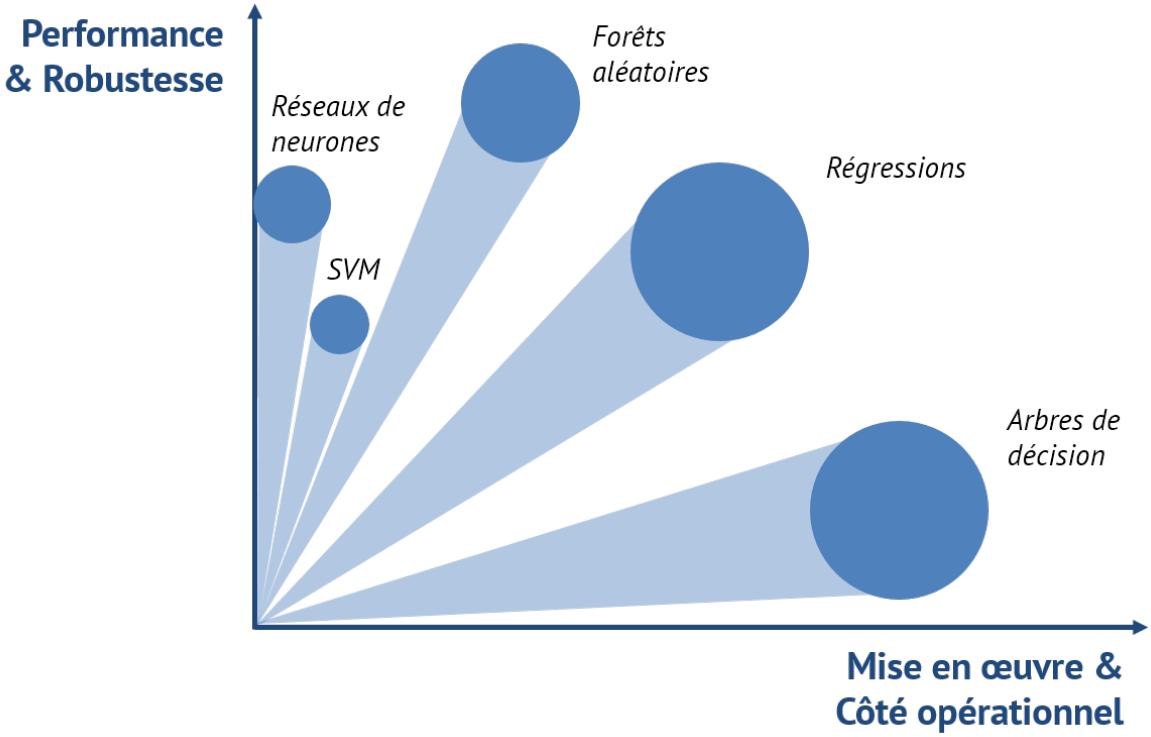


Technique de Big Data Analytics avec Python

Cours conçu par Razvan BIZOÏ

Razvan.BIZOI@laposte.net

2

L'introduction à Python

Le nom de variable

Pour pouvoir accéder aux données, le programme d'ordinateur fait abondamment usage d'un grand nombre de variables de différents types. Une variable est une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive.

Les noms de variables doivent obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (**a ÷ z, A ÷ Z**) et de chiffres (**0 ÷ 9**), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que **$**, **#**, **@**, etc. sont interdits, à l'exception du caractère _ (souligné).
- La casse est significative (les caractères majuscules et minuscules sont distingués). Attention : **Variable**, **variable**, **VARIABLE** sont donc des variables différentes.

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules. Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans **tableDesMatieres**.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser les mots réservés ci-dessous :

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

Pour leur première initialisation, une variable reçoit dans la plupart des cas une constante. Les constantes sont le contraire des variables, ce sont toutes les valeurs numériques, chaînes de caractères, ..., tout ce qui n'est pas désigné par un nom. Les constantes possèdent un type mais pas d'identificateur.

L'affectation

L'affectation désigne l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur. En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe égale « **=** ».

Python fait la distinction entre l'opérateur d'affectation « **=** » et l'opérateur de comparaison **« == »**.

```
>>> varInt = 12
>>> varStr = 'chaîne de caractères'
>>> varFloat = 15.5
>>> varInt == varFloat
False
>>> varInt == varStr
False
```

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un nom de variable ;
- lui attribuer un type bien déterminé ;
- créer et mémoriser une valeur particulière ;
- établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

Les trois noms de variables sont des références, mémorisées dans une zone particulière de la mémoire que l'on appelle **espace de noms**, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres.

L'affichage de la valeur d'une variable

Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable. Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours la fonction **`print()`**.

```
>>> varInt,varStr,varFloat
(12, 'chaîne de caractères', 15.5)
>>> varStr
'chaîne de caractères'
>>> print(varStr)
chaîne de caractères
```

La fonction **`print()`** n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode affiche aussi des apostrophes afin de vous rappeler que la variable traitée est du type **« `chaîne de caractères` »**.

Le typage des variables

Dans Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. On dira à ce sujet que le typage des variables sous Python est un typage dynamique, par opposition au typage statique qui est de règle par exemple en C++ ou en Java.

Le typage dynamique quant à lui permet d'écrire plus aisément des constructions logiques de niveau élevé, en particulier dans le contexte de la programmation orientée objet. Il facilite également l'utilisation de structures de données très riches telles que les listes et les dictionnaires.

```
>>> varInt,varStr,varFloat
 (12, 'chaîne de caractères', 15.5)
>>> varInt = varStr = varFloat
>>> varInt,varStr,varFloat
(15.5, 15.5, 15.5)
>>> varStr = 'chaîne de caractères'
>>> varInt,varStr,varFloat
(15.5, 'chaîne de caractères', 15.5)
>>> varInt == 20                    # ce n'est pas une affectation
False
>>> varInt,varStr,varFloat
(15.5, 'chaîne de caractères', 15.5)
>>> varInt = 20
>>> varInt,varStr,varFloat
(20, 'chaîne de caractères', 15.5)
```

Les affectations multiples

Dans Python, on peut assigner une valeur à plusieurs variables simultanément. On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur.

```
>>> varInt,varStr,varFloat
 (12, 'chaîne de caractères', 15.5)
>>> varInt = varStr = varFloat
>>> varInt,varStr,varFloat
(15.5, 15.5, 15.5)
>>> varInt,varStr,varFloat = (30, "l'autre chaîne", 3.14)
>>> varInt,varStr,varFloat
(30, "l'autre chaîne", 3.14)
>>> r , pi = 12, 3.14159
>>> s = pi * r**2
>>> print(s)
452.38896
>>> print(type(r), type(pi), type(s))
<class 'int'> <class 'float'> <class 'float'>
>>> h, m, s = 15, 27, 34
>>> print("secondes écoulées depuis minuit = ", h*3600 + m*60 + s)
secondes écoulées depuis minuit =  55654
```

La virgule, est très généralement utilisée pour séparer différents éléments comme on le voit dans notre exemple, pour les variables elles-mêmes ainsi que pour les valeurs qu'on leur attribue.

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de règles de priorité. Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique. Vous pouvez les mémoriser aisément à l'aide d'un acronyme mnémotechnique, **PEMDAS** (parenthèses, exposants, multiplications, divisions, additions et soustractions).

La réaffectation

L'effet d'une réaffectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.

```
>>> altitude = 320
>>> print(altitude)
320
>>> altitude = 375
>>> print(altitude)
375
>>> a = 5
>>> b = a       # a et b contiennent des valeurs égales
>>> b = 2       # a et b sont maintenant différentes
>>> print("a = ",a,"b = ",b)
a =  5 b =  2
```

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément.

```
>>> a, b, c, d = 3, 4, 5, 7
>>> print("a = ",a,"b = ",b,"c = ",c,"d = ",d)
a =  3 b =  4 c =  5 d =  7
```

Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **b**.

```
>>> a,b = b,a
>>> print("a = ", a,"b = ", b)
a =  4 b =  3
```

On pourrait bien entendu échanger d'autres variables en même temps, dans la même instruction.

La fonction input

La fonction intégrée « **input** » provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec « **Enter** ».

```
>>> prenom = input("Entrez votre prénom : ")
Entrez votre prénom : Razvan
>>> print("Bonjour,", prenom)
Bonjour, Razvan
>>> print("Veuillez entrer un nombre positif quelconque : ")
Veuillez entrer un nombre positif quelconque :
>>> 10
10
>>> print("Veuillez entrer un nombre positif quelconque : ",
... end=" ")
Veuillez entrer un nombre positif quelconque :
>>> 10
10
>>> ch = input()
20
>>> nn = int(ch)       # conversion de la chaîne en un nombre entier
>>> print("Le carré de", nn, "vaut", nn**2)
Le carré de 20 vaut 400
>>> a = input("Entrez une donnée numérique : ")
Entrez une donnée numérique : 25
>>> print(a)
25
```

Soulignons que la fonction « **input** » renvoie toujours une chaîne de caractères. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée en une valeur numérique du type qui vous convient, par l'intermédiaire des fonctions intégrées « **int** » ou « **float** ».

Atelier 2 - Exercice 1.1

Les types de données

Les types incontournables en Python sont classés le plus souvent en deux catégories : types immuables ou modifiables. Tous les types du langage Python sont également des objets, c'est pourquoi on retrouve dans ce chapitre certaines formes d'écriture similaires à celles présentées plus tard dans le chapitre concernant les classes.

Types immuables

Une variable de type immuable ne peut être modifiée. Une opération sur une variable de ce type entraîne nécessairement la création d'une autre variable du même type, même si cette dernière est temporaire.

Autrement dit, la simple instruction **x + = 3** qui consiste à ajouter à la variable **x** la valeur **3** crée une seconde variable dont la valeur est celle de **x** augmentée de **3** puis à en recopier le contenu dans celui de la variable **x**. Les nombres sont des types immuables tout comme les chaînes de caractères et les tuple qui sont des tableaux d'objets. Il n'est pas possible de modifier une variable de ce type, il faut en recréer une autre du même type qui intègrera la modification.

Type « rien »

Python propose un type **None** pour signifier qu'une variable ne contient rien. La variable est de type **None** et est égale à **None**.

```
>>> s  =  None
>>> print ( s )  # affiche None
None
```

Certaines fonctions utilisent cette convention lorsqu'il leur est impossible de retourner un résultat. Ce n'est pas la seule option pour gérer cette impossibilité : il est possible de générer une exception, de retourner une valeur par défaut ou encore de retourner **None**. Il n'y a pas de choix meilleur, il suffit juste de préciser la convention choisie.

Les types numériques

Il existe deux types de nombres en Python, les nombres réels « **float** » et les nombres entiers « **int** ». L'instruction **x = 3** crée une variable de type « **int** » initialisée à **3** tandis que **y = 3.0** crée une variable de type **« float »** initialisée à **3.0**. Le programme suivant permet de vérifier cela en affichant pour les variables x et y, leurs valeurs et leurs types respectifs grâce à la fonction type.

```
>>> x = 3
>>> y = 3.0
>>> print ("x =", x, type(x))
x = 3 <class 'int'>
>>> print ("y =", y, type(y))
y = 3.0 <class 'float'>
```

La liste des opérateurs qui s'appliquent aux nombres réels et entiers.

opérateur	signification	exemple
<< >>	décalage à gauche, à droite	x = 8 « 1 (résultat = 16)
|	opérateur logique ou bit à bit	x = 8|1 (résultat = 9)
&	opérateur logique et bit à bit	x = 11&2 (résultat = 2)
+ -	addition, soustraction	x = y + z
+= -=	addition ou soustraction puis affectation	x += 3
* /	multiplication, division le résultat est de type réel si l'un des nombres est réel	x = y * z
//	division entière	x = y//3
%	reste d'une division entière (modulo)	x = y%3
*= /=	multiplication ou division puis affectation	x *= 3
**	puissance (entière ou non, racine carrée = ** 0.5)	x = y ** 3

Les fonctions **« int »** et « **float** » permettent de convertir un nombre quelconque ou une chaîne de caractères respectivement en un entier (arrondi) et en un nombre réel.

```
>>> x = int (3.5)
>>> y = float (3)
>>> z = int ("3")
>>> print ("x:", type(x), " y:", type(y), " z:", type(z))
x: <class 'int'>  y: <class 'float'>  z: <class 'int'>
```

Il existe un cas pour lequel cet opérateur cache un sens implicite : lorsque la division opère sur deux nombres entiers ainsi que le montre l'exemple suivant.

```
>>> x  =  11
>>> y  =  2
>>> z = x // y
>>> z  #le résultat est 5 et non 5.5 car la division est entière
5
>>> z = x / y
>>> z  #le résultat est 5.5 car c'est une division entre deux réels
5.5
```

Le type integer

Le type integer en Python, est un entier représenté sous forme décimale, binaire, octale ou hexadécimale. Il n'y a pas de limite de représentation pour les entiers longs mise à part la mémoire virtuelle disponible de l'ordinateur.

```
>>> a, b, c = 1, 1, 1
>>> while c < 80 :
...     print(c,":",b,type(b))
...     a, b, c = b, a+b, c+1
...
1 : 1 <class 'int'>
2 : 2 <class 'int'>
...
74 : 2111485077978050 <class 'int'>
75 : 3416454622906707 <class 'int'>
76 : 5527939700884757 <class 'int'>
77 : 8944394323791464 <class 'int'>
78 : 14472334024676221 <class 'int'>
79 : 23416728348467685 <class 'int'>
```

Python est capable de traiter des nombres entiers de taille illimitée. La fonction **`type()`** nous permet de vérifier à chaque itération que le type de la variable **`b`** reste bien en permanence de ce type.

```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
...     print(c, ": ", b)
...     a, b, c = b, a*b, c+1
...
1 :  2
...
12 :
6488003054466075279073683736910497769500103428422804289182764945618623458261160742092 8
13 :
7005669890111832002923764139957621692162454505797269791738369231327175488362123506443467340026896520469610300883250624900843742470237847552
14 :
4545280764562657998563629404824935120516823987072294615140165565565839864222761633581512382578246019698020614153674711609417355051422794795300591700969504226930790382476340558291752968319462245039335017547760330040127583682 56
```

Vous pouvez donc effectuer avec Python des calculs impliquant des valeurs entières comportant un nombre de chiffres significatifs quelconque. Ce nombre n'est limité en effet que par la taille de la mémoire disponible sur l'ordinateur utilisé. Il va de soi cependant que les calculs impliquant de très grands nombres devront être décomposés par l'interpréteur en calculs multiples sur des nombres plus simples, ce qui pourra nécessiter un temps de traitement considérable dans certains cas.

La forme binaire est obtenue avec le préfixe « **`0b`** » ou « **`0B`** ». La fonction « **`bin`** » permet d'afficher la représentation binaire d'un entier.

```
>>> 0b0101101001
361
>>> bin(14)
```

```
'0b1110'
```

La forme octale est obtenue par une séquence de chiffres de 0 à 7, préfixée d'un « **0o** » ou « **0O** ». La fonction « **oct** » permet d'afficher la représentation octale d'un entier.

```
>>> 0o76453
32043
>>> oct(543)
'0o1037'
```

La forme hexadécimale est obtenue par une séquence de chiffres et de lettres de A à F, préfixée par la séquence « **0x** » ou « **0X** ».

```
>>> 0x3ef7b66
66026342
>>> hex(43676)
'0xaa9c'
```

Le type float

La représentation de valeurs à virgule flottante, que l'on notera littéraux réels, permet de décrire des valeurs réelles. Les parties entière et fractionnelle de la valeur réelle sont séparées par le signe « **.** », chaque partie étant composée de chiffres. Si le premier chiffre de la partie entière est 0, le nombre représenté ne sera néanmoins pas considéré comme un octal et restera traité en base 10. Les nombres à virgule flottante utilisés pour représenter des réels sont tous à double précision en Python, soit des nombres codés sur 64 bits.

```
>>> a, b, c = 1., 2., 1
>>> while c < 18:
...     print(c, ": ", b)
...     a, b, c = b, a*b, c+1
...
1 :  2.0
2 :  2.0
3 :  4.0
4 :  8.0
5 :  32.0
6 :  256.0
7 :  8192.0
8 :  2097152.0
9 :  17179869184.0
10 :  3.602879701896397e+16
11 :  6.189700196426902e+26
12 :  2.2300745198530623e+43
13 :  1.3803492693581128e+70
14 :  3.078281734093319e+113
15 :  4.249103942534137e+183
16 :  1.307993905256674e+297
17 :  inf
```

Au dixième terme, Python passe automatiquement à la notation scientifique « **e+n** » signifie en fait : « `fois dix à l'exposant n` ». Après le seizième terme, nous assistons à nouveau à un dépassement de capacité, ainsi les nombres vraiment trop grands sont tout simplement notés « **inf** ».

Ce type autorise les calculs sur de très grands ou très petits nombres, avec un degré de précision constant. Pour qu'une donnée numérique soit considérée par Python comme étant du type « **float** », il suffit qu'elle contienne dans sa formulation un élément tel qu'un point décimal ou un exposant de 10.

Attention à la conversion des valeurs de type « **integer** » en type « **float** », car il est effectué avec une perte de précision.

```
>>> varInt = 99999999999999999999999999999999999999999999999
>>> varFloat = float(varInt)
>>> print(varFloat,varInt)
1e+48 99999999999999999999999999999999999999999999999
>>> varInt == varFloat
False
```

Les nombres complexes

Les nombres complexes sont formés d'un couple de nombres à virgule flottante et subissent donc les mêmes contraintes.

```
>>> varComplex = 1 + 20j
>>> print(varComplex)
(1+20j)
>>> varComplex.real
1.0
>>> varComplex.imag
20.0
>>> varComplex + 10
(11+20j)
>>> print(varComplex)
(1+20j)
```

Les types booléens

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : « **`True`** » ou « **`False`** ». Voici la liste des opérateurs qui s'appliquent aux booléens.

opérateur	signification	exemple
`and or`	et, ou logique	`x = True or False (résultat = True)`
`not`	négation logique	`x = not x`
`< >`	inférieur, supérieur	`x = 5 < 5`
`<= >=`	inférieur ou égal, supérieur ou égal	`x = 5 <= 5`
`== !=`	égal, différent	`x = 5 == 5`

```
>>> x = 4 < 5
>>> print (x) #affiche True
True
>>> print (not x) #affiche False
False
>>> x
True
```

Python accepte l'écriture résumée qui enchaîne des comparaisons : **`3 < x and x < 7`** est équivalent à **`3 < x < 7`**.

Les instructions conditionnelles

La plus simple de ces instructions conditionnelles est l'instruction if avec la syntaxe suivante :

```
if condition :
     suite
[elif condition :
     suite , ...]
[else :
     suite]
```

```
>>> varInt = 150
>>> if (varInt > 100):
...     print("varInt dépasse la centaine")
...
```

Frappez encore une fois « **Enter** » et le programme s'exécute, et vous obtenez :

```
varInt dépasse la centaine
>>> varInt = 150
>>> if (varInt > 100):
...     print("varInt dépasse la centaine")
... else:
...     print("varInt ne dépasse pas cent")
...
varInt dépasse la centaine
>>> varInt = 10
>>> if (varInt > 100):
...     print("varInt dépasse la centaine")
... else:
...     print("varInt ne dépasse pas cent")
...
varInt ne dépasse pas cent
```

On peut faire mieux encore en utilisant aussi l'instruction « **elif** ».

```
>>> varInt = 0
>>> if varInt > 0 :
...     print("varInt est positif")
... elif varInt < 0 :
...     print("varInt est négatif")
... else:
...     print("varInt est égal à zéro")
...
varInt est égal à zéro
>>> varInt = -1
>>> if varInt > 0 :
...     print("varInt est positif")
... elif varInt < 0 :
...     print("varInt est négatif")
... else:
...     print("varInt est égal à zéro")
```

```
...
varInt est négatif
```

Les opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les opérateurs de comparaison suivants :

```
x == y          # x est égal à y
x != y          # x est différent de y
x > y           # x est plus grand que y
x < y           # x est plus petit que y
x >= y          # x est plus grand que, ou égal à y
x <= y          # x est plus petit que, ou égal à y
and    or   not
```

```
>>> varInt = 4
>>> if (varInt % 2 == 0):
...     print("varInt est pair")
...     print("parce que le reste de sa division par 2 est nul")
... else:
...     print("varInt est impair")
...
varInt est pair
parce que le reste de sa division par 2 est nul
```

Passer, instruction pass

Dans certains cas, aucune instruction ne doit être exécutée même si un test est validé. En Python, le corps d'un test ne peut être vide, il faut utiliser l'instruction « **pass** ». Lorsque celle-ci est manquante, Python affiche un message d'erreur.

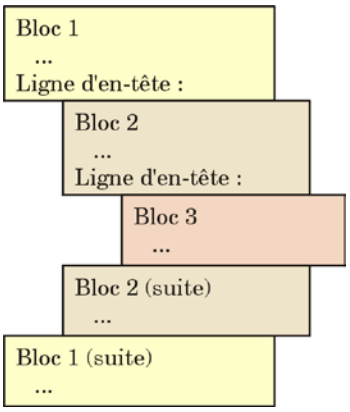
```
>>> signe = 0
>>> x = 0
>>> if x < 0 :
   signe = -1
elif x == 0:
   pass #signe est déjà égal à 0
else :
   signe = 1
>>> print(signe)
0
```

Les blocs d'instructions

La construction que vous avez utilisée avec l'instruction **« `if` »** est votre premier exemple de blocs d'instructions. Sous Python, les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête.

```
Ligne d'en-tête:
    première instruction du bloc
    ...
    dernière instruction du bloc
```

Ces instructions indentées constituent ce que nous appellerons désormais un bloc d'instructions. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête.



Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (**`if, elif, else, while, def`**, etc.) se terminant par un double point.

Les blocs sont délimités par l'indentation : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière. Le nombre d'espaces à utiliser pour l'indentation est quelconque.

Notez que le code du bloc le plus externe ne peut pas lui-même être écarté de la marge de gauche, il n'est imbriqué dans rien.

Les espaces et les commentaires sont normalement ignorés à part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés. Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse « `#` » et s'étendent jusqu'à la fin de la ligne courante.

Attention

Vous pouvez aussi indenter à l'aide de tabulations, mais alors vous devrez faire très attention à ne pas utiliser tantôt des espaces, tantôt des tabulations pour indenter les lignes d'un même bloc. En effet, même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents. Il peut en résulter des erreurs difficiles à déboguer.

L'instruction while

L'instruction tant que commence par évaluer la validité de la condition et si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant le corps de la boucle. Voici la syntaxe de boucle tant que :

```
while expression :
    suite
[else :
    suite]
```

```
>>> varInt = 0
>>> while (varInt < 7):          # ( n'oubliez pas le double point !)
...     varInt = varInt + 1      # ( n'oubliez pas l'indentation !)
...     print("La valeur de varInt est : ",varInt)
...
La valeur de varInt est :  1
La valeur de varInt est :  2
La valeur de varInt est :  3
La valeur de varInt est :  4
La valeur de varInt est :  5
La valeur de varInt est :  6
La valeur de varInt est :  7
```

La variable évaluée dans la condition doit exister au préalable. Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté. Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment.

```
>>> i = 0
>>> while i < 2:
...     varInt = int(input("Entrez une valeur pour varInt :"))
...     while varInt < 8:
...         varInt = varInt +1
...         print(varInt , varInt**2 , varInt**3)
...     else:
...         print('Valeur de varInt >= 8')
...     i = i + 1
...
Entrez une valeur pour varInt :8
Valeur de varInt >= 8
Entrez une valeur pour varInt :4
5 25 125
6 36 216
7 49 343
8 64 512
Valeur de varInt >= 8
```

Le petit programme ci-dessous permet d'afficher les dix premiers termes d'une suite de Fibonacci. Il s'agit d'une suite de nombres dont chaque terme est égal à la somme des deux termes qui le précèdent.

```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
...     print(b, end =" ")
...     a, b, c = b, a+b, c+1
```

```
... else :
...     print(" ")
...
1 2 3 5 8 13 21 34 55 89
```

La fonction **print()** ajoute en effet un caractère de saut à la ligne à toute valeur qu'on lui demande d'afficher. L'argument **« end =" " »** signifie que vous souhaitez remplacer le saut à la ligne par un simple espace. Si vous supprimez cet argument, les nombres seront affichés les uns en-dessous des autres.

Note

La variable évaluée dans la condition doit exister au préalable.

Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.

Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner).

L'instruction break pour sortir d'une boucle

Dans une boucle « **while** » on peut interrompre le bouclage indépendamment de la condition de continuation en faisant appel à l'instruction break.

```
while <condition 1> :
    --- instructions diverses ---
    if <condition 2> :
         break
    --- instructions diverses ---
```

```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
    print(c, ": ", b)
    a, b, c = b, a*b, c+1
    if b > 99999999999 : break


1 :  2
2 :  6
3 :  12
4 :  72
5 :  864
6 :  62208
7 :  53747712
```

Les listes

Les listes sont des collections ordonnées d'objets. Comme les chaînes de caractères, les listes font partie d'un type général que l'on appelle séquences sous Python. Comme les caractères dans une chaîne, les objets placés dans une liste sont rendus accessibles par l'intermédiaire d'un index.

Dans une liste on peut combiner des données de n'importe quel type, y compris des listes, des **`dictionnaires`** et des **`tuples`**. Pour accéder aux éléments d'une liste, on utilise les mêmes méthodes (index, découpage en tranches) que pour accéder aux caractères d'une chaîne.

```
>>> nombres = [5, 38, 10, 25]
>>> mots = ["jambon", "fromage", "confiture", "chocolat"]
>>> stuff = [5000, "Brigitte", 3.1416, ["Albert", "René", 1947]]
>>> print(nombres[2])
10
>>> print(nombres[1:3])
[38, 10]
>>> print(nombres[2:3])
[10]
>>> print(nombres[2:])
[10, 25]
>>> print(nombres[:2])
[5, 38]
>>> print(nombres[-1])
25
>>> print(nombres[-2])
10
```

Une tranche découpée dans une liste est toujours elle-même une liste, même s'il s'agit d'une tranche qui ne contient qu'un seul élément, comme dans notre troisième exemple, alors qu'un élément isolé peut contenir n'importe quel type de donnée.

Les listes sont modifiables

Contrairement aux chaînes de caractères, les listes sont des séquences modifiables. Cela nous permettra de construire plus tard des listes de grande taille, morceau par morceau, d'une manière dynamique.

```
>>> nombres[0] = 17
>>> nombres
[17, 38, 10, 25]
>>> stuff[3][1] = "Isabelle"
>>> stuff
[5000, 'Brigitte', 3.1416, ['Albert', 'Isabelle', 1947]]
>>> nombres[0] = nombres[2:]
>>> nombres
[[10, 25], 38, 10, 25]
```

Les listes sont des objets

Les listes sont des objets à part entière, et vous pouvez donc leur appliquer un certain nombre de méthodes particulièrement efficaces.

x in l	vrai si x est un des éléments de l
x not in l	réciproque de la ligne précédente
l + t	concaténation de l et t
l * n	concatène n copies de l les unes à la suite des autres
len(l)	nombre d'éléments de l
min(l)	plus petit élément de l, résultat difficile à prévoir lorsque les types des éléments sont différents
max(l)	plus grand élément de l
sum(l)	retourne la somme de tous les éléments
del l[i:j]	supprime les éléments d'indices entre i et j exclu. Cette instruction est équivalente à **l[i:j] = [ ]**.
list(x)	convertit x en une liste quand cela est possible
l.count(x)	Retourne le nombre d'occurrences de l'élément x.
l.index(x)	Retourne l'indice de la première occurrence de l'élément x dans la liste l.
l.append(x)	Ajoute l'élément x à la fin de la liste l. Si x est une liste, cette fonction ajoute la liste x en tant qu'élément, au final, la liste l ne contiendra qu'un élément de plus.
l.extend(k)	Ajoute tous les éléments de la liste k à la liste l. La liste l aura autant d'éléments supplémentaires qu'il y en a dans la liste k.
l.insert(i,x)	Insère l'élément x à la position i dans la liste l.
l.remove(x)	Supprime la première occurrence de l'élément x dans la liste l. S'il n'y a aucune occurrence de x, cette méthode déclenche une exception.
l.pop([i])	Retourne l'élément l[i] et le supprime de la liste. Le paramètre i est facultatif, s'il n'est pas précisé, c'est le dernier élément qui est retourné puis supprimé de la liste.
l.reverse(x)	Retourne la liste, le premier et dernier élément échange leurs places, le second et l'avant dernier, et ainsi de suite.
l.sort([key=None, reverse=False])	Cette fonction trie la liste par ordre croissant. Le paramètre key est facultatif, il permet de préciser la fonction qui précise clé de comparaison qui doit être utilisée

```
>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()
>>> nombres
[10, 17, 25, 38, 72]
>>> nombres.append(12)
>>> nombres
[10, 17, 25, 38, 72, 12]
>>> nombres.reverse()
>>> nombres
[12, 72, 38, 25, 17, 10]
>>> nombres.index(17)
4
```

```
>>> nombres.remove(38)
>>> nombres
[12, 72, 25, 17, 10]
>>> del nombres[2]
>>> nombres
[12, 72, 17, 10]
>>> del nombres[1:3]
>>> nombres
[12, 10]
```

Techniques de slicing avancé pour modifier une liste

Il est possible d'utiliser à la place des **« del »** ou **« append »** pour ajouter ou supprimer des éléments dans une liste l'opérateur **« [ ] »**. L'utilisation de cet opérateur est un peu plus délicate que celle d'instructions ou de méthodes dédiées, mais elle permet davantage de souplesse.

Insertion d'un ou plusieurs éléments n'importe où dans une liste

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] =["miel"]
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat']
>>> mots[5:5] =['saucisson', 'ketchup']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat', 'saucisson',
'ketchup']
>>> mots[1:2]
['fromage']
```

Pour utiliser cette technique, vous devez prendre en compte les particularités suivantes :

- Si vous utilisez l'opérateur **« [ ] »** à la gauche du signe égale pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une tranche dans la liste cible, c'est-à-dire deux index réunis par le symbole **« : »**, et non un élément isolé dans cette liste.
- L'élément que vous fournissez à la droite du signe égale doit lui-même être une liste. Si vous n'insérez qu'un seul élément, il vous faut donc le présenter entre crochets pour le transformer d'abord en une liste d'un seul élément.

La même démarche pour les suppressions et le remplacement d'éléments.

```
>>> mots[2:5] = []  # [] désigne une liste vide
>>> mots
['jambon', 'fromage', 'saucisson', 'ketchup']
>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade', 'ketchup']
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
```

Création d'une liste de nombres

Si vous devez manipuler des séquences de nombres, vous pouvez les créer très aisément à l'aide de la fonction intégrée **« range »**. Elle renvoie une séquence

d'entiers que vous pouvez utiliser directement, ou convertir en une liste avec la fonction **« list »**, ou convertir en tuple avec la fonction **« tuple »**.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,13))
[5, 6, 7, 8, 9, 10, 11, 12]
>>> list(range(3,16,3))
[3, 6, 9, 12, 15]
>>> list(range(10, -10, -3))
[10, 7, 4, 1, -2, -5, -8]
```

La fonction **« range »** génère par défaut une séquence de nombres entiers de valeurs croissantes, et différant d'une unité. Si vous appelez **« range »** avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, mais en commençant à partir de zéro. Notez bien que l'argument fourni n'est jamais dans la liste générée. On peut aussi utiliser **« range »** avec deux, ou même trois arguments séparés par des virgules, que l'on pourrait intituler **FROM**, **TO** et **STEP**

Toute boucle **« for »** peut s'appliquer sur un objet muni d'un itérateur tels que les chaînes de caractères, tuples, les listes, les dictionnaires, les ensembles.

```
>>> prov = ['La','raison','du','plus','fort',\
        'est','toujours','la','meilleure']
>>> for mot in prov:
        print(mot, end =' ')

La raison du plus fort est toujours la meilleure
```

Si vous voulez parcourir une gamme d'entiers, la fonction range() s'impose.

```
>>> for n in range(10, 18, 3):
        print(n, n**2, n**3)

10 100 1000
13 169 2197
16 256 4096
```

Il est très pratique de combiner les fonctions **« range »** et **« len »** pour obtenir automatiquement tous les indices d'une séquence (liste ou chaîne).

```
>>> fable = ['Maître','Corbeau','sur','un','arbre','perché']
>>> for index in range(len(fable)):
        print(index, fable[index])

0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

Une conséquence importante du typage dynamique est que le type de la variable utilisée avec l'instruction **« for »** est redéfini continuellement au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de for sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture.

```
>>> divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
>>> for item in divers :
    print(item, type(item))
```

```
3 <class 'int'>
17.25 <class 'float'>
[5, 'Jean'] <class 'list'>
Linux is not Windoze <class 'str'>
```

Les opérations sur des listes

On peut appliquer aux listes les opérateurs **« + »** (concaténation) et **« * »** (multiplication). L'opérateur **« * »** est particulièrement utile pour créer une liste de n éléments identiques.

```
>>> fruits = ['orange','citron']
>>> legumes = ['poireau','oignon','tomate']
>>> fruits + legumes
['orange', 'citron', 'poireau', 'oignon', 'tomate']
>>> fruits * 3
['orange', 'citron', 'orange', 'citron', 'orange', 'citron']
>>> sept_zeros = [0]*7
>>> sept_zeros
[0, 0, 0, 0, 0, 0, 0]
```

Supposons par exemple que vous voulez créer une liste B qui contienne le même nombre d'éléments qu'une autre liste A.

```
>>> deuxieme = [10]*len(sept_zeros)
>>> deuxieme
[10, 10, 10, 10, 10, 10, 10]
```

Le test d'appartenance

Vous pouvez aisément déterminer si un élément fait partie d'une liste à l'aide de l'instruction **« in »** (cette instruction puissante peut être utilisée avec toutes les séquences).

```
>>> v = 'tomate'
>>> if v in legumes:
        print('OK')

OK
```

La copie d'une liste

Attention une simple affectation ne créez pas une véritable copie d'une liste que vous souhaitez recopier dans une nouvelle variable. À la suite de cette instruction, il n'existe toujours qu'une seule liste dans la mémoire de l'ordinateur. Ce que vous avez créé est seulement une nouvelle référence vers cette liste.

```
>>> fable = ['Je','plie','mais','ne','romps','point']
>>> phrase = fable
>>> fable[4] ='casse'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Si la variable phrase contenait une véritable copie de la liste, cette copie serait indépendante de l'original et ne devrait donc pas pouvoir être modifiée par une instruction telle que celle de la troisième ligne, qui s'applique à la variable fable.

En fait, les noms fable et phrase désignent tous deux un seul et même objet en mémoire. Ainsi l'objet phrase est une référence de l'objet fable.

```
>>> phrase2 = phrase[0:len(phrase)]
>>> phrase[4] ='romps'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'romps', 'point']
>>> phrase2
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Python vous autorise à « étendre » une longue instruction sur plusieurs lignes, si vous continuez à encoder quelque chose qui est délimité par une paire de parenthèses, de crochets ou d'accolades. Vous pouvez traiter ainsi des expressions parenthèses, ou encore la définition de longues listes, de grands tuples ou de grands. Le niveau d'indentation n'a pas d'importance : l'interpréteur détecte la fin de l'instruction là où la paire syntaxique est refermée. Cette fonctionnalité vous permet d'améliorer la lisibilité de vos programmes.

```
>>> couleurs = ['noir', 'brun', 'rouge',
                'orange', 'jaune', 'vert',
                'bleu', 'violet', 'gris', 'blanc']
```

Les nombres aléatoires – histogrammes

Le module **« random »**, Python propose une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques.

Les fonctions les plus couramment utilisées sont :

- **choice(sequence)** : renvoie un élément au hasard de la séquence fournie.
- **randint(a, b)** : renvoie un nombre entier compris entre a et b.
- randrange(a,b,c) : renvoie un nombre entier tiré au hasard d'une série limitée d'entiers entre a et b, séparés les uns des autres par un certain intervalle, défini par c.
- **random()** : renvoie un réel compris entre 0.0 et 1.0.
- **sample(sequence, k)** : renvoie k éléments uniques de la séquence.
- **seed([salt])** : initialise le générateur aléatoire.
- **shuffle(sequence[, random])** : mélange l'ordre des éléments de la séquence (dans l'objet lui-même). Si random est fourni, c'est un callable qui renvoie un réel entre 0.0 et 1.0. **« random »** est pris par défaut.
- **uniform(a, b)** : renvoie un réel compris entre a et b.

```
>>> from random import *
>>> def list_aleat(n):
        s = [0]*n
        for i in range(n):
            s[i] =random()
        return s

>>> list_aleat(3)
[0.6735559443377152, 0.09987607185190805, 0.6063188589461471]
>>> list_aleat(3)
[0.5182167354579681, 0.21973841027737828, 0.36650995653460494]

>>> for i in range(15):
```

```
        print(randrange(3, 13, 3), end =' ')

6 12 3 6 9 6 6 3 6 6 12 3 12 6 12
```

Vous pouvez constater que nous avons pris le parti de construire d'abord une liste de zéros de taille n, et ensuite de remplacer les zéros par des nombres aléatoires.

```
>>> import random
>>> good_work = ['Excellent travail!',
        'Très bonne analyse',
        'Les résultats sont là !']
>>> bad_work = ["J'ai gratté la copie pour mettre des points",
        'Vous filez un mauvais coton',
        'Que se passe-t-il ?']
>>> ok_work = ['Bonne première partie mais soignez la présentation',
        'Petites erreurs, dommage !',
        'Des progrès']

>>> def auto_corrector(student):
        note = random.randint(1, 20)
        if note < 8:
            appreciation = random.choice(bad_work)
        elif note < 14:
            appreciation = random.choice(ok_work)
        else:
            appreciation = random.choice(good_work)
        return '%s: %s, %s' %(student,
               note, appreciation)

>>> students = ['Bernard', 'Robert', 'René', 'Gaston',
    'Églantine', 'Aimé', 'Robertine']

>>> for student in students :
        print(auto_corrector(student))

Bernard: 16, Très bonne analyse
Robert: 2, Que se passe-t-il ?
René: 11, Petites erreurs, dommage !
Gaston: 12, Petites erreurs, dommage !
Églantine: 3, J'ai gratté la copie pour mettre des points
Aimé: 18, Excellent travail!
Robertine: 2, J'ai gratté la copie pour mettre des points
```

Atelier 5 - Exercice 2 - Exercice 3

Les tuples

Python propose un type de données appelé **« tuple »**, qui est assez semblable à une liste mais qui, comme les chaînes, n'est pas modifiable.

Du point de vue de la syntaxe, un **« tuple »** est une collection d'éléments séparés par des virgules comprise entre parenthèses. Ce terme n'est pas un mot anglais ordinaire : il s'agit d'un néologisme informatique.

```
>>> tup = (5000, "Brigitte",
        3.1416, ["Albert", "René", 1947])
>>> print(tup)
(5000, 'Brigitte', 3.1416, ['Albert', 'René', 1947])
>>> tup2 = 5000, "Brigitte", \
        3.1416, ["Albert", "René", 1947]
>>> tup2
(5000, 'Brigitte', 3.1416, ['Albert', 'René', 1947])
>>> tup2[0]
5000
>>> tup2[0] = 1
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    tup2[0] = 1
TypeError: 'tuple' object does not support item assignment
```

Bien que cela ne soit pas nécessaire, il est vivement conseillé de mettre le **« tuple »** en évidence en l'enfermant dans une paire de parenthèses, comme la fonction **« print »** de Python le fait elle-même. Il s'agit simplement d'améliorer la lisibilité du code, mais vous savez que c'est important.

x in s	vrai si x est un des éléments de s
x not in s	réciproque de la ligne précédente
s + t	concaténation de s et t
s * n	concatène n copies de s les unes à la suite des autres
s[i]	retourne le $i^{ème}$ élément de s
s[i:j]	retourne un **« tuple »** contenant une copie des éléments de s d'indices i à j exclu
s[i:j:k]	retourne un **« tuple »** contenant une copie des éléments de s dont les indices sont compris entre i et j exclu, ces indices sont espacés de k : i, $i + k$, $i + 2k$, $i + 3k$, ...
len(s)	nombre d'éléments de s
min(s)	plus petit élément de s, résultat difficile à prévoir lorsque les types des éléments sont différents
max(s)	plus grand élément de s
sum(s)	retourne la somme de tous les éléments

Les **« tuples »** composés d'un seul élément ont une écriture un peu particulière puisqu'il est nécessaire d'ajouter une virgule après l'élément, sans quoi l'analyseur syntaxique de Python ne le considérera pas comme un **« tuples »** mais comme l'élément lui-même, et supprimera les parenthèses qu'il analyserait comme superflues.

```
>>> tuple()
```

```
()
>>> tuple('a')
('a',)
>>> color_and_note = ('rouge', 12, 'vert', 14, 'bleu', 9)
>>> colors = color_and_note[::2]
>>> print(colors)
('rouge', 'vert', 'bleu')
>>> notes = color_and_note[1::2]
>>> print(notes)
(12, 14, 9)
>>> color_and_note = color_and_note + ('violet',)
>>> print(color_and_note)
('rouge', 12, 'vert', 14, 'bleu', 9, 'violet')
>>> ('violet')
'violet'
>>> ('violet',)
('violet',)
```

Les **« tuples »** sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les **« tuples »** occupent moins de place en mémoire, et peuvent être traités plus rapidement par l'interpréteur.

Les dictionnaires

Les types de données composites que nous avons abordés jusqu'à présent **« chaînes »**, **« listes »** et **« tuples »** étaient tous des séquences, c'est-à-dire des suites ordonnées d'éléments. Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index (un nombre entier), mais à la condition expresse de connaître son emplacement.

Les **« dictionnaires »** que nous découvrons ici constituent un autre type composite. Ils ressemblent aux listes dans une certaine mesure, ils sont modifiables comme elles, mais ce ne sont pas des séquences. Les éléments que nous allons y enregistrer ne seront pas disposés dans un ordre immuable. En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une clé, laquelle pourra être alphabétique, numérique, ou même d'un type composite sous certaines conditions.

Comme dans une liste, les éléments mémorisés dans un **« dictionnaire »** peuvent être de n'importe quel type. Ce peuvent être des valeurs numériques, des **« chaînes »**, des **« listes »**, des **« tuples »**, des **« dictionnaires »**, et même aussi des **« fonctions »**, des **« classes »** ou des **« instances »**.

La création d'un dictionnaire

Puisque le type **« dictionnaire »** est un type modifiable, nous pouvons commencer par créer un **« dictionnaire »** vide, puis le remplir petit à petit. Du point de vue de la syntaxe, on reconnaît un **« dictionnaire »** au fait que ses éléments sont enfermés dans une paire d'accolades **« { } »**.

```
>>> dico = {}
>>> dico['computer'] = 'ordinateur'
>>> dico['mouse'] ='souris'
>>> dico['keyboard'] ='clavier'
>>> print(dico)
{'computer': 'ordinateur', 'mouse': 'souris', 'keyboard': 'clavier'}
>>> print(dico['mouse'])
souris
```

Un **« dictionnaire »** apparaît dans la syntaxe Python sous la forme d'une série d'éléments séparés par des virgules, le tout étant enfermé entre deux accolades. Chacun de ces éléments est lui-même constitué d'une paire d'objets : un index et une valeur, séparés par un double point. Dans un **« dictionnaire »**, les index s'appellent des clés, et les éléments peuvent donc s'appeler des paires clé-valeur.

Remarquez aussi que contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière (telle que **« append »**) pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur.

Les opérations sur les dictionnaires

Tout comme les listes, les objets de type dictionnaire proposent un certain nombre de méthodes.

Nom	Description
`clear()`	Supprime tous les éléments du dictionnaire.
`copy()`	Renvoie une copie par références du dictionnaire. Lire la remarque sur les copies un peu plus bas.
`items()`	Renvoie sous la forme d'une liste de **« `tuples` »**, es couples (clé, valeur) du dictionnaire. Les objets représentant les valeurs sont es copies complètes et non des références.
`keys()`	Renvoie sous la forme d'une liste l'ensemble des clés du dictionnaire. L'ordre de renvoi des éléments n'a aucune signification ni constance et peut varier à chaque modification du dictionnaire.
`values()`	Renvoie sous forme de liste les valeurs du dictionnaire. L'ordre de renvoi n'a ici non plus aucune signification mais sera le même que pour **« `keys` »** si la liste n'est pas modifiée entre-temps, ce qui permet de faire des manipulations avec les deux listes.
`get(cle,default)`	Renvoie la valeur identifiée par la clé. Si la clé n'existe pas, renvoie la valeur default fournie. Si aucune valeur n'est fournie, renvoie **« `None` »**.
`pop(cle,default)`	Renvoie la valeur identifiée par la clé et retire l'élément du dictionnaire. Si la clé n'existe pas, pop se contente de renvoyer la valeur default. Si le paramètre default n'est pas fourni, une erreur est levée.
`popitem()`	Renvoie le premier couple (clé, valeur) du dictionnaire et le retire. Si le dictionnaire est vide, une erreur est renvoyée. L'ordre de retrait des éléments correspond à l'ordre des clés retournées par **« `keys` »** si la liste n'est pas modifiée entre-temps.
`update(dic,**dic)`	Update permet de mettre à jour le dictionnaire avec les éléments du dictionnaire dic. Pour les clés existantes dans la liste, les valeurs sont mises à jour, sinon créées. Le deuxième argument est aussi utilisé pour mettre à jour les valeurs.
`setdefault(cle, default)`	Fonctionne comme **« `get` »** mais si clé n'existe pas et default est fourni, le couple (cle, default) est ajouté à la liste.
`fromkeys(seq, default)`	Génère un nouveau dictionnaire et y ajoute les clés fournies dans la séquence seq. La valeur associée à ces clés est default si le paramètre est fourni, **« `None` »** le cas échéant.

Voici quelque exemples de ces syntaxes.

```
>>> invent ={"oranges":274, "poires":137, "bananes":312}
>>> for clef in invent:
        print(clef)

oranges
poires
bananes
>>> dico1 = {'a':1,'b':2}
>>> dico1.clear()
>>> dico1
```

```
{}
>>> dico = {'1': 'r', '2': [1,2]}
>>> dico2 = dico.copy()
>>> dico2
{'1': 'r', '2': [1, 2]}
>>> dico['2'].append('E')
>>> dico2['2']
[1, 2, 'E']
>>> dico = {'a': 1, 'b': 2}
>>> 'a' in dico
True
>>> 'c' not in dico
True
>>> a = {'a': 1, 'b': 1}
>>> a.items()
dict_items([('a', 1), ('b', 1)])
>>> a = {(1, 3): 3, 'Q': 4}
>>> a.keys()
dict_keys([(1, 3), 'Q'])
>>> a = {(1, 3): 3, 'Q': 4}
>>> a.values()
dict_values([3, 4])
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> for i,j,k in l.items(),l.keys(),l.values() :
    print(i,j,k)

 (1, 'a') (2, 'b') (3, 'c')
1 2 3
a b c
>>> l.get(1)
'a'
>>> l.get(13)

>>> l.get(13, 7)
7
>>> l
{1: 'a', 2: 'b', 3: 'c'}
>>> l.pop(1)
'a'
>>> l
{2: 'b', 3: 'c'}
>>> l.pop(13, 6)
6
>>> l
{2: 'b', 3: 'c'}
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l.popitem()
(3, 'c')
>>> l.popitem()
(2, 'b')
>>> l.popitem()
(1, 'a')
>>> l
{}
```

```
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l2 = {3: 'ccc', 4: 'd'}
>>> l.update(l2)
>>> l
{1: 'a', 2: 'b', 3: 'ccc', 4: 'd'}
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l.setdefault(4, 'd')
'd'
>>> l
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> l = {}

>>> l.fromkeys([1, 2, 3], 0)
{1: 0, 2: 0, 3: 0}
```

Les clés peuvent être de n'importe quel type de données non modifiables : des entiers, des réels, des chaînes de caractères, et même des tuples.

```
>>> arb = {}
>>> arb[(1,2)] = 'Peuplier'
>>> arb[(3,4)] = 'Platane'
>>> arb[6,5] = 'Palmier'
>>> arb[5,1] = 'Cycas'
>>> arb[7,3] = 'Sapin'
>>> print(arb)
{(1, 2): 'Peuplier', (3, 4): 'Platane', (6, 5): 'Palmier', (5, 1):
'Cycas', (7, 3): 'Sapin'}
>>> print(arb[(6,5)])
Palmier
>>> print(arb[1,2])
Peuplier
>>> print(arb[2,1])
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    print(arb[2,1])
KeyError: (2, 1)
>>> arb.get((2,1), 'néant')
'néant'
>>> print(arb[1:3])
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    print(arb[1:3])
TypeError: unhashable type: 'slice'
```

Les chaînes de caractères

Le terme "chaîne de caractères" ou string en anglais signifie une suite finie de caractères, autrement dit, du texte. Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables.

```
>>> t = "string = texte"
>>> print (type (t), t)
<class 'str'> string = texte
>>> t = 'string = texte, initialisation avec apostrophes'
>>> print (type (t), t)
<class 'str'> string = texte, initialisation avec apostrophes
>>> t = "morceau 1" \
       "morceau 2"
>>> #second morceau ajouté au premier par l'ajout du symbole \,
>>> #il ne doit rien y avoir après le symbole \,
>>> #pas d'espace ni de commentaire
>>> print (t)
morceau 1morceau 2
>>> t = """première ligne
seconde ligne"""
>>> # chaîne de caractères qui s'étend sur deux lignes
>>> print(t)
première ligne
seconde ligne
>>> t = '''première ligne
          seconde ligne'''
>>> t
'première ligne\n          seconde ligne'
>>> print(t)
première ligne
          seconde ligne
```

Python offre la possibilité de couper le texte en deux chaînes de caractères recollées à l'aide du symbole **** à condition que ce symbole soit le dernier de la ligne sur laquelle il apparaît. De même, lorsque le texte contient plusieurs lignes, il suffit de les encadrer entre deux symboles **« """ »** ou **« ''' »** pour que l'interpréteur Python considère l'ensemble comme une chaîne de caractères et non comme une série d'instructions.

opérateur	signification
\"	guillemet
\'	apostrophe
\n	passage à la ligne
\f	saut de ligne
****	insertion du symbole \
\%	pourcentage, ce symbole est aussi un caractère spécial
\t	Tabulation
\v	tabulation verticale
\r	retour à la ligne, peu usité, il a surtout son importance lorsqu'on passe d'un système Windows à Linux car Windows l'ajoute automatiquement à tous ses fichiers textes

Il peut être fastidieux d'avoir à doubler tous les symboles **** d'un nom de fichier. Il est plus simple dans ce cas de préfixer la chaîne de caractères par « **r** » de façon à éviter que l'utilisation du symbole **** ne désigne un caractère spécial.

```
>>> s1 = "C:\\Users\\exemple.txt"
>>> s2 =r"C:\Users\exemple.txt"
>>> s1 == s2
True
```

Triple quotes

Pour insérer plus aisément des caractères spéciaux ou « exotiques » dans une chaîne, sans faire usage de l'antislash, ou pour faire accepter l'antislash lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de triples guillemets ou de triples apostrophes :

```
>>> a1 = """
... Exemple de texte préformaté, c'est-à-dire
... dont les indentations et les
... caractères spéciaux \ ' " sont
... conservés sans
... autre forme de procès."""
>>> print(a1)
... Exemple de texte préformaté, c'est-à-dire
... dont les indentations et les
... caractères spéciaux \ ' " sont
... conservés sans
... autre forme de procès.
```

Manipulation d'une chaîne

Une chaîne de caractères est semblable à un tableau et certains opérateurs qui s'appliquent aux tableaux s'appliquent également aux chaînes de caractères. La fonction « **str** » permet de convertir un nombre, un tableau, un objet en chaîne de caractères afin de pouvoir l'afficher. La fonction **« len »** retourne la longueur de la chaîne de caractères.

```
>>> x = 5.567
>>> s = str(x)
>>> print(type(s),s)
<class 'str'> 5.567
>>> print(len(s))
5
```

Opérations applicables aux chaînes de caractères.

opérateur	signification	exemple
+	concaténation de chaînes de caractères	`t="abc"+"def"`
+=	concaténation puis affectation	`t+="abc"`
in, not in	une chaîne en contient-elle une autre ?	`"ed" in "med"`
*****	répétition d'une chaîne de caractères	`t="abc"*4`
[n]	obtention du $n^{ième}$ caractère, le premier caractère a pour indice 0	`t="abc"` `print(t[0])`
[i : j]	obtention des caractères compris entre les indices i et j − 1 inclus, le premier caractère a pour indice 0	`t="abc"` `print t[0:2]`

Il existe d'autres fonctions qui permettent de manipuler les chaînes de caractères.

opérateur	signification
count(sub[,start[,end]])	Retourne le nombre d'occurrences de la chaîne de caractères « **sub** », les paramètres par défaut « **start** » et « **end** » permettent de réduire la recherche entre les caractères d'indice « **start** » et « **end** » exclu.
find(sub[,start[,end]])	Recherche une chaîne de caractères « **sub** », les paramètres par défaut ont la même signification que ceux de la fonction count.
index(car)	Retrouve l'indice de la première occurrence du caractère « **car** » dans la chaîne
isalpha()	Retourne True si tous les caractères sont des lettres, False sinon.
isdigit()	Retourne True si tous les caractères sont des chiffres, False sinon.
replace(old,new[,count])	Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne « **old** » par « **new** ». Si le paramètre optionnel « **count** » est renseigné, alors seules les premières occurrences seront remplacées.
split(sep=None,maxsplit=-1)	Découpe la chaîne de caractères en se servant de la chaîne « **sep** » comme délimiteur. Si le paramètre « **maxsplit** » est renseigné, au plus « **maxsplit** » coupures seront effectuées.
strip([s])	Supprime les espaces au début et en fin de chaîne. Si le paramètre « **s** » est renseigné, tous les caractères qui font partie de « **s** » au début et en fin de chaîne sont supprimes.
upper()	Remplace les minuscules par des majuscules

lower()	Remplace les majuscules par des minuscules.
join(words)	Fait la somme d'un tableau de chaînes de caractères (une liste ou un T-uple). La chaîne de caractères sert de séparateur qui doit être ajouté entre chaque élément du tableau words.
title	Convertit en majuscule l'initiale de chaque mot (suivant l'usage des titres anglais)
capitalize	Convertit en majuscule seulement la première lettre de la chaîne
swapcase	Convertit toutes les majuscules en minuscules, et vice-versa

Python considère qu'une chaîne de caractères est un objet de la catégorie des séquences, lesquelles sont des collections ordonnées d'éléments. Cela signifie simplement que les caractères d'une chaîne sont toujours disposés dans un certain ordre. Par conséquent, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un index.

```
>>> phrase1 = 'les oeufs durs.'
>>> phrase2 = '"Oui", répondit-il,'
>>> phrase3 = "j'aime bien"
>>> print(phrase2, phrase3, phrase1)
"Oui", répondit-il, j'aime bien les oeufs durs.
>>> phrase = phrase2 + phrase3 + ' ' + phrase1
>>> print(phrase)
"Oui", répondit-il,j'aime bien les oeufs durs.
>>> phrase.count('o')
2
>>> phrase.find('oeuf')
35
>>> phrase[35:40]
'oeufs'
>>> phrase.replace('\"','')
"Oui, répondit-il,j'aime bien les oeufs durs."
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> phrase.split(',', maxsplit=1)
['"Oui"', " répondit-il,j'aime bien les oeufs durs."]
>>> phrase.split(',')
['"Oui"', ' répondit-il', "j'aime bien les oeufs durs."]
>>> s = '#....... Section 3.2.1 Issue #32 .......'
>>> s.strip('.#! ')
'Section 3.2.1 Issue #32'
```

Attention les chaînes constituent un type de données non-modifiables il est impossible de changer les éléments individuels d'une chaîne.

```
>>> s="abcd"
>>> s[1]='c'
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    s[1]='c'
TypeError: 'str' object does not support item assignment
```

À toutes fins utiles, rappelons également ici que l'on peut aussi appliquer aux chaînes un certain nombre de fonctions intégrées dans le langage :

len(ch) renvoie la longueur de la chaîne ch, ou en d'autres termes, son nombre de caractères.

`float(ch)` convertit la chaîne ch en un nombre réel **« `float` »** (bien entendu, cela ne pourra fonctionner que si la chaîne représente bien un nombre, réel ou entier)

`int(ch)` convertit la chaîne ch en un nombre entier (avec des restrictions similaires)

`str(obj)` convertit (ou représente) l'objet **`obj`** en une chaîne de caractères. **`obj`** peut être une donnée d'à peu près n'importe quel type :

Formatage d'une chaîne

Python offre une manière plus concise de former une chaîne de caractères à l'aide de plusieurs types d'informations en évitant la conversion explicite de ces informations et leur concaténation. Il est particulièrement intéressant pour les nombres réels qu'il est possible d'écrire en imposant un nombre de décimales fixe.

Le format est le suivant :

```
".... %c1   ....  %c2 " % (v1,v2)
```

« **c1** » est un code du format dans lequel la variable « **v1** » devra être transcrite. Il en est de même pour le code « **c2** » associé à la variable **« v2 »**. Les codes insérés dans la chaîne de caractères seront remplacés par les variables citées entre parenthèses après le symbole **« % »** suivant la fin de la chaîne de caractères. Il doit y avoir autant de codes que de variables, qui peuvent aussi être des constantes.

code	signification
d	entier relatif
e	nombre réel au format exponentiel
f	nombre réel au format décimal
g	nombre réel, format décimal ou exponentiel si la puissance est trop grande ou trop petite
s	chaîne de caractères

La seconde affectation de la variable « **res** » propose une solution équivalente à la première en utilisant l'opérateur de concaténation « **+** ».

```
>>> x,d,s = 5.5,7,"caractères"
>>> res = "un nombre réel %f et un entier %d, une chaîne de %s, \n"\
      "un réel d'abord converti en  chaîne de  caractères %s" \
      %  (x,d,s, str(x+4))
>>> print(res)
un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
un réel d'abord converti en  chaîne de  caractères 9.5
>>> res = "un nombre réel " + str (x) + \
          " et un entier " + str (d) + \
          ", une chaîne de " + s + \
          ",\n un réel d'abord converti en chaîne de caractères "\
          + str(x+4)
>>> print(res)
un nombre réel 5.5 et un entier 7, une chaîne de caractères,
 un réel d'abord converti en chaîne de caractères 9.5
```

Les deux solutions sont équivalentes, tout dépend des préférences de celui qui écrit le programme. La première option permet néanmoins un formatage plus précis des nombres réels en imposant par exemple un nombre défini de décimal.

```
>>> x = 0.123456789
>>> print (x)
0.123456789
>>> print ("%1.2f"%x)
0.12
>>> print ("%06.2f"%x)
000.12
```

format

La fonction se révèle particulièrement utile dans tous les cas où vous devez construire une chaîne de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses. Vous pouvez préparer une chaîne « **patron** » contenant l'essentiel du texte invariable, avec des balises particulières aux endroits où vous souhaitez qu'apparaissent des contenus variables. Vous appliquerez ensuite à cette chaîne la méthode « **format** », à laquelle vous fournirez comme arguments les divers objets à convertir en caractères et à insérer en remplacement des balises. Les balises à utiliser sont constituées d'accolades, contenant ou non des indications de formatage.

```
>>> coul ="verte"
>>> temp =1.347 + 15.9
>>> ch ="La couleur est {} et la température vaut {} °C"
>>> print(ch.format(coul, temp))
La couleur est verte et la température vaut 17.247 °C
```

Si les balises sont vides, la méthode « **format** » devra recevoir autant d'arguments qu'il y aura de balises dans la chaîne. Python appliquera alors la fonction « **str** » à chacun de ces arguments, et les insérera ensuite dans la chaîne à la place des balises, dans le même ordre. Les arguments peuvent être n'importe quel objet ou expression :

```
>>> pi =3.14159265358979323846264338327950288419716939937510582
>>> r =4.7
>>> ch ="L'aire d'un disque de rayon {} est égale à {:2.4f}."
>>> print(ch.format(r, pi * r**2))
L'aire d'un disque de rayon 4.7 est égale à 69.3978.
```

Les balises peuvent contenir des numéros d'ordre pour désigner précisément lesquels des arguments transmis à « **format** » devront les remplacer. Cette technique est particulièrement précieuse si le même argument doit remplacer plusieurs balises :

```
>>> phrase ="Le{0} chien{0} aboie{1} et le{0} chat{0} miaule{1}."
>>> print(phrase.format("", ""))
Le chien aboie et le chat miaule.
>>> print(phrase.format("s", "nt"))
Les chiens aboient et les chats miaulent.
```

Le formatage permet d'afficher très facilement divers résultats numériques en notation binaire, octale ou hexadécimale :

```
>>> n =789
>>> txt ="Le nombre {0:d} (décimal) \n"\
         "vaut {0:x} en hexadécimal \n"\
         "et {0:b} en binaire."
>>> print(txt.format(n))
Le nombre 789 (décimal)
vaut 315 en hexadécimal
et 1100010101 en binaire.
```

Le type bytes et la page de code

Comme tous les langages de programmation Python est conçu pour le monde anglophone et l'utilisation des accents ne va pas de soi. Si vous avez rédigé votre script avec un éditeur récent (tels ceux que nous avons déjà indiqués), le script décrit ci-dessus devrait s'exécuter sans problème avec la version actuelle de Python 3. Si votre logiciel est ancien ou mal configuré, il se peut que vous obteniez un message d'erreur similaire à celui-ci :

File "fibo2.py", line 2

SyntaxError: Non-UTF-8 code starting with '\xe0' in file fibo2.py on line 2, but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details

Avec les versions de Python antérieures à la version 3.0, comme dans beaucoup d'autres langages, il fallait fréquemment convertir les chaînes de caractères d'une norme d'encodage à une autre. Du fait des conventions et des mécanismes adoptés désormais, vous ne devrez plus beaucoup vous en préoccuper pour vos propres programmes traitant des données récentes.

Afin que Python puisse les interpréter correctement, il vous est conseillé d'y inclure toujours l'un des pseudo-commentaires suivants (obligatoirement à la 1e ou à la 2e ligne).

```
# -*- coding:latin-1 -*-
```

ou

```
# -*- coding:utf-8 -*-
```

Ainsi l'interpréteur Python sait décoder correctement les chaînes de caractères littérales que vous avez utilisées dans le script. Notez que vous pouvez omettre ce pseudo-commentaire si vous êtes certain que vos scripts sont encodés en « **utf-8** », car c'est cet encodage qui est désormais la norme par défaut pour les scripts Python.

```
>>> import  locale
>>> import sys
>>> print (sys.getdefaultencoding ())
utf-8
>>> locale.getdefaultlocale()
('fr_FR', 'cp1252')
>>> import encodings
>>> print (''.join('- ' + e + '\n' \
       for e in sorted(set(encodings.aliases.aliases.values()))))
- ascii
- base64_codec
- big5
- big5hkscs
- bz2_codec
- cp037
...
- cp1254
...
- iso8859_16
...
- latin_1
...
- utf_16
```

```
- utf_16_be
- utf_16_le
- utf_32
- utf_32_be
- utf_32_le
- utf_7
- utf_8
- uu_codec
- zlib_codec
```

Le type « **bytes** » représente un tableau d'octets. Il fonctionne quasiment pareil que le type « **str** ». Les opérations qu'on peut faire dessus sont quasiment identiques. Les deux méthodes suivantes de la classe « **str** » permettent de convertir une chaîne de caractères en **« bytes »** et l'invers.

encode(enc)	Cette fonction permet de passer d'un jeu de caractères, celui de la variable, au jeu de caractères précisé par **enc** à moins que ce ne soit le jeu de caractères par défaut. Cette fonction retourne un type **« bytes »**.
decode(enc)	Cette fonction est la fonction inverse de la fonction encode. Avec les mêmes paramètres, elle effectue la transformation inverse.

Le type « **bytes** » est très utilisé quand il s'agit de convertit une chaîne de caractères d'une page de code à une autre.

```
>>> b = b"345"
>>> print(b, type(b))
b'345' <class 'bytes'>
>>> b = bytes.fromhex('2Ef0 F1f2 ')
>>> print(b, type(b))
b'.\xf0\xf1\xf2' <class 'bytes'>
>>> b = "abc".encode("utf-8")
>>> s = b.decode("ascii")
>>> print(b, s)
b'abc' abc
>>> print(type(b), type(s))
<class 'bytes'> <class 'str'>
>>> varStr = '圖形碼常用字次常用字'
>>> varBytes = varStr.encode()
>>> print(type(varStr), type(varBytes))
<class 'str'> <class 'bytes'>
>>> print(varStr,'\n',varBytes)
圖形碼常用字次常用字

b'\xe5\x9c\x96\xe5\xbd\xa2\xe7\xa2\xbc\xe5\xb8\xb8\xe7\x94\xa8\xe5\x
ad\x97\xe6\xac\xa1\xe5\xb8\xb8\xe7\x94\xa8\xe5\xad\x97'
```

L'instruction for

Le parcours d'une séquence est une opération très fréquente en programmation. Pour en faciliter l'écriture, Python vous propose une structure de boucle plus appropriée que la boucle « **while** », basée sur le couple d'instructions « **for...in...** ».

```
>>> nom ="Cléopâtre"
>>> for car in nom : print(car + ' *', end =' ')
C * l * é * o * p * â * t * r * e *
```

L'instruction for permet donc d'écrire des boucles, dans lesquelles l'itération traite successivement tous les éléments d'une séquence donnée.

```
>>> liste = ['chien', 'chat', 'crocodile', 'éléphant']
>>> for animal in liste:
        print('longueur de la chaîne', animal, '=', len(animal))

longueur de la chaîne chien = 5
longueur de la chaîne chat = 4
longueur de la chaîne crocodile = 9
longueur de la chaîne éléphant = 8
```

Lors de l'affichage d'une liste, les éléments n'apparaissent pas triés, le langage Python propose néanmoins la fonction « **sorted** ».

```
>>> liste = [3, 2, 1, 6, 4, 9, 7, 8, 5]
>>> for x in liste: print (x)

3
2
1
6
4
9
7
8
5
>>> for x in sorted(liste): print (x)

1
2
3
4
5
6
7
8
9
```

La boucle la plus répandue est celle qui parcourt des indices entiers compris entre 0 et n − 1. On utilise pour cela la boucle for et la fonction « **range** ».

```
range (debut, fin [,marche])
```

Retourne une liste incluant tous les entiers compris entre « **debut** » et « **fin** » exclu. Si le paramètre facultatif marche est renseigné, la liste contient tous les entiers n compris « **debut** » et « **fin** » exclu et tels que « **n – debut** » soit un multiple de « **marche** ».

Les expressions régulières

Les expressions régulières sont prises en charge par le module **« re »**. Ainsi, comme avec tous les modules en Python, nous avons seulement besoin de l'importer pour commencer à les utiliser.

Même si les expressions régulières ne sont pas propres à un langage, chaque implémentation introduit généralement des spécificités pour leur notation. L'antislash **« \ »** tient un rôle particulier dans la syntaxe des expressions régulières puisqu'il permet d'introduire des caractères spéciaux. Comme il est également interprété dans les chaînes de caractères, il est nécessaire de le doubler pour ne pas le perdre dans l'expression.

```
>>> expression = "\btest\b"
>>> print(expression)
test
>>> expression = "\\btest\\b"
>>> print(expression)
\btest\b
>>> expression = r"\btest\b"
>>> print(expression)
\btest\b
```

Les chaînes de caractères peuvent éventuellement être précédées d'une lettre « **r** » ou **« R »** Ces chaînes sont appelées chaînes brutes et traitent l'antislash **« \ »** comme un caractère littéral.

La syntaxe des expressions régulières

La syntaxe des expressions régulières peut se regrouper en trois groupes de symboles :

- les symboles simples ;
- les symboles de répétition ;
- les symboles de regroupement.

Les symboles simples

Les symboles simples sont des caractères spéciaux qui permettent de définir des règles de capture pour un caractère du texte et sont réunis dans le tableau ci-dessous.

Symbole	Fonction
.	Remplace tout caractère sauf le saut de ligne.
^	Symbolise le début d'une ligne.
$	Symbolise la fin d'une ligne.
\A	Symbolise le début de la chaîne.
\b	Symbolise le caractère d'espacement. Intercepté seulement au début ou à la fin d'un mot. Un mot est ici une séquence de caractères alphanumériques ou espace souligné.
\B	Comme **« \b »** mais uniquement lorsque ce caractère n'est pas au début ou à la fin d'un mot.
\d	Intercepte tout chiffre.
\D	Intercepte tout caractère sauf les chiffres.
\s	Intercepte tout caractère d'espacement : horizontale **« \t »**, verticale **« \v »**, saut de ligne **« \n »**, retour à la ligne **« \r »**, form feed **« \f »**.
\S	Symbole inverse de \s
\w	Intercepte tout caractère alphanumérique et espace souligné.
\W	Symbole inverse de **« \w »**.
\Z	Symbolise la fin de la chaîne.

Voici quelques exemples :

```
>>> import re
>>> re.findall(r'.', ' test *')
[' ', 't', 'e', 's', 't', ' ', '*']
>>> re.findall(r'.', 'test\n')
['t', 'e', 's', 't']
>>> re.findall(r'.', '\n')
[]
>>> re.findall(r'^le', "c'est le début")
[]
>>> re.findall(r'^le', "le début")
['le']
>>> re.findall(r'mot$', 'mot mot mot')
['mot']
>>> re.findall(r'\Aparoles', 'paroles, paroles, paroles,\nparoles,
encore des paroooooles')
['paroles']
>>> re.findall(r'\bpar\b', 'parfaitement')
[]
>>> re.findall(r'\bpar\b', 'par monts et par veaux')
['par', 'par']
>>> re.findall(r'\Bpar\B', "imparfait")
['par']
```

```
>>> re.findall(r'\Bpar\B', "parfait")
[]
>>> re.findall(r'\d', '1, 2, 3, nous irons au bois (à 12:15h)')
['1', '2', '3', '1', '2', '1', '5']
>>> print(''.join(re.findall(r'\D', '1, 2, 3, nous irons au bois (à
12:15h)')))
, , , nous irons au bois (à :h)
>>> len(re.findall(r'\s', "combien d'espaces dans la phrase ?"))
5
>>> len(re.findall(r'\s', "latoucheespaceestbloquée"))
0
>>> phrase = """Lancez vous!"""
>>> len(re.findall(r'\s', phrase))
1
>>> len(re.findall(r'\S', "combien de lettres dans la phrase ?"))
29
>>> ''.join(re.findall(r'\w', '*!mot-clé_*'))
'motclé_'
>>> ''.join(re.findall(r'\W', '*!mot-clé_*'))
'*!-*'
>>> re.findall(r'end\Z', 'The end will come')
[]
>>> re.findall(r'end\Z', 'This is the end')
['end']
```

Le fonctionnement de chacun de ces symboles est affecté par les options suivantes :

- **(A)SCII** : les symboles **« \w »**, **« \W »**, **« \b »**, **« \B »**, **« \d »**, **« \D »**, **« \s »** et **« \S »** se basent uniquement sur le code ascii.
- **S** ou **DOTALL** : le saut de ligne est également intercepté par le symbole **« \b »**.
- **(M)ULTILINE** : dans ce mode, les symboles **« ^ »** et **« $ »** interceptent le début et la fin de chaque ligne.
- **(U)NICODE** : les symboles **« \w »**, **« \W »**, **« \b »**, **« \B »**, **« \d »**, **« \D »**, **« \s »** et **« \S »** se basent sur de l'unicode.
- **(I)GNORECASE** : rend les symboles insensibles à la casse du texte.
- **X** ou **VERBOSE** : autorise l'insertion d'espaces et de commentaires en fin de ligne, pour une mise en page de l'expression régulière plus lisible.

```
>>> re.findall('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['a', 'B']
>>> chars = ''.join(chr(i) for i in range(256))
>>> " ".join(re.findall(r"\w", chars))              <--------UNICODE
'0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X
Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z ª ² ³ µ ¹
º ¼ ½ ¾ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ
ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ'
>>> " ".join(re.findall(r"\w", chars, flags=re.UNICODE))
'0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X
Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z ª ² ³ µ ¹
º ¼ ½ ¾ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ
ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ'
>>> " ".join(re.findall(r"\w", chars, flags=re.ASCII))
'0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X
Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z'
```

```
>>> re.findall(r"\w+", "这是一个 例子 是一", re.UNICODE)
['这是一个', '例子', '是一']
>>> re.findall(r"\w+", "这是一个 例子 是一")
['这是一个', '例子', '是一']
```

Les symboles de répétition

Les symboles simples peuvent être combinés et répétés par le biais de symboles de répétition.

Symbole	Fonction
`*`	Répète le symbole précédent de 0 à n fois (autant que possible).
`+`	Répète le symbole précédent de 1 à n fois (autant que possible).
`?`	Répète le symbole précédent 0 ou 1 fois (autant que possible).
`{n}`	Répète le symbole précédent n fois.
`{n,m}`	Répète le symbole précédent entre n et m fois inclus. n ou m peuvent être omis comme pour les tranches de séquences. Dans ce cas ils sont remplacés respectivement par 0 et *.
`{n,m}?`	Équivalent à {n,m} mais intercepte le nombre minimum de caractères.
`e1|e2`	Intercepte l'expression e1 ou e2. (OR)
`[]`	Regroupe des symboles et caractères en un jeu.

Voici quelques exemples :

```
>>> import re
>>> re.findall(r'pois*', 'poisson pois poilant poi')
['poiss', 'pois', 'poi', 'poi']
>>> re.findall(r'pois+', 'poisson pois poilant poi')
['poiss', 'pois']
>>> re.findall(r'pois?', 'poisson pois poilant poi')
['pois', 'pois', 'poi', 'poi']
>>> re.findall(r'pois{2}', 'poisson pois poilant poi')
['poiss']
>>> re.findall(r'pois{2,4}', 'poisssssssssssson pois poilant poi')
['poissss']
>>> re.findall(r'pois{,4}', 'poisssssssssssson pois poilant poi')
['poissss', 'pois', 'poi', 'poi']
>>> re.findall(r'pois{2,}', 'poisssssssssssson pois poilant poi')
['poissssssssssss']
>>> re.findall(r'pois{2,4}?', 'poisssssssssssson pois poilant poi')
['poiss']
>>> re.findall(r'pois{2,}?', 'poisssssssssssson pois poilant poi')
['poiss']
>>> re.findall(r'pois{,4}?', 'poisssssssssssson pois poilant poi')
['poi', 'poi', 'poi', 'poi']
>>> re.findall(r'Mr|Mme', 'Mr et Mme')
['Mr', 'Mme']
>>> re.findall(r'Mr|Mme', 'Mr Untel')
['Mr']
>>> re.findall(r'Mr|Mme', 'Mme Unetelle')
['Mme']
>>> re.findall(r'Mr|Mme', 'Mlle Unetelle')
[]
>>> re.findall(r'[abc]def', 'adef bdef cdef')
```

```
['adef', 'bdef', 'cdef']
```

Le regroupement de caractères accepte aussi des caractères d'abréviation, à savoir :

- **-** : définit une plage de valeurs. **« [a-z] »** représente par exemple toutes les lettres de l'alphabet en minuscules.
- **^** : placé en début de jeu, définit la plage inverse. **« [^a-z] »** représente par exemple tous les caractères sauf les lettres de l'alphabet en minuscules.

Les symboles de répétition **« ? »**, **« * »** et **« + »** sont dits gloutons ou greedy : comme ils répètent autant de fois que possible le symbole précédent, des effets indésirables peuvent survenir.

```
>>> telRegex = re.compile(r'''\d{2}[ ]\(\d\)\d
                [\.]\d{2}[\.]\d{2}
                [\.]\d{2}[\.]\d{2}''', re.VERBOSE)
>>> tel = telRegex.search(
    'Mobile 33 (0)6.85.20.70.68 Tél 33 (0)3.88.27.13.34')
>>> tel.group()
'33 (0)6.85.20.70.68'
>>> telRegex.findall(
    'Mobile 33 (0)6.85.20.70.68 Tél 33 (0)3.88.27.13.34')
['33 (0)6.85.20.70.68', '33 (0)3.88.27.13.34']
```

Dans l'exemple suivant, l'expression régulière tente d'extraire les balises html du texte sans succès : le texte complet est intercepté car il correspond au plus grand texte possible pour le motif. La solution est d'ajouter un symbole **« ? »** après le symbole greedy, pour qu'il n'intercepte que le texte minimum.

```
>>> chaine = '<div><span>le titre</span></div>'
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search(chaine)
>>> mo.group()
'<div>'
>>> nongreedyRegex.findall(chaine)
['<div>', '<span>', '</span>', '</div>']
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search(chaine)
>>> mo.group()
'<div><span>le titre</span></div>'
>>> greedyRegex.findall(chaine)
['<div><span>le titre</span></div>']
```

Les symboles de regroupement

Les symboles de regroupement offrent des fonctionnalités qui permettent de combiner plusieurs expressions régulières, au-delà des jeux de caractères **« [ ] »** et de la fonction **« OR »**, et d'associer à chaque groupe un identifiant unique. Certaines d'entre elles permettent aussi de paramétrer localement le fonctionnement des expressions.

Symbole	Fonction
(e)	Forme un groupe avec l'expression e. Si les caractères **« (»** ou **«) »** sont utilisés dans e, ils doivent être préfixés de **« \ »**
(?FLAGS)	Insère directement des flags d'options dans l'expression. S'applique à l'expression complète quel que soit son positionnement.
(?:e)	Similaire à (e) mais le groupe intercepté n'est pas conservé.
(?P<name>e)	Associe l'étiquette name au groupe. Ce groupe peut ensuite être manipulé par ce nom par le biais des API de « **re** », ou même dans la suite de l'expression régulière.
(?#comment)	Insère un commentaire, qui sera ignoré. Le mode **« verbose »** est plus souple pour l'ajout direct de commentaires en fin de ligne.
(?=e)	Similaire à (e) mais le groupe n'est pas consommé.
(?!e)	Le groupe n'est pas consommé et est intercepté uniquement si le pattern (le motif) n'est pas e. (?!e) est le symbole inverse de (?=e)
(?<=e1)e2	Intercepte e2 à condition qu'elle soit préfixée d'e1.
(?<!e1)e2	Intercepte e2 à condition qu'elle ne soit pas préfixée d'e1.
(?(id/name) e1|e2)	Rend l'expression conditionnelle : si le groupe d'identifiant id ou name existe, e1 est utilisée, sinon e2. e2 peut être omise, dans ce cas e1 ne s'applique que si le groupe id ou name existe. Dans l'exemple <123> et 123 sont interceptés mais pas <123.

Voici quelques exemples.

```
>>> re.findall(r'(\(03\))(80)(.*)','(03)80666666')
[('(03)', '80', '666666')]
>>> re.findall(r'(?i)AAZ*', 'aaZzzRr')
['aaZzz']
>>> re.findall(r'(?:\(03\))(?:80)(.*)','(03)80666666')
['666666']
>>> match = re.search(r'(03)(80)(?P<numero>.*)','0380666666')
>>> match.group('numero')
'666666'
>>> re.findall(r'(?# récupération des balises)<.*?>', \
    '<h2><span>hopla</span></h2>')
['<h2>', '<span>', '</span>', '</h2>']
>>> re.findall(r'John(?= Doe)','John Doe')
['John']
>>> re.findall(r'John(?= Doe)','John Minor')
[]
>>> re.findall(r'John(?! Doe)','John Doe')
```

```
[]
>>> re.findall(r'John(?! Doe)','John Minor')
['John']
>>> re.findall(r'(?<=John )Doe','John Doe')
['Doe']
>>> re.findall(r'(?<=John )Doe','John Minor')
[]
>>> re.findall(r'(?<!John )Doe','John Doe')
[]
>>> re.findall(r'(?<!John )Doe','Juliette Doe')
['Doe']
>>> re.match(r'(?P<one><)?(\d+)(?(one)>)', '<123')
>>> match = re.match(r'(?P<one><)?(\d+)(?(one)>)', '123')
>>> match.group()
'123'
>>> match = re.match(r'(?P<one><)?(\d+)(?(one)>)', '<123>')
>>> match.group()
'<123>'
```

Les fonctions et objets de re

Le module **« re »** contient un certain nombre de fonctions qui permettent de manipuler des motifs et les exécuter sur des chaînes :

Fonction	Description
compile(pattern[, flags])	compile le motif pattern et renvoie un objet de type **« SRE_Pattern »**.
escape(string)	ajoute un antislash **« \ »** devant tous les caractères non alphanumériques contenus dans string. Permet d'utiliser la chaîne dans les expressions régulières.
findall(pattern, string[, flags])	renvoie une liste des éléments interceptés dans la chaîne string par le motif pattern. Lorsque le motif est composé de groupes, chaque élément est un tuple composé de chaque groupe.
finditer(pattern, string[, flags])	équivalente à **« findall »**, mais un itérateur sur les éléments est renvoyé. flags est un entier contenant d'éventuels flags, appliqués au motif complet.
match(pattern, string[, flags])	renvoie un objet de type **« MatchObject »** si le début de la chaîne string correspond au motif. flags est un entier contenant d'éventuels flags, appliqués au motif complet.
search(pattern, string[, flags])	équivalente à **« match »** mais recherche le motif dans toute la chaîne.
split(pattern, string[,maxsplit=0])	équivalente au **« split »** de l'objet string. Renvoie une séquence de chaînes délimitées par le motif pattern. Si maxsplit est fourni, limite le nombre d'éléments à maxsplit, le dernier élément regroupant la fin de la chaîne lorsque maxsplit est atteint.
sub(pattern, repl, string[, count])	remplace les occurrences du motif pattern de string par repl. repl peut être une chaîne ou un objet **« callable »** qui reçoit un objet « MatchObject » et renvoie une chaîne. Si count est fourni, limite le nombre de remplacements.
subn(pattern, repl, string[, count])	équivalente à **« sub »** mais renvoie un tuple (nouvelle chaîne, nombre de remplacements) au lieu de la chaîne.

```
>>> import re
>>> motif = re.compile('(Mr|Mme|Mlle)\s([A-Za-z]+)\s([A-Za-z]+)')
>>> print(motif.sub(r'Nom: \3, Prénom: \2', 'Mr John Doe'))
Nom: Doe, Prénom: John
>>> print(motif.sub(r'Mon nom est \g<3>, \g<2> \g<3>'\
          ,'Mr Jean Bon'))
Mon nom est Bon, Jean
```

L'écriture simplifiée des fonctions

Lorsque le code d'une fonction tient en une ligne et est le résultat d'une expression, il est possible de condenser son écriture à l'aide du mot-clé « **lambda** ». Cette syntaxe est issue de langages fonctionnels comme le Lisp.

```
nom_fonction = lambda param_1, ..., param_n : expression
```

L'exemple suivant utilise cette écriture pour définir la fonction min retournant le plus petit entre deux nombres positifs.

```
>>> min = lambda x,y : (abs (x+y) - abs (x-y)) / 2
>>> print(min(1,2))
1.0
>>> print(min(5,4))
4.0
>>> def  min(x,y):
        return (abs (x+y) - abs (x-y))/2

>>> print(min(1,2))
1.0
>>> print(min(5,4))
4.0
```

La fonction lambda considère le contexte de fonction qui la contient comme son contexte. Il est possible de créer des fonctions lambda mais celle-ci utiliseront le contexte dans l'état où il est au moment de son exécution et non au moment de sa création.

```
>>> fs = []
>>> for a in range (0,5) :
...     f = lambda x : x + a
...     fs.append(f)
...     print(a)
0
1
2
3
4
>>> print(a)
4
>>> for f in fs :
...     print('a = ', a,' lambda = ',f(1))
a =  4  lambda =  5
a =  4  lambda =  5
a =  4  lambda =  5
a =  4  lambda =  5
a =  4  lambda =  5
```

Pour que le programme affiche les entiers de 1 à 5, il faut préciser à la fonction lambda une variable y égale à a au moment de la création de la fonction et qui sera intégrée au contexte de la fonction lambda.

```
>>> fs = []
>>> for a in range (0,5) :
...     f = lambda x,y=a : x + y
...     fs.append(f)
```

```
...
>>> for f in fs :
...     print('a = ', a,' lambda = ',f(1))
...
a =  4  lambda =  1
a =  4  lambda =  2
a =  4  lambda =  3
a =  4  lambda =  4
a =  4  lambda =  5
```

La fonction map

La fonction « **map** » renvoie une liste correspondant à l'ensemble des éléments de la séquence.

```
map : map(fonction, séquence[, séquence...]) -> liste
```

Avant d'être inséré dans la liste, chaque élément est passé à la fonction fournie. Cette dernière doit donc être de la forme :

```
fonction(element) -> element
```

Lorsque plusieurs séquences sont fournies, la fonction reçoit une liste d'arguments correspondants à un élément de chaque séquence. Si les séquences ne sont pas de la même longueur, elles sont complétées avec des éléments à la valeur « **None** ».

La fonction peut être définie à « **None** », et dans ce cas tous les éléments des séquences fournies sont conservés.

```
>>> a = lambda x: x**2
>>> b = list(map(a, (2,3,4)))
>>> b
[4, 9, 16]
>>> list(map(pow, (7, 5, 3), (2, 3, -2)))
[49.0, 125.0, 0.1111111111111111]
>>> a, c = [1,2,3,4,5], [2, -3, 5,7, -0.5]
>>> mul = lambda x, y: x*y
>>> d = sum(map(mul, a,c))
>>> d
36.5

>>> sq = lambda x: x**2
>>> cu = lambda y: y**3
>>> fc = (sq,cu)
>>> #Retour carré & cube de r - utilisation de 'map'
>>> def vv(r): return list(map(lambda z: z(r), fc))
>>> res1 = vv(5)
>>> res1
[25, 125]

>>> def meva(dd):
        'Avec dd comme sequence de nombres,\n\
        retrouvez leur moyenne et variance'
        bb = len(dd)
        med = sum(dd)/bb
        vr = sum(list(map(sq,dd)))/bb - med**2
        return med, vr

>>> seq = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, \
       6.0, 7.0, 8.0, 9.0, 10.0, 11.0, \
       12.0, 13.0, 14.0]
>>> res1 = meva(seq)
>>> res1
(7.0, 18.66666666666667)
```

La fonction filter

La fonction « **filter** » renvoie une liste correspondant à l'ensemble des éléments de la séquence pour lesquels la fonction fournie retourne vrai.

```
filter(fonction, séquence[, séquence...]) -> liste
```

Avant d'être inséré dans la liste, chaque élément est passé à la fonction fournie. Cette dernière doit donc être de la forme :

```
fonction(element) -> booléen
```

```
>>> symbols = '&#$€£§ç'
>>> code_car = [ord(s) for s in symbols if ord(s) > 160]
>>> code_car
[8364, 163, 167, 231]
>>>
>>> code_car = list(filter(lambda c: c > 160, map(ord, symbols)))
>>> code_car
[8364, 163, 167, 231]

>>> def factorial(n):
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> map(factorial, range(11))
<map object at 0x0000023169057CC0>
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
>>> list(map(factorial, filter(lambda n: n % 2, range(6))))
[1, 6, 120]
```

3

Les outils indispensables

Les tableaux numériques

Ce type ne fait pas partie du langage python standard mais il est couramment utilisé. Il permet de convertir des listes en une structure plus appropriée au calcul qui sont nettement plus rapides. En contrepartie, il n'est pas aussi rapide d'ajouter ou supprimer des éléments.

Le type de base dans « **NumPy** » est le tableau unidimensionnel ou multidimensionnel composé d'éléments de même type, et est indexé par un **« tuple »** d'entiers non négatifs. La classe correspondante est **« ndarray »**, à ne pas confondre avec la classe Python **« array.array »** qui gère seulement des tableaux unidimensionnels et présente des fonctionnalités comparativement limitées.

ndarray.ndim	dimension du tableau (nombre d'axes)
ndarray.shape	tuple d'entiers indiquant la taille dans chaque dimension ; une matrice à n lignes et m colonnes : (n,m)
ndarray.size	nombre total d'éléments du tableau
ndarray.dtype	type de (tous) les éléments du tableau ; il est possible d'utiliser les types prédéfinis comme numpy.int64 ou numpy.float64 ou définir de nouveaux types
ndarray.data	les données du tableau ; en général, pour accéder aux données d'un tableau on passe plutôt par les indices

L'emploi de raccourcis « **np** » plutôt que **« numpy »** permet de faciliter l'écriture des appels des fonctions de la librairie.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> print(type(a))
<class 'numpy.ndarray'>
>>> print(a.shape)
(3,)
>>> print(a[0], a[1], a[2])
1 2 3
>>> a[0] = 5
>>> print(a)
[5 2 3]
>>> b = np.array([[1,2,3],[4,5,6]])
>>> print(b.shape)
(2, 3)
>>> print(b[0, 0], b[0, 1], b[1, 0])
1 2 4
```

La libraire **« numpy »** fournit également de nombreuses fonctions pour créer des tableaux :

```
>>> import numpy as np
>>> a = np.zeros((2,2))
>>> print(a)
[[ 0.  0.]
 [ 0.  0.]]
>>> b = np.ones((1,2))
>>> print(b)
[[ 1.  1.]]
```

```
>>> c = np.full((2,2), 7)
>>> print(c)
[[ 7.  7.]
 [ 7.  7.]]
>>> d = np.eye(2)
>>> print(d)
[[ 1.  0.]
 [ 0.  1.]]
>>> e = np.random.random((2,2))
>>> print(e)
[[ 0.72843251  0.10508965]
 [ 0.84919262  0.75706259]]
```

Vous pouvez également mélanger l'indexation des entiers avec l'indexation des tranches. Cependant, cela produira un tableau de rang inférieur à celui du tableau original.

```
>>> import numpy as np
>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> row_r1 = a[1, :]
>>> row_r2 = a[1:2, :]
>>> print(row_r1, row_r1.shape)
[5 6 7 8] (4,)
>>> print(row_r2, row_r2.shape)
[[5 6 7 8]] (1, 4)
>>> col_r1 = a[:, 1]
>>> col_r2 = a[:, 1:2]
>>> print(col_r1, col_r1.shape)
[ 2  6 10] (3,)
>>> print(col_r2, col_r2.shape)
[[ 2]
 [ 6]
 [10]] (3, 1)

>>> a.ndim
2
>>> a.shape
(3, 4)
>>> a.size
12
>>> a.dtype
dtype('int32')
```

La création de tableaux

De nombreuses méthodes de création de tableaux sont disponibles. D'abord, un tableau peut être créé à partir d'une **« liste »** ou d'un **« tuple »**, à condition que tous les éléments soient de même type, le type des éléments du tableau est déduit du type des éléments de la **« liste »** ou **« tuple »**.

```
>>> import numpy as np
>>> ti = np.array([1, 2, 3, 4])
>>> ti
array([1, 2, 3, 4])
>>> ti.dtype
dtype('int32')
>>> tf = np.array([1.5, 2.5, 3.5, 4.5])
>>> tf.dtype
dtype('float64')
```

À partir des listes simples sont produits des tableaux unidimensionnels, à partir des listes de listes (de même taille) des tableaux bidimensionnels, et ainsi de suite.

```
>>> tf2d = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> tf2d
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Le type du tableau peut être indiqué explicitement à la création, des conversions sont effectuées pour les valeurs fournies.

```
>>> tfi = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=int)
>>> tfi
array([[1, 2, 3],
       [4, 5, 6]])
>>> tfi.dtype
dtype('int32')
>>> tfi.shape
(2, 3)
>>> tfi.ndim
2
>>> tfi.size
6
```

Il est souvent nécessaire de créer des tableaux remplis de 0, de 1, ou dont le contenu n'est pas initialisé. Par défaut, le type des tableaux ainsi créés est float64.

```
>>> tz2d = np.zeros((3,4))
>>> tz2d
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> tu2d = np.ones((3,4))
>>> tu2d
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> id2d = np.eye(5)
>>> id2d
array([[ 1.,  0.,  0.,  0.,  0.],
```

```
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
>>> tni2d = np.empty((3,4))
>>> tni2d
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Des tableaux peuvent être initialisés aussi par des séquences générées.

```
>>> ts1d = np.arange(0, 40, 5)
>>> ts1d
array([ 0,  5, 10, 15, 20, 25, 30, 35])
>>> ts1d2 = np.linspace(0, 35, 8)
>>> ts1d2
array([  0.,   5.,  10.,  15.,  20.,  25.,  30.,  35.])
>>> ta2d = np.random.rand(3,5)
>>> ta2d
array([[ 0.57132596,  0.00819932,  0.17252526,  0.03082183,  0.53830712],
       [ 0.83042098,  0.75446994,  0.25991065,  0.62847979,  0.15797572],
       [ 0.08598435,  0.00754915,  0.50282216,  0.72654331,  0.90316578]])
```

Les tableaux peuvent être redimensionnés en utilisant **« `reshape` »**.

```
>>> tr = np.arange(20)
>>> tr
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
>>> tr.reshape(4,5)
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> tr.reshape(2,10)
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
>>> tr.reshape(20)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

L'affichage des tableaux

Les tableaux unidimensionnels sont affichés comme des listes, les tableaux bidimensionnels comme des matrices et les tableaux tridimensionnels comme des listes de matrices.

```
>>> ta1d = np.random.rand(5)
>>> ta1d
array([ 0.79064584,  0.06775449,  0.39452303,  0.01723293,  0.25219518])
>>> ta3d = np.random.rand(2,3,5)
>>> ta3d
array([[[ 0.48491945,  0.64296218,  0.36891503,  0.32513357,  0.67010718],
        [ 0.25877942,  0.54814236,  0.76000793,  0.80090898,  0.49214619],
        [ 0.25499884,  0.54651908,  0.33886573,  0.52616233,  0.35187091]],

       [[ 0.32749216,  0.49832111,  0.27457405,  0.48603902,  0.10054472],
        [ 0.0191687 ,  0.06256129,  0.1715306 ,  0.60250356,  0.2672456 ],
        [ 0.78437914,  0.72383496,  0.16868848,  0.84315508,  0.0267777 ]]])
```

Si un tableau est considéré trop grand pour être affiché en entier, **« NumPy »** affiche le début et la fin, avec des **« ... »** au milieu.

```
>>> ta2d = np.random.rand(30,50)
>>> ta2d
array([[ 0.6835294 ,  0.46886275,  0.81712639, ...,  0.01835169,
         0.89688112,  0.37007707],
       [ 0.5791199 ,  0.30927608,  0.14415233, ...,  0.92076745,
         0.57043746,  0.42868106],
       [ 0.66825711,  0.84835774,  0.43771497, ...,  0.24745712,
         0.26247765,  0.74588404],
       ...,
       [ 0.35933294,  0.94859926,  0.77803165, ...,  0.71668177,
         0.74483146,  0.44250986],
       [ 0.69740862,  0.07109011,  0.86203099, ...,  0.58584055,
         0.74100104,  0.32189666],
       [ 0.38432312,  0.39479927,  0.34965807, ...,  0.37925167,
         0.47014018,  0.68201961]])
```

L'accès aux composantes d'un tableau

```
>>> tr = np.arange(20)
>>> tr
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
>>> a = tr[:2]
>>> a
array([0, 1])
>>> a[0] = 3
>>> tr
array([ 3,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Les données ne sont pas copiées de **« tr »** vers un nouveau tableau **« a »**, la modification d'un élément de a avec **« a[0] = 3 »** change aussi le contenu de **« tr[0] »**. Pour obtenir une copie il faut utiliser **« copy »**.

```
>>> a = tr[:2].copy()
>>> a
array([3, 1])
>>> a[0] = 0
>>> a
array([0, 1])
>>> tr
array([ 3,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Pour les tableaux multidimensionnels, lors des itérations c'est le dernier indice qui change le plus vite, ensuite l'avant-dernier, et ainsi de suite. Par exemple, pour les tableaux bidimensionnels c'est l'indice de colonne qui change d'abord et ensuite celui de ligne ainsi le tableau est lu ligne après ligne.

```
>>> ta2d = np.random.rand(3,5)
>>> ta2d
array([[ 0.99327185,  0.85869692,  0.87930205,  0.03911094,  0.93273251],
       [ 0.86430312,  0.16671027,  0.61613967,  0.84354217,  0.1950944 ],
       [ 0.40074433,  0.52467501,  0.71025502,  0.55148182,  0.0599687 ]])
>>> ta2d[0,0]
0.99327184504277644
>>> ta2d[0,:]
array([ 0.99327185,  0.85869692,  0.87930205,  0.03911094,  0.93273251])
>>> ta2d[0]
array([ 0.99327185,  0.85869692,  0.87930205,  0.03911094,  0.93273251])
>>> ta2d[:,0]
array([ 0.99327185,  0.86430312,  0.40074433])
>>> ta2d[:2,:2]
array([[ 0.99327185,  0.85869692],
       [ 0.86430312,  0.16671027]])
>>> for row in ta2d:
...     print(row)
...
[ 0.99327185  0.85869692  0.87930205  0.03911094  0.93273251]
[ 0.86430312  0.16671027  0.61613967  0.84354217  0.1950944 ]
[ 0.40074433  0.52467501  0.71025502  0.55148182  0.0599687 ]
```

Lecture et écriture d'un tableau

Les fonctions de lecture / écriture de tableaux depuis / dans des fichiers sont variées, nous regarderons rapidement deux des plus simples et plus rapides car les fichiers de données ont en général des formats assez simples.

La fonction :

numpy.loadtxt(fname, dtype=<type 'float'>,

comments='#', delimiter=None, converters=None,

skiprows=0, usecols=None, unpack=False, ndmin=0),

qui retourne un **ndarray**, réalise une lecture à partir d'un fichier texte et est bien adaptée aux tableaux bidimensionnels ; chaque ligne de texte doit contenir un même nombre de valeurs. Les principaux paramètres sont :

fname : fichier ou chaîne de caractères ; si le fichier a une extension .gz ou .bz2, il est d'abord décompressé.

dtype : type, optionnel, float par défaut.

comments : chaîne de caractères, optionnel, indique une liste de caractères employée dans le fichier pour précéder des commentaires à ignorer lors de la lecture.

delimiter : chaîne de caractères, optionnel, indique la chaîne de caractères employée pour séparer des valeurs, par défaut l'espace.

converters : dictionnaire, optionnel, pour permettre des conversions.

skiprows : entier, optionnel, pour le nombre de lignes à sauter en début de fichier par défaut 0.

usecols : séquence, optionnel, indique les colonnes à lire ; par ex. **usecols = [1,4,5]** extrait la 2ème, 5ème et 6ème colonne ; par défaut toutes les colonnes sont extraites.

unpack : booléen, optionnel, false par défaut ; si true, le tableau est transposé.

ndmin : entier, optionnel, le tableau a au moins **ndmin** dimensions ; par défaut 0.

```
>>> from io import StringIO
>>> import numpy as np
>>> nutriage = np.loadtxt('donnees/nutriage.csv',
...                delimiter=';',skiprows=1)
>>> nutriage.shape
(226, 13)
>>> nutriage[:10,:6]
array([[   2.,    1.,    0.,    0.,  151.,   58.],
       [   2.,    1.,    1.,    1.,  162.,   60.],
       [   2.,    1.,    0.,    4.,  162.,   75.],
       [   2.,    1.,    0.,    0.,  154.,   45.],
       [   2.,    1.,    2.,    1.,  154.,   50.],
       [   2.,    1.,    2.,    0.,  159.,   66.],
       [   2.,    1.,    2.,    0.,  160.,   66.],
       [   2.,    1.,    0.,    2.,  163.,   66.],
       [   2.,    1.,    0.,    3.,  154.,   60.],
```

```
       [   2.,    1.,    0.,    2.,  160.,   77.]])
```

La fonction :

numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ',
newline='\n', header='',
footer='', comments='# ')

permet d'écrire un tableau dans un fichier texte. Les paramètres sont :

fname : fichier ; si le fichier a une extension .gz ou .bz2, il est compressé.

X : le tableau à écrire dans le fichier texte.

fmt : chaîne de caractères, optionnel ; indique le formatage du texte écrit.

delimiter : chaîne de caractères, optionnel, indique la chaîne de caractères employée pour séparer des valeurs, par défaut `` `` (l'espace).

newline : chaîne de caractères, optionnel, indique le caractère à employer pour séparer des lignes.

header : chaîne de caractères, optionnel, indique le commentaire à ajouter au début du fichier.

footer : chaîne de caractères, optionnel, indique le commentaire à ajouter à la fin du fichier.

comments : caractère à ajouter avant header et footer pour en faire des commentaires ; par défaut #.

```
>>> np.savetxt('donnees/nutriage.txt', nutriage, delimiter=', ')
>>> nutriage = np.loadtxt('donnees/nutriage.txt', delimiter=',')
>>> nutriage.shape
(226, 13)
>>> nutriage[:10,:6]
array([[   2.,    1.,    0.,    0.,  151.,   58.],
       [   2.,    1.,    1.,    1.,  162.,   60.],
       [   2.,    1.,    0.,    4.,  162.,   75.],
       [   2.,    1.,    0.,    0.,  154.,   45.],
       [   2.,    1.,    2.,    1.,  154.,   50.],
       [   2.,    1.,    2.,    0.,  159.,   66.],
       [   2.,    1.,    2.,    0.,  160.,   66.],
       [   2.,    1.,    0.,    2.,  163.,   66.],
       [   2.,    1.,    0.,    3.,  154.,   60.],
       [   2.,    1.,    0.,    2.,  160.,   77.]])
```

Autres opérations d'entrée et sortie :

fromfile(file[, dtype, count, sep]) : Construction d'un tableau à partir d'un fichier texte ou binaire.

fromregex(file, regexp, dtype) : Construction d'un tableau à partir d'un fichier texte, avec un parseur d'expressions régulières.

genfromtxt() : Fonction plus flexible pour la construction d'un tableau à partir d'un fichier texte, avec une gestion des valeurs manquantes.

load(file[, mmap_mode, allow_pickle, ...]) : Lecture de tableaux (ou autres objets) à partir de fichiers .npy, .npz ou autres fichiers de données sérialisées.

`loadtxt(fname[, dtype, comments, delimiter, ...])` : Lecture de données à partir d'un fichier texte.

`ndarray.tofile(fid[, sep, format])` : Ecriture d'un tableau dans un fichier texte ou binaire (par défaut).

`save(file, arr[, allow_pickle, fix_imports])` : Ecriture d'un tableau dans un fichier binaire de type .npy.

`savetxt(fname, X[, fmt, delimiter, newline, ...])` : Ecriture d'un tableau dans un fichier texte.

`savez(file, *args, **kwds)` : Ecriture de plusieurs tableaux dans un fichier de type .npz sans compression.

`savez_compressed(file, *args, **kwds)` : Ecriture de plusieurs tableaux dans un fichier de type .npz avec compression.

Les opérations simples sur les tableaux

Concaténation de tableaux bidimensionnels

```
>>> tu2d = np.ones((2,2))
>>> tu2d
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> tb2d = np.ones((2,2))*2
>>> tb2d
array([[ 2.,  2.],
       [ 2.,  2.]])
>>> tcl = np.concatenate((tu2d,tb2d))
>>> tcl
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 2.,  2.],
       [ 2.,  2.]])
>>> tcl = np.concatenate((tu2d,tb2d),axis=0)   # ou np.vstack
>>> tcl
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 2.,  2.],
       [ 2.,  2.]])
>>> tcl = np.concatenate((tu2d,tb2d),axis=1)   # ou np.hstack
>>> tcl
array([[ 1.,  1.,  2.,  2.],
       [ 1.,  1.,  2.,  2.]])
```

Ajouter un tableau unidimensionnel comme colonne à un tableau bidimensionnel.

```
>>> from numpy import newaxis
>>> tu1d = np.ones(2)
>>> tu1d
array([ 1.,  1.])
>>> tu1d[:,newaxis]
array([[ 1.],
       [ 1.]])
>>> np.column_stack((tu1d[:,newaxis],tb2d))
array([[ 1.,  2.,  2.],
       [ 1.,  2.,  2.]])
>>> np.hstack((tu1d[:,newaxis],tb2d))
array([[ 1.,  2.,  2.],
       [ 1.,  2.,  2.]])
```

Opérations arithmétiques élément par élément

```
>>> tsomme = tb2d - tu1d
>>> tsomme
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> tb2d*5
array([[ 10.,  10.],
       [ 10.,  10.]])
>>> tb2d**2
array([[ 4.,  4.],
```

```
       [ 4.,  4.]])
>>> tc = np.hstack((tb2d,tu1d[:,newaxis]))
>>> tc
array([[ 2.,  2.,  1.],
       [ 2.,  2.,  1.]])
>>> tc > 1
array([[ True,  True, False],
       [ True,  True, False]], dtype=bool)
>>> tb2d * tb2d
array([[ 4.,  4.],
       [ 4.,  4.]])
>>> tb2d *= 3
>>> tb2d
array([[ 6.,  6.],
       [ 6.,  6.]])
>>> tb2d += tu2d
>>> tb2d
array([[ 7.,  7.],
       [ 7.,  7.]])
>>> tb2d.sum()
28.0
>>> nutriage = np.loadtxt('donnees/nutriage.csv',
...                delimiter=';',skiprows=1)
>>> nutriage.shape
(226, 13)
>>> nutriage[:10,:6]
array([[   2.,    1.,    0.,    0.,  151.,   58.],
       [   2.,    1.,    1.,    1.,  162.,   60.],
       [   2.,    1.,    0.,    4.,  162.,   75.],
       [   2.,    1.,    0.,    0.,  154.,   45.],
       [   2.,    1.,    2.,    1.,  154.,   50.],
       [   2.,    1.,    2.,    0.,  159.,   66.],
       [   2.,    1.,    2.,    0.,  160.,   66.],
       [   2.,    1.,    0.,    2.,  163.,   66.],
       [   2.,    1.,    0.,    3.,  154.,   60.],
       [   2.,    1.,    0.,    2.,  160.,   77.]])
>>> nutriage.min()
0.0
>>> nutriage.max()
188.0
>>> nutriage.sum()
75004.0
>>> nutriage.sum(axis=0)
array([   367.,    363.,    161.,    366.,  37055.,  15025.,  16832.,
         847.,    592.,   1014.,    991.,    529.,    862.])
>>> nutriage[:20,:].sum(axis=1)
array([ 307.,  317.,  342.,  306.,  298.,  332.,  332.,  330.,  337.,
       346.,  364.,  344.,  331.,  320.,  342.,  346.,  313.,  311.,
       329.,  349.])
```

Algèbre linéaire

```
>>> n0 = nutriage[:2,5:7]
>>> n0
array([[ 58.,  72.],
       [ 60.,  68.]])
```

```
>>> n0.transpose()
array([[ 58.,  60.],
       [ 72.,  68.]])
>>> np.linalg.inv(n0)
array([[-0.18085106,  0.19148936],
       [ 0.15957447, -0.15425532]])
>>> n0.dot(tu2d)      # ou np.dot(n0,tu2d)
array([[ 130.,  130.],
       [ 128.,  128.]])
>>> tu2d
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> n0.dot(np.eye(2))
array([[ 58.,  72.],
       [ 60.,  68.]])
>>> n0.trace()
126.0
```

Résolution de systèmes linéaires

```
>>> y = np.array([[5.], [7.]])
>>> np.linalg.solve(n0, y)
array([[ 0.43617021],
       [-0.28191489]])
```

Valeurs et vecteurs propres

```
>>> vpvp = np.linalg.eig(n0)
>>> vpvp
(array([  -2.91661399,  128.91661399]), array([[-0.76342008, -0.71244657],
       [ 0.64590231, -0.70172636]]))
```

Vectorisation de fonctions

Des fonctions Python qui travaillent sur des scalaires peuvent être vectorisées, c'est à dire travailler sur des tableaux, élément par élément.

```
>>> def addsubtract(a,b):
...     if a > b:
...         return a - b
...     else:
...         return a + b
...
>>> addsubtract(2,3)
5
>>> vec_addsubtract = np.vectorize(addsubtract)
>>> tu2d
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> n0
array([[ 58.,  72.],
       [ 60.,  68.]])
>>> vec_addsubtract(n0,tu2d)
array([[ 57.,  71.],
       [ 59.,  67.]])
```

La librairie Matplotlib

La librairie Matplotlib a vu le jour pour permettre de générer directement des graphiques à partir de Python. Au fil des années, Matplotlib est devenu une librairie puissante, compatible avec beaucoup de plateformes, et capable de générer des graphiques dans beaucoup de formats différents.

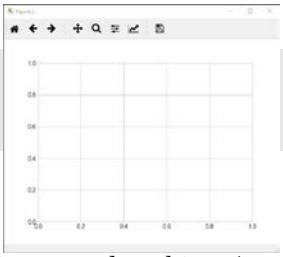
Pour commencer, mettons en place l'environnement de travail.

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('seaborn-whitegrid')
>>> import numpy as np
```

Réaliser des graphiques simples

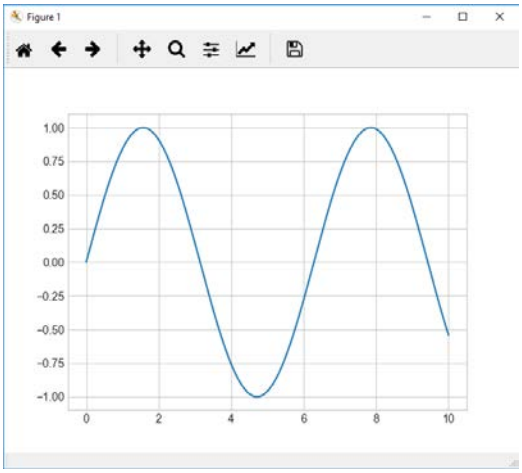
Commençons par étudier un cas simple, comme tracer la courbe d'une fonction. Nous avons vu un exemple de cette utilisation dans le chapitre 3 de la première partie de ce cours. Ici, nous allons le faire d'une manière moins simple, mais qui nous donne plus de possibilités.

```
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> plt.show()
```



La variable **`fig`** correspond à un conteneur qui contient tous les objets (axes, labels, données, etc). Les axes correspondent au carré que l'on voit au-dessus, et qui contiendra par la suite les données du graphe.

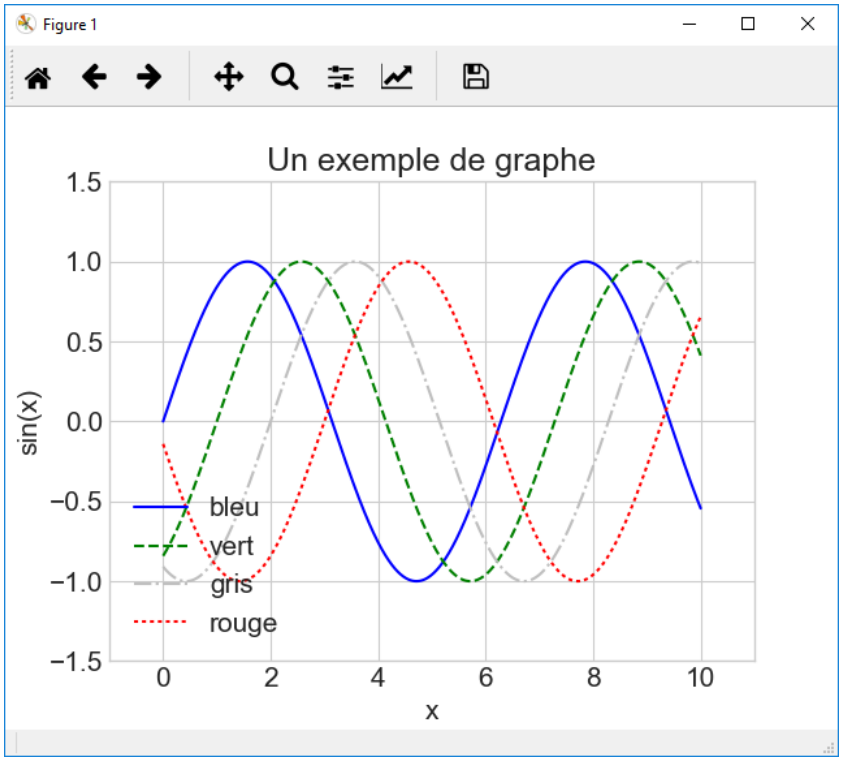
```
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> x = np.linspace(0, 10, 1000)
>>> ax.plot(x, np.sin(x));
[<matplotlib.lines.Line2D object at 0x0000022885641860>]
>>> plt.show()
```



On aurait pu simplement taper **`plt.plot(x, np.sin(x))`**. Maintenant, voyons un exemple un peu plus poussé.

```
>>> # Changer la taille de police par défaut
... plt.rcParams.update({'font.size': 15})
```

```
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> # Couleur spécifiée par son nom, ligne solide
>>> plt.plot(x, np.sin(x - 0), color='blue', linestyle='solid',
label='bleu')
[<matplotlib.lines.Line2D object at 0x00000228865DD198>]
>>> # Nom court pour la couleur, ligne avec des traits
>>> plt.plot(x, np.sin(x - 1), color='g', linestyle='dashed',
label='vert')
[<matplotlib.lines.Line2D object at 0x00000228865DD358>]
>>> # Valeur de gris entre 0 et 1, des traits et des points
>>> plt.plot(x, np.sin(x - 2), color='0.75', linestyle='dashdot',
label='gris')
[<matplotlib.lines.Line2D object at 0x00000228865E6320>]
>>> # Couleur spécifié en RGB, avec des points
>>> plt.plot(x, np.sin(x - 3), color='#FF0000', linestyle='dotted',
label='rouge')
[<matplotlib.lines.Line2D object at 0x00000228865E6AC8>]
>>> # Les limites des axes, essayez aussi les arguments 'tight' et
'equal' pour voir leur effet
>>> plt.axis([-1, 11, -1.5, 1.5]);
[-1, 11, -1.5, 1.5]
>>> # Les labels
>>> plt.title("Un exemple de graphe")
<matplotlib.text.Text object at 0x000002288643DCF8>
>>> # La légende est générée à partir de l'argument label de la
fonction plot. L'argument loc spécifie le placement de la légende
>>> plt.legend(loc='lower left');
<matplotlib.legend.Legend object at 0x00000228865E6B38>
>>> # Titres des axes
>>> ax = ax.set(xlabel='x', ylabel='sin(x)')
>>> plt.show()
```



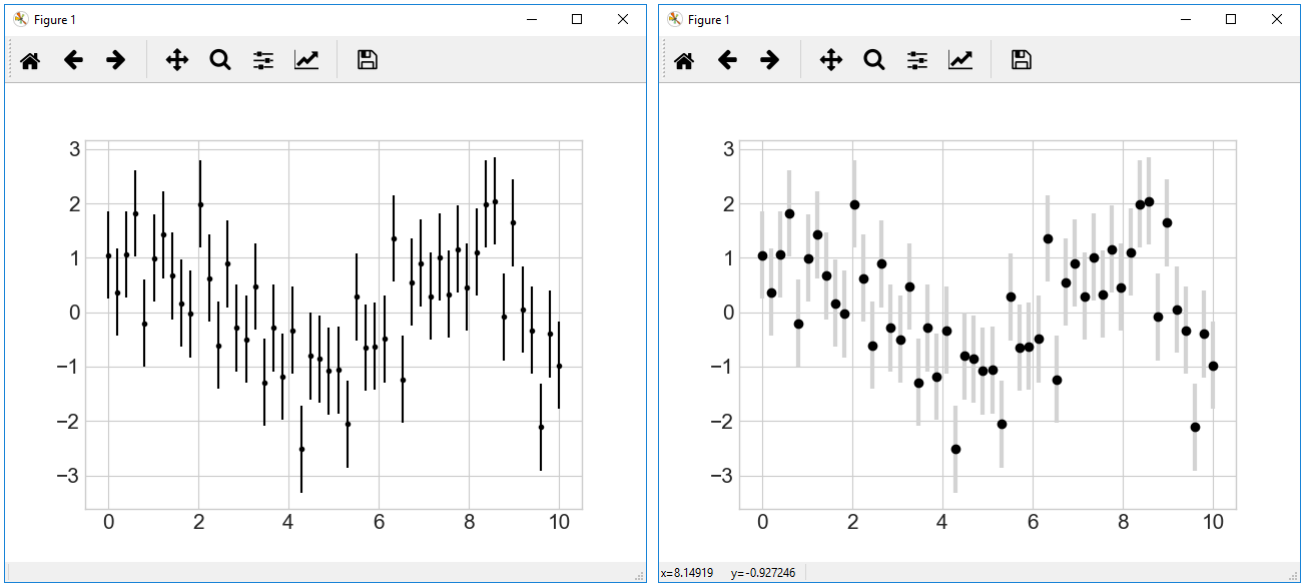
Visualiser l'incertitude

Dans la vie réelle, les données que nous sommes amenés à analyser sont souvent bruitées, c'est-à-dire qu'il existe une part d'incertitude sur leur valeur réelle. Il est extrêmement important d'en tenir compte non seulement lors de l'analyse des données, mais aussi quand on veut les présenter.

Données discrètes

Dans le cas de données discrètes (des points), nous utilisons souvent les barres d'erreur pour représenter, pour chaque point, l'incertitude quant à sa valeur exacte. Souvent la longueur des barres correspond à l'écart type des observations empiriques. C'est chose aisée avec **`Matplotlib`**.

```
>>> x = np.linspace(0, 10, 50)
>>> dy = 0.8
>>> y = np.sin(x) + dy * np.random.randn(50)
>>> plt.errorbar(x, y, yerr=dy, fmt='.k');
<Container object of 3 artists>
>>> plt.show()
```



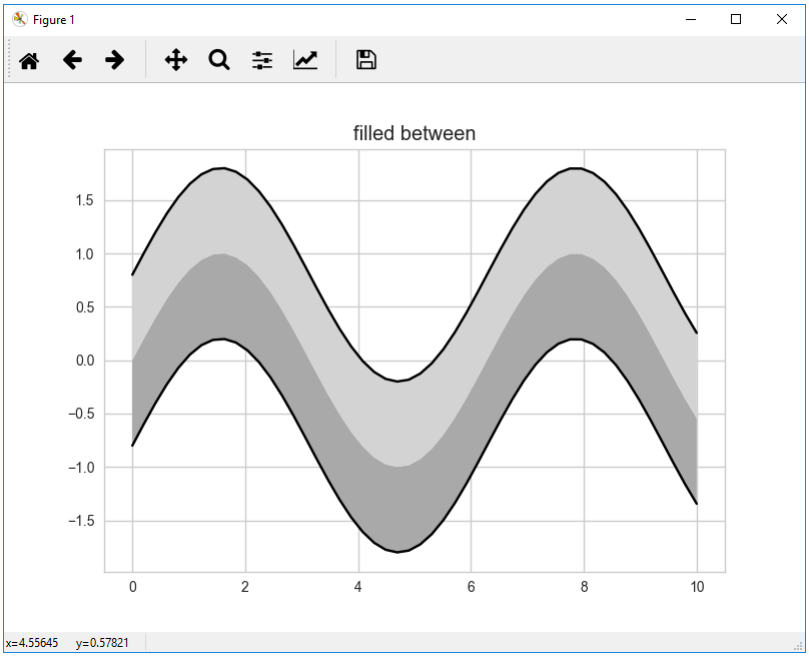
`Errorbar` prend en argument les abscisses **`x`**, les coordonnées **`y`** et les longueurs de chaque barre (une barre par point) **`yerr`**. Notez l'argument **`fmt`**. Il permet de choisir, de façon courte, la couleur (ici noir ou black) et la forme des marqueurs du graphe. **`Errorbar`** permet aussi de personnaliser encore plus l'apparence du graphe.

```
>>> plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
...         ecolor='lightgray', elinewidth=3, capsize=0);
<Container object of 3 artists>
>>> plt.show()
```

Données continues

Parfois, comme quand on essaie d'appliquer la régression par processus gaussien, nous avons besoin de représenter une incertitude sur une fonction continue. On peut faire ceci en utilisant la fonction **`plot`** conjointement avec la fonction **`fill_between`**.

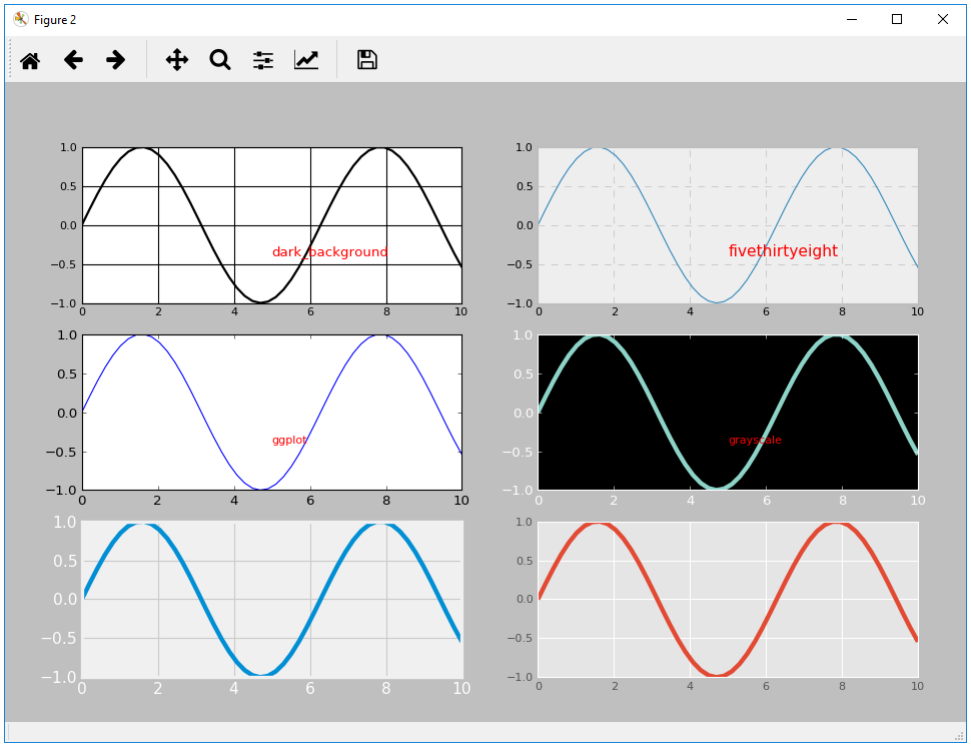
```
>>> x = np.linspace(0, 10, 50)
>>> dy = 0.8
>>> y = np.sin(x)
>>> y1 = y - dy
>>> y2 = y + dy
>>> plt.plot(x, y1, x, y2, color='black')
[<matplotlib.lines.Line2D object at 0x0000022887D49278>,
<matplotlib.lines.Line2D object at 0x0000022887D49BE0>]
>>> plt.fill_between(x, y, y1, where=y>=y1,
... facecolor='darkgray',interpolate=True)
<matplotlib.collections.PolyCollection object at 0x0000022887F24E80>
>>> plt.fill_between(x, y, y2, where=y<=y2,
... facecolor='lightgray',interpolate=True)
<matplotlib.collections.PolyCollection object at 0x0000022887DA2BE0>
>>> plt.title("filled between")
<matplotlib.text.Text object at 0x0000022887E5BF98>
>>> plt.show()
```



La personnalisation et sous-graphes

Matplotlib est très flexible. Quasiment tous les aspects d'une figure peuvent être configurés par l'utilisateur soit pour y ajouter des données, soit pour améliorer l'aspect esthétique. Plutôt que de vous faire une liste des fonctions qui permettent de faire ces actions, j'ai plutôt décidé de vous montrer des exemples. A l'avenir, n'hésitez pas à revenir vers cette partie pour vous remémorer comment réaliser une opération spécifique.

```
>>> print(plt.style.available[:6])
['bmh', 'classic', 'dark_background', 'fivethirtyeight', 'ggplot',
'grayscale']
>>> # Notez la taille de la figure
... fig = plt.figure(figsize=(12,8))
>>> for i in range(6):
...     # On peut ajouter des sous graphes ainsi
...     fig.add_subplot(3,2,i+1)
...     plt.style.use(plt.style.available[i])
...     plt.plot(x, y)
...     # Pour ajouter du texte
...     plt.text(s=plt.style.available[i], x=5, y=2, color='red')
...
>>> plt.show()
```

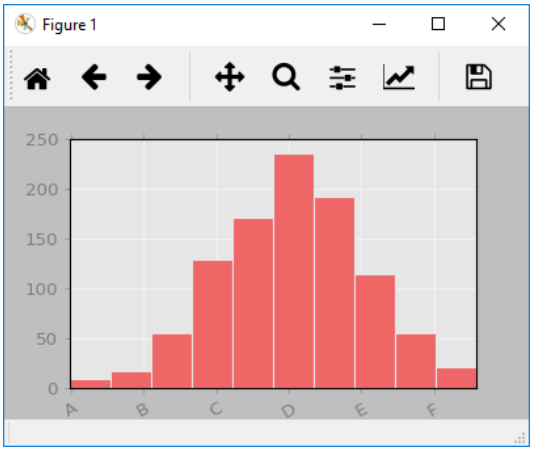


Le premier argument de la fonction **add_subplot** est le nombre de lignes de notre tableau de graphes 3. Le deuxième est le nombre de colonnes 2. Le troisième est le numéro du graphe, parmi les graphes de ce tableau, que nous voulons dessiner.

Pour des raisons historiques, les sous-graphes sont numérotés à partir de 1, au lieu de 0. Le graphe tout en haut à gauche est donc le graphe numéro 1.

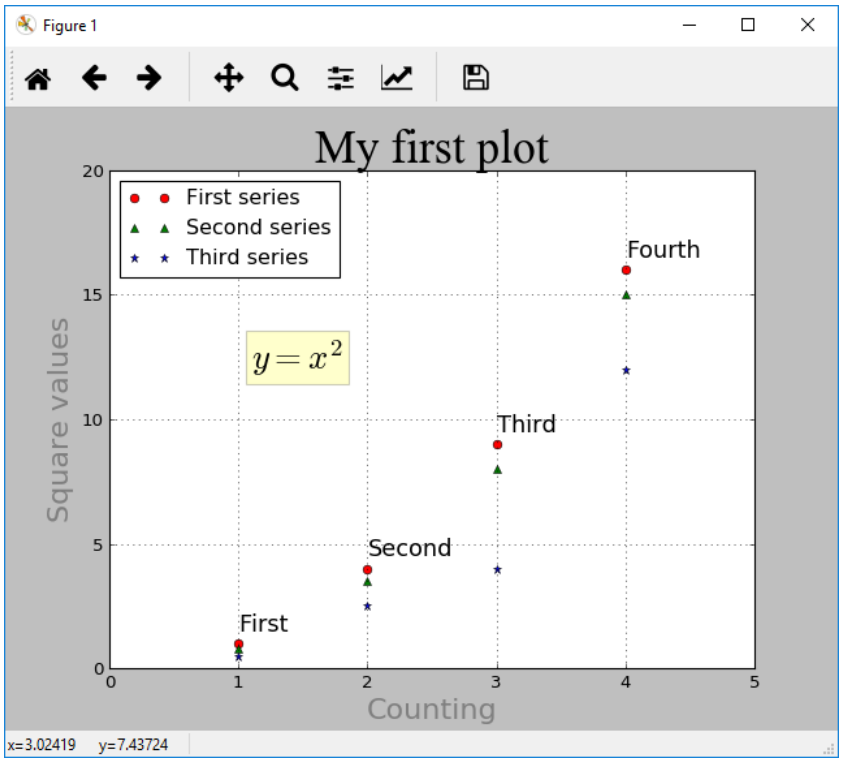
Nous pouvons aussi tout personnaliser à la main.

```
>>> # On peut aussi tout personnaliser à la main
... x = np.random.randn(1000)
>>> plt.style.use('classic')
>>> fig = plt.figure(figsize=(5,3))
>>> ax = plt.axes(facecolor='#E6E6E6')
>>> # Afficher les ticks en dessous de l'axe
>>> ax.set_axisbelow(True)
>>> # Cadre en blanc
>>> plt.grid(color='w', linestyle='solid')
>>> # Cacher le cadre
>>> # ax.spines contient les lignes qui entourent la zone où les
>>> # données sont affichées.
>>> for spine in ax.spines.values():
...     spine.set_visible(False)
...
>>> # Cacher les marqueurs en haut et à droite
>>> ax.xaxis.tick_bottom()
>>> ax.yaxis.tick_left()
>>> # Nous pouvons personnaliser les étiquettes des marqueurs
>>> # et leur appliquer une rotation
>>> marqueurs = [-3, -2, -1, 0, 1, 2, 3]
>>> xtick_labels = ['A', 'B', 'C', 'D', 'E', 'F']
>>> plt.xticks(marqueurs, xtick_labels, rotation=30)
>>> # Changer les couleur des marqueurs
>>> ax.tick_params(colors='gray', direction='out')
>>> for tick in ax.get_xticklabels():
...     tick.set_color('gray')
...
>>> for tick in ax.get_yticklabels():
...     tick.set_color('gray')
...
>>> # Changer les couleur des barres
>>> ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');
>>> plt.show()
```

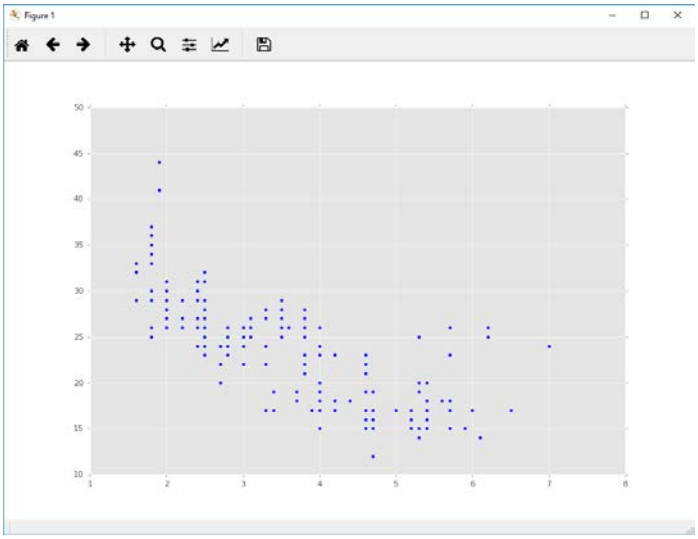



```
>>> plt.axis([0,5,0,20])
[0, 5, 0, 20]
>>> plt.title('My first plot',fontsize=32,fontname='Times New
Roman')
```

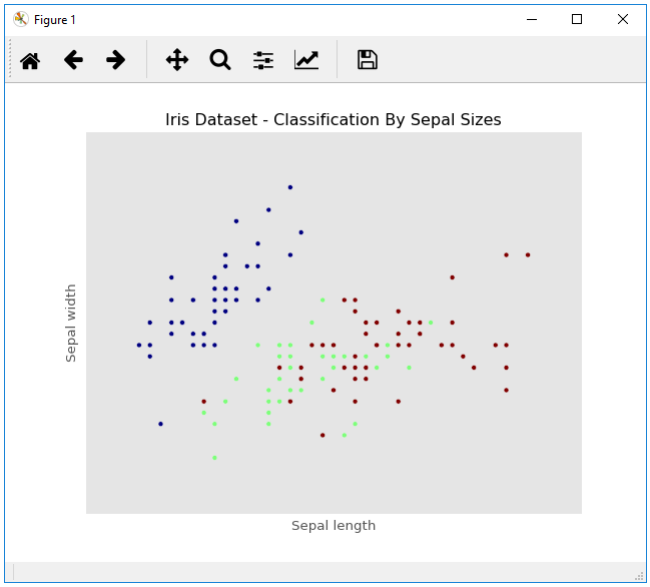
```
>>> plt.xlabel('Counting',color='gray',fontsize=20)
>>> plt.ylabel('Square values',color='gray',fontsize=20)
>>> plt.text(1,1.5,'First',fontsize=16)
>>> plt.text(2,4.5,'Second',fontsize=16)
>>> plt.text(3,9.5,'Third',fontsize=16)
>>> plt.text(4,16.5,'Fourth',fontsize=16)
>>> plt.text(1.1,12,r'$y =
x^2$',fontsize=24,bbox={'facecolor':'yellow','alpha':0.2})
>>> plt.grid(True)
>>> plt.plot([1,2,3,4],[1,4,9,16],'ro')
>>> plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
>>> plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
>>> plt.legend(['First series','Second series','Third
series'],loc=2)
>>> plt.show()
```



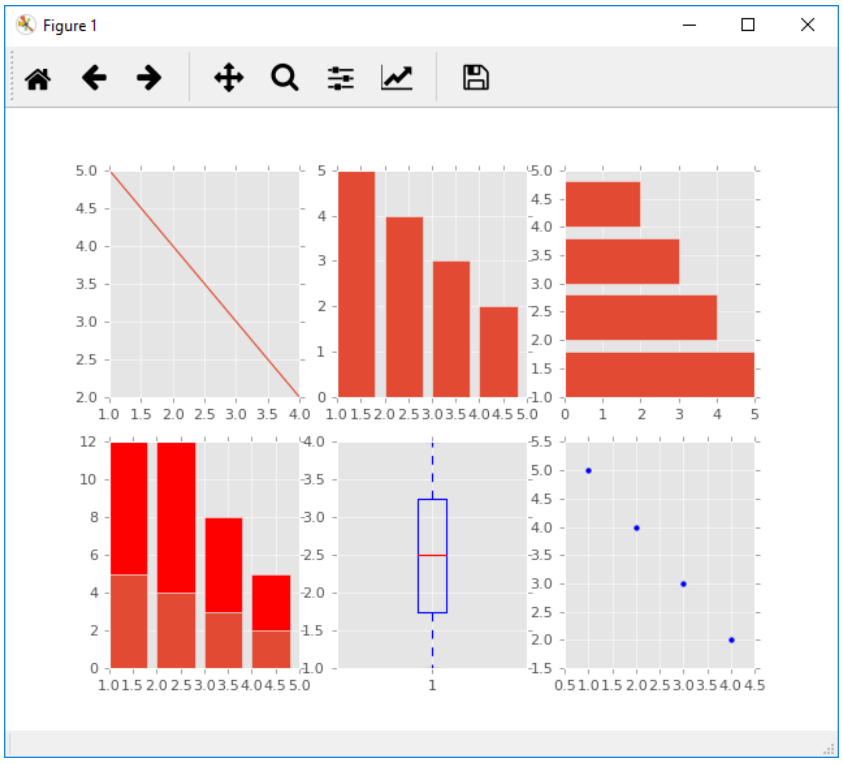

```
>>> df = pd.read_csv('donnees/mpg.csv')
>>> df.head()
  manufacturer model  displ  year  cyl       trans drv  cty  hwy fl    class
1         audi    a4    1.8  1999    4    auto(l5)   f   18   29  p  compact
2         audi    a4    1.8  1999    4  manual(m5)   f   21   29  p  compact
3         audi    a4    2.0  2008    4  manual(m6)   f   20   31  p  compact
4         audi    a4    2.0  2008    4    auto(av)   f   21   30  p  compact
5         audi    a4    2.8  1999    6    auto(l5)   f   16   26  p  compact
>>> plt.style.use('ggplot')
>>> plt.figure(figsize=(12, 8))
<matplotlib.figure.Figure object at 0x0000022889C647F0>
>>> plt.scatter(df.displ,df.hwy)
<matplotlib.collections.PathCollection object at 0x0000022887D68D30>
>>> plt.show()
```



```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> x = iris.data[:,0] #X-Axis - sepal length
>>> y = iris.data[:,1] #Y-Axis - sepal length
>>> species = iris.target #Species
>>> x_min, x_max = x.min() - .5,x.max() + .5
>>> y_min, y_max = y.min() - .5,y.max() + .5
>>> plt.figure()
>>> plt.title('Iris Dataset - Classification By Sepal Sizes')
>>> plt.scatter(x,y, c=species)
>>> plt.xlabel('Sepal length')
>>> plt.ylabel('Sepal width')
>>> plt.xlim(x_min, x_max)
(3.7999999999999998, 8.4000000000000004)
>>> plt.ylim(y_min, y_max)
(1.5, 4.9000000000000004)
>>> plt.xticks(())
([], <a list of 0 Text xticklabel objects>)
>>> plt.yticks(())
([], <a list of 0 Text yticklabel objects>)
>>> plt.show()
```

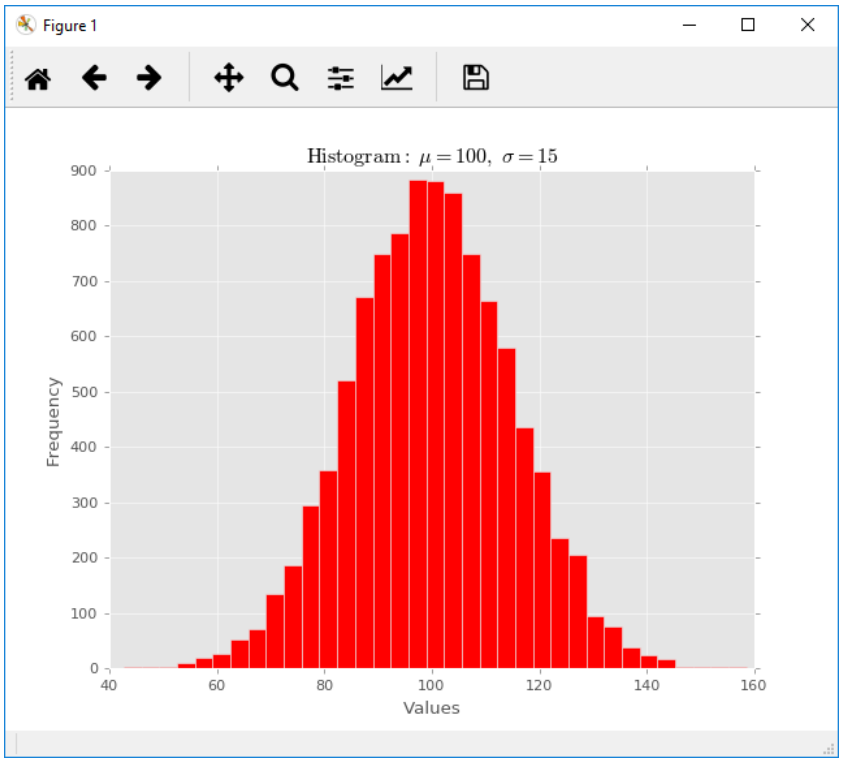


```
>>> from matplotlib.pyplot import *
>>> x = [1,2,3,4]
>>> y = [5,4,3,2]
>>> figure()
>>> subplot(231)
>>> plot(x, y)
>>> subplot(232)
>>> bar(x, y)
>>> subplot(233)
>>> barh(x, y)
>>> subplot(234)
>>> bar(x, y)
>>> y1 = [7,8,5,3]
>>> bar(x, y1, bottom=y, color = 'r')
>>> subplot(235)
>>> boxplot(x)
>>> subplot(236)
>>> scatter(x,y)
>>> show()
```

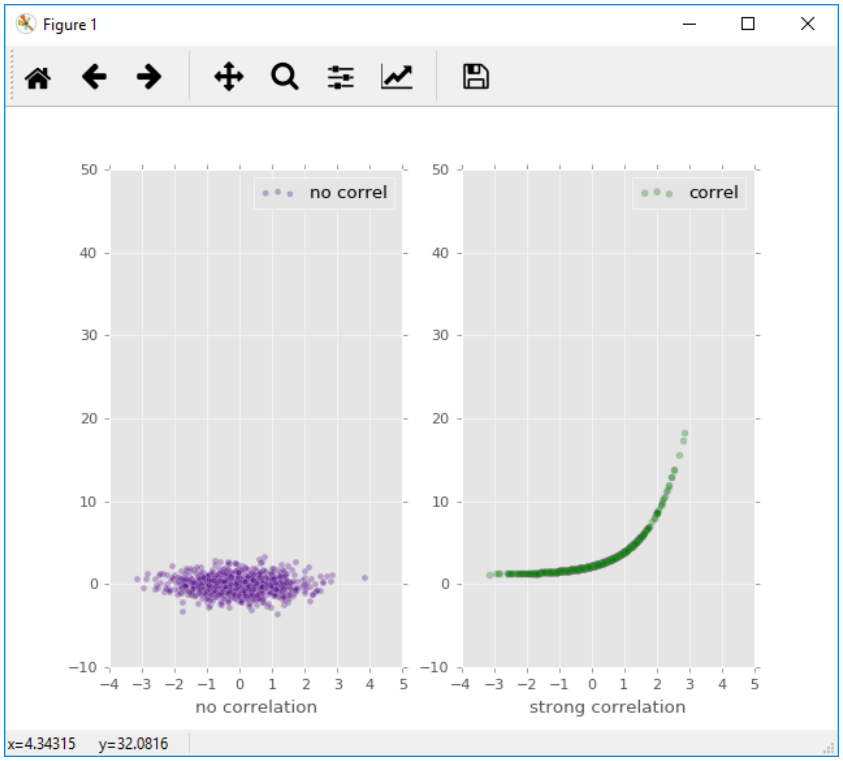



```
>>> mu = 100
>>> sigma = 15
>>> x = np.random.normal(mu, sigma, 10000)
>>> ax = plt.gca()
>>> ax.hist(x, bins=35, color='r')
(array([   2.,    2.,    3.,   10.,   19.,   27.,   53.,   71.,  134.,
        187.,  295.,  358.,  520.,  672.,  748.,  786.,  884.,  881.,
        860.,  749.,  664.,  580.,  437.,  357.,  237.,  205.,   94.,
         75.,   38.,   24.,   16.,    2.,    3.,    3.,    4.]), array([
42.68588348,   45.99367319,   49.3014629 ,   52.6092526 ,
        55.91704231,   59.22483202,   62.53262173,   65.84041143,
        69.14820114,   72.45599085,   75.76378056,   79.07157027,
        82.37935997,   85.68714968,   88.99493939,   92.3027291 ,
```

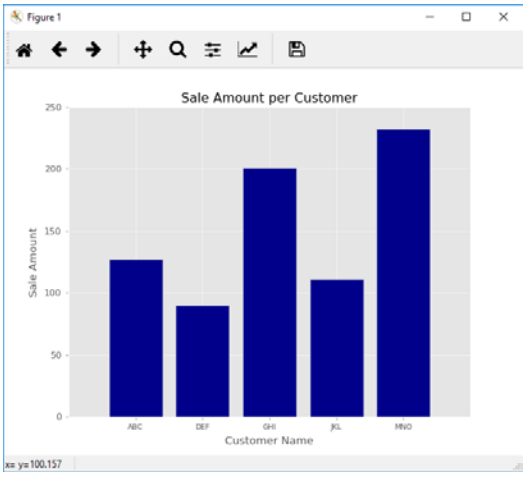
```
         95.6105188 ,   98.91830851,  102.22609822,  105.53388793,
        108.84167764,  112.14946734,  115.45725705,  118.76504676,
        122.07283647,  125.38062617,  128.68841588,  131.99620559,
        135.3039953 ,  138.611785  ,  141.91957471,  145.22736442,
        148.53515413,  151.84294384,  155.15073354,  158.45852325]), <a list
of 35 Patch objects>)
>>> ax.set_xlabel('Values')
>>> ax.set_ylabel('Frequency')
>>> ax.set_title(r'$\mathrm{Histogram:}\ \mu=%d,\ \sigma=%d$' % (mu,
... sigma))
>>> plt.show()
```



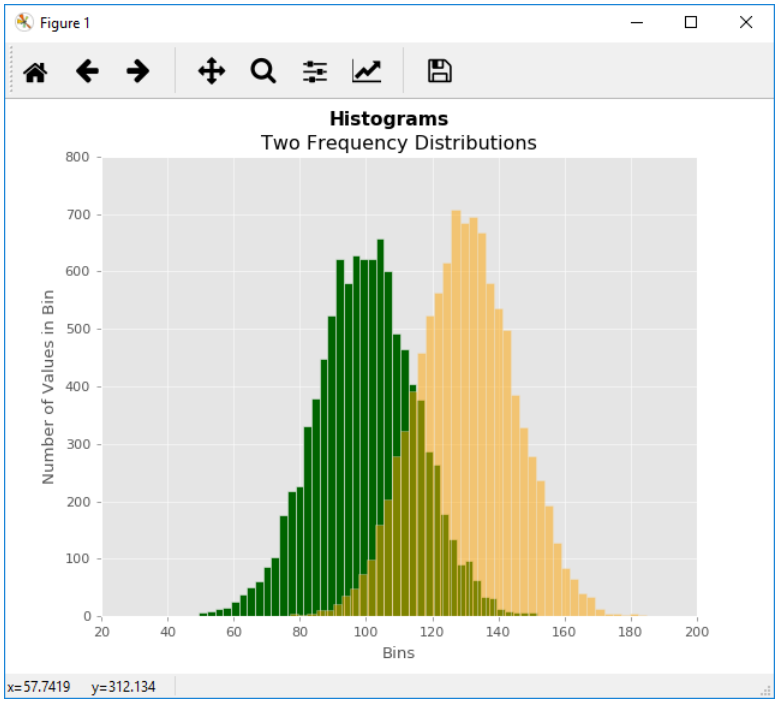

```
>>> x = np.random.randn(1000)
>>> y1 = np.random.randn(len(x))
>>> y2 = 1.2 + np.exp(x)
>>> ax1 = plt.subplot(121)
>>> plt.scatter(x, y1, color='indigo', alpha=0.3,
...             edgecolors='white',label='no correl')
>>> plt.xlabel('no correlation')
>>> plt.grid(True)
>>> plt.legend()
>>> ax2 = plt.subplot(122, sharey=ax1, sharex=ax1)
>>> plt.scatter(x, y2, color='green', alpha=0.3, edgecolors='grey',
... label='correl')
>>> plt.xlabel('strong correlation')
>>> plt.grid(True)
>>> plt.legend()
>>> plt.show()
```




```
>>> plt.style.use('ggplot')
>>> customers = ['ABC', 'DEF', 'GHI', 'JKL', 'MNO']
>>> customers_index = range(len(customers))
>>> sale_amounts = [127, 90, 201, 111, 232]
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1,1,1)
>>> ax1.bar(customers_index, sale_amounts, align='center',
color='darkblue')
>>> ax1.xaxis.set_ticks_position('bottom')
>>> ax1.yaxis.set_ticks_position('left')
>>> plt.xticks(customers_index, customers, rotation=0,
fontsize='small')
>>> plt.xlabel('Customer Name')
>>> plt.ylabel('Sale Amount')
>>> plt.title('Sale Amount per Customer')
>>> plt.savefig('bar_plot.png', dpi=400, bbox_inches='tight')
>>> plt.show()
```

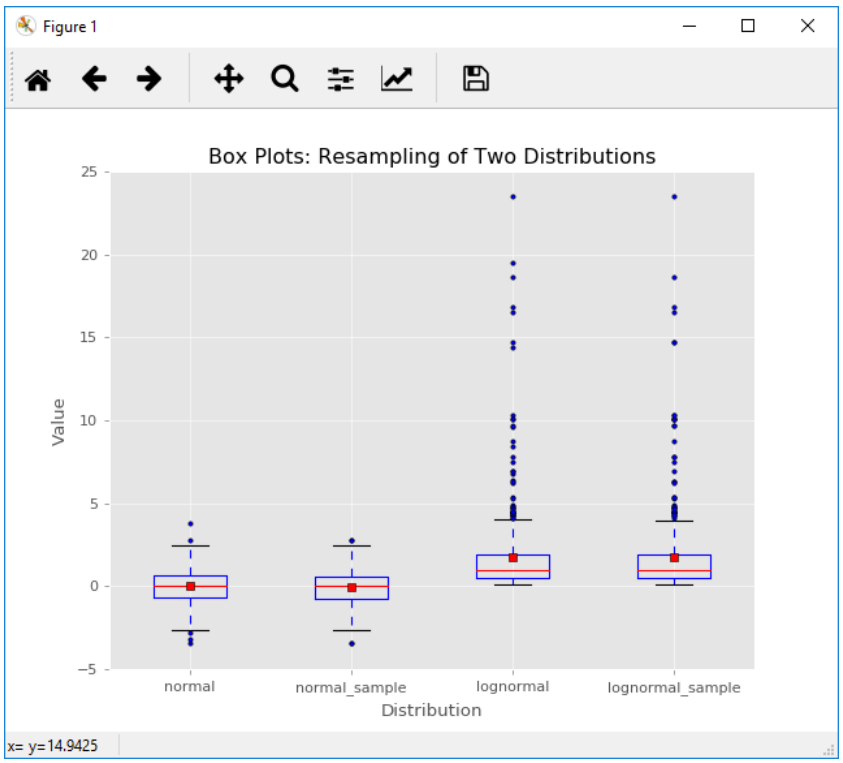


```
>>> plt.style.use('ggplot')
>>> mu1, mu2, sigma = 100, 130, 15
>>> x1 = mu1 + sigma*np.random.randn(10000)
>>> x2 = mu2 + sigma*np.random.randn(10000)
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1,1,1)
>>> n, bins, patches = ax1.hist(x1, bins=50, normed=False,
color='darkgreen')
>>> n, bins, patches = ax1.hist(x2, bins=50, normed=False,
color='orange', alpha=0.5)
>>> ax1.xaxis.set_ticks_position('bottom')
>>> ax1.yaxis.set_ticks_position('left')
>>> plt.xlabel('Bins')
>>> plt.ylabel('Number of Values in Bin')
>>> fig.suptitle('Histograms', fontsize=14, fontweight='bold')
>>> ax1.set_title('Two Frequency Distributions')
>>> plt.savefig('histogram.png', dpi=400, bbox_inches='tight')
>>> plt.show()
```




```
>>> N = 500
>>> normal = np.random.normal(loc=0.0, scale=1.0, size=N)
>>> lognormal = np.random.lognormal(mean=0.0, sigma=1.0, size=N)
>>> index_value = np.random.random_integers(low=0, high=N-1, size=N)
>>> normal_sample = normal[index_value]
>>> lognormal_sample = lognormal[index_value]
>>> box_plot_data =
[normal,normal_sample,lognormal,lognormal_sample]
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1,1,1)
>>> box_labels =
['normal','normal_sample','lognormal','lognormal_sample']
>>> ax1.boxplot(box_plot_data, notch=False, sym='.', vert=True, \
... whis=1.5,showmeans=True, labels=box_labels)
>>> ax1.xaxis.set_ticks_position('bottom')
```

```
>>> ax1.yaxis.set_ticks_position('left')
>>> ax1.set_title('Box Plots: Resampling of Two Distributions')
>>> ax1.set_xlabel('Distribution')
>>> ax1.set_ylabel('Value')
>>> plt.savefig('box_plot.png', dpi=400, bbox_inches='tight')
>>> plt.show()
```



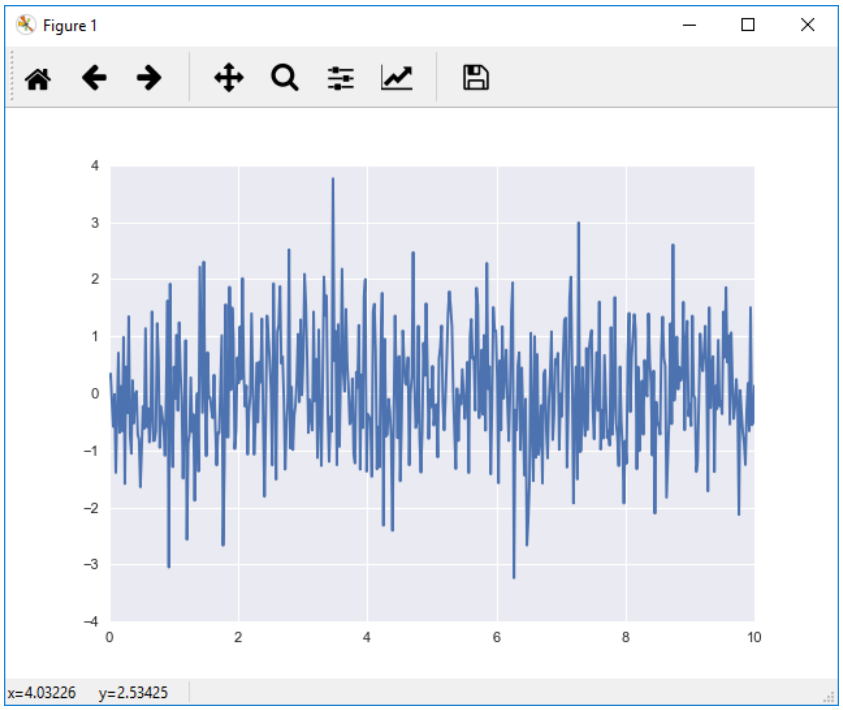
La librairie Seaborn

Seaborn est une librairie qui vient s'ajouter à **Matplotlib**, remplace certains réglages par défaut et fonctions, et lui ajoute de nouvelles fonctionnalités. **Seaborn** vient corriger trois défauts de **Matplotlib** :

- **Matplotlib**, surtout dans les versions avant la 2.0, ne génère pas des graphiques d'une grande qualité esthétique.
- **Matplotlib** ne possède pas de fonctions permettant de créer facilement des analyses statistiques sophistiquées.
- Les fonctions de **Matplotlib** ne sont pas faites pour interagir avec les **Dataframes** de **Panda**.

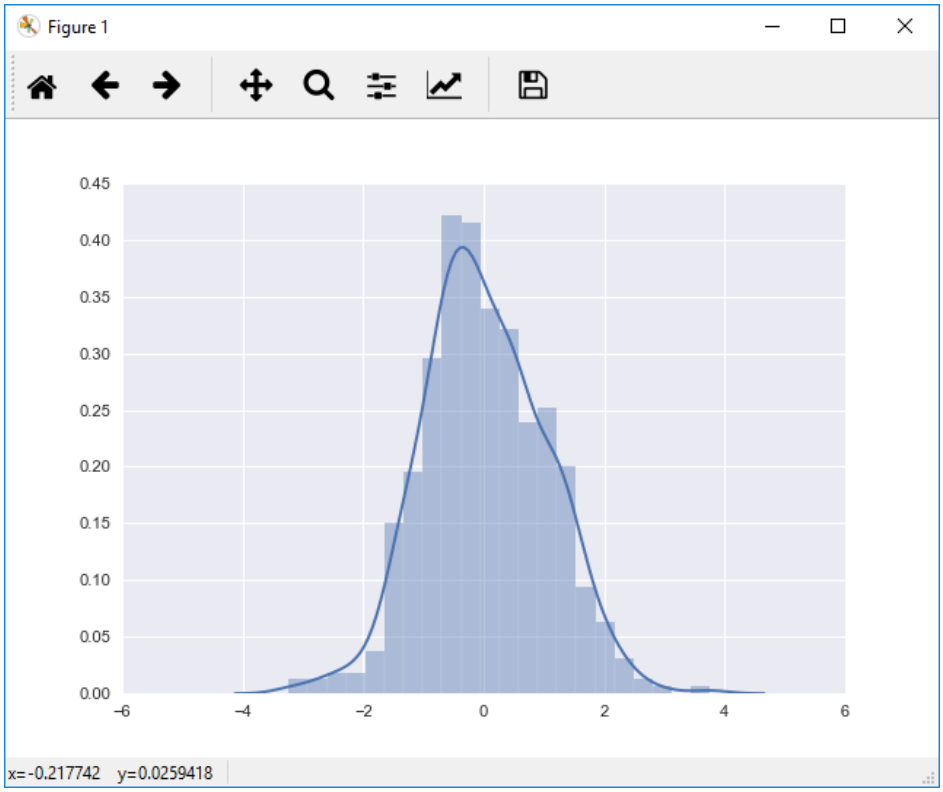
Seaborn fournit une interface qui permet de palier ces problèmes. Il utilise toujours **Matplotlib**, mais le fait en exposant des fonctions plus intuitives. Pour commencer à l'utiliser, rien de plus simple.

```
>>> import seaborn as sns
>>> sns.set()
>>> x = np.linspace(0, 10, 500)
>>> y = np.random.randn(500)
>>> plt.plot(x,y)
>>> plt.show()
```

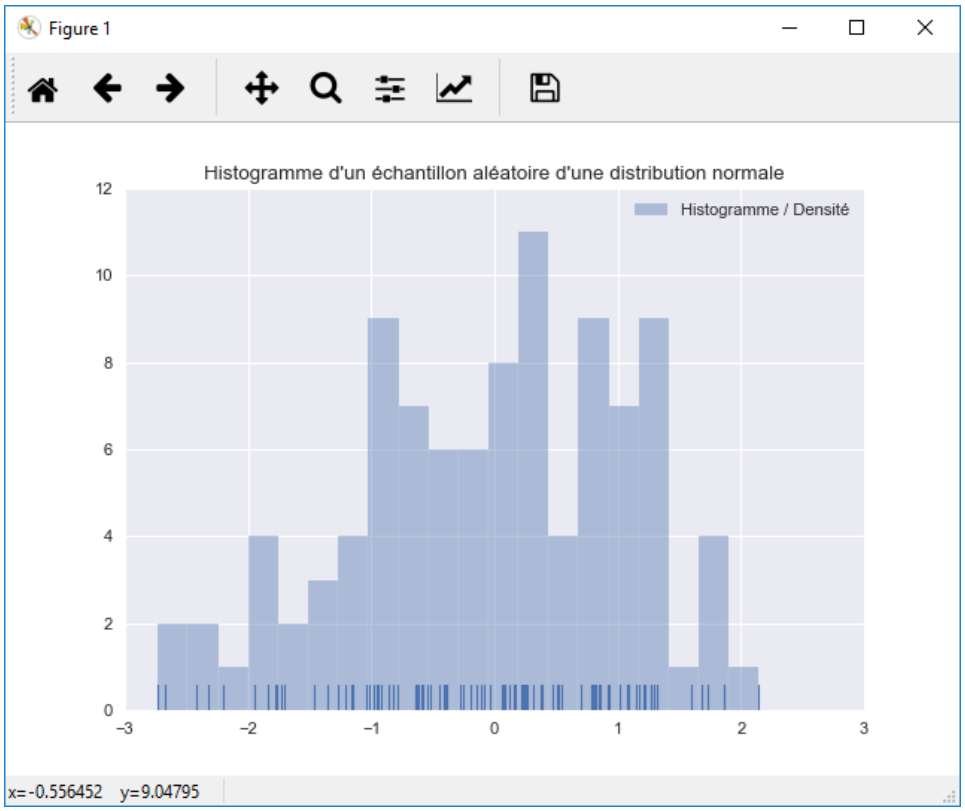


Seaborn nous fournit aussi des fonctions pour des graphiques utiles pour l'analyse statistique. Par exemple, la fonction **distplot** permet non seulement de visualiser l'histogramme d'un échantillon, mais aussi d'estimer la distribution dont l'échantillon est issu.

```
>>> sns.distplot(y, kde=True);
>>> plt.show()
```




```
>>> sns.set(color_codes=True)
>>> x = np.random.normal(size=100)
>>> sns.distplot(x, bins=20, kde=False, rug=True, label="Histogramme
/ Densité")
>>> plt.title("Histogramme d'un échantillon aléatoire d'une
distribution normale")
>>> plt.legend()
>>> plt.show()
```



Imaginons que nous voulons travailler sur un ensemble de données provenant du jeu de données **« Iris »,** qui contient des mesures de la longueur et la largeur des sépales et des pétales de trois espèces d'iris.

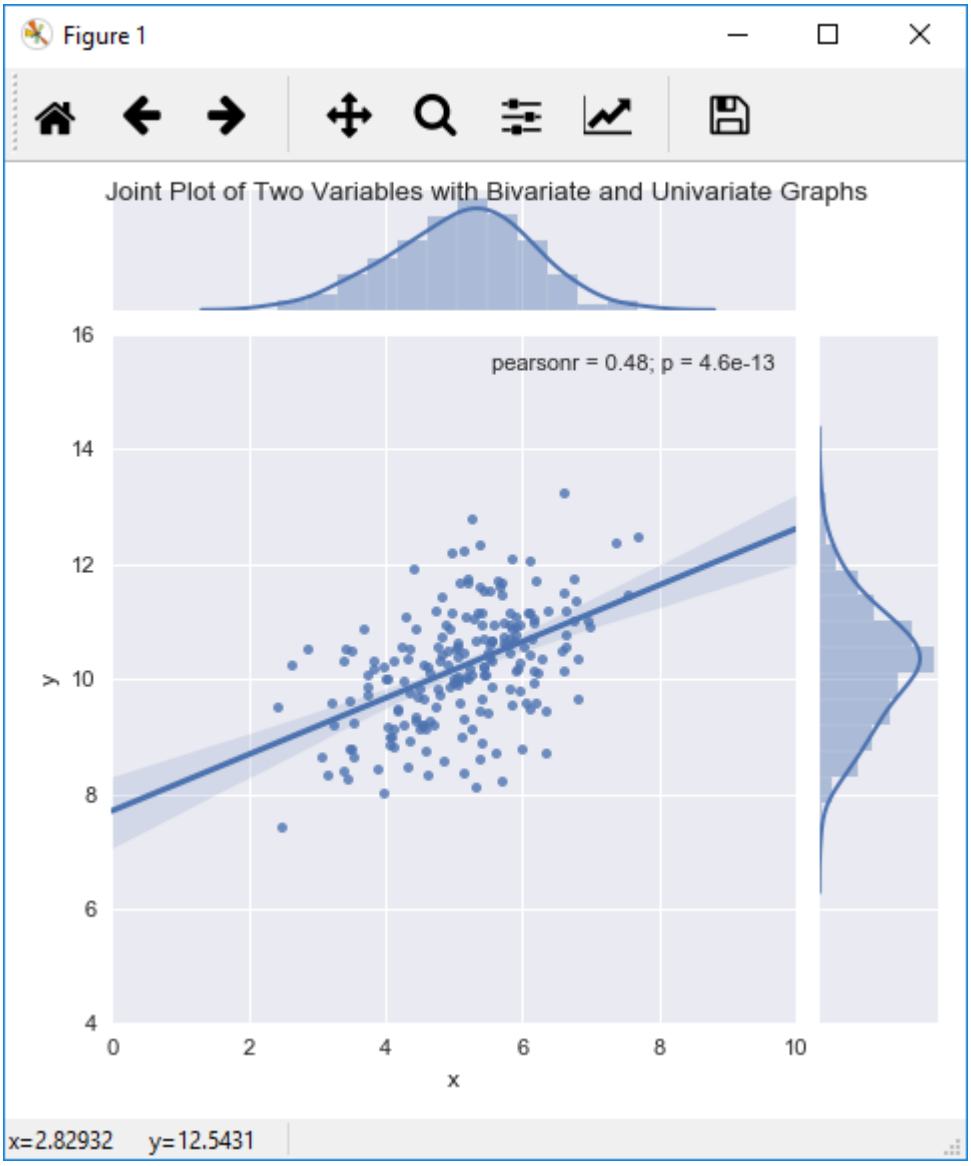
```
>>> iris = pd.read_csv('donnees/Iris.csv')
>>> sns.pairplot(iris)
>>> sns.pairplot(iris, hue='Species', size=2.5);
>>> plt.show()
```

La diagonale est traitée différemment, car tracer une variable en fonction d'elle-même n'aurait aucun intérêt. À la place, **`sns.pairplot`** trace un histogramme des données en fonction de la variable en question pour chaque classe de données.

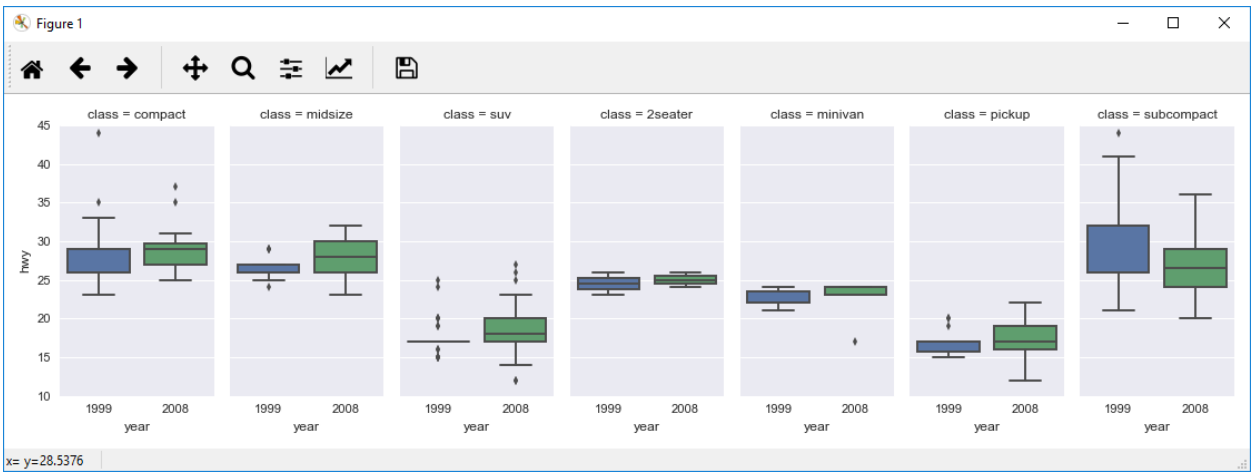
Nous pouvons aussi voir la distribution jointe de deux caractéristiques :

```
>>> mean, cov = [5, 10], [(1, .5), (.5, 1)]
>>> data = np.random.multivariate_normal(mean, cov, 200)
>>> data_frame = pd.DataFrame(data, columns=["x", "y"])
>>> sns.jointplot(x="x", y="y", data=data_frame, kind="reg")\
...                 .set_axis_labels("x", "y")
>>> plt.suptitle("Joint Plot of Two Variables with Bivariate and
Univariate Graphs")
```

```
>>> plt.show()
```




```
>>> mpg = pd.read_csv('donnees/mpg.csv')
>>> sns.factorplot(x="year", y="hwy", \
...         col="class", data=mpg, kind="box",size=4, aspect=.5)
>>> plt.show()
```



4

Les DataFrames

Les structures de données

La librairie **Pandas** fournit deux structures de données fondamentales, la **« Series »** et le **« DataFrame »**. On peut voir ces structures comme une généralisation des tableaux et des matrices de **Numpy**. La différence fondamentale entre ces structures et les versions de **Numpy** est que les objets **Pandas** possèdent des indices explicites. Là où on ne pouvait se référer à un élément d'un tableau **Numpy** que par sa position dans le tableau, chaque élément d'une **« Series »** ou d'un **« DataFrame »** peut avoir un indice explicitement désigné par l'utilisateur.

L'indice explicite est optionnel. On peut très bien utiliser une **« Series »** par exemple comme on utiliserait un tableau **« Numpy »**, en se contentant des indices générés automatiquement en fonction de la position de chaque élément.

Commençons par voir comment créer ces structures et nous en servir pour quelques opérations de base.

```
>>> import numpy as np
>>> import pandas as pd
>>> # On peut créer une Series à partir d'une list
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> print("data ressemble à un tableau Numpy: ", data)
data ressemble à un tableau Numpy:  0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
>>> # On peut spécifier des indices à la main
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0],
...          index=['a', 'b', 'c', 'd'])
>>> print("data ressemble à un dict en Python: ", data)
data ressemble à un dict en Python:  a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
>>> print(data['b'])
0.5
>>> # On peut même créer une Serie directement à partir d'une dict
... population_dict = {'California': 38332521,
...                    'Texas': 26448193,
...                    'New York': 19651127,
...                    'Florida': 19552860,
...                    'Illinois': 12882135}
>>> area_dict = {'California': 423967,
...              'Texas': 695662,
...              'New York': 141297,
...              'Florida': 170312,
...              'Illinois': 149995}
>>> population = pd.Series(population_dict)
>>> area = pd.Series(area_dict)
>>> print(population)
California    38332521
```

```
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
dtype: int64
>>> # Que pensez vous de cette ligne?
>>> print(population['California':'Florida'])
California    38332521
Florida       19552860
dtype: int64
```

De la même façon que les opérations sur les tableaux **Numpy** sont plus rapides que celles sur les **« list »** en Python, les opérations sur les **« Series »** sont plus rapides que celles sur les **« dict »**.

Les **« DataFrame »** permettent de combiner plusieurs **« Series »** en colonnes, un peu comme dans un tableau SQL.

```
>>> # A partir d'une Series
>>> df = pd.DataFrame(population, columns=['population'])
>>> print(df)
            population
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
>>> # A partir d'une list de dict
>>> data = [{'a': i, 'b': 2 * i}
...         for i in range(3)]
>>> df = pd.DataFrame(data)
>>> print(df)
   a  b
0  0  0
1  1  2
2  2  4
>>> # A partir de plusieurs Series
>>> df = pd.DataFrame({'population': population, 'area': area})
>>> print(df)
              area  population
California  423967    38332521
Florida     170312    19552860
Illinois    149995    12882135
New York    141297    19651127
Texas       695662    26448193
>>> # A partir d'un tableau Numpy de dimension 2
>>> df = pd.DataFrame(np.random.rand(3, 2),
...              columns=['foo', 'bar'],
...              index=['a', 'b', 'c'])
>>> print(df)
        foo       bar
a  0.379015  0.789917
b  0.713045  0.660162
c  0.527456  0.634284
>>> # Une fonction pour générer facilement des DataFrame.
>>> # Elle nous sera utile dans la suite de ce chapitre.
```

```
>>> def make_df(cols, ind):
...     """Crée rapidement des DataFrame"""
...     data = {c: [str(c) + str(i) for i in ind]
...             for c in cols}
...     return pd.DataFrame(data, ind)
...
>>> # exemple
>>> make_df('ABC', range(3))
    A   B   C
0  A0  B0  C0
1  A1  B1  C1
2  A2  B2  C2
```

```
>>> raw_data = {'first_name': ['Jason', 'Molly', 'Tina',
...                 'Jake', 'Amy'],
...         'last_name': ['Miller', 'Jacobson', ".",
...                 'Milner', 'Cooze'],
...         'age': [42, 52, 36, 24, 73],
...         'preTestScore': [4, 24, 31, ".", "."],
...         'postTestScore': ["25,000", "94,000", 57, 62, 70]}
>>> df = pd.DataFrame(raw_data, columns = ['first_name',
...        'last_name','age','preTestScore','postTestScore'])
>>> df
  first_name last_name  age preTestScore postTestScore
0      Jason    Miller   42            4        25,000
1      Molly  Jacobson   52           24        94,000
2       Tina         .   36           31            57
3       Jake    Milner   24            .            62
4        Amy     Cooze   73            .            70
```

Lecture et écriture de DataFrame

Aujourd'hui, on n'a plus besoin de réécrire soi-même une fonction de lecture ou d'écriture de données présentées sous forme de tables. Il existe des fonctions plus génériques qui gère un grand nombre de cas.

```
>>> import pandas as pd
>>> l = [ { "date":"2017-06-22", "prix":220.0, "devise":"euros" },
...       { "date":"2017-06-23", "prix":221.0, "devise":"euros" },]
>>> df = pd.DataFrame(l)
>>> # écriture au format texte
>>> df.to_csv("donnees/exemple.txt",sep="\t",
...            encoding="utf-8", index=False)
>>> # on regarde ce qui a été enregistré
... with open("donnees/exemple.txt", "r", encoding="utf-8") as f:
...     text = f.read()
...
>>> print(text)
date    devise  prix
2017-06-22      euros   220.0
2017-06-23      euros   221.0

>>> # on enregistre au format Excel
>>> df.to_excel("donnees/exemple.xlsx", index=False)
```

La librairie **Pandas** fournis un ensemble de fonctions de lecture écriture de haut niveau pour accéder aux fichiers. Le fonctions de lecture renvoient généralement un objet DataFrame.

Format Type	Data Description	Reader	Writer
text	CSV	**read_csv**	**to_csv**
text	JSON	**read_json**	**to_json**
text	HTML	**read_html**	**to_html**
text	Local clipboard	**read_clipboard**	**to_clipboard**
binary	MS Excel	**read_excel**	**to_excel**
binary	HDF5 Format	**read_hdf**	**to_hdf**
binary	Feather Format	**read_feather**	**to_feather**
binary	Parquet Format	**read_parquet**	**to_parquet**
binary	Msgpack	**read_msgpack**	**to_msgpack**
binary	Stata	**read_stata**	**to_stata**
binary	SAS	**read_sas**	
binary	Python Pickle Format	**read_pickle**	**to_pickle**
SQL	SQL	**read_sql**	**to_sql**
SQL	Google Big Query	read_gbq	to_gbq

```
>>> import pandas as pd
>>> villes = pd.read_csv("donnees/temperatures.csv",
...             index_col='Ville')
>>> villes.head()
               Janv  Févr  Mars   Avri    Mai   Juin   Juil   Août   Sept  \
Ville
Abbeville      4.27  4.96  7.27   9.94  13.17  15.89  17.73  17.95  15.47
Lille-Lesquin  3.80  4.73  7.40  10.55  14.09  16.95  18.77  18.74  15.90
Pte De La Hague  7.89  7.66  8.54  10.03  12.38  14.79  16.72  17.48  16.63
Caen-Carpiquet  5.38  5.94  7.80  10.02  13.21  16.19  17.94  18.18  15.85
Rouen-Boos     3.95  4.66  7.28   9.99  13.38  16.38  18.11  18.11  15.33

                Octo   Nove  Déce        Lat      Long  Alt    Moy    Amp  \
Ville
Abbeville      11.90   7.80  4.92  50.136000  1.834000   69  10.94  13.68
Lille-Lesquin  11.81   7.53  4.52  50.570000  3.097500   47  11.23  14.97
Pte De La Hague  14.38  11.31  8.96  49.725167 -1.939833    6  12.23   9.82
Caen-Carpiquet  12.59   8.75  5.96  49.180000 -0.456167   67  11.48  12.80
Rouen-Boos     11.60   7.40  4.46  49.383000  1.181667  151  10.89  14.16

              Zone
Ville
Abbeville       NO
Lille-Lesquin   NE
Pte De La Hague  NO
Caen-Carpiquet  NO
Rouen-Boos      NO
>>> villes.describe()
            Janv       Févr       Mars       Avri        Mai       Juin  \
count  42.000000  42.000000  42.000000  42.000000  42.000000  42.000000
mean    5.335714   5.959762   8.814524  11.577381  15.236190  18.755952
std     2.286357   2.015944   1.705363   1.614030   1.781203   2.179185
min     1.110000   1.610000   4.980000   7.970000  12.100000  14.790000
25%     3.837500   4.662500   7.425000  10.282500  14.107500  17.270000
50%     5.210000   5.860000   8.550000  11.225000  14.955000  18.525000
75%     7.397500   7.607500  10.167500  12.790000  16.567500  20.140000
max     9.270000   9.510000  12.010000  14.680000  18.880000  23.150000

            Juil       Août       Sept       Octo       Nove       Déce  \
count  42.000000  42.000000  42.000000  42.000000  42.000000  42.000000
mean   20.566905  20.473571  17.319286  13.578095   8.947381   5.977857
std     2.342368   2.248508   1.991061   2.079219   2.215485   2.279859
min    16.720000  16.990000  13.520000   9.890000   4.900000   1.830000
25%    19.172500  18.907500  15.905000  11.877500   7.545000   4.555000
50%    20.155000  20.115000  16.740000  13.075000   8.380000   5.530000
75%    21.607500  21.640000  18.437500  14.737500  10.535000   8.015000
max    25.320000  25.000000  21.350000  17.740000  13.350000  10.220000

             Lat       Long         Alt        Moy        Amp
count  42.000000  42.000000   42.000000  42.000000  42.000000
mean   46.251996   2.421921  174.476190  12.711667  15.316429
std     2.450608   3.419851  211.239459   1.836500   2.406873
min    41.918000  -4.412000    2.000000   9.120000   9.530000
25%    43.962000   0.027542   43.250000  11.440000  14.420000
50%    46.320333   2.372083  101.500000  12.205000  15.630000
75%    48.445167   4.991625  231.000000  13.715000  16.665000
max    50.570000   9.485167  871.000000  16.470000  19.180000
>>> villes.axes
[Index(['Abbeville', 'Lille-Lesquin', 'Pte De La Hague', 'Caen-Carpiquet',
       'Rouen-Boos', 'Reims-Prunay', 'Brest-Guipavas', 'Ploumanac'h',
       'Rennes-St Jacques', 'Alencon', 'Orly', 'Troyes-Barberey',
       'Nancy-Ochey', 'Strasbourg-Entzheim', 'Belle Ile-Le Talut',
       'Nantes-Bouguenais', 'Tours', 'Bourges', 'Dijon-Longvic',
       'Bale-Mulhouse', 'Pte De Chassiron', 'Poitiers-Biard',
```

```
       'Limoges-Bellegarde', 'Clermont-Fd', 'Le Puy-Loudes', 'Lyon-St Exupery',
       'Bordeaux-Merignac', 'Gourdon', 'Millau', 'Montelimar', 'Embrun',
       'Mont-De-Marsan', 'Tarbes-Ossun', 'St Girons', 'Toulouse-Blagnac',
       'Montpellier', 'Marignane', 'Cap Cepet', 'Nice', 'Perpignan', 'Ajaccio',
       'Bastia'],
      dtype='object', name='Ville'), Index(['Janv', 'Févr', 'Mars', 'Avri', 'Mai',
'Juin', 'Juil', 'Août', 'Sept',
       'Octo', 'Nove', 'Déce', 'Lat', 'Long', 'Alt', 'Moy', 'Amp', 'Zone'],
      dtype='object')]
>>> villes.dtypes
Janv    float64
Févr    float64
Mars    float64
Avri    float64
Mai     float64
Juin    float64
Juil    float64
Août    float64
Sept    float64
Octo    float64
Nove    float64
Déce    float64
Lat     float64
Long    float64
Alt       int64
Moy     float64
Amp     float64
Zone     object
dtype: object
```

Il est possible de sélectionner les colonnes qui doivent être chargées et changer les noms par défaut.

```
>>> commandes = pd.read_csv("donnees/stagiaire/commandes.csv",
...       sep=';',header=0,usecols=[0,1,2,3,4,5],
...       names=['No','Client','Employe','Commande','Envoi','Port'])
>>> commandes.head()
       No Client  Employe    Commande       Envoi  Port
0  215650  LONEP       84  2010-02-02  2010-03-08  50.1
1  215653  PERIC       78  2010-02-02  2010-03-14  97.6
2  215652  BOTTM       72  2010-02-02  2010-03-02  89.3
3  215674  SPECD      111  2010-02-02  2010-03-01  86.2
4  215672  WELLI       39  2010-02-02  2010-02-12  71.9
```

La projection et la restriction

L'algèbre relationnelle est une théorie permettant de manipuler des données disposées sous forme de tableau ; et ça tombe bien : un **« DataFrame »**, c'est justement un tableau !

On peut référer aux éléments des objets **Pandas** en utilisant soit leurs index implicites, de la même façon que les tableaux **Numpy**, soit les index explicites comme dans les **« dict »**. Pour éviter toute confusion, il est conseillé d'utiliser les attributs **« loc »** qui référence par l'index et **« iloc »** qui référence par la position de chaque objet.

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0],
...                  index=['a', 'b', 'c', 'd'])
>>> print(data)
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
>>> # On peut désigner un élément d'une Series par son index
... print(data.loc['b'])
0.5
>>> # Ou bien par sa position
... print(data.iloc[1])
0.5
```

La différence entre les deux devrait être claire après avoir exécuté ces lignes. Effectuer ces mêmes opérations sur les **DataFrame** se fait de manière analogue.

```
>>> data = pd.DataFrame({'area':area, 'pop':population})
>>> print(data)
              area       pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
New York    141297  19651127
Texas       695662  26448193
>>> data.loc[:'Illinois', :'pop']
              area       pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
```

Il est possible d'accéder à une colonne a l'aide de plusieurs syntaxes.

```
>>> villes.head(3)
               Janv  Févr  Mars   Avri    Mai   Juin   Juil   Août   Sept  \
Ville
Abbeville      4.27  4.96  7.27   9.94  13.17  15.89  17.73  17.95  15.47
Lille-Lesquin  3.80  4.73  7.40  10.55  14.09  16.95  18.77  18.74  15.90
Pte De La Hague  7.89  7.66  8.54  10.03  12.38  14.79  16.72  17.48  16.63

                Octo   Nove  Déce        Lat      Long  Alt    Moy    Amp  \
Ville
Abbeville      11.90   7.80  4.92  50.136000  1.834000   69  10.94  13.68
Lille-Lesquin  11.81   7.53  4.52  50.570000  3.097500   47  11.23  14.97
Pte De La Hague  14.38  11.31  8.96  49.725167 -1.939833    6  12.23   9.82
```

```
              Zone
Ville
Abbeville        NO
Lille-Lesquin    NE
Pte De La Hague  NO
>>> villes.head(3).Janv
Ville
Abbeville          4.27
Lille-Lesquin      3.80
Pte De La Hague    7.89
Name: Janv, dtype: float64
>>> villes.head(3)['Janv']
Ville
Abbeville          4.27
Lille-Lesquin      3.80
Pte De La Hague    7.89
Name: Janv, dtype: float64
```

L'objet que renvoie **`villes.head(3)['Janv']`** est de type **`pandas.Series`**. Pour obtenir les valeurs de la colonne **`Janv`** au format **`numpy`**, il faut saisir **`villes.head(3)['Janv'].values.`**

Accédons maintenant aux données de la ville Abbeville, d'abord par sa position 0, puis par son nom. Le résultat retourné est exactement le même dans les 2 cas.

```
>>> villes.iloc[0,0:3]
Janv    4.27
Févr    4.96
Mars    7.27
Name: Abbeville, dtype: object
>>> villes.loc[['Abbeville'],
...            ['Janv','Févr','Mars']]
           Janv  Févr  Mars
Ville
Abbeville  4.27  4.96  7.27

>>> villes.iloc[0:3,0:3]
                Janv  Févr  Mars
Ville
Abbeville       4.27  4.96  7.27
Lille-Lesquin   3.80  4.73  7.40
Pte De La Hague 7.89  7.66  8.54
```

On désigne généralement une colonne ou variable par son nom. Les lignes peuvent être désignées par un entier.

```
>>> import pandas as pd
>>> villes = pd.read_csv("donnees/temperatures.csv")
>>> villes.head().iloc[:,:5]
             Ville  Janv  Févr  Mars   Avri
0        Abbeville  4.27  4.96  7.27   9.94
1    Lille-Lesquin  3.80  4.73  7.40  10.55
2  Pte De La Hague  7.89  7.66  8.54  10.03
3   Caen-Carpiquet  5.38  5.94  7.80  10.02
4       Rouen-Boos  3.95  4.66  7.28   9.99
>>> villes.iloc[2]
Ville    Pte De La Hague
Janv                7.89
```

```
Févr                7.66
Mars                8.54
Avri               10.03
Mai                12.38
Juin               14.79
Juil               16.72
Août               17.48
Sept               16.63
Octo               14.38
Nove               11.31
Déce                8.96
Lat              49.7252
Long            -1.93983
Alt                   6
Moy               12.23
Amp                9.82
Zone                 NO
Name: 2, dtype: object
>>> villes.iloc[1,2]
4.7300000000000004
>>> villes.iloc[:3,:2]
            Ville  Janv
0        Abbeville  4.27
1    Lille-Lesquin  3.80
2  Pte De La Hague  7.89
```

On extrait une valeur en indiquant sa position dans la table avec des entiers

```
>>> villes.head().loc[1,'Janv']
3.7999999999999998
>>> villes.head().iloc[:,:3]
            Ville  Janv  Févr
0        Abbeville  4.27  4.96
1    Lille-Lesquin  3.80  4.73
2  Pte De La Hague  7.89  7.66
3   Caen-Carpiquet  5.38  5.94
4       Rouen-Boos  3.95  4.66
>>> villes.head().iloc[:,[1,3,5,12]]
   Janv  Mars    Mai  Déce
0  4.27  7.27  13.17  4.92
1  3.80  7.40  14.09  4.52
2  7.89  8.54  12.38  8.96
3  5.38  7.80  13.21  5.96
4  3.95  7.28  13.38  4.46
>>> villes.head().loc[:,:3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
```

Avec loc, il faut préciser le nombre de la colonne

```
>>> villes.columns
Index(['Ville', 'Janv', 'Févr', 'Mars', 'Avri', 'Mai', 'Juin', 'Juil', 'Août',
       'Sept', 'Octo', 'Nove', 'Déce', 'Lat', 'Long', 'Alt', 'Moy', 'Amp',
       'Zone'],
      dtype='object')
>>> villes.head().loc[:,['Janv','Mai','Déce']]
   Janv    Mai  Déce
```

```
0  4.27  13.17  4.92
1  3.80  14.09  4.52
2  7.89  12.38  8.96
3  5.38  13.21  5.96
4  3.95  13.38  4.46
```

Mais il est possible d'utiliser une colonne ou plusieurs colonnes comme index à l'aide de la fonction **`set_index`**.

```
>>> villesI = villes.set_index('Ville')
>>> villesI.head()
                Janv  Févr  Mars   Avri    Mai   Juin   Juil   Août   Sept
\
Ville
Abbeville        4.27  4.96  7.27   9.94  13.17  15.89  17.73  17.95  15.47
Lille-Lesquin    3.80  4.73  7.40  10.55  14.09  16.95  18.77  18.74  15.90
Pte De La Hague  7.89  7.66  8.54  10.03  12.38  14.79  16.72  17.48  16.63
Caen-Carpiquet   5.38  5.94  7.80  10.02  13.21  16.19  17.94  18.18  15.85
Rouen-Boos       3.95  4.66  7.28   9.99  13.38  16.38  18.11  18.11  15.33

                 Octo   Nove  Déce        Lat      Long  Alt    Moy    Amp
\
Ville
Abbeville        11.90   7.80  4.92  50.136000  1.834000   69  10.94  13.68
Lille-Lesquin    11.81   7.53  4.52  50.570000  3.097500   47  11.23  14.97
Pte De La Hague  14.38  11.31  8.96  49.725167 -1.939833    6  12.23   9.82
Caen-Carpiquet   12.59   8.75  5.96  49.180000 -0.456167   67  11.48  12.80
Rouen-Boos       11.60   7.40  4.46  49.383000  1.181667  151  10.89  14.16

               Zone
Ville
Abbeville        NO
Lille-Lesquin    NE
Pte De La Hague  NO
Caen-Carpiquet   NO
Rouen-Boos       NO
>>> villesI.head().iloc[:,[1,3,5,12]]
                Févr   Avri   Juin        Lat
Ville
Abbeville        4.96   9.94  15.89  50.136000
Lille-Lesquin    4.73  10.55  16.95  50.570000
Pte De La Hague  7.66  10.03  14.79  49.725167
Caen-Carpiquet   5.94  10.02  16.19  49.180000
Rouen-Boos       4.66   9.99  16.38  49.383000
>>> villesI.head().iloc[:,['Janv','Mars','Mai','Déce']]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
>>> villesI.head().loc[:,['Janv','Mars','Mai','Déce']]
                Janv  Mars    Mai  Déce
Ville
Abbeville        4.27  7.27  13.17  4.92
Lille-Lesquin    3.80  7.40  14.09  4.52
Pte De La Hague  7.89  8.54  12.38  8.96
Caen-Carpiquet   5.38  7.80  13.21  5.96
Rouen-Boos       3.95  7.28  13.38  4.46
```

Il est possible d'utiliser plusieurs colonnes comme index

```
>>> commandes = pd.read_csv("donnees/stagiaire/commandes.csv",
```

```
...       sep=';',header=0,usecols=[0,1,2,3,4,5],
...       names=['No','Client','Employe','Commande','Envoi','Port'])
>>> commandes.head()
       No Client  Employe    Commande       Envoi  Port
0  215650  LONEP       84  2010-02-02  2010-03-08  50.1
1  215653  PERIC       78  2010-02-02  2010-03-14  97.6
2  215652  BOTTM       72  2010-02-02  2010-03-02  89.3
3  215674  SPECD      111  2010-02-02  2010-03-01  86.2
4  215672  WELLI       39  2010-02-02  2010-02-12  71.9
>>> commandesI =
commandes.set_index(['Client','Employe','Commande'])
>>> commandes.dtypes
No            int64
Client       object
Employe       int64
Commande     object
Envoi        object
Port        float64
dtype: object
>>> commandesI.dtypes
No         int64
Envoi     object
Port     float64
dtype: object
>>> commandesI.iloc[1]
No           215653
Envoi    2010-03-14
Port           97.6
Name: (PERIC, 78, 2010-02-02), dtype: object
>>> commandesI.loc["PERIC",78,"2010-02-02"]
No           215653
Envoi    2010-03-14
Port           97.6
Name: (PERIC, 78, 2010-02-02), dtype: object
```

Si on veut changer l'index ou le supprimer il faut utiliser la fonction **« reset_index »**. Le mot-clé **« drop »** est utilisé pour garder ou non les colonnes servant d'index et **« inplace »** signifie qu'on modifie l'instance et non qu'une copie est modifiée.

```
>>> commandesI.reset_index(drop=False, inplace=True)
>>> commandesI.set_index(['No'],inplace=True)
>>> commandesI.dtypes
Client       object
Employe       int64
Commande     object
Envoi        object
Port        float64
dtype: object
>>> commandesI.head(3)
       Client  Employe    Commande       Envoi  Port
No
215650  LONEP       84  2010-02-02  2010-03-08  50.1
215653  PERIC       78  2010-02-02  2010-03-14  97.6
215652  BOTTM       72  2010-02-02  2010-03-02  89.3
```

Les index sont particulièrement utiles lorsqu'il s'agit de fusionner deux tables. Pour des petites tables, la plupart du temps, il est plus facile de s'en passer.

La restriction

Filter consiste à sélectionner un sous-ensemble de lignes du **`DataFrame`**. Pour filter sur plusieurs conditions, il faut utiliser les opérateurs logique & (et), | (ou), ~ (non).

```
>>> villes[villes.Janv > 7].Janv
Ville
Pte De La Hague       7.89
Brest-Guipavas        7.09
Ploumanac'h           7.79
Belle Ile-Le Talut    8.16
Pte De Chassiron      7.69
Montpellier           7.50
Marignane             7.54
Cap Cepet             9.08
Nice                  8.81
Perpignan             8.78
Ajaccio               9.14
Bastia                9.27
Name: Janv, dtype: float64
>>> villes[(villes.Janv > 7) &
...        (villes.Alt > 50)].loc[:,
...            ['Janv','Lat','Long','Alt']]
               Janv        Lat      Long  Alt
Ville
Brest-Guipavas  7.09  48.444167 -4.412000   94
Ploumanac'h     7.79  48.825833 -3.473167   55
Cap Cepet       9.08  43.079333  5.940833  115
```

Les dernières versions de pandas ont introduit la méthode query qui permet de réduire encore l'écriture.

```
>>> villes.query('(Janv > 7) & (Alt > 50)').loc[:,
...            ['Janv','Lat','Long','Alt']]
               Janv        Lat      Long  Alt
Ville
Brest-Guipavas  7.09  48.444167 -4.412000   94
Ploumanac'h     7.79  48.825833 -3.473167   55
Cap Cepet       9.08  43.079333  5.940833  115

>>> villes.query('((Janv < 5) & (Moy > 10 ) & (Amp < 15)) | (Alt >
500)').loc[:,
...            ['Janv','Moy','Amp','Alt']]
              Janv    Moy    Amp  Alt
Ville
Abbeville     4.27  10.94  13.68   69
Lille-Lesquin 3.80  11.23  14.97   47
Rouen-Boos    3.95  10.89  14.16  151
Alencon       4.37  11.29  14.35  143
Le Puy-Loudes 1.11   9.12  16.84  833
Millau        3.15  10.99  16.61  712
Embrun        1.57  10.89  19.18  871
```

L'union

Une des opérations les plus simples en algèbre relationnelle est l'union de données. Dans notre cas, nous allons nous intéresser à l'union de **« Series »** ou de **« DataFrame »**. Cette opération consiste en l'assemblage de plusieurs structures pour en créer une nouvelle. Avec Pandas, cette opération s'accomplit grâce à la fonction **« pd.concat »**.

```
>>> ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
>>> ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
>>> pd.concat([ser1, ser2])
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Pour une Series, cela paraît facile. Mais pour un DataFrame ?

```
>>> def make_df(cols, ind):
...     """Crée rapidement des DataFrame"""
...     data = {c: [str(c) + str(i) for i in ind]
...             for c in cols}
...     return pd.DataFrame(data, ind)
...
>>> df1 = make_df('AB', [1, 2])
>>> df2 = make_df('AB', [3, 4])
>>> df1
    A   B
1  A1  B1
2  A2  B2
>>> df2
    A   B
3  A3  B3
4  A4  B4
>>> pd.concat([df1, df2])
    A   B
1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4
>>> df1 = make_df('AB', [1, 2])
>>> df2 = make_df('CD', [3, 4])
>>> pd.concat([df1, df2])
     A    B    C    D
1   A1   B1  NaN  NaN
2   A2   B2  NaN  NaN
3  NaN  NaN   C3   D3
4  NaN  NaN   C4   D4
```

La concaténation préserve les index ! Par exemple, si les deux **DataFrames** donnés en arguments ont des index en commun, le résultat final aura des index dupliqués.

Pour accéder à un élément d'un objet Pandas avec un index hiérarchique, il suffit de spécifier plusieurs index.

```
>>> x = make_df('AB', [0, 1])
>>> y = make_df('AB', [2, 3])
>>> y.index = x.index  # Rend les index identiques
>>> # Nous avons alors des index dupliqués
>>> print(pd.concat([x, y]))
    A   B
0  A0  B0
1  A1  B1
0  A2  B2
1  A3  B3
>>> # Nous pouvons spécifier des index hiérarchiques
>>> hdf = pd.concat([x, y], keys=['x', 'y'])
>>> print(hdf)
      A   B
x 0  A0  B0
  1  A1  B1
y 0  A2  B2
  1  A3  B3
```

La jointure

Une autre fonction très utile pour manipuler les **Dataframe** est **« pd.merge »**. Elle permet de réaliser des opérations différentes en fonction des arguments qu'elle reçoit.

Jointure un-à-un

Imaginons que nous disposons de deux **Dataframe**, un contenant une liste d'employés et leurs dates d'entrée dans l'entreprise, et l'autre le nom des départements dans lesquels ils travaillent. La fonction **« pd.merge »** nous permet de transformer ces deux **Dataframes** en un seul contenant les deux informations.

```
>>> df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
...                     'department': ['Accounting',
...                         'Engineering', 'Engineering', 'HR']})
>>> df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
...                     'date': [2004, 2008, 2012, 2014]})
>>> df3 = pd.merge(df1, df2)
>>> df3
    department employee  date
0   Accounting      Bob  2008
1  Engineering     Jake  2012
2  Engineering     Lisa  2004
3           HR      Sue  2014
```

La fonction **« pd.merge »** a automatiquement reconnu que la colonne **employee** était commune aux deux **Dataframe**, et l'a utilisée comme clé de jointure.

Jointure plusieurs-à-un

Maintenant nous voulons ajouter une autre colonne. Chaque département a un chef. Cette information est contenue dans un **Dataframe**. Nous voulons ajouter une colonne à **df3** pour y ajouter le chef de chaque employé.

```
>>> df4 = pd.DataFrame({'department':['Accounting',
...                       'Engineering','HR'],
...                     'supervisor':['Carly','Guido','Steve']})
>>> pd.merge(df3, df4)
    department employee  date supervisor
0   Accounting      Bob  2008      Carly
1  Engineering     Jake  2012      Guido
2  Engineering     Lisa  2004      Guido
3           HR      Sue  2014      Steve
```

Remarquez que Guido apparaît plusieurs fois dans le résultat.

Jointure plusieurs-à-plusieurs

Les jointures plusieurs-à-plusieurs sont un peu compliquées à expliquer, mais elles sont quand même bien définies et très utiles. Pour continuer avec notre exemple, supposons que nous disposions d'un autre **Dataframe** contenant les compétences nécessaires pour travailler dans chaque département. Maintenant, nous souhaitons associer à chaque employé les compétences qu'il doit posséder pour travailler dans son département.

```
>>> df5 = pd.DataFrame({'department': ['Accounting','Accounting',
...                         'Engineering','Engineering','HR','HR'],
...                     'competence': ['math','spreadsheets','coding',
...                       'linux','spreadsheets','organization']})
>>> pd.merge(df1, df5)
    department employee    competence
0   Accounting      Bob          math
1   Accounting      Bob  spreadsheets
2  Engineering     Jake        coding
3  Engineering     Jake         linux
4  Engineering     Lisa        coding
5  Engineering     Lisa         linux
6           HR      Sue  spreadsheets
7           HR      Sue  organization
```

Quand la colonne utilisée comme clé de jointure possède des entrées dupliquées, comme c'est le cas pour **`df5`**, le résultat de **« `pd.merge` »** est une jointure plusieurs-à-plusieurs.

Le produit cartésien

Nous pouvons utiliser les jointures plusieurs-à-plusieurs pour réaliser une autre opération d'algèbre relationnelle, le produit cartésien.

```
>>> # Nous ajoutons une nouvelle colonne à df1 et df2
... # , qui contient toujours la même valeur, ici 0.
>>> df1['key'] = 0
>>> df2['key'] = 0
>>> # La jointure plusieurs-à-plusieurs
>>> produit_cartesien = pd.merge(df1, df2, how='left', on='key')
>>> produit_cartesien
     department employee_x  key  date employee_y
0    Accounting        Bob    0  2004       Lisa
1    Accounting        Bob    0  2008        Bob
2    Accounting        Bob    0  2012       Jake
3    Accounting        Bob    0  2014        Sue
4   Engineering       Jake    0  2004       Lisa
5   Engineering       Jake    0  2008        Bob
6   Engineering       Jake    0  2012       Jake
7   Engineering       Jake    0  2014        Sue
8   Engineering       Lisa    0  2004       Lisa
9   Engineering       Lisa    0  2008        Bob
10  Engineering       Lisa    0  2012       Jake
11  Engineering       Lisa    0  2014        Sue
12           HR        Sue    0  2004       Lisa
13           HR        Sue    0  2008        Bob
14           HR        Sue    0  2012       Jake
15           HR        Sue    0  2014        Sue
>>> # Effaçons la colonne key qui n'est plus utile
>>> produit_cartesien.drop('key',1, inplace=True)
>>> produit_cartesien.dtypes
department    object
employee_x    object
date           int64
employee_y    object
dtype: object
```

L'argument optionnel **« on »** permet de dire explicitement à **« pd.merge »** quelle colonne utiliser comme clé de jointure. L'argument **« how »** spécifie le type de jointure, parmi **« inner »**, **« outer »**, **« left »** et **« right »**.

```
>>> employes = pd.read_csv("donnees/stagiaire/employes.csv",
...         sep=';',header=0, na_values=["null"],
...         usecols=['No','Manager', 'Nom', 'Prenom'],
...         names=['No','Manager', 'Nom', 'Prenom'])
>>> managers = employes.copy()
>>> employes.dtypes
No           int64
Manager    float64
Nom         object
Prenom      object
dtype: object
>>> managers.dtypes
No           int64
Manager    float64
Nom         object
Prenom      object
dtype: object
>>> empman = pd.merge(employes, managers,left_on='Manager',
...            right_on='No',how='left')
>>> empman.head()
   No_x  Manager_x     Nom_x     Prenom_x  No_y  Manager_y     Nom_y  \
0    37        NaN    Giroux  Jean-Claude   NaN        NaN       NaN
1    14       37.0    Fuller       Andrew  37.0        NaN    Giroux
2    18       37.0  Brasseur        Hervé  37.0        NaN    Giroux
3    24       14.0  Buchanan       Steven  14.0       37.0    Fuller
4    95       18.0     Leger       Pierre  18.0       37.0  Brasseur

      Prenom_y
0          NaN
1  Jean-Claude
2  Jean-Claude
3       Andrew
4        Hervé
```

L'agrégation

Comme les tableaux **Numpy**, nous pouvons facilement effectuer des opérations sur l'ensemble des éléments d'une **Series** ou un **Dataframe**.

```
>>> import numpy as np
>>> rng = np.random.RandomState(42)
>>> # Une Series avec cinq nombres aléatoires
>>> ser = pd.Series(rng.rand(5))
>>> print(ser.sum())
2.811925491708157
>>> print(ser.mean())
0.5623850983416314
```

Pour un **Dataframe**, par défaut le calcul est fait par colonne.

```
>>> df = pd.DataFrame({'A': rng.rand(5),
...                    'B': rng.rand(5)})
>>> df
          A         B
0  0.155995  0.020584
1  0.058084  0.969910
2  0.866176  0.832443
3  0.601115  0.212339
4  0.708073  0.181825
>>> # Par colonne
... print(df.mean())
A    0.477888
B    0.443420
dtype: float64
>>> # Par ligne
>>> print(df.mean(axis='columns'))
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Pandas nous permet aussi d'accomplir une agrégation par groupe, semblable à ce qu'on peut obtenir en utilisant le mot clé **« GROUP BY »** en SQL.

```
>>> df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'data': range(6)}, columns=['key', 'data'])
>>> print(df)
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
>>> df.groupby('key').sum()
     data
key
```

```
A       3
B       5
C       7
```

Dans Pandas, cette opération se fait en deux étapes. Nous allons d'abord créer un objet de type **`DataFrame`**. **`GroupBy`** c'est une sorte de vue sur notre **`DataFrame`**. Ensuite, nous pouvons appliquer les opérations que nous souhaitons sur ce nouvel objet. Le résultat sera agrégé !

```
>>> employes = pd.read_csv("donnees/stagiaire/employes.csv",
...   sep=';',header=0,index_col='No',na_values=["null"],
...   usecols=['No','Nom','Fonction','Pays','Salaire','Commission'],
...   names=['No','Manager','Nom','Prenom','Fonction','Titre',
...  'Naissance','Embauche','Salaire','Commission','Pays','Region'])
>>> employes.head()
        Nom         Fonction  Salaire  Commission Pays
No
37    Giroux        Président   150000         NaN  NaN
14    Fuller   Vice-Président    96000         NaN  NaN
18  Brasseur   Vice-Président   147000         NaN  NaN
24  Buchanan  Chef des ventes    13000     12940.0  NaN
95     Leger  Chef des ventes    19000     11150.0  NaN
>>> employes.Commission = employes.apply(lambda x: 0
...       if np.isnan(x['Commission']) else x['Commission'], axis=1)
>>> employes.Pays = employes.apply(lambda x: 'NonAff'
...        if pd.isnull(x['Pays']) else x['Pays'], axis=1)
>>> employes.head()
        Nom         Fonction  Salaire  Commission    Pays
No
37    Giroux        Président   150000         0.0  NonAff
14    Fuller   Vice-Président    96000         0.0  NonAff
18  Brasseur   Vice-Président   147000         0.0  NonAff
24  Buchanan  Chef des ventes    13000     12940.0  NonAff
95     Leger  Chef des ventes    19000     11150.0  NonAff
>>> employes.groupby('Fonction').sum()
                      Salaire  Commission
Fonction
Assistante commerciale    16540         0.0
Chef des ventes           83000     68790.0
Président                150000         0.0
Représentant(e)          692900     88900.0
Vice-Président           243000         0.0
>>> employes.groupby('Fonction').mean()
                             Salaire    Commission
Fonction
Assistante commerciale    1654.000000      0.000000
Chef des ventes          13833.333333  11465.000000
Président               150000.000000      0.000000
Représentant(e)           7531.521739    966.304348
Vice-Président          121500.000000      0.000000
>>> employes.groupby(['Fonction','Pays']).sum()
                                  Salaire  Commission
Fonction              Pays
Assistante commerciale NonAff       16540         0.0
Chef des ventes        NonAff       83000     68790.0
Président              NonAff      150000         0.0
Représentant(e)        Allemagne    51200      9660.0
                       Argentine    38900      5640.0
```

```
                      Autriche       25600      1960.0
                      Belgique       27000      2930.0
                      Brésil         23100      1090.0
                      Canada         35500      5840.0
                      Danemark       27500      3510.0
                      Espagne        36700      4860.0
                      Finlande       29800      2200.0
                      France         32200      3610.0
                      Irlande        36400      4040.0
                      Italie         29000      1590.0
                      Mexique        28900      3860.0
                      Norvège        35300      6160.0
                      Pologne        32400      5030.0
                      Portugal       31600      3860.0
                      Royaume-Uni    42900      4570.0
                      Suisse         36700      5560.0
                      Suède          31300      2790.0
                      Venezuela      37800      6510.0
                      États-Unis     23100      3630.0
Vice-Président        NonAff        243000         0.0
>>> employes.groupby(['Fonction','Pays']).mean()
                                        Salaire    Commission
Fonction              Pays
Assistante commerciale NonAff         1654.000000      0.000000
Chef des ventes       NonAff        13833.333333  11465.000000
Président             NonAff       150000.000000      0.000000
Représentant(e)       Allemagne      7314.285714   1380.000000
                      Argentine      7780.000000   1128.000000
                      Autriche       6400.000000    490.000000
                      Belgique       6750.000000    732.500000
                      Brésil         7700.000000    363.333333
                      Canada         7100.000000   1168.000000
                      Danemark       9166.666667   1170.000000
                      Espagne        7340.000000    972.000000
                      Finlande       7450.000000    550.000000
                      France         8050.000000    902.500000
                      Irlande        7280.000000    808.000000
                      Italie         7250.000000    397.500000
                      Mexique        7225.000000    965.000000
                      Norvège        7060.000000   1232.000000
                      Pologne        8100.000000   1257.500000
                      Portugal       7900.000000    965.000000
                      Royaume-Uni    8580.000000    914.000000
                      Suisse         7340.000000   1112.000000
                      Suède          7825.000000    697.500000
                      Venezuela      7560.000000   1302.000000
                      États-Unis     7700.000000   1210.000000
Vice-Président        NonAff       121500.000000      0.000000
```