tuple vide, on ne peut rien ajouter. Donc ce tuple vide a assez peu d'intérêt.

On peut créer un tuple avec un élément avec la syntaxe suivante : **t = (4,)**. Nous rajoutons une virgule en fin du premier élément. Si on écrit **t = (4)**, sans virgule, pour Python les parenthèses servent ici simplement à grouper des opérations. Et Python va considérer **t** comme un entier qui vaut 4. Donc pour un tuple singleton, un tuple d'un seul élément, il ne faut pas oublier de mettre la virgule.

On peut créer un tuple de plusieurs éléments, qui contient des objets complètement hétérogènes, exactement comme une liste : **t = (True, 3.4, 18)**. Une caractéristique importante du tuple c'est que les parenthèses sont facultatives. **t = True, 3.4, 18** donne le même objet tuple que **t = (True, 3.4, 18)**.

Opérations de base

Comme me tuple est un objet de type séquence, on peut faire toutes les opérations que l'on peut faire sur une séquence. On peut demander : **3.4 in t** renvoie **True**, le flottant est bien dans notre tuple. On peut utiliser les crochets pour accéder aux élément du tuple : **t[1]** renvoie **3.4**. On peut également faire un slice pour aller du début jusqu'à l'élément 2 exclu : **t[:2]** renvoie **(True, 3.4)**.

On peut convertir un tuple en liste : **a = list(t)** renvoie **[True, 3.4, 18]**. Et on peut faire l'opération inverse, transformer notre liste en tuple. On peut ainsi modifier le premier élément **a[0] = False** de telle sorte que la liste vaut maintenant **[False, 3.4, 18]**. Puis on reconvertit cette liste en tuple : t = **tuple(a)** et maintenant **t = (False, 3.4, 18)**. ATTENTION! Le tuple étant immuable, nous n'avons pas modifié notre objet tuple. Nous avions un tuple que l'on a converti en objet liste. On a modifié l'objet liste et on a créé un nouvel objet tuple.

Le tuple unpacking

Le **tuple unpacking** fonctionne de la manière suivante : nous avons dans un tuple deux variables, **a** et **b**, et on dit que ces variables sont égales à une séquence qui doit avoir le même nombre d'éléments que l'on a dans notre tuple. Par exemple : **(a, b) = [3, 4]** renvoie la variable **a** référence l'entier **3** et la variable **b** référence l'entier **4**.

Une des raisons pour lesquelles on peut enlever les parenthèses du tuple, c'est justement pour alléger cette notation. Il est beaucoup plus naturel d'écrire **a, b = 3, 4** plutôt que d'écrire le tuple avec des parenthèses est égal à un tuple ou une liste avec des parenthèses ou crochets.

En Python, il existe également la notion de **extended tuple unpacking**. C'est une simple de pouvoir isoler des éléments lorsqu'on a un grand nombre d'éléments dans une séquence. On prend **a = list(range(10))**. Nous avons donc **a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**. Maintenant supposons que nous voulions prendre uniquement le premier élément de cette liste et les autres éléments dans un autre objet. On peut écrire **x, *y = a**. Ainsi **x = 0** et **y** référence une liste qui va référencer tous les éléments qu'il reste dans **a**. Donc **y = [1, 2, 3, 4, 5, 6, 7, 8, 9]**. On peut utiliser la notation dans l'autre sens : ***x, y = a** pour obtenir un autre **tuple unpacking** dans lequel on isole uniquement le dernier élément : **x = [0, 1, 2, 3, 4, 5, 6, 7, 8]** et **y = 9**.

Leçon 13 : Tables de hash

Jusqu'à maintenant nous avons couvert les types séquence avec notamment listes, les chaînes de caractères et les tuples. Dans cette leçon, nous allons parlé des **tables de hash**, une structure de données qui permet de répondre à certaines limitations des types séquence.

Les limitations des types séquences

Les types séquence ont été optimisés pour l'accès, la modification et l'effacement en fonction d'un numéro de séquence. Cependant ces types n'ont pas été optimisés pour le test d'appartenance.

Un test d'appartenance linéaire et dépendant du nombre d'éléments :

Exemple avec l'instruction **%timeit** qui permet de calculer le temps d'exécution d'une expression et nous allons regarder si la chaîne de caractères **x** est dans **range(100)**.

%timeit 'x' in range(100)

'x' n'étant pas dans range(100), la fonction va devoir vérifier un par un les 100 entiers de 0 à 99. Le test renvoie un temps d'exécution de 2,33 micro-secondes. C'est donc rapide. Prenons maintenant un range plus important, multiplié par 100.

%timeit 'x' in range(10_000)

Le test renvoie un temps d'exécution de 284 micro-secondes. C'est donc à peu près 100 fois plus lent. Prenons un range encore plus important à nouveau multiplié par 100.

%timeit 'x' in range(1_000_000)

Le test renvoie un temps d'exécution de 20 mili-secondes. C'est de nouveau 100 fois plus lent.

On voit donc clairement que l'opération d'appartenance sur une séquence est linéaire en fonction du nombre d'éléments que l'on a dans notre séquence. Ça c'est un problème, parce que le test d'appartenance est quelque chose de tellement commode, qu'on aimerait pouvoir faire un test d'appartenance qui indépendant du nombre d'éléments.

Une capacité d'indice limitée aux entiers et aux slices :

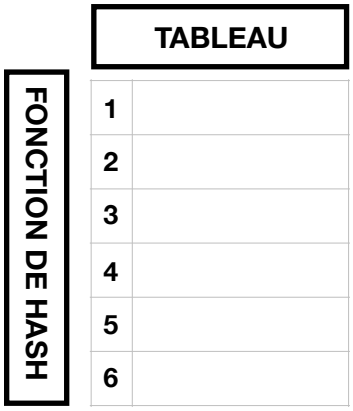
Si on prend maintenant une séquence **t = [1, 2]**, on peut accéder au premier élément en faisant **t[0]**. Mais supposons que nous avons des âges dans notre liste : **t = [18, 35]**. On pourrait vouloir écrire **t['alice'] = 35** pour lier un nom à un âge, donc indicer notre séquence non plus avec des entiers mais avec des chaînes de caractères. Et bien ça on ne peut pas le faire. **t['alice'] = 35** renvoie une erreur qui dit que les indices des listes doivent être des entiers ou des slices mais pas des chaînes de caractères.

La structure de données tables de hash permet de répondre à ces deux limitations.

Fonctionnement des tables de hash

Une table de hash est constituée d'un tableau et d'une fonction dont le but est : quand on passe à notre fonction un objet, elle calcule une valeur comprise entre le nombre de lignes du tableau.

Ci-contre : Un tableau avec 6 éléments et une fonction de hash. Lorsqu'on lui passera un objet, la fonction calculera une valeur comprise entre 1 et 6. Le but de cette fonction est de créer une correspondance entre un objet quelconque et une case du tableau.
f(x) appartient à [1,6]
Cet ensemble **fonction de hash** et **tableau** constitue **une table de hash**.



Supposons que l'on souhaite ajouter une correspondance entre la chaîne de caractères **'eve'** et l'âge **34**. On va passer une clef, spécifiée entre crochets, que l'on va associer à une valeur.

T[eve] = 34
clef valeur
eve passe à la fonction de hash, qui fait un calcul sur cet objet et va retourner une case dans le tableau. Ici la case 2.
T[bob] = 27
Ensuite si on veut accéder à **T[eve]**, on peut faire un **print(T[eve])** qui va afficher **34**.

FONCTION DE HASH

TABLEAU

1	
2	**(eve, 34), (jo, 46)**
3	
4	**(bob, 27)**
5	
6	

Dans une table de hash, l'insertion, l'effacement, la recherche d'élément est indépendante du nombre d'éléments. C'est conditionné par la vitesse de la fonction de hash. On fait un calcul et on obtient directement la case où est stockée la valeur correspondant à la clef.

On voit que le tableau est limité. Donc au bout d'un moment, on va pouvoir se voir retourner des cases qui sont déjà occupées. Ainsi si on fait **T[jo] = 46**, la fonction de hash retourne la case 2, on va donc stocker **(jo, 46)** à la suite de l'autre clef **eve** associée à sa valeur **34**. Si maintenant on fait un **t[jo]**, on va aller chercher **jo** avec la fonction de hash qui va toujours retourner la case 2. Et on va regarder est-ce que **jo** correspond au premier couple ? Non. On passe donc au deuxième couple. La réponse est alors oui. **T[jo]** retourne donc la valeur **46**.

L'efficacité d'une table de hash dépend de 2 conditions :

- la taille du tableau. Si on a un tableau trop petit, on va avoir beaucoup de collisions (une collision : lorsque 2 clefs correspondent à la même entrée dans le tableau).
- la capacité de la fonction de hash à bien répartir les clefs dans les différentes cases du tableau. Cette capacité de la fonction de hash à bien répartir les clefs est une caractéristique majeure d'une table de hash efficace.

En Python, les tables de hash ont été implémentées de manière très efficace. On n'a ni à se préoccuper de la fonction de hash, ni de la taille du tableau, Python gère ces paramètres automatiquement. On peut donc faire l'hypothèse qu'en Python, les tables de hash permettent d'avoir un temps d'accès, un temps d'insertion, un temps d'effacement et un temps de recherche qui soient indépendants du nombre d'éléments.

Les tables de hash permettent d'accéder, d'effacer, de modifier mais également de faire des tests d'appartenance avec **une complexité qu'on appelle O(1)** qui veut essentiellement dire que c'est indépendant du nombre d'éléments dans notre table de hash. C'est donc une structure de données très intéressante et qui permet également d'indexer des valeurs non-pas avec des entiers comme dans une séquence mais avec par exemple des chaînes de caractères. En Python, il y a deux implémentations de tables de hash : **les sets** et **les dictionnaires**.

Leçon 14 : Les dictionnaires

Dans cette leçon nous allons voir l'utilisation des dictionnaires en Python. Les dictionnaires sont des tables de hash. On a donc un temps d'accès, d'insertion, d'effacement et un test d'appartenance qui sont indépendants du nombre d'éléments. De plus, les dictionnaires sont des objets mutables. Cela veut dire que l'on peut les modifier en place avec une excellente efficacité mémoire.

Dans un dictionnaire, on peut avoir comme clef n'importe quel objet qui est **hashable**, c'est-à-dire un objet sur lequel on peut calculer la fonction de hash. Pour l'instant, il faut savoir qu'en Python, tous les objets immuables sont hashables et que tous les objets mutables ne sont pas hashables. La fonction de hash doit faire un calcul sur la clef. Or, si cette clef change en cours d'exécution, la fonction de hash va faire un calcul différent et par conséquent la table de hash va devenir inconsistante. C'est pourquoi, en Python, avec les types built-in, seul les types immuables, c'est-à-dire qui ne peuvent plus changer une fois qu'ils ont été créés, peuvent être utilisés comme clefs d'un dictionnaire.

Fonctionnement des dictionnaires

Pour créer un dictionnaire, on utilise la notation accolade : **age = { }** est un objet de type **dict**, le type dictionnaire. On peut créer un dictionnaire, en écrivant des couples clef-valeur séparés par des deux-points : **age = { 'ana' : 35, 'eve' : 30, 'bob' : 38 }**. Ici nous avons un dictionnaire qui a trois clefs (ana, eve et bob) et trois valeurs (35, 30, 38). Le dictionnaire peut être vu comme une collection de couples **clef : valeur**. Le dictionnaire stocke cette collection qui n'est pas ordonnée. Il n'y a pas d'ordre dans un dictionnaire. Le dictionnaire permet d'accéder aux différents éléments en nommant la clef. Par exemple : **age['ana']** renvoie la valeur correspondante, ici **35**.

Une deuxième manière permet de construire un dictionnaire. Supposons qu'au départ, nous avons une liste qui contient des tuples clef-valeur : **a = [('ana', 35), ('eve', 30), ('bob', 38)]**. Nous avons donc une liste avec des tuples. À partir de là, pour créer un dictionnaire à partir de cette liste de tuples, il suffit d'écrire : **age = dict(a)**. La fonction built-in **dict()** va créer un dictionnaire à partir des couples contenus dans la liste **a**. Avec l'instruction **age['bob']** Python retourne la valeur correspondante **38**. Et comme le dictionnaire est mutable, on peut changer la valeur correspondant à **bob** : **age['bob'] = 45** et si on regarde le dictionnaire, la valeur correspondant à **bob** a bien été modifiée : **age** renvoie **{ 'ana' : 35, 'eve' : 30, 'bob' : 45 }**.

ATTENTION! Les dictionnaires ne sont pas ordonnées. On n'a aucune garantie d'ordre lorsque l'on fait **print()** d'un dictionnaire. L'ordre affiché sera quelconque.

[*Mise à jour*] *En Python 3.6, on avait, par effet de bord de l'implémentation, la préservation de l'ordre d'insertion des clefs. En Python 3.7, cette préservation de l'ordre d'insertion des clefs est devenue officielle.*

On peut également effacer un couple clef-valeur en utilisant l'instruction **del** : **del age['bob']**, Python efface la clef **bob** et la valeur correspondante **45**. Le dictionnaire ne contient maintenant plus que deux couples clef-valeur : **age** renvoie **{ 'ana' : 35, 'eve' : 30 }**.

Les opérations sur les dictionnaires

Les dictionnaires ont des opérations très proches des séquences lorsque cette opération a un sens. Par exemple, pour compter le nombre de clefs ou de couples clef-valeur dans un dictionnaire, on utilise la fonction built-in **len(age)** qui renvoie **2**. On peut également exécuter du test d'appartenance sur les dictionnaires : **'ana' in age** renvoie **True**. **'bob' in age** renvoie **False**.

Il y a un vrai souci d'uniformité en Python. Lorsque c'est possible, on utilise, quelque soit le type, des instructions qui sont les mêmes.

L'accès aux clefs, aux valeurs et aux couples clef-valeur

En Python, les couples clef-valeur sont appelés les **items** d'un dictionnaire.

Pour les clefs, on utilise la méthode **.keys()** : **age.keys()** renvoie **dict_keys(['ana', 'eve'])**.

Pour les valeurs, on utilise la méthode **.values()** : **age.values()** renvoie **dict_values([35, 30])**.

Pour les items, on utilise la méthode **.items()** : **age.items()** renvoie **dict_items([(‘ana’, 35), (‘eve’, 30)])**.

Les vues

Les méthodes **.keys()**, **.values()** et **.items()** retournent un objet particulier qu’on appelle **une vue**. En Python, une vue est un objet sur lequel on peut itérer. On peut donc faire une boucle **for** sur cet objet. Et on peut également faire un test d’appartenance, donc faire par exemple **in** directement sur cette vue. La caractéristique principale des vues, c’est qu’elles sont mises à jour en même temps que le dictionnaire.

Exemple : on référence une vue des clefs de **age** dans une variable **k = age.keys()**. **k** renvoie **dict_keys([‘ana’, ‘eve’])**. Maintenant on modifie le dictionnaire **age** en rajoutant un **item** : **age[‘bob’] = 25** (cette notation permet de modifier une valeur existante ou alors de rajouter un nouvel item comme dans le cas présent). Si on regarde à nouveau notre vue **k**, Python renvoie **dict_keys([‘ana’, ‘eve’, ‘bob’])**. Notre vue a directement été mise à jour avec la nouvelle clé **bob**.

Une vue doit être perçue comme une vue permanente sur notre objet. Si l’objet est modifié, la vue va voir l’objet avec sa modification.

On peut faire un test d’appartenance sur notre vue : **‘ana’ in k** renvoie **True** tandis que **‘bill’ in k** renvoie **False**.

Parcourir les dictionnaires

Une manière classique de parcourir les dictionnaires est d’utiliser une boucle **for** en utilisant la notation de **tuple unpacking** :

```
for k, v in age.items():
   print( f“{k}, {v}“ )
```
renvoie **ana 35**
eve 30
bob 25

L’itérateur sur les dictionnaires sans spécification de vue fonctionne directement sur les clefs :

```
for k in age
   print(k)
```
renvoie **ana**
eve
bob

Leçon 15 : Les ensembles

Dans cette leçon nous allons parlé des sets. Les sets sont très proches des dictionnaires. Comme les dictionnaires, ils permettent de faire des tests d'appartenance, d'accéder / effacer / modifier des éléments indépendamment du nombre d'éléments. Les sets sont également des objets mutables. Mais à la différence des dictionnaires, les sets ne stockent qu'une clef. Il n'y a pas de valeur correspondante. La raison c'est que le set a été optimisé et créé pour des opérations spécifiques. Par exemple pour garder uniquement le nombre d'éléments uniques d'une séquence. Donc si on calcule le set des éléments d'une séquence, on ne va avoir que les éléments uniques. Une autre opération pour laquelle le set est très utilisé : c'est pour faire des tests d'appartenance sur les éléments d'une séquence.

Création d'un set

Les sets sont des objets mutables, on peut donc les modifier en place.

On commence par créer un set vide avec la fonction built-in **set()** : **s = set()**. Ce qui nous donne un objet de type **set** qui est vide.

On peut également créer un set avec des accolades : **s = {1, 2, 3, 'a', True}**. La différence entre les notations d'un set et d'un dictionnaire, c'est que dans le set il n'y a pas la notation deux-points (**:**) qui sépare la clef et la valeur.

On peut aussi créer un set à partir d'une séquence. Pour une liste **a = [1, 2, 4, 1, 18, 30, 4, 1]**, donc composée d'une séquence d'entiers avec des éléments dupliqués. Si on effectue un **set(a)**, on ne conserve que les éléments uniques et on crée un set à partir de ces éléments : **{1, 2, 4, 18, 30}**.

Manipulation d'un set

On peut aussi manipuler un set avec l'option built-in **len()** pour obtenir le nombre d'éléments : **len(s)** renvoie **4**.

[*POURQUOI len(s) = 4 et pas 5 ?*] *Nous avons créé **s = {1, 2, 3, 'a', True}**, or **True** est égal à l'entier **1** (**True == 1**). Par conséquent, Python renvoie un set de 4 éléments uniques : **{1, 2, 3, 'a'}**.*

Le test d'appartenance est fait avec l'instruction **in** : **1 in s** renvoie **True** et **b in s** renvoie **False**.

Dans l'ensemble **s**, on peut ajouter des éléments avec la méthode **.add()** : **s.add('alice')** renvoie **{1, 2, 4, 'alice', 'a'}**. On peut aussi ajouter une séquence d'éléments avec la méthode **.update()** : **s.update([1, 2, 3, 4, 5, 6, 7])** renvoie **{1, 2, 3, 'alice', 'a', 4, 5, 6, 7}**. Évidemment, seuls les nouveaux éléments sont ajoutés au set **s**.

On peut également faire des opérations d'ensemble classiques sur un set. Pour **s1 : {1, 2, 3}** et **s2 = {3, 4, 5}**, on peut calculer une différence entre deux ensemble **s1 - s2** qui va en fait enlever tous les éléments de **s2** dans **s1** et qui renvoie donc ici **{1, 2}**. On peut également prendre **l'union** et **l'intersection** des ensembles :

- Pour l'union : **s1 | s2** renvoie **{1, 2, 3, 4, 5}**
- Pour l'intersection : **s1 & s2** renvoie **{3}**

L'efficacité du test d'appartenance

On peut se demander : lorsque l'on fait un test d'appartenance, est-il rentable de convertir une séquence en set ou vaut-il mieux faire directement le test d'appartenance sur une séquence ?

Lorsque l'on fait un test d'appartenance sur une séquence, on doit parcourir tous les éléments de la séquence jusqu'à trouver l'élément que l'on cherche. Si l'élément n'est pas présent, on doit parcourir tous les éléments de la séquence. Cela correspond essentiellement au fait de regarder une case mémoire dans l'ordinateur et de faire une comparaison entre l'objet stocké dans cette case mémoire et l'objet que l'on veut comparer.

Faire un test d'appartenance sur un set est très différent. **Un set est une table de hash**. Cela veut donc dire que faire un test d'appartenance représente essentiellement un calcul d'une fonction de hash sur l'objet que l'on recherche. Cette fonction de hash donne une case et ensuite

on compare si l'objet est le bon. On peut dès lors se demander combien de temps prend le calcul de cette fonction de hash sur l'objet ? En fait c'est extrêmement rapide et c'est essentiellement de l'ocre de grandeur de l'accès à un élément dans une séquence.

Pour une liste **a : [0]** et un set à partir de cette liste **s = set(a)**, faisons un test d'appartenance avec **%timeit** :

 # on donne un argument -n 50 pour dire
 # à timeit de ne fait que 50 boucles pour
 # que le timeit soit plus rapide
%timeit -n 50 0 in a
renvoie 38,6 nano-secondes

%timeit -n 50 0 in s
renvoie 41,7 nano-secondes

Cela veut dire que dès que l'on a à faire un test d'appartenance, il vaut toujours mieux convertir la séquence en set. Ce sera dans la quasi-totalité des cas, une opération rentable.

Leçon 16 : Les exceptions

Depuis le début de ce MOOC, nous les avons rencontré mais nous ne savons pas quoi en faire ni ce que c'est. Ce sont les exceptions. Dans cette leçon, nous allons voir comment gérer les exceptions.

Il y a trois choses importantes à savoir sur les exceptions :

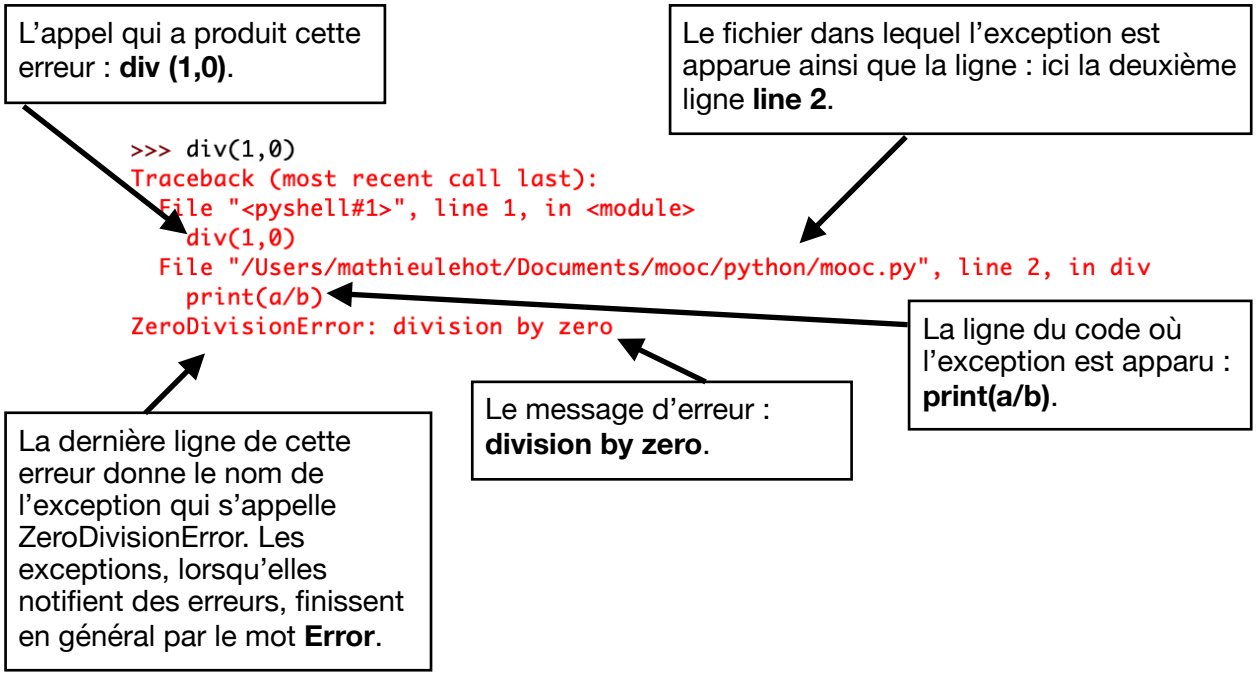
- L'exception n'est pas une fatalité. C'est un mécanisme de communication d'erreur tout à fait normal dans un programme et on est capable de les capturer et de réagir à une exception
- Les exceptions fournissent des informations sur l'erreur qui se produit. C'est donc un mécanisme de notification d'erreur extrêmement utile
- Les exceptions étant très efficaces en Python, c'est un mécanisme qu'on utilise couramment dans un fonctionnement normal d'un programme

Comprendre une exception

Pour commencer, créons une fonction simple :

```
def div (a, b):
    print(a/b)
```

On peut maintenant appeler la fonction : **div(1, 2)** qui renvoie **0.5**. Maintenant si on fait un **div(1, 0)** Python renvoie une erreur d'exécution, la division par 0 étant impossible.



L'exception fournit donc énormément d'information. Elle donne la raison de l'erreur et elle permet également de situer cette erreur dans le contexte d'exécution du programme.

La capture d'une exception

Une exception n'est pas une fatalité. On peut la capturer. Pour capturer une exception on utilise un bloc **try except**. Tout ce qui est entre le **try** et le **except** va être évalué et si on a une exception qui se produit dans ce bloc de code, Python va regarder si cette exception a été capturée par le code.

```
def div (a, b):
    try:
        print(a/b)
    except ZeroDivisionError:
        print("attention, division par 0")
    print("continuons")
```

Ensuite si on exécute **div(1, 0)**, l'exception est produite mais elle est correctement capturée par la clause **except**. Le bloc d'instruction de la clause **except** est exécuté : il renvoie **attention, division par 0** et le programme continue à s'exécuter normalement comme l'atteste le print **continuons**.

```
>>> div(1,0)
attention, division par 0
continuons
```

On peut donc tout à fait capturer une exception, avoir un comportement approprié qui réagisse à cette exception et continuer l'exécution de notre programme.

Maintenant si nous exécutons notre fonction **div** avec un chiffre en type chaîne de caractère : **div(1, '0')**. Python renvoie une erreur qui s'appelle **TypeError**.

```
>>> div(1,'0')
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    div(1,'0')
  File "/Users/mathieulehot/Documents/mooc/python/mooc.py", line 3, in div
    print(a/b)
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Le message d'erreur explique que l'opération division n'est pas supportée entre les entiers et les chaînes de caractères. De nouveau, on peut tout à fait réagir à cette exception en rajoutant une clause **except**.

```
def div (a, b):
    try:
        print(a/b)
    except ZeroDivisionError:
        print("attention, division par 0")
    except TypeError:
        print("il faut des int !")
    print("continuons")
```

Ce qui renverra maintenant **il faut des int !** si on entre une chaîne de caractère dans la fonction **div**. On peut ajouter autant de clauses **except** qu'on le veut pour réagir à des exceptions particulières.

En Python, on peut avoir une clause **except** sans spécifier aucune exception :

```
def div (a, b):
    try:
        print(a/b)
    except:
        print("attention, division par 0")
    print("continuons")
```

C'est en général une très mauvaise pratique. Parce que ça va cacher les exceptions produites par le code. Si on ne connaît pas la raison d'un problème dans un programme, il risque de planter plus loin dans le code.

Donc d'une manière générale, on doit toujours capturer les exceptions que l'on a étudié et que l'on sait qui vont se produire dans notre code et laisser remonter les exceptions que nous n'avons pas prévues.

Le mécanisme de bubbling

Une caractéristique importante des exceptions c'est qu'elles **bubble**. Ça veut dire qu'elle vont remonter la pile d'exécution jusqu'à arrêter le programme.

Prenons deux fonctions **div** et **f** :

```
Def div (a, b):
    print(a/b)

Def f(x):
    div(1,x)
```

Lorsqu'on appelle la fonction **f**, elle exécute la fonction **div**. Mais tant que **div** n'a pas retourné de valeur, la fonction **f** reste en cours d'exécution. On dit qu'elle reste dans **la pile d'exécution**. En fait, la pile d'exécution va contenir toutes les fonctions de notre programme qui ont été appelées mais qui n'ont pas encore retourné de valeur.

Avec **f(1)**, on appelle **div(1,1)**, Python affiche **1**. Si maintenant on fait **f(0)**, on appelle **div(1,0)** on va donc avoir une exception produite par la division **a/b** à l'intérieur du print. L'exception va être produite. Elle va arrêter l'exécution de la fonction. Mais la fonction a été appelée par **f**. L'exception va donc remonter la pile d'exécution, elle va sortir dans la fonction **f**, elle n'a pas été capturée, elle va remonter la pile d'exécution et elle va arrêter le programme.

```
>>> f(0)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    f(0)
  File "/Users/mathieulehot/Documents/mooc/python/mooc.py", line 5, in f
    div(1,x)
  File "/Users/mathieulehot/Documents/mooc/python/mooc.py", line 2, in div
    print(a/b)
ZeroDivisionError: division by zero
```

Ce **mécanisme de bubbling** a deux avantages majeurs :

- on peut capturer l'exception n'importe où le long de la pile d'exécution. On peut donc tout à fait capturer notre exception au niveau du **print**, au niveau de la fonction **f**, dans la fonction **f** au niveau de l'appel de **div** et lors de l'appel de **f**.
- notre trace d'exécution est capable de dire exactement par où est passée l'exception et fournit donc des informations précieuses sur le diagnostic du problème dans notre programme.

Ici, nous noyons que nous avons une exception qui s'appelle **ZeroDivisionError** qui a été produite par **print(a/b)**. En remontant le message d'erreur, on voit que cette exception a eu lieu dans la fonction **div(1,x)** et que cette fonction **div** a été appelée par la fonction **f(0)**. On peut ainsi remonter à l'origine du problème : ici l'appel de **f(0)**.

Conclusion

Une bonne pratique en Python est de capturer les exceptions que l'on connaît et de les capturer au plus près de l'endroit où elles se produisent dans le code. Plus elles sont capturées tôt, et plus notre réaction peut être appropriée à l'origine du problème.

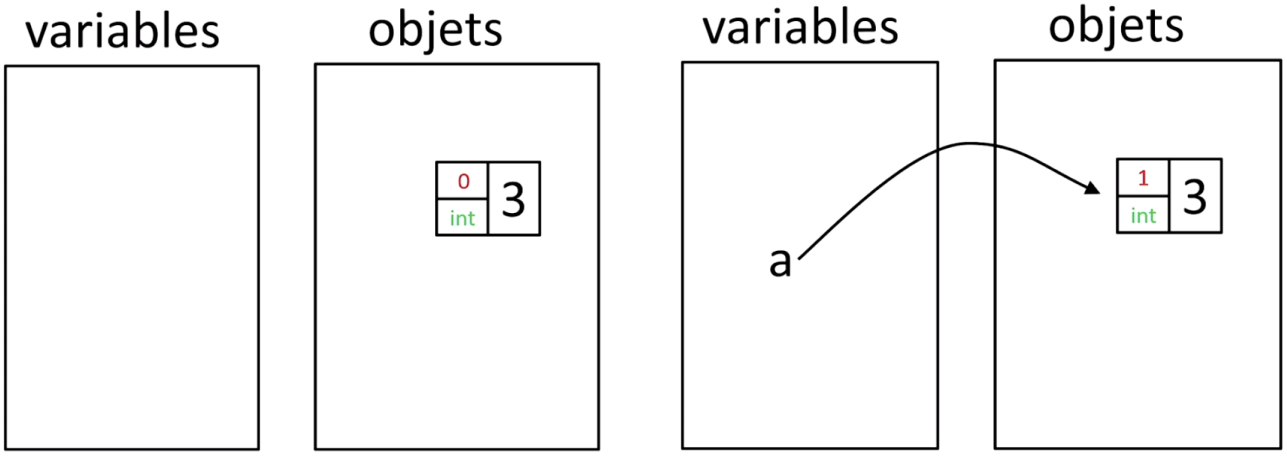
Pour connaître les exceptions que l'on doit capturer, il n'y a pas de miracle. Il faut lire en détail la documentation des modules et des objets que l'on utilise.

Leçon 17 : Les références partagées

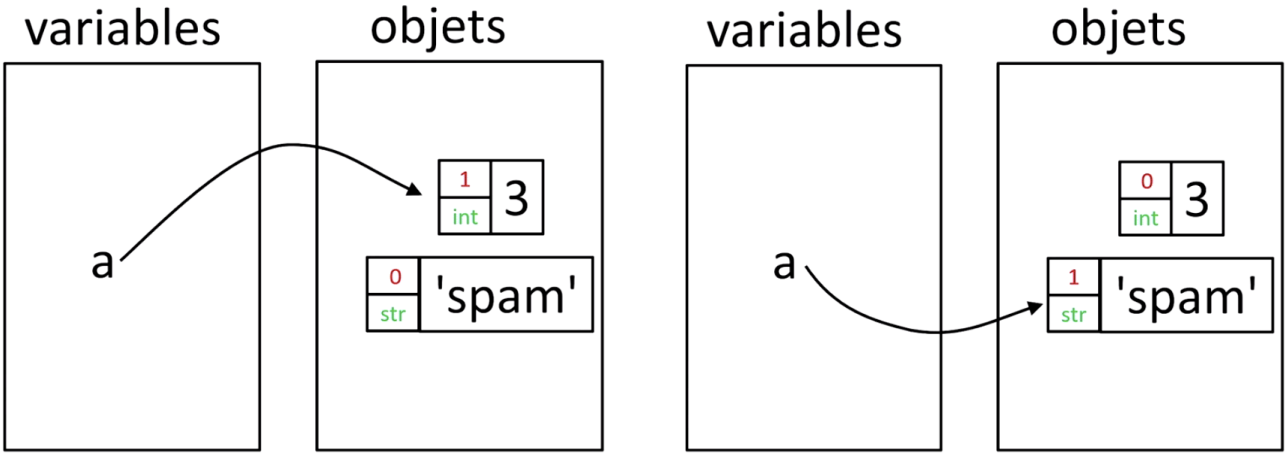
Dans cette leçon, nous allons voir une notion centrale en Python : la notion de références partagées. C'est ce mécanisme de références partagées qui permet d'accéder aux attributs de n'importe quel objet dans un programme.

Fonctionnement des références partagées

Supposons une affectation **a = 3**. Cette affectation va commencer par créer l'entier **3** auquel vont être associés deux champs importants :
• un compteur de référence, ici à 0, qui va représenter le nombre de variables qui référencent cet objet
• un champ de type qui représente le type de l'objet, ici **int**. Python étant un langage à typage fort, le type, une fois qu'il a été défini, ne peut plus être changé.
Ensuite on crée une variable **a** qui référence cet objet entier. Le compteur de référence de **3** passe à 1. On a maintenant une référence vers l'entier.

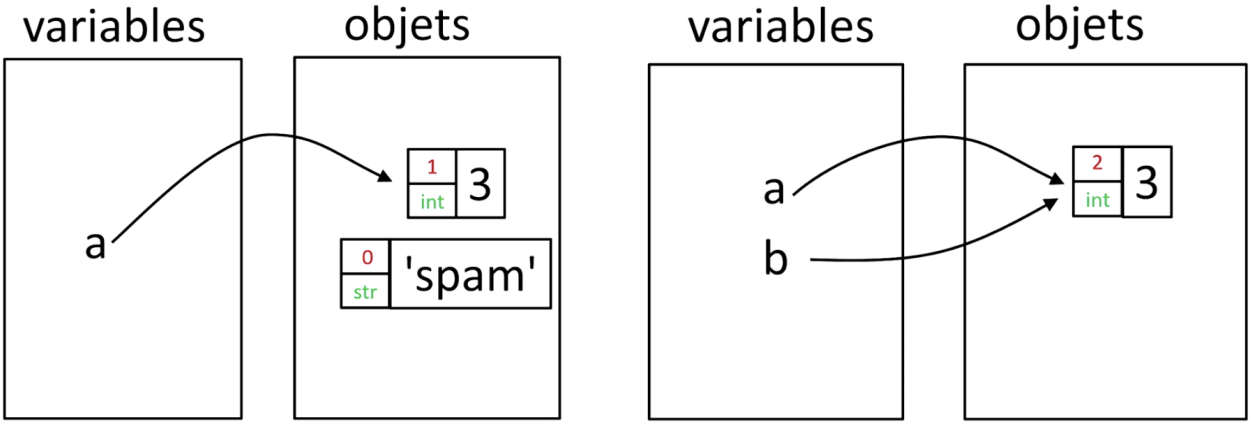


Ensuite nous faisons **a = 'spam'**. On crée un objet **'spam'** de type chaîne de caractère. Le compteur de référence est à 0. Ensuite, la variable **a** ne référence plus l'entier **3**. Le compteur de référence de **3** passe à 0. Et **a** référence maintenant la chaîne de caractère **'spam'** et le compteur de **'spam'** passe à 1.

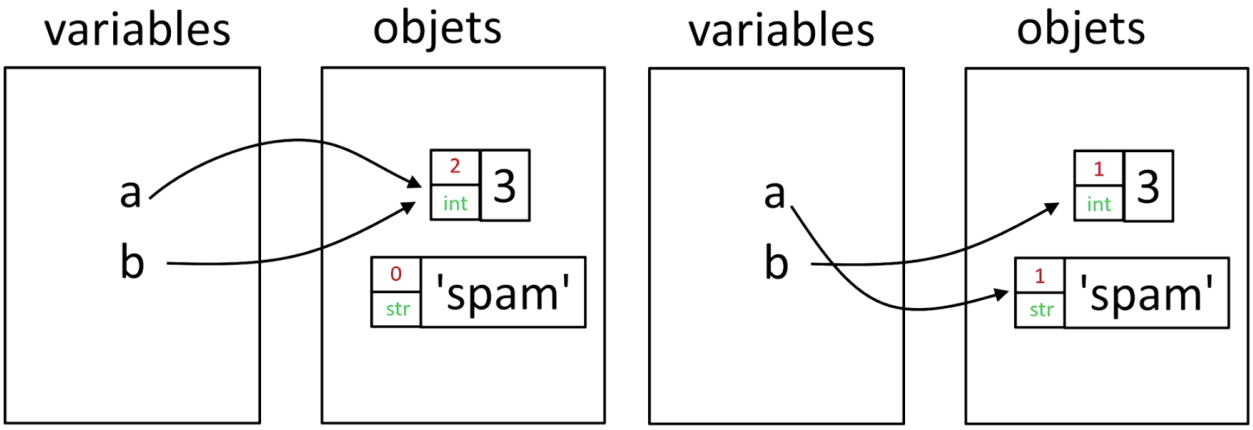


Lorsque le compteur d'un objet passe à 0, le module **garbage collector** prend un accès sur cet objet et va libérer la mémoire occupée par cet objet lors d'un cycle de **garbage collection**. On ne gère pas nous même ces cycles de nettoyage de la mémoire, c'est le module **garbage collector** (**GC**) qui gère ça automatiquement pour nous. Donc en pratique, on n'a pas à se préoccuper de la création des zones mémoires et de la libération des zones mémoires. Python s'en charge pour nous.

Reprenons l'affectation **a = 3** et ajoutons **b = a**. On crée l'objet à droite, sauf que cette fois-ci, c'est une variable qui est à droite. Donc on va en fait simplement réutiliser l'objet qui est déjà référencé par la variable **a**. Ensuite on crée une variable **b** dans l'espace des variables. Et la variable **b** va référencer l'entier **3**. Le compteur de référence passe à 2. On a maintenant deux références vers l'objet entier **3**.



Si on fait ensuite **a = 'spam'**, nous créons un nouvel objet **'spam'**. La variable **a** va enlever sa référence vers l'entier **3** pour référencer l'objet **'spam'**.

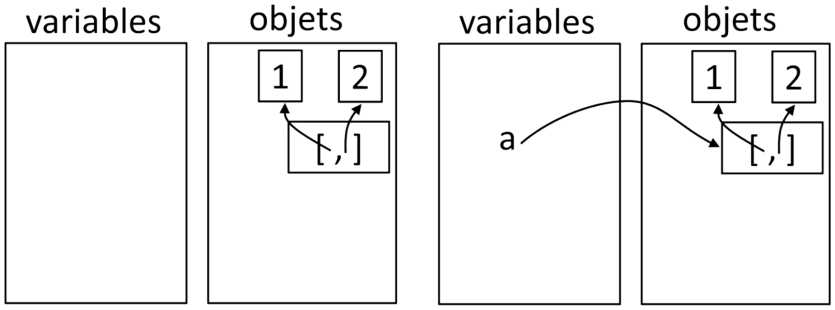


Nous voyons donc qu'à un moment de l'exécution du code, les variables **a** et **b** référençaient toutes les deux l'objet entier **3**. C'est ce qu'on appelle **une référence partagée**.

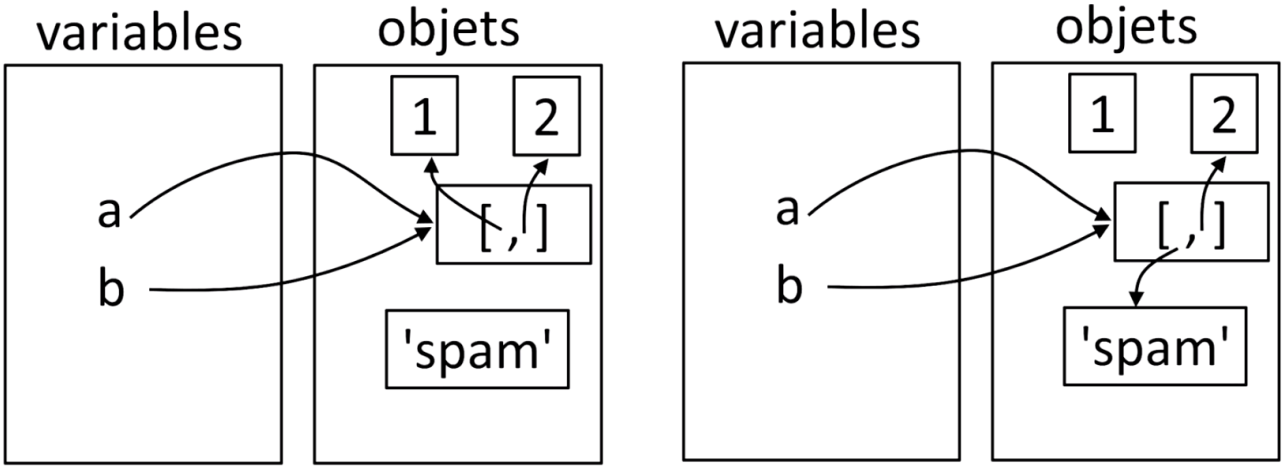
Une référence partagée est une référence qui est partagée par plusieurs variables. Lorsque cette référence est faite vers un objet immuable, comme c'est le cas d'un entier, il n'y a aucun effet de bord possible. Par contre, si l'objet référencé est un objet mutable, il y a un risque d'effet de bord qu'il faut comprendre et être capable de maîtriser.

L'effet de bord

Prenons maintenant une liste : **a = [1, 2]**. Nous créons un objet **list** qui va lui-même référencer un entier **1** et un entier **2**. On ne montre plus ici le compteur de référence, mais il faut savoir que la référence de l'objet **list** vers l'entier **1** va augmenter le compteur de référence l'entier **1**. Le compteur de référence n'est donc pas seulement augmenté / incrémenté lorsqu'on référence l'objet par une variable. Il est aussi incrémenté lorsque l'objet est référencé par un autre objet. Enfin, on crée la variable **a** qui va référencer l'objet **list**.



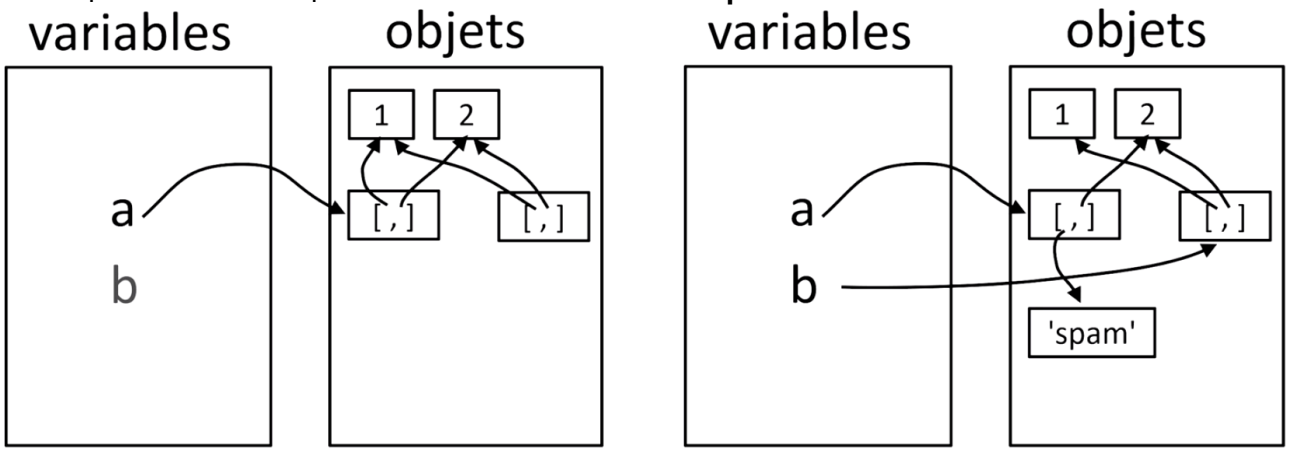
Nous ajoutons maintenant l'affectation **b = a**. Donc on crée une référence partagée : la variable **b** référence le même objet que la variable **a**. Puis, nous faisons **a[0] = 'spam'**. Cette opération va dire : crée l'objet **'spam'**, ensuite la case 0 de la liste référencée par **a** doit maintenant référencer l'objet **'spam'**. L'objet **list** ne référence plus l'entier **1** pour référencer la chaîne de caractère **'spam'**. On a une référence partagée : **a** référence l'objet **list**, **b** référence l'objet **list**. L'objet **list** est mutable. Il a été modifié par **a** et **b** voit également cet objet modifié.



C'est ce qu'on appelle **un effet de bord**. Cette notion d'effet de bord c'est lorsqu'on a une référence partagée vers un objet mutable, si cet objet est modifié par une variable, alors l'autre variable va voir cet objet modifié.

La shallow copy VS l'effet de bord

On crée une liste : **a = [1, 2]** et **b = a[:]**. Un slice vide fait une copie, en fait une **shallow copy** c'est-à-dire une copie superficielle. Cette copie superficielle implique ici que l'on crée un nouvel objet **list** mais que cet objet **list** va référencer les mêmes objets que ceux référencés par l'objet **list** original. Et **b** référence ce nouvel objet **list**.
Si maintenant nous faisons **a[0] = 'spam'**, on crée toujours notre chaîne de caractère **'spam'** mais **a[0]** c'est la case de la liste initiale. L'entier **1** n'est donc plus référencé par cet objet **list** initial qui référence à la place la chaîne de caractères **'spam'**.



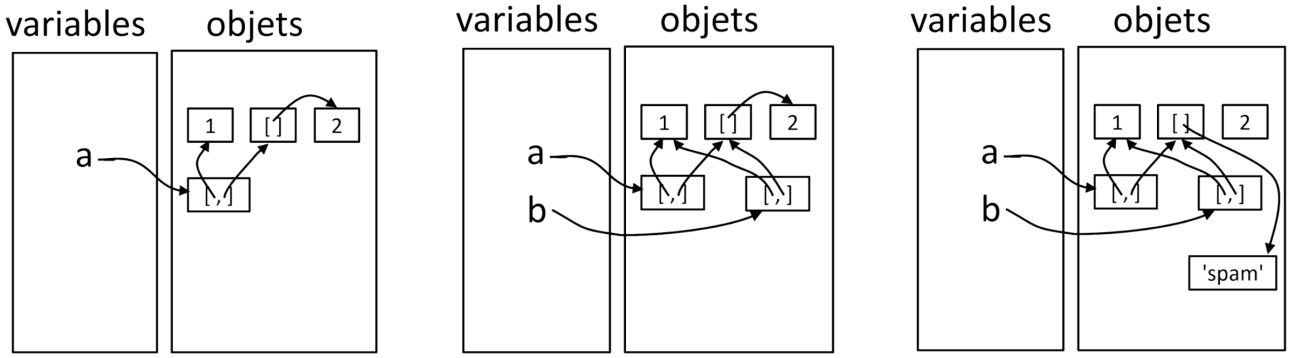
Nous avons donc été capable de supprimer l'effet de bord en faisant une copie de la liste au moment du partage de référence. **a = ['spam', 2]** et **b = [1, 2]**.

La deep copy

Lorsqu'on fait la **shallow copy**, on fait juste une copie superficielle de l'objet **list**. Par contre toutes les références sont partagées. Donc si l'objet **list** référence un objet qui est lui-même mutable comme : **a = [1, [2]]**, alors la **shallow copy** ne suffira pas à empêcher l'effet de bord.

Prenons donc **a = [1, [2]]**. On a un objet **list** qui référence un entier **1** et qui référence un autre objet **list** qui lui-même référence un entier **2**. Et la variable **a** référence l'objet **list** principal. Nous créons ensuite une variable **b = a[:]** qui référence donc une **shallow copy** de l'objet **list**. On a

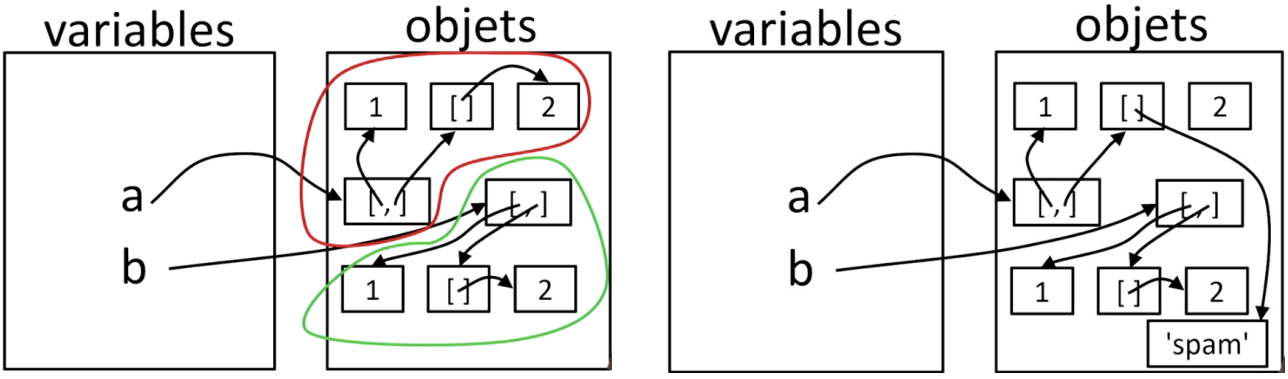
donc une liste dupliquée qui référence les mêmes objets que la liste originale.
Si ensuite, nous faisons **a[1][0] = 'spam'**, on supprime la référence à l'entier **2** dans la « sous-liste » de la liste principale pour le remplacer par une référence à la chaîne de caractère **'spam'**. Or comme cette « sous-liste » est référencée à la fois par **a** et par **b**, alors **a = [1, ['spam']]** et **b = [1, ['spam']]**.



Donc dans le cas d'un objet mutable qui référence des mutables, la **shallow copy** n'est pas suffisante. Il faut en réalité procéder à une **deep copy**, c'est-à-dire une copie profonde qui va copier de manière récursive tous les objets référencés.

Pour **a = [1, [2]]**, on recrée un objet **list** qui référence un entier **1** et un sous-objet **list** qui référence un entier **2**. Et la variable **a** référence l'objet **list** principal. Ensuite, on importe le module **copy** et faire **b = copy.deepcopy(a)**. Le module **copy** permet d'accéder à une méthode **.deepcopy()** qui fait une copie profonde de tous les éléments référencés par **a**. Le résultat, c'est que l'on crée un nouvel objet **list** qui référence un nouvel objet 1 et un nouveau sous-objet **list** qui référence un nouvel objet **2**. Et **b** référence le nouvel objet **list** principal. Nous avons maintenant deux groupes d'objets totalement différenciés. Et si on veut modifier l'objet référencé par le sous-objet **list** de la liste référencée par **a** avec **a[1] [0] = 'spam'**, on crée une chaîne de caractère **'spam'**, on enlève la référence entre le **2** et le sous-objet **list** de la liste référencée par **a**. Et maintenant **a = [1, ['spam']]** et **b = [1, [2]]**.

```
>>> a = [1, [2]]
>>> import copy
>>> b = copy.deepcopy(a)
```



Un élément central en Python

En Python, tout fonctionne avec des références partagées, notamment le passage d'argument aux fonctions. En Python, tout est un objet. Par conséquent tout est un surcoût mémoire. Il est très important de minimiser le nombre de copies des objets en mémoire. Et c'est encore les références partagées qui sont à la manoeuvre.

Certains objets sont immuables. Et Python va optimiser l'utilisation de ces objets immuables. Par exemple, les petits entiers, les petites chaînes de caractères ou certains types de tuples vont tous être des singletons. Ça veut dire que ce sont des objets qui ne vont exister qu'en un seul exemplaire en mémoire. Et on ne va utiliser que des références partagées à chaque fois qu'on aura besoin d'accéder à ces objets. C'est de nouveau un moyen de minimiser le nombre d'objets en mémoire.

Leçon 18 : Introduction aux classes

Nous avons vu jusqu'à maintenant les principaux types built-in : les chaînes de caractères, les séquences avec les listes et les tuples, les dictionnaires, les sets. Ces différents types built-in, bien qu'étant de natures différentes, se manipulent de manière extrêmement proche. Connaître la longueur d'une séquence, d'un set ou d'un dictionnaire se fait avec **len()**, afficher le contenu de ces objets se fait avec **print()**, le test d'appartenance se fait avec **in**.

Cette uniformité dans le comportement des objets est une caractéristique majeure de Python. Ce qui explique la faible courbe d'apprentissage que l'on a en Python. Lorsqu'on sait faire quelque chose avec un type d'objet, on sait le faire avec quasiment tous les autres types d'objets.

Mais Python va bien au-delà de cette puissance des types built-in. En Python, on peut écrire nos propres objets, définir nos propres objets qui vont se comporter exactement comme des types built-in. Donc on peut créer des objets qui vont supporter le test d'appartenance avec **in**, supporter les fonctions **print()** et **len()** ou encore l'affectation avec les crochets.

Dans cette leçon, nous allons découvrir le mécanisme qui nous permet d'implémenter nos propres objets. C'est ce qu'on appelle des **classes**. Avec une classe on peut créer des objets et définir des méthodes pour que nos objets se comportent exactement comme des types built-in.

Lorsque nous avons découvert la notion d'objet, dans la leçon 1, nous avions pris l'exemple d'une usine qui fabrique les objets. Cette usine représentait le type. Et en fait le type, c'est la classe. La classe est donc l'usine qui va créer nos **instances** et les instances ce sont les objets qui vont être produits.

En faisant une analogie avec les types built-in : **list** c'est le type qui va créer les objets **list**. Et tous les objets **list** que l'on produit sont des instances du type **list**.

Fonctionnement des classes

On crée une classe avec l'instruction **class** suivie du nom de la classe. Ci-dessous une classe qui ne fait rien :

```
class C:
    pass
```

Nous avons maintenant une classe **C** que l'on peut appeler avec des parenthèses **C()**. Lorsqu'on appelle **C()**, on crée des instances. Ainsi, en entrant : **c1 = C()** on crée une première instance, puis en entrant **c2 = C()** on crée une deuxième instance de cette classe. **c1** et **c2** sont deux objets distincts puisqu'ils sont situés à des adresses mémoires différentes.

Comme cette classe n'a défini aucune méthode, on n'a aucun comportement spécifique avec notre classe. Donc on a essentiellement une classe inutile, qui ne fait rien.

Ajouter des méthodes aux classes

La première méthode que l'on veut ajouter à notre classe c'est **l'initialisateur** ou **le constructeur de nos instances**. On crée donc une méthode **def** qui s'appelle **__init__** et qui va prendre deux arguments **self** et **phrase**. Et on renomme la classe, en **Phrase** pour avoir un nom un peu plus explicite.

```
class Phrase
    def __init__(self, phrase):
        self.mots = phrase.split()
```

Nous avons donc défini une fonction avec **def** à l'intérieur d'une classe. Les fonctions définies dans les classes s'appellent des **méthodes**. Et cette méthode prendre deux arguments : **self** et **phrase**. En fait lorsque l'on va appeler la méthode sur la classe, l'instance va automatiquement être passée comme premier argument. Donc **self** va correspondre à l'instance créée par la classe dans la méthode **__init__** qui est appelée lorsqu'on crée notre instance. Et l'argument que l'on passe lors de la création de cette instance est passé à **phrase**.

Pour utiliser la classe, on entre la commande suivante :

p = Phrase(‘je fais un mooc sur Python’)

Lorsque l’on fait un retour chariot, la classe **Phrase** va appeler la méthode _**init**_. La chaîne de caractères qu’on a passé ici **‘je fais un mooc sur Python’** va être passée au deuxième argument de la méthode _**init**_. Et cette chaîne de caractères va perte découpée par **.split()** dans une liste. Et cette liste est affectée à **self.mots**. **self** c’est notre instance, c’est donc ici **p**. Donc **self.mots** va être un attribut **mots** dans notre instance.

De telle sorte qu’après avec exécuté la classe, notre instance **p** a désormais un attribut **.mots** : **p.mots** qui contient la liste des mots contenus dans la chaîne de caractères passée à **Phrase**.

p.mots
renvoie **[‘je’, ‘fais’, ‘un’, ‘mooc’, ‘sur’, ‘Python’]**

On est donc capable de créer des instances qui héritent d’un comportement de la classe **Phrase**, donc du type qui a été utilisé pour créer l’instance. Donc ici, le comportement est basique. La phrase que l’on passe est automatiquement mise dans une liste qui est référencée par l’attribut **.mots** de cette instance.

Maintenant on pourrait vouloir ajouter des méthodes à notre instance. Imaginons une méthode qui mette tous les mots en majuscule :

```
class Phrase
   def __init__(self, phrase):
      self.mots = phrase.split()

   def upper(self):
      self.mots = [m.upper() for m in self.mots]
```

Donc lorsque l’on va appeler **.upper** sur notre instance, on va récupérer l’attribut **mots** de l’instance, on va dire que cet attribut **.mots** référence maintenant une nouvelle liste qui prend tous les mots en majuscule.

Donc on refait notre commande :

p = Phrase(‘je fais un mooc sur Python’)

puis **p.mots** qui renvoie la liste **[‘je’, ‘fais’, ‘un’, ‘mooc’, ‘sur’, ‘Python’]**

Et on applique la méthode **p.upper()**

Et maintenant **p.mots** renvoie **[‘JE’, ‘FAIS’, ‘UN’, ‘MOOC’, ‘SUR’, ‘PYTHON’]**

La méthode sur l’instance a donc été capable de modifier un attribut de l’instance.

Si nous faisons un **print(p)**, Python affiche l’adresse de l’objet. Or ce n’es pas ce que l’on veut. On voudrait le contenu de la liste. Pour cela, il suffit d’implémenter une nouvelle méthode _**str**_ qui va faire un **return** d’une chaîne de caractères. Et la chaîne de caractères que l’on va afficher ici va simplement être **“\n“.join(self.mots)** ce qui va regrouper tous les mots dans une chaîne de caractères et faire un retour chariot à chaque mot, donc afficher les mots sur une colonne.

```
class Phrase
   def __init__(self, phrase):
      self.mots = phrase.split()

   def upper(self):
      self.mots = [m.upper() for m in self.mots]

   def __str__(self):
      return “\n“.join(self.mots)
```

On refait la commande **p = Phrase(‘je fais un mooc sur Python’)**. Et si maintenant on fait **print(p)**, **print** va automatiquement appelé la méthode _**str**_ sur l’instance et va donc afficher les mots dans une colonne.

Leçon 19 : Fonctions

Dans cette leçon, nous allons revenir sur le fonctionnement des fonctions (introduit dans la leçon 8) et voir comment définir une fonction.

Comment définir une fonction

Une fonction se définit avec l'instruction **def** suivi du nom de la fonction à laquelle on passe des arguments. On peut passer un nombre quelconque d'arguments séparés par une virgule à une fonction.

```
def f(a, b, c):
    print(a, b, c)
```

Donc on a là une fonction **f** qui va simplement afficher les trois arguments **a**, **b**, **c**. Lorsque ce bloc de code est évalué, on crée un objet fonction et le nom de la fonction **f** va être une variable qui va référencer l'objet fonction. Comme tous les objets en Python et comme toutes les variables en Python, une variable est simplement un nom qui référence un objet. On peut donc renommer notre objet fonction avec une autre variable en faisant **une référence partagée**. On peut donc écrire **g = f**. Ce qui fait que nous avons maintenant une variable **g** qui référence le même objet fonction. On peut donc appeler la fonction à partir de la variable **f** ou à partir de la variable **g** : **f(1, 2, 3)** renvoie **1 2 3** et **g(1, 2, 3)** renvoie **1 2 3**.

En Python, tout est un objet. Cela implique un coût mémoire important mais comme nous l'avons vu dans la leçon 18, le mécanisme de référence partagée permet de minimiser les copies des objets. Python ne copie jamais d'objet sauf si cela est demandé de manière explicite.

Fonctions et effet de bord

Lorsque l'on passe des arguments à une fonction (donc lorsque l'on passe des objets à une fonction), ces objets ne sont jamais copiés, ils sont toujours passés par référence. Prenons une liste vide **L = [ ]** et définissons une fonction :

```
def add_1(a):
    a.append(1)
```

La fonction prend un argument **a** et appelle la méthode .**append()** sur cet argument. On suppose donc que cet argument est une liste. Et on va lui ajouter **1**. Si on applique cette fonction sur notre liste **L** : **add_1(L)**, notre liste **L = [1]**. À aucun moment nous n'avons réaffecté **L**. À aucun moment nous n'avons fait de retour. Pourtant la liste a été modifié. En fait la liste **L** a été modifiée par **effet de bord** parce que nous avons une référence partagée vers un objet mutable.

Les références partagées sont : **L** référence un objet list, cet objet list a été passé à notre fonction **add_1()** et notre fonction a une variable locale **a** qui référence le même objet **list** que la variable **L**. Lorsqu'on modifie l'objet mutable par la méthode **.append()**, on modifie l'objet partagé, donc lorsque la fonction **add_1()** retourne, la variable **L** référence l'objet qui a été modifié.

Ce comportement par effet de bord peut être un comportement tout à fait souhaitable. Son intérêt est d'être extrêmement économe en terme de mémoire. Son inconvénient, en revanche, c'est qu'il est fait de manière implicite. Par conséquent, comme on veut qu'en Python tout soit fait de manière explicite, on doit extrêmement bien documenter son code.

Regardons l'exemple de la méthode **.sort()** sur la liste. Rappel : la méthode **.sort()** sur les listes permet de faire un tri en place. L'intérêt c'est que c'est très économe au niveau mémoire. L'inconvénient c'est que c'est fait par effet de bord.

```
In [1]: help(list.sort)
Help on method_descriptor:

sort(...)
    L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
```

L'aide de la méthode **.sort()** nous dit que l'on fait un tri mais que ce tri est fait en place (***IN PLACE***). Il n'y a donc aucune ambiguïté, le retour de **.sort()** c'est **None** (l'objet vide) et la liste est modifiée en place.

Définir une fonction de manière explicite

Maintenant, on pourrait dire que l'on ne veut pas modifier notre liste **L** et donc passer une copie à notre fonction **add_1()** : **add_1(L[:])**. Comme prévu, lors du retour, notre liste **L** n'a pas été modifiée : **L = [1]** mais la **shallow copy** a été perdue. Donc pour récupérer une référence vers cet objet **list**, il faut faire une valeur de retour dans la fonction.

```
def add_1(a):
    a = a[:]
    a.append(1)
    return a
```

Et si maintenant, on fait **add_1(L)** la fonction renvoie **[1, 1]**. La valeur de retour est une nouvelle liste qui vaut **[1, 1]** et la liste **L** originale n'a pas été modifiée : **L = [1]**. Et si maintenant on veut modifier notre liste originale, on le fait de manière explicite : **L = add_1(L)**. Et maintenant tout est explicite : on passe la liste **L** à notre fonction dont le nom laisse entendre qu'elle va lui ajouter **1** et la valeur de retour est réaffectée à la liste **L** dont la valeur de retour est désormais la nouvelle liste **[1, 1]**.

Les fonctions imbriquées

Lorsque l'on écrit le code d'une fonction, par exemple dans un module et que l'on importe le module. Lors de l'importation du module, l'objet fonction va être créé et le nom de la fonction va être une variable qui va référencer cet objet fonction. Par contre, le bloc de code de la fonction ne sera évalué que lors de l'appel de la fonction.

```
def f(a):
    func(a)
```

Dans ce code, nous avons une variable **f** qui référence une fonction. Par contre la variable **func** n'existe pas. Nous n'avons pas encore d'objet fonction. L'absence de définition de la fonction **func**, on ne le verra que lors de l'appel de la fonction **f**. En entrant **f(1)** Python renvoie une erreur **NameError** qui dit clairement que le nom **func** n'a pas été défini. On peut donc définir une fonction qui appelle du code qui n'a pas encore été défini et que c'est possible jusqu'au moment où on appelle effectivement notre fonction. Si maintenant nous définissons notre fonction **func** :

```
def func(a):
    return a
```

On peut dès lors appeler notre fonction **f** qui va appeler **func** : **f(1)** fait un retour de **1**.

Le polymorphisme

Polymorphisme est un mot étrange pour un concept très simple.

```
def my_add(a, b):
    print( f"{a} et {b}" )
    return a + b
```

On a donc une fonction qui prend deux arguments **a** et **b** qui va les afficher et qui va faire un retour de la somme de ces arguments. À aucun moment nous n'avons spécifié le type. On peut donc appeler notre fonction **my_add** avec des **int**, des **float** et des **str**. Dans le cas **str** la fonction va concaténer les arguments.

La caractéristique de la notion de polymorphisme c'est qu'une fois qu'on a défini une fonction, cette fonction va pouvoir s'exécuter sur n'importe quel type compatible avec les opérations contenues dans le bloc de code de la fonction. L'intérêt de polymorphisme c'est qu'on réduit énormément le code que l'on a écrire puisqu'on n'a pas besoin d'écrire une fonction pour les entiers, une fonction pour les float et une fonction pour les chaînes de caractères. Notre fonction va être unique et va pouvoir se comporter correctement avec n'importe quel type qu'on lui passe du moment que les opérations définies dans la fonction sont définis pour les types qui sont passés à cette fonction.

Grâce au typage dynamique, nous n'avons jamais à définir le type des objets que nous passons aux fonctions. Cependant Python permet de donner des indications de type. C'est ce qu'on appelle **les type hints.** Les type hints sont uniquement des indications qui peuvent être utilisées par exemple pour améliorer la documentation du code alors pour effectuer une validation statique du code. Cependant, l'auteur de Python a été très clair sur cette notion : les type hints resteront toujours optionnels et ne nous obligeront jamais à définir du typage statique dans notre code.

Leçon 20 : Tests if / elif / else et opérateurs booléens

Syntaxe et fonctionnement des test if / elif / else

if test1:
    <bloc d'instructions 1>

Le bloc d'instruction ne sera exécuté que si le test est vrai. Ensuite, on peut avoir de manière optionnelle un ou plusieurs **elif** avec un ou des blocs d'instructions :

if test1:
    <bloc d'instructions 1>
elif test2:
    <bloc d'instructions 2>
elif test3:
    <bloc d'instructions 3>
…

L'intérêt de ces **elif** c'est de faire des tests supplémentaires. Le fonctionnement de cette suite de tests est simple. On commence avec le premier **if**. Si le **test1** est **True**, le bloc d'instruction est exécuté et on sort du **if else**. Si le **test1** est **False**, on passe au **test2**. Si le **test2** est **True** le bloc d'instruction est exécuté. Si **test2** est **False**, on passe au **test3** et on continue de manière successive jusqu'au dernier test.

Si l'intégralité des tests sont faux. On peut avec le **else** optionnel faire une exécution d'un bloc d'instruction.

if test1:
    <bloc d'instructions 1>
elif test2:
    <bloc d'instructions 2>
elif test3:
    <bloc d'instructions 3>
…
else:
    <bloc d'instructions n>

En résumé, si **test1** est **True** on exécute le bloc d'instruction correspondant. Si **False** on passe à **test2**, si **False** on passe à **test3**. Et si aucun des tests n'est **True**, on finit dans **else** et on exécute le bloc d'instruction de **else**.

Donc dans une structure **if / elif / else**, un seul bloc d'instruction sera exécuté.

Ce que peut contenir un test

Dans un test d'un **if** ou d'un **elif**, on peut avoir n'importe quelle expression. Le test va appeler la fonction built-in **bool** sur le résultat de l'évaluation de l'expression. Donc on a une expression qui va être exécutée. Elle va produire un **objet** et on va appeler **bool** sur cet objet.

bool(objet) va appeler :

- soit la méthode **objet._bool_()** qui est une méthode spéciale sur laquelle nous reviendrons dans les leçons sur les classes. Cette méthode **._bool_()** va retourner **True** ou **False**.
- soit la méthode **objet._len_()**. Si la méthode **._len_()** retourne **0** ce sera **False**, si la méthode **._len_()** retourne quelque chose d'autre, ce sera **True**. L'intuition derrière ça c'est qu'un objet vide est considéré comme **False**. Un objet qui n'est pas vide est considéré comme **True**.

Exemples d'expressions

1) Un type built-in :

Un type **built-in** :

- sera considéré comme **False** s'il est **False**, **0**, **None** ou **n'importe quel type liste, tuple, dictionnaire chaîne de caractères vides**.
- tout le reste est **True**.

Exemple avec le dictionnaire **d = { 'marc' : 10 }** :

```
If d:
    print(d)
```

Si le dictionnaire **d** était vide, le **print(d)** ne s'exécuterait pas. S'il y a quelque chose dans le dictionnaire **d**, le bloc d'instruction va être exécuté.

2) Une comparaison :

On peut aussi mettre des comparaisons : supérieur **>**, supérieur ou égal **>=**, inférieur **<**, inférieur ou égal **<=**, égal **==** ou différent **!=**.

```
If a != b:
    print('faux')
```

Si **a** est différent de **b**, le bloc d'instruction **print('faux')** va être exécuté.

3) Le test d'appartenance :

Le test d'appartenance **in** :

```
If 'a' in 'marc':
    print('ok')
```

Si la lettre **'a'** est dans la chaîne de caractère **'marc'**, le bloc d'instruction **print('ok')** est exécuté.

4) Un retour de fonction :

On peut utiliser un retour de fonction. C'est-à-dire que l'on va évalué l'objet retourné par l'appel d'une fonction.

```
s = '123'

if s.isdecimal():
    print( int(s) + 10 )
```

isdecimal() va retourner un booléen **True** ou **False**. Ici la fonction renvoie **True**, le bloc d'instruction **print(int(s) + 10)** va donc être exécuté.

5) Les opérateurs de test booléen :

On peut aussi utiliser les opérateurs booléens **and** / **or** / **not** pour combiner plusieurs conditions.

- **A and B** est **True** si **A** et **B** sont **True**
  *Si **A** est **False**, on sait que le **A and B** sera **False**, donc on évalue pas le **B** (short-circuit)*
- A or B est True si soit A soit B est True
  *Si **A** est **True**, dans le **A or B** on évaluera pas le **B** (short-circuit)*
- **not A** est **True** si **A** est **False**

Exemple :

```
s = '123'

If '1' in s and s.isdecimal():
    print( int(s) + 10 )
```

On regarde si **1** est dans notre chaîne de caractère et que **s** est décimal, alors le bloc d'instruction **print(int(s) + 10)** est exécuté.

Leçon 21 : Boucles while

Nous avons vu l'instruction **if** qui permet d'évaluer des tests et nous avons vu également la boucle **for** qui permet de répéter un bloc d'instructions un certain nombre de fois. Nous allons voir dans cette leçon l'instruction **while** qui permet de combiner ces deux possibilités.

Fonctionnement d'une boucle while

```
a = list( range(1, 10) )

while a:
    a.pop()
    print(a)
```

Ici **while** est l'instruction et on met un test à côté qui est exactement de même nature qu'avec un **if**. Donc tant que ce test est vrai, on continue à boucler et lorsque le test devient faux, on sort du bloc de code du **while**. Ici, notre code va ainsi boucler tant que **a** est **True**. **a** étant une liste, on sait qu'une liste est **True** tant qu'elle contient des éléments et elle devient **False** quand elle est vide. Donc **a.pop()** va enlever le dernier élément de la liste **a** et lorsqu'il n'y aura plus d'élément dans la liste **a**, le test va naturellement s'arrêter.

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
[1, 2, 3]
[1, 2]
[1]
[]
```

Instructions break et continue

Dans un **while** on peut également mettre des instructions **break** et **continue** qui permettent d'interrompre l'exécution dans **while** ou alors de remonter au début.

```
a = list( range(1, 10) )

while a:
    a.pop()
    if len(a) == 5:
        continue
    print(a)
```

Lorsque la liste aura exactement 5 éléments, on va arriver à l'instruction **continue** et on va remonter au début du **while**. Cela veut donc dire que l'on va sauter un **print(a)**.

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4]
[1, 2, 3]
[1, 2]
[1]
[]
```

On constate que la liste **[1, 2, 3, 4, 5]** n'a pas été imprimée. Comme elle contient exactement 5 éléments, **continue** a été activé et on est remonté en haut du **while** en sautant le **print(a)**.

On peut également avoir un **break** qui va sortir du bloc d'instruction, donc du **while** dès qu'il est rencontré.

```
a = list( range(1, 10) )

while a:
    a.pop()
    if len(a) == 5:
        break
    print(a)
```

Ce que dit le code c'est qu'à partir du moment où la liste comporte 5 éléments, on fait un **break** donc on arrête le **while**.

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6]
```

Et voit effectivement que le **while** s'est interrompu quand la liste est arrivée à 5 éléments.

While True

L'intérêt de faire un **while True** c'est qu'on fait une boucle infinie. Et on sortira de cette boucle en fonction d'un certain critère de test avec un **break**.

```
while True:
    s = input('Quelle est votre question ?\n')
    if 'aucune' in s:
        break
```

Ce **while** va faire une boucle infinie. Il va en permanence nous demander **'Quelle est votre question ?'** et lorsque notre réponse contiendra '**aucune'**, on sortira du **while**.

Leçon 22 : Portée des variables - Règle LEGB

Nous avons parlé de la notion de bloc de code, par exemple avec les instructions **for** et **if**. Un bloc de code est un ensemble d'instructions contigües, indentées du même nombre de caractères vers la droite. Lorsque l'on fait une opération d'affectation : par exemple **x = 1**, on dit que l'on définit la variable **x**, cela veut dire qu'une variable référence un objet. Il y a plusieurs synonymes : **définition**, **affectation**, **assignation**, **binding**. Ces termes sont utilisés de manière inter-changeable.

La portée d'une variable détermine de quel endroit du code on peut accéder à cette variable. Python utilise **la portée lexicale**. Cela veut dire que la portée d'une variable est déterminée en fonction de l'endroit dans le code où cette variable est définie.

Une variable locale au bloc de code d'une fonction est **une variable dite locale**. Lorsque la fonction retourne, toutes les variables locales de la fonction sont détruites.

Une variable définie en-dehors de toute fonction est **une variable globale**.

Il y a donc principalement deux catégories de variables : les variables locales qui sont définies dans les blocs de fonction et les variables globales qui sont définies en dehors de toute fonction.

La règle LEGB

Il y a une règle qui s'appelle la règle **LEGB** qui veut dire que lorsque l'on référence une variable, on va d'abord chercher si elle a été définie localement à l'endroit où elle a été référencée. Donc lorsque l'on référence une variable dans une fonction, on regarde si cette variable a été définie localement à la fonction.

Si elle n'a pas été définie localement à cette fonction, on va aller la chercher dans **les fonctions englobantes**. Donc on va remonter de la fonction la plus proche (là où on a référencé la variable) jusqu'à la fonction la plus externe. Si on ne trouve pas cette variable définie dans les fonctions englobantes, alors on va la chercher **globalement**, c'est-à-dire au niveau **des variables globales**. Et pour finir, si on ne la trouve toujours pas, on la cherchera dans **le module builtins**.

La notion de portée de variable

Prenons trois variables **a**, **b**, **c** et une fonction **g** :

```
a, b, c = 1, 1, 1

def g( ):
    b, c = 2, 4
    b = b + 10
    def h( ):
        c = 5
        print(a, b, c)
    h( )
```

La question que l'on doit se poser c'est : qu'est-ce qui va se passer lorsqu'on va faire un **print(a, b, c)** ? Quelle variable **a** va être affichée ? Quelle variable **b** va être affichée ? Quelle variable **c** va être affichée ?

Pour la variable **a** : selon la règle **LEGB**, nous commençons par vérifier si **a** est défini localement dans la fonction **h()**. La réponse est non. Est-ce que **a** est défini dans les fonctions englobantes ? **g()** est une fonction qui englobe **h()**, c'est donc une fonction englobante. Mais non, **a** n'est pas défini dans la fonction englobante. Est-ce que **a** est défini globalement ? Oui et il se trouve qu'au moment où on exécute le code, la variable globale **a** référence l'entier **1**. Donc **print(a)** va afficher **1**.

Pour la variable **b** : est-ce que **b** est défini localement à la fonction **h()** ? Non. Est-ce que **b** est défini dans les fonctions englobantes ? Oui : la fonction englobante **g()** définit la variable **b**. **b = 2** et ensuite **b = b + 10**. Donc au moment où on recherche la variable **b**, elle référence l'entier **12**. Donc **print(b)** va afficher **12**.

Pour la variable **c** : est-ce que **c** est défini localement à la fonction **h()** ? Oui : **c** est définie dans **h()** et **c = 5**.

Donc **g()** renvoie **1 12 5**

En résumé, nous voyons qu'une variable définie dans une fonction devient locale à cette fonction. Ça veut dire qu'elle peut-être vue dans les fonctions qui sont englobées. Par contre une variable locale à une fonction ne peut pas être vue à l'extérieure de cette fonction.

Une variable globale, définie au niveau du module, peut en revanche être vue par toutes les fonctions qui sont définies dans ce module.

Le module builtins

Si après avoir exécuté notre fonction **g()**, nous tapons **print(a, b, c)**, Python renvoie **1 1 1**, puisque les variables **a**, **b**, **c** sont définies globalement et que **print()** est demandé depuis le contexte global du module.

Mais en fait lorsqu'on exécute **print(a, b, c)**, nous avons en réalité 4 variables : les variables **a**, **b**, **c** et la variable **print()**. **print()** est une variable qui référence un objet fonction. En fait, toutes les fonctions directement définies dans **le module builtins** sont directement accessibles sans avoir à importer **le module builtins** grâce à cette règle **LEGB**. En dernier ressort, si on ne trouve pas un nom de variable, on le cherche dans **le module builtins** pour pouvoir, si cette variable est définie dans **builtins**, appeler la fonction correspondante.

On peut importer **le module builtins** manuellement avec l'instruction **import builtins** et on peut regarder tout ce que contient **le module builtins** avec **dir(builtins)**. On y retrouve tout un ensemble de fonctions déjà vues : **min()**, **tuple()**, **type()**, etc.

Il est important de comprendre que ces fonctions **builtins** sont référencées par des variables et qu'une fonction c'est un objet et qu'une variable c'est un nom qui référence un objet. Par conséquent, on peut redéfinir des fonctions **builtins**.

Ainsi, si on fait **print(1)**, on va chercher **print()** dans **le module builtins**. Par contre si on fait **print = 10**, on définit une variable **print** qui référence l'entier **10**. Et donc si maintenant on fait un **print(1)**, Python renvoie une exception **TypeError**. Pour récupérer la référence vers le bon objet fonction, on peut entrer : **print = builtins.print** après quoi **print()** fonctionne de nouveau normalement.

La toute puissance de la variable globale

Il n'y a rien de supérieur à une variable globale. Chaque module va définir ses propres variables globales. Cela est rendu possible grâce à un mécanisme d'isolation qui s'appelle **espace de nommage**. Chaque module va définir son propre **espace de nommage** et les variables définies dans **l'espace de nommage** d'un module sont des variables globales.

Leçon 23 : Modification de la portée avec global et nonlocal

Dans cette leçon, nous allons voir comment modifier la portée des variables avec les instructions **global** et **nonlocal**.

global permet de rendre une variable locale de portée globale.

nonlocal permet de rendre une variable locale de portée locale dans une fonction englobante.

Le fonctionnement de global

La méthode implicite :

```
a = 'a globale'

def f():
    a = 'a dans f'
    print(a)
```

Si on fait maintenant **print(a)** Python renvoie **a globale**. Si on exécute la fonction **f()**, elle définit une variable locale **a** et va faire un **print(a)** qui renvoie **a dans f**. Mais si on refait un **print(a)**, la Python renvoie de nouveau **a globale**.

La question que l'on peut se poser est donc : comment modifier une variable globale depuis une fonction ? Pour cela on peut utiliser l'instruction **global**.

```
a = 'a globale'

def f():
    global a
    a = 'a dans f'
    print(a)
```

Et maintenant, si on exécute **print(a)**, Python renvoie **a global**. Si on exécute **f()**, la fonction renvoie **a dans f**. Et si maintenant, on refait **print(a)** après l'exécution de la fonction **f()**, Python renvoie **a dans f**.

Nous avons défini **global a** dans la fonction **f()**, et ainsi désormais, l'interpréteur Python considère que la variable **a** qui va être utilisée dans la fonction n'est plus une variable locale, c'est la variable globale **a**. Donc ici, ce n'est même pas une référence partagée. On accède directement à la variable définie dans le module. C'est donc une manière, depuis une fonction, de modifier directement des variables globales.

Cette notation a cependant un inconvénient majeur. C'est que lorsqu'on définit une variable globale à l'intérieur d'une fonction, on fait une modification implicite d'une variable globale définie dans le module. Or, en Python on n'aime pas les modifications implicites.

La méthode explicite :

```
a = 10

def f():
    global a
    a = a +10
```

Si on exécute **print(a)**, Python renvoie **10**. Si on appelle la fonction **f()**, elle travaille sur la variable globale et fait **a = a +10**, **a** vaut donc **20** à la sortie de la fonction **f()**. Et si maintenant on fait **print(a)**, Python renvoie **20**.

Seulement ce code a tout faux ! Premièrement, les noms de variable sont très mal choisis. Il est très difficile de comprendre explicitement ce que veut faire ce code. Deuxièmement, l'accès à la variable est fait de manière implicite et la modification de la variable globale est faite de manière implicite également.

La bonne manière de manipuler une variable globale est la suivante :

```
note = 10

def add_10(n):
    return n + 10
```

Maintenant, si on veut modifier notre variable globale **note**, nous n'avons qu'à écrire :

```
note = add_10(note)
```

Cette notation dit : on prend notre variable globale **note** et on lui affecte un retour de fonction. La fonction s'appelle **add_10()** et elle prendre comme paramètre **note**. On voit donc maintenant de manière très explicite que **note** va avoir le résultat de **add_10(note)**. **note** vaut maintenant **20**.

Cette notion d'instruction **global** est donc très trompeuse puisqu'elle rend toutes les modifications implicites et qu'il vaut bien mieux travailler avec des retours de fonction pour contrôler de manière explicite le changement de nos variables globales.

Le fonctionnement de nonlocal

nonlocal sert à modifier la portée d'une variable locale pour accéder à une variable également locale mais dans une fonction englobante.

```
a = 'a global'

def f():
    a = 'a de f'
    def g():
        a = 'a de g'
        print(a)
    g()
    print(a)
```

Sans surprise, la fonction **f()** commence par renvoyer le **print(a)** de la fonction **g()** donc **'a de g'** puis le **print(a)** de la fonction **f()** qui renvoie **'a de f'**. Enfin, si après avoir exécuté la fonction **f()**, on fait un **print(a)**, Python renvoie la variable global **a = 'a global'**.

La question que l'on peut donc se poser : c'est comment modifier la variable locale à **f()** depuis **g()** ? La réponse est dans l'instruction **nonlocal** :

```
a = 'a global'

def f():
    a = 'a de f'
    def g():
        nonlocal a
        a = 'a de g'
        print(a)
    g()
    print(a)
```

Nous rajoutons dans **g()** l'instruction **nonlocal a** qui va dire que maintenant la variable **a** définie dans **g()** n'est plus une variable locale de **g()**, c'est la variable locale définie dans **f()**. On peut directement modifier la variable locale **a** de **f()** directement depuis **g()**.

Ainsi, si on exécute la fonction **f()**, le **print(a)** de la fonction **g()** renvoie **'a de g'**. Et comme nous avons redéfinie la variable **a** de la fonction **f()** dans **g()**, donc maintenant si on fait un **print(a)** dans **f()** après avoir appelé **g()**, on voit que **a** de la fonction **f()** vaut maintenant **'a de g'**.

Leçon 24 : Passage des arguments et appel de fonctions

Lorsque l'on définit une fonction, par exemple :

```
def sum(a, b):
    return a + b
```

Les variables **a** et **b** sont **des paramètres** de la fonction. Et lorsqu'on appelle une fonction, comme par exemple :

```
x, y = 1, 2

sum(x, y)
```

Les variables **x** et **y** sont **les arguments** de la fonction.

Dans cette leçon nous allons voir les différentes manières de définir les paramètres d'une fonction et les différentes manières de passer des arguments à une fonction.

Paramètres ordonnés / paramètres par défaut & arguments ordonnés / arguments nommés

```
def agenda(nom, prénom, tel):
    return {'nom' : nom, 'prenom' : prenom, 'tel' : tel}
```

Pour appeler la fonction la méthode standard consiste à exécuter **agenda('idle', 'eric', '07070707')** qui nous renvoie un dictionnaire : **{'nom' : 'idle', 'prenom' : 'eric', 'tel' : '07070707'}**.

Si on choisit bien les noms des paramètres d'une fonction, il est beaucoup plus facile de se souvenir du nom des paramètres que de leur ordre. Python offre la possibilité d'appeler une fonction avec **des arguments nommés**.

Lorsqu'on appelle la fonction **agenda**, on peut donner explicitement le nom des paramètres auxquels on passe nos arguments : **agenda(tel = '07070707', nom='idle', prenom = 'eric')**. Nous avons ainsi passé nos arguments à notre fonction dans un ordre quelconque mais nous les avons nommé en fonction des paramètres définis dans notre fonction. Le retour est le même dictionnaire construit parfaitement : **{'nom' : 'idle', 'prenom' : 'eric', 'tel' : '07070707'}**.

On peut aussi définir des paramètres optionnels. Pour cela on suit le paramètre du signe égal et de la valeur qui sera passée à ce paramètre s'il n'est pas spécifié lors de l'appel de la fonction :

```
def agenda(nom, prénom, tel = '?'):
    return {'nom' : nom, 'prenom' : prenom, 'tel' : tel}
```

Si maintenant on exécute la fonction : **agenda('idle', 'eric')**, Python renvoie **{'nom' : 'idle', 'prenom' : 'eric', 'tel' : '?'}**. Si on avait passé un numéro de téléphone, l'argument optionnel n'aurait pas été utilisé.

Lorsqu'on définit l'entête d'une fonction, il est très important de mettre en premier les paramètres ordonnés et ensuite les paramètres optionnels.

Les formes étoiles * et ** pour les paramètres

Il existe en tout 4 manières de définir les paramètres d'une fonction et 4 manières de passer les arguments d'une fonction.

La forme étoile :

```
def f(*t):
    print(t)
```

L'intérêt de cette notation étoile, c'est qu'elle permet de passer une liste quelconque d'arguments à la fonction. Ces arguments vont être mis dans **un tuple** par la variable qui suit l'étoile, ici la variable **t**. Ainsi **f()** renvoie **()**, **f(1)** renvoie **(1,)**, **f(1, 2, 3, 4)** renvoie **(1, 2, 3, 4)**.

Si avec la fonction built-in **print()** on peut passer autant d'arguments qu'on le souhaite, c'est parce que dans **print** il y a un paramètre étoile.

La forme double étoile :

```
def f(**d):
    print(d)
```

La forme double étoile permet d'appeler la fonction sans aucun argument mais le **print** affiche désormais un **dictionnaire**. La forme double étoile permet de passer n'importe quel argument nommé à la fonction : **f(nom = 'idle', prenom = 'eric')** renvoie **{'nom' : 'idle', 'prenom' : 'eric'}**.

La forme double étoile est notamment utilisée lorsqu'on construit des **wrapper**.

Les formes étoiles * et ** pour les arguments

La forme étoile :

```
def f(a, b):
    print(a, b)
```

Si nous avons un objet liste **L = [1, 2]** que l'on souhaite appeler avec **f()**. La manière classique serait d'utiliser les indices : **f(L[0], L[1])**. Mais comme nous l'avons vu dans le notebook sur le **sequence unpacking**, la manipulation des indices n'est pas pythonique.

Pour passer la liste on peut en fait utiliser la forme étoile : **f(*L)**. Ici, la forme étoile va faire du **tuple unpacking** : elle va prendre chaque élément de la liste **L** et le **unpacké** vers les paramètres de la fonction. Donc avec **f(*L)**, **1** et **2** sont automatiquement passés à **a** et **b**. ATTENTION ! Il faut évidemment que l'on ait le même nom d'éléments dans la liste que l'on a de paramètres dans la fonction. Mais si on a le même nombre, le **unpacking** va être automatique.

La forme double étoile :

On peut directement passer **un dictionnaire** à la fonction **f()** avec une forme double étoile. Pour **d = {'a' : 1, 'b' : 2}**, si on appelle **f(**d)**, le dictionnaire **d** est directement passé à la fonction.

Cette forme double étoile a de nombreux usages, comme par exemple avec la fonction built-in **print()**. La fonction **print()** peut comporter des arguments nommés comme par exemple : **sep** qui permet de séparer les valeurs, **end** qui permet de définir une valeur de fin. Et ces arguments, si on veut les afficher systématiquement de la même manière, il faut les écrire à chaque fois dans **print()**. Exemple : **print(1, 2, sep = ';', end = 'FIN')** renvoie **1;2FIN**. Pour parer à ce travail laborieux, la forme double étoile permet de mettre directement ces arguments dans un dictionnaire pour directement les passer à notre fonction **print()**.

Exemple : nous définissons un dictionnaire **pp = {'sep' : ';', 'end' : 'FIN'}**. Et maintenant on peut écrire **print(1, 2, **pp)** qui renvoie **1;2FIN**. Les arguments ont été automatiquement passés à la fonction **print()**.

Leçon 25 : Itérable, itérateur, itération

Les itérateurs en Python sont des objets simples qui définissent une interface unique que l'on appelle **le protocole d'itération**. En plus de la simplicité et de l'efficacité de ce mécanisme, la notion **d'itérateur** permet de découpler l'objet qui itère de l'objet qui contient les données.

L'avantage, c'est qu'avec **un itérateur**, nous avons un objet simple et compact que l'on peut parcourir de manière intuitive. Un objet que l'on peut parcourir grâce à **un itérateur** s'appelle **un objet itérable**. Donc **un itérable** est un objet que l'on peut parcourir de multiples fois.

La notion d'itération

En Python, tous les types built-in sont **itérables** sauf évidemment les types numériques puisque ça n'aurait pas de sens de les parcourir. Nous allons maintenant déconstruire la manière dont fonctionne une boucle **for** sur les itérables.

Pour un ensemble **s = {1, 2, 3, 'a'}** on peut faire une boucle **for** qui affiche chaque élément de l'ensemble :

```
for i in s:
    print(i)
```

Ce qui retourne : ***1***
2
3
a

On peut également utiliser la notation **compréhension de liste** qui va retourner la liste de tous les éléments de l'ensemble **s** qui sont des entiers :

```
[x for x in s if type(x) is int]
```

Ce qui retourne ***[1, 2, 3]***

Maintenant essayons de comprendre comment la boucle **for** a fait pour parcourir notre objet **s**. En fait la boucle **for** commence par récupérer **l'itérateur** sur l'ensemble **s**. On peut tout à fait définir un objet **iter()** sur l'ensemble **s** : **it = iter(s)**. **iter()** est la fonction built-in qui permet de créer **un itérateur** sur **un objet itérable**. Désormais la boucle **for** va appeler une méthode **next()**. Ainsi si on appelle **next(it)**, on obtient le premier élément de notre ensemble **s**. Ensuite on peut le rappeler jusqu'à avoir nos quatre éléments **1 2 3 'a'**. Et lorsque l'on n'a plus d'éléments, la méthode **next()** retourne une exception qui s'appelle **StopIteration**. Ce qui représente bien le fait qu'**un itérateur** ne peut se parcourir qu'une seule fois.

En pratique, nous n'aurons pas à appeler les méthodes **iter()** et **next()** sur nos objets pour être capable de les parcourir. Ce sont **les mécanismes d'itération** comme par exemple les boucles **for** ou **les compréhensions** qui appellent ces méthodes pour nous.

Il est cependant très important de comprendre **ce protocole d'itération** parce que ça nous permettra par la suite de créer nos propres **objets itérables** ou nos propres **itérateurs**.

Les notions d'itérable et d'itérateur

Il y a deux types d'objets : **les itérables** et **les itérateurs**. Ce sont deux types d'objets qui sont conceptuellement différents.

Un itérable est un objet qui a une méthode **_iter_()** qui renvoie un nouvel objet qui s'appelle **un itérateur**. Cette méthode peut s'appeler directement sur l'objet **objet._iter_()** ou alors avec la fonction built-in **iter(objet)**.

Un itérateur est un objet qui a une méthode **_iter_()** qui renvoie **l'itérateur** lui même et une méthode **_next_()** qui retourne un nouvel élément à chaque fois qu'on l'appelle jusqu'à ce qu'il n'y ait plus d'élément à parcourir, auquel cas, **_next_()** renvoie l'exception **StopIteration**. Un itérateur ne peut donc se parcourir qu'une seule fois sur chacun de ses éléments. Et lorsqu'il n'y a plus d'éléments à parcourir on a **StopIteration**. On peut appeler la méthode **_next_()** directement sur l'objet **it._next_()** ou alors avec la fonction built-in **next(it)**.

Pourquoi l'itérateur retourne un itérateur ?

On peut se demander pourquoi on a une méthode **_iter_()** sur **l'itérateur** qui retourne **l'itérateur** lui-même ? La raison c'est que de la même manière qu'un objet **itérable** est **itérable** parce qu'il a une méthode **_iter_()** qui retourne **un itérateur**, **un itérateur** est également **étirable** parce qu'il a une méthode **_iter_()** qui retourne **un itérateur**.

Par conséquent, tous **les mécanismes d'itération** (boucle **for**, **compréhension de liste**) peuvent prendre soit un itérable, soit un itérateur et le parcourir de manière totalement simple et intuitive.

Pourquoi deux notions itérable et itérateur ?

On peut également se demander pourquoi a-t-on deux notions **itérable** et **itérateur**, dans la mesure où les boucles **for** peuvent prendre ces deux objets de manière indifférente ?

En fait, ces deux objets sont conceptuellement différents. **L'itérable** est l'objet qui contient les données. **L'itérateur** est un objet simple et compact qui parcourt les données qui sont contenues dans **l'itérable**.

Le cas particulier des fichiers

Lorsqu'on manipule des objets **itérables** comme les listes, c'est **notre mécanisme d'itération** qui va s'occuper de parcourir ces objets. Mais dans certains cas, on n'aura pas **d'itérable**, nous aurons directement **un itérateur**. C'est par exemple le cas des fichiers.

Si on avait à lire un fichier qui fait des dizaines megabytes ou des centaines de megabytes, ça serait une mauvaise idée d'avoir entièrement à le charger en mémoire. Or le seul moyen d'avoir **un itérable** c'est d'avoir un objet qui contient toutes les données en mémoire. Le choix de Python a donc été de dire : pour les fichiers on a **un itérateur** qui va parcourir ligne par ligne le fichier qui est contenu sur le disque dur. Évidemment, si on a besoin de stocker toutes les lignes d'un fichier dans une liste, on peut le faire mais ce sera de manière explicite.

Le "one shot" des itérateurs

Regardons maintenant de nouveau le fonctionnement des itérateurs et notamment le fait qu'un itérateur ne peut se parcourir qu'une seule fois.

Nous créons deux listes : **a = [1, 2]** et **b = [3, 4]**. On peut maintenant prendre **un itérateur** sur notre liste **a** : **iter(a)** qui crée un nouvel objet **list_iterator**. Mais en pratique nous n'avons pas besoin de le faire puisque l'objet liste est **un itérable** et que l'on peut donc faire autant de boucles **for** qu'on le souhaite sur cet objet. Comme on l'a vu, la liste va contenir une référence vers les objets qu'elle contient. L'objet liste existe donc par conséquent en mémoire.

Regardons maintenant un autre type d'objet : **z = zip(a, b)**. La fonction built-in **zip()** prend le premier élément de chaque liste, les met dans un tuple, puis prend le deuxième élément de chaque liste et les met dans un tuple. On voit bien dans ce cas là, qu'il n'y aurait pas vraiment d'intérêt à créer une structure de données temporaire qui contiendrait la liste de tous les tuples. Le choix de Python a donc été de créer un objet **zip** qui est en fait **un itérateur**.

Si on entre **z is iter(z)**, c'est-à-dire est-ce que **z** est son propre itérateur, Python renvoie **True**. Nous avons donc maintenant la certitude qu'il s'agit d'un objet **itérateur**. Par conséquent, on ne peut le parcourir qu'une seule fois.

Si on fait **une compréhension de liste** : **[i for i in z]**, Python renvoie **[(1, 3), (2, 4)]** c'est-à-dire la liste des tuples qui contiennent le premier élément de chaque liste puis le deuxième élément de chaque liste. Mais si maintenant on fait une deuxième **compréhension de liste** : **[i for i in z]**, Python renvoie **[ ]** c'est-à-dire une liste vide. Notre **itérateur** a été consommé, maintenant il est vide, on ne peut plus le parcourir. Et on peut le vérifier en faisant **next(z)** qui renvoie l'exception **StopIteration**, donc **l'itérateur** a été consommé.

Le principes **des itérateurs** c'est que ce sont des objets simples et compacts qui sont très peu coûteux à créer. Par conséquent, si on veut une nouvelle fois parcourir nos couples de premier et de deuxième éléments de nos listes **a** et **b**, on n'a qu'à recréer un nouvel objet **itérateur** : **z = zip(a, b)** qui est extrêmement peu coûteux à créer puisque cet objet ne va rien parcourir et ne va faire aucun calcul. Le calcul ne sera fait qu'au moment où on va **itérer** sur cet **itérateur**.

Leçon 26 : Objet fonction, fonction lambda, map et filter

Python est un langage multi-paradigme qui supporte la programmation objet mais qui supporte également certains concepts de programmation fonctionnelle. Si Python supporte ces différents paradigmes, c'est pour en faire un langage simple, puissant et facile à utiliser. Python n'est pas un langage dogmatique, c'est un langage qui utilise ce dont on a besoin là où on en a besoin.

Dans cette leçon nous allons parler des fonctions **lambda**, nous allons expliquer que les fonctions étant des objets, on peut les passer comme argument à d'autres fonctions. Et nous allons parler des fonctions built-in **map** et **filter**.

Les fonctions lambda

Les fonctions **lambda** sont des fonctions qui sont anonymes. Tout ce qu'on peut faire avec une fonction **lambda**, on peut le faire avec une fonction traditionnelle. Par conséquent, les fonctions **lambda** ne sont en rien nécessaires. Elles constituent simplement une facilité d'utilisation pour écrire du code parfois qui est plus simple et expressif. En fait, la principale différence entre une fonction **lambda** et une fonction traditionnelle, c'est qu'une fonction **lambda** est une expression. Par conséquent, on peut définir une fonction **lambda** partout où on peut avoir une expression. Par exemple lorsque l'on écrit une liste, un dictionnaire ou alors lorsque l'on veut passer directement un argument à une fonction.

Pour écrire une fonction **lambda** :

```
carre = lambda x: x**2 - 1
```

Et on peut appeler notre fonction **lambda** comme une fonction traditionnelle : **carre(1)** renvoie **0**.

Prenons maintenant une fonction **image()** :

```
def image(f):
    for x in range(10):
        print( f"{f(x)}: {x}" )
```

Nous avons maintenant une fonction **image()** qui accepte un argument qui doit être une fonction. On peut donc lui passer directement une fonction **lambda** : **image(lambda x: x**2 - 1)**. Nous avons ainsi directement passé un objet fonction à notre fonction **image()**, ce qui nous évite d'avoir à définir une seconde fonction avec l'instruction **def**. Si on exécute **image(lambda x: x**2 - 1)**, Python renvoie comme convenu deux colonnes avec la valeur de **f(x)** suivie de **x**.

Il faut bien comprendre qu'à aucun moment nous n'avons besoin d'avoir cette fonction **lambda**. On peut tout à faire définir une fonction traditionnelle **carre(x)** qui retourne **x**2 -** 1, exécuter **image(carre)** et le résultat serait le même qu'avec la fonction **lambda**.

Donc il faut vraiment voir la fonction **lambda** comme quelque chose qui permet d'écrire du code parfois plus simple et expressif qu'avec une autre fonction classique.

Les fonctions map et filter

map et **filter** sont deux primitives de programmation fonctionnelle.

La fonction map :

map permet d'appliquer une fonction à chaque élément d'**un itérable**.

Pour la fonction **carre()** :

```
def carre(x):
    return x**2 - 1
```

On crée un objet **map** affecté à une variable **m** :

```
m = map(carre, range(10))
```

Cette notation va appliquer la fonction **carre()** à chaque élément de **range(10)**. **m** est un objet **map**. Pour voir les différents éléments de l'objet **m**, il suffit de faire **list(m)** qui renvoie **[-1, 0, 3, 8, 15, 24, 35, 48, 63, 80]**. On pourrait également appliquer une somme directement sur **m**.

La fonction filter :

La fonction **filter** permet de filtrer les éléments d'**un itérable** en fonction d'un test.

On crée un objet **filter** affecté à une variable **f** :

f = filter(lambda x: x%2 == 0, range(10))

On peut parcourir l'objet **f** ou afficher la liste correspondant à cet objet pour voir les objets qui ont été filtrés : **list(f)** renvoie **[0, 2, 4, 6, 8]**.

Les fonctions **map** et **filter** permettent d'appliquer une fonction à chaque élément d'**un itérable** et permettent de filtrer les éléments d'un itérable. Cela devrait nous faire penser à **la compréhension de liste** qui permet de faire exactement ce que font les fonctions **map** et **filter**. **map** et **filter** sont dans un paradigme de programmation fonctionnelle et **les compréhensions de liste** correspondent plus à une manière pythonique d'écrire l'application d'une expression à chaque élément d'**un itérable** et un critère de test.

En Python moderne, on préfère utiliser **les compréhensions de liste** aux fonctions **map** et **filter**. Cependant les fonctions **map** et **filter** ont un avantage majeur. Avec **une compréhension de liste** on crée un nouvel objet liste même si on n'a pas forcément besoin de cet objet. Les fonctions **map** et **filter**, au contraire, produisent un objet extrêmement compact.

ATTENTION! **map** et **filter** produisent **des itérateurs**. Cela veut dire qu'on ne peut les parcourir qu'une seule fois. Si on veut parcourir de multiples fois le résultat d'un **map**, il faut produire un nouvel objet **map**.

Leçon 27 : Compréhension des listes, sets et dictionnaires

Dans une précédente leçon, nous avons introduit la notion de **compréhension de liste**. Les **compréhensions de liste**, lorsqu'elles ont été introduites en Python, ont très vite rencontré un énorme succès. Et ce succès a été tel qu'elles ont été étendues aux notions de **set** et de **dictionnaire**.

La compréhension de set

Prenons une liste de prénoms qui n'ont pas de capitalisation consistante : **prenoms = ['ana', 'eve', 'ALICE', 'anne', 'bob']**. Maintenant, imaginons que nous souhaitons créer une liste de tous les prénoms commençant par a et mettre tous ces prénoms en minuscule. Nous pouvons le faire avec une **compréhension de liste** : **a_prenoms = [p.lower() for p in prenons if p.lower().startswith('a')]** qui renvoie **['ana', 'alice', 'anne']**.

Maintenant, nous voulons reprendre notre liste de prénoms et l'étendre. Nous pouvons utiliser la méthode **.extend()** au moyen de laquelle nous pouvons dupliquer tous les éléments de notre liste : **prenoms.extend(prenoms)** qui renvoie **['ana', 'eve', 'ALICE', 'anne', 'bob', 'ana', 'eve', 'ALICE', 'anne', 'bob']**.

Et si nous refaisons notre **compréhension de liste** : **a_prenoms = [p.lower() for p in prenons if p.lower().startswith('a')]**, nous avons cette fois-ci la liste de prénoms commençant par a en minuscule mais avec des prénoms dupliqués : **['ana', 'alice', 'anne', 'ana', 'alice', 'anne']**.

On pourrait vouloir calculer uniquement les prénoms uniques. La méthode traditionnelle en Python, c'est de transformer notre liste en **set** : **set(a_prenoms)** renvoie **{'alice', 'ana', 'anne'}**. Mais entre temps, nous avons créé une liste temporaire qui ne nous sert essentiellement à rien puisque nous voulions uniquement les prénoms uniques.

En fait, on peut obtenir cette liste de prénoms uniques directement à partir d'une **compréhension de liste**. On n'a qu'à reprendre **a_prenoms = [p.lower() for p in prenons if p.lower().startswith('a')]** et y remplacer les crochets par des accolades : **a_prenoms = { p.lower() for p in prenons if p.lower().startswith('a') }** après quoi **a_prenoms** comprendre bien **l'ensemble** des prénoms uniques mis en minuscule qui commencent par la lettre a : **{'alice', 'ana', 'anne'}**.

L'avantage de cette méthode, c'est qu'avec cette méthode de **compréhension de set**, on ne crée à aucun moment une liste temporaire. Le **set** est créé à la volée en parcourant la liste et en faisant les calculs au fur et à mesure.

La compréhension de dictionnaire

Prenons un dictionnaire **ages** en commençant par créer une liste de **tuples** : **ages = [('ana', 20), ('EVE', 30), ('bob', 40)]** puis en appelant la fonction **dict()** : **ages = dict(ages)** qui renvoie **{ 'EVE' : 30, 'ana' : 20, 'bob' : 40 }**.

Maintenant supposons que l'on veuille corriger la capitalisation des clés de ce dictionnaire en mettant toutes les clés en minuscule. On peut le faire directement avec une **compréhension de dictionnaire** : **ages_fix = { p.lower():a for p, a in ages.items() }**. Ainsi nous parcourons **les items** de notre dictionnaire **ages**, et nous retranscrire les clés en minuscule. Le résultat renvoie **{ 'ana' : 20, 'bob' : 40, 'eve' : 30 }**.

On aurait de même très bien pu rajouter un test pour dire par exemple que l'on veut simplement les prénoms mis en minuscule si l'âge est inférieur à 40 : **ages_fix = { p.lower():a for p, a in ages.items() if a < 40 }**. Et le résultat est alors un dictionnaire sans **bob** : **{ 'ana' : 20, 'eve' : 30 }**.

Leçon 28 : Expressions et fonctions génératrices

Nous avons jusque là abordé les notions de **compréhensions de liste, de set et de dictionnaire** et nous savons que les **compréhensions** constituent un moyen extrêmement simple et expressif de parcourir des **itérables** et d'appliquer des traitement sur ces **itérables**. Ces **compréhensions** ont cependant un inconvénient majeur : c'est qu'elles créent des structures de données temporaires.

Nous allons voir dans cette leçon la notion d'**expression génératrice**. Une **expression génératrice** s'écrit exactement comme une **compréhension de liste**, mais la différence c'est que ce qui sera retourné sera un **itérateur** et non pas une liste temporaire. C'est donc un procédé qui permet d'économiser énormément de mémoire tout en gardant toute la fonctionnalité d'une **compréhension**.

Nous verrons également dans cette leçon, la généralisation des **expressions génératrices** que l'on appelle **fonctions génératrices**.

Fonctionnement d'une expression génératrice

Commençons par écrire une **compréhension de liste** : **carre = [x**2 for x in range(1000)]**. Nous avons donc maintenant une liste en mémoire référencée par la variable **carre** qui comprend tous les entiers au carré allant de 0 à 999.

Supposons maintenant que nous voulions simplement calculer la somme de ces carrés. Pour calculer la somme des éléments d'un **itérable** on peut utiliser la fonction built-in **sum()** : **sum(carre)** renvoie **332833500**.

Sauf que pour arriver à ce résultat nous aurions pu nous passer de créer la liste temporaire référencée par **carre**, en utilisant une **expression génératrice**. Pour cela, il suffit de reprendre notre **compréhension** et de remplacer les crochets par des parenthèses : **carre = (x**2 for x in range(1000))**. Grâce à cette commande aucun objet **list** n'a été créé. À la place **carre** référence maintenant un **generator objet** qui est en fait un **itérateur** qui va calculer à la volée la carré de chaque élément retourné par **range(1000)**. Comme **range()** est également un objet qui ne crée pas de liste temporaire mais qui retourne à chaque fois qu'on l'appelle un entier suivant, on peut parcourir tous les éléments de cette **expression génératrice** sans jamais créer de structure temporaire.

De telle sorte qu'on peut désormais faire **sum(carre)** qui renvoie de nouveau **332833500** sauf qu'à la différence de la méthode par **compréhension de liste**, on n'a pas créé de liste temporaire.

ATTENTION! L'**expression génératrice** est un **itérateur**. Par conséquent, si on effectue une deuxième fois **sum(carre)**, on va obtenir **0** puisque l'**itérateur** a été consommé une première fois et que de ce fait, il ne génère plus aucun élément.

Mais comme la création d'un **générateur** est une opération extrêmement peu coûteuse puisqu'on ne fait aucun calcul lorsque l'on crée notre **générateur**, les calculs seront faits à la volée lorsque l'on en aura besoin, il est très facile de recréer un **générateur** : **carre = (x**2 for x in range(1000))** et de calculer une nouvelle fois la somme des carrés avec **sum(carre)**.

Générateurs en chaîne

L'un des intérêts des **expressions génératrices** c'est qu'on peut les chaîner. Ça nous permet donc d'avoir une succession de traitements qui peut s'exécuter sans jamais avoir besoin de créer des objets temporaires.

Commençons par créer un générateur de carré : **gen_carre = (x**2 for x in range(1_000))**. Ensuite nous créons un deuxième générateur capable de détecter les palindromes dans les carrés des entiers allant de 0 à 999 (c'est-à-dire les nombres que l'on peut lire de gauche à droite ou de droite à gauche, comme 121) : **palin = (x for x in gen_carre if str(x) == str(x)[::-1])**. **palin** est maintenant un objet **générateur**. Si on veut obtenir le résultat de **palin** dans une liste, on peut entrer **list(palin)** qui nous renvoie la liste de tous les palindromes des carrés des nombres qui vont de 0 à 999.

La tendance en Python c'est de dire : on manipule toujours des **itérateurs**, le plus loin possible, sans créer de structures de données temporaires. Et on ne créera notre structure de données qu'uniquement à la fin par exemple pour stocker nos résultats.

Cette logique est de dire qu'en Python tout est un objet, tout prend de la place, donc c'est inutile de créer des objets si on n'en a pas besoin. On ne les crée que lorsque c'est strictement nécessaire.

Les fonctions génératrices

Cependant, l'**expression génératrice** a une limitation : exactement comme pour une **compréhension de liste**, on ne peut y mettre qu'une expression. Or si on voulait faire un traitement plus sophistiqué qu'une **compréhension de liste**, on pourrait se demander comment faire, dans la mesure où on ne pourrait mettre ce traitement dans une expression. En fait les **expressions génératrices** ont été généralisées dans le concept des **fonctions génératrices**.

On peut créer des fonctions qui vont se comporter comme des **expressions génératrices**. Et comme ce sera défini à l'intérieur d'une fonction, nous aurons toute la souplesse de ce que l'on peut définir dans une fonction.

Rappel : lorsque l'on définit une fonction, elle retourne forcément une valeur. Si on ne met pas d'instruction **return** elle va retourner l'objet **None**, si on met une instruction **return** elle va retourner le résultat de l'expression. Une fonction lorsqu'elle a retourné, détruit toutes ses variables locales et par conséquent, elle ne garde aucun état entre deux appels. Une fonction génératrice a un comportement différent.

Définissons une fonction normale **gen()** mais au-lieu d'y incorporer un **return**, on y écrit l'instruction **yield** :

```
def gen():
    yield 10
```

Si on exécute cette fonction **gen()**, on obtient un nouvel objet **generator object**. Cet objet est en fait un **itérateur** qui se manipule comme un **itérateur**. Pour **g = gen()**, si on exécute **next(g)**, Python renvoie la valeur **10** qui est après le **yield** dans la fonction **gen()**. Si on exécute à nouveau **next(g)**, Python va s'attendre à un autre **yield** mais comme il n'y a pas dans notre définition de **gen()**, Python renvoie l'exception **StopIteration**.

Définissons un **générateur** plus sophistiqué :

```
def gen(x):
    yield x
    x = x + 1
    yield x
```

Puis on entre **g = gen(10)**. Et maintenant **next(g)** renvoie **10**. Ensuite le générateur se stoppe, il attend avec toutes ses variables locales préservées. Et si on rappelle **next(g)**, l'exécution reprend juste là où on s'est arrêté : c'est-à-dire après le **yield x**, on calcule **x = x +** 1, on on **yield** ce nouveau **x** et Python renvoie donc la valeur **11**. Si on rappelle une troisième fois **next(g)**, Python nous renvoie l'exception **StopIteration**.

En pratique nous ne mettrons pas une multitude de **yield**. En pratique nous ferons plutôt une boucle avec les instructions **for** ou **while**.

```
def carre(a, b):
    for i in range (a, b):
        yield i**2
```

Maintenant si on exécute **carre(1, 10)**, on obtient un **generator object** que l'on référence avec une variable **c** : **c = carre(1, 10)**. Puis on peut récupérer les résultats dans une liste avec : **list(c)** qui renvoie **[1, 4, 9, 16, 25, 36, 49, 64, 81]**. Ce cas relativement simple peut encore tenir dans une **expression génératrice**, nous allons donc définir un cas plus sophistiqué qu'il serait moins commode de définir dans une **expression génératrice**.

Nous créons une fonction palindrome **palin()** qui prend comme paramètre un itérable **it** et qui va parcourir cet itérable pour déterminer si c'est un entier ou une chaîne de caractères, le convertir en chaîne de caractères et vérifier si cette chaîne de caractère est égale à la chaîne prise dans l'autre sens, donc si c'est un palindrome :

```
def palin(it):
    for I in it:
        if isinstance(i, (str, int)) and str(i) == str(i)[::-1]:
            yield i
```

Pour **p = palin [121, 10, 12321, 'abc', 'abba']**, si on récupère les résultats dans une liste : **list(p)** renvoie les palindromes dans une liste **[121, 12321, 'abba']**.

Évidemment, l'intérêt de cette fonction **palin()** n'est pas de prendre une liste temporaire mais de travailler directement sur un **itérateur**. Par exemple, on aurait tout à fait pu lire un fichier qui contient sur chaque ligne un nombre ou une chaîne de caractère et passer directement cet objet fichier à notre **générateur** palindrome.

Dernier exemple, on peut entrer : **list(palin(x**2 for x in range(1_000)))** c'est-à-dire passer une **expression génératrice** à notre **fonction génératrice** palindrome, nous n'avons donc aucune structure de donnée temporaire qui est créée. L'**expression génératrice** va passer les carrés un à un au **générateur** palindrome qui va ensuite produire les palindromes contenus dedans. Le résultat est donc l'ensemble des palindromes identifiés dans les carrés des entiers de 0 à 999, sans avoir eu à créer aucune structure de donnée temporaire.

Leçon 29 : Modules et espaces de nommage

Lorsque nous avons parlé de la notion de **portée de variable**, nous avons expliqué que nous pouvions avoir une variable d'un nom donné qui coexiste dans le même fichier, à l'intérieur d'une fonction et l'intérieur d'un module. Nous avons également expliqué que les modules isolent complètement leur variable. Mais comment est-ce que ce mécanisme d'isolation des variables fonctionne ?

Ce mécanisme fonctionne en fait avec des **espaces de nommage**. Un **espace de nommage** regroupe un ensemble de variables appartenant à un objet. En Python, les modules, les fonctions ainsi que les classes et les instances définissent des **espaces de nommage**.

Dans des langages comme le **C**, où l'on n'a pas cette notion d'**espace de nommage**, il faut faire extrêmement attention de ne pas définir des variables qui se surchargent, s'écrasent l'une l'autre. En **java**, qui est un langage orienté objet, qui a également une notion d'espace de nommage, on doit créer des classes à chaque fois que l'on veut isoler les **espaces de nommage**.

En Python, comme les modules définissent des **espaces de nommage**, on n'a quasiment gratuitement cette notion d'isolation des variables dès que l'on commence à écrire notre première ligne de code.

Nous allons voir ici comment les modules isolent les variables à travers les **espaces de nommage**.

La notion d'isolation des variables dans les espaces de nommage des modules

On commence par créer un module **spam.py**, donc un fichier Python **spam.py**, ainsi qu'un deuxième fichier **egg.py**.

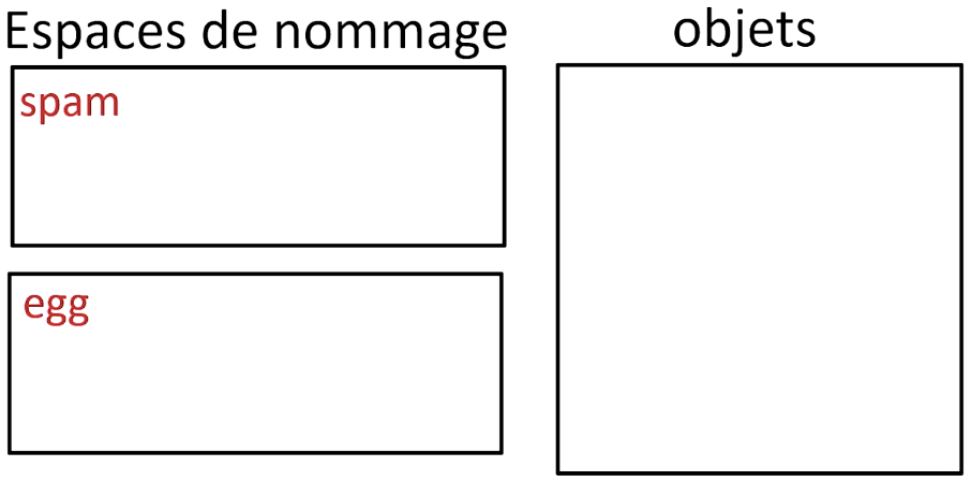
spam.py

```
x = 1

def f():
    print(x)
```

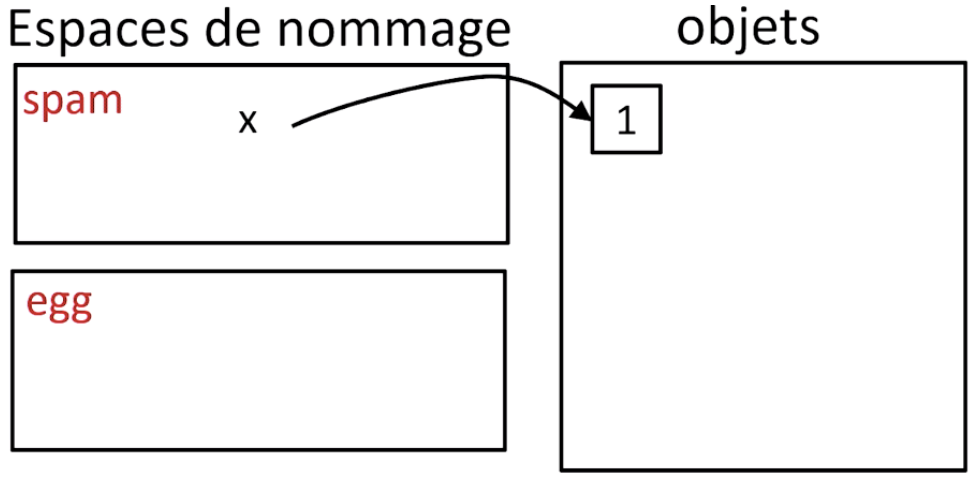
egg.py

```
import spam

x = 2

def f():
    print(x)

f()

spam.f()

print(spam.x)
```

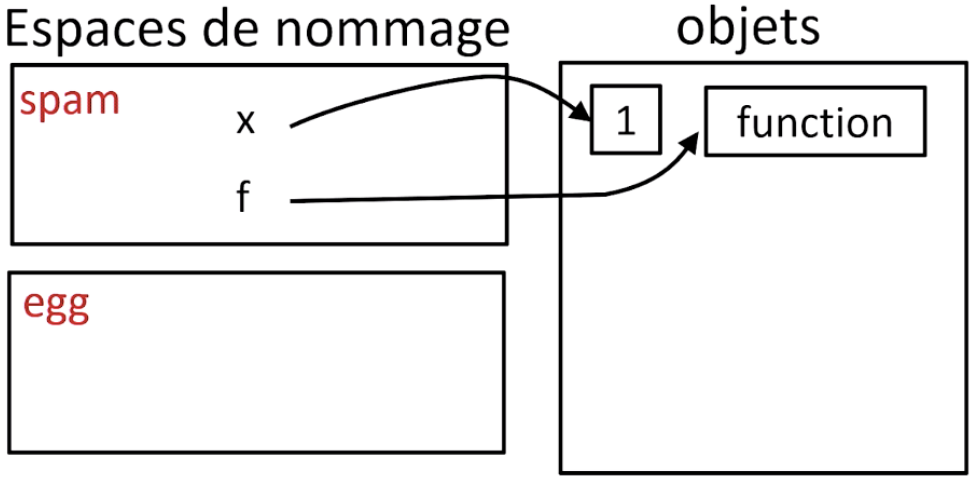
On ouvre un terminal et on entre la ligne de commande : **python egg.py**, ce qui veut dire que l'on va lancer le programme à partir de **egg.py**. Pour regarder ce qui va se passer, on définit l'espace des objets et l'**espace de nommage**. On va avoir un **espace de nommage** pour **spam** et un **espace de nommage** pour **egg**.



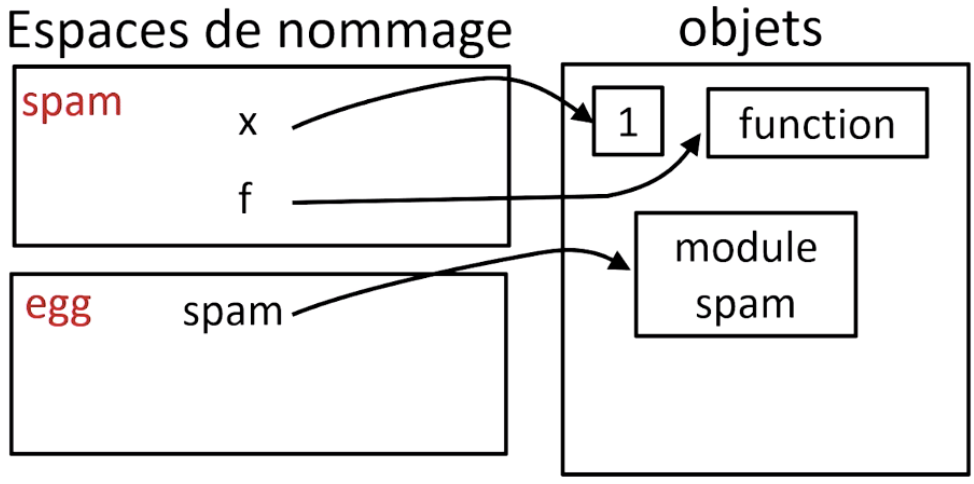
La première ligne de code de **egg.py** indique **import spam**. On va donc aller chercher le fichier **spam.py** et pour créer l'objet module **spam**, nous allons commencer à évaluer le code de **spam.py**. La première ligne de code c'est **x = 1**, nous créons donc un objet entier **1** dans notre espace des objets et une variable **x** qui est dans l'**espace de nommage** de **spam**. Cette variable **x** référence l'entier **1**.



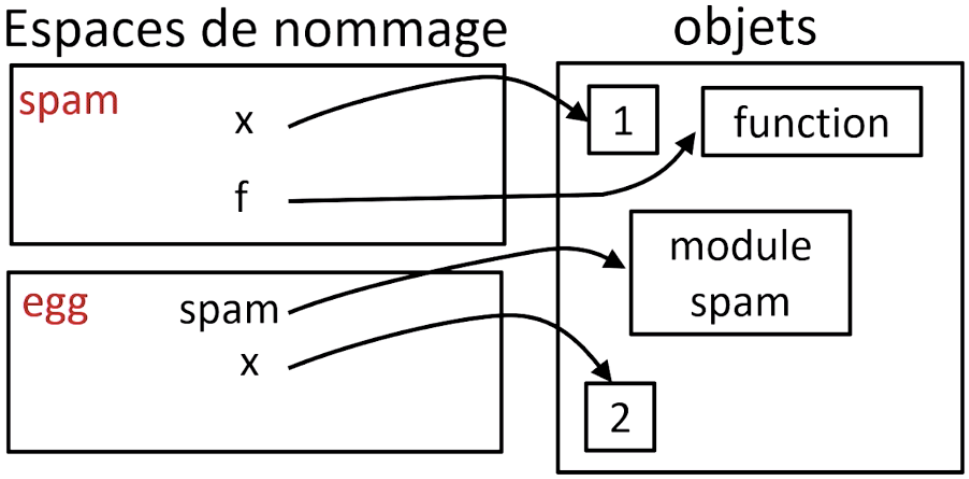
On arrive à la ligne **def f():**, on crée donc un objet **function** dans l'espace des objets et une variable **f** dans l'**espace de nommage** de **spam** qui va référencer l'objet **function**. RAPPEL : le bloc de code d'une fonction n'est évalué que lors de l'appel, pas lors de l'importation.



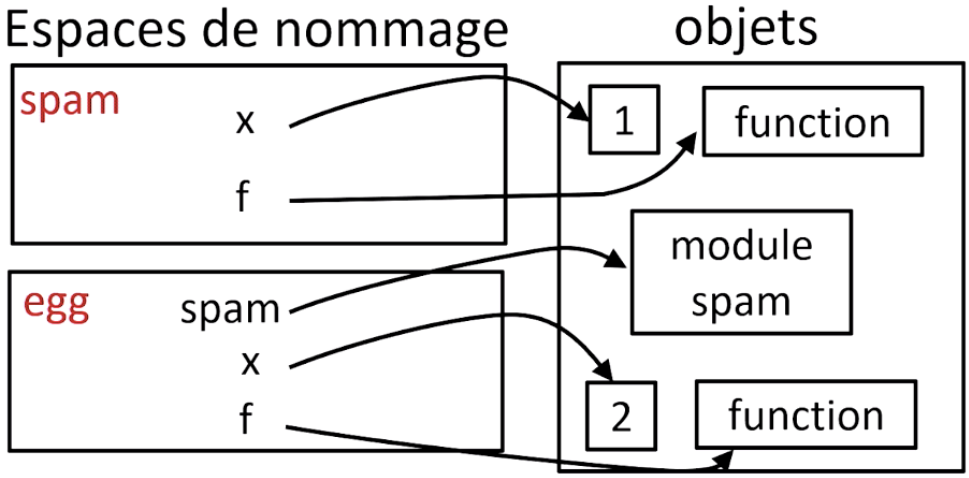
Maintenant que nous avons évalué le code de **spam.py**, on retourne dans le code de **egg.py**. On a fini d'importer l'objet **module spam** qui est donc créé dans l'espace des objets. Et on va créer une variable **spam** dans l'**espace de nommage** de **egg** qui va référencer cet objet **module spam**. Évidemment l'objet **module spam** référence lui-même l'objet **function** et l'entier **1**.



Maintenant on exécute **x = 2**, on crée donc l'entier **2** dans l'espace des objets ainsi qu'une variable **x** dans l'**espace de nommage** de **egg** qui référence **2**.



On arrive à **def f():**, donc on définit un nouvel objet **function** dans l'espace des objets et une variable **f** dans l'**espace de nommage** de **egg** qui référence cet objet **function**.



Maintenant on commence à appeler nos différentes fonctions. Lorsqu'on appelle **f()**, on recherche la variable **f** avec la règle **LEGB**. **f()** étant définie en dehors de toute fonction ou de toute fonction englobante, on cherche donc **f** dans l'**espace de nommage** du module **egg**. **f()** est définie et fait un **print(x)**. **x** est une variable que l'on cherche avec la règle **LEGB**. **x** n'est pas définie dans la fonction. Il n'y a pas de fonction englobante. Elle est définie globalement, c'est la variable globale définie dans l'**espace de nommage** de **egg** qui vaut l'entier **2**. Le programme va donc nous retourner l'entier **2**.

Ensuite nous exécutons **spam.f()**, ce qui veut dire que l'on va accéder à **f()** dans l'**espace de nommage** de **spam**. Cette fonction fait un **print(x)**, **x** est une variable dans **spam**, on cherche **x** avec la règle **LEGB**, **x** n'est ni définie localement, ni dans une fonction englobante, **x** est définie globalement, c'est la variable définie dans l'**espace de nommage** de **spam** qui vaut **1**. **spam.f()** va donc afficher l'entier **1**.

Enfin, on fait **print(spam.x)**, qui veut dire que l'on accède à l'attribut **x** qui est défini dans l'**espace de nommage** de **spam**. **x** vaut **1** dans cet **espace de nommage**, donc **print(spam.x)** affiche l'entier **1**.

Conclusion

Nous avons cette notation très importante : **objet.attribut** qui veut dire que l'on accède à l'**attribut** dans l'**espace de nommage** de notre **objet**. Lorsque cet objet est un **module**, cela veut dire que l'on accède à l'**attribut** dans l'**espace des variables globales** de notre **module**.

Nous verrons que dans le cas des **instances** et des **classes**, nous avons un mécanisme de recherche des **attributs** un peu différent dont nous parlerons dans une prochaine leçon.

Les **espaces de nommage** en Python isolent les variables mais ils n'isolent pas les objets. Par conséquent, avec des **mécanismes d'espaces de nommage**, on peut tout de même avoir des **références partagées** vers des objets mutables. Si un même objet mutable est référencé par deux variables isolées dans deux **espaces de nommage** différents, et que l'objet mutable est modifié, il sera modifié par **effet de bord**, donc les deux variables dans les deux **espaces de nommage** différents verront cet objet modifié.

On peut également se demander comment les **espaces de nommage** sont implémentés en Python ? En Python rien n'est caché. Les **espaces de nommage** sont, en général, implémentés sous forme de **dictionnaire**. La **clef** du **dictionnaire** correspond au nom de la variable et la **valeur** correspondante à la **clef** va être la référence vers l'objet qui est référencé par la variable. Donc lorsqu'on ajoute une variable dans un **module**, on crée une entrée dans l'**espace de nommage** de ce **module**, donc une **clef** correspondant au nom de la variable et une référence vers l'objet associé à la variable.

Les **espaces de nommage** permettent une parfaite isolation des variables. Leur grande force c'est que pour accéder à l'**attribut** dans un autre **module**, dans un autre **objet**, on doit utiliser une notation explicite : **objet.attribut**. En rendant ce mécanisme explicite on n'a pas de risque d'erreur ou de collision de variable par erreur.

Enfin la règle **LEGB** permet en fait de savoir où la variable que l'on référence a été définie, c'est-à-dire dans quel **espace de nommage** cette variable a été définie.

Leçon 30 : Processus d'importation des modules

Le processus d'importation d'un **module**, c'est les différentes étapes que va suivre l'interpréteur Python du moment où on entre l'instruction **import** jusqu'au moment où on a l'objet **module**.

Exemple avec l'importation du **module os** :

import os

os a deux rôles :

- il va définir le nom du fichier qui va être cherché sur le disque dur, qui va s'appeler **os.py** (certains **modules** sont directement écrits en **C**)
- le nom **os** va également servir à définir le nom de la variable qui va référencer l'objet **module**.

Trouver le fichier module

Il faut trouver le fichier sur le disque dur. Pour cela, l'interpréteur va commencer par regarder si le **module os** est défini dans le répertoire courant, là où on a démarré notre interpréteur. Ensuite, s'il ne trouve pas le fichier, l'interpréteur va chercher le **module os** dans la variable système qui s'appelle **PYTHONPATH**.

Nous avons dans notre **module os**, un dictionnaire **environ** qui en fait veut dire **environnement** et qui contient toutes les variables d'**environnement** dans notre système.

os.environ['PYTHONPATH']

renvoie dans la leçon **'C:\\Users\\alegout\\Desktop'**, ce qui veut dire que la variable **PYTHONPATH** est ici définie vers le bureau de l'ordinateur.

Si on ne trouve toujours pas le fichier, Python va aller le chercher dans le répertoire des librairies standards. Ce qui explique pourquoi on peut importer n'importe quelle librairie standard sans avoir à se soucier de l'endroit où se situe ce fichier **module**.

Lorsqu'on a un doute sur le chemin de recherche, on peut regarder dans une variable qui s'appelle **sys.path** :

import sys

sys.path est une liste qui contient tous les chemins qui sont suivis par l'interpréteur Python dans l'ordre. Du premier chemin au dernier.

```
In [17]:  import sys

In [18]: sys.path
Out[18]:
['',
 '/anaconda3/lib/python36.zip',
 '/anaconda3/lib/python3.6',
 '/anaconda3/lib/python3.6/lib-dynload',
 '/anaconda3/lib/python3.6/site-packages',
 '/anaconda3/lib/python3.6/site-packages/aeosa',
 '/anaconda3/lib/python3.6/site-packages/IPython/extensions',
 '/Users/mathieulehot/.ipython']
```

Cette variable étant une liste, on peut la modifier en cours d'exécution. Et lorsqu'on fera une importation, le processus d'importation regardera l'état actuel de cette variable. C'est ce qui permet dans un programme de pouvoir adapter les chemins de recherche des modules.

La précompilation

Maintenant que nous avons trouvé notre fichier **module**, nous allons devoir le **précompiler**. La **précompilation** consiste à générer du **bytecode**. Ce **bytecode** qui en général est un fichier d'extension **.pyc**, va être mis dans un répertoire qui s'appelle **__pycache__**. Ce répertoire est en général là où on exécute notre programme.

Pour finir, une fois que l'interpréteur Python a généré le **bytecode**, il va évaluer ce **bytecode** pour générer l'objet **module**. Un **module** s'importe toujours de manière séquentielle, comme vu dans

la leçon précédente. Donc on va parcourir les lignes de la première à la dernière ligne de code, dans l'ordre du début à la fin. Lorsqu'on y rencontre une fonction, on va créer l'objet fonction, par contre le bloc de la fonction ne sera évalué qu'à l'appel de la fonction.

Module et références partagées

Le processus d'importation étant une opération coûteuse, l'interpréteur, lorsque l'on fait de multiples **import** vers le même **module**, ne va importer ce **module** qu'une seule fois et l'interpréteur va ensuite créer des **références partagées** vers cet objet **module**. C'est pourquoi il est très important de comprendre qu'un objet **module** est mutable. Et que la manière d'importer un **module** peut avoir un impact sur l'**espace de nommage** de ce module ou l'**espace de nommage** de notre programme.

Leçon 31 : Importation des modules et espaces de nommage

Lorsqu'on importe un **module**, il n'est importé qu'une seule fois. De multiples **import** vont donc créer des **références partagées** vers cet objet **module** qui n'existe qu'en un seul exemplaire. L'objet **module** étant mutable, il est possible de le modifier. Il est par conséquent très important de comprendre l'interaction des différents mécanismes d'importation avec l'**espace de nommage** des **modules**.

La méthode import …

Nous créons deux fichiers **spam.py** et **egg.py** :

spam.py
```
x = 1
```

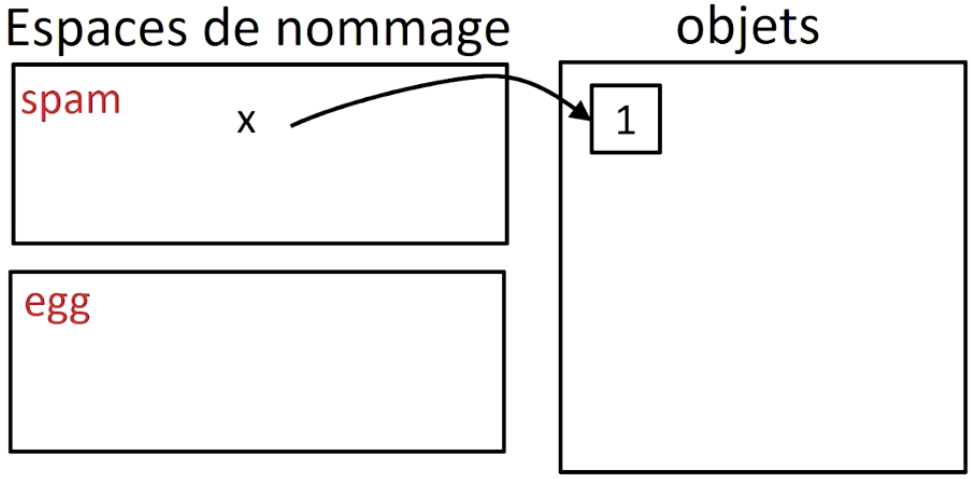
egg.py
```
import spam
x = 2
```

Une fois que ces fichiers sont créés, on peut exécuter le fichier **egg.py** en ligne de commande depuis notre terminal avec la commande : **python egg.py**.

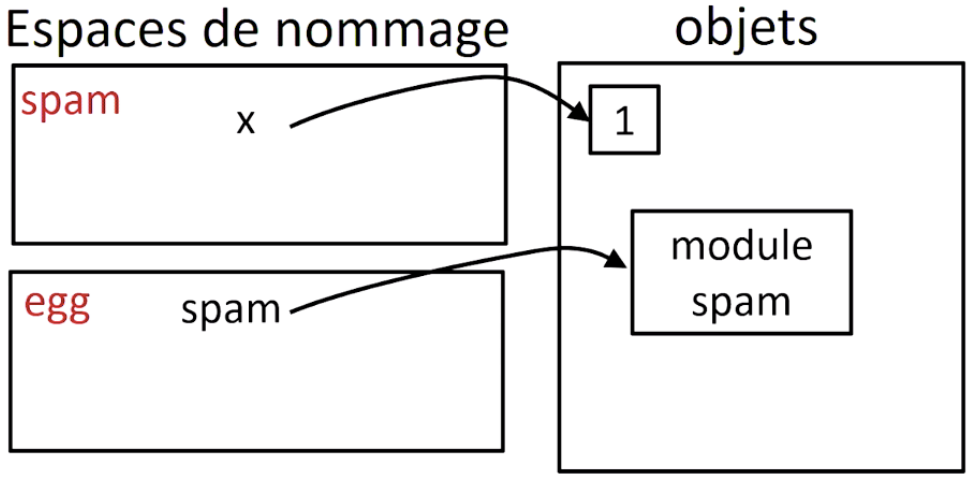
Si on veut voir les **espaces de nommage** des **modules**, on peut utiliser l'instruction **vars()** équivalente dans ce contexte à l'instruction **globals()**. Et si on veut voir l'**espace de nommage** d'un **module** importé, on peut utiliser **vars(nom_du_module)**. On peut également simplement voir les attributs d'un **module** en utilisant **dir(nom_du_module)**.

Maintenant regardons l'impact de l'importation sur les **espaces de nommage**.

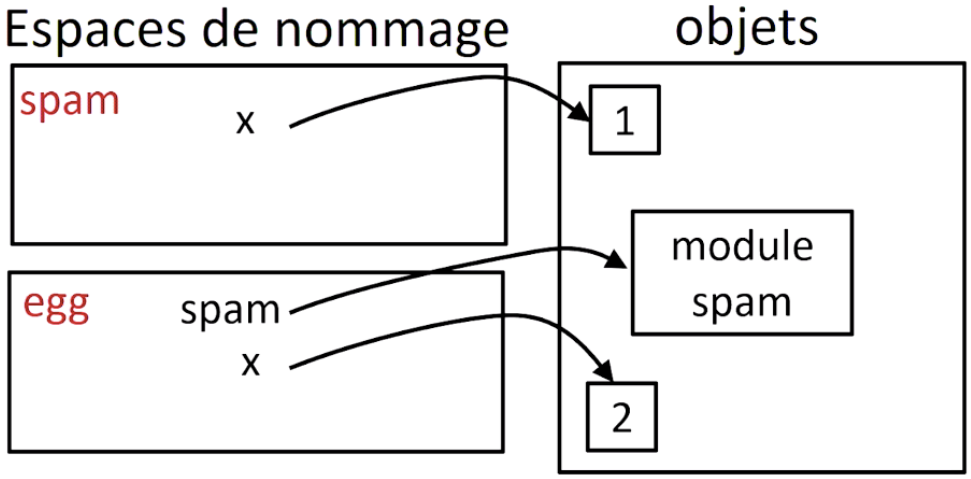
Lorsque l'on exécute **egg.py**, on va commencer par importer notre **module spam**. On crée donc l'entier **1** dans l'espace des objets, la variable **x** dans l'**espace de nommage** de **spam** et **x** va référencer l'entier **1**.



L'objet **module spam** a été créé dans l'espace des objets et il est référencé par une variable **spam** dans l'**espace de nommage** de **egg**.



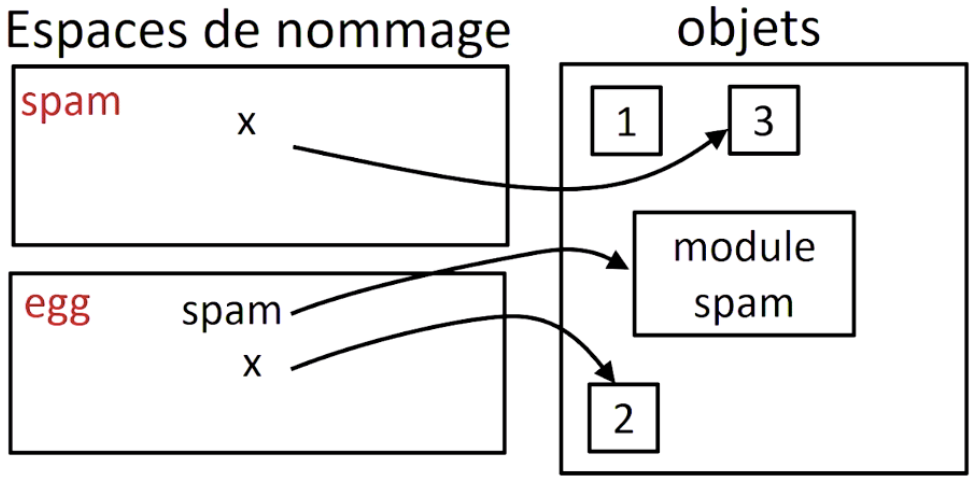
On crée enfin l'entier **2** dans l'espace des objets et une variable **x** dans l'**espace de nommage** de **egg** qui référence l'entier **2**.



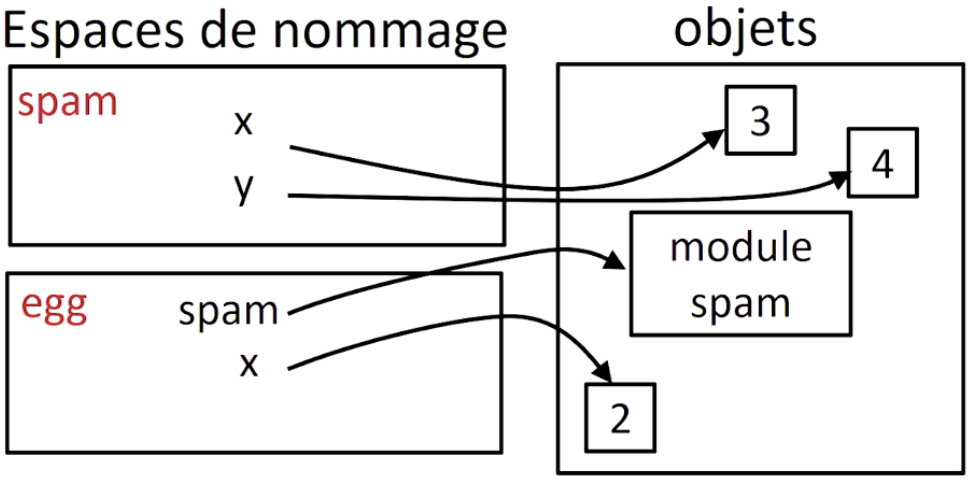
Maintenant, si on exécute **print(x)**, selon la règle **LEGB** la variable **x** est définie globalement dans le **module egg**, elle référence l'entier **2**. Donc **print(x)** renvoie l'entier **2**.

Si on exécute **print(spam.x)**, on va rechercher **x** dans l'**espace de nommage** de **spam**. Donc **print(spam.x)** renvoie l'entier **1**.

Maintenant si nous faisons **spam.x = 3**, nous créons un nouvel entier **3** et la variable **x** de l'**espace de nommage** de **spam** référence maintenant cet entier **3**.



Si maintenant, on entre **spam.y = 4**, nous créons un entier **4** et une nouvelle variable **y** dans l'**espace de nommage** de **spam** qui référence l'entier **4**.



Il fait bien comprendre l'importance de la notation **import spam**. Avec **import spam**, on a une parfaite isolation des **espaces de nommage**. Le seul moyen d'accéder à un attribut dans l'**espace de nommage** de **spam**, c'est d'utiliser la notation explicite **spam.attribut**.

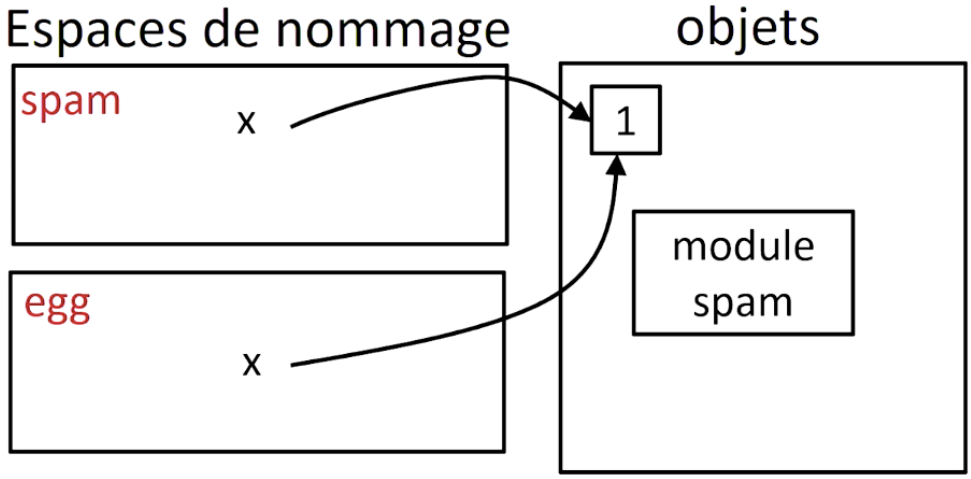
La méthode from … import …

Nous allons voir maintenant une autre manière d'importer avec la notation **from spam import x** :

spam.py
```
x = 1
```

egg.py
```
from spam import x
x = 2
```

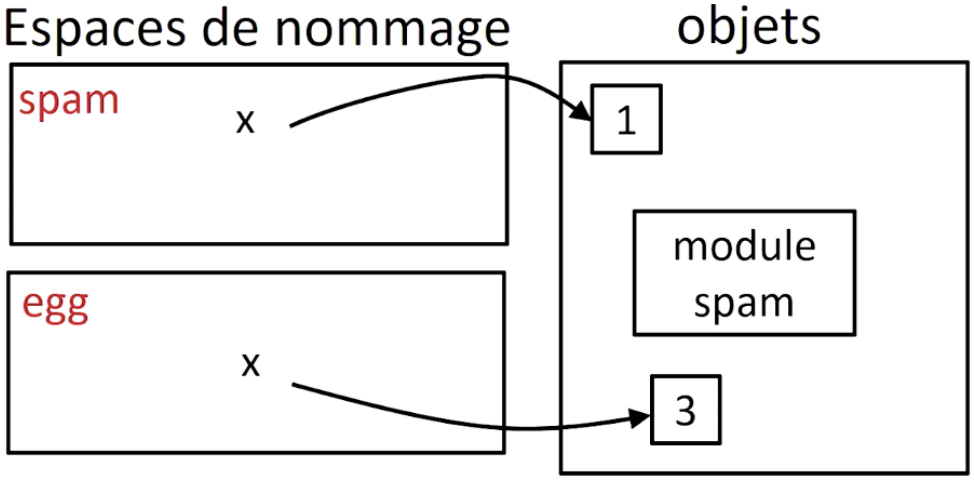
Nous avons toujours un entier **1** dans l'espace des objets et une variable **x** dans **spam** qui référence cet entier **1**. Par contre l'exécution de **from spam import x** va créer un objet **module spam** et une variable **x** dans l'**espace de nommage** de **egg** qui va référencer le même entier **1** que **x** qui est dans l'**espace de nommage** de **spam**.



Il y a donc une différence fondamentale avec **import spam** : c'est que maintenant nous avons une variable locale **x** dans **egg** qui référence l'objet référencé par la variable **x** de **spam**. Et on n'a plus de variable **spam** qui permet d'accéder au **module spam**.

Si maintenant on fait **print(x)**, le **x** fait référence au **x** local qui renvoie donc l'entrée **1**.

Si on fait maintenant **x = 3**, on crée l'entier **3** dans l'espace des objets et la variable **x** locale va maintenant référencer ce nouvel objet **3**. Et si maintenant on fait **print(x)**, Python renvoie l'entier **3**.



ATTENTION! Il faut bien comprendre que la manière choisie ici de présenter les **espaces de nommage** séparés des **modules** est une vue abstraite. Il faut bien comprendre que l'**espace de nommage** c'est un **dictionnaire**, c'est donc un objet qui est un attribut du **module**. Donc chaque **module** a un attribut qui est son **espace de nommage**.

Les différences de comportement entre *import …* et *from … import …*

import spam :

Lorsqu'on fait **import spam** on a une isolation parfaite des **espaces de nommage**. On peut accéder à tous les attributs de **spam** avec la notation explicite **spam.attribut**. Il n'y a donc aucun risque d'erreur.

from spam import x :

Lorsque l'on fait **from spam import x**, on va importer dans notre **espace de nommage** un nom **x** qui va référencer un objet défini dans le **module spam**. Il y a donc un risque de collision puisque **x** va importer une variable dans notre **espace de nommage** et que si on déjà une variable **x** définie dans notre **espace de nommage**, on va avoir une collision de variables. Enfin, on n'a pas accès aux autres attributs de **spam** puisque **from spam import x** ne permet que d'accéder à l'objet référencé par **x** dans le module **spam**.

Conclusion

Nous avons donc deux mécanismes d'importation des **modules** :

- import module qui va importer les **modules** avec une parfaite isolation des **espaces de nommage**. Donc c'est totalement sûr
- from module import attribut qui va importer uniquement l'attribut dans notre **espace de nommage** avec un risque de collision. Le principal intérêt d'utiliser ce mécanisme est juste d'avoir une notation plus courte pour éviter d'avoir en permanence à manipuler **module.attribut**. En pratique, on ne l'utilise que lorsqu'on utilise systématiquement une fonction définie dans un autre **module**.

from module import attribut on n'a pas de référence vers l'objet **module**. On pourrait croire que cet import va donc coûter moins de mémoire. Mais en fait ça ne change absolument rien puisque l'objet **module** va quand même être créé et va quand même être maintenu en mémoire tant que le programme est en cours d'exécution. Il n'y a donc aucune différence avec le mécanisme **import module** en matière d'utilisation mémoire.

Leçon 32 : Classes, instances et méthodes

En Python, tout est un objet et les caractéristiques de chaque objet sont définies par leur type. En python, les types built-in sont très puissants mais ils ne peuvent pas couvrir tous nos besoins. C'est pourquoi nous avons le concept de **classe**. Une **classe** en Python permet de définir nos propres types. C'est-à-dire que l'on va pouvoir créer un modèle pour des objets que l'on peut produire et qui auront leur propre caractéristique.

Classes, instances et relations d'héritage

Comme en Python, tout est un objet, les **classes** sont aussi des objets. La **classe** est la définition des caractéristiques que l'on va écrire dans le module. Et lorsque le module sera importé, l'objet **classe** sera créé. Nous aurons ainsi ce que l'on appelle une usine à **instances**. À chaque fois que l'on appellera notre **classe**, la **classe** créera de nouvelles **instances**. Et il va y avoir une relation particulière entre l'**instance** et sa **classe**. En fait on dit que l'on a une **relation d'héritage** entre l'**instance** et la **classe**. Ce qui veut dire que l'**instance** va pouvoir hériter, observer tous les attributs qui sont définis dans la **classe**.

Cette **relation d'héritage** est en fait liée à la notion d'**espace de nommage**. Une **classe** a son propre **espace de nommage**. Et une **instance** a son propre **espace de nommage**. Lorsque l'on recherche un attribut dans une instance, on va le rechercher dans l'**espace de nommage** de l'**instance**. Si on ne le trouve pas dans l'**instance**, on va remonter l'**arbre d'héritage** et le chercher dans l'**espace de nommage** de sa **classe**. C'est cette recherche d'attribut dans les **espaces de nommage** de l'**instance** et de la **classe** que l'on appelle **arbre d'héritage**.

Classes et espaces de nommage

Pour créer une **classe**, on utilise l'instruction **class** suivie du nom de la **classe** :

```
class Phrase:
    ma_prase = "je fais un mooc sur Python"
```

Nous avons maintenant un objet **classe Phrase**. Et maintenant nous allons pouvoir produire des **instances** à partir de cette **classe Phrase**. On définit une **instance** : **p = Phrase()**. Pour produire l'**instance**, il faut mettre des parenthèses après le nom de la **classe**, comme pour appeler une fonction. **p** est également un objet.

On peut accéder aux **espaces de nommage** de nos objets **classe** et **instance** avec l'attribut **__dict__** :

```
In [6]: Phrase.__dict__
Out[6]:
mappingproxy({'__module__': '__main__',
              'ma_phrase': 'je fais un mooc sur python',
              '__dict__': <attribute '__dict__' of 'Phrase' objects>,
              '__weakref__': <attribute '__weakref__' of 'Phrase' objects>,
              '__doc__': None})
```

En pratique, ces méthodes spéciales __methode__ ne s'utilisent pas directement à la main. Il y a des fonctions built-in pour y accéder. En l'occurence, la fonction built-in qui permet d'accéder au dictionnaire c'est **vars()** :

```
In [7]: vars(Phrase)
Out[7]:
mappingproxy({'__module__': '__main__',
              'ma_phrase': 'je fais un mooc sur python',
              '__dict__': <attribute '__dict__' of 'Phrase' objects>,
              '__weakref__': <attribute '__weakref__' of 'Phrase' objects>,
              '__doc__': None})
```

Dans cet **espace de nommage**, on voit un certain nombre de modules :

- l'attribut **__module__** qui référence **__main__**, ça veut donc dire que notre **classe** est dans le module **__main__**.
- On voit également l'attribut **ma_phrase** que l'on a défini dans la **classe** et qui référence la chaîne caractères **'je fais un mooc sur python'**.

L'**espace de nommage** de notre **classe** est un objet particulier qui s'appelle un **mappingproxy**. La manière dont sont implémentés les **espaces de nommage** en Python sont des détails d'implémentation. Nous n'avons donc pas vraiment à nous soucier des caractéristiques de cet objet. Il faut simplement savoir que le **mappingproxy** pour une **classe** est une sorte de **dictionnaire** qui ne peut être qu'en lecture seule, on ne peut pas le modifier directement. Cependant une **classe** est malgré tout un objet mutable. Cela veut simplement dire que l'on ne peut pas modifier le **dictionnaire** à la main. Ce choix a été fait pour des raisons de stabilité et de performance.

Si on regarde l'**espace de nommage** de l'instance **p**, on voit que l'**espace de nommage** est vide :

```
In [8]: vars(p)
Out[8]: {}
```

Donc lorsque l'on crée une **instance**, cette **instance** est créée avec un **espace de nommage** vide. Mais il y a une **relation d'héritage** entre l'**instance** et la **classe**. Cela veut dire que si on recherche un attribut dans l'**espace de nommage** de l'**instance** et qu'on ne trouve pas cet attribut, on va aller le trouver dans l'**espace de nommage** de la **classe**.

Ainsi si on recherche l'attribut **ma_phrase** dans **p**, on arrive à le trouver grâce à la remontée dans l'**arbre d'héritage** :

```
In [9]: p.ma_phrase
Out[9]: 'je fais un mooc sur python'
```

Classes et instances, des objets mutables

Les **classes** et les **instances** sont des objets mutables. Et la résolution des attributs le long de l'**arbre d'héritage** est fait de manière dynamique en fonction de l'état des **espaces de nommage** au moment où on fait la résolution de l'attribut.

Ainsi on peut définir dans notre classe **Phrase** un nouvel **attribut .mots** : **Phrase.mots = Phrase.ma_phrase.split()** qui va découper la chaîne de caractères en une liste de mots qui va maintenant être référencé par l'attribut **.mots**. **Phrase.mots** renvoie **['je', 'fais', 'un', 'mooc', 'sur', 'python']**.

On peut accéder à cet attribut **.mots** avec l'**instance p** : **p.mots** renvoie **['je', 'fais', 'un', 'mooc', 'sur', 'python']**.

Si on vérifie l'**espace de nommage** dans la **classe** :

```
In [12]: vars(Phrase)
Out[12]:
mappingproxy({'__module__': '__main__',
              'ma_phrase': 'je fais un mooc sur python',
              '__dict__': <attribute '__dict__' of 'Phrase' objects>,
              '__weakref__': <attribute '__weakref__' of 'Phrase' objects>,
              '__doc__': None,
              'mots': ['je', 'fais', 'un', 'mooc', 'sur', 'python']})
```

On constate que nous avons un **attribut ma_phrase** qui référence la chaîne de caractère **'je fais un mooc sur python'** et un **attribut .mots** qui référence la liste **['je', 'fais', 'un', 'mooc', 'sur', 'python']**. **vars(p)** renvoie toujours **{ }** ce qui confirme bien que l'on a cette résolution d'**attribut** qui est faite de manière dynamique le long de l'**arbre d'héritage**.

Classes, instances et comportements

Les **classes** et les **instances** sont tous les deux des objets mutables qui ont leur propre **espace de nommage** et qui ont une **relation d'héritage** : de l'**instance** on remonter vers la **classe** pour rechercher les **attributs**.

Cependant, il nous manque un mécanisme majeur pour faire de vraies **classes**. Lorsqu'on définit une **classe**, on définit des **comportements** dont les **instances** vont hériter. Or pour l'instant nous n'avons défini aucun **comportement**.

Les **comportements** s'implémentent dans les **classes** par l'intermédiaire de méthodes, c'est-à-dire de fonctions que l'on définit dans les **classes**. Et ces fonctions ont une caractéristique particulière, c'est qu'elles sont capables de travailler sur les **attributs** de l'**instance**.

s = "je fais un mooc sur python"

```
class Phrase:
    def initia(self, ma_phrase):
        self.ma_phrase = ma_phrase
```

Notre méthode **initia()** prend deux arguments : **self** et **ma_phrase**. Lorsqu'on appelle une méthode sur une **instance**, la référence de l'**instance** sur laquelle on appelle cette méthode va être automatiquement passée comme premier argument. La variable **self** va donc référencer l'**instance**. Et ensuite, on peut passer d'autres arguments à la méthode comme ici **ma_phrase**. Maintenant, comme **self** est une référence de l'**instance**, lorsque la méthode va s'exécuter, **self.ma_phrase** va créer un **attribut ma_phrase** dans l'**instance** et lui faire référencer l'objet **ma_phrase** passé en deuxième argument, ici probablement une chaîne de caractères.

Maintenant on définit une **instance** de **Phrase()** :

p = Phrase()

Puis on appelle la méthode sur l'**instance p** :

p.initia(s)

Si on regarde maintenant l'**espace de nommage** de l'**instance p**, **vars(p)** renvoie **{'ma_phrase' : 'je fais un mooc sur python'}**, nous avons donc maintenant un **attribut .ma_phrase** qui est **'je fais un mooc sur python'**. Donc l'appel de la méthode **initia()** a permet de créer un **attribut .ma_phrase** dans l'**instance** qui référence l'argument que l'on a passé à la méthode **initia()**.

On peut se demander comment Python fait pour passer automatiquement l'**instance** ? En fait c'est un mécanisme défini par Python qui s'appelle un **mécanisme de fonction bound**. Sur **Phrase** on a un **attribut .initia** qui est une fonction classique définie avec **def**. Par contre si on appelle .**initia** sur l'**instance p**, nous avons un autre type d'objet qui s'appelle un **objet bound** :

```
>>> Phrase.initia
<function Phrase.initia at 0x10142ee18>
>>> p.initia
<bound method Phrase.initia of <__main__.Phrase object at 0x1030abc50>>
```

Un **objet bound** veut dire que c'est une fonction qui est liée à l'**instance**. Et lors de l'appel, Python va automatiquement passer l'**instance** comme premier argument. Donc c'est le fait qu'on appelle la fonction sur l'**instance** qui crée cet **objet bound** et qui indique à Python qu'on doit lui passer l'**instance** comme premier argument.

Donc **p.initia(s)** est totalement équivalent à **Phrase.initia(p, s)**.

Leçon 33 : Méthodes spéciales

En Python, une caractéristique des classes, c'est qu'on peut créer nos objets qui se manipulent exactement comme des types built-in.

Pour une classe **Phrase** avec laquelle nous initialisons l'instance **p** :

p = Phrase("je fais un mooc sur Python")

On peut obtenir le nombre de mots avec la fonction built-in **len()** : **len(p)**, faire un test d'appartenance avec l'instruction **in** : **m in p**, accéder au troisième mot avec la notation crochet **p[2]**, faire un **print()** directement sur l'instance pour afficher la liste des mots en colonne : **print(p)** ou alors si on a deux objets **Phrase()** : **p1 = Phrase("je fais")** et **p2 = Phrase("un mooc")**, pouvoir les concaténer avec une addition **p1 + p2**.

Toutes ces opérations peuvent être implémentées sur notre propre classe **Phrase**. Cela se fait par l'intermédiaire de **méthodes spéciales**. Les **méthodes spéciales** comment toutes par un double underscore et finissent toutes par un double underscore. Et elles sont appelées automatiquement quand on utilise par exemple une fonction built-in, un opérateur ou une instruction.

Ainsi, par exemple, le test d'appartenance correspond à la méthode **__contains__()**. Nous allons voir ici comment exploiter ces **méthodes spéciales** pour implémenter pour notre propre objet **Phrase** des comportements qui soient exactement les mêmes qu'avec les objets built-in.

L'initialisateur__init__()

Nous reprenons la classe **Phrase** définie dans la leçon 32, et nous y ajoutons une nouvelle méthode qui compte le nombre de caractères de notre phrase :

```
class Phrase:
    def initia(self, ma_phrase):
        self.ma_phrase = ma_phrase

    def nb_lettres(self):
        return len(self.ma_phrase)
```

Nous allons maintenant implémenter le premier comportement que l'on implémente pour toutes nos classes : l'**initialisation** de notre instance.

Jusque là, pour **initialiser** une instance, on entrait d'abord **p = Phrase()**, puis on appelait la méthode **.initia()** et on lui passait une chaîne de caractères : **p.initia("chaîne de caractères")**. Pour que ce processus d'**initialisation** se fasse automatiquement, on utilise la **méthode spéciale __init__()**. Et cette méthode va être appelée automatiquement lorsque l'on va créer notre instance.

```
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase

    def nb_lettres(self):
        return len(self.ma_phrase)
```

Si maintenant on exécute **p = Phrase('je fais un mooc sur python')**, alors **p.ma_phrase** contient désormais un attribut **.ma_phrase** qui renvoie **'je fais un mooc sur python'**.

Cet **initialisateur**, qu'on appelle parfois par abus de langage un **constructeur**, permet de créer des attributs automatiquement avec une certaine valeur par défaut lorsque l'on crée notre instance.

La méthode __len__()

Maintenant, nous souhaitons afficher par exemple le nombre de mots que l'on a dans notre phrase. Pour cela il faut que l'on implémente un attribut qui contienne la liste des mots dans notre **initialiseur**.

```
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()

    def nb_lettres(self):
        return len(self.ma_phrase)
```

Donc lorsque nous allons créer notre instance, nous allons avoir un attribut **.ma_phrase** dans l'instance qui référence la chaîne de caractères et un attribut **.mots** qui va référencer une liste qui contient chaque mot.

Ensuite, si on veut obtenir le nombre de mots avec la fonction built-in **len()**, exactement comme on le ferait avec un type built-in, donc en faisant un **len(p.mots)**, on peut l'implémenter en définissant une méthode spéciale qui va s'appeler **__len__()** :

```
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()

    def nb_lettres(self):
        return len(self.ma_phrase)

    def __len__(self):
        return len(self.mots)
```

Maintenant si on recharge notre classe et que l'on réexécute **p = Phrase('je fais un mooc sur python')**, on peut directement faire un **len(p)** qui renvoie **6**, les six mots de notre phrase.

Donc on commence à construire avec les **méthodes spéciales** une classe qui produit des instances qui se comportent comme des types built-in.

Le test d'appartenance __contains__()

On peut également vouloir implémenter le test d'appartenance. Pour cela, nous avons une **méthode spéciale** qui s'appelle **__contains__()**.

```
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()

    def nb_lettres(self):
        return len(self.ma_phrase)

    def __len__(self):
        return len(self.mots)

    def __contains__(self, mot):
        return mot in self.mots
```

On recharge notre classe et on réexécute **p = Phrase('je fais un mooc sur python')**. Et maintenant **'moon' in p** Python renvoie **False** et **'mooc' in p** renvoie **True**. On vient donc d'implémenter le test d'appartenance directement sur notre classe.

La méthode __str__()

print(p) renvoie l'adresse de l'objet **<__main__.Phrase object at 0x10fa43fd0>**. Donc si on veut afficher ce que contient notre instance, on doit implémenter la **méthode spéciale** qui s'appelle **__str__**.

```
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()

    def nb_lettres(self):
        return len(self.ma_phrase)

    def __len__(self):
        return len(self.mots)

    def __contains__(self, mot):
        return mot in self.mots

    def __str__(self):
        return "\n".join(self.mots)
```

On recharge notre classe et on réexécute **p = Phrase('je fais un mooc sur python')**. Et maintenant si on exécute **print(p)**, Python affiche les mots en colonne :

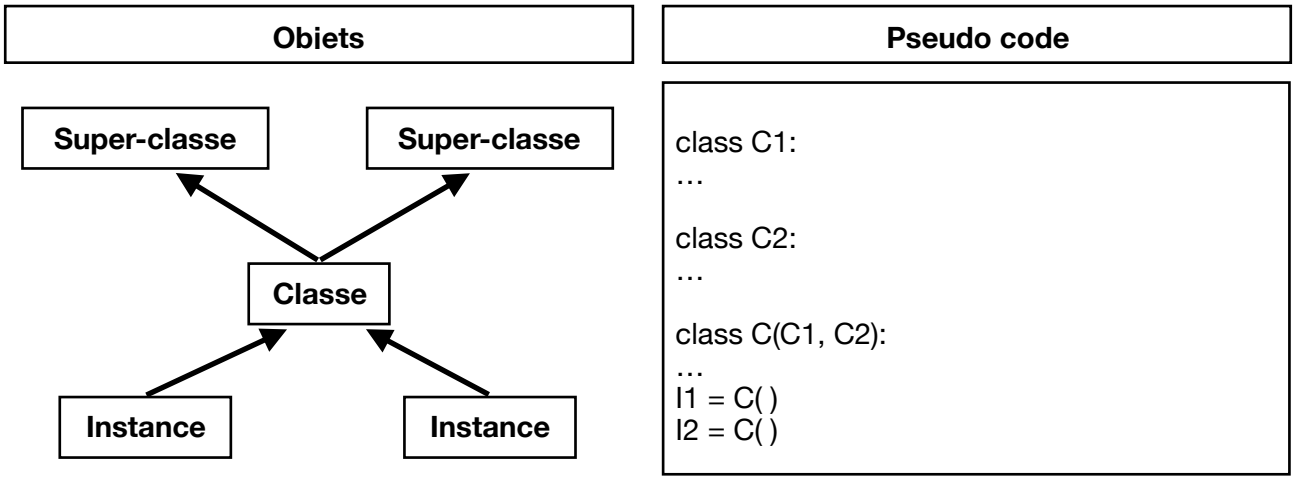
```
>>> print(p)
je
fais
un
mooc
sur
python
```

Leçon 34 : Héritage

Il y a une relation d'héritage entre l'instance et la classe qui crée cette instance. C'est-à-dire que lorsque l'on cherche un attribut dans une instance, si on ne le trouve pas dans l'espace de nommage de l'instance, on remonte l'arbre d'héritage et on arrive dans la classe.

Cette notion d'arbre d'héritage s'étend également aux classes. Les classes peuvent hériter d'autres classes. On peut donc avoir un arbre d'héritage qui va partir des instances, remonter aux classes et remonter aux classes dont héritent classes.

L'arbre d'héritage



Nous avons deux classes **C1** et **C2** ainsi qu'une troisième classe **C** qui hérite de **C1** et **C2**. Cette notion d'héritage est représentée par des parenthèses mises après le nom de la classe, qui contiennent la liste des classes dont hérite la classe : **class C(C1, C2)**.

On appelle ici **C** une **classe** et **C1** et **C2** les **super-classes** de **C**. On aurait tout aussi bien pu regarder le problème dans l'autre-sens, en considérant **C1** et **C2** comme deux **classes** et **C** leur **sous-classe**.

Ensuite si on créer les **instances I1** et **I2**, ces **instances** sont des objets qui vont hériter de la **classe**.

On remarque maintenant que les **instances**, la **classe** et les **super-classes** forment un **arbre d'héritage**. Donc lorsque l'on recherche un attribut dans l'**instance**, si on ne le trouve pas dans l'espace de nommage de l'**instance**, on remonte dans sa **classe** et si on ne trouve toujours pas l'attribut, on remonter dans les **super-classes**. Les **super-classes** étant des **classes** comme les autres, elles peuvent elles-mêmes hériter d'autres **classes**. On peut donc avoir un très grand **arbre d'héritage**.

La notion d'héritage

Reprenons la classe **Phrase** telle que définie à l'issue de la leçon 33 :

```
s = "Je fais un MOOC sur Python"

class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()

    def nb_lettres(self):
        return len(self.ma_phrase)

    def __len__(self):
        return len(self.mots)
```

```
def __contains__(self, mot):
    return mot in self.mots

def __str__(self):
    return "\n".join(self.mots)
```

Nous voulons créer une classe dotée des mêmes caractéristiques que **Phrase** mais qui permette de faire un test d'appartenance qui ne prend pas en compte la casse, c'est-à-dire qui ne prend pas en compte le fait que l'on a des lettres majuscules et minuscules.

Lorsque l'on dit que l'on voudrait avoir une classe qui se comporte comme une autre classe, c'est exactement ce que l'on appelle une relation d'héritage. Donc la bonne manière de définir une nouvelle classe qui se comporte comme **Phrase** mais qui modifie un peu son comportement, c'est d'hériter de **Phrase**.

Pour cela nous créons une nouvelle classe **PhraseSansCasse** qui va hériter de **Phrase** :

```
class PhraseSansCasse(Phrase):
    pass
```

On peut maintenant définir une instance **p_no** de notre classe **PhraseSansCasse** : **p_no = PhraseSansCasse(s)**. Maintenant on peut vérifier que **p_no** est une instance de la classe **Phrase** et de la classe **PhraseSansCasse** à l'aide de la fonction built-in **isinstance()** : **isinstance(p_no, Phrase)** renvoie **True** et **isinstance(p_no, PhraseSansCasse)** renvoie également **True**.

La fonction **isinstance()** permet de vérifier si un objet est une instance directement d'une classe ou une instance de super-classes le long de l'arbre d'héritage.

Si on regarde exactement l'objet **p_no**, on voit qu'il a été créé directement par la classe **PhraseSansCasse** :

```
>>> p_no
<__main__.PhraseSansCasse object at 0x105e67a90>
```

Nous allons maintenant ajouter une nouvelle méthode **__init__** dans **PhraseSansCasse** qui va définir un ensemble **self.mots_lower** qui va regrouper tous les mots en minuscule de la chaîne de caractères passée à notre classe :

```
class PhraseSansCasse(Phrase):
    def __init__(self, ma_phrase):
        self.mots_lower = {m.lower() for m in self.mots}
```

On remarque ici que l'on a une méthode **__init__** définie dans **Phrase** et une autre méthode **__init__** définie dans **PhraseSansCasse**. Lorsque l'on hérite d'une classe, on hérite de toutes ses méthodes mais si on redéfinit une méthode, on appelle ça **surcharger** une méthode, alors la méthode originale de la **super-classe** ne sera pas appelée automatiquement. En l'occurence, ici nous avons **surchargé** la méthode **__init__**, donc la méthode **__init__** de **Phrase** ne sera pas appelée automatiquement. Le seul moyen de l'appeler automatiquement c'est de le faire de manière explicite, et lorsque l'on a un initialisateur (**__init__**) c'est en général ce que l'on veut faire.

Ci-dessous nous appelons explicitement la méthode **__init__** dans la classe **Phrase** :

```
class PhraseSansCasse(Phrase):
    def __init__(self, ma_phrase):
        Phrase.__init__(self, ma_phrase)
        self.mots_lower = {m.lower() for m in self.mots}
```

Ainsi, nous avons **forcé l'appel de la méthode __init__ de Phrase** avant de faire l'initialisation spécifique à la classe **PhraseSansCasse**.

```
>>> p_no = PhraseSansCasse(s)
>>> p_no.mots_lower
{'sur', 'un', 'mooc', 'python', 'je', 'fais'}
```

Notre objectif est de faire un test d'appartenance sans prendre en compte la casse. Pour cela, nous définissons une nouvelle méthode **__contains__** qui **surcharge** la méthode **__contains__** définie dans la **super-classe** :

```
class PhraseSansCasse(Phrase):
    def __init__(self, ma_phrase):
        Phrase.__init__(self, ma_phrase)
        self.mots_lower = {m.lower() for m in self.mots}

    def __contains__(self, mot):
        return mot.lower() in self.mots_lower
```

Et si on exécute maintenant notre classe :

```
>>> p_no = PhraseSansCasse(s)
>>> 'Mooc' in p_no
True
```

Le test d'appartenance renvoie **True** alors que dans la chaîne de caractères référencée par la variable **s**, le mot **'Mooc'** est orthographié **'MOOC'**. Nous n'avons pas rappelé la méthode **__contains__** de la **super-classe Phrase** parce que ça n'avait pas de sens de l'appeler dans ce contexte.

Au final, nous pouvons définir une instance **p** de **Phrase** et définir une instance **p_no** de **PhraseSansCasse**.

Si on fait **'mooc' in p**, **p** étant une instance de **Phrase**, Python renvoie **False**. Par contre **'mooc' in p_no**, **p_no** étant une instance de **PraseSansCasse**, Python renvoie **True**.

Avec notre méthode **.nb_lettres()**, on peut vérifier combien on a de lettres dans notre phrase. Et on peut appeler cette méthode avec **p.nb_lettres()** aussi bien qu'avec **p_no.nb_lettres()**, car même si la méthode n'est pas définie dans **PhraseSansCasse**, la méthode étant définie dans la **super-classe Phrase**, l'instance **p_no** peut remonter l'arbre d'héritage pour récupérer cette méthode.

Leçon 35 : Héritage multiple et ordre de résolution des attributs

Lorsque l'on recherche un attribut, on appelle cela la recherche d'attribut mais également **la résolution d'attribut** pour exprimer le fait que le mécanisme de recherche va remonter l'arbre d'héritage. Par ailleurs, Python supporte ce que l'on appelle l'**héritage multiple** qui veut dire qu'une classe peut hériter de plusieurs classes.

Lorsque l'héritage est simple, la recherche ou la résolution des attributs est simple, on remonte simplement l'arbre d'héritage qui est juste constitué d'une seule ligne.

Lorsque l'héritage est multiple, l'arbre d'héritage va être plus complexe. Ça va être un **graph acyclique**. Il faut donc dans ce cas définir une méthode de résolution des attributs qui va définir un parcours spécifique dans notre **graph d'héritage**.

Nous allons ici expliquer les **méthodes de résolution d'attributs** appelés également **MRO** pour **Method Resolution Order**.

La super-classe object

Commençons par créer une classe **C** qui ne fait rien et une instance **c** qui référence la classe **C** :

```
class C:
    pass

c = C()
```

Pour savoir quelle est la classe qui a créé l'instance et quelles sont les super-classes de la classe, il y a des attributs spéciaux. Ainsi sur l'instance **c** on peut passer un attribut **.__class__** qui retourne une référence vers l'objet classe qui a créé l'instance. Ainsi **c.__class__** renvoie **__main__.C**.

Ensuite les classes ont aussi un attribut qui s'appelle **.__bases__** qui contient un tuple qui contient toutes les super-classes de la classe. Ici **C.__bases__** renvoie **(object,)**. Ce résultat peut sembler curieux puisqu'on a définit notre classe **C** sans aucune super-classe. Mais en fait, en Python, lorsque l'on crée une classe sans aucune super-classe, la classe hérite par défaut d'une classe particulière qui s'appelle **object**. **object** est la super-classe de toutes les classes en Python.

Pourquoi a-t-on besoin de cette super-classe **object** en Python ? C'est parce que quand on a une classe ou une instance, on peut faire un **print(instance)** ou **print(la_classe)** :

```
>>> print(c)
<__main__.C object at 0x10dd4fc50>
>>> print(C)
<class '__main__.C'>
```

Cela veut dire que la méthode **print()** est supportée par notre classe **C**, alors que l'on n'a défini aucune méthode à l'intérieur. En fait dans **object** sont implémentés un certain nombre de comportements par défaut. Donc lorsque ces méthodes et ces comportements ne sont pas surchargés dans nos classes, c'est l'implémentation d'**object** qui va être utilisée.

La méthode .mro()

Appelons la méthode **.mro()** sur notre classe **C** :

```
>>> C.mro()
[<class '__main__.C'>, <class 'object'>]
```

En fait **.mro()** retourne une liste qui indique dans quel ordre on doit résoudre nos attributs lorsque l'on arrive dans notre classe. En l'occurence, nous devons d'abord rechercher dans l'espace de nommage de **C** et si on ne le trouve pas, on va cherche notre attribut dans l'espace de nommage de **object**.

Maintenant nous créons deux classes **SuperA** et **SuperB** et une troisième classe **C** qui hérite de **SuperA** et de **SuperB** :

```
class SuperA:
    pass

class SuperB:
    pass

class C(SuperA, SuperB):
    pass
```

Maintenant nous appelons la méthode **.mro()** sur la classe **C** :

```
>>> C.mro()
[<class '__main__.C'>, <class '__main__.SuperA'>, <class '__main__.SuperB'>, <cl
ass 'object'>]
```

Nous voyons que l'ordre de résolution des attributs commence par la classe **C**, suivi par la super-classe **SuperA**, ensuite la super-classe **SuperB** et ensuite **object**.

Si on modifie la classe **C** en mettant **SuperB** avant **SuperA** :

```
class C(SuperB, SuperA):
    pass
```

Et que l'on rappelle la méthode **.mro()** sur **C** :

```
>>> C.mro()
[<class '__main__.C'>, <class '__main__.SuperB'>, <class '__main__.SuperA'>, <cl
ass 'object'>]
```

On voit que maintenant l'ordre de résolution commence par la classe **C**, suivi par la super-classe **SuperB**, ensuite **SuperA** et ensuite **object**.

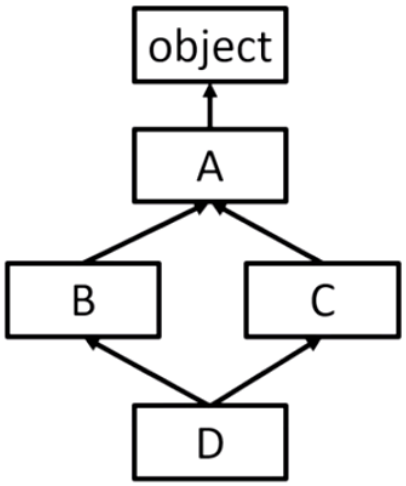
Cela veut donc dire que l'ordre de résolution des attributs va dépendre de l'ordre dans lequel on définit les super-classes d'une classe. Lorsque l'on fait de l'**héritage multiple**, il est donc très important de prendre conscience de ce phénomène et de le prendre en compte lorsque l'on définit les super-classes d'une classe.

Fonctionnement de l'algorithme de recherche des attributs en Python

Regardons maintenant l'algorithme utilisé par Python pour trouver ce parcours de recherche des attributs.

Nous prenons une classe **A** qui par défaut hérite de **object**, puis une classe **B** qui hérite de **A**, une classe **C** qui hérite de **A** et enfin une classe **D** qui hérite de **B** et de **C**.

```
>>> class A: pass
>>> class B(A): pass
>>> class C(A): pass
>>> class D(B, C): pass
```



On a ici typiquement ce que l'on appelle un **diagramme en diamant**. En fait pour parcourir les classes, on va parcourir d'abord de bas en haut puis de gauche à droite : **D**, **B**, **A**, **object** puis **C**, **A**, **object**.

Et ensuite, si jamais on a des classes dupliquées, on enlève toutes les classes dupliquées sauf la dernière. Donc on enlève la première occurence de **A** qui est présent deux fois et la première occurence d'**object** qui est aussi présent deux fois.

Donc si on regarde regarde la **.mro** effectivement calculée par Python, on obtient bien la classe **D**, puis la classe **B**, la classe **C**, la classe **A** et finalement **object**.

```
>>> D.mro()
[__main__.D, __main__.B,
__main__.C, __main__.A,
object]
```

Il est très important de prendre conscience que l'on a une technique particulière de recherche des attributs en cas d'**héritage multiple**. Imaginons que nous avons une classe héritière de deux super-classes et que l'on appelle une méthode définie dans les deux super-classes. Si on ne connaît pas l'ordre de résolution des attributs, alors on ne sera pas capable de savoir laquelle des méthodes va être appelée dans l'une ou l'autre de nos deux super-classes.

Leçon 36 : Variables et attributs

Nous avons vu dans les précédentes leçons, la notion de **portée de variable**, notamment dans le contexte des fonctions et des modules, et nous avons aussi vu la notion de recherche d'attribut le long des **arbres d'héritage**. Tous ces mécanismes servent à savoir dans quel espace de nommage chercher nos variables et nos attributs.

Le but de cette leçon est de faire toute la lumière sur les mécanismes de référencement et d'affectation des variables et des attributs quelque soit le contexte.

La distinction entre variables et attributs

On dit que l'on a une variable lorsqu'un nom est référencé ou affecté directement **variable = objet**.

On dit que l'on a un attribut lorsque ce nom est référencé ou affecté en utilisant la notation **objet.attribut**.

Pourquoi faire une distinction entre variable et attribut ? C'est que la recherche de ces noms utilise des mécanismes complètement différents suivant que l'on a une variable ou un attribut :

- lorsque l'on a une variable, on utilise un mécanisme qui s'appelle la **liaison lexicale** qui est un mécanisme statique qui est défini à l'écriture du programme.
- lorsque l'on a un attribut, la recherche de ce nom va être fait en utilisant un mécanisme que l'on appelle la **résolution d'attribut**.

Il est important de comprendre que ces mécanismes de recherche ont pour but de définir dans quel espace de nommage ce nom a été défini.

Les attributs

Lorsqu'on définit un attribut **x** : **objet.x = 10**, on dit que ce nom **x** doit être défini dans l'espace de nommage de l'**objet**. Puis lorsque l'on référence l'attribut **x** : **print(objet.x)**, on dit que l'on va chercher notre attribut dans l'espace de nommage de l'**objet**.

Mais on a deux cas différents, selon la nature de **objet** :

1) si **objet** est un **module**, une référence du nom **x** dans l'**objet** veut dire que l'on va chercher **x** dans l'espace des variables globales du module **objet**
2) si **objet** est une **classe** ou une **instance**, on va chercher l'attribut **x** le long de l'arbre d'héritage

Les variables

Une variable est définie dans un bloc de code d'une **fonction**, d'une **classe** ou d'un **module** lorsqu'elle est écrite directement dans un bloc de code de la **fonction**, de la **classe** ou du **module** et on dit alors qu'elle devient locale à ce bloc de code.

La liaison d'une variable :

Les variables sont liées aux espaces de nommage statiquement en fonction de là où elles sont écrites au moment de l'écriture de notre programme avec le mécanisme de **liaison lexicale**.

La seule exception est lorsque l'on déclare notre variable comme étant **global** ou **nonlocal**.

On a différentes manières de faire une définition, on dit également une liaison d'une variable :

- la définition explicite : **x = 1** permet à la fois à la variable **x** de référencer l'objet **1**, mais également de lier la variable **x** au bloc de code dans lequel elle est déclarée
- la déclaration de paramètres dans une entête de fonction : **def f(a):**, la variable **a** définie ici comme paramètre de la fonction **f()** est une variable locale liée au bloc de code de la fonction
- les définitions de classes et de fonctions : **class C():** et **def f():**, **C** étant un nom qui va référencer l'objet **classe** et **f** étant un nom qui va référencer l'objet **fonction**

- les boucles for : **for i in obj:**, ici **i** est une variable qui va référencer les objets retournés par la méthode **next()** sur l'itérateur de l'objet
- les imports : **import sys** et **from os import times**, lorsque l'on fait un import on définit un nom qui va référencer un objet, soit un objet **module**, soit un objet **fonction**

ATTENTION! La liaison est le mécanisme qui lie une variable à un bloc de code et la référence est le mécanisme qui associe une variable à un objet.

Le référencement d'une variable :

Une variable est donc liée avec différents mécanismes. On définit un nom qui est lié à un objet. Maintenant, on référence une variable lorsque par exemple on fait **print(x)**, c'est-à-dire lorsque l'on utilise cette variable **x**.

La question, c'est dans quel espace de nommage va-t-on prendre cette variable **x** ?

Pour répondre à cette question, nous avons recours à la règle **LEGB** : nous allons d'abord chercher la variable localement à la fonction, puis on remonte aux fonctions englobantes, ensuite on va dans le module et finalement on la cherche dans le module built-in.

Mais lorsque l'on rajoute le cas des classes, il y a un cas particulier supplémentaire à prendre en compte. Le bloc de code des classes est systématiquement sauté lors de la résolution des attributs. En fait, le bloc de code des classes est très particulier. **Une variable définie dans le bloc de code d'une classe n'est pas accessible en dehors d'une classe mais elle n'est pas non plus accessible aux méthodes de la classe.** Cela peut paraître étonnant de prime abord, mais ce choix architectural a été fait pour éviter d'avoir une interaction bizarre entre le mécanisme de résolution d'attribut et le mécanisme de recherche des variables. Les classes et les instances sont naturellement construites pour rechercher des attributs le long des arbres d'héritage.

Cas pratiques

On définit une variable globale **a = 1**, une fonction **f** qui définit une variable locale **a = 2** et une classe **C** qui définit une variable **a = 3** dans le bloc de code de la classe **C**. Ensuite on appelle la fonction **f()**, on crée une instance de la classe **ins = C()** et on fait **print(a)**. La question est que va afficher **print(a)** ?

```
In [1]: a = 1

In [2]: def f():
   ...:     a = 2
   ...:

In [3]: class C():
   ...:     a = 3
   ...:

In [4]: f()

In [5]: ins = C()

In [6]: print(a)
```

On sait que les variables définies dans les fonctions sont locales aux fonctions et que les variables définies dans les classes ne peuvent pas être vues en dehors de la classe. Donc on a la certitude que **print(a)** va afficher la variable globale **a = 1** :

```
In [6]: print(a)
1
```

Maintenant prenons un deuxième cas :

```
In [1]: a = 1

In [2]: class C():
   ...:     a = 2
   ...:     def f(self):
   ...:         print(a)
   ...:         print(C.a)
   ...:

In [3]: ins = C()

In [4]: ins.f()
1
2
```

Ici **print(a)** est une variable que l'on recherche avec la règle **LEGB**. Est-ce que **a** est définie localement ? Non. Il n'y a pas de fonction englobante. **a** est définie globalement, le programme affiche donc l'entier **1**. On voit ici que dans ce mécanisme de résolution de recherche des variables, on a sauté le bloc de code de la classe.

Ensuite, on fait **print(C.a)**, et on change donc de mécanisme : nous avons maintenant une **résolution d'attribut**. **C** est une classe, on va donc chercher l'attribut le long de l'arbre d'héritage. **a** est défini dans l'espace de nommage de la classe **C**, donc **print(C.a)** renvoie l'entier **2**. On voit donc que les classes sont conçues pour qu'on accède à leurs attributs avec le mécanisme de résolution d'attribut, c'est-à-dire la notation **C.attribut**.

Troisième exemple :

```
In [5]: a = 1

In [6]: class A:
   ...:     a = 2
   ...:     class B:
   ...:         def f(self):
   ...:             print(a)
```

```
In [7]: ins = A.B()

In [8]: ins.f()
1
```

Ici, dans **print(a)**, **a** est une variable, on la recherche donc avec la règle **LEGB**. **a** n'est pas définie localement à la fonction. On n'a pas de fonction englobante. **a** est référencée globalement, c'entier **1**, donc **print(a)** affiche l'entier **1**.

Leçon 37 : Conception d'itérateurs

Pour rappel, un **itérable** est un objet qui a une méthode **.__iter__()** qui retourne un **itérateur** et un **itérateur** est un objet qui a une méthode **.__iter__()** qui retourne lui-même et une méthode **.__next__()** qui à chaque fois qu'on l'appelle va retourner un nouvel élément jusqu'à ce qu'il n'y ait plus d'élément, auquel cas on a l'exception **StopIteration**.

Ce mécanisme est extrêmement puissant et peut être exploité par n'importe quel mécanisme d'**itération** en Python : les boucles **for**, les **compréhensions**, les fonctions **map** et **filter**. Nous allons voir dans cette leçon comment implémenter des **itérateurs** et des **itérables** pour nos propres objets.

Définir une classe comme itérateur

Nous commençons par créer une classe **Phrase** :

```
s = "je fais un MOOC sur Python"

class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()
```

L'object est de définir cet objet pour qu'il devienne un **itérateur**. Pour cela, nous lui donnons une méthode __**iter**__ et une méthode __**next**__ :

```
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()

    def __iter__(self):
        return self

    def __next__(self):
        if not self.mots:
            raise StopIteration
        return self.mots.pop(0)
```

Nous avons donc une méthode __**iter**__ qui retourne l'objet lui-même. Puis nous avons une méthode __**next**__ qui doit retourner à chaque appel un élément suivant, ici on veut qu'il retourne un mot de la phrase à chaque tour de boucle, et lorsque l'on n'a plus de mot, on veut avoir une exception **StopIteration**. Avec **if not self.mots: raise StopIteration** qui veut dire que si on n'a plus de mot on fait un **raise** de l'exception **StopIteration**. Et si on a encore des mots dans la classe alors on fait un **return self.mots.pop(0)**, avec la méthode **.pop(0)** qui permet de retourner le premier mot de la liste et l'enlever de la liste. Donc à chaque tour, notre liste va se raccourcir jusqu'à ce qu'on n'ait plus de mots.

Si on évalue maintenant ce code avec **p = Phrase(s)**, on peut maintenant faire une **compréhension** : **[m for m in p]** qui renvoie **['Je', 'fais', 'un', 'MOOC', 'sur', 'Python']**. Comme notre classe **Phrase** a été définie comme un **itérateur**, on ne peut pas renouveler l'opération. Si on refait une **compréhension** : **[m for m in p]**, le programme renvoie cette fois une liste vide **[ ]**. Et si on force la méthode **next()** avec **next(p)**, le programme renvoie l'erreur **StopIteration**.

Définir une classe comme itérable

Nous allons maintenant définir notre classe **Phrase** comme **itérable**.

On modifie la méthode __**iter**__ pour qu'au lieu de renvoyer l'objet lui-même, elle retourne un nouvel objet **IterPhrase** qui est un objet **itérateur** sur **Phrase** auquel on passe notre liste de mots : **IterPhrase(self.mots)**.

```
def __iter__(self):
    return IterPhrase(self.mots)
```

Puis on définit l'objet **IterPhrase** qui est une nouvelle classe et qui va définir une méthode **__init__** qui prend **self** et **mots** et qui va dire **self.mots = mots[:]** (*Shallow copy*).

Ensuite on définit une méthode **__iter__** puisque tous les **itérateurs** doivent avoir une méthode **__iter__**, qui prend **self** et qui va retourner l'**itérateur** lui-même.

Puis on rajoute la méthode **__next__** qui est exactement la même que précédemment.

```
class IterPhrase:
    def __init__(self, mots):
        self.mots = mots[:]
    def __iter__(self):
        return self
    def __next__(self):
        if not self.mots:
            raise StopIteration
        return self.mots.pop(0)
```

Le résultat de notre code complet ci-dessous :

```
s = "je fais un MOOC sur Python"
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()
    def __iter__(self):
        return IterPhrase(self.mots)
class IterPhrase:
    def __init__(self, mots):
        self.mots = mots[:]
    def __iter__(self):
        return self
    def __next__(self):
        if not self.mots:
            raise StopIteration
        return self.mots.pop(0)
```

Pourquoi avoir défini une **shallow copy** dans la méthode **__init__** de **IterPhrase** ? En fait, on remarque que la méthode **__next__** fait un **.pop(0)** sur **mots**, qui veut dire qu'elle va modifier l'objet référencé par **mots**.

Si on référençait l'objet contenu dans la classe **Phrase**, ça voudrait dire qu'une fois que l'**itérateur** a parcouru une fois la liste des mots, l'attribut **.mots** référencerait une liste vide. Aussi, il faut donc que l'on fasse une **shallow copy** pour que ce soit cette copie qui va réduire jusqu'à ce qu'on n'ait plus d'élément.

Si on exécute maintenant le code, que l'on crée une instance **p = Phrase(s)**, on peut faire une **compréhension** : **[m for m in p]** qui renvoie **['Je', 'fais', 'un', 'MOOC', 'sur', 'Python']**. Et on peut maintenant appeler cette **compréhension** autant de fois que l'on veut.

Créer un objet itérable avec les fonctions génératrices

Nous revenons sur notre classe **Phrase** et au lieu de définir une classe **IterPhrase**, on va simplement modifier notre méthode __**iter**__ pour être une **fonction génératrice** :

```
class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()

    def __iter__(self):
        for m in self.mots:
            yield m
```

Et maintenant nous avons une classe **Phrase** qui est **itérable**. La méthode __**iter**__ va produire les **itérateurs**. Et pour rappel, à chaque fois qu'on appelle une **fonction génératrice**, cela crée un nouvel **itérateur**, la **fonction génératrice** va donc ici produire automatiquement des **itérateurs**. Cette fonction va parcourir la liste des mots et à chaque tourne elle retourne un nouvel élément de mots. Cette méthode ne modifie pas la liste de mots **self.mots**, elle ne fait que la parcourir, nous n'avons donc rien d'autre à faire.

Si on exécute maintenant le code, que l'on crée une instance **p = Phrase(s)**, on peut faire une **compréhension** : **[m for m in p]** qui renvoie **['Je', 'fais', 'un', 'MOOC', 'sur', 'Python']**. Et on peut maintenant appeler cette **compréhension** autant de fois que l'on veut.

Leçon 38 : Conceptions d'exceptions personnalisées

Créer une exception est extrêmement simple à faire et cela permet d'avoir une exception qui correspond vraiment à notre besoin. En Python, toutes les exceptions que l'on crée doivent hériter de la classe exception ou alors d'une de ses sous-classes. En fait les exceptions créent un arbre d'héritage.

Créer une exception

Nous reprenons notre classe **Phrase** pour lui ajouter un mécanisme de contrôle.

```
s = "Je fais un MOOC sur Python"

class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        self.mots = ma_phrase.split()
```

Le mécanisme que l'on peut ajouter ici c'est : si on crée une phrase avec une phrase tout va fonctionner normalement mais si la phrase est vide c'est qu'il y a un problème et la bonne manière de faire est alors de lancer une **exception**. Cette **exception** ne veut pas dire que le programme va s'arrêter. C'est une manière de communiquer qu'on a un comportement spécifique dans le cadre de l'exécution de notre programme.

Pour créer une **exception** en Python, il faut définir une nouvelle classe avec un nom explicite. Les **exceptions** doivent toujours avoir un nom explicite et il est recommandé de toujours terminer le nom par le mot **Error**. Et l'**exception** doit forcément hériter de la classe **Exception**. Dans 99% des cas, on n'a rien à mettre dans la classe. Il faut simplement mettre **pass** pour que la classe syntaxiquement valide. La classe **exception** est une vraie classe, donc on peut définir ce que l'on veut dedans. Mais dans un usage standard, il n'y a pas besoin d'attribut particulier. Et pour l'utiliser nous rajoutons un test **if** dans notre classe **Phrase**.

```
s = "Je fais un MOOC sur Python"

class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        if not ma_phrase:
            raise PhraseVideError()
        self.mots = ma_phrase.split()

class PhraseVideError(Exception):
    pass
```

Si maintenant, on exécute le code, on crée une instance **p** de **Phrase** en lui passant une chaîne de caractères vide, alors nous avons une exception qui est lancée.

```
>>> p = Phrase('')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    p = Phrase('')
  File "/Users/mathieulehot/Documents/mooc/python/mooc.py", line 7, in __init__
    raise PhraseVideError()
PhraseVideError
```

Exceptions et arguments

Les **exceptions** en Python peuvent prendre des arguments. On peut passer une liste d'arguments séparés par une virgule à notre exception **PhraseVideError**. Tous ces arguments vont être mis dans un tuple que l'on pourra ensuite manipuler lorsque l'on capturera notre **exception**. Ici nous passons une phrase d'erreur d'erreur **'phrase vide'** et un code d'erreur **18**.

```
s = “Je fais un MOOC sur Python“

class Phrase:
    def __init__(self, ma_phrase):
        self.ma_phrase = ma_phrase
        if not ma_phrase:
            raise PhraseVideError(‘phrase vide’, 18)
        self.mots = ma_phrase.split()

class PhraseVideError(Exception):
    pass
```

Si maintenant, on exécute le code, on crée une instance **p** de **Phrase** en lui passant une chaîne de caractères vide, alors on voit une exception et nos arguments passés à l’instance sont affichés au moment de l’exception.

```
>>> p = Phrase('')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    p = Phrase('')
  File "/Users/mathieulehot/Documents/mooc/python/mooc.py", line 7, in __init__
    raise PhraseVideError('Phrase vide', 18)
PhraseVideError: ('Phrase vide', 18)
```

On peut les manipuler avec un **try except**. Avec **except PhraseVideError as e**, le **as e** veut dire que **e** va référencer l’instance de l’objet **exception** qui a été lancé. Et dans cet objet **exception**, on a un argument qui s’appelle **args** qui est simplement un tuple qui va référencer le tuple que l’on a passé lorsque l’on a construit notre exception.

```
try:
    Phrase(‘’)
except PhraseVideError as e:
    print(e.args)
```

Et lorsqu’on exécute le code, on va apparaître le tuple qui contient comme premier élément le message d’erreur et comme deuxième élément le message d’erreur.

```
========= RESTART: /Users/mathieulehot/Documents/mooc/python/mooc.py =========
('Phrase vide', 18)
```

Leçon 39 : Conception de contexte manager

En programmation, il est courant de vouloir faire des opérations de finalisation. Par exemple, lorsque l'on a un objet fichier dont on a plus besoin, on doit le fermer. Ou lorsque l'on a une socket, on doit également la fermer. L'idée c'est de libérer des ressources lorsqu'on n'a plus besoin de cet objet.

Ces opérations de finalisation doivent être faites même si on a une exception en cours d'exécution. L'idée c'est : si notre programme plante, on finalise nos objets avant que le programme ne s'arrête définitivement.

En Python il y a une manière de faire ça avec l'instruction **try finally** :

try:
    <bloc de code>
finally:
    <bloc de code>

Il s'agit en fait d'un **try except** qui a une clause **finally** qui va s'exécuter que l'on ait ou non une exception dans le bloc de code **try**. On a donc la certitude que le bloc de code du **finally** s'exécutera quelque soit l'exécution du bloc de code du **try**.

Il y a cependant ici un problème : lorsque l'on fait de la finalisation, on demande à la personne qui utilise l'objet de penser à faire un **try finally** et surtout cette personne doit connaître les bonnes opérations de finalisation à faire.

Il serait donc beaucoup plus pratique de laisser ces opérations de finalisation à celui qui a conçu l'objet et d'offrir un mécanisme pour dire "je n'ai plus besoin de l'objet", et qu'automatiquement ces opérations de finalisation s'exécutent. C'est exactement ce que permet un **context manager**.

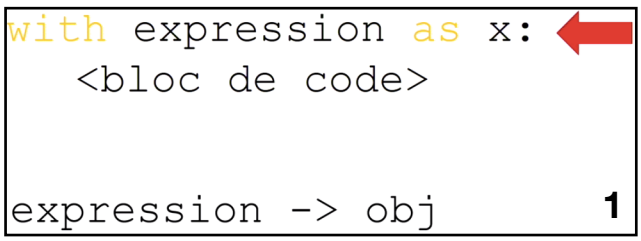
Nous allons voir dans cette leçon comment concevoir des **context manager** pour nos propres objets.

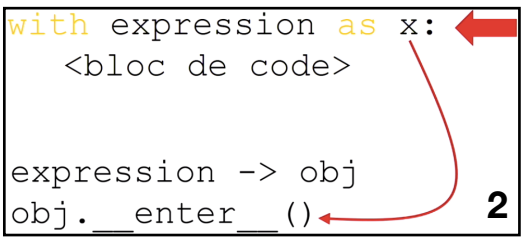
La syntaxe d'un context manager

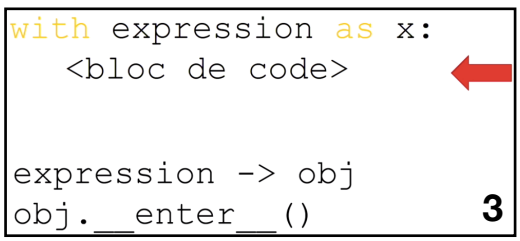
La syntaxe d'un **context manager** consiste à utiliser **with expression as x:** suivi d'un bloc de code :

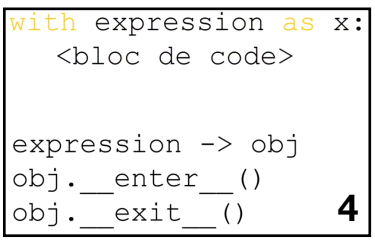
with expression as x:
    <bloc de code>

Lorsque l'on évalue notre **context manager**, l'expression va retourner un objet. Et cet objet va être un objet qui implémente un protocole qu'on appelle le protocole de **context manager**. Dans ce protocole, on a une méthode **__enter__()** qui va retourner un objet qui va être référencé par la variable **x**. Ensuite on évalue le bloc de code. Et ensuite, en sortie du bloc de code, ou si on a une exception lors de l'évaluation de ce bloc de code, on va dans tous les cas exécuter une méthode **__exit__()** sur cet objet.









Le protocole de **context manager** ce sont donc ces deux méthodes **__enter__()** et **__exit__()** qui sont automatiquement exécutées lorsque l'on a l'instruction **with**.

La création d'un context manager

Nous allons illustrer la création d'un **context manager** avec l'implémentation d'une classe que l'on va appeler **Timer**. L'objectif de notre classe **Timer** sera d'évaluer le temps d'exécution d'un bloc de code qui est à l'intérieur d'une instruction **with**.

Pour cela nous commençons par importer le module **time** qui permet d'avoir un temps courant, puis on crée notre classe **Timer**. Dans cette classe on définit notre protocole de **context manager**.

Il faut donc une méthode **__enter__()** qui prend **self** comme argument et qui doit faire quelque chose dans notre bloc de code pour ensuite retourner un objet qui sera référencé par la variable qui suit le **as**. En général, les **context manager** retournent l'objet lui-même, mais on pourrait imaginer retourner autre chose. Ici, nous retournons l'objet lui-même. Nous définissons **self.start**, un attribut dans notre instance, qui est égal à **time.time()**, c'est-à-dire le temps courant. Puis on fait un **return self**.

Ensuite, nous définissons une deuxième méthode **__exit__()** qui est la méthode qui sera exécutée lorsque l'on sortira du bloc de code du **context manager** ou si on a une exception. La méthode **__exit__()** prend comme arguments **self** et un attribut ***args**. Puis nous définissons un attribut **duree** qui est égal à **time.time() - self.start**, donc le temps que l'on avait au démarrage que l'on soustrait au temps courant pour ainsi obtenir le temps d'exécution. L'attribut **duree** est affiché avec **print(f"{duree}s")**. Et enfin, notre méthode **.__exit__()** doit retourner un booléen : soit **False**, soit **True**. Si on retourne **False**, cela veut dire qu'en cas d'exception, l'exception va être remontée et va faire arrêter le programme. Si on retourne **True**, cela veut dire qu'en cas d'exception, l'exception va être capturée par l'instruction **with** et par conséquent l'exception ne sortira pas.

RAPPEL ! *D'une manière générale, c'est une bonne pratique de laisser remonter les exceptions sauf si on sait précisément ce que l'on fait.*

Nous terminons notre classe **Timer** par la définition avec la méthode appelée **__str__()** qui est appelée par la fonction built-in **print()**. Ici **__str__()** prend comme argument **self** et contient un attribut **duree** égal à **time.time() - self.start**. La méthode retourne **f"intermédiaire: {duree}s"**. Grâce à cette méthode, lorsque l'on appellera **print** sur notre objet qui implémente le protocole de **context manager**, on va avoir un temps intermédiaire d'exécution.

```
import time

class Timer:
    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, *args):
        duree = time.time() - self.start
        print(f"{duree}s")
        return False

    def __str__(self):
        duree = time.time() - self.start
        return f"intermédiaire: {duree}s"
```

Une fois la casse évaluée, nous pouvons désormais définir notre instruction **with** :

```
with Timer() as t:

    sum(x for x in range(10_000_000)
    print(t)
    sum(x**2 for x in range(10_000_000)
```

Ici la variable **t** prend la valeur de retour de la méthode **__enter__()**, c'est donc l'instance de **Timer** que l'on récupère. Dedans, nous faisons une somme avec une expression génératrice **sum(x for x in range(10_000_000)**. Ensuite, **print(t)** affiche un temps intermédiaire. Enfin, nous faisons une deuxième compréhension avec **x**2** au lieu de **x** : **sum(x**2 for x in range(10_000_000)**.

Si maintenant nous exécutons ce code, lorsque nous allons rentrer dans notre bloc de code, l'attribut **start** de la classe **Timer** va être initialisé au temps courant. Ensuite on exécute la somme de l'expression génératrice **sum(x for x in range(10_000_000)**. Ensuite on affiche un temps intermédiaire avec **print(t)** puisque **print** appelle la méthode **__str__** qui affiche le temps intermédiaire. Puis on calcule de nouveau une deuxième somme **sum(x**2 for x in range(10_000_000)**. Et en sortie de notre bloc de code, on va afficher le temps d'exécution.

```
================== RESTART: C:\Users\alegout\De
sktop\mooc.py ==================
intermédiaire: 0.7496917247772217s
4.020910024642944s
```

Maintenant si on ajoute une exception :

with Timer() as t:

sum(x for x in range(10_000_000)
print(t)
1/0
sum(x2 for x in range(10_000_000)**

1/0 va provoquer l'exception **ZeroDivisionError**

On voit dans le résultat ci-dessous, que le temps intermédiaire s'affiche bien correctement, tout comme le temps final. Les deux temps sont évidemment proches puisque le bloc d'instruction s'est arrêté au moment de l'exception et on n'a donc pas eu le temps d'évaluer la dernière expression génératrice. On voit cependant que la méthode **__exit__()** a bien été appelée bien que l'on ait une exception.

```
================== RESTART: C:\Users\alegout\De
sktop\mooc.py ==================
intermédiaire: 0.7342061996459961s
0.7498579025268555s
Traceback (most recent call last):
  File "C:\Users\alegout\Desktop\mooc.py", lin
e 21, in <module>
    1/0
ZeroDivisionError: division by zero
```