



Python, perfectionnement

Cours conçu par **Razvan BIZOI**

Razvan@BIZOI.fr

Table des matières

MODULE 1 : L'INTRODUCTION	1-1
Un peu d'histoire	1-2
Présentation du langage Python	1-3
Installation sous Linux	1-5
Installation sous Windows	1-6
L'écosystème Python scientifique.....	1-7
PyCharm	1-10
MODULE 2 : LA SYNTAXE BASIQUE	2-1
Le nom de variable	2-2
L'affectation	2-3
La réaffectation.....	2-5
Les types de données	2-6
Les types numériques	2-7
Le type integer	2-8
Le type float.....	2-10
Les types booléens.....	2-12
Les listes.....	2-13
Les instructions conditionnelles	2-14
Les blocs d'instructions.....	2-16
L'instruction while	2-17
Les chaînes de caractères	2-19
Manipulation d'une chaîne	2-21
Formatage d'une chaîne	2-24
Le type bytes et la page de code.....	2-26
L'instruction for	2-28

MODULE 3 : LES FONCTIONS.....	3-1
Les fonctions prédéfinies	3-2
Les modules internes	3-3
La définition d'une fonction	3-5
La fonction sans paramètres	3-6
La fonction avec paramètres	3-7
Les paramètres modifiables.....	3-9
La fonction récursive	3-11
Les variables locales et globales	3-12
L'aide sur la fonction	3-14
L'écriture simplifiée des fonctions	3-15
La fonction map.....	3-17
La fonction filter	3-18
Les iterators	3-19
Les fonctions générateur.....	3-20
La compilation dynamique	3-22
MODULE 4 : L'UTILISATION DES FICHIERS.....	4-1
Les fonctions prédéfinies	4-2
Le répertoire courant.....	4-4
L'écriture séquentielle	4-6
Lecture séquentielle d'un fichier	4-8
Les fichiers texte	4-10
L'enregistrement et restitution de variables.....	4-13
Les exceptions try – except – else	4-15
Le type d'exception	4-17
Les d'exceptions standards.....	4-19
Lancer une exception	4-20
Les instructions try, except imbriquées.....	4-21
Les fichiers compressés	4-23
La manipulation de fichiers.....	4-25
Les expressions régulières	4-27
Les symboles simples.....	4-28
Les symboles de répétition.....	4-31
Les symboles de regroupement	4-33
Les fonctions et objets de re	4-35
MODULE 5 : LES STRUCTURES DE DONNEES.....	5-1
Les chaînes de caractères.....	5-2
Les listes	5-5

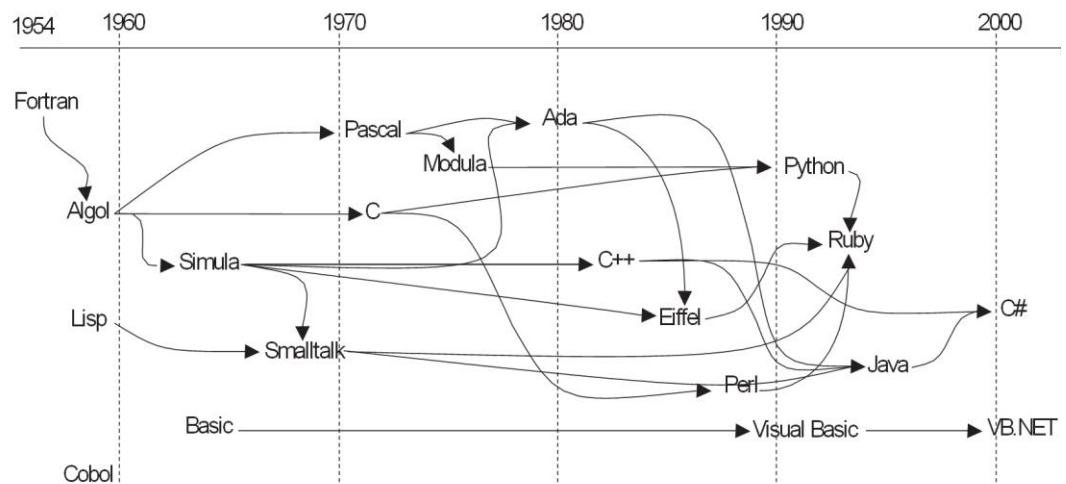
Les tuples	5-12
Les dictionnaires	5-14
MODULE 6 : LES OUTILS INDISPENSABLES	6-1
Les tableaux numériques	6-2
La création de tableaux	6-4
L’affichage des tableaux	6-6
L’accès aux composantes d’un tableau	6-7
Lecture et écriture d’un tableau	6-8
Les opérations simples sur les tableaux	6-11
La librairie Matplotlib	6-14
Visualiser l’incertitude	6-16
La personnalisation et sous-graphes	6-18
La librairie Seaborn	6-27
MODULE 7 : LES DATAFRAMES	7-1
Les structures de données	7-2
Lecture et écriture de DataFrame	7-5
La projection et la restriction	7-8
La restriction	7-13
L’union	7-14
La jointure	7-16
L’agrégation	7-19

1

L'Introduction

Un peu d'histoire

Un peu d'histoire



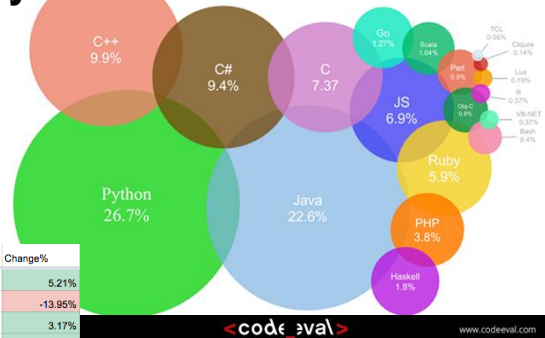
Les des langages de programmation ont ouvert la voie de l'abstraction, principe fondamental qui cherche à éloigner autant que possible le programmeur des contingences matérielles, pour le rapprocher des concepts, de la "chose" concrète ou abstraite qu'il doit traiter.

C'est ainsi qu'au fil du temps différents ensembles de règles et de principes de conception de programmation se sont développés. Programmation structurée, fonctionnelle, orientée objet... À chaque évolution, l'objectif est toujours le même : permettre la meilleure représentation informatique possible, la plus intuitive possible, de l'idée sur laquelle opère le programme.

Il ne s'agit là que d'une vue vraiment très simplifiée, presque caricaturale. Mais vous pouvez constater que les langages s'inspirent fortement les uns des autres. Par ailleurs, la plupart de ceux présentés sur ce schéma sont toujours utilisés et continuent d'évoluer – c'est, par exemple, le cas du vénérable ancêtre Fortran, notamment dans les applications très gourmandes en calculs scientifiques.

Présentation du langage Python

Présentation du langage Python



2015 Rank		2015	Change%	2014	Change%	2013	Change%
1	Python	26.67%	-14.64%	31.24%	3.10%	30.30%	5.21%
2	Java	22.58%	15.37%	19.57%	-11.85%	22.20%	-13.95%
3	C++	9.96%	1.76%	9.79%	-24.70%	13.00%	3.17%
4	C#	9.39%	27.37%	7.37%	47.37%	5.00%	100.00%
5	C	7.37%	21.37%	6.07%	48.14%	4.10%	-16.33%
6	JavaScript	6.88%	6.09%	6.48%	24.66%	5.20%	33.33%
7	Ruby	5.88%	-17.27%	7.11%	-32.90%	10.60%	10.42%
8	PHP	3.82%	5.45%	3.62%	9.84%	3.30%	-54.79%
9	Haskell	1.77%	17.24%	1.51%	25.83%	1.20%	
10	Go	1.27%	-44.00%	2.26%	50.67%	1.50%	-25.00%
11	Scala	1.04%	-17.80%	1.27%	27.00%	1.00%	66.67%
12	Perl	0.95%	-37.33%	1.52%	-6.17%	1.62%	
13	Objective-C	0.82%	-17.62%	1.00%	265.76%	0.27%	173.40%
14	Bash	0.46%	7.21%	0.43%	290.91%	0.11%	
15	R	0.37%	165.71%	0.14%	-30.00%	0.20%	
16	Visual Basic.NET	0.37%	825.50%	0.04%			
17	Lua	0.19%	-44.51%	0.35%	337.50%	0.08%	
18	Clojure	0.14%	-8.53%	0.15%	-48.28%	0.29%	-63.75%
19	Tcl	0.06%	-8.57%	0.07%	133.33%	0.03%	50.00%



Détaillons un peu les principales caractéristiques de Python, plus précisément, du langage et de ses deux implantations actuelles.

- Python est portable, non seulement sur les différentes variantes d'Unix, mais aussi sur les OS propriétaires.
- Python est gratuit, mais on peut l'utiliser sans restriction dans des projets commerciaux.
- Python convient aussi bien à des scripts d'une dizaine de lignes qu'à des projets complexes de plusieurs dizaines de milliers de lignes.
- La syntaxe de Python est très simple et, combinée à des types de données évolués (listes, dictionnaires...), conduit à des programmes à la fois très compacts et très lisibles. À fonctionnalités égales, un programme Python (abondamment commenté et présenté selon les canons standards) est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
- Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de comptage de références (proche, mais différent, d'un garbage collector).
- Python est orienté-objet. Il supporte l'héritage multiple et la surcharge des opérateurs. Dans son modèle objets, et en reprenant la terminologie de C++, toutes les méthodes sont virtuelles.
- Python intègre, comme Java ou les versions récentes de C++, un système d'exceptions, qui permettent de simplifier considérablement la gestion des erreurs.
- Python est **dynamique** (l'interpréteur peut évaluer des chaînes de caractères représentant des expressions ou des instructions Python), **orthogonal** (un petit nombre de concepts suffit à engendrer des constructions très riches), **réflectif** (il supporte la métaprogrammation, par exemple la capacité pour un objet de se

rajouter ou de s'enlever des attributs ou des méthodes, ou même de changer de classe en cours d'exécution) et **introspectif** (un grand nombre d'outils de développement, comme le debugger ou le profiler, sont implantés en Python lui-même).

- Comme Scheme ou SmallTalk, Python est dynamiquement typé. Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.
- Python possède actuellement deux implémentations. L'une, interprétée, dans laquelle les programmes Python sont compilés en instructions portables, puis exécutés par une machine virtuelle (comme pour Java, avec une différence importante : Java étant statiquement typé, il est beaucoup plus facile d'accélérer l'exécution d'un programme Java que d'un programme Python). L'autre génère directement du bytecode Java.
- Python est extensible : comme Tcl ou Guile, on peut facilement l'interfacer avec des bibliothèques C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- La bibliothèque standard de Python, et les paquetages contribués, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standards (fichiers, pipes, signaux, sockets, threads...), protocoles Internet (Web, News, FTP, CGI, HTML...), persistance et bases de données, interfaces graphiques.
- Python est un langage qui continue à évoluer, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Parallèlement à l'interpréteur principal, écrit en C et maintenu par le créateur du langage, un deuxième interpréteur, écrit en Java, est en cours de développement.

Installation sous Linux

Installation sous Linux



```
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tgz
tar xzf Python-3.6.1.tgz
cd Python-3.6.1
./configure
make
make install
```

```
which python3
python3 -V
```



```
wget \
https://repo.continuum.io/archive/Anaconda3-4.3.1-Linux-x86_64.sh
```

```
bash Anaconda3-4.3.1-Linux-x86_64.sh
```

Python est souvent préinstallé sur la plupart des distributions GNU/Linux. Vous pouvez contrôler sa présence en tapant la commande python dans un terminal.



```
[root@ocean Python-3.6.1]# which python3
/usr/local/bin/python3
[root@ocean Python-3.6.1]# python3 -V
Python 3.6.1
[root@ocean Python-3.6.1]# which python
/usr/bin/python
[root@ocean Python-3.6.1]# python -V
Python 2.6.6
```

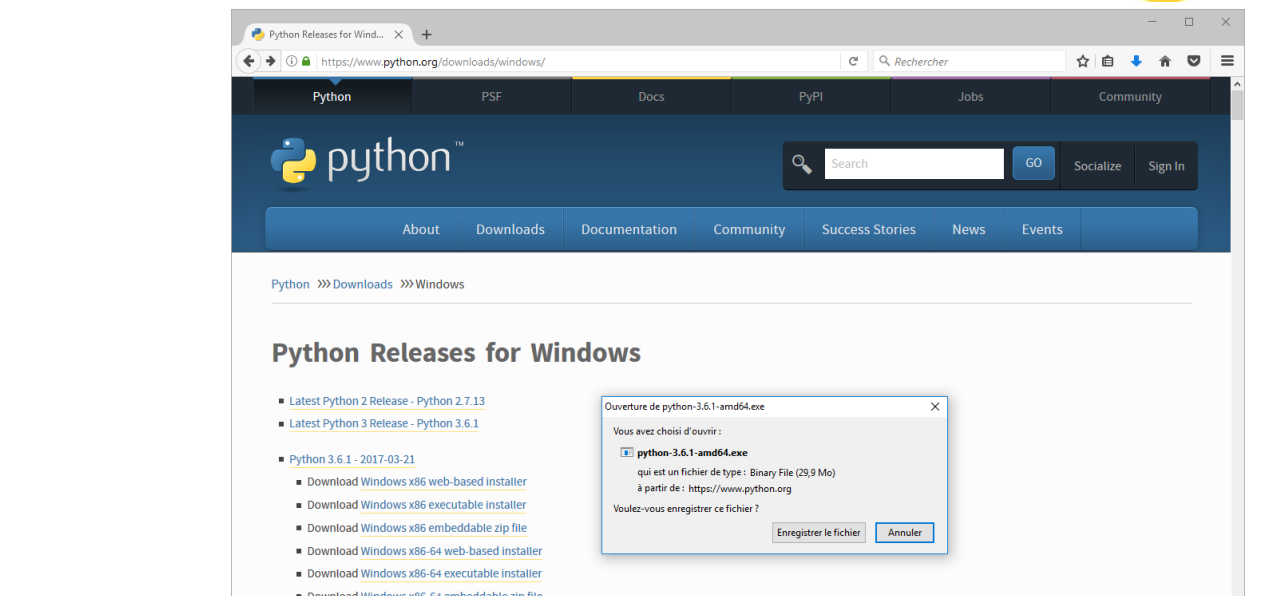
S'il vous est nécessaire d'installer Python ou de mettre à jour une version existante, vous pouvez le faire par le biais du système de package de la distribution ou le recompiler manuellement. Les manipulations d'installation se font en tant que super-utilisateur, ou root.



```
[root@mercure ~]# yum -y install gcc make zlib-devel
[root@mercure ~]# wget
https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tgz
...
[root@mercure ~]# tar xzf Python-3.6.1.tgz
[root@mercure ~]# cd Python-3.6.1
[root@mercure Python-3.6.1]# ./configure
...
[root@mercure Python-3.6.1]# make
...
Successfully installed pip-9.0.1 setuptools-28.8.0
[root@mercure Python-3.6.1]# which python3
/usr/local/bin/python3
[root@mercure Python-3.6.1]# python3 -V
Python 3.6.1
```

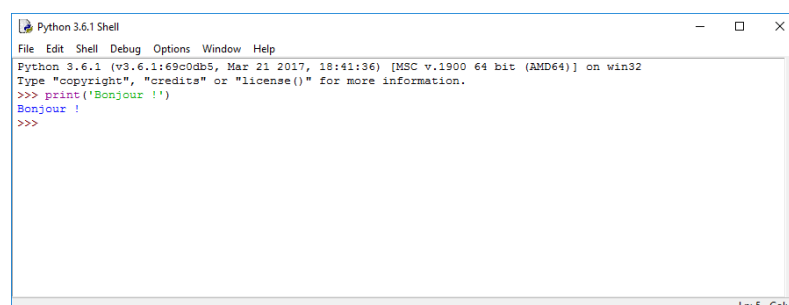
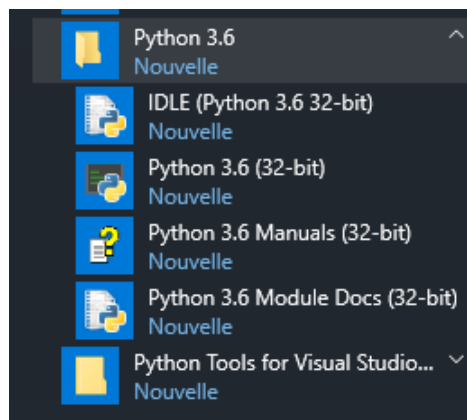
Installation sous Windows

Installation sous Windows



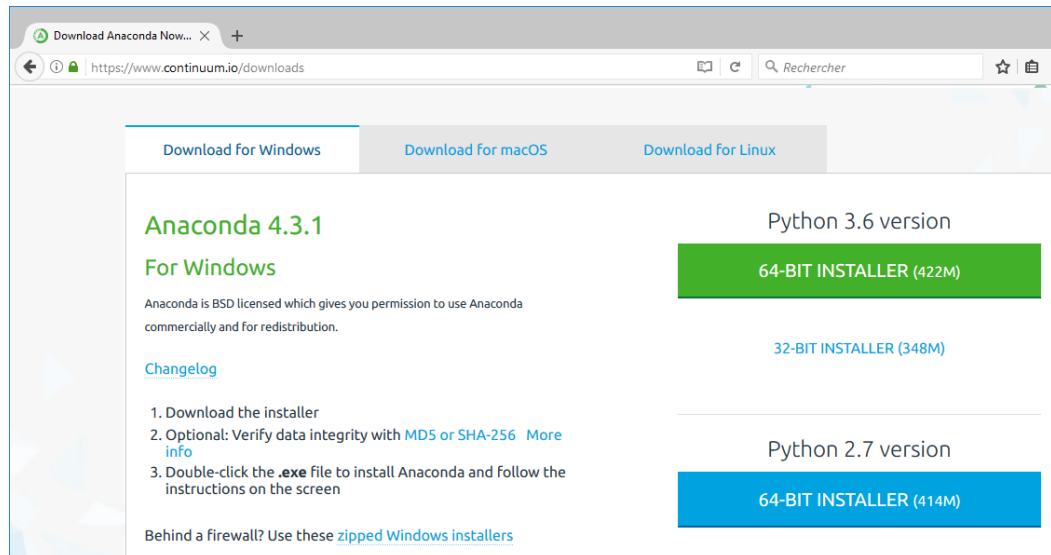
Les plates-formes Windows bénéficient d'un installeur graphique automatique, présenté sous la forme d'un assistant. Si vous n'avez pas les droits administrateurs sur la machine, il est possible de sélectionner dans les options avancées une installation en tant que non-administrateur. L'installation de Python par ce biais ne présente aucune difficulté.

Une fois l'installation achevée, une nouvelle entrée Python apparaît dans le menu Démarrer>Programmes.



L'écosystème Python scientifique

L'écosystème Python scientifique



Au fil des années Python est devenu un outil du quotidien pour les ingénieurs et chercheurs de toutes les disciplines scientifiques. Il est devenu un des outils incontournables des Data Scientists!

Pourquoi utiliser Python pour le calcul scientifique ?

Python est devenu une alternative viable aux solutions propriétaires leader du marché, comme MatLab, Maple, Mathematica, Statistica, ...

Anaconda

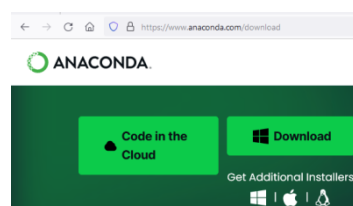
Anaconda est peut-être la distribution la plus répandue en raison de la diversité des plateformes qu'elle supporte et d'une plus grande ouverture des outils sur lesquels elle se base.

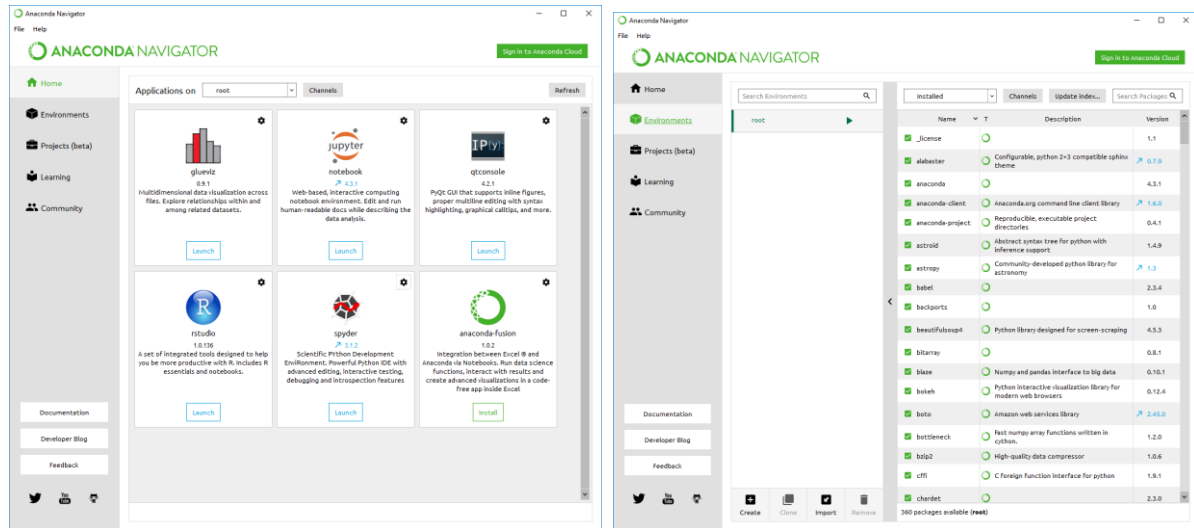
Elle propose une distribution communautaire et partiellement optimisée ainsi qu'une version payante mieux optimisée offrant de nombreux outils complémentaires.

Elle est « cross-platform » et s'appuie sur l'outil de virtualisation d'environnement « conda »

Elle inclue plus de 100 librairies pré-installées et en propose 600 de plus en téléchargement. Enfin, elle est compatible avec « pip ».

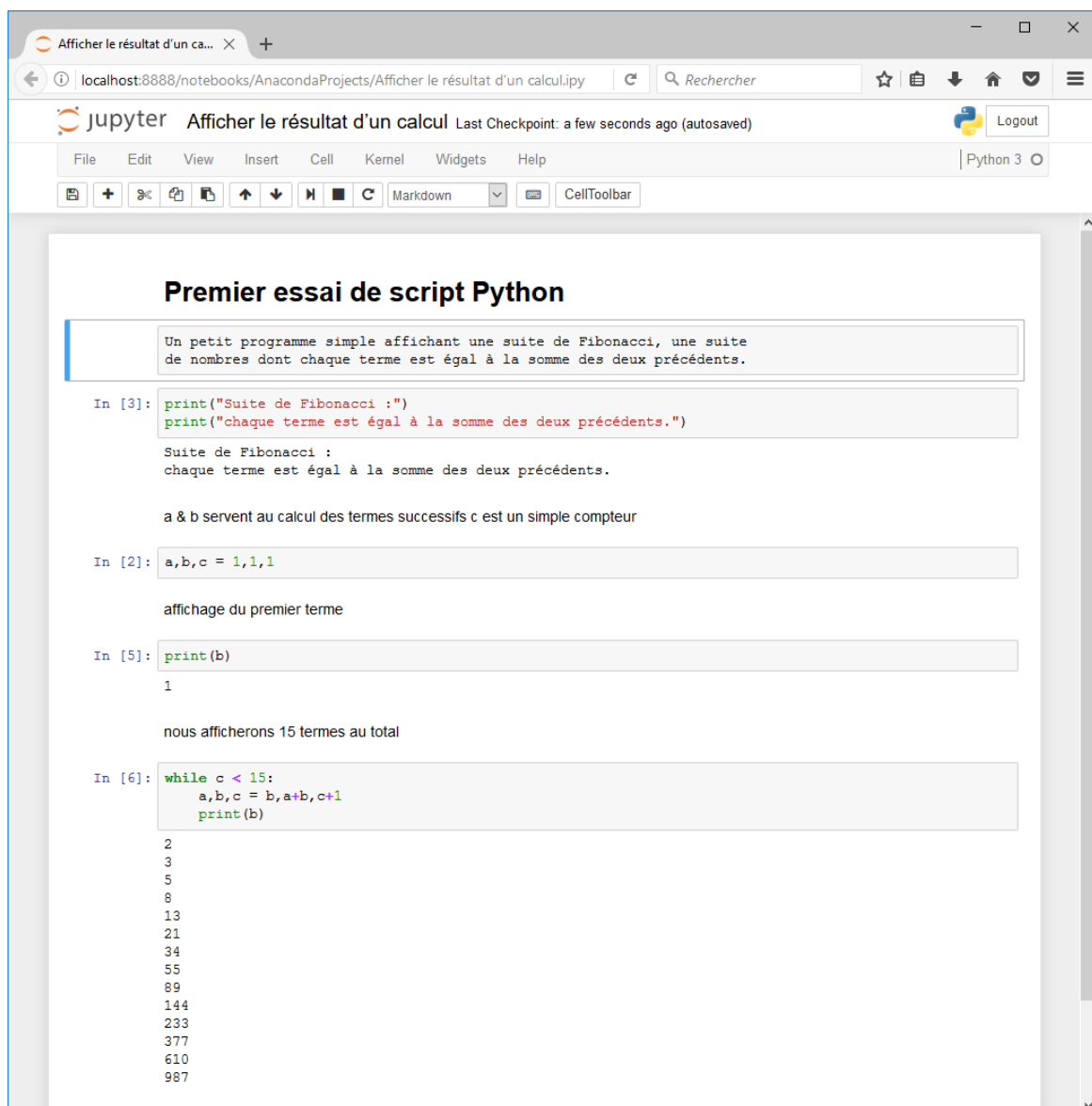
<https://www.anaconda.com/download>





Jupyter Notebook est un projet open source dérivé d'iPython et fournit une interface web riche dans le cadre d'une programmation interactive.

Jupyter Notebook permet à ses utilisateurs de travailler sur des documents associant du code, du texte, des équations, des images, des vidéos et des graphiques.



JupyterLab est le successeur de Jupyter Notebook avec plus de fonctionnalités dans la gestion des développements.

The screenshot displays the JupyterLab environment. On the left, a file explorer shows the directory structure: `/ IntelligenceEnDonneesDeSante / Pandas /`. The main area is divided into two panes. The top pane, titled 'Table of Contents', lists the document sections: 1. Les imports et configuration du document (with a sub-item 1.1 La lecture des fichiers d'un répertoire), 2. La température mensuelle, 3. La météo mensuelle, 4. La météo hebdomadaire, and 5. La météo journalière. The bottom pane, titled 'Les imports et configuration du document', contains a code cell with the following Python code:

```
[1]: import pandas as pd
import seaborn as sns
import warnings
import os
from datetime import datetime
from matplotlib import pyplot as plt

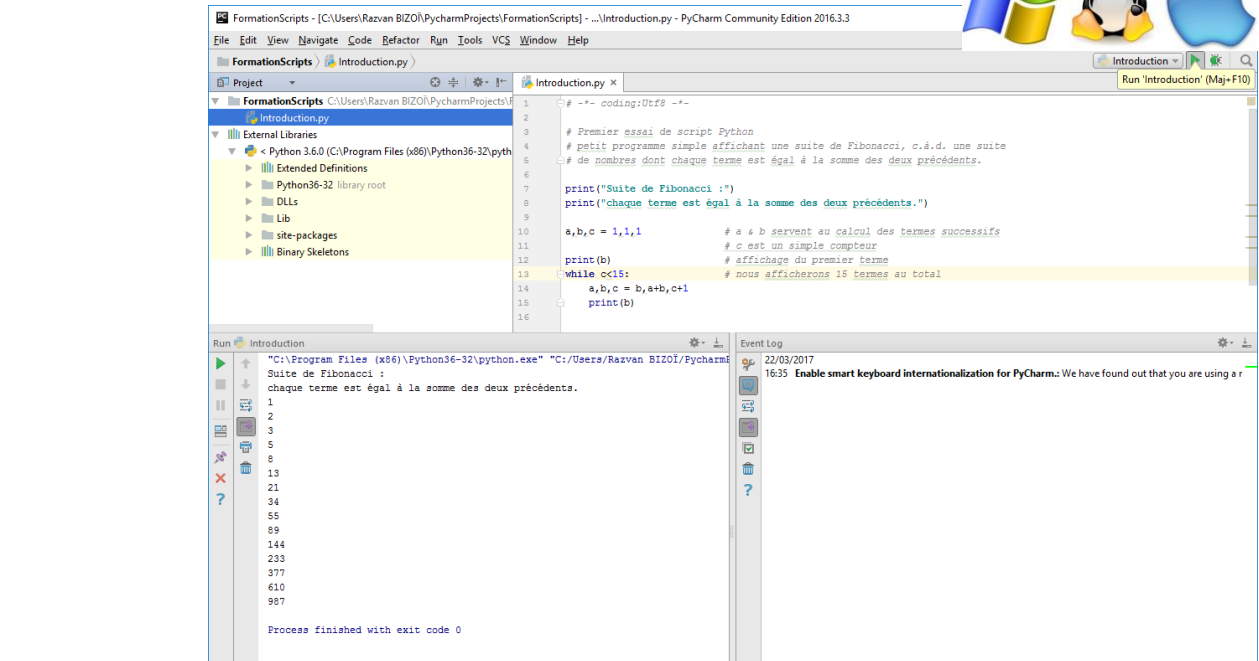
warnings.filterwarnings(action="ignore")

%matplotlib inline
```

The status bar at the bottom indicates the current mode is 'Command', the kernel is 'Python 3 (ipykernel)', and the file is 'LectureDesDonneesMeteo.ipynb'.

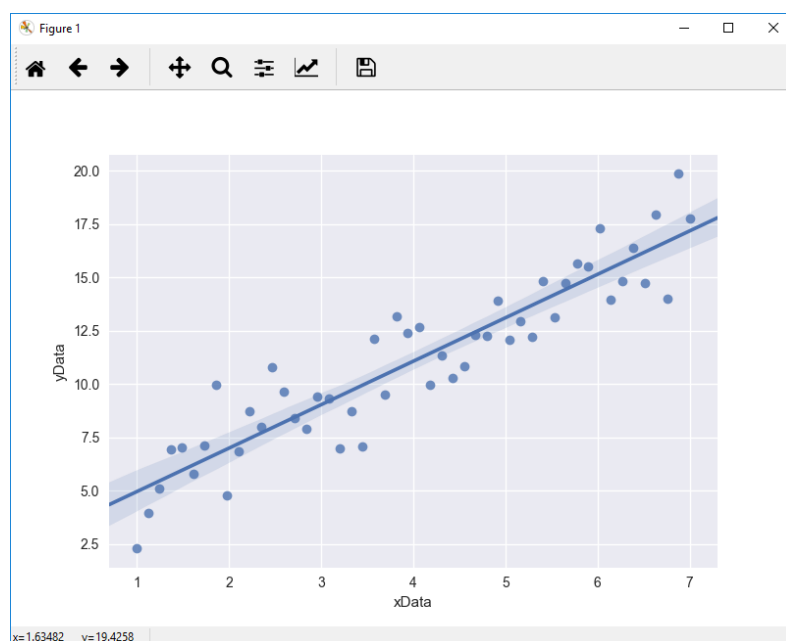
PyCharm

PyCharm



```
import numpy as np
import matplotlib.pyplot as plt

import pandas as pd
import seaborn as sns
x = np.linspace(1, 7, 50)
y = 3 + 2*x + 1.5*np.random.randn(len(x))
df = pd.DataFrame({'xData':x, 'yData':y})
sns.regplot('xData', 'yData', data=df)
plt.show()
```



2

La syntaxe basique

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 **tableDesMatières**.

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.

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 :
64880030544660752790736837369104977695001034284228042891827649456186
234582611607420928
13 :
70056698901118320029237641399576216921624545057972697917383692313271
75488362123506443467340026896520469610300883250624900843742470237847
552
14 :
45452807645626579985636294048249351205168239870722946151401655655658
39864222761633581512382578246019698020614153674711609417355051422794
79530059170096950422693079038247634055829175296831946224503933501754
776033004012758368256
```

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'
```

PYA - Python, perfectionnement

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	<code>x = True or False</code> (résultat = True)
not	négation logique	<code>x = not x</code>
< >	inférieur, supérieur	<code>x = 5 < 5</code>
<= >=	inférieur ou égal, supérieur ou égal	<code>x = 5 <= 5</code>
== !=	égal, différent	<code>x = 5 == 5</code>

```
>>> 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 listes

Les chaînes que nous avons abordées à la rubrique précédente constituaient un premier exemple de données composites. Vous apprendrez progressivement à utiliser plusieurs autres types de données composites, parmi lesquelles les listes, les tuples et les dictionnaires. Sous Python, on peut définir une liste comme une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets.

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, \
            20.357, 'jeudi', 'vendredi']
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print(jour[3])
1800
>>> print(jour[6])
vendredi
```

À la différence de ce qui se passe pour les chaînes, qui constituent un type de données non-modifiables, il est possible de changer les éléments individuels d'une liste :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi']
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi']
>>> jour[3] = jour[3] + 47
>>> jour[3] = 'Juillet'
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'Juillet', 20.357, 'jeudi']
```

La fonction intégrée « **len** », que nous avons déjà rencontrée à propos des chaînes, s'applique aussi aux listes. Elle renvoie le nombre d'éléments présents dans la liste.

Une autre fonction intégrée permet de supprimer d'une liste un élément quelconque (à partir de son index). Il s'agit de la fonction « **del** ».

```
>>> len(jour)
7
>>> del(jour[3])
>>> del(jour[4])
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'jeudi']
```

Il est également tout à fait possible d'ajouter un élément à une liste, mais pour ce faire, il faut considérer que la liste est un objet, dont on va utiliser l'une des méthodes.

```
>>> jour.append('vendredi')
>>> jour.append('samedi')
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']
```

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 :

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



```
>>> 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
```

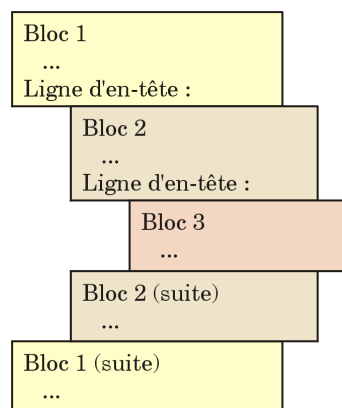
Les blocs d'instructions

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
```

Atelier 2 - Exercice 1.2- Exercice 1.3- Exercice 1.4- Exercice 1.5

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 supprimés.
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 {:.2f}."
>>> 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'inverse.

`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 convertir 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\xad\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 « $n - \text{debut}$ » soit un multiple de « **marche** ».

```
>>> liste = [3, 2, 1, 6, 4, 9, 7, 8, 5]
>>> for x in range(0,len(liste)):print(x,":",liste[x])

0 : 3
1 : 2
2 : 1
3 : 6
4 : 4
5 : 9
6 : 7
7 : 8
8 : 5
>>> for x in range(0,len(liste),2):print(x,":",liste[x])

0 : 3
2 : 1
4 : 4
6 : 7
8 : 5
```


3

Les fonctions

Les fonctions prédéfinies

L'un des concepts les plus importants en programmation est celui de fonction. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable de calculer une racine carrée, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la réécrire à chaque fois.

Vous avez déjà rencontré d'autres fonctions intégrées au langage lui-même, « **print** », « **input** », « **len** », etc.

Importer un module de fonctions

Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des modules.

Il existe un grand nombre de modules préprogrammés qui sont fournis d'office avec Python. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées, que l'on appelle des bibliothèques.

Le module « **math** », contient les définitions de nombreuses fonctions mathématiques. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

```
from math import [* | listeObjets]
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant toutes les fonctions du module « **math** », lequel contient une bibliothèque de fonctions mathématiques préprogrammées.

```
>>> from math import sqrt
>>> nombre = 121
>>> print("racine carrée de", nombre, "=", sqrt(nombre))
racine carrée de 121 = 11.0
>>> angle = pi/6 # soit 30°
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined

>>> from math import *
>>> nombre, angle = 121, pi/6 # soit 30°
>>> # (la bibliothèque math inclut aussi la définition de pi)
>>> print("racine carrée de", nombre, "=", sqrt(nombre))
racine carrée de 121 = 11.0
>>> print("sinus de", angle, "radians", "=", sin(angle))
sinus de 0.5235987755982988 radians = 0.49999999999999994
```


Les modules internes

Python dispose de nombreux modules préinstallés. Cette liste est trop longue pour figurer dans ce document, elle est aussi susceptible de s'allonger au fur et à mesure du développement du langage Python. Voici une liste des modules les plus utilisés.

module	signification
asyncio	Thread, socket, protocol.
calendar	Gérer les calendriers, les dates.
cgi	Utilisé dans les scripts CGI (programmation Internet)
cmath	Fonctions mathématiques complexes.
copy	Copies d'instances de classes.
csv	Gestion des fichiers au format CSV.
datetime	Calculs sur les dates et heures.
gc	Gestion du garbage collector.
getopt	Lire les options des paramètres passés en arguments d'un programme Python.
glob	Chercher des fichiers.
hashlib	Fonctions de cryptage.
htmlib	Lire le format HTML.
math	Fonctions mathématiques standard telles que cos, sin, exp, log...
os	Fonctions systèmes dont certaines fonctions permettant de gérer les fichiers
os.path	Manipulations de noms de fichiers
pathlib	Manipulation de chemins.
pickle	Sérialisation d'objets, la sérialisation consiste à convertir des données structurées de façon complexe en une structure linéaire facilement enregistrable dans un fichier.
profile	Etudier le temps passé dans les fonctions d'un programme.
random	Génération de nombres aléatoires.
re	Expressions régulières.
shutil	Copie de fichiers.
sqlite3	Accès aux fonctionnalités du gestionnaire de base de données SQLite3.
string	Manipulations des chaînes de caractères.
sys	Fonctions systèmes, fonctions liées au langage Python.
threading	Utilisation de threads.
time	Accès à l'heure, l'heure système, l'heure d'un fichier.
tkinter	Interface graphique.
unittest	Tests unitaires (ou comment améliorer la fiabilité d'un programme).
urllib	Pour lire le contenu de page HTML sans utiliser un navigateur.
xml.dom	Lecture du format XML.
xml.sax	Lecture du format XML.
zipfile	Lecture de fichiers ZIP.

Vous pouvez installer d'autres packages à l'aide de la commande :

pip install nomPackage

```
C:\Users\Razvan BIZOI>pip install pandas
Requirement already satisfied: pandas in c:\programdata\anaconda3\lib\site-packages
Requirement already satisfied: python-dateutil>=2 in c:\programdata\anaconda3\lib\site-packages (from pandas)
```

```
Requirement already satisfied: pytz>=2011k in
c:\programdata\anaconda3\lib\site-packages (from pandas)
Requirement already satisfied: numpy>=1.7.0 in
c:\programdata\anaconda3\lib\site-packages (from pandas)
Requirement already satisfied: six>=1.5 in
c:\programdata\anaconda3\lib\site-packages (from python-dateutil>=2->pandas)
```

Le programme suivant par exemple utilise les modules « **random** » et « **math** » pour estimer le nombre pi.

```
>>> import random
>>> import math
>>> somme = 0
>>> nb = 1000000
>>> for i in range (0,nb) :
    # nombre aléatoire entre [0,1]
    x = random.random()
    y = random.random()
    r = math.sqrt(x*x + y*y) #racine carrée
    if r <= 1:
        somme += 1
>>> print("estimation ", 4 * float (somme) / nb)
estimation 3.140048
>>> print("PI = ", math.pi)
PI = 3.141592653589793
```

Le programme suivant calcule l'intégrale de Monte Carlo de la fonction $f(x) = \sqrt{x}$ qui consiste à tirer des nombres aléatoires dans l'intervalle $[a, b]$ puis à faire la moyenne des \sqrt{x} obtenu.

```
>>> import random
>>> import math
>>> def integrale_monte_carlo(a, b, f, n):
    somme = 0.0
    for i in range(0, n):
        x = random.random() * (b-a) + a
        y = f(x)
        somme += f(x)
    return somme / n

>>> def racine(x):
    return math.sqrt(x)

>>> print(integrale_monte_carlo(0, 1, racine, 100000))
0.6676537057136576
```

La définition d'une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.

Les fonctions et les classes d'objets sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par décrire ici la définition de fonctions sous Python. Les objets et les classes seront examinés plus loin.

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nomDeLaFonction(liste de paramètres):  
    ...  
    bloc d'instructions  
    ...
```

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage, et à la condition de n'utiliser aucun caractère spécial ou accentué, le caractère souligné « `_` » est permis. Comme c'est le cas pour les noms de variables, il vous est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom, les noms commençant par une majuscule seront réservés aux classes.
- Comme les instructions « `if` » et « `while` » que vous connaissez déjà, l'instruction « `def` » est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.
- La liste de paramètres spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction, les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments.
- Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un appel de fonction est constitué du nom de la fonction suivi de parenthèses.

Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans la définition de la fonction, encore qu'il soit possible de définir pour ces paramètres des valeurs par défaut.

La fonction sans paramètres

Le mode interactif de Python est en effet idéal pour effectuer des petits tests comme ceux qui suivent. C'est une facilité que n'offrent pas tous les langages de programmation !

```
>>> def table7():
    t = list()
    n = 1
    while n < 11:
        print(n * 7, end=' ')
        t.append(n * 7)
        n = n + 1
    print('\n')
    return t

>>> t = table7()
7 14 21 28 35 42 49 56 63 70
>>> print(t)
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons.

```
>>> def table7triple():
    t = list()
    print('La table par 7 en triple exemplaire :')
    t += table7()
    t += table7()
    t += table7()
    return t

>>> table7triple()
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70

7 14 21 28 35 42 49 56 63 70

7 14 21 28 35 42 49 56 63 70

[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 7, 14, 21, 28, 35, 42, 49,
56, 63, 70, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc.

- Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.
- Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent.

La fonction avec paramètres

Une fonction est une partie d'un programme qui fonctionne indépendamment du reste du programme. Elle reçoit une liste de paramètres et retourne un résultat. Le corps de la fonction désigne toute instruction du programme qui est exécutée si la fonction est appelée.

```
def fonction_nom (par_1, ..., par_n) :
    instruction_1
    ...
    instruction_n
    return res_1, ..., res_n
```

Dans la définition d'une telle fonction, il faut prévoir une ou plusieurs variables pour recevoir les arguments transmis. Une telle variable particulière s'appelle un paramètre. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude, et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

```
x_1, ..., x_n = fonction_nom(val_1, val_2, ..., val_n)
```

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer l'information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un argument.

```
>>> import math
>>> def coordonnees_polaires (x,y):
    rho    = math.sqrt(x*x+y*y)
    theta  = math.atan2 (y,x)
    return rho, theta

>>> def affichage (x,y):
    r,t = coordonnees_polaires(x,y)
    print ("cartésien (%f,%f) --> polaire (%f,%f degrés)" \
          % (x,y,r,math.degrees(t)))

>>> affichage (1,1)
cartésien (1.000000,1.000000) --> polaire (1.414214,45.000000 degrés)
>>> affichage (0.5,1)
cartésien (0.500000,1.000000) --> polaire (1.118034,63.434949 degrés)
>>> affichage (-0.5,1)
cartésien (-0.500000,1.000000) --> polaire (1.118034,116.565051 degrés)
>>> affichage (-0.5,-1)
cartésien (-0.500000,-1.000000) --> polaire (1.118034,-116.565051 degrés)
>>> affichage (0.5,-1)
cartésien (0.500000,-1.000000) --> polaire (1.118034,-63.434949 degrés)
```

Les paramètres avec des valeurs par défaut

Lorsqu'une fonction est souvent appelée avec les mêmes valeurs pour ses paramètres, il est possible de spécifier pour ceux-ci une valeur par défaut.

```
def fonction_nom (param_1, \
                  param_2 = valeur_2, ..., param_n = valeur_n) :
```

Où `fonction_nom` est le nom de la fonction. `param_1` à `param_n` sont les noms des paramètres, `valeur_2` à `valeur_n` sont les valeurs par défaut des paramètres `param_2` à `param_n`. La seule contrainte lors de cette définition est que si une valeur par défaut est spécifiée pour un paramètre, alors tous ceux qui suivent devront eux aussi avoir une valeur par défaut.

```
>>> def commander_carte_orange (nom, prenom, \
    paiement = "carte", nombre = 1, zone = 2) :
    print("nom : ",nom)
    print("prénom : ",prenom)
    print("paiement : ", paiement)
    print("nombre : ", nombre)
    print("zone :", zone)

>>> commander_carte_orange("BIZOI", "Razvan", "chèque")
nom : BIZOI
prénom : Razvan
paiement : chèque
nombre : 1
zone : 2
```

L'ordre des paramètres

Lors de l'appel, le nom des paramètres n'intervient plus, supposant que chaque paramètre reçoit pour valeur celle qui a la même position que lui lors de l'appel à la fonction. Il est toutefois possible de changer cet ordre en précisant quel paramètre doit recevoir quelle valeur.

```
x_1, ..., x_n = fonction_nom (param_1 = valeur_1 \
    , ..., param_n = valeur_n)
```

```
>>> commander_carte_orange(prenom="Razvan",nom="BIZOI")
nom : BIZOI
prénom : Razvan
paiement : carte
nombre : 1
zone : 2
```

Cette possibilité est intéressante surtout lorsqu'il y a de nombreux paramètres par défaut et que seule la valeur d'un des derniers paramètres doit être changée.

Atelier 3 - Exercice 1.1- Exercice 1.2- Exercice 1.3- Exercice 1.4

Les paramètres modifiables

Les paramètres de types immuables et modifiables se comportent de manières différentes à l'intérieur d'une fonction. Ces paramètres sont manipulés dans le corps de la fonction, voire modifiés parfois. Selon le type du paramètre, ces modifications ont des répercussions à l'extérieur de la fonction.

Les types immuables ne peuvent être modifiés et cela reste vrai. Lorsqu'une fonction accepte un paramètre de type immuable, elle ne reçoit qu'une copie de sa valeur. Elle peut donc modifier ce paramètre sans que la variable ou la valeur utilisée lors de l'appel de la fonction n'en soit affectée. On appelle ceci un passage de paramètre par valeur. A l'opposé, toute modification d'une variable d'un type modifiable à l'intérieur d'une fonction est répercutée à la variable qui a été utilisée lors de l'appel de cette fonction. On appelle ce second type de passage un passage par adresse.

```
>>> def somme_n_premier_terme(n,liste):
    somme = 0
    for i in liste:
        somme += i
        n -= 1
        # modification de n (type immuable)
        if n <= 0: break
    liste[0] = 0
    # modification de liste (type modifiable)
    return somme

>>> l = [1,2,3,4]
>>> nb = 3
>>> print("avant la fonction ",nb,l)
avant la fonction  3 [1, 2, 3, 4]
>>> s = somme_n_premier_terme (nb,l)
>>> print("après la fonction ",nb,l)
après la fonction  3 [0, 2, 3, 4]
>>> print("somme : ", s)
somme : 6
```

La liste est modifiée à l'intérieur de la fonction. La variable nb est d'un type immuable. Sa valeur a été recopiée dans le paramètre n de la fonction. Toute modification de n à l'intérieur de cette fonction n'a aucune répercussion à l'extérieur de la fonction.

Dans l'exemple précédent, il faut faire distinguer le fait que la liste passée en paramètre ne soit que modifiée et non changée.

```
>>> def fonction (liste):
    liste = list()

>>> liste = [0,1,2]
>>> print(liste)
[0, 1, 2]
>>> fonction (liste)
>>> print(liste)
[0, 1, 2]
```

Il faut considérer dans ce programme que la fonction reçoit un paramètre appelé liste mais utilise tout de suite cet identificateur pour l'associer à un contenu différent.

L'identificateur `liste` est en quelque sorte passé du statut de paramètre à celui de variable locale.

```
>>> def fonction (liste):  
    del liste  
  
>>> print(liste)  
[0, 1, 2]  
>>> fonction (liste)  
>>> print(liste)  
[0, 1, 2]
```

Le programme qui suit permet cette fois-ci de vider la liste passée en paramètre à la fonction. La seule instruction de cette fonction modifie vraiment le contenu désigné par l'identificateur `liste` et cela se vérifie après l'exécution de cette fonction.

```
>>> def fonction (liste):  
    del liste[0:len(liste)]  
  
>>> print(liste)  
[0, 1, 2]  
>>> fonction (liste)  
>>> print(liste)  
[]
```


La fonction récursive

Une fonction récursive est une fonction qui s'appelle elle-même. La fonction récursive la plus fréquemment citée en exemple est la fonction factorielle. Celle-ci met en évidence les deux composantes d'une fonction récursive, la récursion proprement dite et la condition d'arrêt.

```
>>> def factorielle(n) :
        if n == 0 : return 1
        else : return n * factorielle(n-1)
```

La dernière ligne de la fonction factorielle est la récursion tandis que la précédente est la condition d'arrêt, sans laquelle la fonction ne cesserait de s'appeler, empêchant le programme de terminer son exécution. Si celle-ci est mal spécifiée ou absente, l'interpréteur Python affiche une suite ininterrompue de messages.

```
>>> factorielle(9)
362880
>>> factorielle(99)
93326215443944152681699238856266700490715968264381621468592963895217
59999322991560894146397615651828625369792082722375825118521091686400
00000000000000000000
```

Python n'autorise pas plus de 1000 appels récursifs : `factorielle(999)` provoque nécessairement une erreur d'exécution même si la condition d'arrêt est bien spécifiée.

```
>>> factorielle(999)
Traceback (most recent call last):
  File "<pyshell#283>", line 1, in <module>
    factorielle(999)
  File "<pyshell#281>", line 3, in factorielle
    else : return n * factorielle(n-1)
  File "<pyshell#281>", line 3, in factorielle
    else : return n * factorielle(n-1)
  File "<pyshell#281>", line 3, in factorielle
    else : return n * factorielle(n-1)
[Previous line repeated 989 more times]
  File "<pyshell#281>", line 2, in factorielle
    if n == 0 : return 1
RecursionError: maximum recursion depth exceeded in comparison
```

La liste des messages d'erreurs est aussi longue qu'il y a eu d'appels à la fonction récursive. Dans ce cas, il faut transformer cette fonction en une fonction non récursive équivalente, ce qui est toujours possible.

```
>>> def factorielle_non_recursive (n) :
        r = 1
        for i in range (2, n+1) :
            r *= i
        return r

>>> factorielle_non_recursive(999)
40238726007709377354370243392300398571937486421071463254379991042993
85123986290205920442084869694048004799886101971960586316668729948085
58901323829669944590997424504087073759918823627727188732519779505950
99527612087497546249704360141827809464649629105639388743788648733711
...
```

Les variables locales et globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction. Les contenus des variables locales sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

```
>>> def mask():
    p = 20
    print(p, q)

>>> p, q = 15, 38
>>> mask()
20 38
>>> print(p, q)
15 38
>>> def mask():
    p = 20
    print(p, q)
    q = p
    <-----

>>> mask()
Traceback (most recent call last):
  File "<pyshell#249>", line 1, in <module>
    mask()
  File "<pyshell#248>", line 3, in mask
    print(p, q)
UnboundLocalError: local variable 'q' referenced before assignment
```

Nous commençons par définir une fonction très simple (qui n'utilise d'ailleurs aucun paramètre). À l'intérieur de cette fonction, une variable « **p** » est définie, avec 20 comme valeur initiale. Cette variable « **p** » qui est définie à l'intérieur d'une fonction sera donc une variable locale.

Une fois la définition de la fonction terminée, nous revenons au niveau principal pour y définir les deux variables « **p** » et « **q** » auxquelles nous attribuons les contenus 15 et 38. Ces deux variables définies au niveau principal seront donc des variables globales.

Ainsi le même nom de variable « **p** » a été utilisé ici à deux reprises, pour définir deux variables différentes : l'une est globale et l'autre est locale. On peut constater dans la suite de l'exercice que ces deux variables sont bel et bien des variables distinctes, indépendantes, obéissant à une règle de priorité qui veut qu'à l'intérieur d'une fonction, ce sont les variables définies localement qui ont la priorité.

Cela signifie en effet que vous pourrez toujours utiliser une infinité de fonctions sans vous préoccuper le moins du monde des noms de variables qui y sont utilisées : ces variables ne pourront en effet jamais interférer avec celles que vous aurez vous-même définies par ailleurs.

Il est toutefois possible de définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction

« **global** ». Cette instruction permet d'indiquer, à l'intérieur de la définition d'une fonction, quelles sont les variables à traiter globalement.

```
>>> def monter():
    global a
    a = a+1
    print(a)

>>> monter()
Traceback (most recent call last):
  File "<pyshell#252>", line 1, in <module>
    monter()
  File "<pyshell#251>", line 3, in monter
    a = a+1
TypeError: can only concatenate list (not "int") to list

>>> a = 15
>>> monter()
16
>>> monter()
17
>>> print(a)
17
```

L'aide sur la fonction

Le langage Python propose une fonction `help` qui retourne pour chaque fonction un commentaire ou mode d'emploi qui indique comment se servir de cette fonction.

```
>>> help(round)
Help on built-in function round in module builtins:

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0
    digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

Lorsqu'on utilise cette fonction `help` sur une fonction, le message affiché n'est pas des plus explicites. Pour changer ce message, il suffit d'ajouter en première ligne du code de la fonction une chaîne de caractères.

```
>>> def coordonnees_polaires (x,y):
    """convertit des coordonnées cartésiennes \n\
    en coordonnées polaires (x,y) --> (rho,theta)"""
    rho  = math.sqrt(x*x+y*y)
    theta = math.atan2 (y,x)
    return rho, theta

>>> help(coordonnees_polaires)
Help on function coordonnees_polaires in module __main__:

coordonnees_polaires(x, y)
    convertit des coordonnées cartésiennes
    en coordonnées polaires (x,y) --> (rho,theta)
```

Il est conseillé d'écrire ce commentaire pour toute nouvelle fonction avant même que son corps ne soit écrit. L'expérience montre qu'on oublie souvent de l'écrire après.

Atelier 3 - Exercice 1.5- Exercice 1.6- Exercice 1.7- Exercice 1.8

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, 0.11111111111111111]
>>> 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, 18.666666666666667)
```

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 = '&#x26;#x24;#x26;#x21;$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]
```


Les iterators

À chaque fois qu'un objet est utilisé dans une boucle **for**, l'interpréteur génère en interne un « **iterator** » avec lequel il travaille. Un « **iterator** » est un objet qui contient une méthode « **next** » qui est appelée à chaque cycle et qui renvoie la séquence, élément par élément. Lorsqu'il n'y a plus d'éléments, l'iterator déclenche une exception de type « **StopIteration** ».

Les objets « **iterator** » peuvent être créés par le biais de la primitive « **iter** » qui prend en paramètre tout objet compatible avec les itérations.

```
>>> ma_chaine = "-test-"
>>> iterateur_de_ma_chaine = iter ( ma_chaine )
>>> iterateur_de_ma_chaine
<str_iterator object at 0x0000023169079080>
>>> next ( iterateur_de_ma_chaine )
'_'
>>> next ( iterateur_de_ma_chaine )
't'
>>> next ( iterateur_de_ma_chaine )
'e'
>>> next ( iterateur_de_ma_chaine )
's'
>>> next ( iterateur_de_ma_chaine )
't'
>>> next ( iterateur_de_ma_chaine )
'_'
>>> next ( iterateur_de_ma_chaine )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> for i in ma_chaine : print(i)
...
-
t
e
s
t
-
```

Les fonctions générateur

Il est possible d'utiliser une notation abrégée pour créer un générateur, à l'aide d'une expression.

```
>>> genexp = (i for i in range(5) if i % 2 == 0)
>>> next(genexp)
0
>>> next(genexp)
2
>>> next(genexp)
4
>>> next(genexp)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

La création d'un « **iterator** » par le biais d'un générateur se résume à l'écriture d'une fonction qui parcourt les éléments de la séquence. Au lieu de retourner ces éléments par la directive `return`, la fonction doit faire appel à la directive « **yield** », qui sert à définir un point de sauvegarde.

Le mot-clé « **yield** » est un peu à part. Utilisé à l'intérieur d'une fonction, il permet d'interrompre le cours de son exécution à un endroit précis de sorte qu'au prochain appel de cette fonction, celle-ci reprendra le cours de son exécution exactement au même endroit avec des variables locales inchangées. Le mot-clé `return` ne doit pas être utilisé. Ces fonctions ou générateurs sont utilisées en couple avec le mot-clé « **for** » pour simuler un ensemble.

```
>>> def fonction_yield(n):
    i = 0
    while i < n-1:
        print("yield 1 ", i)
        yield i
        i = i+1
    print("yield 2 ", i)
    yield i

>>> for a in fonction_yield(2):
    print(a)

yield 1  0
0
yield 2  1
1
>>> for a in fonction_yield(3):
    print(a)

yield 1  0
0
yield 1  1
1
yield 2  2
2
```

Le programme affiche tous les entiers compris entre 0 et 2 inclus ainsi que le texte « **yield 1** » ou « **yield 2** » selon l'instruction « **yield** » qui a retourné le résultat. Lorsque la fonction a finalement terminé son exécution, le prochain appel agit comme si c'était la première fois qu'on l'appelait.

La compilation dynamique

Identificateur callable

La fonction « **callable** » retourne un booléen permettant de savoir si un identificateur est une fonction, de savoir par conséquent si tel identificateur est callable comme une fonction.

```
>>> x = 5
>>> def y() :
>>>     return None

>>> print(callable(x))
False
>>> #affiche False car x est une variable
>>> print(callable(y))
True
>>> #affiche True car y est une fonction
```

eval

La fonction évalue toute chaîne de caractères contenant une expression écrite avec la syntaxe du langage python. Cette expression peut utiliser toute variable ou toute fonction accessible au moment où est appelée la fonction « **eval** ».

```
>>> x = 3
>>> y = 4
>>> def carre(x) : return x**2

>>> print(eval("carre(x)+carre(y)+2*x*y"))
49
>>> print(carre(x+y))
49
```

Si l'expression envoyée à la fonction « **eval** » inclut une variable non définie, l'interpréteur python génère une erreur.

compile, exec

Plus complète que la fonction « **eval** », la fonction compile permet d'ajouter une ou plusieurs fonctions au programme, celle-ci étant définie par une chaîne de caractères. Le code est d'abord compilé « **compile** » puis incorporé au programme « **exec** ».

```
>>> import math
>>> str = """def coordonnees_polaires(x,y):
>>>     rho      = math.sqrt(x*x+y*y)
>>>     theta = math.atan2 (y,x)
>>>     return rho,  theta"""
>>> # fonction définie par une chaîne de caractères
```

La fonction compile prend en fait trois arguments. Le premier est la chaîne de caractères contenant le code à compiler. Le second paramètre contient un nom de fichier dans lequel seront placées les erreurs de compilation. Le troisième paramètre est une chaîne de caractères à choisir parmi « **exec** » ou « **eval** ». Selon ce

choix, ce sera la fonction « **exec** » ou « **eval** » qui devra être utilisée pour agréger le résultat de la fonction compile au programme.

```
>>> obj = compile(str,"","exec") #fonction compilée
>>> exec(obj) # fonction incorporée au programme
>>> print(coordonnees_polaires(1,1))
(1.4142135623730951, 0.7853981633974483)
```

L'exemple suivant donne un exemple d'utilisation de la fonction compile avec la fonction « **eval** ».

```
>>> import math
>>> str = """math.sqrt(x*x+y*y)"""
>>> obj = compile(str,"","eval")
>>> x = 1
>>> x,y = 1,2
>>> print(eval(obj))
2.23606797749979
```


4

L'utilisation des fichiers

Les fonctions prédéfinies

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'informations plus importante.

Pour que cela devienne possible, nous devons doter nos programmes de divers mécanismes permettant de créer des fichiers, d'y envoyer des données et de les récupérer par la suite.

Les langages de programmation proposent des jeux d'instructions plus ou moins sophistiqués pour effectuer ces tâches.

Travailler avec des fichiers

L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver, puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations, mais généralement vous ne faites pas les deux à la fois. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de pages. Vous lisez la plupart des livres en suivant l'ordre normal des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre.

Tout ce que nous venons de dire des livres s'applique également aux fichiers informatiques. Un fichier se compose de données enregistrées sur votre disque dur, sur une disquette, une clef USB ou un CD. Vous y accédez grâce à son nom lequel peut inclure aussi un nom de répertoire. En première approximation, vous pouvez considérer le contenu d'un fichier comme une suite de caractères, ce qui signifie que vous pouvez traiter ce contenu, ou une partie quelconque de celui-ci, à l'aide des fonctions servant à traiter les chaînes de caractères.

Liste des modules importés

Le dictionnaire modules du module « **sys** » contient l'ensemble des modules importés.

```
>>> import sys
>>> for m in sys.modules:
    print(m, " " * (14 - len(str(m))), sys.modules[m])

builtins      <module 'builtins' (built-in)>
sys           <module 'sys' (built-in)>
...
```

Lorsque le programme stipule l'import d'un module, Python vérifie s'il n'est pas déjà présent dans cette liste. Dans le cas contraire, il l'importe. Chaque module n'est importé qu'une seule fois.

Le dictionnaire « **sys.modules** » peut être utilisé pour vérifier la présence d'un module ou lui assigner un autre identificateur. Un module est un objet qui n'autorise qu'une seule instance.

Les attributs communs à tout module

Une fois importés, tous les modules possèdent cinq attributs qui contiennent des informations comme leur nom, le chemin du fichier correspondant, l'aide associée.

<code>__all__</code>	Contient toutes les variables, fonctions, classes du module.
<code>__builtins__</code>	Ce dictionnaire contient toutes les fonctions et classes inhérentes au langage Python utilisées par le module.
<code>__doc__</code>	Contient l'aide associée au module.
<code>__file__</code>	Contient le nom du fichier qui définit le module. Son extension est pyc.
<code>__name__</code>	Cette variable contient a priori le nom du module sauf si le module est le point d'entrée du programme auquel cas cette variable contient « <code>__main__</code> ».

Ces attributs sont accessibles si le nom du module est utilisé comme préfixe. Sans préfixe, ce sont ceux du module lui-même.

```
>>>
import os
>>> print(os.__name__)
os
>>> print(os.__doc__)
OS routines for NT or Posix depending on what system we're on.

This exports:
- all functions from posix or nt, e.g. unlink, stat, etc.
- os.path is either posixpath or ntpath
- os.name is either 'posix' or 'nt'
- os.curdir is a string representing the current directory (always '.')
- os.pardir is a string representing the parent directory (always '..')
- os.sep is the (or a most common) pathname separator ('/' or '\\')
- os.extsep is the extension separator (always '.')
- os.altsep is the alternate pathname separator (None or '/')
- os.pathsep is the component separator used in $PATH etc
- os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
- os.defpath is the default search path for executables
- os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms. Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).

>>> if __name__ == "__main__" :
        print("ce fichier est le point d'entrée")
else :
        print("ce fichier est importé")

ce fichier est le point d'entrée
```

Le répertoire courant

os

Le module `os` regroupe plusieurs fonctions ou objets qui sont dans certains cas des alias vers des éléments d'autres modules.

On peut regrouper ces éléments en quatre sous-ensembles :

- opérations sur les descripteurs de fichiers ;
- manipulation des fichiers et répertoires ;
- manipulation des processus ;
- informations sur le système.

Pour simplifier les explications qui vont suivre, nous indiquerons seulement des noms simples pour les fichiers que nous allons manipuler. Si vous procédez ainsi dans vos exercices, les fichiers en question seront créés et/ou recherchés par Python dans le répertoire courant. Si vous travaillez avec IDLE, vous souhaitez donc certainement forcer Python à changer son répertoire courant, afin que celui-ci corresponde à vos attentes. Même si vous travaillez sous Windows, vous pouvez utiliser « `/` » et non « `\` » en guise de séparateurs. Python effectuera automatiquement les conversions nécessaires, suivant que vous travaillez sous Mac OS, Linux, ou Windows.

```
>>> from os import chdir, getcwd
>>> chdir("F:/PYT - Python programmation objet/Scripts")
>>> rep_cour = getcwd()
>>> print(rep_cour)
F:\PYT - Python programmation objet\Scripts
```

La première commande importe la fonction « `chdir` » du module « `os` ». La seconde commande provoque le changement de répertoire « `chdir` ».

- Vous avez également la possibilité d'insérer ces commandes en début de script, ou encore d'indiquer le chemin d'accès complet dans le nom des fichiers que vous manipulez, mais cela risque peut-être d'alourdir l'écriture de vos programmes.
- Choisissez de préférence des noms de fichiers courts. Évitez dans toute la mesure du possible les caractères accentués, les espaces et les signes typographiques spéciaux. Dans les environnements de travail de type Unix (Mac OS, Linux, BSD...), il est souvent recommandé de n'utiliser que des caractères minuscules.

La lecture des fichiers d'un répertoire :

```
>>> repertoire = r'D:\donnees\meteo\2023'
>>> for fichier in os.listdir(repertoire):
...     if not os.path.isdir(os.path.join(repertoire, fichier)):
...         print(fichier)
...
synop.202301.csv
synop.202302.csv
synop.202303.csv
synop.202304.csv
synop.202305.csv
synop.202306.csv
...
```

La lecture des tous les fichiers d'une arborescence :

```
>>> import os
>>> for dirname, _, filenames in os.walk('meteo'):
...     for filename in filenames:
...         print(os.path.join(dirname, filename))
...
meteo\postesSynop.csv
meteo\1996\synop.199601.csv
meteo\1996\synop.199602.csv
meteo\1996\synop.199603.csv
meteo\1996\synop.199604.csv
meteo\1996\synop.199605.csv
meteo\1996\synop.199606.csv
meteo\1996\synop.199607.csv
meteo\1996\synop.199608.csv
meteo\1996\synop.199609.csv
meteo\1996\synop.199610.csv
meteo\1996\synop.199611.csv
meteo\1996\synop.199612.csv
meteo\1997\synop.199701.csv
meteo\1997\synop.199702.csv
meteo\1997\synop.199703.csv
meteo\1997\synop.199704.csv
...
```

Les deux formes d'importation

Les lignes d'instructions que nous venons d'utiliser sont l'occasion d'expliquer un mécanisme intéressant. Vous savez qu'en complément des fonctions intégrées dans le module de base, Python met à votre disposition une très grande quantité de fonctions plus spécialisées, qui sont regroupées dans des modules.

Pour utiliser les fonctions d'un module, il suffit de les importer. Mais cela peut se faire de deux manières différentes, comme nous allons le voir ci-dessous. Chacune des deux méthodes présente des avantages et des inconvénients.

```
>>> import os
>>> rep_cour = os.getcwd()
>>> print(rep_cour)
F:\PYT - Python programmation objet\Scripts
```

Nous pouvons même importer toutes les fonctions d'un module, comme dans :

```
from tkinter import *
```

Cette méthode d'importation présente l'avantage d'alléger l'écriture du code. Elle présente l'inconvénient (surtout dans sa dernière forme, celle qui importe toutes les fonctions d'un module) d'encombrer l'espace de noms courant. Il se pourrait alors que certaines fonctions importées aient le même nom que celui d'une variable définie par vous-même, ou encore le même nom qu'une fonction importée depuis un autre module. Si cela se produit, l'un des deux noms en conflit n'est évidemment plus accessible.

L'écriture séquentielle

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un objet-interface particulier, que l'on appelle objet-fichier. On crée cet objet à l'aide de la fonction intégrée « **open** ». Celle-ci renvoie un objet doté de méthodes spécifiques, qui vous permettront de lire et écrire dans le fichier.

L'exemple ci-après vous montre comment ouvrir un fichier en écriture, y enregistrer deux chaînes de caractères, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé automatiquement. Par contre, si le nom utilisé concerne un fichier préexistant qui contient déjà des données, les caractères que vous y enregistrerez viendront s'ajouter à la suite de ceux qui s'y trouvent déjà. Vous pouvez faire tout cet exercice directement à la ligne de commande :

```
>>> import os
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> print(os.getcwd())
F:\PYT - Python programmation objet\Scripts
>>> with open('Monfichier.txt', 'a') as obFichier:
>>>     obFichier.write('Bonjour, fichier !')
>>>     obFichier.write("Quel beau temps, aujourd'hui !")
>>> print(os.listdir(path='.'))
['Monfichier.txt']
>>> print(os.path.isfile("F:/PYT - Python programmation
objet/Scripts/Monfichier.txt"))
True
>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
>>>     print('Le dossier actuel est ' + repN)
>>>     for fichier in fichiers:
>>>         print('Fichier -- ' + repN + ': ' + fichier)
>>>     print('')

Le dossier actuel est F:\PYT - Python programmation objet\Scripts
Fichier -- F:\PYT - Python programmation objet\Scripts: Monfichier.txt
```

La fonction « **open** » attend deux arguments, qui doivent tous deux être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, et le second est le mode d'ouverture.

La méthode « **write** » réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Ces données sont enregistrées dans le fichier les unes à la suite des autres. Chaque nouvel appel de « **write** » continue l'écriture à la suite de ce qui est déjà enregistré.

La méthode « **close** » referme le fichier. Celui-ci est désormais disponible pour tout usage.

Mode	Signification
r	Ouverture du fichier en lecture.
w	Ouverture du fichier en écriture.
x	Ouverture du fichier en création exclusive.
a	Ouverture du fichier en mise à jour (append). Si le fichier n'existe pas, il est créé. S'il existe déjà, les nouvelles informations sont ajoutées à la fin.
+	Ouverture du fichier en lecture et en écriture.

t	Ouverture du fichier en lecture et en écriture. Si le fichier n'existe pas, il est créé. S'il existe déjà, son contenu est écrasé.
b	Ouverture du fichier en lecture et en mise à jour. Si le fichier n'existe pas, il est créé. S'il existe déjà, les nouvelles informations sont ajoutées à la fin.

Lecture séquentielle d'un fichier

Vous allez maintenant ouvrir le fichier, mais cette fois en lecture, de manière à pouvoir y relire les informations que vous avez enregistrées dans l'étape précédente :

```
>>> import os
>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
    print('Le dossier actuel est ' + repN)
    for fichier in fichiers:
        print('Fichier -- ' + repN + ': ' + fichier)
    print('')

Le dossier actuel est F:\PYT - Python programmation objet\Scripts
Fichier -- F:\PYT - Python programmation objet\Scripts: Monfichier.txt
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> def lireFichier (nomFichier):
    with open('Monfichier.txt', 'at') as ofi:
        t = ofi.read()
        print(t)

>>> lireFichier('Monfichier.txt')
Bonjour, fichier !Quel beau temps, aujourd'hui !
```

Comme on pouvait s'y attendre, la méthode « **read** » lit les données présentes dans le fichier et les transfère dans une variable de type « **string** ». Si on utilise cette méthode sans argument, la totalité du fichier est transférée.

La méthode « **read** » peut également être utilisée avec un argument. Celui-ci indiquera combien de caractères doivent être lus, à partir de la position déjà atteinte dans le fichier.

```
>>> ofi = open('Monfichier.txt', 'r')
>>> t = ofi.read(7)
>>> print(t)
Bonjour
>>> print(ofi.read(15))
, fichier !Quel
>>> print(ofi.read(1000))
beau temps, aujourd'hui !
```

Si la fin du fichier est déjà atteinte, « **read** » renvoie une chaîne vide.

```
>>> t = ofi.read()
>>> print(t)

>>> if len(t) <= 0 : ofi.close()
>>> ofi.closed
True
```

Dans tout ce qui précède, nous avons admis sans explication que les chaînes de caractères étaient échangées telles quelles entre l'interpréteur Python et le fichier. En réalité, ceci est inexact, parce que les séquences de caractères doivent être converties en séquences d'octets pour pouvoir être mémorisées dans les fichiers. De plus, il existe malheureusement différentes normes pour cela. En toute rigueur, il faudrait donc préciser à Python la norme d'encodage que vous souhaitez utiliser dans vos fichiers.

La copie d'un fichier

Il va de soi que les boucles de programmation s'imposent lorsque l'on doit traiter un fichier dont on ne connaît pas nécessairement le contenu à l'avance. L'idée de base consistera à lire ce fichier morceau par morceau, jusqu'à ce que l'on ait atteint la fin du fichier.

```
>>> def copieFichier(source, destination):
    "copie intégrale d'un fichier"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.read(50)
        if txt == "":
            break
        fd.write(txt)
    fs.close()
    fd.close()
    return

>>> copieFichier('Monfichier.txt', 'MonfichierC.txt')
>>> for repN, susRep, fichiers in os.walk(os.getcwd()):
    print('Le dossier actuel est ' + repN)
    for fichier in fichiers:
        print('Fichier -- ' + repN + ': ' + fichier)
    print('')

Le dossier actuel est F:\PYT - Python programmation objet\Scripts
Fichier -- F:\PYT - Python programmation objet\Scripts: Monfichier.txt
Fichier -- F:\PYT - Python programmation objet\Scripts: MonfichierC.txt
>>> lireFichier ('MonfichierC.txt')
Bonjour, fichier !Quel beau temps, aujourd'hui !
```

Vous savez en effet que l'instruction « **while** » doit toujours être suivie d'une condition à évaluer ; le bloc d'instructions qui suit est alors exécuté en boucle, aussi longtemps que cette condition reste vraie. Or nous avons remplacé ici la condition à évaluer par une simple constante, et vous savez également que l'interpréteur Python considère comme vraie toute valeur numérique différente de zéro. Nous pouvons cependant interrompre ce bouclage en faisant appel à l'instruction « **break** », laquelle permet éventuellement de mettre en place plusieurs mécanismes de sortie différents pour une même boucle.

Les fichiers texte

Un fichier texte est un fichier qui contient des caractères « **imprimables** » et des espaces organisés en lignes successives, ces lignes étant séparées les unes des autres par un caractère spécial non imprimable appelé marqueur de fin de ligne.

Les fichiers texte sont donc des fichiers que nous pouvons lire et comprendre à l'aide d'un simple éditeur de texte, par opposition aux fichiers binaires dont le contenu est – au moins en partie – inintelligible pour un lecteur humain, et qui ne prend son sens que lorsqu'il est décodé par un logiciel spécifique. Par exemple, les fichiers contenant des images, des sons, des vidéos, etc. sont presque toujours des fichiers binaires. Nous donnons un petit exemple de traitement de fichier binaire un peu plus loin, mais dans le cadre de ce cours, nous nous intéresserons presque exclusivement aux fichiers texte.

Il est très facile de traiter des fichiers texte avec Python. Par exemple, les instructions suivantes suffisent pour créer un fichier texte de quatre lignes :

```
>>> import os
>>> nomFichier='Fichiertexte'
>>> os.chdir(r"F:\00--Développement avec un langage
fonctionnel\Scripts")
>>>
>>> with open(nomFichier, 'at') as fichier:
...     fichier.write("Ceci est la ligne un\nVoici la ligne deux\n")
...     fichier.write("Voici la ligne trois\nVoici la ligne
quatre\n")
...     fichier.write("#Voici la ligne quatre\n")
...
>>> with open(nomFichier, 'r') as fichier:
...     print(fichier.read())
...
Ceci est la ligne un
Voici la ligne deux
Voici la ligne trois
Voici la ligne quatre
#Voici la ligne quatre
```

Notez bien le marqueur de fin de ligne « `\n` » inséré dans les chaînes de caractères, aux endroits où l'on souhaite séparer les lignes de texte dans l'enregistrement.

Lors des opérations de lecture, les lignes d'un fichier texte peuvent être extraites séparément les unes des autres à l'aide de la méthode « **readline** ».

La méthode « **readlines** » transfère toutes les lignes restantes dans une liste de chaînes. Ainsi cette méthode permet donc de lire l'intégralité d'un fichier en une instruction seulement. Cela n'est possible toutefois que si le fichier à lire n'est pas trop gros : puisqu'il est copié intégralement dans une variable, c'est-à-dire dans la mémoire vive de l'ordinateur.

```
>>> with open(nomFichier, 'r') as fichier:
...     print(fichier.readlines())
...
['Ceci est la ligne un\n', 'Voici la ligne deux\n', 'Voici la ligne
trois\n', 'Voici la ligne quatre\n', '#Voici la ligne quatre\n']

>>> def filtre(source,destination):
...     "recopier un fichier en"
```



```

"éliminant les lignes de remarques"
fs = open(source, 'r')
fd = open(destination, 'w')
while 1:
    txt = fs.readline()
    if txt == '':
        break
    if txt[0] != '#':
        fd.write(txt)
fs.close()
fd.close()
return

>>> filtre(nomFichier, 'Fichiertexte.txt')
>>> f = open('Fichiertexte.txt', 'r')
>>> print(f.read())
Ceci est la ligne un
Voici la ligne deux
Voici la ligne trois
Voici la ligne quatre

>>> f.close()

```

La fonction « **open** » renvoie un objet de type file, qui contient les méthodes suivantes.

<code>close()</code>	ferme le flux.
<code>flush()</code>	vide le tampon interne.
<code>fileno()</code>	renvoie le descripteur de fichier.
<code>isatty()</code>	renvoie vrai si le fichier est branché sur un terminal tty.
<code>next()</code>	renvoie la prochaine ligne lue, ou provoque une exception <code>StopIteration</code> .
<code>read([size])</code>	lit au plus <code>size</code> octets. Si <code>size</code> est omis, lit tout le contenu.
<code>readline([size])</code>	lit la prochaine ligne. Si <code>size</code> est fourni, limite le nombre d'octets lus.
<code>readlines([sizehint])</code>	appelle « readline » en boucle, jusqu'à la fin du flux. Si « sizehint » est fourni, s'arrête lorsque ce nombre est atteint ou dépassé par la ligne en cours.
<code>seek(offset[, whence])</code>	positionne le curseur de lecteur en fonction de la valeur d'offset. <code>whence</code> permet de faire varier le fonctionnement (0 : position absolue – valeur par défaut, 1 : relative à la position courante, 2 : relative à la fin du fichier).
<code>tell()</code>	renvoie la position courante.
<code>truncate([size])</code>	tronque la taille du fichier. Si <code>size</code> est fourni, détermine la taille maximum.
<code>write(str)</code>	écrit la chaîne « str » dans le fichier.
<code>writelines(sequence)</code>	écrit la séquence de chaînes.

Les objets de type file sont des itérateurs, qui peuvent donc être utilisés directement comme des séquences.

```

>>> import os
>>> os.chdir("F:\PYT - Python programmation objet\Scripts")
>>> mon_fichier = open('infos.txt', mode='w', encoding='utf-8')

```

```

>>> mon_fichier.write('1. première info\n')
17
>>> mon_fichier.write('2. deuxième info\n')
17
>>> mon_fichier.write('3. troisième info\n')
18
>>> mon_fichier.write('4. 圖形碼常用字次常用字\n')
14
>>> mon_fichier.close()
>>> mon_fichier = open('infos.txt', 'r')
>>> for line in mon_fichier:
    print(line)

1. premiÃ`re info
2. deuxiÃ`me info
3. troixiÃ`me info
4. âœ-â½çç¼â,,ç""â--æ¡â,,ç""â--

>>> mon_fichier.close()
>>> print(mon_fichier.name,\
        mon_fichier.encoding, \
        mon_fichier.newlines.encode(), \
        mon_fichier.closed)
infos.txt cpl252 b'\r\n' True
>>> mon_fichier = open('infos.txt', mode='r', encoding='utf-8')
>>> for line in mon_fichier:
    print(line)

1. première info
2. deuxième info
3. troisième info
4. 圖形碼常用字次常用字

>>> mon_fichier.close()
>>> print(mon_fichier.name,\
        mon_fichier.encoding, \
        mon_fichier.newlines.encode(), \
        mon_fichier.closed)
infos.txt utf-8 b'\r\n' True

```

Le type file possède en outre un certain nombre d'attributs.

Closed	renvoie vrai si le fichier a été fermé.
Encoding	renvoie l'encoding utilisé par le fichier pour l'écriture. Si des chaînes unicode sont écrites dans le flux, elles sont encodées avec ce codec.
Name	renvoie le nom du fichier.
Newlines	renvoie le type de passage à la ligne utilisé (\r, \n, ou \r\n)

L'enregistrement et restitution de variables

L'argument de la méthode « **write** » utilisée avec un fichier texte doit être une chaîne de caractères. Avec ce que nous avons appris jusqu'à présent, nous ne pouvons donc enregistrer d'autres types de valeurs qu'en les transformant d'abord en chaînes de caractères « **string** ». Nous pouvons réaliser cela à l'aide de la fonction intégrée « **str** ».

```
>>> import os
>>> nomFichier='FichierFormate.txt'
>>> os.chdir("F:/PYT - Python programmation objet/Scripts")
>>> f = open(nomFichier, 'w')
>>> 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\n" \
        % (x,d,s, str(x+4))
>>> f.write(res)
121
>>> res = "un nombre réel " + str(x) + \
        " et un entier " + str(d) + \
        ", une chaîne de " + s + \
        ",\nun réel d'abord converti en chaîne de caractères " \
        + str(x+4)
>>> f.write(res)
114
>>> f.close()
>>> f = open(nomFichier, 'r')
>>> print(f.read())
un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
un nombre réel 5.5 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
>>> f.close()
```

Si nous enregistrons les valeurs numériques en les transformant d'abord en chaînes de caractères, nous risquons de ne plus pouvoir les retransformer correctement en valeurs numériques lorsque nous allons relire le fichier.

Il existe plusieurs solutions à ce genre de problèmes. L'une des meilleures consiste à importer un module Python spécialisé : le module « **pickle** ».

```
>>> import pickle
>>> a, b, c = 27, 12.96, [5, 4.83, "René"]
>>> f = open('donnees_binaires', 'wb')
>>> pickle.dump(a, f)
>>> pickle.dump(b, f)
>>> pickle.dump(c, f)
>>> f.close()
>>> f = open('donnees_binaires', 'rb')
>>> j = pickle.load(f)
>>> k = pickle.load(f)
>>> l = pickle.load(f)
>>> print(j, type(j))
27 <class 'int'>
>>> print(k, type(k))
```

```
12.96 <class 'float'>
>>> print(1, type(1))
[5, 4.83, 'René'] <class 'list'>
```

Comme vous pouvez le constater dans ce court exemple, le module « **pickle** » permet d'enregistrer des données avec conservation de leur type.

Attention, les fichiers traités à l'aide des fonctions du module « **pickle** » ne seront pas des fichiers texte, mais bien des fichiers binaires. Pour cette raison, ils doivent obligatoirement être ouverts comme tels à l'aide de la fonction « **open** ». Vous utiliserez l'argument « **wb** » pour ouvrir un fichier binaire en, et l'argument « **rb** » pour ouvrir un fichier binaire en lecture.

La fonction « **dump** » du module « **pickle** » attend deux arguments : le premier est la variable à enregistrer, le second est l'objet fichier dans lequel on travaille. La fonction « **load** » effectue le travail inverse, c'est-à-dire la restitution de chaque variable avec son type.

De la même manière le module « **shelve** » vous permettra de écrire d'ajouter ou lire les données sauvegardées.

```
>>> import shelve
>>> shelfFile = shelve.open('shelveFichier')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
>>> shelfFile = shelve.open('shelveFichier')
>>> print(shelfFile['cats'], type(shelfFile))
['Zophie', 'Pooka', 'Simon'] <class 'shelve.DbfilenameShelf'>
>>> shelfFile.close()
>>> shelfFile = shelve.open('shelveFichier')
>>> fruits = ['apples', 'oranges', 'cherries', 'banana']
>>> pers = ['Alice', 'Bob', 'Carol', 'David']
>>> animaux = ['dogs', 'cats', 'moose', 'goose']
>>> shelfFile['fruits'] = fruits
>>> shelfFile['pers'] = pers
>>> shelfFile['animaux'] = animaux
>>> shelfFile.close()
>>> shelfFile = shelve.open('shelveFichier')
>>> list(shelfFile.keys())
['cats', 'fruits', 'pers', 'animaux']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon'], ['apples', 'oranges', 'cherries',
'banana'], ['Alice', 'Bob', 'Carol', 'David'], ['dogs', 'cats',
'moose', 'goose']]
>>> for categorie in shelfFile.keys() :
    print(shelfFile[categorie], type(shelfFile[categorie]))

['Zophie', 'Pooka', 'Simon'] <class 'list'>
['apples', 'oranges', 'cherries', 'banana'] <class 'list'>
['Alice', 'Bob', 'Carol', 'David'] <class 'list'>
['dogs', 'cats', 'moose', 'goose'] <class 'list'>
```

Après avoir exécuté le code précédent sur Windows, vous verrez trois nouveaux fichiers dans le répertoire de travail courant: shelveFichier.bak, shelveFichier.dat et shelveFichier.dir.

Les exceptions try – except – else

Les exceptions sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue, et un message d'erreur plus ou moins explicite est affiché.

```
>>> print(10/0)
Traceback (most recent call last):
  File "<pyshell#340>", line 1, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le type d'erreur, et ensuite une information spécifique de cette erreur.

Dans de nombreux cas, il est possible de prévoir à l'avance certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible d'associer un mécanisme de surveillance à tout un ensemble d'instructions, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.

Un mécanisme de ce type s'appelle en général mécanisme de traitement des exceptions. Celui de Python utilise l'ensemble d'instructions « **try – except – else** », qui permettent d'intercepter une erreur et d'exécuter une portion de script spécifique de cette erreur. Il fonctionne comme suit.

Le bloc d'instructions qui suit directement une instruction « **try** » est exécuté par Python sous réserve. Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction « **except** ». Si aucune erreur ne s'est produite dans les instructions qui suivent « **try** », alors c'est le bloc qui suit l'instruction « **else** » qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

```
>>> try:
    print(10/0)
except:
    print("La division par zéro n'est pas acceptée")
```

La division par zéro n'est pas acceptée

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, lequel fichier étant destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se plante. Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
>>> def existe(fname):
    try:
        f = open(fname, 'r')
        f.close()
        return 1
    except:
        return 0
```

```
>>> filename = input("Veuillez entrer le nom du fichier : ")
Veuillez entrer le nom du fichier : Il n existe pas
>>> if existe(filename):
    print("Ce fichier existe bel et bien.")
else:
    print("Le fichier-", filename, "-est introuvable.")
```

Le fichier-Il n existe pas-est introuvable.

Il est également possible de faire suivre l'instruction « **try** » de plusieurs blocs « **except** », chacun d'entre eux traitant un type d'erreur spécifique.

Il est aussi possible d'ajouter une clause qui sert de préfixe à une liste d'instructions qui ne sera exécutée que si aucune exception n'est déclenchée.

```
try:
    # ... instructions à protéger
except:
    # ... que faire en cas d'erreur
else:
    # ... que faire lorsque aucune erreur n'est apparue

>>> def inverse(x):
...     y = 1.0 / x
...     return y
...
>>> try:
...     print(inverse(2)) # pas d'erreur
... except:
...     print("le programme a déclenché une erreur")
... else:
...     print("tout s'est bien passé")
...
0.5
tout s'est bien passé
```

Le type d'exception

Parfois, plusieurs types d'erreurs peuvent être déclenchés à l'intérieur d'une portion de code protégée. Pour avoir une information sur ce type, il est possible de récupérer une variable de type « **Exception** ».

```
>>> def inverse(x):
...     return 1.0 / x
...
>>> try:
...     print(inverse(2))
...     print(inverse(0))
... except Exception as exc:
...     print("exception de type ", exc.__class__)
...     # affiche exception de type exceptions.ZeroDivisionError
...     print("message", exc)
...     # affiche le message associé à l'exception
...
0.5
exception de type <class 'ZeroDivisionError'>
message float division by zero
```

La variable « **exc** » est en fait une instance d'une classe d'erreur, « **__class__** » correspond au nom de cette classe. A l'aide de la fonction « **isinstance** », il est possible d'exécuter des traitements différents selon le type d'erreur.

```
>>> try:
...     print((-2.1) ** 3.1) # première erreur
...     print(inverse(2))
...     print(inverse(0)) # seconde erreur
... except Exception as exc:
...     if isinstance(exc, ZeroDivisionError):
...         print("division par zéro")
...     else:
...         print("erreur insoupçonnée :", exc.__class__)
...         print("message", exc)
...
(-9.48606594010979-3.0822096637057887j)
0.5
division par zéro
```

Une autre syntaxe plus simple permet d'attraper un type d'exception donné en accolant au mot-clé « **except** » le type de l'exception qu'on désire attraper.

```
try:
    # ... instructions à protéger
except type_exception_1:
    # que faire en cas d'erreur de
    #     type type_exception_1
except (type_exception_i, type_exception_j):
    # que faire en cas d'erreur de type
    #     type_exception_i ou type_exception_j
```

```
except type_exception_n:
    # que faire en cas d'erreur de
    #     type type_exception_n
except:
    # que faire en cas d'erreur d'un type
    #     différent de tous les précédents types
else:
    # que faire lorsqu'aucune erreur n'est apparue

>>> try:
...     print((-2.1) ** 3.1)
...     print(inverse(0))
... except ZeroDivisionError:
...     print("division par zéro")
... except Exception as exc:
...     print("erreur insoupçonnée :", exc.__class__)
...     print("message ", exc)
...
(-9.48606594010979-3.0822096637057887j)
division par zéro
```


Les d'exceptions standards

Le langage Python propose une liste d'exceptions standards. Lorsqu'une erreur ne correspond pas à l'une de ces exceptions, il est possible de créer une exception propre à un certain type d'erreur. Lorsqu'une fonction ou une méthode déclenche une exception non standard, généralement, le commentaire qui lui est associé l'indique. Voici quelques types d'exception courantes :

Nom Erreur	Description
AttributeError	Une référence à un attribut inexistant ou une affectation a échoué.
OSError	Une opération concernant les entrées/sorties (Input/Output) a échoué. Cette erreur survient par exemple lorsqu'on cherche à lire un fichier qui n'existe pas.
ImportError	Cette erreur survient lorsqu'on cherche à importer un module qui n'existe pas.
IndentationError	L'interpréteur ne peut interpréter une partie du programme à cause d'un problème d'indentation. Il n'est pas possible d'exécuter un programme mal indenté mais cette erreur peut se produire lors de l'utilisation de la fonction compile.
IndexError	On utilise un index erroné pour accéder à un élément d'une liste, d'un dictionnaire ou de tout autre tableau.
KeyError	Une clé est utilisée pour accéder à un élément d'un dictionnaire dont elle ne fait pas partie.
NameError	On utilise une variable, une fonction, une classe qui n'existe pas.
TypeError	Erreur de type, une fonction est appliquée sur un objet qu'elle n'est pas censée manipuler.
UnicodeError	Erreur de conversion d'un encodage de texte à un autre.
ValueError	Cette exception survient lorsqu'une valeur est inappropriée pour une certaine opération, par exemple, l'obtention du logarithme d'un nombre négatif.

```
>>> try:
...     d = ["un", "deux"]
...     print(d[2])
...     print(inverse(0))
... except IndexError:
...     print("index erroné")
... except ZeroDivisionError:
...     print("division par zéro")
... except Exception as exc:
...     print("erreur insoupçonnée :", exc.__class__)
...
index erroné
```

Lancer une exception

Lorsqu'une fonction détecte une erreur, il lui est possible de déclencher une exception par l'intermédiaire du mot-clé « **raise** ». Cette exception est attrapée plus bas.

Il est parfois utile d'associer un message à une exception afin que l'utilisateur ne soit pas perdu. Le déclenchement d'une exception suit la syntaxe suivante :

```
raise exception_type(message)
```

```
>>> def inverse(x):
...     if x == 0:
...         raise ValueError(
...             "valeur nulle interdite, fonction inverse")
...     y = 1.0 / x
...     return y
...
>>> try:
...     print(inverse(0)) # erreur
... except ValueError as exc:
...     print("erreur, message :", exc)
...
erreur, message : valeur nulle interdite, fonction inverse
```

Les instructions try, except imbriquées

Comme pour les boucles, il est possible d'imbriquer les portions protégées de code les unes dans les autres. Dans l'exemple qui suit, la première erreur est l'appel à une fonction non définie, ce qui déclenche l'exception `NameError`.

```
>>> def inverse(x):
...     return 1.0 / x
...
>>> try:
...     try:
...         print(inverses(0))
...         # fonction inexistante --> exception NameError
...         print(inverse(0))
...         # division par zéro --> ZeroDivisionError
...     except NameError:
...         print("appel à une fonction non définie")
... except ZeroDivisionError as exc:
...     print("erreur", exc)
...
appel à une fonction non définie

>>> try:
...     try:
...         print(inverse(0))
...         # division par zéro --> ZeroDivisionError
...         print(inverses(0))
...         # fonction inexistante --> exception NameError
...     except NameError:
...         print("appel à une fonction non définie")
... except ZeroDivisionError as exc:
...     print("erreur", exc)
...
erreur float division by zero
```

Une autre imbrication possible est l'appel à une fonction qui inclut déjà une partie de code protégée.

```
>>> def inverse(x):
...     try:
...         y = 1.0 / x
...     except ZeroDivisionError as exc:
...         print("erreur ", exc)
...         if x > 0:
...             return 1000000000
...         else:
...             return -1000000000
...     return y
...
>>> try:
...     print(inverse(0))
...     # division par zéro --> la fonction inverse sait gérer
...     print(inverses(0))
...     # fonction inexistante --> exception NameError
```

```
... except NameError:
...     print("appel à une fonction non définie")
...
erreur float division by zero
-10000000000
appel à une fonction non définie
```

Les fichiers compressés

Les fichiers « **zip** » ou « **gzip** » sont très répandus de nos jours et constituent un standard de compression facile d'accès quelque soit l'ordinateur et son système d'exploitation. Le langage Python propose quelques fonctions pour compresser et décompresser ces fichiers par l'intermédiaire du module « **zipfile** » respectif « **gzip** ».

Création d'une archive

Pour créer un fichier « **zip** », le procédé ressemble à la création de n'importe quel fichier. La seule différence provient du fait qu'il est possible de stocker le fichier à compresser sous un autre nom à l'intérieur du fichier « **zip** », ce qui explique les deux premiers arguments de la méthode « **write** ». Le troisième paramètre indique si le fichier doit être compressé « **zipfile.ZIP_DEFLATED** » ou non « **zipfile.ZIP_STORED** ».

```
>>> import os, zipfile
>>> os.chdir(r"F:\Scripts")
>>> os.mkdir(r"F:\Scripts\repCompress")
>>>
>>> def compressRepertoire(nomRepertoire,
...                         archive=r"F:\Scripts\repCompress\compressRep.zip") :
...     with zipfile.ZipFile (archive, "w") as file :
...         for fichier in os.listdir(nomRepertoire):
...             if not os.path.isdir(os.path.join(
...                                     nomRepertoire, fichier)):
...                 file.write(fichier,fichier,
...                             zipfile.ZIP_DEFLATED)
...     return archive

>>> def afficheFichiers(nomRepertoire) :
...     for fichier in os.listdir(nomRepertoire):
...         if not os.path.isdir(os.path.join(
...                                 nomRepertoire, fichier)):
...             print(fichier)

>>> afficheFichiers(os.getcwd())
donnees_binaires
FichierB
FichierB.txt
FichierB1
FichierFormate.txt
FichierTexte
FichierTexte.txt
infos.txt
Monfichier.txt
MonfichierC.txt
Premier essai de script Python.py
shelveFichier.bak
...

>>> fichier = compressRepertoire(os.getcwd())
```

Lecture d'une archive

L'exemple suivant permet par exemple d'obtenir la liste des fichiers inclus dans un fichier zip

```
>>> def listeArchive(archive):
...     with zipfile.ZipFile(archive, "r") as file:
...         for info in file.infolist():
...             print(info.filename, info.date_time, info.file_size)
...
>>> def lireFichierArchive(nomFichier, archive):
...     with zipfile.ZipFile(archive, "r") as file:
...         print(file.read(nomFichier).decode())
...

>>> listeArchive(fichier)
donnees_binaires (2017, 3, 25, 23, 9, 6) 48
FichierB (2017, 3, 25, 21, 43, 46) 30
FichierB.txt (2017, 3, 25, 21, 35, 8) 30
FichierBl (2017, 3, 25, 22, 16, 8) 44
FichierFormate.txt (2017, 3, 25, 23, 3, 36) 237
Fichiertexte (2017, 3, 25, 16, 16, 54) 112
Fichiertexte.txt (2017, 3, 25, 16, 22, 16) 88
infos.txt (2017, 3, 26, 11, 44, 52) 93
Monfichier.txt (2017, 3, 25, 15, 29, 54) 48
MonfichierC.txt (2017, 3, 25, 15, 53, 8) 48
Premier essai de script Python.py (2017, 3, 26, 20, 45, 38) 566
shelveFichier.bak (2017, 3, 25, 23, 52, 2) 81
shelveFichier.dat (2017, 3, 25, 23, 51, 58) 1590
shelveFichier.dir (2017, 3, 25, 23, 52, 2) 81
test.txt (2017, 3, 26, 18, 38, 34) 18
>>> lireFichierArchive('infos.txt', fichier)
1. première info
2. deuxième info
3. troisième info
4. 圖形碼常用字次常用字
```

La manipulation de fichiers

Le module « **os.path** » propose plusieurs fonctions très utiles qui permettent entre autres de tester l'existence d'un fichier, d'un répertoire, de récupérer diverses informations comme sa date de création, sa taille...

abspath (path)	Retourne le chemin absolu d'un fichier ou d'un répertoire.
commonprefix (list)	Retourne le plus grand préfixe commun à un ensemble de chemins.
dirname (path)	Retourne le nom du répertoire.
exists (path)	Dit si un chemin est valide ou non.
getatime (path) getmtime (path) getctime (path)	Retourne diverses dates concernant un chemin, date du dernier accès (getatime), date de la dernière modification (getmtime), date de création (getctime).
getsize (file)	Retourne la taille d'un fichier.
isabs (path)	Retourne True si le chemin est un chemin absolu.
isfile (path)	Retourne True si le chemin fait référence à un fichier.
isdir (path)	Retourne True si le chemin fait référence à un répertoire.
join (p1, p2, ...)	Construit un nom de chemin étant donné une liste de répertoires.
split (path)	Découpe un chemin, isole le nom du fichier ou le dernier répertoire des autres répertoires.
splitext (path)	Découpe un chemin en nom + extension.

La liste des fichiers

La fonction « **listdir** » du module « **os** » permet de retourner les listes des éléments inclus dans un répertoire (fichiers et sous-répertoires). Toutefois, le module « **glob** » propose une fonction plus intéressante qui permet de retourner la liste des éléments d'un répertoire en appliquant un filtre.

```
>>> def liste_fichier_repertoire (folder, filter):
    file,fold = [], []
    res = glob.glob (folder + "\\*" + filter)
    rep = glob.glob (folder + "\\*")
    for r in rep :
        if r not in res and os.path.isdir (r) :
            res.append (r)
    for r in res :
        path = r
        if os.path.isfile (path) :
            file.append (path)
        else :
            fi,fo = liste_fichier_repertoire (path, filter)
            file.extend (fi)
            fold.extend (fo)
    return file,fold
```

```

>>> import os
>>> os.chdir("F:\PYT - Python programmation objet\Scripts")
>>> folder = os.getcwd()
>>> filter = "*.txt"
>>> import glob
>>> file,fold = liste_fichier_repertoire (folder, filter)
>>> for f in file : print("fichier ", f)

fichier F:\PYT - Python programmation objet\Scripts\FichierB.txt
fichier F:\PYT - Python programmation objet\Scripts\FichierFormate.txt
fichier F:\PYT - Python programmation objet\Scripts\Fichiertexte.txt
fichier F:\PYT - Python programmation objet\Scripts\Monfichier.txt
fichier F:\PYT - Python programmation objet\Scripts\MonfichierC.txt
fichier F:\PYT - Python programmation
objet\Scripts\repCompress\Monfichier.txt
>>> for f in fold : print("répertoire ", f )

```

Liste non exhaustive des fonctionnalités offertes par les modules `shutil` et `os`.

<code>copy(f1,f2)</code>	« shutil » Copie le fichier f1 vers f2
<code>chdir(p)</code>	« os » Change le répertoire courant, cette fonction peut être importante lorsqu'on utilise la fonction <code>system</code> du module « os » pour lancer une instruction en ligne de commande ou lorsqu'on écrit un fichier sans préciser le nom du répertoire.
<code>getcwd()</code>	« os » Retourne le répertoire courant.
<code>makedirs(p)</code>	« os » Crée le répertoire p.
<code>makedirs(p)</code>	« os » Crée le répertoire p et tous les répertoires des niveaux supérieurs s'ils n'existent pas.
<code>remove(f)</code>	« os » Supprime un fichier.
<code>rename(f1,f2)</code>	« os » Renomme un fichier
<code>rmdir(p)</code>	« os » Supprime un répertoire

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 parooooles')
['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 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 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')
['poisssssssssssss']
>>> 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', '6666666')]
>>> re.findall(r'(?i)AAZ*', 'aaZzzRr')
['aaZzz']
>>> re.findall(r'(?:\(03\)) (?:(80) (.*)', '(03)80666666')
['6666666']
>>> match = re.search(r'(03) (80) (?P<numero>.*)', '0380666666')
>>> match.group('numero')
'6666666'
>>> 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
<code>compile(pattern[, flags])</code>	compile le motif <code>pattern</code> et renvoie un objet de type « SRE_Pattern ».
<code>escape(string)</code>	ajoute un antislash « <code>\</code> » devant tous les caractères non alphanumériques contenus dans <code>string</code> . Permet d'utiliser la chaîne dans les expressions régulières.
<code>findall(pattern, string[, flags])</code>	renvoie une liste des éléments interceptés dans la chaîne <code>string</code> par le motif <code>pattern</code> . Lorsque le motif est composé de groupes, chaque élément est un tuple composé de chaque groupe.
<code>finditer(pattern, string[, flags])</code>	équivalente à « findall », mais un itérateur sur les éléments est renvoyé. <code>flags</code> est un entier contenant d'éventuels flags, appliqués au motif complet.
<code>match(pattern, string[, flags])</code>	renvoie un objet de type « MatchObject » si le début de la chaîne <code>string</code> correspond au motif. <code>flags</code> est un entier contenant d'éventuels flags, appliqués au motif complet.
<code>search(pattern, string[, flags])</code>	équivalente à « match » mais recherche le motif dans toute la chaîne.
<code>split(pattern, string[, maxsplit=0])</code>	équivalente au « split » de l'objet <code>string</code> . Renvoie une séquence de chaînes délimitées par le motif <code>pattern</code> . Si <code>maxsplit</code> est fourni, limite le nombre d'éléments à <code>maxsplit</code> , le dernier élément regroupant la fin de la chaîne lorsque <code>maxsplit</code> est atteint.
<code>sub(pattern, repl, string[, count])</code>	remplace les occurrences du motif <code>pattern</code> de <code>string</code> par <code>repl</code> . <code>repl</code> peut être une chaîne ou un objet « callable » qui reçoit un objet « MatchObject » et renvoie une chaîne. Si <code>count</code> est fourni, limite le nombre de remplacements.
<code>subn(pattern, repl, string[, count])</code>	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
```


5

Les structures de données

Les chaînes de caractères

À la différence des données numériques, qui sont des entités singulières, les chaînes de caractères constituent un **type de donnée composite**. Nous entendons par là une entité bien définie qui est faite elle-même d'un ensemble d'entités plus petites, en l'occurrence : les caractères. Suivant les circonstances, nous serons amenés à traiter une telle donnée composite, tantôt comme un seul objet, tantôt comme **une suite ordonnée d'éléments**.

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

```
>>> chaine = 'abcdefghijkl'
>>> for i in chaine : print(i+'-',end=' ')

a- b- c- d- e- f- g- h- i- j- k-
```

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
```

Les chaînes sont des séquences non modifiables

Vous ne pouvez pas modifier le contenu d'une chaîne existante. En d'autres termes, vous ne pouvez pas utiliser l'opérateur « `[]` » dans la partie gauche d'une instruction d'affectation.

```
>>> salut = 'bonjour à tous'
>>> salut[0] = 'B'
Traceback (most recent call last):
  File "<pyshell#367>", line 1, in <module>
    salut[0] = 'B'
TypeError: 'str' object does not support item assignment
>>> salut = salut[0].upper()+salut[1:]
>>> print(salut)
Bonjour à tous
```

Les chaînes sont comparables

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux fonctionnent aussi avec les chaînes de caractères.

```
>>> while True:
    mot = input(\
        "Entrez un mot quelconque : (<enter> pour terminer)")
    if mot == "":
```

```

        break
    if mot < "limonade":
        place = "précède"
    elif mot > "limonade":
        place = "suit"
    else:
        place = "se confond avec"
    print("Le mot", mot, place, \
        "le mot 'limonade' dans l'ordre alphabétique")

```

```

Entrez un mot quelconque : (<enter> pour terminer)limonade
Le mot limonade se confond avec le mot 'limonade' dans l'ordre
alphabétique
Entrez un mot quelconque : (<enter> pour terminer)limonade <--espace
Le mot limonade suit le mot 'limonade' dans l'ordre alphabétique
Entrez un mot quelconque : (<enter> pour terminer)limonadel
Le mot limonadel suit le mot 'limonade' dans l'ordre alphabétique
Entrez un mot quelconque : (<enter> pour terminer)test
Le mot test suit le mot 'limonade' dans l'ordre alphabétique
Entrez un mot quelconque : (<enter> pour terminer)

```

Les séquences d'octets

Les concepteurs de langages de programmation, de compilateurs ou d'interpréteurs pourront décider librement de représenter ces caractères par des entiers sur 8, 16, 24, 32, 64 bits. Nous ne devons donc pas nous préoccuper du format réel des caractères, à l'intérieur d'une chaîne string de Python.

Il en va tout autrement, par contre, pour les entrées/sorties. De même, nous devons pouvoir choisir le format des données que nous exportons vers n'importe quel dispositif périphérique, qu'il s'agisse d'une imprimante, d'un disque dur, d'un écran...

Pour toutes ces entrées ou sorties de chaînes de caractères, nous devons donc toujours considérer qu'il s'agit concrètement de séquences d'octets, et utiliser divers mécanismes pour convertir ces séquences d'octets en chaînes de caractères, et vice-versa.

```

>>> import os
>>> os.chdir("F:\PYT - Python programmation objet\Scripts")
>>> chaine = "Amélie et Eugène\n"
>>> of = open("test.txt", "w")
>>> of.write(chaine)
17
>>> of.close()
>>> of = open("test.txt", "rb")
>>> octets = of.read()
>>> of.close()
>>> type(octets)
<class 'bytes'>
>>> print(octets)
b'Am\xe9lie et Eug\xe8ne\r\n'
>>> print(octets.decode())
Traceback (most recent call last):
  File "<pyshell#466>", line 1, in <module>
    print(octets.decode())

```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position
2: invalid continuation byte
>>> print(octets.decode(errors="replace"))
Amélie et Eugène
>>> import locale
>>> print(locale.getdefaultlocale())
('fr_FR', 'cp1252')
>>> print(octets.decode('cp1252'))
Amélie et Eugène
```

Accéder à d'autres caractères que ceux du clavier

Voyons à présent quel parti vous pouvez tirer du fait que tous les caractères possèdent leur identifiant numérique universel Unicode. Pour accéder à ces identifiants, Python met à votre disposition un certain nombre de fonctions prédéfinies.

La fonction « **ord(ch)** » accepte n'importe quel caractère comme argument. En retour, elle fournit la valeur de l'identifiant numérique correspondant à ce caractère.

La fonction « **chr(num)** » fait exactement le contraire, en vous présentant le caractère typographique dont l'identifiant Unicode est égal à num. Pour que cela fonctionne, il faut cependant que deux conditions soient réalisées :

- la valeur de num doit correspondre effectivement à un caractère existant (la répartition des identifiants unicode n'est pas continue : certains codes ne correspondent donc à aucun caractère)
- votre ordinateur doit disposer d'une description graphique du caractère, ou, en d'autres termes, connaître le dessin de ce caractère, que l'on appelle un glyphe. Les systèmes d'exploitation récents disposent cependant de bibliothèques de glyphes très étendues, ce qui devrait vous permettre d'en afficher des milliers à l'écran.

Vous pouvez exploiter ces fonctions prédéfinies pour vous amuser à explorer le jeu de caractères disponible sur votre ordinateur. Vous pouvez par exemple retrouver les caractères minuscules de l'alphabet grec, en sachant que les codes qui leur sont attribués vont de 945 à 969. Ainsi le petit script ci-dessous :

```
>>> s = ""      # chaîne vide
>>> i = 945     # premier code
>>> while i <= 969: # dernier code
        s += chr(i)
        i = i + 1

>>> print("Alphabet grec (minuscule) : ", s)
Alphabet grec (minuscule) : αβγδεζηθικλμνξοπρςστυφχψω
```

Atelier 5 - Exercice 1

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 crée 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>i</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, i + k, i + 2k, i + 3k, ...</i>
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 pourrions 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
<code>clear()</code>	Supprime tous les éléments du dictionnaire.
<code>copy()</code>	Renvoie une copie par références du dictionnaire. Lire la remarque sur les copies un peu plus bas.
<code>items()</code>	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.
<code>keys()</code>	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.
<code>values()</code>	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.
<code>get(cle, default)</code>	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 ».
<code>pop(cle, default)</code>	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.
<code>popitem()</code>	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.
<code>update(dic, **dic)</code>	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.
<code>setdefault(cle, default)</code>	Fonctionne comme « get » mais si clé n'existe pas et default est fourni, le couple (cle, default) est ajouté à la liste.
<code>fromkeys(seq, default)</code>	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'
```

Atelier 5 - Exercice 4

6

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 <code>numpy.int64</code> ou <code>numpy.float64</code> 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 librairie « **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
>>> tulld = np.ones(2)
>>> tulld
array([ 1.,  1.])
>>> tulld[:,newaxis]
array([[ 1.],
       [ 1.]])
>>> np.column_stack((tulld[:,newaxis],tb2d))
array([[ 1.,  2.,  2.],
       [ 1.,  2.,  2.]])
>>> np.hstack((tulld[:,newaxis],tb2d))
array([[ 1.,  2.,  2.],
       [ 1.,  2.,  2.]])
```

Opérations arithmétiques élément par élément

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

```

    [ 4.,  4.])
>>> tc = np.hstack((tb2d,tuld[:,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.

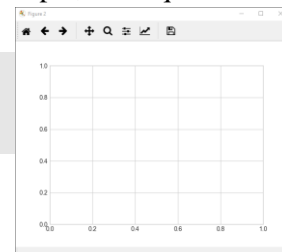
Pour commencer, mettons en place l'environnement de travail.

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

Réaliser des graphiques simples

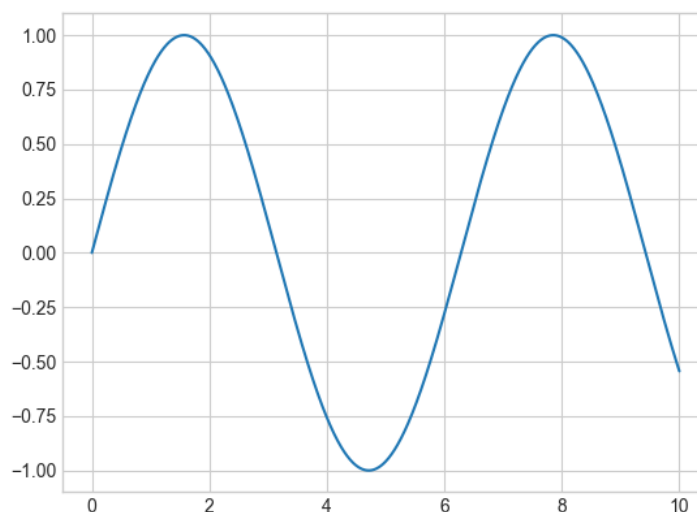
Commençons par étudier un cas simple, comme tracer la courbe d'une fonction. Nous avons vu un exemple de cette utilisation dans le chapitre 3 de la première partie de ce cours. Ici, nous allons le faire d'une manière moins simple, mais qui nous donne plus de possibilités.

```
>>> 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()
```

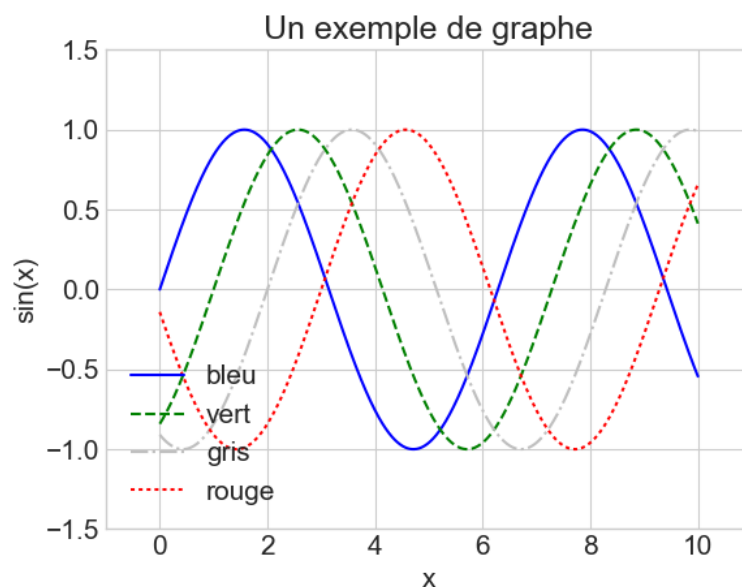


On aurait pu simplement taper `plt.plot(x, np.sin(x))`. Maintenant, voyons un exemple un peu plus poussé.

```

>>> # 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()

```



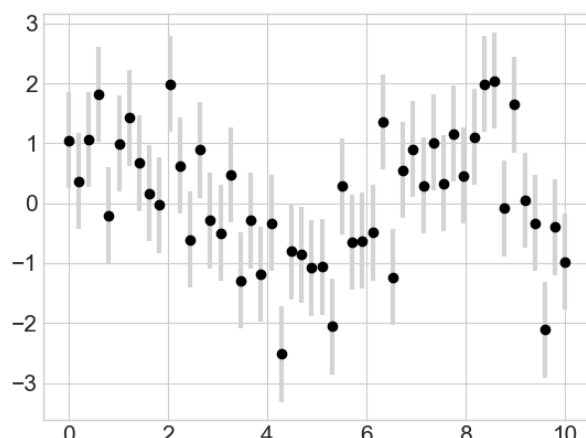
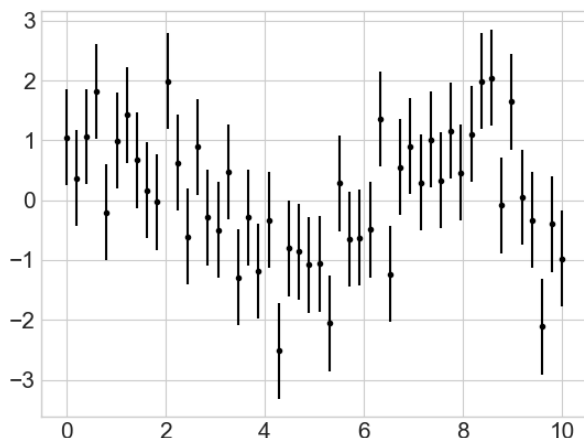
Visualiser l'incertitude

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.

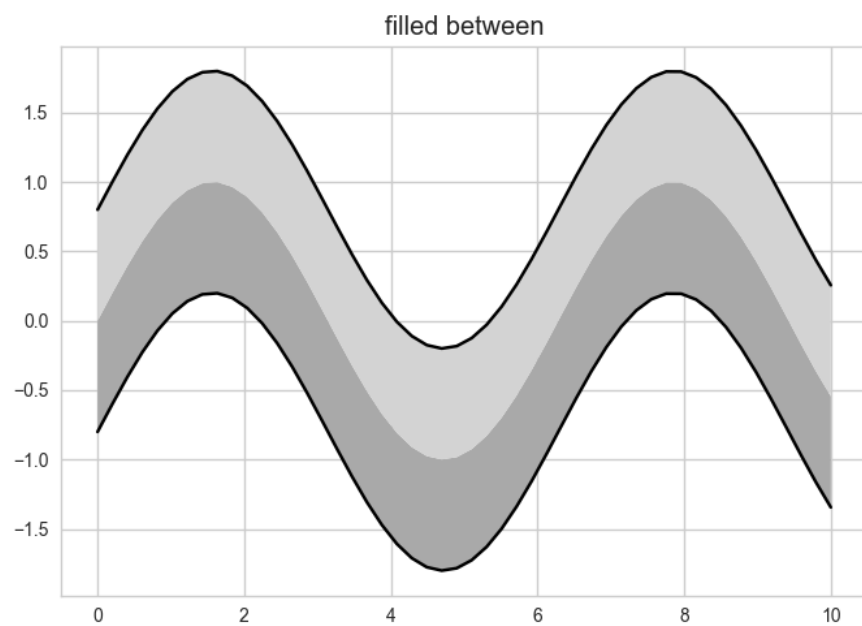
```
>>> plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
...             ecolor='lightgray', elinewidth=3, capsize=0);
<Container object of 3 artists>
>>> plt.show()
```

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)
```

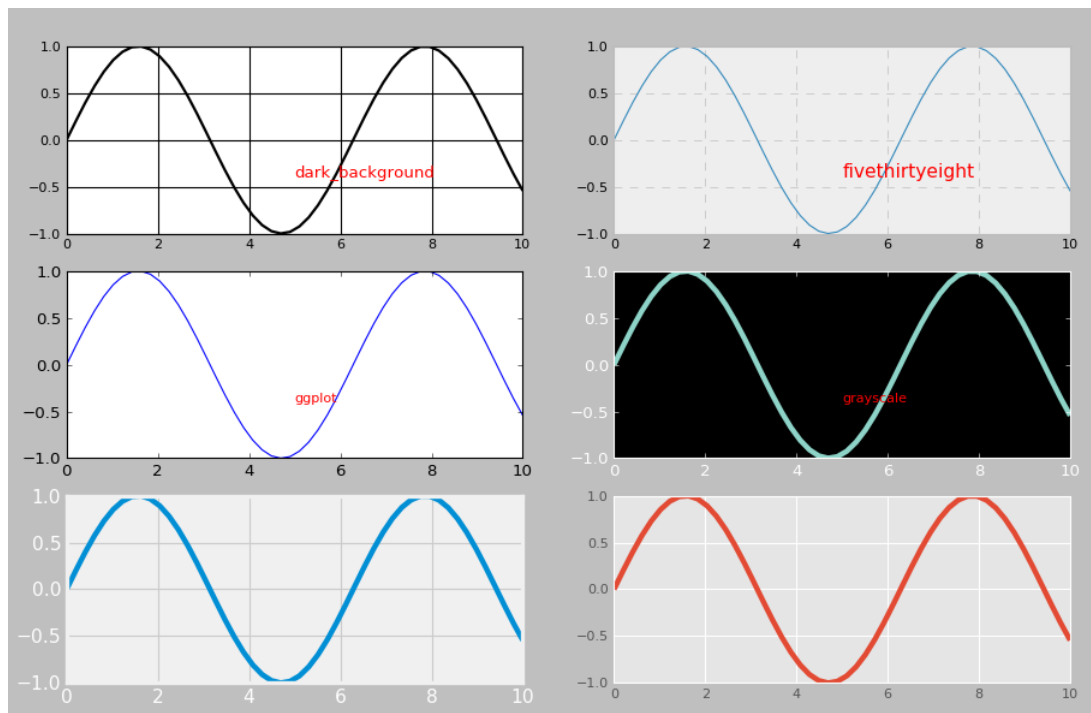
```
>>> y1 = y - dy
>>> y2 = y + dy
>>> plt.plot(x, y1, x, y2, color='black')
[<matplotlib.lines.Line2D object at 0x0000022887D49278>,
 <matplotlib.lines.Line2D object at 0x0000022887D49BE0>]
>>> plt.fill_between(x, y, y1, where=y>=y1,
... facecolor='darkgray',interpolate=True)
<matplotlib.collections.PolyCollection 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.

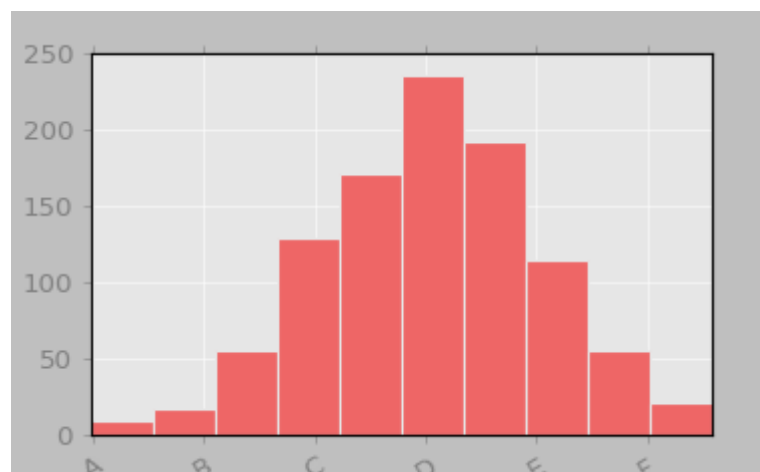
Pour des raisons historiques, les sous-graphes sont numérotés à partir de 1, au lieu de 0. Le graphe tout en haut à gauche est donc le graphe numéro 1.

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()

```

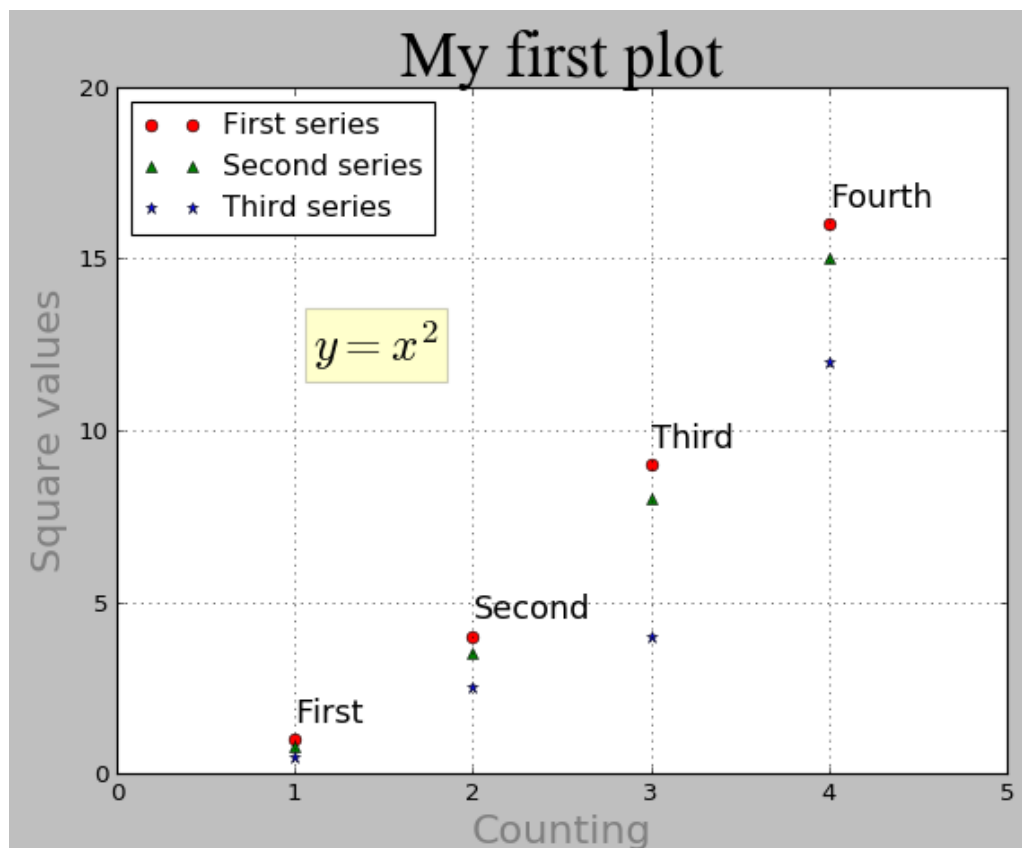


```

>>> 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)

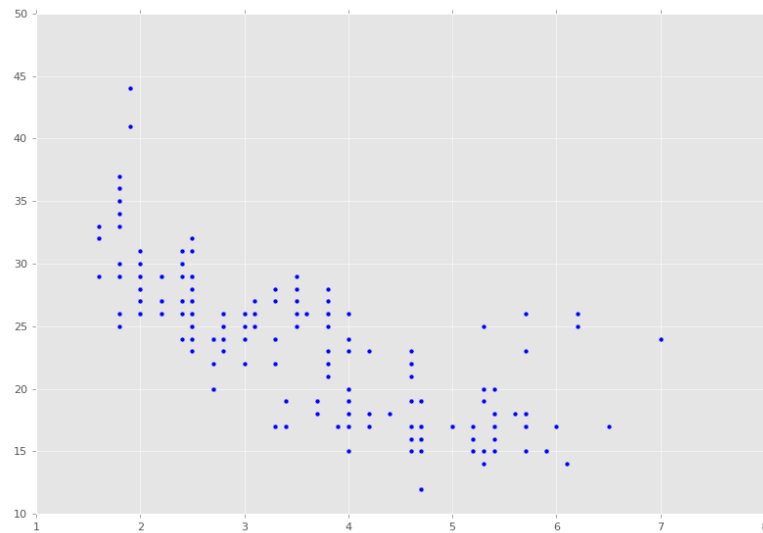
```

```
>>> 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()
```

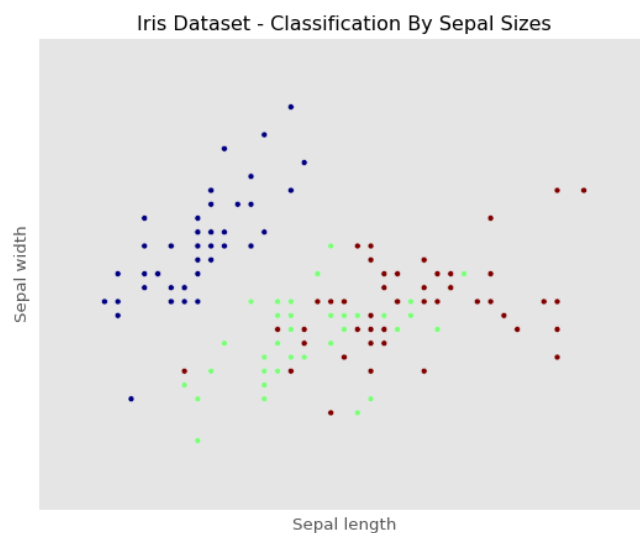


```
>>> df = pd.read_csv('donnees/mpg.csv')
>>> df.head()
  manufacturer model  displ  year  cyl    trans  drv  cty  hwy  fl  class
1         audi   a4    1.8  1999    4  auto(l5)   f   18   29   p  compact
2         audi   a4    1.8  1999    4  manual(m5)  f   21   29   p  compact
3         audi   a4    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

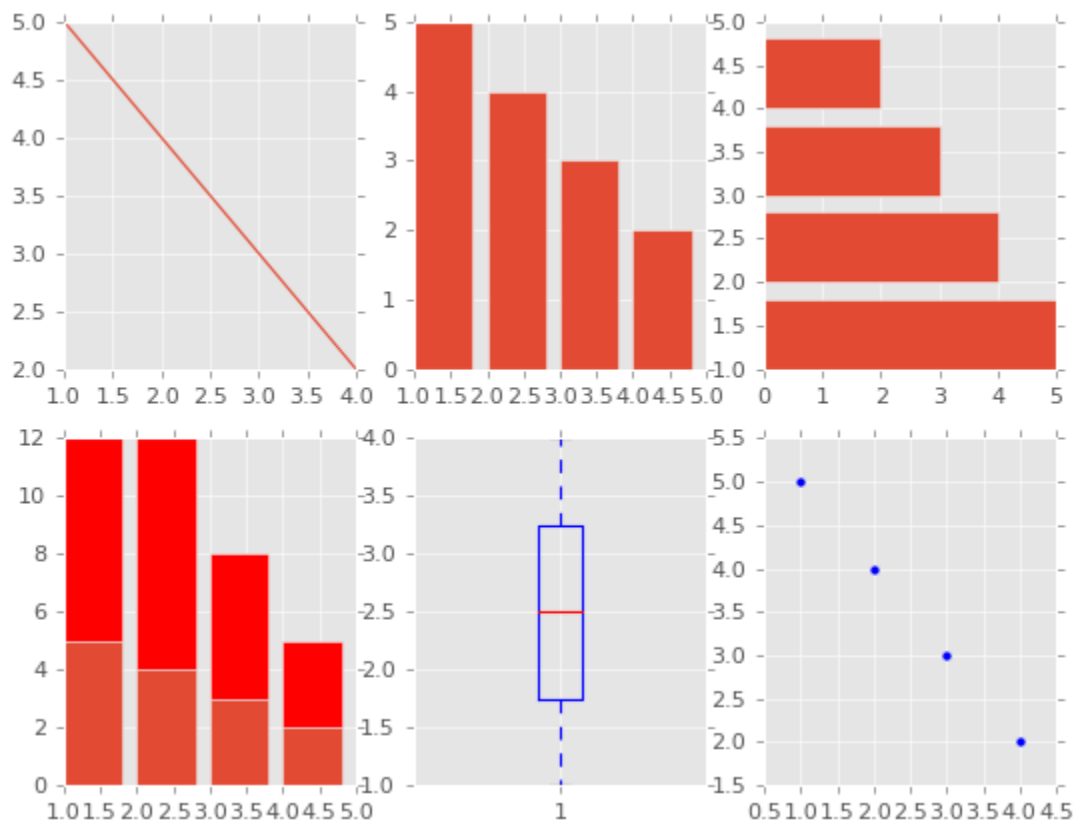
>>> plt.style.use('ggplot')
>>> plt.figure(figsize=(12, 8))
<matplotlib.figure.Figure object at 0x0000022889C647F0>
>>> plt.scatter(df.displ,df.hwy)
<matplotlib.collections.PathCollection object at 0x0000022887D68D30>
>>> plt.show()
```

```
>>> 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()
```

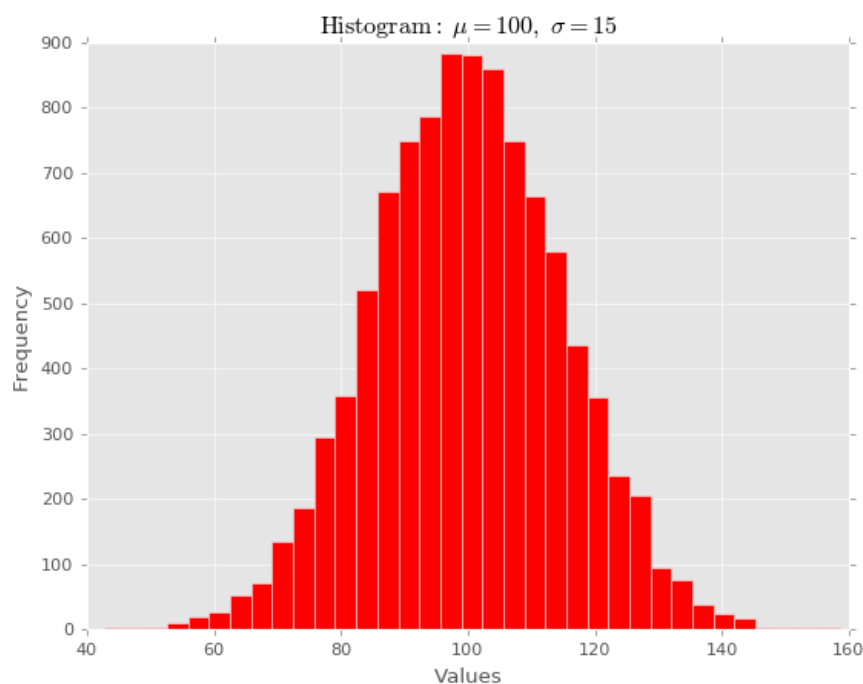


```
>>> mu = 100
>>> sigma = 15
>>> x = np.random.normal(mu, sigma, 10000)
>>> ax = plt.gca()
>>> ax.hist(x, bins=35, color='r')
(array([ 2.,  2.,  3., 10., 19., 27., 53., 71., 134.,
        187., 295., 358., 520., 672., 748., 786., 884., 881.,
        860., 749., 664., 580., 437., 357., 237., 205., 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()

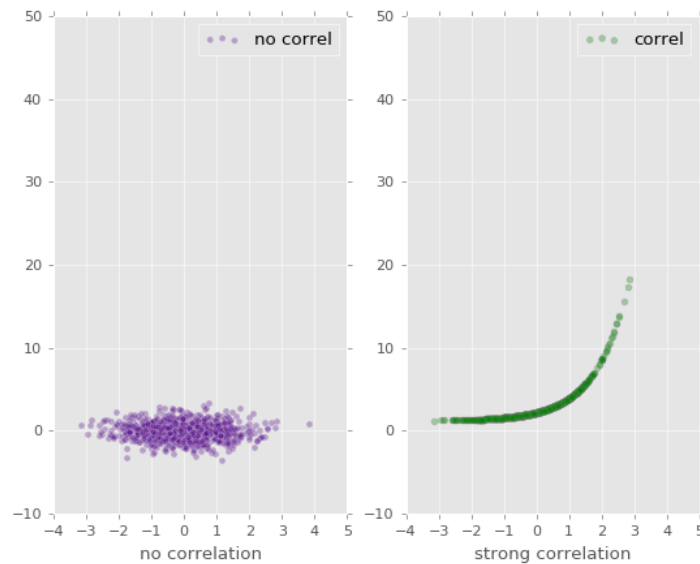
```



```

>>> 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()
```

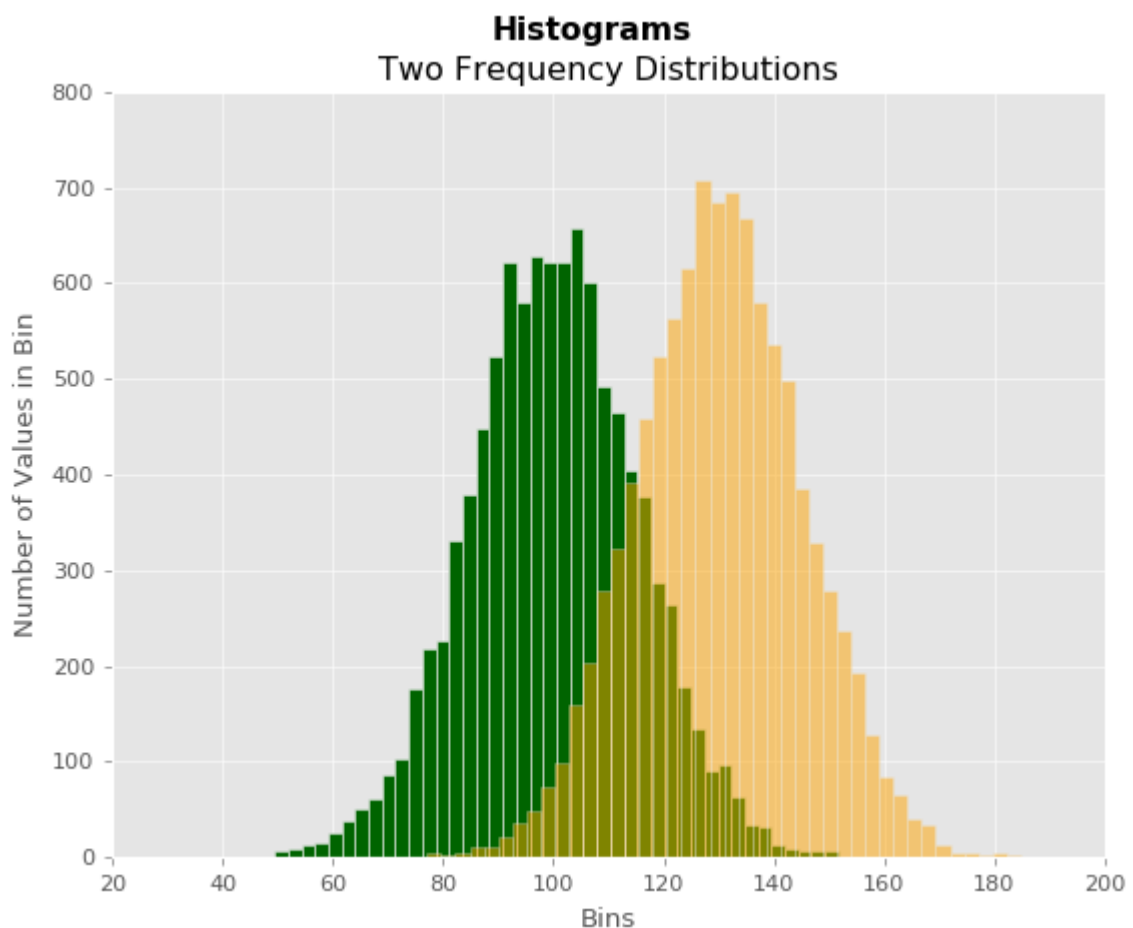


```
>>> plt.style.use('ggplot')
```

```

>>> 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()

```



```

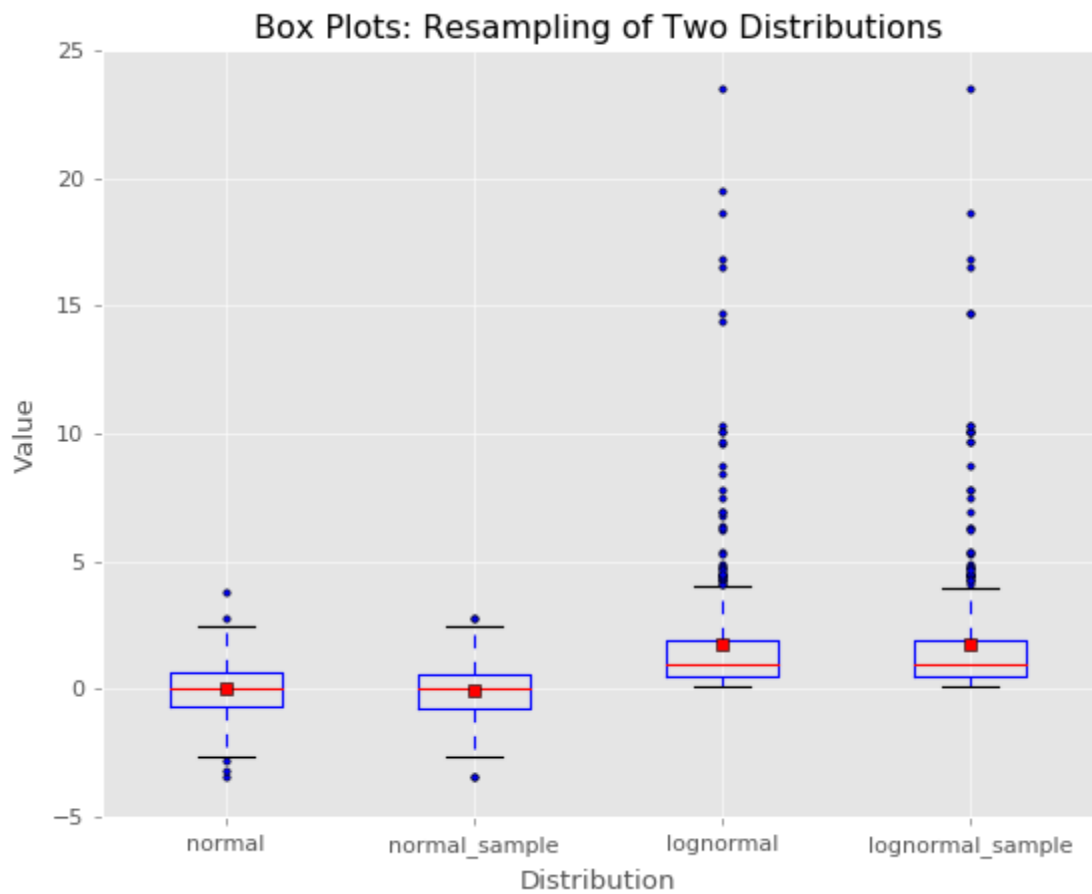
>>> N = 500
>>> normal = np.random.normal(loc=0.0, scale=1.0, size=N)
>>> lognormal = np.random.lognormal(mean=0.0, sigma=1.0, size=N)
>>> index_value = np.random.random_integers(low=0, high=N-1, size=N)
>>> normal_sample = normal[index_value]
>>> lognormal_sample = lognormal[index_value]
>>> box_plot_data =
[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()

```



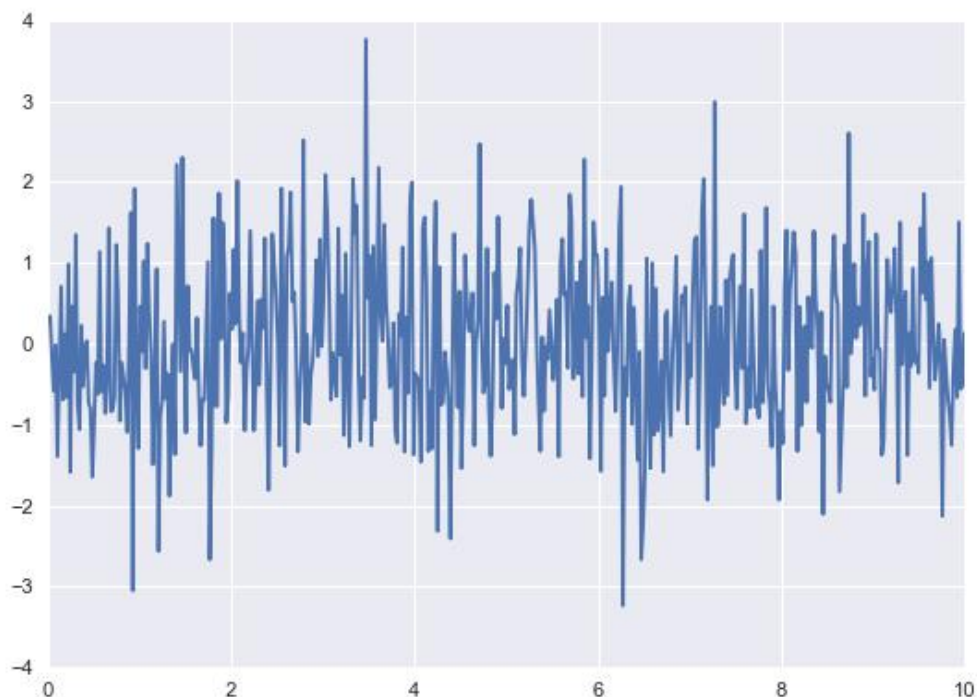
La librairie Seaborn

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**.

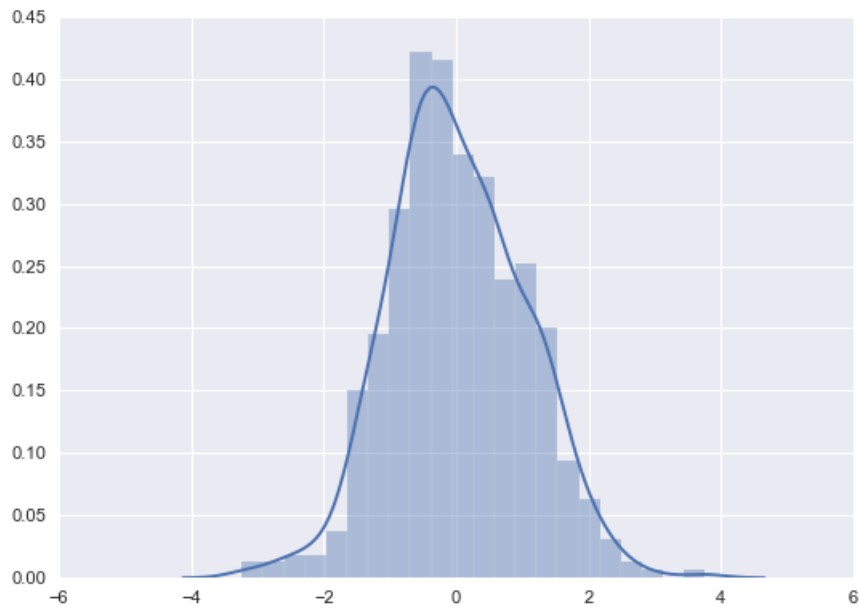
Seaborn fournit une interface qui permet de palier ces problèmes. Il utilise toujours **Matplotlib**, mais le fait en exposant des fonctions plus intuitives. Pour commencer à l'utiliser, rien de plus simple.

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

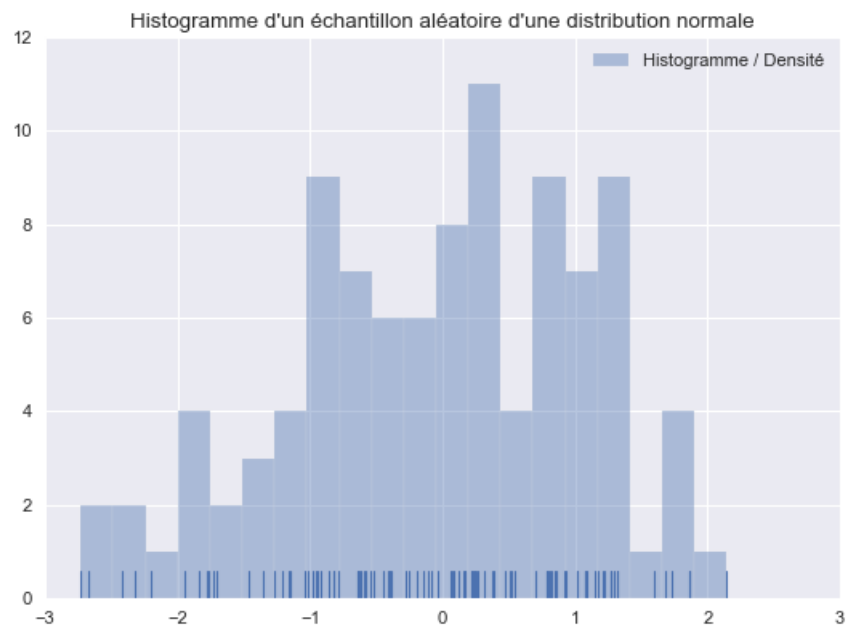


Seaborn nous fournit aussi des fonctions pour des graphiques utiles pour l'analyse statistique. Par exemple, la fonction **distplot** permet non seulement de visualiser l'histogramme d'un échantillon, mais aussi d'estimer la distribution dont l'échantillon est issu.

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

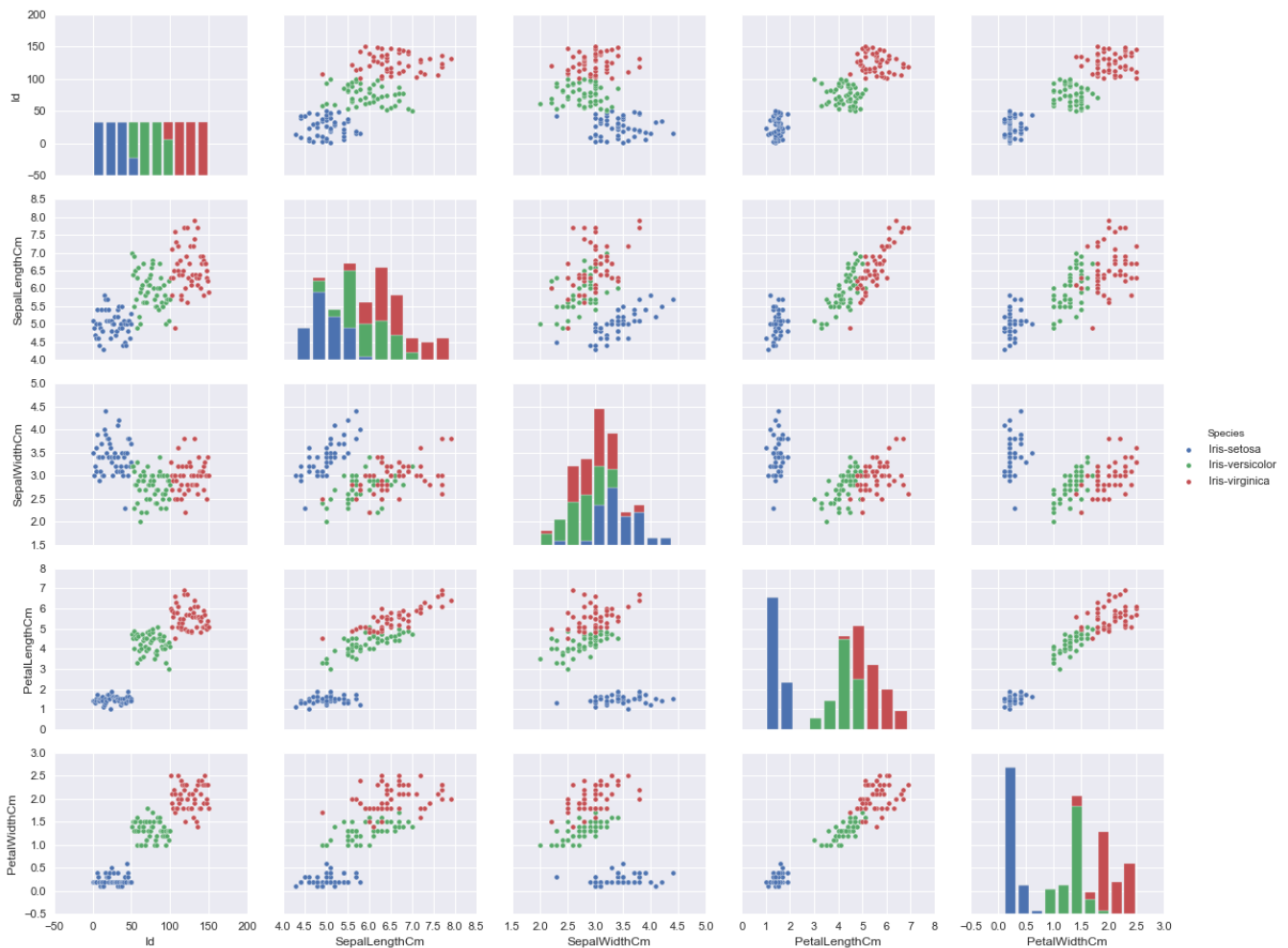


```
>>> sns.set(color_codes=True)
>>> x = np.random.normal(size=100)
>>> sns.distplot(x, bins=20, kde=False, rug=True, label="Histogramme
/ Densité")
>>> plt.title("Histogramme d'un échantillon aléatoire d'une
distribution normale")
>>> plt.legend()
>>> plt.show()
```



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()
```

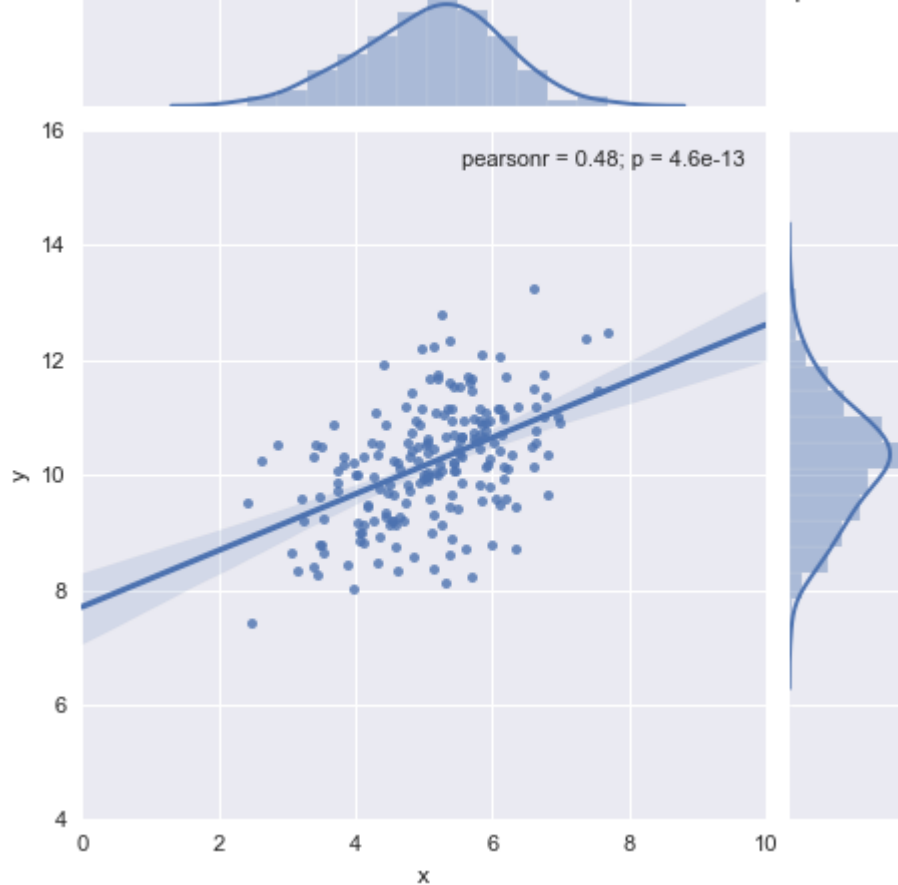



La diagonale est traitée différemment, car tracer une variable en fonction d'elle-même n'aurait aucun intérêt. À la place, **sns.pairplot** trace un histogramme des données en fonction de la variable en question pour chaque classe de données.

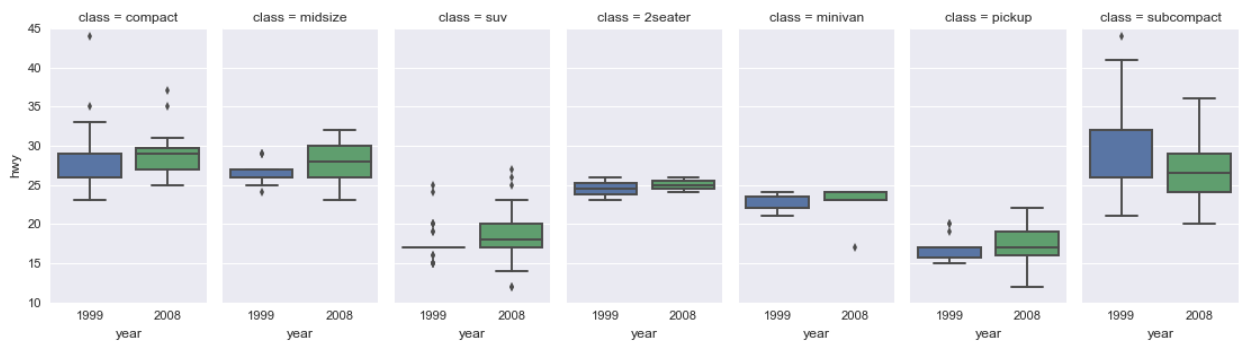
Nous pouvons aussi voir la distribution jointe de deux caractéristiques :

```
>>> 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()
```

Joint Plot of Two Variables with Bivariate and Univariate Graphs



```
>>> 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()
```



7

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** fournit un ensemble de fonctions de lecture/écriture de haut niveau pour accéder aux fichiers. Les fonctions de lecture renvoient généralement un objet DataFrame.

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

```
>>> 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-Carpique	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-Carpique	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-Carpique	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-Carpique',
       'Rouen-Boos', 'Reims-Prunay', 'Brest-Guipavas', 'Ploumanac'h',
       'Rennes-St Jacques', 'Alencon', 'Orly', 'Troyes-Barbère',
       '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 Giron', '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 à 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
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
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
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
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
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
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 dupliques
>>> 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** ».

```
>>> employees = pd.read_csv("donnees/stagiaire/employees.csv",
...                          sep=';', header=0, na_values=["null"],
...                          usecols=['No', 'Manager', 'Nom', 'Prenom'],
...                          names=['No', 'Manager', 'Nom', 'Prenom'])
>>> managers = employees.copy()
>>> employees.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(employees, 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()

```

Fonction	Pays	Salaire	Commission
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