Introduction au Python

1. Calculs

Une des premières fonctionnalités d'un interpréteur est de faire des calculs:

```
>>> 1+2

3
```

Vous pouvez ajouter des espaces, cela n'aura pas d'incidences:

```
>>> 1 + 2

3
```

Tous les opérateurs sont utilisables:

```
>>> 1-10

-9

>>> 2*10

20

>>> 100/4

25

>>> 10%4

2

>>> 2**3

8
```

La double étoile représente l'exposant.

Il existe cependant une erreur à éviter pour les versions Python inférieures à Python 3 :

```
>>> 10/3

3
```

Surprise 10/3 = 3. Alors quelle est donc cette folie? Python raisonne en nombres entiers puisque nous lui avons fourni deux nombres entiers. Pour avoir un résultat en décimales, il vous faudra utiliser cette syntaxe:

```
>>> 10.0/3

3.3333333333333335

>>> 10/3.0

3.3333333333333335
```

Les variables

Une variable est une sorte de **boite virtuelle** dans laquelle on peut mettre une (ou plusieurs) donnée(s). L'idée est de stocker temporairement une donnée pour travailler avec. Pour votre machine une variable est une adresse qui indique l'emplacement de la mémoire vive où sont stockées les informations que nous avons liées avec.

Affectons une valeur à la variable **age** que nous allons ensuite afficher:

```
>>> age = 30

>>> age

30
```

On va ensuite ajouter 10 à la valeur de cette variable:

```
>>> age = 30

>>> age = age + 10

>>> age

40
```

Il est possible de mettre une variable dans une autre variable.

```
>>> age = 30

>>> age2 = age

>>> age2

30
```

Vous pouvez mettre à peu près tout ce que vous voulez dans votre variable, y compris du texte:

```
>>> age = "J'ai 30 ans"

>>> age

"J'ai 30 ans"
```

Il est possible de concaténer, c'est à dire d'ajouter du texte à du texte:

```
>>> age = age + " et je suis encore jeune!"

>>> age

"J'ai 30 ans et je suis encore jeune!"
```

Vous pouvez même multiplier une chaine de caractères:

```
>>> age = "jeune"

>>> age * 3

'jeunejeunejeune'
```

Evidemment, si vous essayez de faire des additions avec des variables qui sont des chiffres et d'autres qui sont du texte, l'interpréteur va vous gronder:

```
>>> age = "J'ai 30 ans"

>>> age

"J'ai 30 ans"

>>> age + 1

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: cannot concatenate 'str' and 'int' objects
```

Vous remarquez que l'interpréteur est sympa puisqu'il vous dit ce qui ne va pas: Il ne peut pas concaténer **str** et **int** .

Echapper les quotes

Comment inclure le caractère quote puisqu'il indique un début de données et en fin de données?

```
>>> texte = "Bonjour je m'appelle \"Olivier\""

>>>  texte

Bonjour je m'appelle "Olivier"
```

ou

```
>>> texte = 'Bonjour je m\'appelle "Olivier"'

>>> texte

'Bonjour je m\'appelle "Olivier"'
```

Il existe une autre manière de stocker du texte dans une variable: l'utilisation d'un triple quote. Cela permet de ne pas échapper les caractères quotes des données

```
>>> texte = """

... Bonjour je m'appelle "olivier"

... """

>>> texte

'\nBonjour je m\'appelle "olivier"\n'
```

Nommer une variable

Vous ne pouvez pas nommer les variables comme bon vous semble, puisqu'il existe déjà des mots utilisés par Python. Voici la liste des mots réservé par python:

```
print in and or if del for is raise assert elif from lambda return break else global not try class exc
ept while continue exec import pass yield def finally
```

Pourquoi ces mots sont-ils réservés? Parce qu'ils servent à faire autre chose. Nous verrons quoi plus en détail dans les prochains chapitres.

Pour nommer une variable vous devez obligatoirement utiliser les lettres de l'alphabet, les chiffres et le caractère "_" et "-". N'utilisez pas les accents, ni les signe de ponctuation ou le signe @. De plus les chiffres ne doivent jamais se trouver en première position dans votre variable:

```
>>> 1var = 1

  File "<stdin>", line 1

    1var = 1
```

```
    ^

SyntaxError: invalid syntax
```

Comme vous le remarquez, python refuse ce genre de syntaxe, mais il acceptera **var1 = 1** .

Les types de variables

En python une variable est typée , c'est à dire qu'en plus d'une valeur, une variable possède une sorte d'étiquette qui indique ce que contient cette boite virtuelle.

Voici une liste de type de variable:

- **Les *integer* ou nombres entiers** : comme son nom l'indique un entier est un chiffre sans décimales.
- **Les *float* ou nombre à virgules** : exemple : 1.5
- **Les *strings* ou chaine de caractères** : pour faire simple tout ce qui n'est pas chiffre.

Il en existe plein d'autres mais il est peut-être encore un peu trop tôt pour vous en parler.

Pour connaitre le type d'une variable, vous pouvez utiliser la fonction " **type()** "

```
>>> v = 15

>>> type(v)

<type 'int'>

>>> v = "Olivier"

>>> type(v)

<type 'str'>

>>> v = 3.2

>>> type(v)

<type 'float'>
```

La lib Décimal pour les calculs

Le résultat de calculs en python peut être différent en fonction des machines que vous utilisez si vous travaillez avec plus de précisions utilisez Décimal.

```
from decimal import Decimal

>>> a = 10
```

```
>>> b = 3

>>> a/b

3.3333333333333335

>>> a = Decimal('10')

>>> b = Decimal('3')

>>> a/b

Decimal('3.3333333333333333333333333333')
```

Traitement Conditions En Python

Cette notion est l'une des plus importante en programmation. L'idée est de dire que **si** telle variable a telle valeur **alors** faire cela **sinon** cela.

Prenons un exemple, on va donner une valeur à une variable et si cette valeur est supérieure à 5, alors on va incrémenter la valeur de 1

```
>>> a = 10

>>> if a > 5:

...     a = a + 1

...

>>> a

11
```

Que se passe-t-il si la valeur était inférieure à 5?

```
>>> a = 3

>>> if a > 5:

...     a = a + 1

...

>>> a

3
```

On remarque que si la condition n'est pas remplie, les instructions dans la structure conditionnelle sont ignorées.

Condition if else

Il est possible de donner des instructions quel que soit les choix possibles avec le mot clé `else`.

```
>>> a = 20

>>> if a > 5:

...     a = a + 1

... else:

...     a = a - 1

...

>>> a

21
```

Changeons uniquement la valeur de la variable **a** :

```
>>> a = 3

>>> if a > 5:

...     a = a + 1

... else:

...     a = a - 1

...

>>> a

2
```

Condition elif

Il est possible d'ajouter autant de conditions précises que l'on souhaite en ajoutant le mot clé `elif` , contraction de "else" et "if", qu'on pourrait traduire par "sinon".

```
>>> a = 5
```

```
>>> if a > 5:
...     a = a + 1
... elif a == 5:
...     a = a + 1000
... else:
...     a = a - 1
...
>>> a
1005
```

Dans cet exemple, on a repris le même que les précédent mais nous avons ajouté la conditions "Si la valeur est égale à 5" que se passe-t-il? Et bien on ajoute 1000.

Les comparaisons possibles

Il est possible de comparer des éléments:

```
==    égal à
!=    différent de (fonctionne aussi avec )
>     strictement supérieur à
>=    supérieur ou égal à
<     strictement inférieur à
<=    inférieur ou égal à
```

Comment fonctionne les structures conditionnelles?

Les mots clé if, elif et else cherchent à savoir si ce qu'on leur soumet est `True`. En anglais *True* signifie "Vrai". Donc si c'est la valeur est True, les instructions concernant la condition seront exécutées.

Comment savoir si la valeur qu'on soumet à l'interpréteur est True? Il est possible de le voir directement dans l'interpréteur.

Demandons à python si 3 est égal à 4:

```
>>> 3 == 4

False
```

Il vous répondra gentiment que c'est `False` , c'est à dire que c'est faux .

Maintenant on va donner une valeur à une variable est on va lui demander si la valeur correspond bien à ce que l'on attend.

```
>>> a = 5

>>> a == 5

True
```

AND / OR

Il est possible d'affiner une condition avec les mots clé `AND` qui signifie " **ET** " et `OR` qui signifie " **OU** ".

On veut par exemple savoir si une valeur est plus grande que 5 mais aussi plus petite que 10:

```
>>> v = 15

>>> v > 5 and v < 10

False
```

Essayons avec la valeur **7** :

```
>>> v = 7

>>> v > 5 and v < 10

True
```

Pour que le résultat soit `TRUE`, il faut que **les deux conditions soient remplies** .

Testons maintenant la condition `OR`

```
>>> v = 11

>>> v > 5 or v > 100

True
```

Le résultat est `TRUE` parce qu' **au moins une des deux conditions est respectée** .

```
>>> v = 1
```

```
>>> v > 5 or v > 100

False
```

Dans ce cas-là aucune condition n'est respectée, le résultat est donc `FALSE`.

Chainer les comparateurs

Il est également possible de chainer les comparateurs:

```
>>> a, b, c = 1, 10, 100

>>> a < b < c

True

>>> a > b < c

False
```

Les boucles for et while Python

La boucle while

En anglais " *while* " signifie "Tant que". Pour créer une **boucle**, il faut donc utiliser ce mot clé suivi d'une indication qui dit quand la boucle s'arrête.

Un exemple sera plus parlant:

On désire écrire 100 fois cette phrase:

" *Je ne dois pas poser une question sans lever la main* "

Ecrire à la main prend beaucoup de temps et beaucoup de temps x 100 c'est vraiment beaucoup de temps, et peu fiable, même pour les chanceux qui connaissent le copier-coller. Et un bon programmeur est toujours un peu ~~*fainéant*~~ perfectionniste, il cherchera la manière la plus élégante de ne pas répéter du code.

```
>>> i = 0

>>> while i < 10:

...     print("Je ne dois pas poser une question sans lever la main")

...     i = i +1
```

```
...

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main

Je ne dois pas poser une question sans lever la main
```

La boucle for

La boucle `for` permet de faire des **itérations** sur un élément, comme une chaine de caractères par exemple ou une liste.

Exemple:

```
>>> v = "Bonjour toi"

>>> for lettre in v:

...     print(lettre)

...

B

o

n

j

o

u

r
```

```
t

o

i
```

Range

Il est possible de créer une boucle facilement avec `range` :

```
for i in range(0,100):

 print(i)
```

Stopper une boucle avec break

Pour stopper immédiatement une boucle on peut utiliser le mot clé `break` :

```
>>> liste = [1,5,10,15,20,25]

>>> for i in liste:

...     if i > 15:

...         print("On stoppe la boucle")

...         break

...     print(i)

...

1

5

10

15
```

Les listes python

Les **listes** (ou **list** / **array**) en **python** sont une variable dans laquelle on peut mettre plusieurs variables.

1. Créer une liste en python

Pour créer une **liste** , rien de plus simple:

```
>>> liste = []
```

Vous pouvez voir le contenu de la **liste** en l'appelant comme ceci:

```
>>> liste

<type 'list'>
```

Ajouter une valeur à une liste python

Vous pouvez ajouter les valeurs que vous voulez lors de la création de la **liste python** :

```
>>> liste = [1,2,3]

>>> liste

[1, 2, 3]
```

Ou les ajouter après la création de la liste avec la méthode `append` (qui signifie "ajouter" en anglais):

```
>>> liste = []

>>> liste

[]

>>> liste.append(1)

>>> liste

[1]

>>> liste.append("ok")

>>> liste

[1, 'ok']
```

On voit qu'il est possible de mélanger dans une même liste des variables de type différent. On peut d'ailleurs mettre une liste dans une liste.

Afficher un item d'une liste

Pour lire une liste, on peut demander à voir l'index de la valeur qui nous intéresse:

```
>>> liste = ["a","d","m"]

>>> liste[0]

'a'

>>> liste[2]

'm'
```

Le premier item commence toujours avec l'index 0. Pour lire la premier item on utilise la valeur 0, le deuxième on utilise la valeur 1, etc.

Il est d'ailleurs possible de modifier une valeur avec son index

```
>>> liste = ["a","d","m"]

>>> liste[0]

'a'

>>> liste[2]

'm'

>>> liste[2] = "z"

>>> liste

['a', 'd', 'z']
```

Supprimer une entrée avec un index

Il est parfois nécessaire de supprimer une entrée de la liste. Pour cela vous pouvez utiliser la fonction `del` .

```
>>> liste = ["a", "b", "c"]

>>> del liste[1]

>>> liste
```

```
['a', 'c']
```

Supprimer une entrée avec sa valeur

Il est possible de supprimer une entrée d'une liste avec sa valeur avec la méthode `remove` .

```
>>> liste = ["a", "b", "c"]

>>> liste.remove("a")

>>> liste

['b', 'c']
```

Inverser les valeurs d'une liste

Vous pouvez inverser les items d'une liste avec la méthode `reverse` .

```
>>> liste = ["a", "b", "c"]

>>> liste.reverse()

>>> liste

['c', 'b', 'a']
```

Compter le nombre d'items d'une liste

Il est possible de compter le nombre d'items d'une liste avec la fonction `len` .

```
>>> liste = [1,2,3,5,10]

>>> len(liste)

5
```

Compter le nombre d'occurrences d'une valeur

Pour connaitre le nombre d'occurrences d'une valeur dans une liste, vous pouvez utiliser la méthode `count` .

```
>>> liste = ["a","a","a","b","c","c"]

>>> liste.count("a")

3

>>> liste.count("c")
```

```
2
```

Trouver l'index d'une valeur

La méthode `index` vous permet de connaitre la position de l'item cherché.

```
>>> liste = ["a","a","a","b","c","c"]

>>> liste.index("b")

3
```

Manipuler une liste

Voici quelques astuces pour manipuler des listes:

```
>>> liste = [1, 10, 100, 250, 500]

>>> liste[0]

1

>>> liste[-1] # Cherche la dernière occurence

500

>>> liste[-4:] # Affiche les 4 dernières occurrences

[500, 250, 100, 10]

>>> liste[:] # Affiche toutes les occurences

[1, 10, 100, 250, 500]

>>> liste[2:4] = [69, 70]

[1, 10, 69, 70, 500]

>>> liste[:] = [] # vide la liste

[]
```

Boucler sur une liste

Pour afficher les valeurs d'une liste, on peut utiliser une boucle:

```
>>> liste = ["a","d","m"]
```

```
>>> for lettre in liste:
...     print lettre
...
a
d
m
```

Si vous voulez en plus récupérer l'index, vous pouvez utiliser la fonction `enumerate` .

```
>>> for lettre in enumerate(liste):
...     print lettre
...
(0, 'a')
(1, 'd')
(2, 'm')
```

Les valeurs retournées par la boucle sont des **tuples.**

Copier une liste

Beaucoup de débutants font l'erreur de copier une liste de cette manière

```
>>> x = [1,2,3]
>>> y = x
```

Or si vous changez une valeur de la liste **y** , la liste **x** sera elle aussi affectée par cette modification:

```
>>> x = [1,2,3]
>>> y = x
>>> y[0] = 4
>>> x
```

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

En fait cette syntaxe permet de travailler sur un même élément nommé différemment

Alors comment copier une liste qui sera indépendante?

```
>>> x = [1,2,3]

>>> y = x[:]

>>> y[0] = 9

>>> x

[1, 2, 3]

>>> y

[9, 2, 3]
```

Pour des données plus complexes, vous pouvez utiliser la fonction `deepcopy` du module `copy`

```
>>> import copy

>>> x = [[1,2], 2]

>>> y = copy.deepcopy(x)

>>> y[1] = [1,2,3]

>>> x

[[1, 2], 2]

>>> y

[[1, 2], [1, 2, 3]]
```

Transformer une string en liste

Parfois il peut être utile de transformer une chaine de caractère en liste. Cela est possible avec la méthode `split` .

```
>>> ma_chaine = "Olivier:ENGEL:Strasbourg"

>>> ma_chaine.split(":")
```

```
['Olivier', 'ENGEL', 'Strasbourg']
```

Transformer une liste en string

L'inverse est possible avec la méthode " **join** ".

```
>>> liste = ["Olivier","ENGEL","Strasbourg"]

>>> ":".join(liste)

'Olivier:ENGEL:Strasbourg'
```

Trouver un item dans une liste

Pour savoir si un élément est dans une liste, vous pouvez utiliser le mot clé **in** de cette manière:

```
>>> liste = [1,2,3,5,10]

>>> 3 in liste

True

>>> 11 in liste

False
```

La fonction range

La fonction `range` génère une liste composée d'une simple suite arithmétique.

```
>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Agrandir une liste par une liste

Pour mettre bout à bout deux listes, vous pouvez utiliser la méthode `extend`

```
>>> x = [1, 2, 3, 4]

>>> y = [4, 5, 1, 0]

>>> x.extend(y)

>>> print x

[1, 2, 3, 4, 4, 5, 1, 0]
```

Permutations

La permutation d'un ensemble d'éléments est une liste de tous les cas possibles. Si vous avez besoin de cette fonctionnalité, inutile de réinventer la roue, **itertools** s'en occupe pour vous.

```
>>> from itertools import permutations

>>> list(permutations(['a', 'b', 'c']))

[('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'), ('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a')]
```

Permutation d'une liste de liste

Comment afficher tous les cas possibles d'une liste elle-même composée de liste? Avec l'outil **product** de **itertools** :

```
>>> from itertools import product

>>> list(product(['a', 'b'], ['c', 'd']))

[('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')]
```

Astuces

Afficher les 2 premiers éléments d'une liste

```
>>> liste = [1,2,3,4,5]

>>> liste[:2]

[1, 2]
```

Afficher le dernier item d'une liste:

```
>>> liste = [1, 2, 3, 4, 5, 6]

>>> liste[-1]

6
```

Afficher le 3ème élément en partant de la fin:

```
>>> liste = [1, 2, 3, 4, 5, 6]

>>> liste[-3]

4
```

Afficher les 3 derniers éléments d'une liste:

```
>>> liste = [1, 2, 3, 4, 5, 6]

>>> liste[-3:]

[4, 5, 6]
```

Vous pouvez additionner deux listes pour les combiner ensemble en utilisant l'opérateur `+` :

```
>>> x = [1, 2, 3]

>>> y = [4, 5, 6]

>>> x + y

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

Vous pouvez même mutiplier une liste:

```
>>> x = [1, 2]

>>> x*5

[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Ce qui peut être utile pour initialiser une liste:

```
>>> [0] * 5

[0, 0, 0, 0, 0]
```

Les tuples python

Un **tuple** est une **liste** qui ne peut plus être modifiée.

Créer un tuple

Pour créer un **tuple**, vous pouvez utiliser la syntaxe suivante:

```
>>> mon_tuple = ()
```

Ajouter une valeur à un tuple

Pour créer un **tuple** avec des valeurs, vous pouvez le faire de cette façon:

```
>>> mon_tuple = (1, "ok", "olivier")
```

Les parenthèses ne sont pas obligatoires mais facilite la lisibilité du code (rappelons que la force de python est sa simplicité de lecture):

```
>>> mon_tuple = 1, 2, 3

>>> type(mon_tuple)

<type 'tuple'>
```

Lorsque vous créez un tuple avec une seule valeur, n'oubliez pas d'y ajouter une virgule, sinon ce n'est pas un tuple.

```
>>> mon_tuple = ("ok")

>>> type(mon_tuple)

<type 'str'>

>>> mon_tuple = ("ok",)

>>> type(mon_tuple)

<type 'tuple'>
```

Afficher une valeur d'un tuple

Le tuple est une sorte de liste, on peut donc utiliser la même syntaxe pour lire les données du tuple.

```
>>> mon_tuple[0]

1
```

Et évidemment si on essaie de changer la valeur d'un index, l'interpréteur nous insulte copieusement:

```
>>> mon_tuple[1] = "ok"

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment
```

A quoi sert un tuple alors?

Le tuple permet une affectation multiple:

```
>>> v1, v2 = 11, 22
```

```
>>> v1

11

>>> v2

22
```

Il permet également de renvoyer plusieurs valeurs lors d'un appel d'une fonction:

```
>>> def donne_moi_ton_nom():

...     return "olivier", "engel"

...

>>> donne_moi_ton_nom()

('olivier', 'engel')
```

Les dictionnaires python

Un **dictionnaire** en **python** est une sorte de **liste** mais au lieu d'utiliser des **index**, on utilise des clés alphanumériques.

Comment créer un dictionnaire python?

Pour initialiser un **dictionnaire**, on utile la syntaxe suivante:

```
>>> a = {}
```

ou

```
>>> a = dict()
```

Comment ajouter des valeurs dans un dictionnaire python?

Pour ajouter des valeurs à un **dictionnaire** il faut indiquer une clé ainsi qu'une valeur:

```
>>> a = {}

>>> a["nom"] = "Wayne"

>>> a["prenom"] = "Bruce"

>>> a

{'nom': 'Wayne', 'prenom': 'Bruce'}
```

Vous pouvez utiliser des clés numériques comme dans la logique des listes .

Comment récupérer une valeur dans un dictionnaire python?

La méthode *get* vous permet de **récupérer une valeur** dans un **dictionnaire** et si la clé est introuvable, vous pouvez donner une valeur à retourner par défaut:

```
>>> data = {"name": "Wayne", "age": 45}

>>> data.get("name")

'Wayne'

>>> data.get("adresse", "Adresse inconnue")

'Adresse inconnue'
```

Comment vérifier la présence d'une clé dans un dictionnaire python?

Vous pouvez utiliser la méthode *haskey* pour vérifier la présence d'une clé que vous cherchez:

```
>>> a.has_key("nom")

True
```

Comment supprimer une entrée dans un dictionnaire python?

Il est possible de supprimer une entrée en indiquant sa clé, comme pour les listes:

```
>>> del a["nom"]

>>> a

{'prenom': 'Bruce'}
```

Comment récupérer les clés d'un dictionnaire python par une boucle?

Pour récupérer les clés on utilise la méthode *keys* .

```
>>> fiche = {"nom":"Wayne", "prenom":"Bruce"}

>>> for cle in fiche.keys():

...     print cle

...
```

```
nom

prenom
```

Comment récupérer les valeurs d'un dictionnaire python par une boucle?

Pour cela on utilise la méthode *values* .

```
>>> fiche = {"nom":"Wayne", "prenom":"Bruce"}

>>> for valeur in fiche.values():

...     print valeur

...

Wayne

Bruce
```

Comment récupérer les clés et les valeurs d'un dictionnaire python par une boucle?

Pour récupérer les clés et les valeurs en même temps, on utilise la méthode *items* qui retourne un **tuple** .

```
>>> fiche = {"nom":"Wayne","prenom":"Bruce"}

>>> for cle,valeur in fiche.items():

...     print cle, valeur

...

nom Wayne

prenom Bruce
```

Comment utiliser des tuples comme clé dans un dictionnaire python?

Une des forces de python est la combinaison **tuple/dictionnaire** qui fait des merveilles dans certains cas comme lors de l'utilisation de coordonnées.

```
>>> b = {}

>>> b[(32)]=12

>>> b[(45)]=13
```

```
>>> b

{(4, 5): 13, (3, 2): 12}
```

Comment créer une copie indépendante d'un dictionnaire python?

Comme pour toute variable, vous ne pouvez pas **copier** un **dictionnaire** en faisant *dic1 = dic2* :

```
>>> d = {"k1":"Bruce", "k2":"Wayne"}

>>> e = d

>>> d["k1"] = "BATMAN"

>>> e

{'k2': 'Wayne', 'k1': 'BATMAN'}
```

Pour créer une **copie indépendante** vous pouvez utiliser la méthode **copy** :

```
>>> d = {"k1":"Bruce", "k2":"Wayne"}

>>> e = d.copy()

>>> d["k1"] = "BATMAN"

>>> e

{'k2': 'Wayne', 'k1': 'Bruce'}
```

Comment fusionner des dictionnaires python?

La méthode update permet de **fusionner deux dictionaires** .

```
>>> a = {'nom': 'Wayne'}

>>> b = {'prenom': 'bruce'}

>>> a.update(b)

>>> print(a)

{'nom': 'Wayne', 'prenom': 'Bruce'}
```