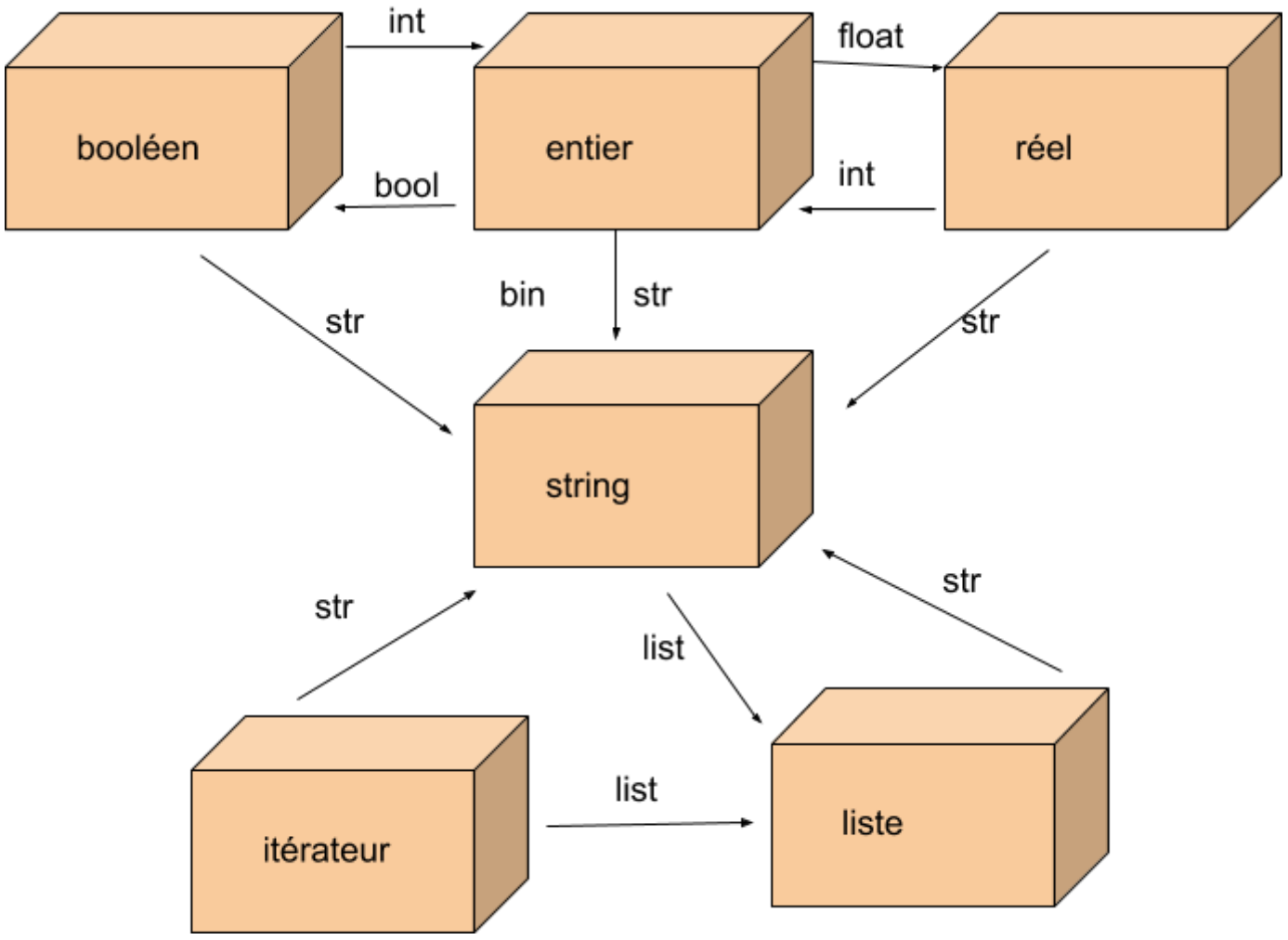
Notions&Exercices.18.11.py

Sommaire

I. “Hello Python”

T’es pas mon type? Alors change !

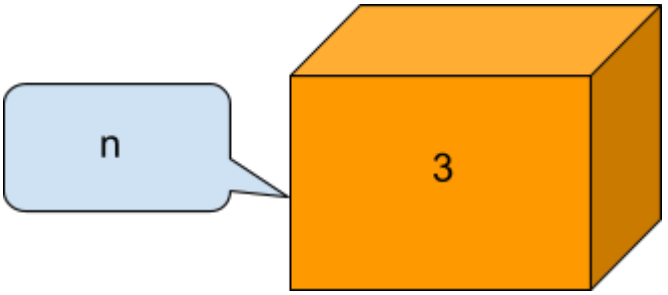
Pour modifier le type d’une variable, on lui applique une fonction, qui porte le même nom que le type de destination



II. Concepts

A. Variables et affectation

Dans Python, tout est objet. On peut assimiler un objet à une boîte contenant quelque chose qui dépend de son type (un entier si l'objet est de type *int*, un réel si l'objet est de type *float*, voire d'autres boîtes si l'objet est une liste par exemple). Une *variable* est un objet doté d'(au moins) un nom: On peut imaginer qu'on a collé une étiquette sur la boîte: *Affecter* une variable, c'est coller une étiquette sur la boîte.



Ci-dessus, la variable est de type *int* (c'est un entier). Noter que Python le sait sans qu'on aie eu à le lui dire. 3 est la valeur de la variable et *n* est son nom. Cette situation a été obtenue à l'aide du script suivant:

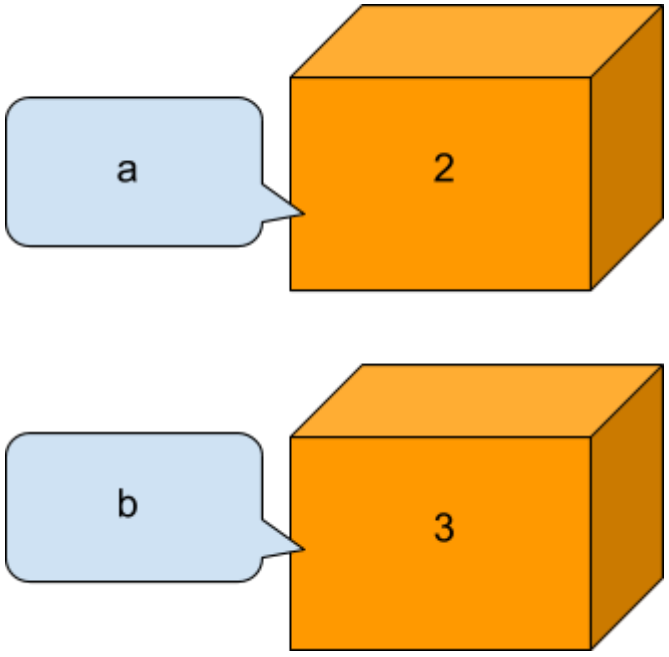
john.doe@linux-is-great:~

```
>>> n=3 # le signe = réalise l'affectation
>>> print(n) # permet de voir que (la valeur de) n vaut 3
3
>>> print(type(n)) # permet de voir que (la valeur de) n est un entier
<type 'int'>
>>> print(globals()) # parmi les variables globales on voit 'n':3 qui décrit
l'état actuel de n
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'n':
3, '__doc__': None, '__package__': None}
```

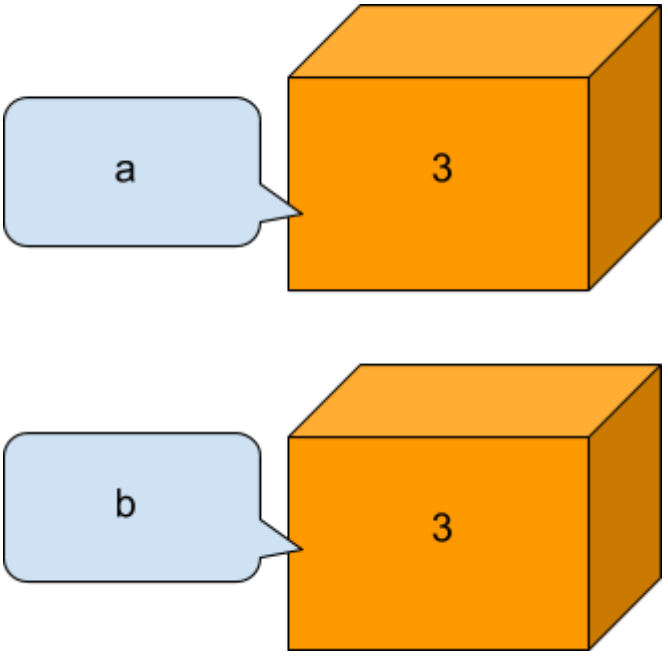
Si une boîte est vide, on considère qu'elle contient quand même quelque chose et ce vide contenu dans la boîte est noté `None` en Python. *globals()* est une fonction qui, à chaque étiquette, associe le contenu de la boîte sur laquelle elle est collée. Ce n'est ni une surjection (la boîte existait avant qu'on lui colle une étiquette), ni une injection (on peut coller plusieurs étiquettes sur une même boîte). Une petite expérience permet de vérifier ça:

```
>>> a=2 # on a collé l'étiquette a sur l'entier 2
>>> print(id(a)) # chaque boîte a une adresse (là où elle est)
123400
>>> b=3 # en collant l'étiquette b sur la boîte 3
>>> print(id(b)) # on a 2 boîtes à des endroits différents
123416
>>> a=3 # on a collé l'étiquette a sur l'entier 3
>>> print(id(a)) # et du coup l'étiquette a déménagé
123416
```

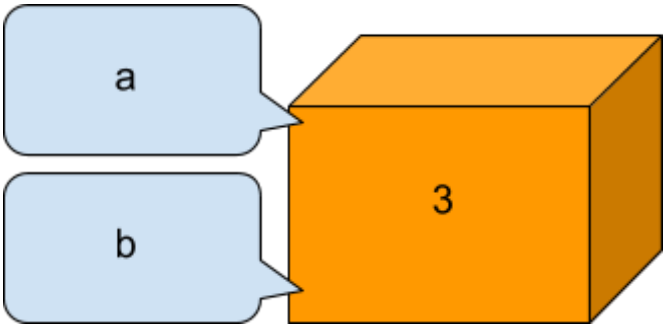
Après les créations des variables a et b on a cette situation:



Après avoir mis 3 dans a (et ainsi avoir réalisé par accident l'égalité entre a et b), on n'a donc pas cette situation classique



Mais celle-là, typique de Python:



Par exemple l'incrémentation a=a+1 a le même effet, par la suite d'opérations suivantes (on suppose que la valeur de a est 2):

- Python lit de droite à gauche, et crée une nouvelle boîte de type entier (puisque le contenu de a et 1 sont tous deux des entiers, il en est de même pour leur somme);
- Python calcule le contenu de cette nouvelle boîte, c'est 2+1=3 puisque "a" désigne, à droite du "=" de l'affectation, le contenu (et pas le nom) de la variable de nom a.
- Le "a=" de l'affectation amène alors à coller l'étiquette "a" sur cette nouvelle boîte. Ce qui nécessite de la décoller, avant cela, de l'ancienne boîte.

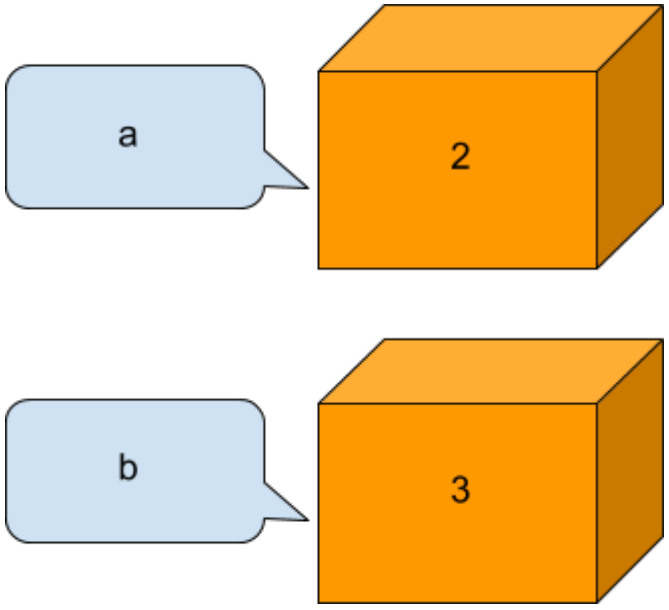
Peu après, un monstre pythonien appelé garbage collector, qui passe son temps à chercher puis exterminer les boîtes sans étiquette, aperçoit la pauvre boîte contenant 2, et désormais privée de son ancienne étiquette a. Impitoyable, il pulvérise ce 2 et ne reste plus qu'une boîte de contenu 3 et d'étiquette a.
L'effet final est le même que si on avait directement modifié le contenu de la boîte sans transférer une étiquette. Sauf qu'il y a eu un déménagement qui montre la particularité du fonctionnement de Python.

⚠ Attention !

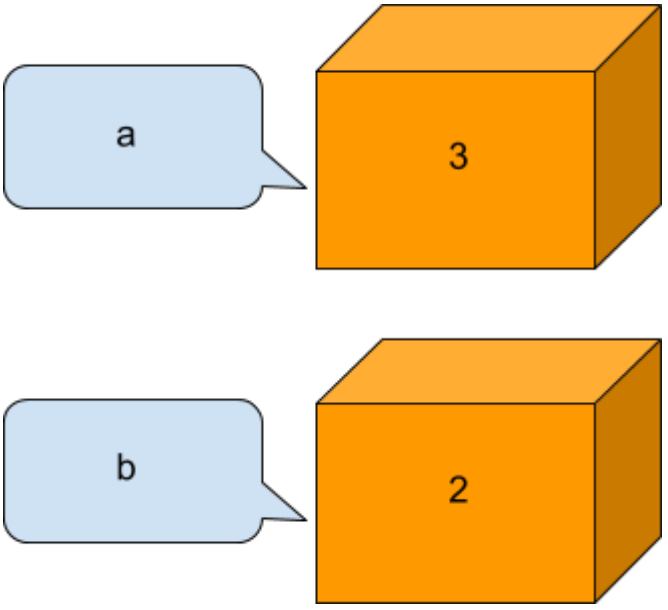
En fait, sur les objets mutables (par exemple les listes) on peut avoir des surprises...

```
>>> L=list(range(3)) # les trois entiers contenus dans L s'appellent
L[0], L[1] et L[2]
>>> print(L) # chaque boîte a une adresse (là où elle est)
[0,1,2]
>>> M=L # La liste [0,1,2] a maintenant 2 étiquettes: L et M
>>> M.append(5) # on croit avoir ajouté 5 à la liste M
>>> print(L) # mais L aussi a changé !
[0,1,2,5]
```

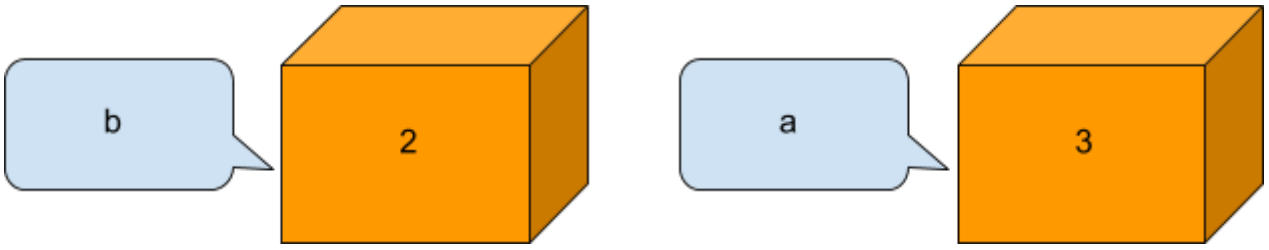
Comment passer de cette situation



à celle-là?



La philosophie de Python est basée sur le fait que la situation précédente est équivalente à celle-ci:

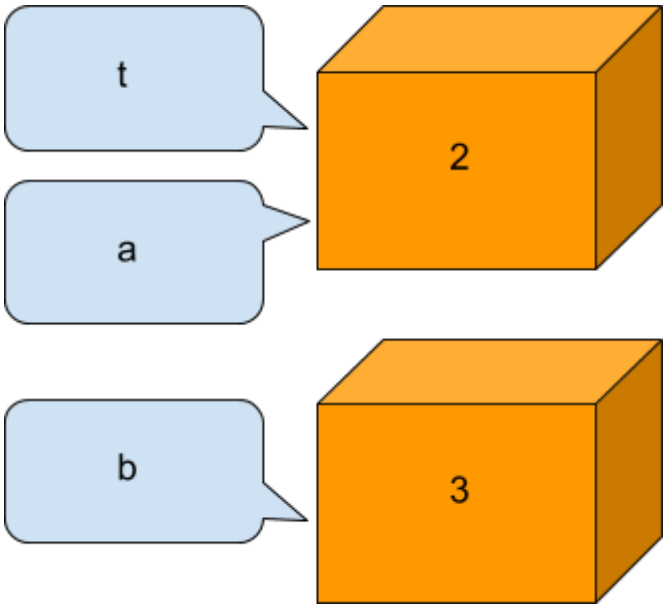


Autrement dit, au lieu d'échanger les contenus des deux boîtes, autant simplement échanger les étiquettes. Pour cela il y a deux moyens :

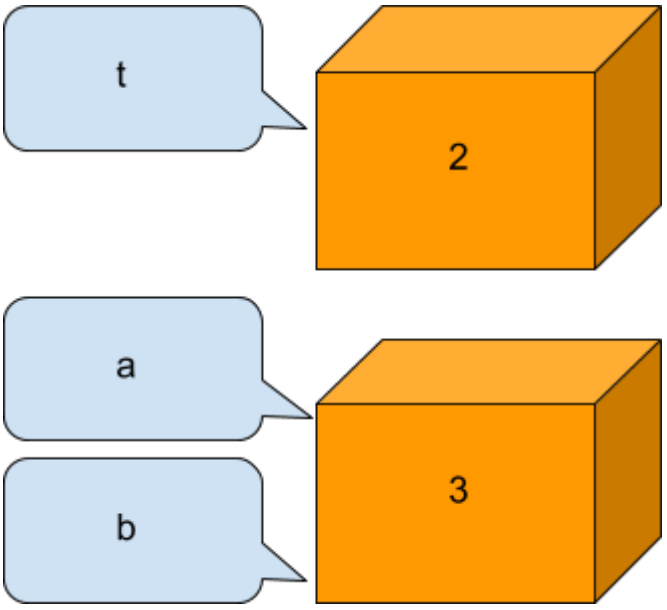
Avec une variable auxiliaire

Si on décolle l'étiquette 'a' pour la rajouter sur la boîte 3, on ne peut plus ensuite coller l'étiquette 'b' sur la boîte 2 puisque celle-ci n'a plus d'étiquette: Comment peut-on savoir où coller l'étiquette 'b'? Il est nécessaire, avant même de décoller l'étiquette 'a', de mémoriser son ancienne boîte avec une nouvelle variable. Comme celle-ci est temporaire on va la noter t :

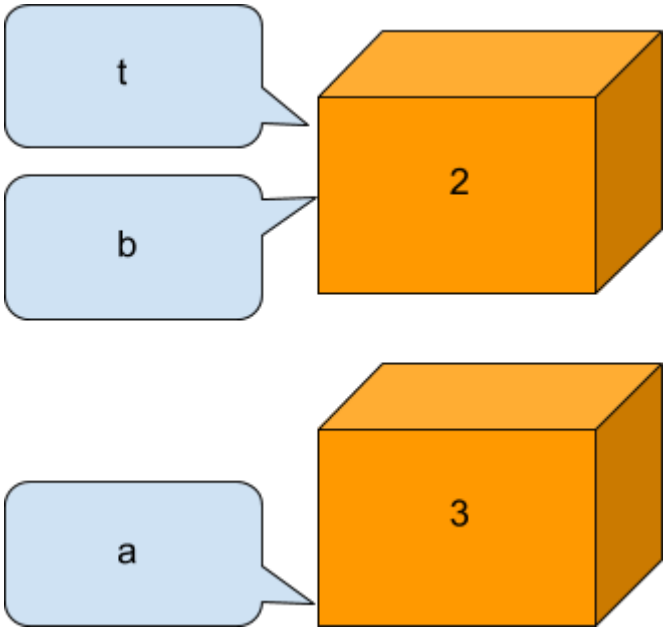
1. On colle l'étiquette 't' sur la boîte 2



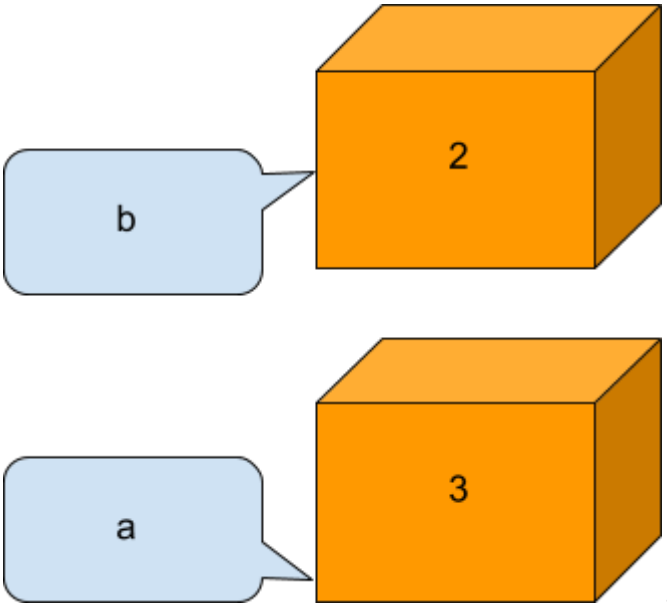
2. On peut maintenant décoller l'étiquette 'a' de la boîte 2 et la coller sur sa nouvelle destination



3. On peut maintenant décoller l'étiquette 'b' de la boîte 3 et la coller sur la boîte 2



4. Enfin il suffit d'enlever l'étiquette 't', devenue inutile, pour avoir le résultat souhaité



Le script Python résumant cela est:

```
>>> a=2 # on a collé l'étiquette a sur l'entier 2
>>> b=3 # situation de départ
>>> t=a # on colle l'étiquette t sur la même boîte que a
>>> a=b # transfert de l'étiquette a sur l'entier 3
>>> b=a # transfert de l'étiquette b sur l'entier 2
>>> print(a)
3
>>> print(b)
2
>>> t=None # si on tient à décoller l'étiquette t (inutile)
```

Échange "in situ"

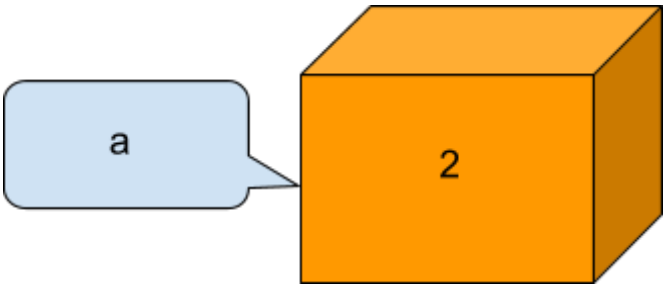
Les boîtes a et b (2 et 3) constituent un couple (a,b) valant (2,3). En Python ce genre d'objet s'appelle un *tuple* (il peut y avoir plus que 2 dimensions). Python peut donc sur commande, fabriquer une nouvelle boîte de type *tuple* laquelle contiendra les deux boîtes précédentes, simplement en écrivant (b,a) ou même sans les parenthèses b,a. On remarque au passage que cette définition du tuple permet d'inverser l'ordre des boîtes. Ensuite, au lieu de coller une étiquette sur le tuple, on colle les étiquettes a et b sur chacune des boîtes qu'il contient. Ce qui permet, au final, d'échanger les contenus de a et b en une seule ligne de Python:

```
>>> a=2 # on a collé l'étiquette a sur l'entier 2
>>> b=3 # situation de départ
>>> a,b= b,a # création, affectation puis dissolution du tuple
>>> print(a)
3
>>> print(b)
2
```

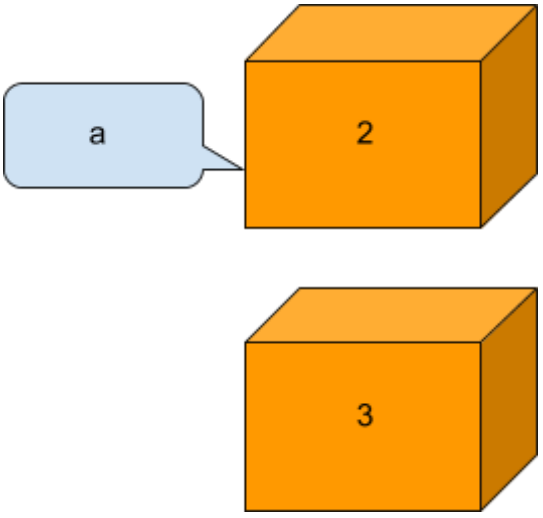
Voyons maintenant comment se déroule une incrémentation, avec ce script:

```
>>> a = 2 # 2 est la valeur initiale de la variable a
>>> a = a + 1 # l'incrémentation
>>> print(a)
3
```

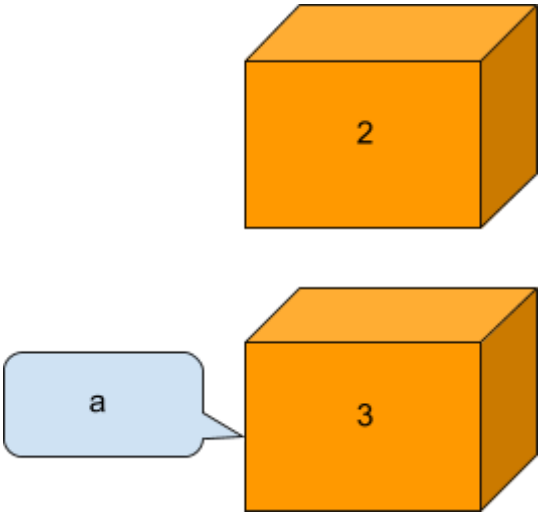
La situation de départ est celle-ci:



L'expression "a+1" désigne, en Python comme dans d'autres langages, "la somme (du contenu) de la variable a et de 1": Comme l'étiquette a est collé sur une boîte de valeur 2, le calcul effectué est 2+1=3. Python va alors créer une boîte contenant 3:



Comme on l'a vu plus haut, le "a=" qui est situé avant "a+1" signifie "coller l'étiquette a sur la boîte contenant. Ce qui aboutit à ce transfert d'étiquette:



Remarque: Ceci n'a été possible que parce que l'interpréteur Python est plus rapide que le redoutable Garbage collector qui, sinon, aurait détruit la boîte 3 alors orpheline. Mais maintenant que la situation s'est stabilisée, c'est la boîte 2 qui est orpheline et qui se fait nettoyer par le Garbage Collector. La situation finale, une fois stabilisée, est donc à une augmentation de la valeur de la variable a (mais aussi, et ça on ne le voit pas, à son déménagement, qu'on peut vérifier en affichant son "id").

B. Principaux types & opérations de bases

Exercice 1 : Tests d'égalité et d'inégalité

L'opérateur "double égal" est booléen: L'expression x==y prend la valeur True si les étiquettes x et y sont collées sur la même boîte, False sinon. L'opérateur "!=" a l'effet contraire. Et s'il existe une relation d'ordre sur le type commun à x et y, on peut les comparer avec "<", "<=", ">" et ">="

```
>>> x = "bon" # x est de type str (c'est un mot)
>>> y = "bo" + "n" # en collant la lettre n derrière le mot bo, on a le mot bon
>>> print(x==y)
True
>>> print(x!=y) # on interprète le "!" comme une négation
False
>>> print(x<"boo") # x est avant boo dans le dictionnaire
True
>>> print("baba"<x<"boo") # Python 3 permet les encadrements
True
```

Exercice 2 : Opérations booléennes

La négation d'une proposition (un booléen) s'écrit en précédant celle-ci du mot *not*. La conjonction entre deux propositions s'écrit en écrivant, entre les deux propositions, au choix, *and* ou *&*. La disjonction inclusive entre deux propositions s'écrit en mettant entre les deux propositions, soit le mot *or*, soit le symbole "|" [AltGr + 6]. Ainsi, Python permet l'implication, non entre deux propositions, mais entre une proposition et une action. Par exemple, l'implication entre une proposition p et une proposition q est définie dans la logique classique par **not p or q**, et il est possible de remplacer la proposition q par une action comme `print("OK")` qui n'est pas une proposition:

```
>>> from random import * # on simule un pile ou face avec randrange(2) qui donne aléatoirement 0 (pile) ou 1 (face)
>>> randrange(2)==0 or print("OK")
OK
>>> if randrange(2)==0: print("OK"); # variante du précédent
OK
```

Les quantificateurs sont

- la conjonction sur liste de booléens (**`all`** pour "quel que soit", quantificateur universel)
- la disjonction sur liste de booléens (**`any`** pour "il existe", quantificateur existentiel)

Par exemple, les affichages ci-dessous signifient respectivement qu'il existe un entier pair parmi les entiers de 0 à 4, et que ces entiers ne sont pas tous pairs:

```
>>> L=[x%2==0 for x in range(5)]
>>> print(L)
[True, False, True, False, True]
>>> print(any(L))
True
>>> print(all(L))
False
```

Exercice 3 : Tests et boucles

Le script ci-dessous est difficile à tester parce que dans la plupart des cas il n'affiche rien:

```
>>> from random import * # on simule un dé avec randrange(1,7)
>>> if randrange(1,7)==6: # le dé est tombé sur 6
...     print("gagné")
```

Alors on propose d'afficher quand même un message dans les autres cas:

```
>>> from random import * # on simule un dé avec randrange(1,7)
>>> if randrange(1,7)==6: # le dé est tombé sur 6
...     print("gagné")
... else:
...     print("perdu")
```

Pour éviter les *if* dans les *else* à répétition, Python permet des "else if" résumés en *elif*:

```
>>> from random import *
>>> dé = randrange(1,7) # on lance le dé une seule fois
>>> if dé==6:
...     print("on a gagné")
... elif dé==1:
...     print("on est tombé sur un as")
... else:
...     print("on est tombé sur un os")
perdu
```

Malgré l'élégance de cette solution, on peut lui préférer l'usage d'un dictionnaire (voir plus bas):

```
>>> from random import * # on simule un dé avec randrange(1,7)
>>> annonce = {1: "as", 2: "os", 3: "os", 4: "os", 5: "os", 6: "gagné"}
>>> print(annonce[randrange(1,7)])
os
```

Les booléens sont aussi utiles dans les boucles, qui sont parcourues tant que le booléen est vrai. Par exemple si on veut absolument un 6, on lance le dé sans arrêt:

```
>>> from random import * # on simule un dé avec randrange(1,7)
>>> lancers = 0 # au début on n'a pas encore lancé le dé une seule fois
>>> while randrange(1,7) != 6: # tant qu'on n'a pas 6 on continue
...     lancers = lancers + 1 # on relance le dé
>>> print(lancers)
13
```

Exercice 4 : Types numériques - int, float, complex

Si on entre un nombre sans le point décimal, ce nombre est considéré comme entier (type *int*) par Python. Les opérations sont codées +, –, *, / pour ce qui est des opérations classiques, et ** pour l'exponentiation (élévation à une puissance).

```
>>> a = 2017 # si on avait entré 2017.0 a serait de type float
>>> print(a+a)
4034
>>> print(a-a)
0
>>> print(a*a)
4068289
>>> print(a**2)
4068289
>>> print(a**a) # il n'y a pas de limite au nombre de chiffres d'un entier
```

Il y a deux sortes de divisions pour les entiers de Python: Le quotient exact est donné par "/", le quotient euclidien par "//":

```
>>> print(22/7) # le quotient n'est pas un entier mais un flottant
3.142857142857143
>>> print(22//7) # le quotient euclidien est entier
3
>>> print(divmod(22,7)) # toute la division euclidienne donne un tuple (quotient,reste)
(3, 1)
>>> print(22%7) # pour avoir seulement le reste
3
```

Pour convertir un entier relatif en entier naturel, utiliser la fonction *abs*. La fonction *int* convertit son argument en entier. Avec un flottant elle effectue une troncature.
Un *flottant* est le quotient d'un entier par une puissance de 2. Les flottants de Python modélisent les nombres réels (mais approximativement).

⚠ Attention !

Essayer ça :

```
>>> print(0.1*3) # normalement on devrait avoir 0.3
0.30000000000000004
```

Ce phénomène illustre que 0,1 n'est pas un flottant puisque son dénominateur n'est pas une puissance de 2. En entrant 0.1 on a donc une approximation de 0,1 par un flottant et une fois multipliée par 3, cette approximation est plus proche de 0,30000000000000004 que de 0,3 lui-même.

Les opérations sur les flottants s'écrivent de la même façon que pour les entiers: +, -, *, / et **. Mais l'exposant peut être un flottant aussi, comme le montrent ces manières de calculer une racine carrée:

```
>>> print(2**0.5) # élever à la puissance 0,5 c'est extraire la racine
1.4142135623730951
>>> from math import * # on importe tout ("*") du module math
>>> print(sqrt(2)) # abréviation de "square root"
1.4142135623730951
```

Beaucoup de fonctions intéressantes sur les flottants se trouvent dans le module *math*. Par exemple cos, sin, tan (en radians) et leurs inverses acos, asin et atan (en radians). Noter la fonction *atan2* qui accepte deux arguments: Le côté opposé et le côté adjacent, et qui renvoie l'angle en radians. Les fonctions *degrees* et *radians* effectuent la conversion entre degrés et radians. Mais le module math contient aussi des constantes comme pi.
Il y a deux flottants très particuliers, le premier d'entre eux est l'infini:

```
>>> I = float("inf") # I est infini
>>> print(I+2) # au-delà de l'infini c'est encore l'infini
inf
>>> print(I-2) # il ne suffit pas d'enlever un morceau à l'infini
inf
>>> print(I*2) # doubler l'infini ne le change pas
inf
>>> print(I*I) # cas connu
inf
>>> print(1/I) # cas connu
0.0
>>> print(I-I) # idem avec I/I ou I*0: formes indéterminées
nan
```

Les formes indéterminées révèlent l'autre flottant spécial: *nan* est une abréviation pour "not a number", soit, flottant non numérique. Pour créer un nombre à la Magritte (un nombre qui n'est pas numérique) on fait `float("nan")`.
Le nombre de carré – 1 ne se note pas i en Python, mais j. Ou plutôt, 1j.

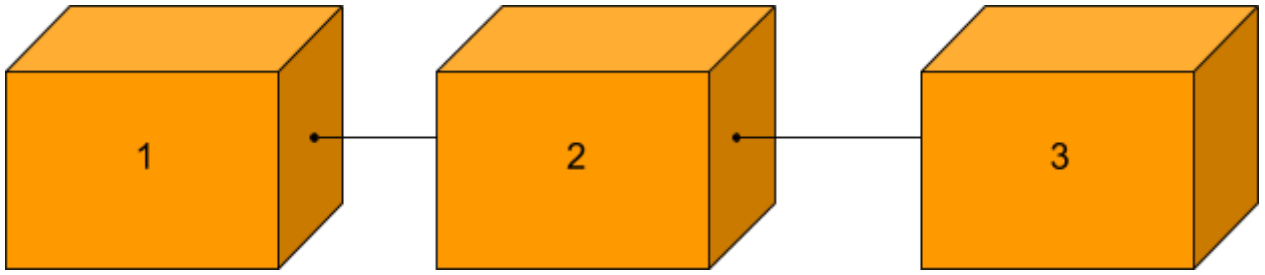
```
>>> a=2+1j # a=2+i
>>> b=4+3j # b=4+3i
>>> print(a+b) # a+b=6+4i
(6+4j)
>>> print(a-b) # a-b=-2-2i
(-2-2j)
>>> print(a*b) # a×b=5+10i
(5+10j)
>>> print(a/b) # a/b=0,44-0,08i
(0.44-0.08j)
>>> print(a**2) # a²=3+4i
(3+4j)
```

Les propriétés *real* et *imag* calculent les parties réelle et imaginaire d'un complexe (ce sont des entiers ou des flottants); la méthode *conjugate()* calcule le conjugué. Et la fonction *abs* calcule le module. Voici leurs syntaxes:

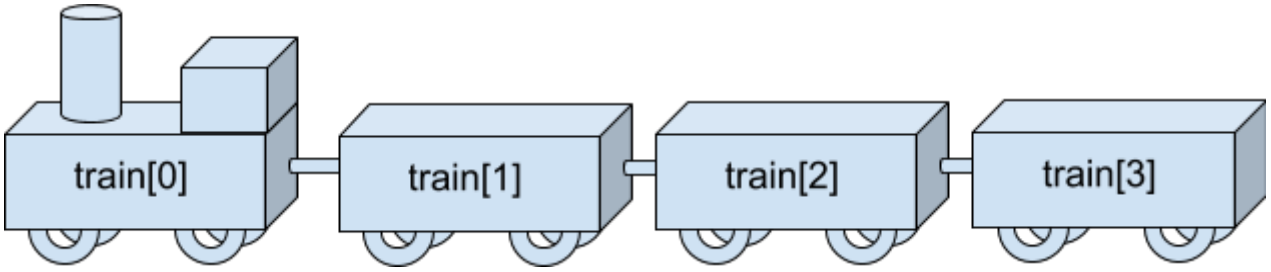
```
>>> print(b.real) # Re(b)=4
4
>>> print(b.imag) # Im(b)=3
3
>>> print(b.conjugate())
(4-3j)
>>> print(abs(b))
5.0
```

Exercice 5 : Listes de Python

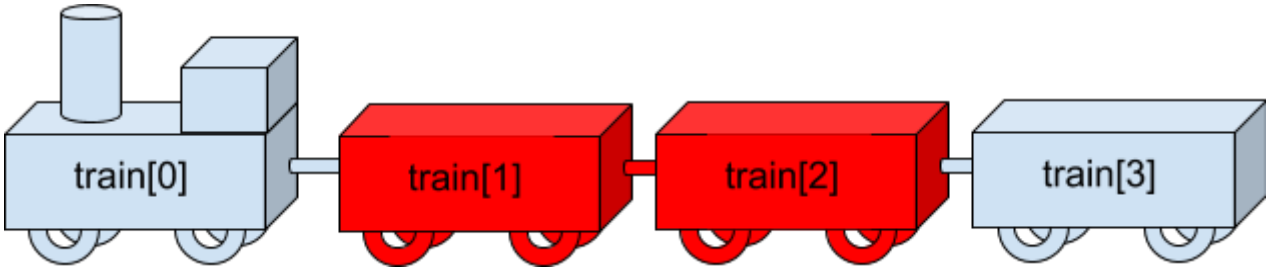
Une liste est une boîte contenant des boîtes. C'est donc un objet plus compliqué que les nombres et les booléens. En plus, les boîtes qui sont à l'intérieur de la grosse boîte, sont reliées entre elles ("chaînées"). Par exemple la liste [1,2,3]:



Il n'est pas nécessaire de coller une étiquette sur chaque boîte interne, pour avoir accès à celle-ci: Si on a collé l'étiquette L sur la boîte externe, la locomotive (portant le numéro 1) est notée L[0], le wagon contenant 2 est noté L[1] et le wagon contenant 3 est noté L[2].

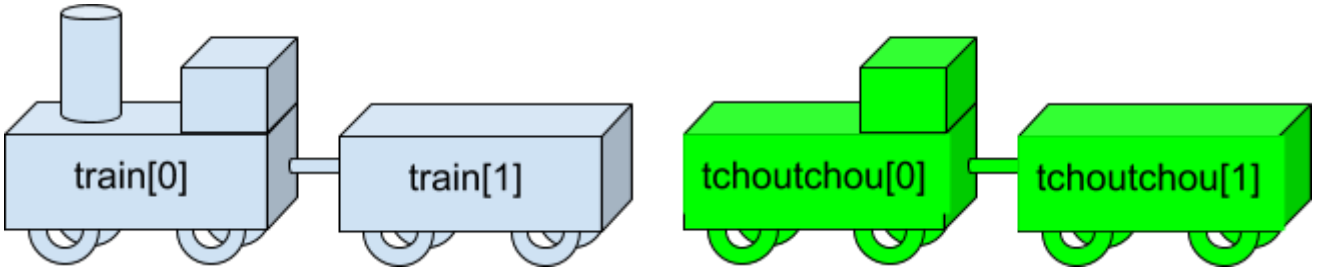


Ci-dessus on a représenté une liste Python dont les boîtes (non étiquetées mais habitées) sont numérotées depuis train[0] (locomotive) jusqu'à train[3] (wagon de queue). On peut donc dire que poser une étiquette "train" sur la liste, a pour effet d'étiqueter automatiquement les wagons (locomotive comprise).

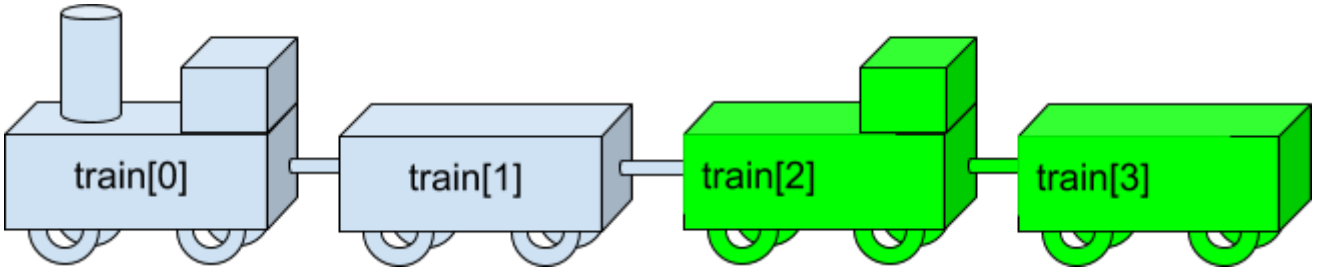


Les wagons en rouge ci-dessus constituent à eux deux un train ("sous-liste) que l'on peut noter `[train[1],train[2]]` ou plus simplement `train[1:3]` (3 est exclu).

L'opération consistant à obtenir un train long en attachant un train (vert) à un autre train (bleu) s'appelle la *concaténation*. À partir de



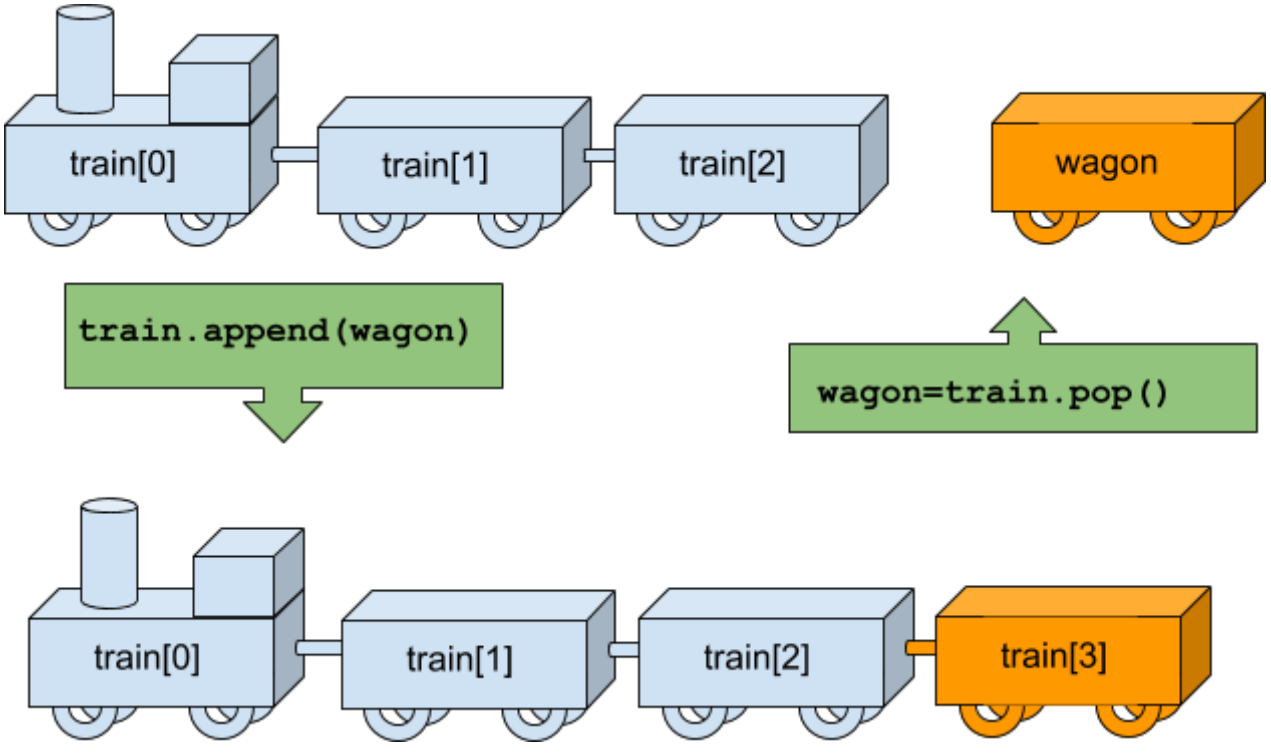
on obtient



Il y a deux manières d'attacher un train à un autre:

- train = train + tchoutchou
- train.extend(tchoutchou)

Mais pour attacher un seul wagon à la fin du train on fait train.append(wagon). Pour détacher le wagon de queue on fait train.pop():



Pour accrocher un wagon ailleurs qu'à la fin du train, on utilise insert ; par exemple pour insérer une nouvelle locomotive on fait *L.insert(0,"loco")*.

Syntaxe

# commentaires	# console input	# console output
# créer une liste # afficher la liste # ajouter 1 à la liste # afficher la liste # ajouter "ok" # afficher la liste # affecter une liste # Lire l'élément 0 (n°1) # Lire le 3ème élément # mettre à jour la liste # afficher la liste # supprimer un élément # afficher la liste # supprimer par l'indice # afficher la liste # affecter une liste # inverser la liste # afficher la liste # longueur de la liste # affecter une liste # compter le nombre de a # compter le nombre de c # position de b (indice)	new_list = [function(item) for item in list if condition(item)]	 [] [1] [1, 'ok'] 'a' 'm' ['a', 'd', 'z'] ['d', 'z'] ['d'] ['c', 'b', 'a'] 3 3 2 3

Compréhensions de liste

Syntaxe

```
new_list = [function(item) for item in list if condition(item)]
```

Prenons l'exemple d'une liste.

```
>>> a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
```

Filtrons les éléments de cette liste et ne gardons que ceux dont la valeur est supérieure à 5:

```
>>> b=[]
>>> for x in a:
...     if x > 5:
...             b.append(x)
...
>>> b
[7, 9, 6, 6, 6, 8]
```

Il est possible de faire exactement ce que fait ce bloc de code en une seule ligne:

```
>>> [x for x in a if x > 5]
[7, 9, 6, 6, 6, 8]
```

Exécuter une fonction sur chaque item d'une liste

Prenons l'exemple d'une conversion de string en integer de plusieurs items:

```
>>> items = ["5", "10", "15"]
>>> items = [int(x) for x in items] # avec items=list(map(int,items)) ça marche aussi
>>> print items
[5, 10, 15]
```

Exercice 6 : Tuples

Quelle est la différence entre

- la liste [5,8]
- le couple (5,8)
- la paire {5,8}?

La paire {5,8} est un ensemble (voir les dictionnaires plus bas), c'est-à-dire qu'elle est égale à la paire {8,5}. Cette indifférence aux permutations la distingue à la fois de la liste et du couple. De plus l'ensemble {5,5} n'est pas une paire (c'est le singleton {5}) alors que la liste [5,5] et le couple (5,5) existent. Reste à voir en quoi [5,8] et (5,8) diffèrent.

```
>>> T = (5,8) # T = 5,8 marche aussi, si si!
>>> print(T) # même avec T = 5,8 on a (5,8)
(5, 8)
>>> print(T[0]) # pour lire le premier élément c'est comme avec une liste
5
>>> T[0]=6 # avec une liste on peut faire ça mais pas avec un tuple
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    T[0]=6
TypeError: 'tuple' object does not support item assignment
```

De plus, append permet de changer la longueur d'une liste alors que la longueur d'un tuple est constante: On dit que la liste est *mutable* (elle peut subir des mutations) alors que le tuple ne l'est pas.

Les couples, et les tuples en général, ont une grande importance en Python, parce qu'ils servent à décrire les antécédents des fonctions. Par exemple le pgcd de 18 et 24 peut être considéré comme l'image du couple (18,24) par la fonction pgcd:

```
>>> def pgcd(a,b):
...     while b > 0:
......         a,b = b,a%b # en fait c'est une histoire de couples (a,b)=(b,a%b)
...     return a
>>> print(pgcd(18,24)) # on dirait que c'est le pgcd de deux nombres
6
>>> print(pgcd (18,24) ) # mais elle s'applique au tuple (18,24)
6
```

Les couples sont également fondamentaux pour les dictionnaires de Python, dont ils sont les "items". Et bien entendu les points et vecteurs du plan sont des couples de coordonnées.

Exercice 7 : Type Séquence de texte - str

On crée un objet de type str en l'écrivant entre guillemets, simples ou doubles. La fonction len donne le nombre de lettres d'un mot, et les lettres sont numérotées à partir de 0.

```
>>> mot = "bon" # "bon" ou 'bon' donnent 'bon'
>>> print(mot)
'bon'
>>> print(len(mot))
3
>>> print(mot[0])
'b'
>>> print(mot[2])
'n'
>>> print(mot[-1])
'n'
```

La lettre numéro -1 est la dernière lettre: La position -1 est en fait n-1 où n est la dernière position.

La concaténation est notée additivement, et du coup on peut multiplier un mot par un entier:

```
>>> print("bon"+"jour")
'bonjour'
>>> print("bon"*2)
'bonbon'
>>> print(2*"bon")
'bonbon'
```

On peut itérer sur les lettres d'un mot:

```
>>> print("o" in "bon")
True
>>> for lettre in "bon":
...     print(lettre)
'b'
'o'
'n'
```

Un mot d'une seule lettre est un *caractère*. Il possède un code Ascii entier, donné par la fonction ord:

```
>>> print(ord("o"))
111
>>> print(chr(111))
'o'
```

Les bijections *ord* et *chr* permettent d'utiliser l'arithmétique pour faire de la cryptographie.
En plaçant la lettre minuscule "u" avant le guillemet ouvrant, on ne change rien à l'écriture d'une chaîne de caractères. Mais on peut maintenant y insérer des caractères unicode, sous la forme d'un "\u" suivi de 4 chiffres hexadécimaux:

```
>>> c = u"as de \u2764" # 2764 est l'unicode d'un cœur rempli
>>> print(c)
'as de ❤'
>>> print(len(c))
7
>>> print(type(c))
<class 'str'>
>>> print(c[6])
'❤'
```

Le code 221e donne "∞" et le code 2208 donne "∈" qui peuvent être utiles pour des activités mathématiques[1].On peut remplacer certaines lettres (ou morceaux de texte) par d'autres avec *replace*

```
>>> c ="coucourroucoucou"
>>> print(c)
'coucourroucoucou'
>>> c = c.replace("ou","a") # on remplace chaque ou par un a
>>> print(c)
'cacarracaca'
>>> c = c.replace("c","t") # on remplace chaque a par un t
>>> print(c)
'tatarratata'
```

Exercice 8 : Type dictionnaire - dict

Python permet aussi de faire de la théorie des ensembles (utile en probabilités, ou pour la résolution des équations) avec le type *set*[2]. Pour construire un ensemble, on fournit à la fonction `set` un itérable, comme par exemple une liste, un range ou une chaîne de caractères. Par exemple pour avoir l'ensemble des lettres du mot "coucourroucoucou", on peut faire :

```
>>> E = set("coucourroucoucou")
>>> print(E)
{'c', 'o', 'u', 'r'}
>>> print(len(E))
4
>>> print('o' in E)
True
```

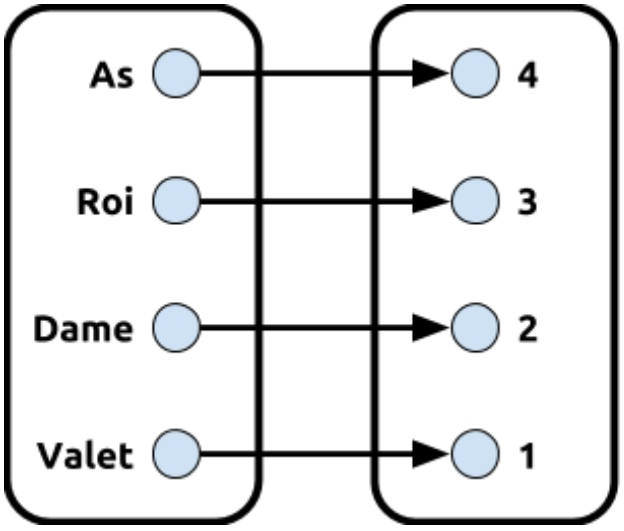
[1] Voir ici pour en trouver d'autres : https://en.wikipedia.org/wiki/Mathematical_operators_and_symbols_in_Unicode
[2] "set" est la traduction anglaise de "ensemble"

Comme on le voit, le cardinal d'un ensemble s'écrit *len*, et l'appartenance d'un élément à un ensemble s'écrit *in*. L'inclusion est notée *issubset*, l'intersection est notée *intersection*, la réunion est notée *union* et le complément est noté *difference*. Voici une découverte expérimentale de la notion de pgcd que permet Python:

```
>>> def diviseurs(n):
...     return {d for d in range(1,n+1) if n%d==0}
>>> A = diviseurs(18)
>>> B = diviseurs(24)
>>> print(A)
{1, 2, 3, 6, 9, 18}
>>> print(B)
{1, 2, 3, 4, 6, 8, 12, 24}
>>> print(A.issubset(B))  # A n'est pas inclus dans B
False
>>> print(A.union(B))  # calcul de la réunion des ensembles
{1, 2, 3, 4, 6, 8, 9, 12, 18, 24}
>>> print(A.intersection(B))  # ensemble des diviseurs communs à 18 et 24
{1, 2, 3, 6}
```

Parmi les 4 diviseurs communs à 18 et 24, il y en a un qui est plus grand que les autres, et qui est même divisible par les autres: Le pgcd…

Un dictionnaire de Python est une application entre deux ensembles. Les éléments de l'ensemble de départ s'appellent les *clés* (en anglais *keys*) et les éléments de l'ensemble d'arrivée sont les *valeurs* (en anglais, *values*). Mais comme l'ordre importe (pour savoir quel élément est associé à quel autre), on préfère représenter les clés et les valeurs par des listes que par des ensembles. Cependant, chaque clé est censée n'exister qu'en un seul exemplaire. Par exemple on souhaite, pour un jeu de cartes, compter les points de la manière suivante:



(les figures 7, 8, 9 et 10 comptent pour 0 point). L'ensemble de départ est {as, roi, dame, valet} et l'ensemble d'arrivée est {4, 3, 2, 1}. Chaque flèche du diagramme sagittal s'écrit par un double-point.

Noter qu'ici les clés sont des chaînes de caractères:

```
>>> D = {"as": 4, "roi": 3, "dame": 2, "valet": 1}
>>> print(D)
{'dame': 2, 'roi': 3, 'valet': 1, 'as': 4}
>>> print(D.keys())
dict_keys(['dame', 'roi', 'valet', 'as'])
>>> print(D.values())
dict_values([2, 3, 1, 4])
>>> print("as" in D) # être dans le dictionnaire, cela signifie être une clé du dictionnaire
True
>>> for k in D:
...     print(D[k])
2
3
1
4
```

On boucle sur les clés k, et pour avoir les valeurs on écrit D[k] (par exemple pour connaître la valeur d'un as on écrit `D['as']`). Les dictionnaires sont une bonne façon de représenter les tableaux d'effectifs, et la fonction `globals()` vue dans la partie sur les affectations renvoie l'espace des variables sous forme d'un dictionnaire (les clés sont les étiquettes, les valeurs sont les variables elle-mêmes). Les items d'un dictionnaire sont des tuples (clé,valeur). On peut les récupérer avec `D.items()`.

Puisqu'un dictionnaire est une application d'un ensemble de clés dans un ensemble de valeurs, on peut créer un tableau de valeurs (sous forme de dictionnaire) à partir d'une fonction. Par exemple la fonction *bin*, qui, à un entier naturel, associe sa représentation binaire:

```
>>> tabval=dict((n,bin(n)) for n in range(8)) # à n on associe bin(n)
>>> print(tabval[3])
'0b11'
>>> print({x for x in tabval.keys() if tabval[x]=='0b101'}) # recherche d'antécédents
{5}
```

Exercice 9 : Itérateurs

L'objet Python noté *range(1, 9)* est un itérateur, dont le comportement est similaire à celui d'un croupier (au poker au baccarat ou au blackjack) ou d'un sabot:

```
>>> sabot=range(1,9)
>>> print(sabot) # sabot pas trop affichable avec Python 3
range(1, 9)
>>> print(type(sabot)) # le type range est très class
<class 'range'>
>>> print(list(sabot)) # converti en liste le range est affichable
[1, 2, 3, 4, 5, 6, 7, 8]
>>> print(10 in sabot) # la carte 10 n'est pas dans le sabot
False
```

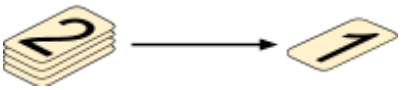
Lorsque l'objet *sabot* est initialisé, il contient les cartes de 1 à 8. Voici l'objet *range(1, 6)*, sous forme de sabot (à l'initialisation, avec ses 5 cartes), puis converti en liste:

On a vu que pour savoir si une carte est dans le sabot, Python n'a pas besoin de faire cette transformation en liste[3]. Ce qui permet d'itérer en demandant des cartes, l'une après l'autre, au croupier. La syntaxe est *for carte in sabot*:

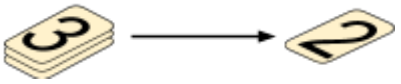
```
>>> sabot = range(1,9)
>>> for carte in sabot: # noter le double-point et l'indentation
...   print(carte)
1
2
3
4
5
6
7
8
```

[3] L'utilité essentielle de la conversion de range en list est de permettre d'afficher celle-ci, ce qui se fait d'ailleurs par une autre conversion, en chaîne de caractères. Mais au lieu de `print(str(list(sabot)))` on peut se contenter de `print(list(sabot))`, la conversion en str par print étant automatique.

Avec range(1,6), avant la boucle, le sabot contient initialement 5 cartes, et la première porte le numéro 1. Lors du premier passage dans la boucle, on affecte à la variable *carte* la valeur 1.



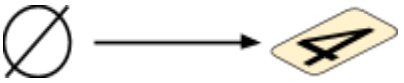
Après le premier passage dans la boucle, le sabot ne contient plus que 4 cartes, celle du dessus étant la numéro 2. Aussi, le second passage dans la boucle a-t-il pour effet d'affecter la variable carte avec la valeur 2, et de réduire la taille du sabot.



Et lors du troisième passage dans la boucle, c'est la carte numéro 3 qui est fournie.

Lors du quatrième passage, c'est la carte 4 (puisqu'elle est au-dessus) qui est affectée à la variable.

Lors du cinquième passage la carte 5 est fournie ce qui a pour effet de vider le sabot: Il n'y aura pas de sixième passage dans la boucle[4].



Exercice 10 : Fonctions

Une fonction se crée en la définissant ("def"). La définition de la fonction s'écrit dans un bloc indenté, et précédé d'un double-point:

```
>>> def cube(x):
...     return(x**3)
>>> print(cube(5))
125
>>> print(type(cube))
<class 'function'>
```

Une bonne habitude à prendre, dès le début, est d'écrire une "python-doc" dans le corps de la fonction:

```
>>> def cube(x):
...     """élève un nombre à la puissance 3"""
...     return(x**3)
>>> print(cube(5))
125
>>> help(cube) # ça peut servir. Vraiment!
Help on function cube in module __main__:
cube(x)
    élève un nombre à la puissance 3
```

[4] Lorsque le range est vide, Python lève une exception de fin de boucle qui est gérée comme une erreur mais de façon plus transparente.

On peut aussi restreindre la définition d'une fonction avec assert:

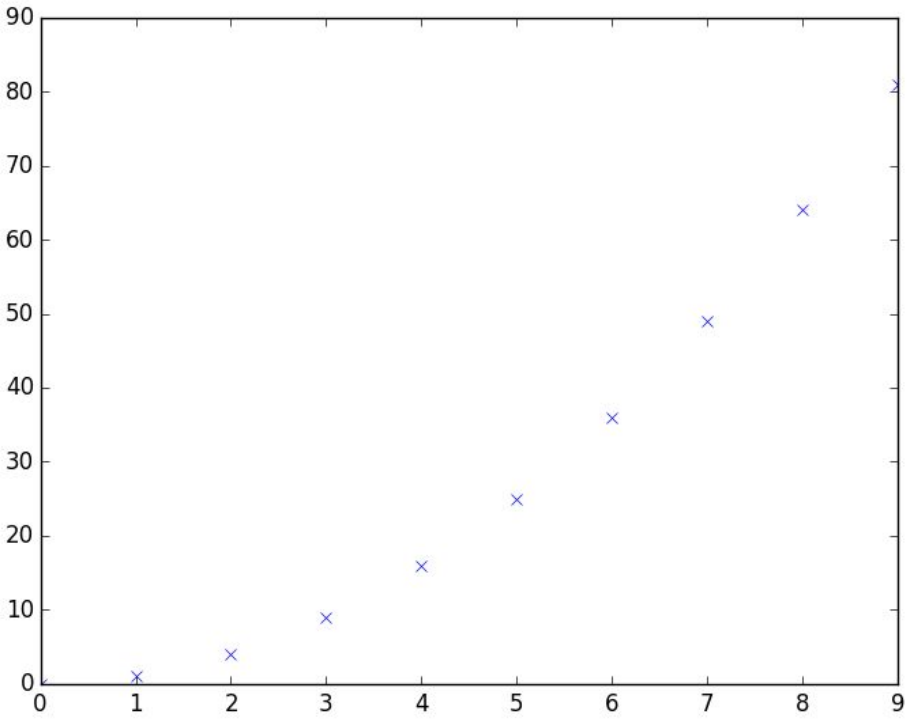
```
>>> def racine(x):
...     assert x>=0
...     return(x**0.5)
>>> print(racine(-1))
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    racine(-1)
  File "<pyshell#0>", line 2, in racine
    assert x>=0
AssertionError
```

Si on n'avait pas mis *assert*, la réponse affichée aurait été `6.123233995736766e-17+1j` dont la partie réelle est nulle (enfin presque, à cause d'erreurs d'approximation) et la partie imaginaire 1: Il s'agit d'un nombre complexe.

Le module matplotlib permet de représenter graphiquement des nuages de points avec pyplot; cette fonction accepte en entrée deux listes de nombres, et dessine le nuage de points dont les abscisses sont puisées dans la première liste, et les ordonnées dans la seconde liste. Par exemple, pour dessiner des points sur la parabole $y=x^2$, on crée la liste X des entiers et la liste Y des carrés d'entiers et on affiche le nuage de points:

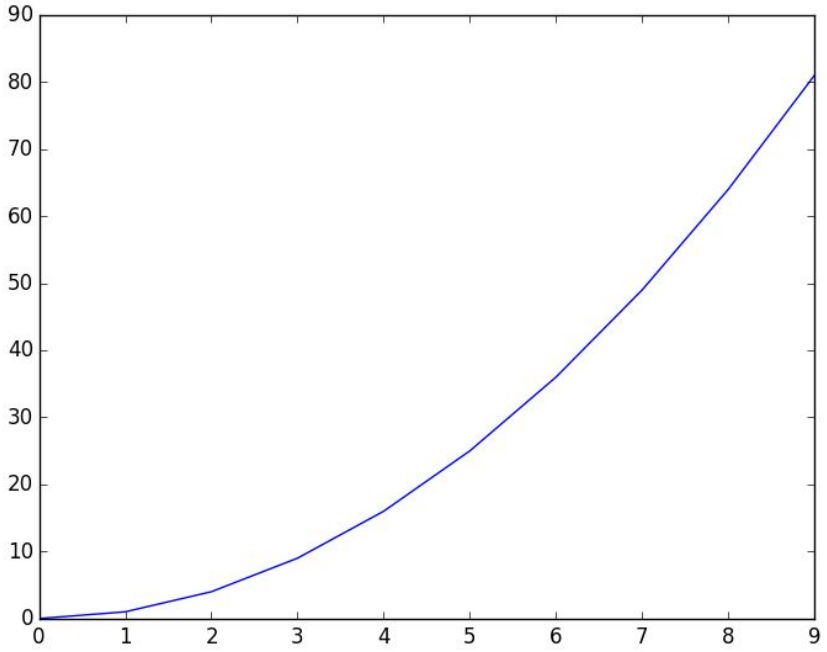
```
>>> from matplotlib.pyplot import * # on importe tout, sans faire dans le
détail, pour se simplifier la vie
>>> X = list(range(10)) # X=[0,1,2,3,4,5,6,7,8,9]
>>> Y = [x**2 for x in range(10)] # Y=[0,1,4,9,16,25,36,49,64,81]
>>> plot(X,Y,"x") # "x" pour des croix, "o" pour des ronds, "*" pour des étoiles
```

On a bien un nuage de points:



Très utiles en statistiques à deux variables, les nuages de points ne sont pas, tels quels, adaptés à des représentations graphiques de fonctions. Mais si on remplace les "x" par des "-" dans plot, des

segments sont tracés entre les points successifs, et on a un polygone qui ressemble à une représentation graphique de fonction:



Heureusement, le "-" est optionnel et le script suivant produit le même effet, à savoir le dessin d'un polygone qui ressemble à une parabole.

```
>>> from matplotlib.pyplot import *
>>> X = list(range(10)) # X=[0,1,2,3,4,5,6,7,8,9]
>>> Y = [x**2 for x in range(10)] # Y=[0,1,4,9,16,25,36,49,64,81]
>>> plot(X,Y)
>>> show()
```

Exercice 11 : Fonctions et procédures

Il est indispensable pour réaliser une tâche compliquée de la décomposer en sous-tâches[5].
Avantages :

- la lecture est infiniment plus facile, le programme principal est minimal, il s'agit d'un plan donnant les intentions
- on évite les répétitions en factorisant[6] le code
- on évite les erreurs, chaque sous-tâche est plus facile à contrôler

Inconvénients :

- Il faut distinguer sortie et retour[7]

Exemple 1 : sans fonction, on peut faire ainsi :

```
nbDeLigne=30
for i in range(nbDeLigne):
    print(' '*(nbDeLigne-i)+"*"*(2*i-1) )
```

Remarques :

[5] Voir Descartes, "discours de la méthode" pour en savoir plus sur ce principe.
[6] "factoriser", ou "refactorer" en québécois, consiste essentiellement à répartir les instructions entre diverses fonctions ce qui évite de retrouver plusieurs fois la même portion de code. Cela simplifie et améliore le code en même temps. en anglais "refactor" signifie "réusiner" ce qui colporte assez bien l'idée.
[7] cette distinction est d'ailleurs l'objet d'une préconisation de l'inspection générale de mathématiques en 2017.

- Mais que fait ce code?
- Est-ce lisible?
- Est-ce vraiment si élégant si on n'arrive pas à le lire?

Ou bien comme ceci :

```
def espaces(n):
    space=""
    for i in range(n):
        space = space + " "
    return space
def etoiles(n):
    etoiles = ""
    for i in range(n):
        etoiles = etoiles + "*"
    return etoiles
def pyramide(nbDeLigne):
    for i in range(nbDeLigne):
        print( espaces( nbDeLigne - i ) + etoiles( 2*i-1 ) )
pyramide(30)
```

Exercice 12 : Modules

Le Python de base (celui qui tourne au démarrage) est incomplet pour certaines applications. On peut le personnaliser en chargeant des modules, ce qui se fait par "import". Voici quelques modules qui peuvent être utiles:

- *fractions* permet d'effectuer du calcul sur les fractions
- *decimal* permet de faire du calcul sur un grand nombre de décimales
- *math* contient les constantes e, pi et les fonctions trigonométriques, logarithme, exponentielle et surtout, la racine carrée sqrt
- *random* contient des fonctions utiles pour simuler le hasard: Variables aléatoires uniformes, exponentielles, normales, mélange aléatoire d'une liste (utile pour simuler le battage des cartes, choix au hasard dans une liste, etc
- *time* permet de faire du chronométrage (estimer la durée d'exécution d'une fonction)
- *turtle* permet de faire du graphisme tortue. Il contient l'objet Vec2D (vecteur du plan) qui permet de faire de la géométrie repérée plane.
- *threading* permet de simuler le calcul parallèle
- *Tkinter* permet de faire du graphisme dans une fenêtre. Il est utilisé dans le module turtle mais aussi le logiciel IDLE

III. Bibliographie

site	Description
https://openclassrooms.com/courses/apprenez-a-programmer-en-python/le-monde-merveilleux-des-variables	openclassroom : formation en autonomie en langage Python et d'autres langages de programmation
http://apprendre-python.com/	tutoriel en ligne
http://www.apmep.fr/Prendre-en-main-Python#1	tuto pour profs de maths
https://download.tuxfamily.org/edupython/EduPython1.0.pdf	Manuel d'EduPython; contient plein d'idées pour le cours de maths même si on n'utilise pas PythonLycée
http://revue.sesamath.net/spip.php?article385	présentation du livre "python pour les mathématiques", de Guillaume Connan
https://fr.wikibooks.org/wiki/Mathématiques_avec_Python_et_Ruby	wikibook dont la première moitié est consacrée à Python
http://irem.univ-reunion.fr/spip.php?rubrique100	activités sur les évènements et probabilités (réunion, intersection, complémentaire)
http://irem.univ-reunion.fr/spip.php?article232	Article de didactique sur l'itération
http://revue.sesamath.net/spip.php?article385	sommation et calcul intégral, par Guillaume Connan
http://irem.univ-reunion.fr/spip.php?article924	Présentation de SofusPy: Exemples ultra-courts (pourcentages, suites géométriques, méthode des rectangles et graphisme tortue)
https://fr.wikibooks.org/wiki/Programmation_objet_et_géométrie/Objets_en_Python_sous_Gimp	Dessin en 2D avec la console Python de Gimp
https://fr.wikibooks.org/wiki/Programmation_objet_et_géométrie/Objets_Python_sous_Blender/Objets_3D_en_Python_sous_Blender	Dessin en 3D avec la console Python de Blender3D
https://fr.wikibooks.org/wiki/Pygame/Version_imprimable	Création d'un jeu Python avec Pygame (utilitaire graphique et multimédia)
http://irem.univ-reunion.fr/spip.php?article607	Comment faire des exerciciels de Python avec Python
https://www.youtube.com/watch?v=Q63Tpbhnt1E	Exo7 : Premiers pas avec Python en vidéo
http://exo7.emath.fr/	Exo7 : des tutoriels en pdf sur scratch et sur python
http://www.zotweb.re/irem/SofusPy974/	SofusPy974 : Passer des blocks à Python

Alter Way

TYPES — écriture littérale

NoneType	None	non mutable
int() long() float() complex()	12 12L 12.0 1+2j	non mutable
bool()	True False	non mutable
str() unicode()	"truc", 'truc' u"trûc", u'trüç'	non mutable
tuple()	(1, 2, "3", 4.0)	non mutable
list()	[1, 2, "3", 4.0]	mutable
set()	{1, 2, "3", 4.0}	mutable
dict()	{'cle': 'valeur', 'a': 1, 'b': 2}	mutable

PyCon FR 2010

OPÉRATEURS

unaires	*not - +*
binaires	*+ - * / // % ** and or in == < > <= >= !=*
ternaire	*a if b else c*

exemples d'EXPRESSIONS :

```
x + 2 < 12 or "abc"
(5 ** 8 == x)
```

STRUCTURES DE CONTRÔLE

condition :

```
if expression:
    qque chose
elif expression:
    autre chose
else:
    3eme chose
```

gestion d'erreur :

```
try:
    qque chose
except:
    autre chose
finally:
    truc final
```

boucles :

```
while expression:
    qque chose

for truc in trucs:
    qque chose
```

gestion de contexte :

```
with expr as truc:
    qque chose
```

python™

FONCTION

```
def ma_fonction():
    print "truc"
```

CLASSE

```
class MaClasse(object):
    un_attribut = 12
    def __init__(self):
        print "constructeur"
    def une_methode(self):
        print "truc"
```

NOTATIONS — Exemple

affectation	truc = "contenu" a, b, c = (1, 2, 3) truc += 2
indiçage (indexing)	truc['cle'] truc[3]
segmentation (slicing)	truc[3:6] truc[3:6:2]
attribut d'objet	truc.attribut
appel	truc() truc(a, b, c=2, d='z')

MOTS CLÉS

print
del
from
import
raise
return
break
continue
assert
pass

PRIMITIVES

abs()	type()
max()	enumerate()
min()	help()
sum()	getattr()
open()	hasattr()
file()	delattr()
map()	exit()
reduce()	id()
filter()	dir()
range()	xrange()

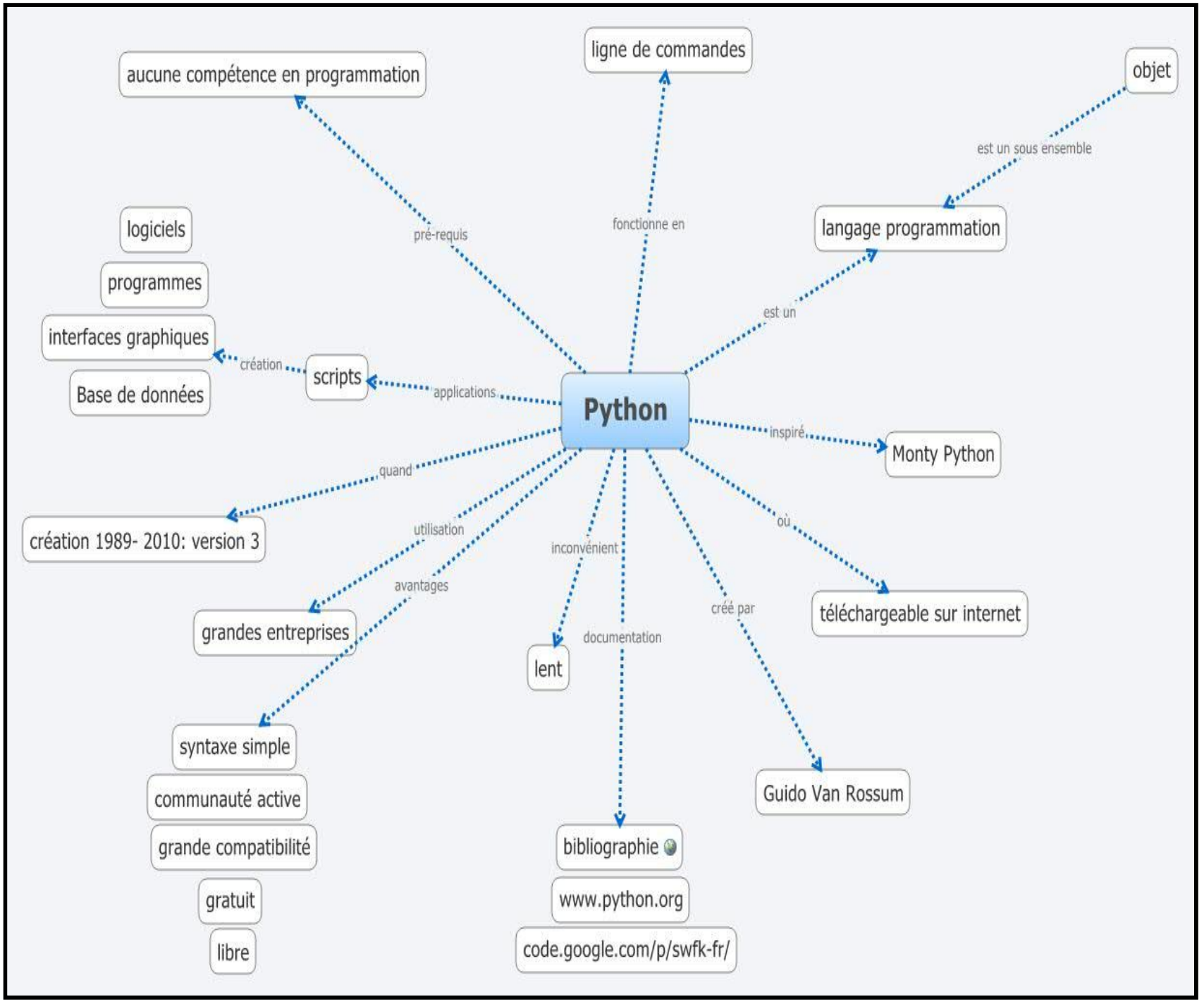
AIDE

dir(truc) help(truc) truc.__doc__

WEB : http://docs.python.org http://afpy.org

IRC : #python-fr et #afpy sur freenode

[Sommaire]


Python
pré-requis
aucune compétence en programmation
fonctionne en
ligne de commandes
est un
langage programmation
est un sous ensemble
objet
applications
scripts
création
logiciels
programmes
interfaces graphiques
Base de données
inspiré
Monty Python
où
téléchargeable sur internet
créé par
Guido Van Rossum
quand
création 1989- 2010: version 3
utilisation
avantages
grandes entreprises
syntaxe simple
communauté active
grande compatibilité
gratuit
libre
inconvénient
lent
documentation
bibliographie
www.python.org
code.google.com/p/swfk-fr/

[Sommaire]

Mémento Python 3

Dernière version sur :
https://perso.limsi.fr/pointal/python:memento

Types de base

entier, f ottant, booléen, chaîne, octets

```
int 783  0  -192  0b010 0o642 0xF3
          zéro    binaire octal hexa
float 9.23 0.0  -1.7e-6
                        ×10⁻⁶
bool True False
str "Un\nDeux"
     retour à la ligne échappé
    'L\'âme'
      ' échappé
bytes b"toto\xfe\775"
       hexadécimal  octal
```

Chaîne multiligne :
```
"""X\tY\t2
1\t2\t3"""
```
tabulation échappée

⁉ immutables

Types conteneurs

- séquences ordonnées, accès par index rapide, valeurs répétables

```
list [1,5,9]   ["x",11,8.9]   ["mot"]
tuple (1,5,9)  11,"y",7.4     ("mot",)
```
Valeurs non modif ables (immutables) — ⁉ *expression juste avec des virgules →* `tuple`

```
str bytes  (séquences ordonnées de caractères / d'octets)
```

- conteneurs clés, sans ordre *a priori*, accès par clé rapide, chaque clé unique

```
dictionnaire dict {"clé":"valeur"}   dict(a=3,b=4,k="v")
(couples clé/valeur) {1:"un",3:"trois",2:"deux",3.14:"n"}
ensemble     set {"clé1","clé2"}     {1,9,3,0}
⁉ clés=valeurs hachables (types base, immutables…)   frozenset ensemble immutable
```
vide : `set()`

Identif cateurs

pour noms de variables, fonctions, modules, classes…

`a…zA…Z_` suivi de `a…zA…Z_0…9`

- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

```
☺ a toto x7 y_max BigOne
☹ my and for
```

Variables & af ectation

⁉ af ectation ⇔ association d'un *nom* à une *valeur*

1) évaluation de la valeur de l'expression de droite
2) af ectation dans l'ordre avec les noms de gauche

```
x=1.2+8+sin(y)
a=b=c=0      af ectation à la même valeur
y,z,r=9.2,-7.6,0   af ectations multiples
a,b=b,a      échange de valeurs
a,*b=seq     dépaquetage de séquence en
*a,b=seq     élément et liste
x+=3         incrémentation ⇔ x=x+3
x-=2         décrémentation ⇔ x=x-2
x=None       valeur constante « non déf ni »
del x        suppression du nom x
```
et `*= /= %= …`

Conversions

```
int("15") → 15                type(expression)
int("3f",16) → 63             spécif cation de la base du nombre entier en 2nd paramètre
int(15.56) → 15               troncature de la partie décimale
float("-11.24e8") → -1124000000.0
round(15.56,1)→15.6           arrondi à 1 décimale (0 décimale → nb entier)
bool(x)  False pour x zéro, x conteneur vide, x None ou False ; True pour autres x
str(x)→"…"   chaîne de représentation de x pour l'af chage (cf. Formatage au verso)
chr(64)→'@'  ord('@')→64      code ↔ caractère
repr(x)→"…"  chaîne de représentation littérale de x
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"trois"),(1,"un")]) → {1:'un',3:'trois'}
set(["un","deux"]) → {'un','deux'}
str de jointure et séquence de str → str assemblée
    ':'.join(['toto','12','pswd']) → 'toto:12:pswd'
str découpée sur les blancs → list de str
    "des mots  espacés".split() → ['des','mots','espacés']
str découpée sur str séparateur → list de str
    "1,4,8,2".split(",") → ['1','4','8','2']
séquence d'un type → list d'un autre type (par liste en compréhension)
    [int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Indexation conteneurs séquences

pour les listes, tuples, chaînes de caractères, bytes…

```
index négatif   -5  -4  -3  -2  -1
index positif    0   1   2   3   4
lst=           [10, 20, 30, 40, 50]
tranche positive 0   1   2   3   4   5
tranche négative -5  -4  -3  -2  -1
```

Nombre d'éléments `len(lst)→5`

⁉ index à partir de 0 (de 0 à 4 ici)

Accès individuel aux éléments par `lst[index]`

`lst[0]→10` *⇒ le premier* `lst[1]→20`
`lst[-1]→50` *⇒ le dernier* `lst[-2]→40`

Sur les séquences modif ables (`list`), suppression avec `del lst[3]` et modif cation par af ectation `lst[4]=25`

Accès à des sous-séquences par `lst[`*tranche début*`:`*tranche f n*`:`*pas*`]`

```
lst[:-1]→[10,20,30,40]   lst[::-1]→[50,40,30,20,10]   lst[1:3]→[20,30]   lst[:3]→[10,20,30]
lst[1:-1]→[20,30,40]     lst[::-2]→[50,30,10]         lst[-3:-1]→[30,40] lst[3:]→[40,50]
lst[::2]→[10,30,50]      lst[:]→[10,20,30,40,50]
```
copie superf cielle de la séquence

Indication de tranche manquante → à partir du début / jusqu'à la f n.
Sur les séquences modif ables (`list`), suppression avec `del lst[3:5]` et modif cation par af ectation `lst[1:4]=[15,25]`

Logique booléenne

Comparateurs: `< > <= >= == !=`
(résultats booléens) ≤ ≥ = ≠

`a and b` et logique — *les deux en même temps*

`a or b` ou logique — *l'un ou l'autre ou les deux*

⁉ piège : `and` et `or` retournent la *valeur* de `a` ou de `b` (selon l'évaluation au plus court).
⇒ *s'assurer que* `a` *et* `b` *sont booléens.*

`not a` non logique

`True` `False` constantes Vrai/Faux

Blocs d'instructions

instruction parente :
bloc d'instructions 1…
instruction parente :
bloc d'instructions 2…
instruction suivante après bloc 1

indentation !

⁉ *régler l'éditeur pour insérer 4 espaces à la place d'une tabulation d'indentation.*

Imports modules/noms

module `truc` ⇔ *f chier* `truc.py`

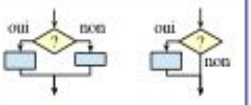
```
from monmod import nom1,nom2 as fct
```
→accès direct aux noms, renommage avec `as`
```
import monmod
```
→accès via `monmod.nom1` …

⁉ modules et packages cherchés dans le *python path* (cf. `sys.path`)

Instruction conditionnelle

un bloc d'instructions exécuté, uniquement si sa condition est vraie

```
if condition logique :
    bloc d'instructions
```



Combinable avec des *sinon si, sinon si…* et un seul *sinon* f nal. Seul le bloc de la première condition trouvée vraie est exécuté.

```
if age<=18:
    etat="Enfant"
elif age>65:
    etat="Retraité"
else:
    etat="Actif"
```

⁉ *avec une variable* `x`:
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:` ⇔ `if not x:`

Exceptions sur erreurs

Signalisation :
`raise ExcClass(…)`

Traitement :
```
try:
    bloc traitement normal
except ExcClass as e:
    bloc traitement erreur
```

⁉ *bloc* `finally` *pour traitements f naux dans tous les cas.*

Maths

⁉ *nombres f ottants… valeurs approchées !*

Opérateurs : `+ - * / // % **`
Priorités `(…)` × ÷ ↑ ↑ a^b
÷ entière reste ÷

`@` → × matricielle *python3.5+numpy*

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```
⁉ *priorités usuelles*

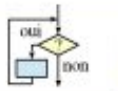
angles en radians

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0      √
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```
modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, etc.

Instruction boucle conditionnelle

bloc d'instructions exécuté tant que la condition est vraie

```
while condition logique :
    bloc d'instructions
```




```
s = 0          } initialisations avant la boucle
i = 1          } condition avec au moins une valeur variable (ici i)
while i <= 100:
    s = s + i**2
    i = i + 1      # faire varier la variable de condition !
print("somme:",s)
```

Algo : $s = \sum_{i=1}^{i=100} i^2$

⚠ attention aux boucles sans fin !

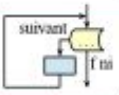
Contrôle de boucle

`break` *sortie immédiate*
`continue` *itération suivante*
⚠ *bloc* `else` *en sortie normale de boucle.*

Instruction boucle itérative

bloc d'instructions exécuté pour chaque élément d'un conteneur ou d'un itérateur

```
for var in séquence :
    bloc d'instructions
```



Parcours des valeurs d'un conteneur

```
s = "Du texte"   } initialisations avant la boucle
cpt = 0
for c in s:      # variable de boucle, affectation gérée par l'instruction for
    if c == "e":
        cpt = cpt + 1
print("trouvé",cpt,"'e'")
```

Algo : comptage du nombre de `e` *dans la chaîne.*

boucle sur dict/set ⇔ boucle sur séquence des clés
utilisation des *tranches* pour parcourir un sous-ensemble d'une séquence

Parcours des index d'un conteneur séquence
- changement de l'élément à la position
- accès aux éléments autour de la position (avant/après)

```
lst = [11,18,9,12,23,4,17]
perdu = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        perdu.append(val)
        lst[idx] = 15
print("modif:",lst,"-modif:",perdu)
```

Algo: bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

Parcours simultané index et valeurs de la séquence :

```
for idx,val in enumerate(lst):
```

⚠ bonne habitude : ne pas modifier la variable de boucle

Affichage

```
print("v=",3,"cm :",x,",",y+4)
```

éléments à afficher : valeurs littérales, variables, expressions

Options de `print` :
- `sep=" "` séparateur d'éléments, défaut espace
- `end="\n"` fin d'affichage, défaut fin de ligne
- `file=sys.stdout` print vers fichier, défaut sortie standard

Saisie

```
s = input("Directives:")
```

⚠ `input` retourne toujours une chaîne, la convertir vers le type désiré (cf. encadré *Conversions* au recto).

Opérations génériques sur conteneurs

`len(c)` → nb d'éléments
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` *copie triée*

Note: Pour dictionnaires et ensembles, ces opérations travaillent sur les clés.

`val in c` → booléen, opérateur `in` de test de présence (`not in` d'absence)
`enumerate(c)` → *itérateur* sur (index, valeur)
`zip(c1,c2…)` → *itérateur* sur tuples contenant les éléments de même index des c_i
`all(c)` → `True` si tout élément de `c` évalué vrai, sinon `False`
`any(c)` → `True` si au moins un élément de `c` évalué vrai, sinon `False`
`c.clear()` supprime le contenu des dictionnaires, ensembles, listes

Spécif que aux conteneurs de séquences ordonnées (listes, tuples, chaînes, bytes…)
`reversed(c)` → *itérateur inversé* `c*5` → duplication `c+c2` → concaténation
`c.index(val)` → *position* `c.count(val)` → *nb d'occurences*

```
import copy
copy.copy(c) → copie superficielle du conteneur
copy.deepcopy(c) → copie en profondeur du conteneur
```

Opérations sur listes

⚠ modification de la liste originale

`lst.append(val)`	ajout d'un élément à la fin
`lst.extend(seq)`	ajout d'une séquence d'éléments à la fin
`lst.insert(idx,val)`	insertion d'un élément à une position
`lst.remove(val)`	suppression du premier élément de valeur *val*
`lst.pop([idx])` → *valeur*	supp. & retourne l'item d'index *idx* (défaut le dernier)
`lst.sort()` `lst.reverse()`	tri / inversion de la liste sur place

Opérations sur dictionnaires

`d[clé]=valeur` `del d[clé]`
`d[clé]` → *valeur*
`d.update(d2)` mise à jour/ajout des couples
`d.keys()` `d.values()` `d.items()` → *vues itérables sur les clés / valeurs / couples*
`d.pop(clé[,défaut])` → *valeur*
`d.popitem()` → *(clé,valeur)*
`d.get(clé[,défaut])` → *valeur*
`d.setdefault(clé[,défaut])` → *valeur*

Opérations sur ensembles

Opérateurs :
- `|` → union (caractère barre verticale)
- `&` → intersection
- `-` `^` → différence/différence symétrique
- `<` `<=` `>` `>=` → relations d'inclusion

Les opérateurs existent aussi sous forme de méthodes.

`s.update(s2)` `s.copy()`
`s.add(clé)` `s.remove(clé)`
`s.discard(clé)` `s.pop()`

Fichiers

stockage de données sur disque, et relecture

```
f = open("fic.txt","w",encoding="utf8")
```

- variable fichier pour les opérations
- nom du fichier sur le disque (+chemin…) — cf modules `os`, `os.path` et `pathlib`
- mode d'ouverture
  - `'r'` lecture (read)
  - `'w'` écriture (write)
  - `'a'` ajout (append)
  - …`'+'` `'x'` `'b'` `'t'`
- encodage des caractères pour les *fichiers textes:* `utf8` `ascii` `latin1` …

en écriture

```
f.write("coucou")
f.writelines(list de lignes)
```

en lecture

⚠ *lit chaîne vide si fin de fichier*

```
f.read([n])          → caractères suivants
                       si n non spécifié, lit jusqu'à la fin !
f.readlines([n])     → list lignes suivantes
f.readline()         → ligne suivante
```

⚠ *par défaut mode texte* `t` *(lit/écrit* `str`*), mode binaire* `b` *possible (lit/écrit* `bytes`*). Convertir de/vers le type désiré !*

`f.close()` ⚠ ne pas oublier de refermer le fichier après son utilisation !

`f.flush()` écriture du cache `f.truncate([taille])` retaillage

lecture/écriture progressent séquentiellement dans le fichier, modifiable avec :
`f.tell()` → *position* `f.seek(position[,origine])`

Très courant : ouverture en bloc gardé (fermeture automatique) et boucle de lecture des lignes d'un fichier texte.

```
with open(…) as f:
    for ligne in f :
        # traitement de ligne
```

Séquences d'entiers

`range([début,] fin [,pas])`

⚠ *début* défaut 0, *fin* non compris dans la séquence, *pas* signé et défaut 1

`range(5)` → 0 1 2 3 4 `range(2,12,3)` → 2 5 8 11
`range(3,8)` → 3 4 5 6 7 `range(20,5,-5)` → 20 15 10
`range(len(séq))` → *séquence des index des valeurs dans séq*

⚠ *range fournit une séquence immutable d'entiers construits au besoin*

Définition de fonction

nom de la fonction (identificateur) — paramètres nommés

```
def fct(x,y,z):
    """documentation"""
    # bloc instructions, calcul de res, etc.
    return res
```

`return res` ← valeur résultat de l'appel, si pas de résultat calculé à retourner : `return None`

⚠ les paramètres et toutes les variables de ce bloc n'existent que *dans* le bloc et *pendant* l'appel à la fonction (penser *"boîte noire"*)

Avancé : `def fct(x,y,z,*args,a=3,b=5,**kwargs):`

**args nb variables d'arguments positionnels (→* `tuple`*), valeurs par défaut, **kwargs nb variable d'arguments nommés (→* `dict`*)*

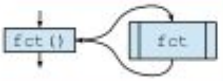
Appel de fonction

```
r = fct(3,i+2,2*i)
```

stockage/utilisation de la valeur de retour — *une valeur d'argument par paramètre*

⚠ c'est l'utilisation du nom de la fonction *avec les parenthèses* qui fait l'appel

*Avancé: *séquence **dict*



Opérations sur chaînes

`s.startswith(pref x[,début[,fin]])` `s.endswith(suf x[,début[,fin]])` `s.strip([caractères])`
`s.count(sub[,début[,fin]])` `s.partition(sep)` → *(avant,sep,après)*
`s.index(sub[,début[,fin]])` `s.find(sub[,début[,fin]])`
`s.is…()` *tests sur les catégories de caractères (ex.* `s.isalpha()`*)*
`s.upper()` `s.lower()` `s.title()` `s.swapcase()`
`s.casefold()` `s.capitalize()` `s.center([larg,rempl])`
`s.ljust([larg,rempl])` `s.rjust([larg,rempl])` `s.zfill([larg])`
`s.encode(codage)` `s.split([sep])` `s.join(séq)`

Formatage

directives de formatage — valeurs à formater

```
"modele{} {} {}".format(x,y,r) → str
"{sélection:formatage!conversion}"
```

- Sélection :
  - `2`
  - `nom`
  - `0.nom`
  - `4[clé]`
  - `0[2]`

Exemples :

```
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'      toto'
"{x!r}".format(x="L'ame")
→'"L\'ame"'
```

- Formatage :

car-rempl. *alignement* *signe* *larg mini* . *précision-larg max* *type*

`<` `>` `^` `=` `+` `-` *espace* `0` au début pour remplissage avec des 0

entiers : `b` binaire, `c` caractère, `d` décimal (défaut), `o` octal, `x` ou `X` hexa…
flottant : `e` ou `E` exponentielle, `f` ou `F` point fixe, `g` ou `G` approprié (défaut),
chaîne : `s` … `%` pourcentage

- Conversion : `s` (texte lisible) ou `r` (représentation littérale)