# [Découvrez les librairies Python pour la Data Science](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science)

Table des matières

[Découvrez les librairies Python pour la Data Science 1](#_Toc57638909)

[Familiarisez-vous avec l'écosystème Python 5](#_Toc57638910)

[Choisir le bon outil 5](#_Toc57638911)

[La simplicité VS la puissance 5](#_Toc57638912)

[Python 7](#_Toc57638913)

[Le matériel 9](#_Toc57638914)

[Pourquoi travailler avec un notebook ? 10](#_Toc57638915)

[Installez Jupyter sur votre propre ordinateur 12](#_Toc57638916)

[Ce que nous allons faire dans ce chapitre 12](#_Toc57638917)

[Installer la totale : Python + librairies + notebook (distribution Anaconda) 12](#_Toc57638918)

[Installer uniquement le notebook Jupyter (sans Anaconda) 14](#_Toc57638919)

[Installer uniquement les librairies Python (sans Anaconda) 15](#_Toc57638920)

[Lancez une session de notebook Jupyter sur AWS 16](#_Toc57638921)

[Configurez l'accès web à votre serveur 16](#_Toc57638922)

[Installez Anaconda sur votre serveur 19](#_Toc57638923)

[Configurez Jupyter pour les accès distants 20](#_Toc57638924)

[Lancez un notebook Jupyter 22](#_Toc57638925)

[Faîtes vos premiers pas dans un notebook Jupyter 24](#_Toc57638926)

[Créer un nouveau notebook 24](#_Toc57638927)

[Lancer un notebook déjà créé 25](#_Toc57638928)

[Enregistrez les fonctions très souvent utilisées dans un module 26](#_Toc57638929)

[Découvrez le problème de Monty Hall 28](#_Toc57638930)

[Le problème de Monty Hall 28](#_Toc57638931)

[Simulation 28](#_Toc57638932)

[Analyse des résultats 34](#_Toc57638933)

[Visualisation 34](#_Toc57638934)

[Explication théorique 38](#_Toc57638935)

[Petit exercice 39](#_Toc57638936)

[Utilisez Numpy pour illustrer le théorème central limite 40](#_Toc57638937)

[La distribution Gaussienne 40](#_Toc57638938)

[Le théorème central limite 41](#_Toc57638939)

[Générez des réalisations de variables aléatoires avec Numpy 42](#_Toc57638940)

[Les opérations de base sur les matrices Numpy 43](#_Toc57638941)

[Visualisation 44](#_Toc57638942)

[Les histogrammes 45](#_Toc57638943)

[Calculer les propriétés d'un échantillon 47](#_Toc57638944)

[Entraînez-vous en simulant le problème de Monty Hall avec Numpy 48](#_Toc57638945)

[À vous de jouer ! 48](#_Toc57638946)

[Vérifiez votre travail 48](#_Toc57638947)

[Plongez en détail dans la librairie NumPy 49](#_Toc57638948)

[Créez des tableaux Numpy 49](#_Toc57638949)

[Les propriétés des tableaux 51](#_Toc57638950)

[Indexation et Slicing 51](#_Toc57638951)

[Opérations sur les tableaux Numpy 54](#_Toc57638952)

[Broadcasting 57](#_Toc57638953)

[Pour aller plus loin 58](#_Toc57638954)

[Maîtrisez les possibilités offertes par Matplotlib 60](#_Toc57638955)

[Réaliser des graphiques simples 60](#_Toc57638956)

[Visualiser l'incertitude 63](#_Toc57638957)

[Personnalisation et sous-graphes 65](#_Toc57638958)

[Pour aller plus loin 68](#_Toc57638959)

[Réalisez de beaux graphiques avec Seaborn 69](#_Toc57638960)

[Entraînez-vous en effectuant une régression linéaire 74](#_Toc57638961)

[À vous de jouer ! 74](#_Toc57638962)

[Vérifiez votre travail 74](#_Toc57638963)

[Passez de Numpy à Pandas 75](#_Toc57638964)

[Réfléchissons un peu 75](#_Toc57638965)

[La librairie Pandas 77](#_Toc57638966)

[Manipulez les données contenues dans vos DataFrames 84](#_Toc57638967)

[Aperçu rapide 84](#_Toc57638968)

[Données manquantes 85](#_Toc57638969)

[Renommer une colonne 87](#_Toc57638970)

[Supprimer des axes 88](#_Toc57638971)

[Tableaux croisés dynamiques 88](#_Toc57638972)

[Effectuez les opérations d'algèbre relationnelle sur les DataFrames 92](#_Toc57638973)

[L'algèbre relationnelle 92](#_Toc57638974)

[Les structures de données de Pandas 92](#_Toc57638975)

[La projection et la restriction 95](#_Toc57638976)

[L'union 96](#_Toc57638977)

[La jointure 98](#_Toc57638978)

[Le produit cartésien 103](#_Toc57638979)

[L'agrégation 104](#_Toc57638980)

[Entraînez-vous en générant des cartes de vœux avec Pandas 107](#_Toc57638981)

[À vous de jouer ! 107](#_Toc57638982)

[Vérifiez votre travail 107](#_Toc57638983)

Découvrez les **bonnes pratiques** et les **connaissances fondamentales** qui vous aideront à effectuer vos analyses de données à l'aide de **Python**.

Vous verrez comment utiliser les **notebooks Jupyter** et des **librairies Python** comme **Pandas**, **Matplotlib** ou encore **Numpy** explorer vos données et les analyser.

Python possède de nombreuses librairies, utilisées dans tous les domaines. Mais pour pouvoir traiter une grande quantité de données, **il est essentiel d'observer quelques règles de base**, que vous allez découvrir dans ce cours.

**Objectifs pédagogiques**

* Modifier et exécuter un notebook Jupyter
* Créer et installer des modules Python
* Visualiser des données avec Matplotlib et Seaborn
* Manipuler des tableaux avec Numpy
* Manipuler un jeu de données grâce à Pandas

**Prérequis**

Pour suivre ce cours, vous devez être familier avec la programmation Python. Nous allons aussi invoquer des notions basiques de probabilités et de statistiques, ainsi que de l'algèbre relationnelle. Pour en profiter pleinement, n'hésitez pas à vous rafraîchir la mémoire, avant ou pendant le cours, sur :

* [**La programmation en Python**](https://openclassrooms.com/courses/demarrez-votre-projet-avec-python) (ces connaissances sont nécessaires)
* [**L'algèbre relationnelle**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql) : seules les 2 premières parties de ce cours sont nécessaires, c'est-à-dire les 12 premiers chapitres. Les 2 parties qui suivent (chapitres 13 à 24) ne sont pas utiles. *Si vous connaissez le SQL, alors vous avez déjà manipulé l'algèbre relationnelle sans le savoir ! ;)*
* Optionnellement, quelques notions de [**Programmation Orientée Objet en Python**](https://openclassrooms.com/courses/decouvrez-la-programmation-orientee-objet-avec-python) seront un plus pour bien comprendre certaines des lignes de code dans ce cours, *mais vous ne serez absolument pas bloqués si vous ne maîtrisez pas ces notions*.
* [**La ligne de commande en Linux**](https://openclassrooms.com/courses/reprenez-le-controle-a-l-aide-de-linux). Ces notions sont ici facultatives si vous installez le notebook sur votre propre ordinateur. Mais si vous souhaitez aller plus loin en lançant le notebook sur un serveur distant (un autre ordinateur, bien plus puissant que le vôtre, situé quelque part dans le monde !), alors sachez notamment comment créer une instance AWS et vous y connecter en SSH.

## Familiarisez-vous avec l'écosystème Python

### Choisir le bon outil

Faire de la data science nécessite la maîtrise de deux disciplines. D'un côté, il convient de maîtriser les bases mathématiques et les algorithmes qui permettent de modéliser des données, classifier, et comprendre. On pourrait qualifier cela d'aspect théorique.

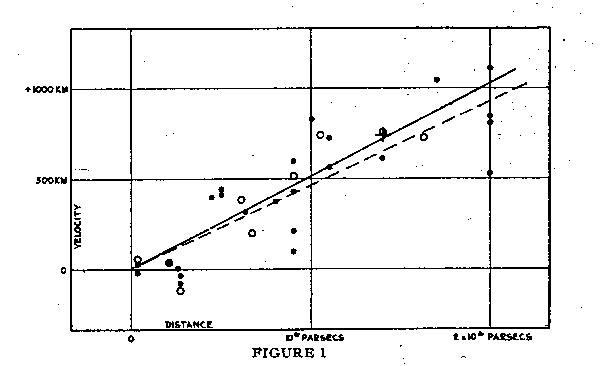
D'un autre côté, vous aurez aussi besoin d'un savoir-faire technique, pour vous permettre d'implémenter votre vision sur des données réelles. Et comme pour toute tâche, il est essentiel de posséder les bons outils et de les maîtriser. Le choix de l'outil vous appartient, bien sûr. Le bon outil est en premier lieu celui avec lequel vous vous sentez le plus à l'aise. Néanmoins, il existe certains critères pour ce choix, et ce chapitre vise à vous sensibiliser à ces critères et vous permettre de choisir vos outils de départ.

### La simplicité VS la puissance

Comme souvent, le choix de l'outil correspond à un compromis entre la simplicité et la puissance de l'outil. Ainsi, vous devrez souvent réfléchir à l'opportunité d'apprendre à vous servir d'un nouvel outil par rapport aux bénéfices qu'il apporte.

#### Une calculatrice

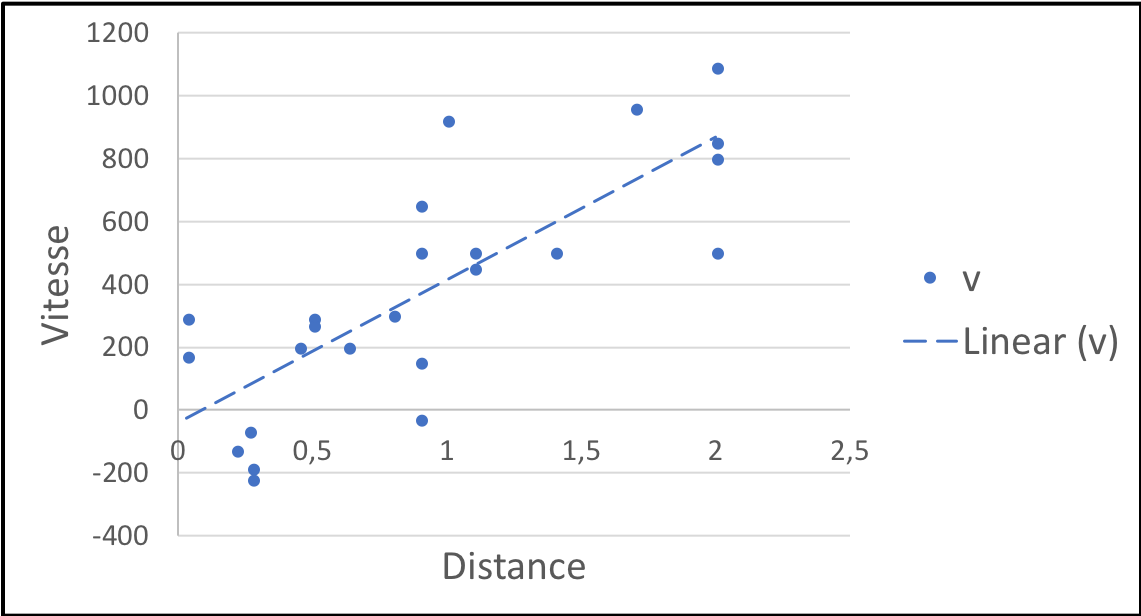
Vous maîtrisez certainement l'utilisation d'une calculatrice, et techniquement, vous pouvez tout à fait effectuer une régression linéaire armé d'une simple calculatrice. Après tout, Edwin Hubble, n'a pas eu besoin de machines hyper puissantes pour découvrir l'expansion de l'univers.

Vitesse apparente des galaxies en fonction de leur distance

Mais les temps ont changé, et nous ne voulons plus ressembler à ces pionniers passant des jours sur leur calculatrices primitives pour découvrir les secrets de l'univers.

#### Un tableur

L'étape suivante serait par exemple l'utilisation d'un logiciel tableur. Il serait très simple à utiliser (vous êtes certainement déjà familier avec au moins un logiciel de ce genre), et assez puissant pour tracer la même figure en quelques secondes.

La même relation

Les deux figures ne sont pas identiques. Si vous voulez savoir pourquoi, je vous conseille la lecture de cette [**page**](https://www.r-bloggers.com/linear-modeling-in-r-and-the-hubble-bubble/).

C'est pas mal. Vous pouvez même encore aujourd'hui utiliser ce genre de logiciel pour beaucoup de tâches. Mais si vous voulez vous adonner à la data science, vous rencontrerez très vite les limites de ces outils :

1. Ils ne peuvent pas faire face à l'énorme quantité de données que nous manipulons régulièrement. Par exemple, le tableur de LibreOffice ne peut gérer que 1024 colonnes et un peu plus d'un million de lignes.
2. Leur puissance de calcul est très insuffisante.
3. Ils ne permettent pas de faire tourner les algorithmes à la pointe.

#### Logiciels spécialisés

Face à ces limites, les statisticiens ont pendant longtemps eu recours aux logiciels spécialisés. Longtemps réservé aux spécialistes à cause de leurs coûts, on peut notamment citer SAS, surtout utilisé par les statisticiens, et Matlab, plutôt pour les ingénieurs et les chercheurs.

Aujourd'hui, ce sont plutôt les solutions libres et open source qui sont les références dans la boîte à outils du data scientist, pour plusieurs raisons :

1. Ils sont gratuits
2. Leur nature libre permet à chacun d'implémenter de nouveaux algorithmes et/ou améliorer les algorithmes existants.
3. Leur disponibilité fait que la plupart des chercheurs mettent à disposition leurs travaux dans ces solutions.

R est un descendant de S, lui-même un logiciel spécialisé pour les statisticiens. R est très populaire, et est très puissant.

Mon choix personnel s'est porté sur Python, qui techniquement est un langage de programmation, mais qui est devenu grâce à ses nombreux outils une, sinon la, référence en matière de traitement de données. Ce n'était pas chose évidente, d'ailleurs. Rien ne prédestinait à la base Python à devenir autant utilisé par les scientifiques. C'est seulement parce que le langage a attiré l'attention de quelques scientifiques qu'ils ont décidé de s'associer à l'inventeur du langage pour créer un module de calcul matriciel. Numpy était né, et a entraîné la création de toute la suite scientifique Python que nous connaissons aujourd'hui.

### Python

Python possède plusieurs propriétés qui en font un outil de choix pour l'analyse de données.

#### Simplicité

Python est un langage de programmation simple. Par exemple, comparez la complexité d'un programme simple, qui imprime "Hello world", en langage C et en Python.

#include <stdio.h>

*int* main(*void*)

{

printf("hello, world\n");

}

print("Hello World")

#### Interactivité

Python est un langage interprété. Cela signifie en pratique que vous pouvez lancer un script en Python sans étape de compilation. Cette propriété, bien qu'elle nuise en principe aux performances (nous y reviendrons) permet une grande interactivité. Vous pouvez écrire et exécuter votre programme ligne par ligne, en examinant à chaque fois le résultat. Cette propriété est exploitée a merveille par les notebooks Jupyter (que nous utiliserons dans la suite de ce cours).

#### L'écosystème

Peut-être le point le plus important, est l'ensemble des outils, des librairies, la communauté, bref, tout ce qui gravite autour du langage lui-même. Python possède des librairies pour à peu près tout ce que vous pouvez imaginer, et notamment :

* numpy et scipy pour les calculs
* Matplotlib et Seaborn pour la visualisation
* Scikit-learn pour les algorithmes
* Pandas pour les gérer les données (les charger, appliquer des opérations d'algèbre relationnelle, etc.)
* Tensorflow et PyTorch pour le deep learning
* etc.

Python est supporté sur à peu près toutes les plateformes, sans coût supplémentaire. Ses librairies lui permettent de tirer avantage de chaque plateforme (instructions CPU parallèles, GPU), le rendant très efficace malgré sa nature de langage interprété.

### Le matériel

Nous avions été habitués depuis quelques années à ne plus nous soucier de la performance de nos machines. Même les ordinateurs portables les plus légers semblaient être assez puissants pour répondre à tous nos besoins, hormis peut-être pour les jeux vidéos.

L'essor de la data science a bouleversé cet équilibre, notamment parce que nous avons énormément de données à traiter. De plus, il est souvent essentiel de pouvoir interagir avec les données avec un minimum de délai. Devoir attendre une semaine pour avoir les résultats d'une simulation nuit vraiment à notre capacité à mener une analyse complexe à son terme.

Une attention particulière est portée, dans la conception de nos outils, à la question de **la performance**. Il s'agit de s'assurer d'utiliser toutes les technologies à portée de main (instructions CPU spécifiques, GPU, Puces spécialisée) pour accélérer autant que possible les calculs, et donc permettre d'analyser des quantités plus importantes de données. Il convient par conséquent de veiller à utiliser ces outils autant que possible, plutôt que de les redévelopper vous-même. Dans la suite de ce cours, nous verrons un exemple de cela avec la librairie Numpy.

Malgré tout, nous nous confrontons assez vite aux limites de notre matériel, et plusieurs choix s'offrent à nous. Pour simplifier, imaginons que vous venez de réaliser que votre ordinateur portable favori met des jours à entraîner un modèle sur vos données. Vous avez alors le choix entre plusieurs solutions, que nous allons voir plus en détails ici.

#### Acquérir un ordinateur portable plus puissant

C'est une solution à envisager. Vous êtes familier avec cet environnement, il sera toujours disponible, et vous n'aurez qu'un ordinateur à gérer. Malheureusement, ce choix a des inconvénients. Un ordinateur portable est soumis à des contraintes de puissance et d'énergie. Il est très rare de trouver un ordinateur portable avec des performances comparables à un ordinateur fixe. Et ceux qui le sont sont souvent volumineux et lourd, perdant les avantages de la portabilité. Cela dit, cette solution peut vous convenir si la quantité de données que vous avez à traiter est limitée (c'est à dire que les données tiennent dans la mémoire vive de votre ordinateur, et que vos algorithmes peuvent tourner assez rapidement sans un GPU puissant).

#### Acquérir un PC fixe

Vous pouvez décider de vous procurer un PC avec la dernière génération de composants. Le coût est certes considérable, mais souvent pas excessif au regard des enjeux métiers. Le désavantage de cette solution est bien sûr les contraintes physiques. Mais bon nombre d'outils, notamment tout l'éco système Python, peuvent être utilisés à distance. En clair, vous pouvez utiliser votre navigateur pour faire tourner les traitements depuis votre ordinateur portable (ou même, si vous êtes un peu aventurier, votre téléphone) sur un ordinateur puissant.

Cette solution nécessite un certain savoir-faire en administration système (soit de votre part soit de la part de quelqu'un dans votre équipe) mais elle est souvent adaptée aux vraies besoins des entreprises, et peut se révéler moins onéreux que la solution suivante. De plus, elle répond aux besoins de confidentialité, au cœur des préoccupations des entreprises avec l'arrivée de la RGPD.

#### Les services en ligne

Parfois, les données que vous aurez à gérer seront tellement importantes, et les algorithmes tellement complexes, que vous aurez besoin d'une énorme puissance de calcul. Ou peut-être avez-vous un besoin ponctuel en puissance qui ne justifie pas l'achat et la maintenance d'un serveur. Dans ce cas, vous pouvez faire appel à des outils en ligne. Beaucoup d'entreprise (Amazon AWS, Google Cloud, Microsoft Azure...) proposent désormais de tels services. Ils sont faciles à utiliser et vous permettent d'accéder à une puissance de calcul quasi-infinie, limitée principalement par votre budget. De plus, utiliser ces solutions vous dispense de devoir acheter, configurer, héberger et maintenir votre propre machine.

#### Et pour ce cours ?

Dans la suite de ce cours, vous allez apprendre à installer Python et ses librairies sur votre machine et/ou sur une machine distante sous Linux, en l'occurence un serveur AWS. Le choix entre les deux vous appartient. Le faire sur votre machine est plus simple à priori, mais le faire sur la machine distante vous sera sûrement utile à l'avenir, et notamment si vous suivez une formation diplômante OpenClassrooms en Data Science demandent la mise en place d'une API.

### Pourquoi travailler avec un notebook ?

La Data Science est itérative : il faut souvent tenter plusieurs approches et étudier les résultats avant de décider de la bonne façon de traiter un problème. Les notebooks, que nous allons prendre en main dans ce chapitre, sont parfaitement adaptés à cette particularité. Un notebook est une interface web dans laquelle vous pouvez taper du code Python, l'exécuter et voir directement les résultats, y compris une visualisation à l'aide de graphiques.

Les notebooks supportent aussi les commentaires formatés, ce qui vous permettra d'expliquer ce que vous faites (cela sera très utile, y compris pour vous-même) et commenter les résultats. J'ai par exemple préparé la plus grande partie de ce cours directement sur des notebooks

Sans notebook, vous devez écrire votre programme Python dans un fichier, l'enregistrer, puis le lancer à l'aide de la console avec la ligne de commande python monfichier.py.

Cependant, travailler de cette manière sera fastidieux. En effet, si votre programme affiche des graphiques (ce que nous ferons beaucoup dans ce cours !), ils s'afficheront soit tous en même temps dans des fenêtres séparées, soit ils s'afficheront les uns après les autres au fur et à mesure que vous fermez les fenêtres graphiques. De plus, il sera difficile de revenir en arrière dans votre programme sans tout relancer depuis le début.

Un notebook est la solution idéale !

Déjà, il est plus convivial car vous tapez votre code directement dans un navigateur web (type Firefox, Chrome, Safari, Spartan, feu Internet Explorer, etc.). Vous pourrez facilement revenir en arrière en exécutant les lignes de code que vous souhaitez. Enfin, les graphiques s'affichent dans le navigateur, et c'est bien plus pratique ! Vous pouvez essayer le notebook Jupyter [ici](https://jupyter.org/try), ou avoir un aperçu de son utilisation dans les vidéos de ce cours.

## Installez Jupyter sur votre propre ordinateur

Comme pour tous les sujets intéressants, nous avons besoin de quelques préparatifs pour bien nous lancer dans l'analyse de données avec Python.

Si vous choisissez d'installer le notebook en local, suivez ce chapitre et sautez le suivant. Si vous souhaitez utiliser un serveur distant, sautez ce chapitre et passer directement au suivant ! ;)

### Ce que nous allons faire dans ce chapitre

 Comment s'orienter dans ce vaste chapitre ?

* Si vous n'avez jamais intallé installé Python, alors il existe une [solution miracle](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5559646-installez-jupyter-sur-votre-propre-ordinateur#r-5061800), qui installera directement Python, toutes les librairies nécessaires, ainsi que le notebook Jupyter !
* Si vous avez déjà installé Python, alors vous aurez besoin d'[installer le notebook Jupyter](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5559646-installez-jupyter-sur-votre-propre-ordinateur#r-5062358), ainsi que [les librairies](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5559646-installez-jupyter-sur-votre-propre-ordinateur#r-5062363) suivantes :
  + Pandas
  + Matplotlib
  + Numpy
  + SciPy

### Installer la totale : Python + librairies + notebook (distribution Anaconda)

Si vous n'avez jamais installé Python, alors autant installer directement la **distribution Anaconda**. C'est ce que nous allons faire dans cette section.

En gros, une distribution, c'est un langage de programmation + certaines librairies et autres fonctionnalités.

Anaconda est donc une distribution Python, faite pour la Data Science.

Il installera donc :

* Python
* les librairies de Data Science dont nous aurons besoin : Matplotlib, Scipy, Numpy, Pandas
* le notebook Jupyter, que je vous conseille vivement d'utiliser

Anaconda installera tout ce dont nous avons besoin oui, mais il se peut qu'il en installe un peu trop (librairies que nous n'utiliserons pas, etc.). Il existe donc une version épurée d'Anaconda ici : [**https://conda.io/miniconda.html**](https://conda.io/miniconda.html). Mais certaines librairies, en particulier Jupyter, ne seront pas installés par défaut. C'est donc une solution réservée aux utilisateurs avertis.

Pour télécharger la distribution Anaconda, c'est par ici : <https://www.anaconda.com/distribution/>

Au cas où les instructions suivantes seraient obsolètes, les instructions d'installation officielles se trouvent sur cette page : [**https://docs.anaconda.com/anaconda/install**](https://docs.anaconda.com/anaconda/install)

#### Installer Anaconda sous Windows ou Mac

* Téléchargez le fichier d'installation pour [Windows](https://www.anaconda.com/download/#windows) ou [MacOs](https://www.anaconda.com/downloads#macos), puis lancez-le en double-cliquant sur le fichier qui s'est téléchargé.
* Répondez aux questions qui vous sont posées. Les options par défaut sont en général acceptables, pas besoin de les modifier. Une fois l'installation terminée, vérifiez que tout s'est bien passé en lançant le programme Jupyter (voir ci-dessous).

#### Installer Anaconda sous Linux Ubuntu

* Ouvrez une console
* Optionnellement, placez-vous dans le répertoire dans lequel vous souhaitez télécharger et décompresser les fichiers d'installation, grâce à la commande cd.
* Adaptez la commande ci-dessous en remplaçant 2020-02 par la dernière version disponible d'Anaconda (vous pouvez trouver le numéro de version sur cette page : <https://www.anaconda.com/distribution/>.

wget https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86\_64.sh

* Une fois adaptée, lancez cette commande.
* Le fichier est assez volumineux, et peut prendre quelques secondes à être téléchargé. Une fois qu'il est téléchargé, vous pouvez vérifier sa présence grâce à la commande ls.
* Adaptez la commande suivante en fonction de la version, puis exécutez-la.

bash Anaconda3-2020.02-Linux-x86\_64.sh

* Ceci lance le processus d'installation d'Anaconda. Vous devrez accepter la licence d'utilisation. Choisissez d'installer Anaconda dans le répertoire par défaut, et patientez. A la fin de l'installation, répondez yes à la question suivante :

Do you wish the installer to prepend the Anaconda3 install location

to PATH in your /home/ec2-user/.bashrc ? [yes|no]

* La commande précédente a ajouté au fichier .bashrc le chemin (le dossier) dans lequel se trouve Anaconda. Ainsi, vous pourrez lancer Anaconda directement en tapant seulement anaconda dans votre console ! Cette opération prendra effet au redémarrage de votre ordinateur, ou dès l'exécution de cette commande :

source .bashrc

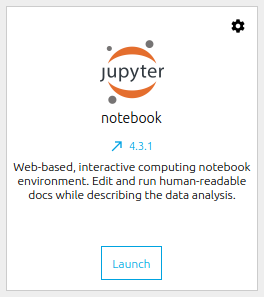
* Vérifiez que tout s'est bien passé en lançant le programme Jupyter (voir ci-dessous).

#### Lancer le programme Jupyter

* Sous MacOs, lancez Anaconda Navigator à partir du Launchpad (repérable sur votre Bureau grâce à l'icône ).
* Sous Linux Ubuntu, ouvrez une console et lancez la commande anaconda-navigator .
* Sous Windows, lancez Anaconda Navigator en le recharchant dans les programmes (selon la version de Windows, vous le trouverez probablement en cliquant sur démarrer > (Programmes) > Anaconda > Anaconda Navigator :

Si vous rencontrez des difficultés à installer Anaconda, parlez-en sur [**ce forum**](https://openclassrooms.com/forum/sujet/cours-nettoyez-et-decrivez-votre-jeu-de-donnees) ;).

Une fois le navigateur Anaconda lancé, plusieurs applications vous sont proposées. Cliquez sur Jupyter :



Quel que soit votre système d'exploitation, il existe un autre moyen de lancer Jupyter, en ouvrant une console (si vous ne savez pas comment faire, [**c'est par ici**](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/installez-r-ou-python-1#r-4927997)). Dans celle-ci, lancez la commande jupyter notebook. Si le notebook auquel vous voulez accéder existe déjà, alors changez de répertoire grâce à la [**commande cd**](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/installez-r-ou-python-1#r-4927987), afin de vous placer dans le répertoire dans lequel se trouve votre notebook. Lancez ensuite la commande jupyter notebook mon\_notebook.ipynb. Pour plus de précisions, rendez-vous [**ici**](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/installez-r-ou-python-1#r-4929093).

### Installer uniquement le notebook Jupyter (sans Anaconda)

Pour installer Python ainsi que le notebook Jupyter, il est fortement recommandé d'installer la distribution Anaconda. Pour cela, consultez la section [**Installer la totale : Python + librairies + notebook**](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5559646-installez-jupyter-sur-votre-propre-ordinateur#r-5061800).

Si vous ne souhaitez pas installer Anaconda, vous pouvez suivre les instructions suivantes, après avoir installé Python :

Les instructions suivantes proviennent de cette page : [**http://jupyter.org/install.html**](https://jupyter.org/install.html)

Assurez-vous que le programme pip est installé sur votre ordinateur. Pour cela, tapez tout simplement pip dans une [console](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/installez-r-ou-python-1#r-4927997). Normalement, le [programme pip](https://docs.python.org/fr/3.5/installing/index.html) s'est installé en même temps que Python.

Tapez ensuite ces lignes de code l'une après l'autre :

python -m pip install --upgrade pip

python -m pip install jupyter

Pour vérifier si l'installation s'est bien déroulée, tapez dans votre console la commande suivante :

jupyter notebook

Un redémarrage de votre ordinateur peut parfois être requis.

Vous êtes maintenant prêt(e) à [créer un nouveau notebook](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5559646-installez-jupyter-sur-votre-propre-ordinateur#r-5062053) !

### Installer uniquement les librairies Python (sans Anaconda)

Pour installer python ainsi que les librairies de Data Science, il est fortement recommandé d'installer la distribution Anaconda. Pour cela, consultez la section [**Installer la totale : Python + librairies + notebook**](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5559646-installez-jupyter-sur-votre-propre-ordinateur#r-5061800).

Assurez-vous que le programme  pip  est installé sur votre ordinateur. Pour cela, tapez tout simplement  pip  dans une [console](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/installez-r-ou-python-1#r-4927997). Normalement, le programme pip s'est installé en même temps que Python. Si ce n'est pas le cas, vous pouvez visiter [cette page](https://docs.python.org/fr/3.5/installing/index.html).

Ensuite, saisissez dans votre console ces commandes :

pip install scipy

pip install numpy

pip install matplotlib

pip install pandas

Pour les utilisateurs de linux, il est parfois nécessaire d'ajouter sudo avant pip  :  sudo pip install scipy.

## Lancez une session de notebook Jupyter sur AWS

Ce chapitre est utile seulement si vous n'avez pas installé le notebook Jupyter lors du chapitre précédent.

Pour bien profiter de ce chapitre, vous devez être un peu familiarisé avec les instances AWS et [la ligne de commande](https://openclassrooms.com/courses/reprenez-le-controle-a-l-aide-de-linux) sous Linux. En résumé, vous devez vous être inscrit sur AWS, pouvoir lancer une instance Ubuntu, et vous y connecter en SSH.

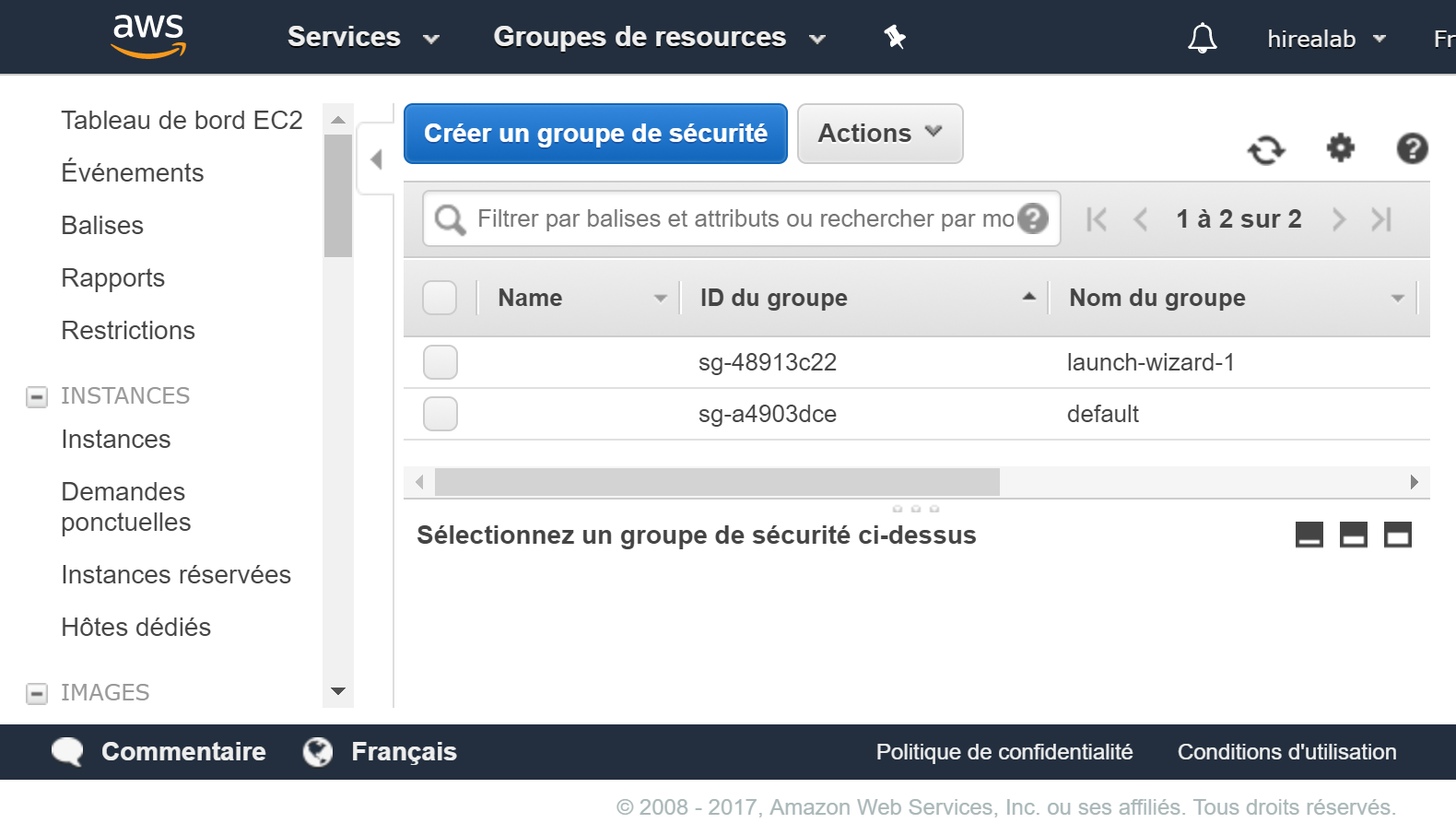
Vous pouvez aussi utiliser des services comme [**Colaboratory**](https://colab.research.google.com/notebook) qui vous permettent de facilement accéder à un notebook Jupyter.

Mais être capable d'utiliser AWS vous sera utile si vous vous formez au métier de [**Data Scientist**](https://openclassrooms.com/fr/paths/164-data-scientist). Cette plateforme a des avantages, comme utiliser plusieurs processeurs et même des cartes graphiques pour accélérer vos modèles. De manière plus prosaïque, vous pourrez aussi créer une API pour les modèles que vous élaborerez et permettre aux autres d'y accéder. Cela peut vous être utile pour vos projets si vous suivez les parcours [**Data Analyst**](https://openclassrooms.com/paths/data-analyst) ou [**Data Scientist**](https://openclassrooms.com/fr/paths/164-data-scientist).

### Configurez l'accès web à votre serveur

Les notebooks Jupyter s'utilisent dans le navigateur. Mais pour accéder à notre machine depuis un navigateur, nous devons demander à Amazon d'ouvrir un port utilisé par Jupyter. Par défaut, nous allons utiliser les ports 8888 et 443 (pour les protocoles http et https, respectivement).

Dans l'interface de gestion d'AWS, dans la barre de menu à gauche, cliquez sur Groupes de sécurité dans la rubrique Réseau et Sécurité. Vous devriez arriver sur une page ressemblant à ceci :

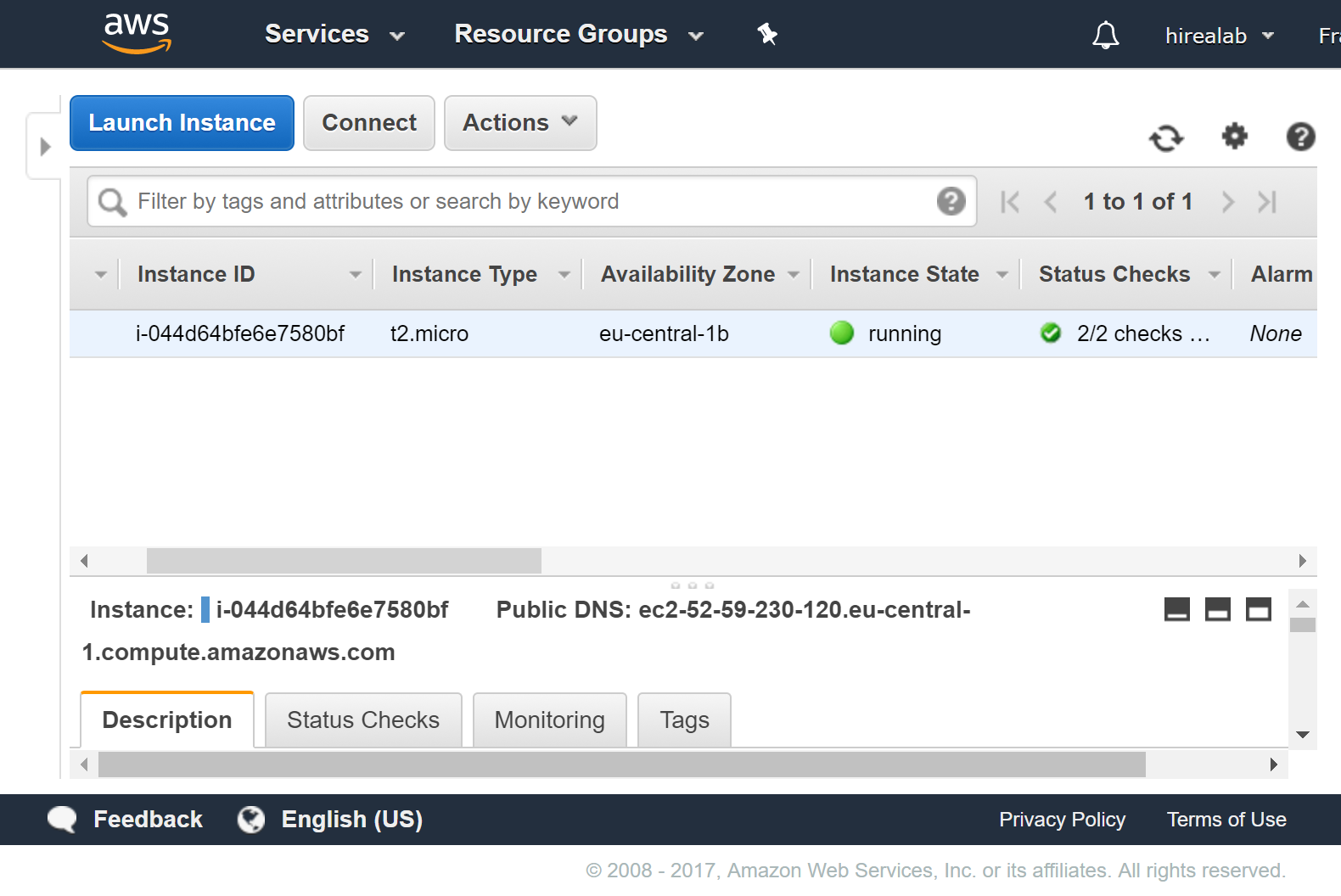


Interface de gestion des groupes de sécurité

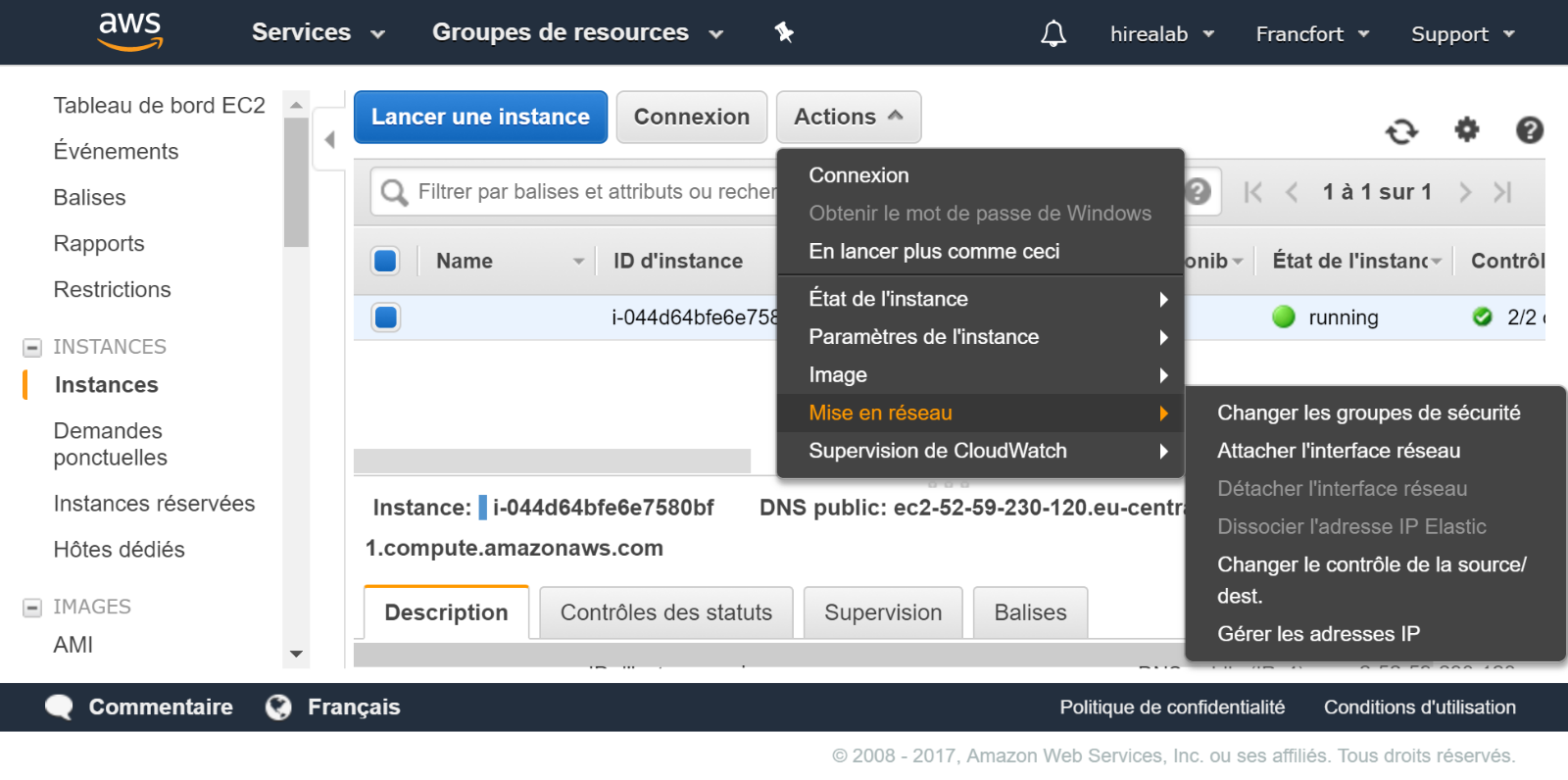
Cliquez sur le bouton Créer un groupe de sécurité. Dans le formulaire qui s'ouvre, remplissez les champs comme ceci et cliquez sur Créer.

Réglages du groupe de sécurité

Rendez-vous sur l'interface web d'AWS, et accédez à la liste de vos instances. Elle devrait ressembler à ceci:

La liste des instances AWS

Cliquez sur votre instance favorite, et ensuite sur le bouton Actions en haut de l'écran. Dans les menus qui s'ouvrent, choisissez Mise en réseau, puis Changer les groupes de sécurité.

Changer les groupes de sécurité de votre serveur

Dans la liste qui s'affiche, choisissez le groupe que nous venons de créer. Vous pouvez décocher les autres groupes éventuellement présents. Enfin, cliquer sur Attribuer les groupes de sécurité.

Voilà, c'est terminé. Nous pouvons maintenant nous concentrer sur la préparation de notre machine.

### Installez Anaconda sur votre serveur

Anaconda est une distribution de Python, c'est-à-dire un paquet qui contient Python et plusieurs bibliothèques et utilitaires associés. Par défaut, elle contient toutes (ou quasiment) les librairies dont vous aurez besoin pour la Data Science. Nous allons l'installer sur notre machine. Cela est très facile grâce à l'installateur fourni sur le [site officiel](https://www.anaconda.com/distribution/#linux).

Si vous êtes à l'aise avec le gestionnaire de paquets d'Ubuntu et l'outil pip (Python Package Manager) vous pouvez aussi installer Python et les librairies Numpy, Scipy, Matplotlib, Panda et Flask manuellement.

Nous allons faire télécharger le fichier à notre machine, et le lui faire installer. Pour cela, vous devez d'abord vous connecter en SSH à votre serveur pour avoir une ligne de commande. Une fois que c'est fait, tapez ceci dans la ligne de commande :

wget https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86\_64.sh

Ceci installera la version 5.0.1 de Anaconda. Pour connaître le nom de la dernière version, il suffit d'un clic droit sur le bouton télécharger sur le site officiel et de copier la cible du lien.

Le fichier est assez volumineux et peut prendre quelques secondes à être téléchargé. Une fois qu'il est téléchargé, vous pouvez vérifier sa présence grâce à la commande ls.

Nous allons exécuter ce script.

bash Anaconda3-2020.02-Linux-x86\_64.sh

Ceci lance le processus d'installation d'Anaconda. Vous devrez accepter la licence d'utilisation. Choisissez d'installer Anaconda dans le répertoire par défaut, et patientez. A la fin de l'installation, répondez yes à la question :

Do you wish the installer to prepend the Anaconda3 install location

to PATH in your /home/ec2-user/.bashrc ? [yes|no]

Une fois l'installation finie, il nous reste un tout petit réglage en plus. Nous pouvons soit fermer la session SSH et en ouvrir une autre, soit taper cette commande :

source .bashrc

Si tout va bien, nous devrions pouvoir vérifier notre installation en tapant which pip.

~/anaconda3/bin/pip

Voilà ! Nous avons installé Anaconda. Nous devons maintenant configurer Jupyter pour permettre les accès distants.

### Configurez Jupyter pour les accès distants

Par défaut, Jupyter ne permet que les connexions depuis la même machine que celle sur laquelle le serveur tourne. Pour nous y connecter depuis l'extérieur, nous avons quelques réglages à faire. Ce qui se passe exactement lorsque nous allons taper les commandes suivantes sort du champ de ce cours. Vous pouvez vous contenter des copier-coller dans une fenêtre de commande SSH. Je vais quand même tenter de vous donner une idée de ce qui se passe.

#### Générez un fichier de configuration et un mot de passe

La première étape est de générer un fichier de configuration. Cette commande crée un fichier de configuration par défaut, que nous allons modifier par la suite.

jupyter notebook --generate-config

Ensuite, nous allons générer un mot de passe.

key=$(python -c "from notebook.auth import passwd; print(passwd())")

Cette commande vous demandera de taper deux fois un mot de passe de votre choix, et l'enregistre dans une variable. Tâchez de vous souvenir de ce mot de passe. Vous en aurez besoin pour vous connecter au serveur.

#### Configurez le protocole HTTPS pour Jupyter

cd ~

mkdir certs

cd certs

certdir=$(pwd)

openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mycert.key -out mycert.pem

Ces commandes créent un dossier appelé certs, et y génèrent les clés de chiffrement nécessaires pour une connexion sécurisée (enfin, à peu près sécurisée, mais cela sort du champ de ce cours). La dernière commande vous posera plein de questions. Vous pouvez laisser la réponse vide à toutes les questions. Contentez-vous d'appuyer sur Entrée jusqu'à ce que la commande termine.

#### Modifiez le fichier de configuration de Jupyter

C'est presque fini. Nous n'avons plus qu'à insérer toutes ces informations dans le fichier de configuration de Jupyter. Pour cela, il suffit de taper les commandes suivantes (vous pouvez les copier-coller) :

cd ~

sed -i "1a\

c = get\_config()\\

c.NotebookApp.certfile = u\'$certdir/mycert.pem\'\\

c.NotebookApp.keyfile = u\'$certdir/mycert.key\'\\

c.NotebookApp.ip = u\'0.0.0.0\'\\

c.NotebookApp.open\_browser = False\\

c.NotebookApp.password = u\'$key\'\\

c.NotebookApp.port = 8888" .jupyter/jupyter\_notebook\_config.py

Ça y est, tout est prêt.

### Lancez un notebook Jupyter

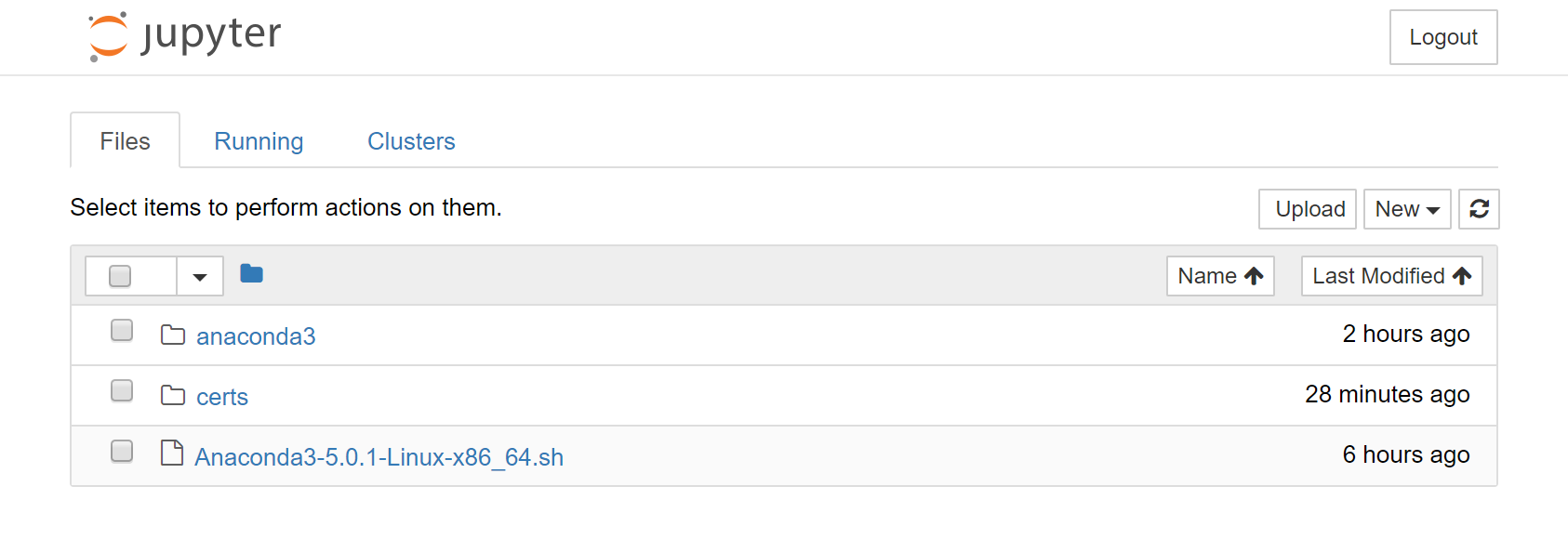
Nous pouvons enfin lancer une session Jupyter et nous y connecter depuis notre machine personnelle. Tout ce que nous avons fait jusqu'ici n'a besoin d'être fait qu'une seule fois. Pour lancer une session du notebook, tapez la commande suivante dans une fenêtre SSH.

jupyter notebook

Ensuite, nous devons nous connecter à cette session. Lancez un navigateur sur votre machine personnelle. Vous devez aller à l'adresse  https://<adresse de votre serveur aws>:8888.

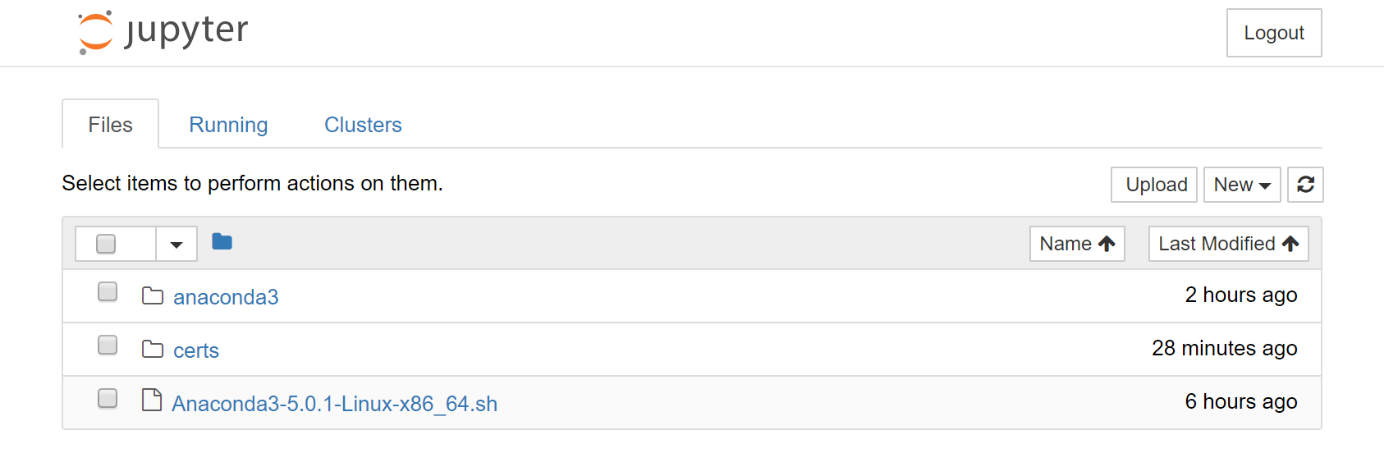
Faites attention au HTTPS ! Nous devons utiliser un protocole sécurisé. Comme nous n'utilisons pas un vrai certificat pour notre connexion HTTPS, votre navigateur vous alertera certainement sur le fait que votre connexion n'est pas sécurisée. Vous pouvez ignorer cet avertissement pour le moment. Dans le navigateur Chrome, cela se fait en cliquant sur le bouton "Avancé" en bas à gauche de la page d'avertissement.

Jupyter vous demandera ensuite de taper votre mot de passe. Une fois cela fait, vous arriverez enfin sur la page d'accueil des notebooks.

La page d'accueil de Jupyter

## Faîtes vos premiers pas dans un notebook Jupyter

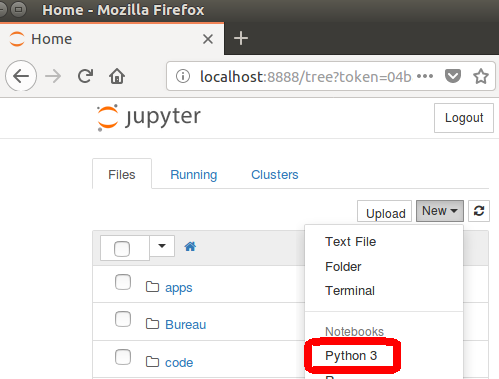
Félicitations, vous êtes arrivé à la page d'accueil de Jupyter!



La page d'accueill de Jupyter

### Créer un nouveau notebook

A partir de la fenêtre principale de Jupyter, cliquez sur "New" puis sur "Python 3", comme ceci :

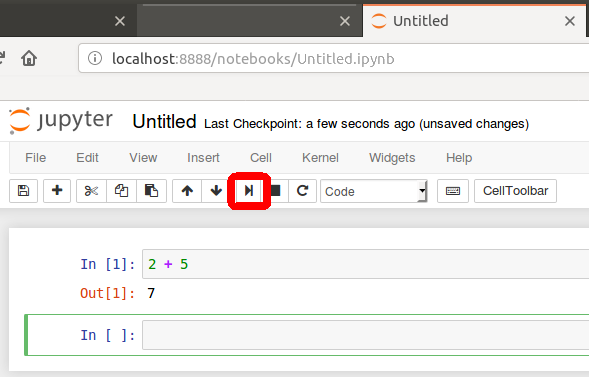


Créez un notebook

Un fichier nommé Untitled.ipynb s'est normalement créé dans le répertoire à partir duquel vous avez lancé Jupyter.

Votre navigateur devrait vous afficher le notebook créé. Vous pouvez changer le nom du notebook en cliquant sur son nom actuel, qui est Untitled. Les commandes Python doivent être tapées dans la case devant l'étiquette In [ ]. Vous pouvez taper plusieurs instructions à la fois. Vous pouvez même définir des fonctions. Les variables générées dans chaque case seront disponibles dans toutes le cases du notebook. Une fois que vous avez fini de les taper, appuyez sur Shift+Entrée pour les exécuter. Le résultat s'affichera directement en dessous et une nouvelle case est créée pour vous permettre de taper les instructions suivantes.

Pour le tester, tapez 2 + 5 dans la case (cellule) vide au centre de la fenêtre. Cliquez ensuite sur ce bouton :



Exécuter une cellule du notebook

Il devrait s'afficher le résultat de l'opération : 7.

### Lancer un notebook déjà créé

Une fois votre notebook créé, vous pouvez y réaccéder à tout instant en ouvrant une console, en vous plaçant dans le répertoire qui contient votre notebook (que vous appelons ici Untitled.ipynb).

Pour changer de répertoire dans une console, lisez le paragraphe sur la commande [**cd**](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/installez-r-ou-python-1#r-4927987).

Saisissez dans la console cette ligne :

jupyter notebook Untitled.ipynb

Normalement, votre navigateur devrait s'ouvrir automatiquement et afficher votre notebook !

### Enregistrez les fonctions très souvent utilisées dans un module

Les notebooks sont très pratiques pour un travail itératif. Cependant, vous aurez souvent besoin d'utiliser une fonction que vous avez écrite, à plusieurs endroits différents. Par exemple, vous aurez peut-être un jour une fonction qui normalise vos données comme vous le souhaitez, et vous souhaitez y avoir accès depuis tous vos notebooks, sans avoir besoin de copier-coller dans chaque notebook.

Pour cela, vous pouvez écrire un module Python. Un module Python est un simple script. Une fois importé dans un autre script ou notebook, toutes les variables (et donc les fonctions) du module seront à disposition du script qui l'a importé.

Pour écrire un module, cliquez sur "Text file" dans menu "New" de la page d'accueil de Jupyter. Une nouvelle page s'ouvre où vous pouvez taper du code. À l'intérieur, nous allons taper une fonction qui permet d'imprimer les éléments de la suite de Fibonacci jusqu'à n. Changer le nom de ce fichier en fibo.py.

def fib(n):

a, b = 0, 1

while b < n:

print(b, end=' ')

a, b = b, a+b

print()

Retournons maintenant dans notre notebook. Nous pouvons importer cette fonction et l'exécuter dans le notebook comme ceci :

import fibo

fibo.fib(10)

Voilà ! C'est tout pour ce chapitre. Après l'installation du notebook Jupyter en local ou sur le cloud (AWS), nous allons l'utiliser pour résoudre une énigme célèbre, le problème de Monty Hall !

## Découvrez le problème de Monty Hall

Cette partie du cours concerne plutôt les **Data Scientists**. Si vous êtes **Data Analyst**, passez directement à ce chapitre : [**Plongez en détail dans la librairie NumPy**](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/4740941-plongez-en-detail-dans-la-librairie-numpy).

Dans ce chapitre, nous allons nous rappeler les notions de base de la programmation en Python. Et plutôt que de faire une liste de tout ce que nous avons besoin de maîtriser, nous allons simuler un problème de probabilités.

Le problème de Monty Hall est connu pour sa solution contre-intuitive, même une fois qu’elle est expliquée. La première fois que j’ai entendu parler de ce problème, mon incrédulité m’a amené à simuler les conditions du jeu dans un programme pour me convaincre du bien-fondé de la solution. Aujourd’hui, c’est à votre tour.

### Le problème de Monty Hall

L'exposé du problème est plutôt simple. Imaginez un jeu télévisé où il y a trois portes sur le plateau de jeu. Seule une de ces portes cache un trésor. Il n'y a rien derrière les deux autres portes. Rien ne permet de savoir quelle porte cache le trésor.

La tâche du joueur consiste à choisir parmi les trois portes celle qu'il veut ouvrir. Il aura droit au trésor s'il choisit la bonne porte, et rien sinon. Pour faire son choix, il n'a aucune information. Il doit donc simplement s'en remettre au hasard.

Jusqu'ici, le problème n'a rien de remarquable. Mais il y a un twist ! Une fois que le joueur a fait son choix, mais avant d'ouvrir la porte, le présentateur élimine, parmi les deux portes non choisies, une porte qui ne contient pas de trésor. Si les deux portes restantes ne contiennent rien, le présentateur élimine simplement une d'entre elles au hasard.

Le joueur doit alors faire un nouveau choix. Il peut soit choisir d'ouvrir la première porte qu'il avait choisie, soit changer pour la porte non éliminée par le présentateur. La question est, qu'a-t-il intérêt à faire ? Réfléchissez-y. Vous n'avez en principe pas besoin de connaissances poussées en probabilités pour résoudre ce problème. Que feriez-vous ?

### Simulation

#### Génération d'une seule partie de jeu

Allons-y. Nous allons commencer par écrire une fonction qui génère une partie du jeu. Cette fonction choisit aléatoirement une porte parmi les trois pour y cacher le trésor. Elle choisit ensuite aléatoirement le premier choix du participant, et élimine une des deux portes restantes. Ensuite, une nouvelle porte est choisie selon la stratégie adoptée par le joueur :

* Changer de porte
* Ne pas changer de porte

Enfin, le gain du participant est calculé.

Tout d’abord, préparons notre environnement de travail dans un notebook :

# Pour afficher les graphiques dans la continuité du code,

# et non pas dans une fenêtre à part:

%matplotlib inline

# Pour utiliser la fonction randint, qui génère des nombres

# entiers de façon aléatoire:

from random import randint, seed

# Un Enum est une structure de données qui consiste en un

# ensemble d'éléments nommés. Une variable de ce type peut

# avoir comme valeur un de ces éléments.

from enum import Enum

# Pour pouvoir afficher des graphiques:

import matplotlib.pyplot as plt

#### Les stratégies du joueur

Maintenant nous allons définir les stratégies possibles pour le joueur. Il peut soit choisir de changer, soit de garder son choix initial. Pour cela, nous tirons profit de la classe  Enum  . Cette approche nous permet d'avoir des noms compréhensibles pour nos stratégies. Une autre approche consisterait à désigner chaque stratégie par un entier, par exemple 0 pour changer de porte, et 1 pour garder le choix initial.

Mais cette approche a des défauts :

* elle ne permet pas de contraindre le choix au moment où quelqu'un d'autre utilisera notre fonction. Un utilisateur final pourra appeler notre fonction de simulation en utilisant comme stratégie un entier non pris en charge, comme 2, et ne saura pas que cela posera problème à moins que nous implémentions des vérifications.
* le code est moins lisible. Il faut se rappeler ce que chaque nombre signifie pour comprendre le programme.

# Ici nous définissons une sous-classe de Enum, qui contiendra

# les stratégies possibles.

class Strategie(Enum):

CHANGER = 1

GARDER = 2

Ce code fait appel à des notions de [**programmation orientée objet**](https://openclassrooms.com/courses/decouvrez-la-programmation-orientee-objet-avec-python), notamment aux notions de classe et d'héritage. Vous ne devriez cependant pas avoir de mal à comprendre ce chapitre si vous ne maîtrisez pas ces concepts.

Enfin, nous pouvons définir notre fonction. Nous allons écrire une fonction très simple. Elle ne simule qu'une seule partie du jeu pour une stratégie donnée. Rappelez-vous comment on définit une fonction en Python.

Le commentaire en plusieurs lignes après la déclaration du nom de la fonction (lignes 6 à 18) est un docstring. Par convention, il est utilisé par plusieurs outils pour générer une documentation. C'est une très bonne idée de toujours commenter vos fonctions. Un bon commentaire doit expliquer non seulement ce que la fonction fait, mais aussi pourquoi.

Notez la séparation entre la première ligne du docstring, qui contient une description abrégée, et le reste, qui sert à décrire plus précisément la fonction. Dans la liste des arguments, on peut définir le type attendu entre parenthèses. Pour plus d'exemples d'utilisation des docstring, je vous invite à consulter [**cette page**](https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html).

# Utilise l'horloge système pour initialiser le générateur de

# nombres pseudo-aléatoires.

seed()

def play\_game(strategie):

'''Simule une partie du jeu Monty Hall.

Cette fonction simule le choix de la porte par le participant,

l'élimination d'une mauvaise porte par le présentateur, et le

choix final. Elle ne retourne que le résultat de la partie, parce

que nous n'aurons besoin que du résultat pour effectuer nos calculs.

Args:

strategie (Strategie): La stratégie du joueur

Returns:

bool: Le joueur a-t-il gagné?

'''

portes = [0, 1, 2]

bonne\_porte = randint(0,2)

# Choix du joueur

premier\_choix = randint(0,2)

# Il nous reste deux portes

portes.remove(premier\_choix)

# Le présentateur élimine une porte

if premier\_choix == bonne\_porte:

portes.remove(portes[randint(0,1)])

else:

portes = [bonne\_porte]

deuxieme\_choix = 0

# Le deuxieme choix depend de la strategie

if strategie == Strategie.CHANGER:

deuxieme\_choix = portes[0]

elif strategie == Strategie.GARDER:

deuxieme\_choix = premier\_choix

else:

raise ValueError("Stratégie non reconnue!")

return deuxieme\_choix == bonne\_porte

La fonction randint retourne un entier aléatoire compris entre ses deux arguments. Par exemple, randint(0,2) retournera 0, 1, ou 2. Chaque appel à la fonction entraînera la génération d'une nouvelle valeur. Il est donc important de sauvegarder la valeur de retour de la fonction dans une variable si on a besoin de la réutiliser par la suite.

randint, ainsi que d'autres fonctions de la librairie random, utilisent en fait un générateur pseudo-aléatoire. Les détails de cette implémentation ne sont pas importants pour nous. Ce qu'il faut savoir, c'est que ces fonctions génèrent une suite de nombres aléatoires en se basant sur une valeur de base. Cette valeur de base peut être définie par l'utilisateur grâce à la fonction seed. En utilisant cette fonction avec une valeur fixe, la même séquence sera générée d'une exécution du programme à l'autre. Si aucun argument n'est donné à seed, l'horloge du système est utilisée pour avoir une valeur.

seed()

print("Premier nombre aléatoire: {}".format(randint(0,100)))

print("Deuxième nombre aléatoire: {}".format(randint(0,100)))

seed(1)

print("Premier nombre aléatoire: {}".format(randint(0,100)))

print("Deuxième nombre aléatoire: {}".format(randint(0,100)))

seed(1)

# Les deux prochains appels vont retourner le même résultat que les deux précédents

print("Premier nombre aléatoire: {}".format(randint(0,100)))

print("Deuxième nombre aléatoire: {}".format(randint(0,100)))

#### Simulation de plusieurs parties

Maintenant nous allons essayer notre fonction. N'hésitez pas à lancer la ligne suivante plusieurs fois pour vous convaincre que le résultat est aléatoire.

play\_game(Strategie.CHANGER)

Bien ! Nous pouvons simuler une partie du jeu. Mais pour pouvoir nous convaincre de la réponse à l'énigme, nous avons besoin de jouer beaucoup de fois, noter les résultats, et ensuite en tirer des conclusions. Heureusement, nous avons un ordinateur sous la main.

Nous allons définir une fonction qui lancera le jeu autant de fois que nous le souhaitons, et retournera le résultat de chaque partie dans une  list. Pour pouvoir exécuter des calculs sur ces résultats, nous allons aussi les stocker non plus comme des variables booléennes (Vrai ou Faux) mais en terme du gain du joueur (1 s'il a gagné, 0 s'il a perdu).

def play(strategie, nb\_tours):

'''Simule une suite de tours du jeu.

Cette fonction renvoie les résultats de plusieurs parties

du jeu Monty Hall sous forme d'une liste de gains par le

joueur.

Args:

strategie (Strategie): La strategie du joueur

nb\_tours (int): Nombre de tours

Returns:

list: Liste des gains du joueurs à chaque partie

'''

# Ceci est une liste en compréhension. Pour en savoir plus, consulter

# le cours "Apprenez à programmer en Python" sur OpenClassrooms

return [1 if play\_game(strategie) else 0 for i in range(nb\_tours)]

### Analyse des résultats

La première chose à faire est de savoir, pour un nombre de parties donné (disons, 10000), quelle stratégie rapporte le plus au joueur. Nous avons une liste contenant autant de 1 que de nombre de parties gagnées par le joueur. Il nous suffit de calculer la somme de tous les éléments de cette liste, avec la fonction sum, pour connaître le nombre de 1.

print("En changeant de porte, le joueur a gagné {} sur 10000 parties."

.format(sum(play(Strategie.CHANGER, 10000))))

print("En gardant son choix initial, le joueur a gagné {} sur 10000 parties."

.format(sum(play(Strategie.GARDER, 10000))))

En changeant de porte, le joueur a gagné 6666 sur 10000 parties.

En gardant son choix initial, le joueur a gagné 3320 sur 10000 parties.

Le joueur qui a changé de porte a donc gagné approximativement deux fois plus souvent que le joueur qui a conservé son choix initial. Vous y attendiez-vous ? Et même après l'avoir simulé, n'avez-vous pas, comme moi, du mal à le croire ?

### Visualisation

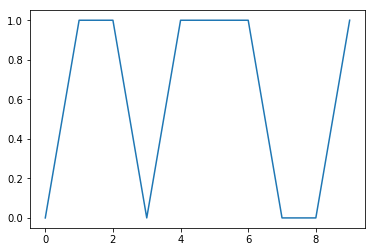
Regardons les résultats de la simulation de plus près. Nous n'allons pas pouvoir regarder les 10000 parties une par une, mais heureusement, nous pouvons créer des graphiques grâce à Matplotlib.

#### La fonction  plot

# plot renvoie un objet, que l'on pourra manipuler plus tard pour

# personnaliser le graphique

plot = plt.plot(play(Strategie.CHANGER, 10))



Premier graphique généré avec Matplotlib

Voilà. C'est aussi simple que cela. La fonction plot place les points donnés en argument sur un graphique. Essayons de créer un graphique avec 10000 points.

plot = plt.plot(play(Strategie.CHANGER, 10000))

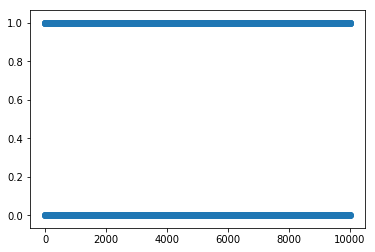


Le même graphique mais avec 10000 points

Avons-nous dessiné un carré ? Que se passe-t-il ? Le carré que vous voyez est dû au fait que plot relie les points que nous lui demandons de placer sur le graphe par des lignes. Ce sont ces lignes qui recouvrent toute la surface du graphe, et non pas les points que nous voudrions en fait voir. De manière générale, ces lignes peuvent être trompeuses, parce qu'elles peuvent aussi donner l'impression que nous avons des données entre deux points, alors que ce n'est pas le cas. Pour cette raison, nous allons placer les points sur le graphe sans les relier entre eux.

#### La fonction scatter

plot = plt.scatter(range(10000), play(Strategie.CHANGER, 10000))



Scatter ne relie pas les points entre eux

La fonction scatter se comporte comme plot, mais ne relie pas les points entre eux. Ici nous voyons bien que nos points sont bien situés là où ils doivent être.

Une compétence essentielle pour tout data scientist est de savoir présenter les résultats dans le bon format. Dans notre cas, nous voulons montrer la différence entre les gains des deux stratégies. Que pensez-vous être le bon type de graphique pour communiquer ce résultat ?

#### La fonction  bar

plot = plt.bar([1,2],[sum(play(Strategie.CHANGER, 10000)),

sum(play(Strategie.GARDER, 10000))],

tick\_label=["Changer","Garder"])



Les gains pour chaque joueur

Si vous avez pensé à des colonnes, bravo. Vous êtes sur la bonne voie.  bar est un peu différente des fonctions  plot  et  scatter. Elle requiert en entrée les coordonnées des colonnes, ici [1,2]. Remarquez que vous pouvez définir vos propres titres pour chaque colonne grâce à l'argument optionnel tick\_label.

#### Visualisez une liste de nombres

Comment pourriez-vous suivre l'évolution des gains des joueurs en fonction du nombre de parties ? Nous pouvons vérifier que notre programme se comporte de manière raisonnable si nous détectons une relation linéaire entre les gains des joueurs et le nombre de parties qu'ils ont joué.

gains\_changer = []

gains\_garder = []

samples = [1000, 10000, 20000, 50000, 80000, 100000]

for tours in samples:

gains\_changer.append(play(Strategie.CHANGER, tours))

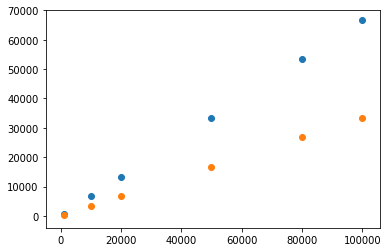
gains\_garder.append(play(Strategie.GARDER, tours))

Nous avons maintenant deux listes. L'une contenant les gains d'un joueur qui change de porte systématiquement après 1000, 10000, 20000, 50000, 80000 et 100000 parties. L'autre contient la même chose, mais pour un joueur qui ne change jamais de porte. Nous allons maintenant afficher les deux courbes correspondant à ces listes sur le même graphique. Remarquez que nous donnons deux arguments à scatter. Le premier est la liste des abscisses. Pourquoi pensez-vous que c'est nécessaire ?

figure = plt.figure()

plot = plt.scatter(samples, [sum(x) for x in gains\_changer])

plot = plt.scatter(samples, [sum(x) for x in gains\_garder])



L'évolution des gains des joueurs

Notez la fonction figure. Elle crée un graphique. Les appels suivants aux fonctions telles que scatter et plot génèrent des courbes sur ce même graphique.

Nous remarquons que les gains des joueurs semblent avoir une relation linéaire avec le nombre de parties qu'ils ont jouées. C'est bon signe. Par la suite, vous allez apprendre à parcourir le chemin inverse, c'est-à-dire partir de données telles que celles présentées ici, pour arriver à la relation qui les lie à leurs paramètres.

Pour le moment, regardons une autre donnée, la moyenne de gains de chaque stratégie par partie. Il suffit pour cela de diviser la somme des gains par le nombre de parties. Je vous laisse créer ce graphique par vous-même. Vous devriez voir que les moyennes se situent autour de 1/3 pour le joueur qui ne change pas de porte, et 2/3 pour celui qui change.

### Explication théorique

L'explication de ce résultat est finalement assez simple. Quand le joueur choisit une porte au début du jeu, il a une chance sur trois de tomber sur la porte qui cache le trésor. La probabilité que le trésor se cache derrière une des deux portes restantes est donc de 2/3.

L'élimination par le présentateur d'une des deux portes restantes ne change pas ces probabilités. Il élimine forcément une porte qui ne cache pas de trésor. La probabilité que la porte qu'il n'élimine pas contienne le trésor devient égale à 2/3.

Ce résultat est assez contre-intuitif. Mais nous venons de le montrer de façon empirique. Si vous voulez en savoir plus, n'hésitez pas à consulter [l'article Wikipédia](https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_Monty_Hall) qui y est dédié.

### Petit exercice

Nous pouvons imaginer une autre stratégie, où le joueur choisit aléatoirement une porte entre la première et celle que le présentateur n'a pas éliminée. Implémentez cette stratégie et observez les résultats. Vous devriez trouver que le gain attendu se situe à mi-chemin des gains attendus avec les deux stratégies initiales.

## Utilisez Numpy pour illustrer le théorème central limite

Ce chapitre contient quelques notions de [**probabilités**](https://openclassrooms.com/courses/maitrisez-les-bases-des-probabilites). Vous pouvez cependant le comprendre sans maîtriser ce domaine. Pour avoir une première intuition de ce qu'est le théorème de la limite centrale en 6minutes, faites un petit tour sur [**cette vidéo**](https://www.youtube.com/watch?v=4dhm2QAA2x4) ;).

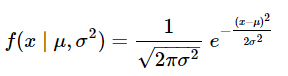
Dans le chapitre précédent, nous nous sommes brièvement rappelé les bases de la programmation en Python, et nous nous sommes frottés à la visualisation des données avec Matplotlib. Dans ce chapitre, nous allons commencer à utiliser une autre librairie de la pile scientifique de Python, Numpy.

Numpy contient de nombreuses fonctions et structures de données orientées vers l'algèbre linéaire et l'analyse de données. Vous pouvez imaginer Numpy comme l'équivalent en Python du logiciel Matlab (sans les capacités graphiques, qui sont fournies par Matplotlib).

Pour cette première prise en main, nous allons nous attaquer à un théorème fondamental de la théorie des probabilités, le [théorème de la limite centrale](https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_central_limite).

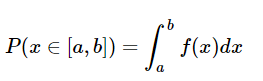
### La distribution Gaussienne

Si à un moment de votre vie, vous vous êtes intéressés aux probabilités, vous avez certainement entendu parler de la distribution Gaussienne, que l'on peut représenter par sa densité :



où  **μ** représente la moyenne, et  **σ** l'écart type de la distribution. Pour une variable aléatoire, la densité a une définition simple.

La probabilité que la variable **x** prenne une valeur entre **a** et **b** est donnée par l'intégrale de la densité de probabilité de **x** entre **a** et **b** . En termes mathématiques :



La forme de cette fonction est ... eh bien pourquoi ne pas utiliser Python pour la dessiner ?

%matplotlib inline

import numpy as np

import matplotlib.pyplot as plt

from math import sqrt, pi, exp

domaine = range(-100,100)

mu = 0

sigma = 20

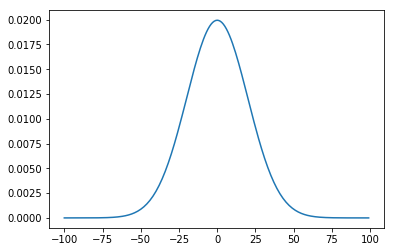
# Notez l'utilisation du mot clé lambda. Il est très bien expliqué dans le cours

# "Apprenez à programmer en Python" sur OpenClassrooms.

f = lambda x : 1/(sqrt(2\*pi\*pow(sigma,2))) \* exp(-pow((x-mu),2)/(2\*pow(sigma,2)))

y = [f(x) for x in domaine]

plot = plt.plot(domaine, y)



La courbe en cloche

Et voilà la fameuse courbe en cloche. Remarquez qu'elle est centrée en 0 (parce que  **μ=0**). Cette fonction est intégrable entre **−∞** et **∞**. Mais la fonction vaut presque zéro (sans jamais être égale à 0) dès qu'on s'éloigne de son centre. La vitesse à laquelle  **f** tend vers 0 en s'éloignant de **μ** est donnée par **σ**. Essayez de varier les paramètres **μ** et **σ** pour vous rendre compte de leur impact. Par exemple, observez la courbe avec **μ=50**, et tentez aussi **σ=5**.

### Le théorème central limite

Cette distribution est aussi couramment appelée la distribution [*normale*](https://fr.wikipedia.org/wiki/Loi_normale). La raison de cette appellation est une propriété remarquable : beaucoup d'observations dans le monde réel semblent suivre cette loi. Par exemple, si on mesurait la taille de tous les humains sur terre et affichait l'histogramme de nos observations, nous obtiendrions une courbe très proche de celle que vous voyez au-dessus (avec une moyenne et un écart type différents, bien évidemment).

Cette propriété découle d'un théorème fondamental des probabilités, le **théorème central limite**. Intuitivement, ce théorème déclare que toute somme de variables aléatoires indépendantes et identiquement distribuées tend vers une variable aléatoire gaussienne. Nous allons dans ce chapitre illustrer ce théorème, sans tenter de le démontrer (ce n'est pas chose aisée).

### Générez des réalisations de variables aléatoires avec Numpy

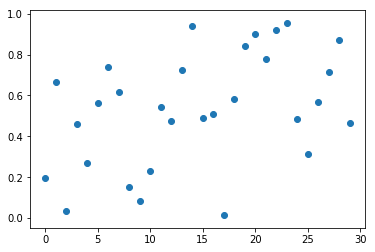
Pour commencer, nous allons générer des réalisations de variables aléatoires identiquement distribuées.

Ouf ! 6 mots de vocabulaire en une même phrase ! Pas d'inquiétude : ce n'est rien d'autre que ce que nous avons vu au chapitre précédent en utilisant les fonctions de la librairie random.

Dans ce chapitre, nous allons plutôt utiliser la librairie Numpy, qui rend cette tâche bien plus facile.

vecteur\_aleatoire = np.random.rand(30)

plot = plt.scatter(range(30),vecteur\_aleatoire)



Réalisation d'une variable aléatoire uniformément distribuée

Voilà. Ce n’est pas bien difficile. Remarquez que ces variables sont uniformément distribuées, c'est-à-dire qu'elles ne semblent pas favoriser une valeur en particulier. Et si au lieu d'un vecteur, nous voulions une matrice ?

Une [**matrice**](https://fr.wikipedia.org/wiki/Matrice_(math%C3%A9matiques)), c'est un tableau de nombres, avec des lignes et des colonnes.

matrice\_aleatoire\_a\_imprimer = np.random.rand(3,5)

print(matrice\_aleatoire\_a\_imprimer)

matrice\_aleatoire\_a\_grapher = np.random.rand(2,30)

fig = plt.figure()

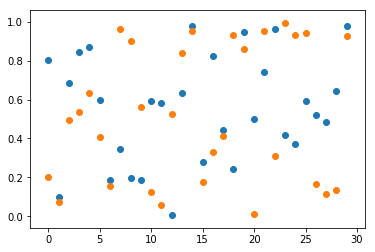
plot1 = plt.scatter(range(30),matrice\_aleatoire\_a\_grapher[0,:])

plot2 = plt.scatter(range(30),matrice\_aleatoire\_a\_grapher[1,:])

[[ 0.28346313 0.31098521 0.85605644 0.06346737 0.94141766]

[ 0.85912141 0.03400195 0.41018347 0.88199074 0.41266676]

[ 0.61338095 0.98663708 0.99128196 0.46062721 0.69029766]]



Une matrice de variable aléatoires uniformément distribuées

La fonction rand prend en argument la taille de la matrice que nous souhaitons générer. Pour accéder à une colonne ou une ligne dans cette matrice, il suffit d'utiliser la syntaxe [], comme s'il s'agissait d'une liste. Le signe :  demande à Numpy de nous donner toutes les colonnes, ce qui nous dispense d'écrire explicitement la taille de la matrice. Par exemple, pour une matrice de taille 2x3,  matrice[0:2][0:3]  et  matrice[:,:]  sont équivalents.

### Les opérations de base sur les matrices Numpy

Intéressons-nous maintenant à notre problème. Nous voulons simuler un grand nombre de variables aléatoires. Donc nous devons générer, pour chaque variable aléatoire, un grand échantillon (c'est-à-dire un échantillon avec beaucoup d'individus).

Pour les notions de variable, d'individu et d'échantillon, suivez-moi par [**ici**](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/decouvrez-les-statistiques-vocabulaire-et-tour-dhorizon-1) !

En prenant par exemple le cas où ces variables aléatoires sont uniformément distribuées dans l'intervalle [0,1], il nous suffit de générer une matrice avec la fonction rand.

[***Uniformément distribuée***](https://fr.wikipedia.org/wiki/Loi_uniforme_continue) dans [0, 1] signifie qu'aucune valeur entre 0 et 1 ne sera favorisée par rapport aux autres.

Considérons ici 100 variables aléatoires. On tire de chacune d'elles 1 échantillon de taille 200 (c'est-à-dire 200 individus). On se retrouve donc avec 100 échantillons de 200 individus chacun.

# Nous allons considérer 100 variables aléatoires,

# chacune avec 200 échantillons.

matrice\_aleatoire = np.random.rand(100,200)

sommes = np.sum(matrice\_aleatoire,0)

Le deuxième argument de la fonction sum de Numpy dit à la fonction dans quel sens effectuer la somme. S'il vaut 0, la somme sera effectuée le long des lignes, et le résultat sera un vecteur avec autant d'éléments que de colonnes dans la matrice. S'il vaut 1, la somme sera effectuée le long des colonnes, et le résultat sera un vecteur avec autant d'éléments que de lignes dans la matrice. Si l'argument est absent, la fonction renverra simplement un nombre représentant la somme de tous les éléments dans la matrice. Nous pouvons inspecter la taille d'une variable de Numpy grâce à la propriété shape.

print("La taille de la variable sommes est {}.".format(sommes.shape))

La taille de la variable sommes est (200,).

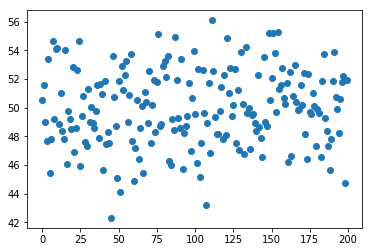
Remarquez que la taille est un peu étrange. C'est un  tuple  avec deux membres, mais le deuxième est vide, au lieu de valoir 1 (après tout nous avons une colonne de nombres). C'est une particularité des tableaux unidimensionnels de Numpy. Grâce à ceci, ce tableau peut être vu, en fonction des opérations, soit comme une ligne, soit comme une colonne. Pour certaines opérations d'algèbre linéaire, il sera nécessaire de convertir ces vecteurs soit en tableaux (1,n), soit en tableaux (n,1). Cette opération se fait grâce à la fonction  reshape.

En pratique, vous passerez certainement beaucoup de temps à vérifier les tailles des tableaux que vous manipulez grâce à leur propriété  shape. La raison en est simple. Pour les opérations d'algèbre linéaire, il y a souvent des contraintes sur la taille des matrices que vous serez amenés à manipuler. Si à un moment vous rencontrez un message d'erreur de la forme :  ValueError: shapes not aligned, pensez à vérifier les tailles de vos matrices.

### Visualisation

Ça y est. Dans la variable sommes , nous avons un échantillon de taille 200, qui est la réalisation  d'une nouvelle variable aléatoire. Cette nouvelle variable aléatoire est la somme des 100 variables aléatoires identiquement distribuées que nous avions au départ. Si nous plaçons ces points sur un graphique, que va-t-on voir?

plot = plt.scatter(range(200), sommes)



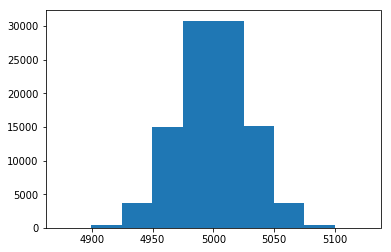
Un échantillon d'une variable aléatoire gaussienne

Les points semblent se concentrer autour de la valeur 50. Mais il n'est pas évident dans ce graphique de bien comprendre leur comportement. Nous devons certainement changer d'outil de visualisation. Nous voulons voir la fréquence à laquelle chaque valeur est générée par notre variable aléatoire. Le bon outil est un ...

### Les histogrammes

Un histogramme est très utile pour étudier une distribution. Et grâce à Matplotlib, nous pouvons en générer un facilement.

plot = plt.hist(sommes)



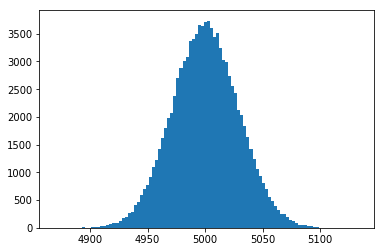
L'histogramme des sommes

Nous commençons à deviner une courbe qui ressemble beaucoup à notre distribution favorite (la distribution gaussienne). Et si nous faisions la même chose avec beaucoup plus que 100 variables, et avec des échantillons beaucoup plus grands que 200 ? Attention, cela risque de prendre quelques secondes de calcul pour votre ordinateur !

matrice\_aleatoire = np.random.rand(10000,100000)

sommes = np.sum(matrice\_aleatoire,0)

plot = plt.hist(sommes, bins=100)



L'histogramme d'un plus grand échantillon

C'est beaucoup mieux ! Remarquez l'argument bins de la fonction hist. Il dicte à la fonction dans combien de classes répartir les valeurs. C'est essentiel quand nous avons affaire à des données sous forme de nombres réels.

Pour une variable aléatoire X continue (par exemple si X prend des valeurs réelles), la probabilité . Pour pouvoir raisonner sur ces variables, nous considérons les intervalles de valeurs, c'est à dire . Pour savoir combien d'intervalles créer, nous utilisons l'argument  bins.

### Calculer les propriétés d'un échantillon

Nous aimerions connaître de façon empirique les propriétés de la distribution que nous avons générée. Et heureusement, nous avons Numpy.

print("La moyenne empirique de notre distribution est {}."

.format(np.mean(sommes)))

print("La moyenne empirique de la variable généré par la fonction rand est {}."

.format(np.mean(np.random.rand(100000))))

print("La variance empirique de notre distribution est {}."

.format(np.var(sommes)))

print("La variance empirique de la variable généré par la fonction rand est {}."

.format(np.var(np.random.rand(100000))))

La moyenne empirique de notre distribution est 5000.20455456666.

La moyenne empirique de la variable généré par la fonction rand est 0.5010223279561243.

La variance empirique de notre distribution est 830.6854133759763.

La variance empirique de la variable généré par la fonction rand est 0.0832842324179482.

La moyenne de notre distribution est égale à la somme des moyennes des distributions qui la composent. La variance est la somme des variances des distributions qui composent notre distribution. Attention, ces propriétés ne sont valables que parce que nos distributions sont indépendantes.

## Entraînez-vous en simulant le problème de Monty Hall avec Numpy

### À vous de jouer !

Pour vous entraîner, réalisez cet exercice étape par étape. Une fois terminé, vous pouvez comparer votre travail avec les pistes que je vous propose.

#### Contexte

Pour cet exercice, je vous demande simplement de refaire ce que nous avons fait dans le chapitre sur le problème de Monty Hall, mais en utilisant cette fois Numpy.  Vous allez donc simuler le problème de Monty Hall, et visualiser les résultats de vos simulations. Vous devrez assembler les deux fonctions play\_game et play au sein d'une même fonction. Cette fonction devra générer un tableau Numpy des gains du joueur, et non pas une list.

Vous ne devrez pas faire appel à la librairie random de Python (mais vous utiliserez la librairie random de Numpy).

Votre code ne doit pas comporter de boucle (pas de for) dans la partie calculatoire. Pour cela, vous devrez sûrement vous creuser un peu la tête. N'oubliez pas que Numpy peut très rapidement comparer les éléments de deux tableaux un à un.

Vous devrez intégrer cette fonction dans un module Python. Votre notebook ne doit comporter aucune définition de fonction.

Votre notebook doit s'exécuter et générer les mêmes graphiques que ceux du cours.

#### Consigne

1. Un fichier .py doit accompagner le notebook, avec à l'intérieur la définition d'une fonction.
2. Le notebook doit utiliser la fonction du module pour simuler le jeu.
3. La fonction ou le notebook ne doivent pas importer le module random (mais numpy.random), ou tout au moins ne pas l'utiliser.
4. Les graphiques du chapitre doivent être reproduits (attention, nous faisons appels à des variables aléatoires, donc les graphiques ne seront pas forcément identiques).
5. Le notebook doit être bien segmenté (tout le code ne doit pas être dans une seule cellule).
6. Toutes les cellules doivent s'exécuter sans erreur (il faut les exécuter dans l'ordre !).

### Vérifiez votre travail

Alors, vous êtes allé au bout ? Suivez le guide pour vérifier votre travail !

[Voici](https://s3-eu-west-1.amazonaws.com/course.oc-static.com/courses/4452741/P2_fr_V2.pdf) un exemple pour vous permettre de vérifier votre travail !

## Plongez en détail dans la librairie NumPy

Dans ce chapitre, nous allons voir des techniques pour charger, stocker et manipuler efficacement les données. Elles peuvent venir de sources très variées, mais nous pouvons toujours les considérer comme des tableaux de nombres. Par exemple, une image peut être considérée comme un tableau de deux dimensions (une matrice) où chaque nombre représente l'intensité lumineuse d'un pixel. Pour cette raison, il est fondamental de pouvoir manipuler efficacement ces tableaux. Dans ce chapitre, nous allons voir un outil pour manipuler ces tableaux : Numpy.

NumPy (diminutif de Numerical Python) fournit une interface pour stocker et effectuer des opérations sur les données. D'une certaine manière, les tableaux Numpy sont comme les listes en Python, mais Numpy permet de rendre les opérations beaucoup plus efficaces, surtout sur les tableaux de large taille. Les tableaux Numpy sont au cœur de presque tout l'écosystème de data science en Python.

Commençons par importer Numpy.

import numpy as np

### Créez des tableaux Numpy

Contrairement aux listes en Python, les tableaux Numpy ne peuvent contenir des membres que d'un seul type. Ce type est automatiquement déduit au moment de la création du tableau, et a un impact sur les opérations qui y seront appliquées. On peut aussi spécifier le type manuellement. Nous allons voir des exemples pour les deux cas tout de suite. On peut créer des tableaux de différentes façons dans Numpy.

#### Depuis une liste Python

# Tableau d'entiers:

np.array([1, 4, 2, 5, 3])

Si dans la liste de départ, il y a des données de types différents, Numpy essaiera de les convertir toutes au type le plus général. Par exemple, les entiers (int) seront convertis en nombres à virgule flottante (float) :

np.array([3.14, 4, 2, 3])

array([ 3.14, 4. , 2. , 3. ])

Nous pouvons aussi manuellement spécifier un type :

np.array([1, 2, 3, 4], dtype='float32')

Pour plus d'informations sur les types NumPy, consultez [**cette page**](https://docs.scipy.org/doc/numpy/user/basics.types.html) de la documentation officielle.

Contrairement aux listes Python, les tableaux Numpy peuvent être explicitement multi-dimensionnels. C'est-à-dire que le tableau multi-dimensionnel (par exemple un tableau de nombres avec des lignes et des colonnes) est reconnu par NumPy. Mais en Python natif, on représente un tableau multidimensionnel par une liste de listes, car au final, un tableau à 2 entrées (lignes et colonnes), ce n'est rien d'autre qu'une liste de lignes, et une ligne est une liste de nombres !

# Une liste de listes est transformée en un tableau multi-dimensionnel

np.array([range(i, i + 3) for i in [2, 4, 6]])

Nous avons utilisé une compréhension de liste pour générer la liste qui sert à créer le tableau Numpy. Rappelez-vous de la syntaxe  [f(x) for x in another\_list]  qui crée une liste en appliquant la fonction  f  à chaque membre de  another\_list.

#### Créer les tableaux directement

Il est souvent plus efficace, surtout pour les tableaux larges, de les créer directement. Numpy contient plusieurs fonctions pour cette tâche.

# Un tableau de longueur 10, rempli d'entiers qui valent 0

np.zeros(10, dtype=int)

# Un tableau de taille 3x5 rempli de nombres à virgule flottante de valeur 1

np.ones((3, 5), dtype=float)

# Un tableau 3x5 rempli de 3,14

np.full((3, 5), 3.14)

# Un tableau rempli d'une séquence linéaire

# commençant à 0 et qui se termine à 20, avec un pas de 2

np.arange(0, 20, 2)

# Un tableau de 5 valeurs, espacées uniformément entre 0 et 1

np.linspace(0, 1, 5)

# Celle-ci vous la conaissez déjà! Essayez aussi "randint" et "normal"

np.random.random((3, 3))

# La matrice identité de taille 3x3

# (matrice identité : https://fr.wikipedia.org/wiki/Matrice\_identit%C3%A9)

np.eye(3)

### Les propriétés des tableaux

Chaque tableau Numpy a des propriétés qui se révèlent souvent utiles.

np.random.seed(0)

x1 = np.random.randint(10, size=6) # Tableau de dimension 1

print("nombre de dimensions de x1: ", x1.ndim)

print("forme de x1: ", x1.shape)

print("taille de x1: ", x1.size)

print("type de x1: ", x1.dtype)

Essayez ces fonctions sur un tableau multidimensionnel.

### Indexation et Slicing

Nous aurons souvent besoin d'accéder à un ou plusieurs éléments contigus d'un tableau. Heureusement, avec Numpy, c'est chose aisée.

#### Accéder à un seul élément

print(x1)

# Pour accéder au premier élément

print(x1[0])

# Pour accéder au dernier élément

print(x1[-1])

x2 = np.random.randint(10, size=(3, 4)) # Tableau de dimension 2

print(x2[0,1])

# On peut aussi modifier les valeurs

x1[1] = "1000"

print(x1)

# Attention au type

x1[1] = 3.14

print(x1)

#### Accéder à plusieurs éléments

De la même façon que nous pouvons indexer des éléments grâce à [], nous pouvons accéder à un ensemble d'éléments en combinant [] et :. La syntaxe suit une règle simple :  x[début:fin:pas].

Le début peut être omis si on veut commencer au début de la liste (c'est à dire si début = 0). La fin peut être omise si on veut aller jusqu'au bout de la liste (c'est à dire fin = -1 ou fin = len(liste)). Le pas, ainsi que le dernier :, peuvent être omis si le pas est de 1 (-1 si la fin est inférieure au début).

print(x1[:5]) # Les cinq premiers éléments

print(x1[5:]) # Les éléments à partir de l'index 5

print(x1[::2]) # Un élément sur deux

Si le pas est négatif, le début et la fin du slice sont inversés. On peut utiliser cette propriété pour inverser un tableau.

x1[::-1]

On peut accéder de la même façon aux éléments d'un tableau multi-dimensionnel. Par exemple, on a souvent besoin d'accéder à une ligne ou une colonne d'une matrice.

print(x2)

x2[0,:] # La première ligne

#### Concaténation

On peut concaténer deux ou plusieurs tableaux.

x = np.array([1, 2, 3])

y = np.array([3, 2, 1])

np.concatenate([x, y])

Si les tableaux sont de dimension > 1, on peut utiliser soit vstack (vertical) ou hstack.

x = np.array([1, 2, 3])

grid = np.array([[9, 8, 7],

[6, 5, 4]])

np.vstack([x, grid])

### Opérations sur les tableaux Numpy

Jusqu'à maintenant dans ce chapitre, nous avons vu des choses très basiques sur les tableaux Numpy. A partir d'ici, nous allons voir ce qui rend Numpy vraiment indispensable.

#### Les boucles peuvent être lentes en Python

L'implémentation de référence de Python, encore appelée CPython, est très flexible, mais cette flexibilité l'empêche d'utiliser toutes les optimisations possibles. Par exemple, observez le temps d'exécution de ce morceau de code.

def calcul\_inverse(values):

output = np.empty(len(values))

for i in range(len(values)):

output[i] = 1.0 / values[i]

return output

values = np.random.randint(1, 10, size=5)

print(calcul\_inverse(values))

tableau\_large = np.random.randint(1, 100, size=1000000)

# Ceci est une facilité des notebooks jupyter pour

# mesurer le temps d'exécution d'une instruction

%timeit calcul\_inverse(tableau\_large)

Il faut plusieurs secondes pour accomplir un million d'opérations. Sachant que les processeurs actuels sont capables d'exécuter des **milliards** d'opérations par seconde, cette durée peut apparaître absurde. Ce délai est dû à toutes les opérations annexes que doit accomplir l'interprète, comme les appels de fonction et vérifications de type.

Dans beaucoup de cas, Numpy fournit une interface pour ces opérations qui n'implique que des données du même type. Cette interface utilise une implémentation en langage C, qui peut faire appel à toutes les fonctionnalités des processeurs modernes. Par exemple, on peut calculer les inverses de tous les éléments d'un tableau Numpy comme ceci.

%timeit (1.0 / tableau\_large)

L'opération a pris presque 1000 fois moins de temps sur ma machine.

À chaque fois que vous vous trouvez en train d'utiliser une boucle pour effectuer une opération en Python, demandez-vous si cette opération ne peut pas s'accomplir grâce à Numpy sans boucle.

Nous allons maintenant faire un tour d'horizon de ces opérations.

#### Les fonctions universelles

# Il y a tout d'abord des opération mathématiques simples

x = np.arange(4)

print("x =", x)

print("x + 5 =", x + 5)

print("x - 5 =", x - 5)

print("x \* 2 =", x \* 2)

print("x / 2 =", x / 2)

print("x // 2 =", x // 2) # Division avec arronid

Vous pouvez aussi appeler des fonctions sur les tableaux Numpy, et même sur les listes Python.

x = [-2, -1, 1, 2]

print("La valeur absolue: ", np.abs(x))

print("Exponentielle: ", np.exp(x))

print("Logarithme: ", np.log(np.abs(x)))

Il y a plein d'autres fonctions, que je vous invite à découvrir dans la [**documentation de Numpy**](https://docs.scipy.org/doc/numpy-1.13.0/reference/ufuncs.html). La leçon à retenir ici est simple : évitez tant que possible les boucles.

#### Opérations Booléennes

Vous pouvez aussi exécuter des opérations booléennes sur vos tableaux. En clair, vous pouvez demander si une certaine condition est vraie pour chaque élément d'un tableau. Par exemple :

x = np.random.rand(3,3)

x > 0.5

Vous pouvez coupler cette capacité avec la fonction  np.where, pour retourner parmi tous les éléments d'un tableau **les index** de ceux qui vérifient une certaine propriété :

np.where(x > 0.5)

#### Agrégation

Très souvent, face à de larges quantités de données, la première chose à faire est de calculer des statistiques sur nos données, comme la moyenne ou l'écart type. Numpy a des fonctions pour calculer ces données sur ses tableaux.

L = np.random.random(100)

np.sum(L)

Cette fonction existe aussi en Python, mais la version Numpy est beaucoup plus rapide. De même, Numpy a des équivalents pour min et max. Faites cependant attention aux arguments optionnels de chaque version de ces fonctions. Gardez aussi à l'esprit que seule la version Numpy gère correctement les tableaux multidimensionnels.

%timeit sum(tableau\_large)

%timeit np.sum(tableau\_large)

Les fonctions d'agrégation peuvent optionnellement ne s'appliquer qu'à une dimension d'un tableau multidimensionnel. Par exemple, nous pouvons avoir besoin de la somme des éléments de chaque colonne d'une matrice (nous avons vu un cas d'utilisation dans le chapitre 3 de la première partie de ce cours). Pour cela nous utilisons l'argument optionnel axis.

M = np.random.random((3, 4))

print(M)

# Notez la syntax variable.fonction au lieu de

# np.fonction(variable). Les deux sont possibles si

# la variable est un tableau Numpy.

print("La somme de tous les éléments de M: ", M.sum())

print("Les sommes des colonnes de M: ", M.sum(axis=0))

Parmi les nombreuses fonctions disponibles, notons :  
-  np.std  pour calculer l'écart type  
-  np.argmin  pour trouver l'index de l'élément minimum  
-  np.percentile  pour calculer des statistiques sur les éléments.

### Broadcasting

Le broadcasting désigne un ensemble de règles pour appliquer une opération qui normalement ne s'applique que sur une seule valeur à l'ensemble des membres d'un tableau Numpy. Par exemple, pour les tableaux de même taille, les opérations comme l'addition s'appliquent normalement élément par élément.

a = np.array([0, 1, 2])

b = np.array([5, 5, 5])

a + b

Le broadcasting nous permet d'appliquer ces opérations sur des tableaux de dimensions différentes. Par exemple :

a + 5

C'est comme si Numpy avait converti la valeur 5 en un tableau de taille 3, et ensuite avait additionné ce tableau à celui contenu dans a. Ce n'est qu'une vue d'esprit faite pour comprendre ce qui se passe. Numpy se passe de toutes ces opérations annexes, ce qui rend le tout plus rapide.

Cela marche aussi quand les deux arguments sont des tableaux :

M = np.ones((3, 3))

print("M vaut: \n", M)

print("M+a vaut: \n", M+a)

N'hésitez pas à vous essayer à ces opérations avec des tableaux de dimensions et tailles différentes pour bien prendre en main cet outil. Un dernier exemple cependant : que pensez-vous sera le résultat de l'opération suivante ?

a = np.arange(3)

# La ligne suivante crée une matrice de taille 3x1

# avec trois lignes et une colonne.

b = np.arange(3)[:, np.newaxis]

a+b

### Pour aller plus loin

La librairie Numpy est très avancée et contient énormément de fonctions et de mécanismes. N'hésitez pas à aller sur [le site officiel](https://www.numpy.org/) pour vous renseigner un peu plus sur ces fonctions et des choses que nous n'avons pas évoquées ici, comme les  mask, qui permettent d'accéder à un sous-ensemble des tableaux Numpy composé d'éléments qui vérifient une certaine propriété (par exemple tous les éléments positifs).

## Maîtrisez les possibilités offertes par Matplotlib

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.

Dans ce chapitre, nous allons nous concentrer sur l'utilisation de Matplotlib comme outil de visualisation dans les notebooks Jupyter.

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

%matplotlib inline

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()

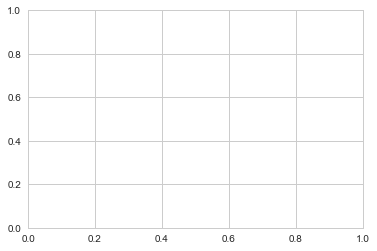


Figure vide

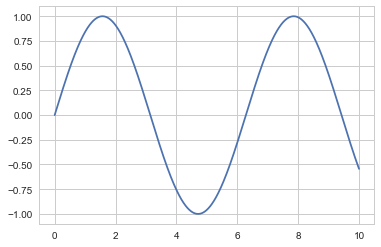
La variable fig correspond à un conteneur qui contient tous les objets (axes, labels, données, etc). Les axes correspondent au carré que l'on voit au-dessus, et qui contiendra par la suite les données du graphe.

fig = plt.figure()

ax = plt.axes()

x = np.linspace(0, 10, 1000)

ax.plot(x, np.sin(x));



Courbe de sinus

On aurait pu simplement taper plt.plot(x, np.sin(x)).

Maintenant, voyons un exemple un peu plus poussé. Pour cet exemple, je vous invite à lire le code et les commentaires, et à observer le résultat sur le graphique généré.

# Chanegr 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')

# Nom court pour la couleur, ligne avec des traits

plt.plot(x, np.sin(x - 1), color='g', linestyle='dashed', label='vert')

# 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')

# Couleur spécifié en RGB, avec des points

plt.plot(x, np.sin(x - 3), color='#FF0000', linestyle='dotted', label='rouge')

# Les limites des axes, essayez aussi les arguments 'tight' et 'equal'

# pour voir leur effet

plt.axis([-1, 11, -1.5, 1.5]);

# Les labels

plt.title("Un exemple de graphe")

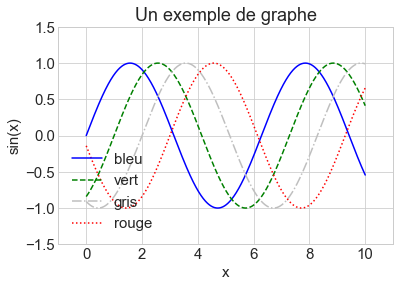
# La légende est générée à partir de l'argument label de la fonctio

# plot. L'argument loc spécifie le placement de la légende

plt.legend(loc='lower left');

# Titres des axes

ax = ax.set(xlabel='x', ylabel='sin(x)')



Un graphique plus complexe

### 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

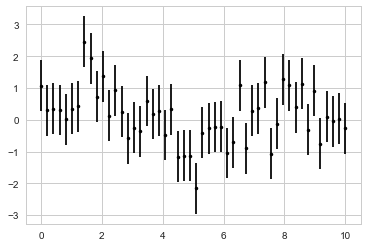
Dans le cas de données discrètes (des points), nous utilisons souvent les barres d'erreur pour représenter, pour chaque point, l'incertitude quant à sa valeur exacte. Souvent la longueur des barres correspond à l'écart type des observations empiriques. C'est chose aisée avec Matplotlib.

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');

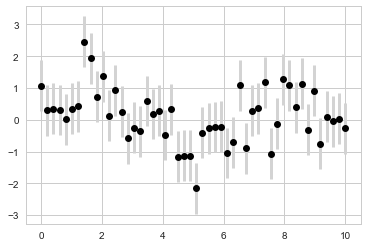


Les barres d'erreur

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 blac**k**) et la forme des marqueurs du graphe.  Errorbar permet aussi de personnaliser encore plus l'apparence du graphe. Je vous invite fortement à consulter sa [documentation](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html).

plt.errorbar(x, y, yerr=dy, fmt='o', color='black',

ecolor='lightgray', elinewidth=3, capsize=0);



Des barres d'erreur d'aspect différent

#### 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. Mais nous allons voir plus tard dans ce chapitre comment le faire plus simplement avec la librairie Seaborn.

### 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])

# Notez la taille de la figure

fig = plt.figure(figsize=(12,8))

for i in range(6):

# On peut ajouter des sous graphes ainsi

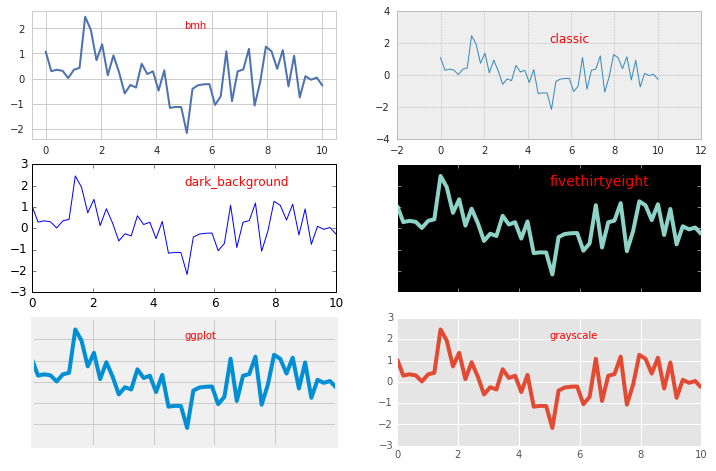
fig.add\_subplot(3,2,i+1)

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')



Exemples de stylesheet

Le premier argument de la fonction add\_subplot est le nombre de lignes de notre tableau de graphes (ici 3). Le deuxième est le nombre de colonnes (ici 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():

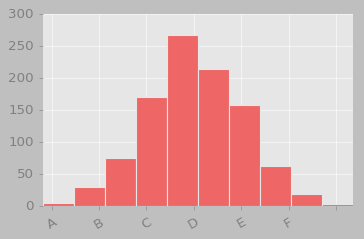
tick.set\_color('gray')

for tick in ax.get\_yticklabels():

tick.set\_color('gray')

# Changer les couleur des barres

ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');



Exemple de graphique personnalisé

### Pour aller plus loin

* Je vous recommande ce [notebook](https://nbviewer.jupyter.org/urls/gist.githubusercontent.com/Jwink3101/e6b57eba3beca4b05ec146d9e38fc839/raw/f486ca3dcad44c33fc4e7ddedc1f83b82c02b492/Matplotlib_Cheatsheet), qui vous servira de référence pour des bouts de code simples et plus avancées de Matplotlib.

## Réalisez de beaux graphiques avec 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 (que nous verrons au chapitre suivant).

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

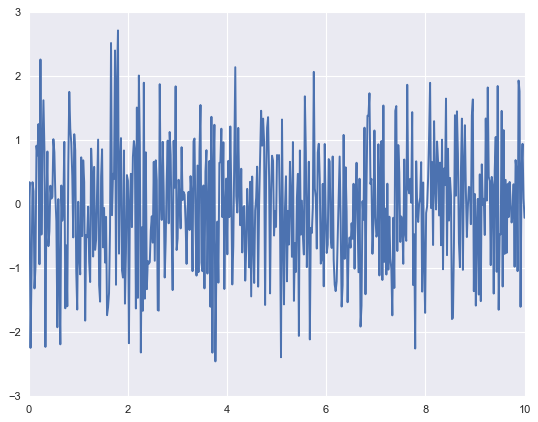
import seaborn as sns

sns.set()

x = np.linspace(0, 10, 500)

y = np.random.randn(500)

plt.plot(x,y)

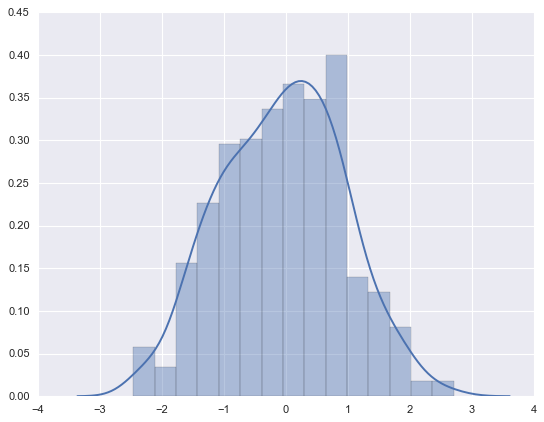


Graphique généré par Seaborn

Qu'en pensez-vous ? Trouvez-vous celui-ci plus agréable à l'œil ?

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);



Estimation d'une distribution

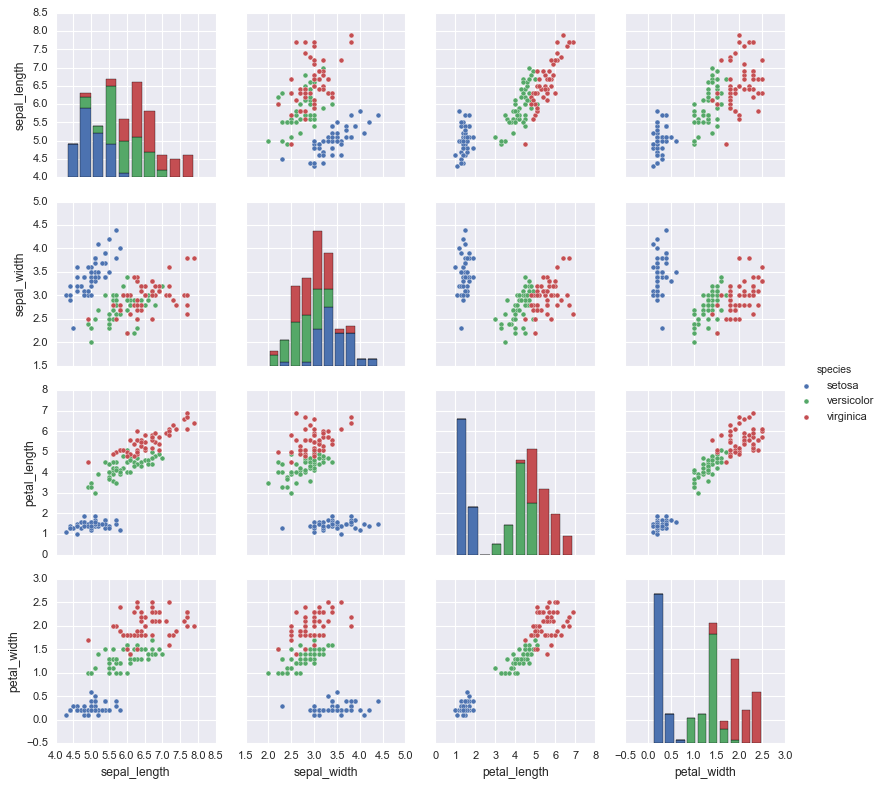
Imaginons que nous voulons travailler sur un ensemble de données provenant du jeu de données "Iris", qui contient des mesures de la longueur et la largeur des sépales et des pétales de trois espèces d'iris. C'est un jeu de données très souvent utilisé pour se faire la main sur des problèmes de machine learning.

iris = sns.load\_dataset("iris")

iris.head()

Pour voir les relations entre ces caractéristiques, on peut faire des graphiques par paire :

sns.pairplot(iris, hue='species', height=2.5);



Corrélations par paire

Prenez le temps d'étudier ce graphique.

Pourquoi les graphes sur la diagonale sont-ils des histogrammes ? Que pouvez-vous dire des relations entre les différentes caractéristiques ? Et sur les différences entre les espèces ?

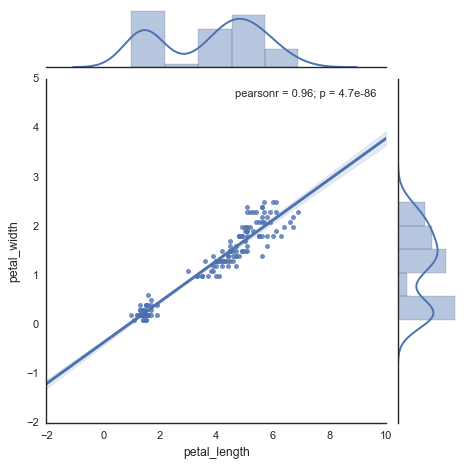
La diagonale est traitée différemment, car tracer une variable en fonction d'elle-même n'aurait aucun intérêt. À la place, sns.pairplot  trace un histogramme des données en fonction de la variable en question pour chaque classe de données.

Ici nous voyons par exemple que les variables petal\_length  et  petal\_width  permettent de bien discriminer l'espèce Setosa des autres. Autant sur les nuages de points de la troisième et la quatrième ligne/colonne, ainsi que sur les histogrammes 3 et 4, les points et les barres en bleues sont distinctes des barres des autres couleurs. Ainsi, si nous voulions créer un algorithme pour détecter les setosa, nous utiliserions certainement petal\_length  et  petal\_width  de manière prédominante.

Nous pouvons aussi voir la distribution jointe de deux caractéristiques :

with sns.axes\_style('white'):

sns.jointplot("petal\_length", "petal\_width", data=iris, kind='reg')



Distribution jointe

Nous n'avons pas le temps ici de plonger en détail dans toutes les possibilités offertes par Seaborn. N'hésitez pas à découvrir ses fonctions par vous-même dans la [documentation](https://seaborn.pydata.org/).

## Entraînez-vous en effectuant une régression linéaire

### À vous de jouer !

Pour vous entraîner, réalisez cet exercice étape par étape. Une fois terminé, vous pouvez comparer votre travail avec les pistes que je vous propose.

#### Contexte

Dans cette activité, vous allez faire appel à tout ce que vous avez étudié dans la deuxième partie du cours. Nous allons nous intéresser à la relation entre la distance qui nous sépare d'une galaxie, et la vitesse à laquelle elle s'éloigne de nous. Cette relation fut découverte pour la première fois par Erwin Hubble en 1929. Son article est disponible [ici](http://www.pnas.org/content/15/3/168.full).

Pour cela, vous aurez besoin du fichier [hubble.csv](https://s3-eu-west-1.amazonaws.com/static.oc-static.com/prod/courses/files/Parcours_data_scientist/decouvrez-les-librairies-python-pour-la-data-science/hubble_data.csv).

Votre tâche consiste à **charger le contenu de ce fichier** grâce à Pandas, **regarder les données** qu'elle contient, et **effectuer une régression linéaire** entre les deux variables distance et velocity.

Pour faire cette régression, vous devez utiliser la bibliothèque scikit-learn. La page de documentation la plus approprié pour cette activité est [ici](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html). Il y a aussi un exemple complet d'une regression linéaire [ici](https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html#sphx-glr-auto-examples-linear-model-plot-ols-py).

Vous devrez aussi visualiser vos résultats, avec des graphiques de qualité (n'hésitez pas à relire le chapitre sur [**matplotlib**](https://openclassrooms.com/courses/4452741/parts/4740942/edit)). Ces graphiques doivent inclure à la fois les points de données, et la courbe obtenue grâce à la régression linéaire.

#### Consigne

1. N'oubliez pas de fournir les coordonnées de la courbe de régression.
2. Votre graphique devrait être présentable :  titres, labels, taille de police appropriée, et qui représente les données et la courbe.

### Vérifiez votre travail

Alors, vous êtes allé au bout ? Suivez le guide pour vérifier votre travail !

[Voici](https://s3-eu-west-1.amazonaws.com/course.oc-static.com/courses/4452741/P3_fr+2.pdf) un exemple pour vous permettre de vérifier votre travail !

## Passez de Numpy à Pandas

Avec Numpy et Matplotlib, la librairie Pandas fait partie des librairies de base pour la data science en Python. Pandas fournit des structures de données puissantes et simples à utiliser, ainsi que les moyens d'opérer rapidement des opérations sur ces structures. Dans ce chapitre, nous verrons l'intérêt de la librairie Pandas, ainsi que les opérations basiques sur l'objet phare de cette librairie, le **dataframe**.

Mais commençons par un peu de réflexion !

### Réfléchissons un peu

Imaginons que nous souhaitions représenter des animaux. Disons... des pandas par exemple !

Nous pouvons caractériser un panda par sa taille ; ou plutôt, par ses tailles : disons par exemple, la taille de ses pattes, la longueur moyenne des poils de sa fourrure, la taille de sa queue, et le diamètre de son ventre.

On peut représenter ce pandas par un tableau numpy :

import numpy as np

un\_panda\_numpy = np.array([100,5,20,80])

un\_panda\_numpy

array([100, 5, 20, 80])

Ici, notre panda a des pattes de 100cm, des poils de 5cm en moyenne, une queue de 20cm et un ventre de 80cm de diamètre.

Et si je veux représenter plusieurs pandas, je crée une liste de np.array?

C'est une bonne idée. Il est effectivement possible de faire cela :

famille\_panda = [

np.array([100, 5 , 20, 80]), # maman panda

np.array([50 , 2.5, 10, 40]), # bébé panda

np.array([110, 6 , 22, 80]), # papa panda

]

Mais comme nous l'avons vu précédemment, nous pouvons faire mieux car notre liste de 3 pandas, c'est en fait un tableau multidimensionnel, que NumPy gère très bien !

famille\_panda = [

[100, 5 , 20, 80], # maman panda

[50 , 2.5, 10, 40], # bébé panda

[110, 6 , 22, 80], # papa panda

]

famille\_panda\_numpy = np.array(famille\_panda)

famille\_panda\_numpy

array([[ 100. , 5. , 20. , 80. ],

[ 50. , 2.5, 10. , 40. ],

[ 110. , 6. , 22. , 80. ]])

Ici, nous avons créé une liste de pandas, c'est à dire une liste de listes ; des listes imbriquées en fait ! En langage Numpy, on dit que le rang de famille\_panda\_numpy est de 2, car nous avons 2 niveaux d'imbrication.

Quel avantage?

Par exemple, supposons que je veuille obtenir la taille des pattes (situé en position 0 dans chaque liste qui décrit un panda) du panda situé en position 2 dans ma liste (c'est-à-dire papa panda). Numpy offre la possibilité de le faire élégamment :

famille\_panda\_numpy[2, 0] # taille des pattes de papa panda, à la manière numpy

110.0

Et si je veux connaître les tailles des pattes de toute ma famille panda ?

Facile ! Il suffit de supprimer le 2 (qui correspondait au papa panda), et de le remplacer par le caractère:, signifiant que je veux TOUS les pandas !

famille\_panda\_numpy[:, 0]

array([ 100., 50., 110.])

Voilà ! La taille des pattes de la maman est de 100cm, celle du bébé 50cm, et celle du papa 110cm.

C'est assez pratique certes, mais écrire famille\_panda\_numpy[:, 0] quand on veut connaître la taille des pattes des pandas, ce n'est pas très explicite. J'aimerais pouvoir spécifier quelque-part que ce  0  correspond à la taille des pattes !

Cela nous amène à la librairie **Pandas** !

### La librairie Pandas

#### Tout est une histoire de tableau !

Au fait, la manière dont j'ai écrit la variable famille\_panda :

famille\_panda = [

[100, 5 , 20, 80],

[50 , 2.5, 10, 40],

[110, 6 , 22, 80],

]

... ça ressemble un peu à un tableau avec des lignes et des colonnes non ? Vous ne trouvez pas que c'est très similaire à cela ?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **pattes** | **poil** | **queue** | **ventre** |
| **maman\_panda** | 100 | 5 | 20 | 80 |
| **bébé\_panda** | 50 | 2.5 | 10 | 40 |
| **papa\_panda** | 110 | 6 | 22 | 80 |

(La bonne réponse est : "Oui, c'est très similaire")

Dans ce cas, pourquoi ne pas utiliser une librairie Python qui manipulerait des tableaux comme celui-ci ? Cette librairie existe, et elle s'appelle Pandas.

C'est parti, représentons nos pandas avec Pandas !

import pandas as pd

famille\_panda\_df = pd.DataFrame(famille\_panda)

famille\_panda\_df

0 1 2 3

0 100 5.0 20 80

1 50 2.5 10 40

2 110 6.0 22 80

L'objet qui permet de représenter des "tableaux" est l'objet DataFrame. Pour instancier un tel objet (et pour lui donner de la donnée), on lui transmet une liste de rang 2, c'est à dire une liste de listes.

Nous pouvons même faire mieux, en indiquant les noms de colonnes et les noms des lignes. Et encore mieux : la librairie Pandas se base en grande partie sur la librairie Numpy dans son fonctionnement interne. Comme pandas connaît intimement Numpy, on peut très bien transmettre à l'objet DataFrame de la donnée au format ndarray :

famille\_panda\_df = pd.DataFrame(famille\_panda\_numpy,

index = ['maman', 'bebe', 'papa'],

columns = ['pattes', 'poil', 'queue', 'ventre'])

famille\_panda\_df

pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

bebe 50.0 2.5 10.0 40.0

papa 110.0 6.0 22.0 80.0

Le nom des lignes est appelé index. Un index peut être une chaîne de caractères (un label) ou un nombre entier. Quand aucun index n'est spécifié à la création du dataframe, il est initialisé par défaut avec une suite continue d'entiers commençant par 0.

Vous allez encore me demander quel est l'intérêt de cette librairie n'est-ce pas ?

En fait, l'objet DataFrame est très similaire à certains concepts que l'on trouve en dehors du cadre du langage Python. Il est similaire :

* aux tables des bases de données relationnelles (type MySQL, PostgreSQL, etc.) que l'on manipule grâce au [langage SQL](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql)
* à l'objet dataframe sur lequel se base tout le [langage R](https://openclassrooms.com/courses/initiez-vous-au-langage-de-statistiques-r), langage destiné aux statisticiens et aux [Data Analysts](https://openclassrooms.com/paths/data-analyst)

Du coup, si vous connaissez déjà le langage SQL ou le langage R, vous aurez beaucoup de facilité à utiliser le DataFrame de Pandas !

#### Quelques fonctionnalités des DataFrames

Voici quelques petites fonctionnalités des Dataframes.

Tout d'abord, accédons à la colonne ventre de notre table. Il y a deux syntaxes possibles, qui renvoient exactement le même résultat :

famille\_panda\_df.ventre

famille\_panda\_df["ventre"]

maman 80.0

bebe 40.0

papa 80.0

Name: ventre, dtype: float64

L'objet que renvoie famille\_panda\_df["ventre"] est de type pandas.Series. Pour obtenir les valeurs de la colonne ventre au format numpy, il faut saisir famille\_panda\_df["ventre"].values

 Parcourrons maintenant tous les pandas un à un, grâce à la méthode iterrows qui renvoie (à chaque itération de la boucle for) un tuple dont le premier élément est l'index de la ligne, et le second le contenu de la ligne en question :

for ind\_ligne, contenu\_ligne in famille\_panda\_df.iterrows():

print("Voici le panda %s :" % ind\_ligne)

print(contenu\_ligne)

print("--------------------")

Voici le panda maman :

pattes 100.0

poil 5.0

queue 20.0

ventre 80.0

Name: maman, dtype: float64

--------------------

Voici le panda bebe :

pattes 50.0

poil 2.5

queue 10.0

ventre 40.0

Name: bebe, dtype: float64

--------------------

Voici le panda papa :

pattes 110.0

poil 6.0

queue 22.0

ventre 80.0

Name: papa, dtype: float64

--------------------

Accédons maintenant au papa panda : d'abord par sa position (position 2), puis par son nom "papa". Le résultat retourné est exactement le même dans les 2 cas.

famille\_panda\_df.iloc[2] # Avec iloc(), indexation positionnelle

famille\_panda\_df.loc["papa"] # Avec loc(), indexation par label

pattes 110.0

poil 6.0

queue 22.0

ventre 80.0

Name: papa, dtype: float64

Déterminons les pandas dont le diamètre du ventre est de 80cm :

famille\_panda\_df["ventre"] == 80

maman True

bebe False

papa True

Name: ventre, dtype: bool

Ici, on teste **chaque** élément de la colonne "ventre" en demandant s'il est égal à 80. On obtient la réponse suivante : "maman" : True, "bebe" : False, "papa" : True.

Le résultat de cette opération est très pratique pour filtrer des lignes ! Par exemple, pour sélectionner **uniquement** les pandas dont le ventre est de 80cm, il suffit d'intégrer ce précédent résultat en tant que masque, comme ceci :

masque = famille\_panda\_df["ventre"] == 80

pandas\_80 = famille\_panda\_df[masque]

# On écrit plus souvent cela de cette manière :

# pandas\_80 = famille\_panda\_df[famille\_panda\_df["ventre"] == 80]

pandas\_80

pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

papa 110.0 6.0 22.0 80.0

Qu'est-ce qu'un masque ?

Lorsque vous mettez un masque sur votre visage, celui-ci cache certaines parties de votre tête, mais en laisse visible d'autres : vos yeux, votre nez et votre bouche. Ici, c'est un peu la même chose : comme nous souhaitons ne garder que certaines lignes du dataframe, et en cacher d'autres, nous appliquons un masque. Un masque est ici une liste de variables booléennes (TrueouFalse) dans laquelle chaque élément est associé à une ligne du dataframe. Si cet élément estTrue, cela signfie que nous souhaitons garder la ligne en question. Si nous souhaitons ne pas garder la ligne, cet élément estFalse.

Pour inverser le masque, il suffit d'utiliser l'opérateur~, et nous sélectionnons les pandas qui n'ont pas un ventre de 80cm :

famille\_panda\_df[~masque]

pattes poil queue ventre

bebe 50.0 2.5 10.0 40.0

Maintenant, ajoutons des lignes à notre dataframe. Il y a plusieurs méthodes pour cela, mais voyons ici la plus simple : assemblons ensemble deux dataframes.

quelques\_pandas = pd.DataFrame([[105,4,19,80],[100,5,20,80]], # deux nouveaux pandas

columns = famille\_panda\_df.columns)

# même colonnes que famille\_panda\_df

tous\_les\_pandas = famille\_panda\_df.append(quelques\_pandas)

tous\_les\_pandas

pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

bebe 50.0 2.5 10.0 40.0

papa 110.0 6.0 22.0 80.0

0 105.0 4.0 19.0 80.0

1 100.0 5.0 20.0 80.0

Dans le dataframe tous\_les\_pandas, il y a des doublons. En effet, le premier panda (maman) et le dernier panda (dont l'index est 1) ont exactement les mêmes mesures. Si nous souhaitions dédoublonner, nous ferions ceci :

tous\_les\_pandas.drop\_duplicates()

  pattes poil queue ventre

maman 100.0 5.0 20.0 80.0

bebe 50.0 2.5 10.0 40.0

papa 110.0 6.0 22.0 80.0

0 105.0 4.0 19.0 80.0

Le résultat de cette ligne ne modifie pas le dataframe  tous\_les\_pandas  . Il ne fait que renvoyer un autre dataframe. Dans l'état actuel, ce nouveau dataframe est juste affiché et n'est pas enregistré en mémoire : il sera donc perdu. Pour le garder, on peut soit l'enregistrer dans une nouvelle variable, ...

pandas\_uniques = tous\_les\_pandas.drop\_duplicates()

 ... soit remplacer la variable tous\_les\_pandas par ce nouveau dataframe dédoublonné :

tous\_les\_pandas = tous\_les\_pandas.drop\_duplicates()

Allez, encore un petit effort ! Voici quelques autres fonctionnalités en vrac, que nous utiliserons dans le projet :

# accéder aux noms des colonnes

famille\_panda\_df.columns

# créer une nouvelle colonne, composée de chaînes de caractères

famille\_panda\_df["sexe"] = ["f", "f", "m"]

# la maman et le bébé sont des femelles, le papa est un mâle

# obtenir le nombre de lignes

len(famille\_panda\_df)

# obtenir les valeurs distinctes d'une colonne :

# pour la colonne ventre, il y a deux valeurs distinctes : 40 et 80

famille\_panda\_df.ventre.unique()

array([ 80., 40.])

#### Lire un fichier CSV avec Pandas

Un fichier [CSV](https://fr.wikipedia.org/wiki/Comma-separated_values) (comma separated values), c'est un fichier permettant de représenter des données sous forme de tableau. D'ailleurs, si vous utilisez un logiciel de type tableur, il y a fort à parier qu'il puisse lire et exporter au format CSV !

Vous me voyez venir ? La librairie Pandas est justement spécialisée dans la manipulation de tableaux ! Lire un fichier CSV avec Pandas est donc un jeu d'enfant : il ne suffit que d'une ligne pour créer un dataframe à partir d'un CSV :

data = pd.read\_csv("data.csv", sep=";")

Ici, donner data.csv en premier argument à read\_csv suppose que le fichier data.csv se trouve dans le répertoire courant. Le répertoire courant, c'est le répertoire (le dossier) à partir duquel vous avez lancé votre notebook (ou Python le cas échéant). Si vous ne le connaissez pas, vous pouvez saisir ceci

:

import os

os.getcwd()

Pour lire un fichier qui ne se trouve pas dans le répertoire courant, utilisez une syntaxe de cette forme :  "/home/user1/Bureau/data.csv"  . Pour plus de précisions, consultez [**cette page**](https://www.alsacreations.com/astuce/lire/78-quelle-est-la-diffrence-entre-les-chemins-relatifs-et-absolus.html) ou la section "Avant de commencer" de ce [**chapitre**](https://openclassrooms.com/courses/apprenez-a-programmer-en-python/les-fichiers-2#r-232331).

C'est fait ! La variable data contient maintenant un dataframe contenant les données du fichier csv. Petite particularité ici : dans notre fichier CSV, les valeurs sont séparées par le caractère;. Par défaut, pd.read\_csv attend des valeurs séparées par une virgule. Nous sommes donc obligés de spécifier sep=";". Pour en savoir plus sur  read\_csv , faites un petit tour sur la [doc](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html) ;) ! Pour avoir un exemple d'utilisation, visitez le haut de [ce chapitre](https://openclassrooms.com/courses/nettoyez-et-decrivez-votre-jeu-de-donnees/telechargez-les-donnees).

## Manipulez les données contenues dans vos DataFrames

Maintenant que vous savez comment créer un DataFrame, intéressons à d'autres opérations usuelles sur les données. Pour ce faire, je vous propose d'utiliser un DataSet disponible dans la librairie Seaborn ! Le dataset en question comprend des données sur les survivants du naufrage du Titanic !

Dans ce chapitre, nous allons suivre une session de travail "typique".

import numpy as np

import pandas as pd

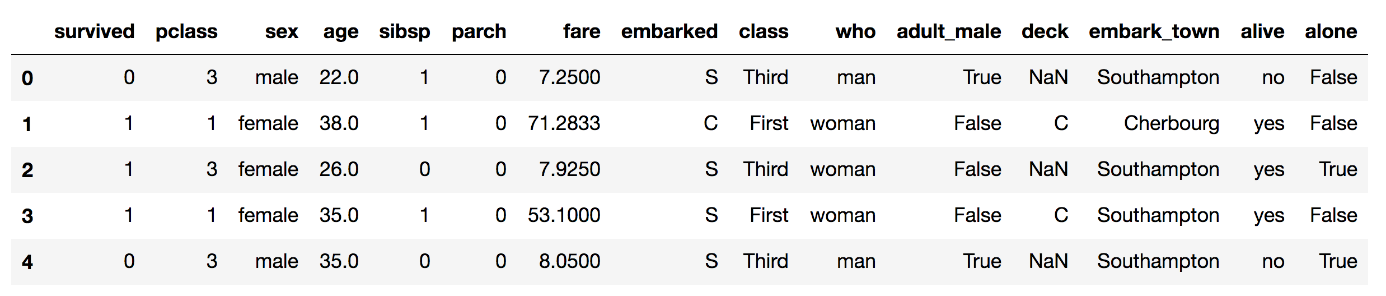
import seaborn as sns

titanic = sns.load\_dataset('titanic')

### Aperçu rapide

La première chose à faire est de jeter un rapide coup d'oeil à nos données.

titanic.head()



Aperçu du dataset Titanic

La fonction  tail  est le pendant de la fonction  head  . Elle affiche les derniers éléments du DataFrame.

Jetons un coup d'oeil à tous les âges. La fonction unique renvoie les valeurs uniques présentes dans une structure de données Pandas.

titanic.age.unique()

array([22. , 38. , 26. , 35. , nan, 54. , 2. , 27. , 14. ,

4. , 58. , 20. , 39. , 55. , 31. , 34. , 15. , 28. ,

8. , 19. , 40. , 66. , 42. , 21. , 18. , 3. , 7. ,

49. , 29. , 65. , 28.5 , 5. , 11. , 45. , 17. , 32. ,

16. , 25. , 0.83, 30. , 33. , 23. , 24. , 46. , 59. ,

71. , 37. , 47. , 14.5 , 70.5 , 32.5 , 12. , 9. , 36.5 ,

51. , 55.5 , 40.5 , 44. , 1. , 61. , 56. , 50. , 36. ,

45.5 , 20.5 , 62. , 41. , 52. , 63. , 23.5 , 0.92, 43. ,

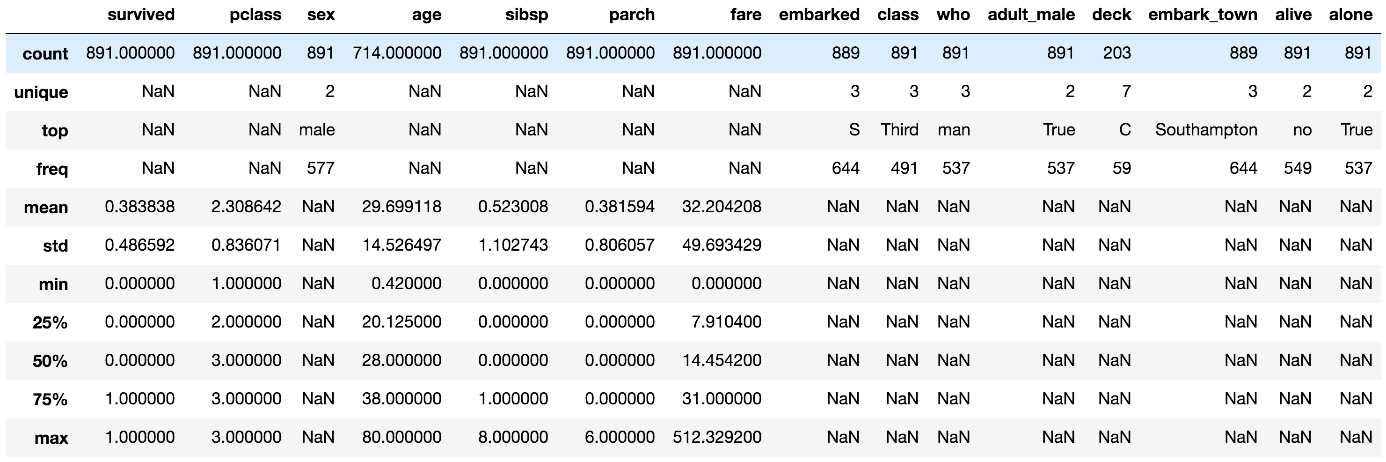
60. , 10. , 64. , 13. , 48. , 0.75, 53. , 57. , 80. ,

70. , 24.5 , 6. , 0.67, 30.5 , 0.42, 34.5 , 74. ])

Remarquez la présence de valeurs inférieures à 1 !

Mentionnons aussi l'excellente fonction describe . Elle donne des statistiques diverses (moyenne, maximum, minimum, etc.) sur les données contenues dans chaque colonne :

titanic.describe(include="all")



Description de titanic

L'argument include="all" sert à inclure les colonnes non-numérique dans l'analyse. Cette fonction nous fournit pleins de données très utiles sur la répartition de nos données (minimum, maximum, moyenne, etc.)

### Données manquantes

Vous aurez remarqué, dans la sortie de la fonction describe, des valeurs  NaN. C'est une valeur définie pour représenter quelque chose qui n'est pas un nombre (Not a Number) alors que son type l'exige. Par exemple, on obtient NaN si on demande à Pandas de calculer la moyenne d'une colonne de texte.

Plus généralement, le résultat de toute opération impliquant une NaN est à son tour un NaN. Par exemple, si une de vos colonnes contient une NaN (parce que la vraie valeur n'est pas connue), le résultat de toutes les opérations arithmétiques qui impliquent cette valeur (comme la moyenne de la colonne) sera NaN, sauf si vous prenez soin de ne pas prendre en compte cette valeur (Pandas le fait, par exemple sur la colonne age, dans notre exemple).

Le traitement à des valeurs manquantes est abordé dans [un autre cours](https://openclassrooms.com/fr/courses/4525266-decrivez-et-nettoyez-votre-jeu-de-donnees/4928121-traitez-les-valeurs-manquantes-les-outliers-et-les-doublons). Nous allons simplement voir deux opérations à appliquer aux NaN.

La première consiste à remplacer les NaN par d'autres valeurs. Cette opération s'effectue grâce à la fonction fillna.  Regardons son application sur la colonne age.

titanic.age.head(10)

0 22.0

1 38.0

2 26.0

3 35.0

4 35.0

5 NaN

6 54.0

7 2.0

8 27.0

9 14.0

Name: age, dtype: float64

titanic.fillna(value={"age": 0}).age.head(10)

Renvoie un DataFrame où toutes les NaN dans la colonne age ont été remplacés par 0.

0 22.0

1 38.0

2 26.0

3 35.0

4 35.0

5 0.0

6 54.0

7 2.0

8 27.0

9 14.0

Name: age, dtype: float64

Nous aurions aussi pu remplir les NaN par les valeurs précédentes :

titanic.fillna(method="pad").age.head(10)

0 22.0

1 38.0

2 26.0

3 35.0

4 35.0

5 35.0

6 54.0

7 2.0

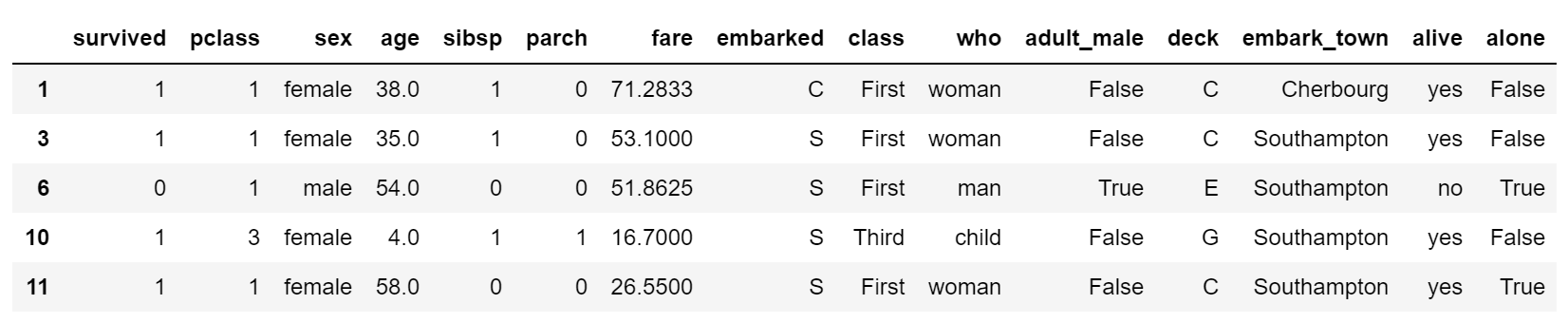
8 27.0

9 14.0

Name: age, dtype: float64

Enfin, la fonction dropna permet de supprimer les axes (colonnes ou lignes) qui contiennent des NaN. Par défaut, elle supprime les lignes concernées :

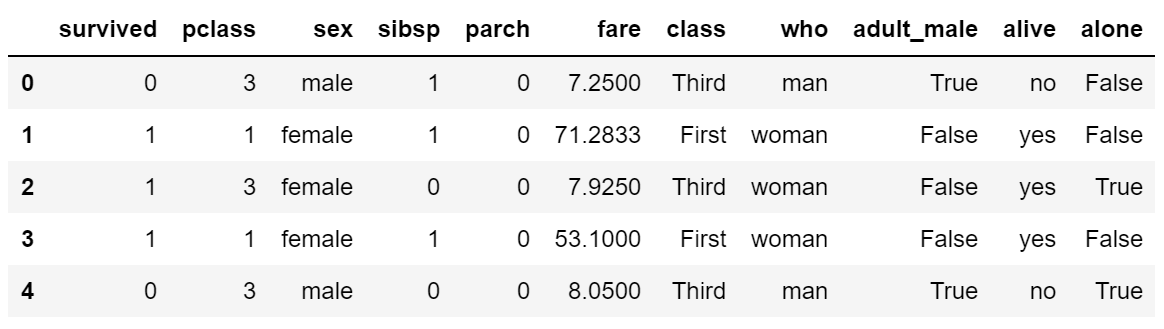
titanic.dropna().head(10)



Fonction DropNa

Mais on peut aussi lui demander de supprimer carrément les colonnes !

titanic.dropna(axis="columns").head()



Supprimer les colonnes avec DropNa

### Renommer une colonne

La fonction rename permet de renommer les colonnes ou les lignes d'un DataFrame. Elle s'utilise de deux façons.

titanic.rename(columns={"sex":"sexe"})

Renomme la colonne "sex" en "sexe". Tandis que :

f = lambda x: x+1

titanic.rename(index=f)

applique la fonction f à tous les index.

### Supprimer des axes

La fonction drop permet de supprimer des axes (colonnes ou lignes) d'un DataFrame. Son utilisation est plutôt simple.

titanic.drop(0)

Supprimera la ligne dont l'index est égal à 0.

titanic.drop(columns=["age"])

Supprime la colonne "age".

Bon nombre de fonctions Pandas, telles que dropna, fillna, drop, etc acceptent un argument inplace.  Si la valeur de cet argument est True, le DataFrame donné en argument est modifié. Sinon, une copie du DataFrame est retournée par la fonction.

### Tableaux croisés dynamiques

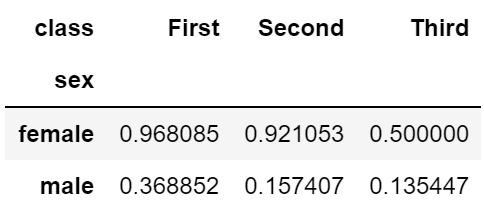
Avant de vous lancer dans l'algèbre relationnelle, mentionnons les tableaux croisés dynamiques. Vous êtes peut-être familier avec ces concepts, par exemple parce que vous les avez utilisés dans des logiciels tableurs.

Ces tableaux, encore appelés tables de pivots (ou pivot table), permettent de synthétiser les données contenues dans un DataFrame. Essayons de voir cela par l'exemple.

Pour voir la répartition des survivants en fonction de leurs sexes et de leur type de billet, nous n'avons besoin que d'une seule ligne:

titanic.pivot\_table('survived', index='sex', columns='class')

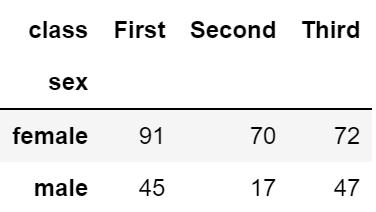
Le résultat est parfaitement compréhensible :



Taux de survie

Par défaut, la fonction pivot\_table groupe les données en fonction des critères que nous spécifions, et agrège les résultats en moyenne. Nous pouvons spécifier d'autres fonctions. Par exemple, si nous voulons savoir quelle est le nombre total de survivants dans chaque cas, nous utiliserons la fonction sum  .

titanic.pivot\_table('survived', index='sex', columns='class', aggfunc="sum")



Le nombre de survivants

Bien sûr, cela ne fonctionne que parce que les auteurs du dataset ont judicieusement choisi de représenter la survie avec des 0 et des 1 !

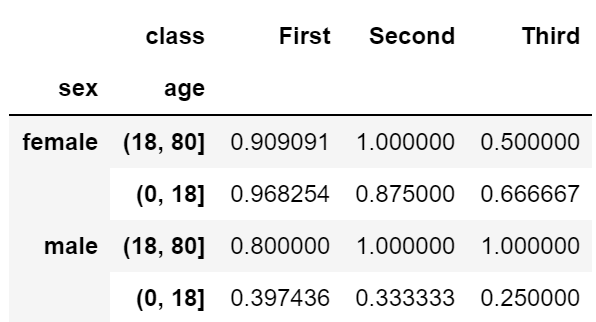
La fonction pivot\_table est très puissante, et permet même de faire des agrégations à plusieurs niveaux. Par exemple, nous pouvons voir l'âge des survivants comme une dimension supplémentaires. Le nombre exact d'années nous intéressant peu, nous regrouperons les âges en deux catégories, grâce à la fonction cut.

titanic.dropna(inplace=True)

age = pd.cut(titanic['age'], [0, 18, 80])

titanic.pivot\_table('survived', ['sex', age], 'class')

Le résultat est un DataFrame multi-indexé:



Agrégation à plusieurs niveaux

Remarquez les deux niveaux d'indexation à gauche.

Nous n'allons pas parler en détail des DataFrame à indexation multiple. Sachez seulement que Pandas est capable de les gérer, et que les fonctions pandas sont capables, en utilisant les bons arguments, de travailler avec ces DataFrames.

Voilà. J'espère que vous avez une meilleure idée des capacités de Pandas. Dans le chapitre suivant, vous allez voir comment effectuer des opérations d'algèbre linéaires (comme dans le langage SQL) sur les DataFrames.

## Effectuez les opérations d'algèbre relationnelle sur les DataFrames

Le cours écrit donne bien plus de détails que la vidéo (notamment une mise en garde concernant les jointures). N'hésitez pas à le consulter.

Dans ce chapitre, nous continuons à investiguer l'objet **dataframe**. Plus précisément, nous allons voir comment effectuer les opérations de base de l'**algèbre relationnelle**.

### L'algèbre relationnelle

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 !

Dans ce domaine, un tableau, on appelle cela **une relation**.

Plus précisément, l'algèbre relationnelle définit des opérations sur ces tableaux. Elles sont regroupables en 5 grandes catégories :

* [La projection et la restriction](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/decouvrez-la-projection-et-la-restriction)
* [Les opérations ensemblistes (union, différence, intersection)](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/decouvrez-les-operateurs-ensemblistes)
* [Le produit cartésien](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/effectuez-un-produit-cartesien)
* [Les jointures](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/liez-des-relations-grace-aux-jointures)
* [L'agrégation](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/noubliez-pas-lagregation)

A priori, pas besoin de les connaître à l'avance, les exemples de ce chapitre sont assez explicites. Cependant pour mieux comprendre, il vous sera peut-être utile de jeter un rapide coup d’œil à la jointure et à l'agrégation.

Si vous avez déjà pratiqué le SQL, alors je vous conseille de lire en parallèle de ce chapitre cette page de la documentation officielle, qui donne la "traduction" du SQL vers pandas ;) : <https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html>

### Les structures de données de Pandas

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 un rappel pour 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)

# 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)

print(data['b'])

# 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)

# Que pensez vous de cette ligne?

print(population['California':'Florida'])

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. Construire un DataFrame est chose aisée :

# A partir d'une Series

df = pd.DataFrame(population, columns=['population'])

print(df)

# 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 partir de plusieurs Series

df = pd.DataFrame({'population': population,

'area': area})

print(df)

# 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)

# 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))

### La projection et la restriction

Commençons par 2 opérations de l'algèbre relationnelle : [la projection et la restriction](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/decouvrez-la-projection-et-la-restriction). Elles sont très simples : la première est une sélection de certaines colonnes, et la seconde une sélection de certaines lignes.

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)

# On peut désigner un élément d'une Series par son index

print(data.loc['b'])

# Ou bien par sa position

print(data.iloc[1])

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)

data.loc[:'Illinois', :'pop']

### L'union

#### L'union grâce à pd.concat

L'une des opérations les plus simples en algèbre relationnelle est l'[union](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/decouvrez-les-operateurs-ensemblistes) de données. Unir deux tableaux, c'est simplement créer un troisième tableau qui contient toutes les lignes du premier et toutes les lignes du second.

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 ?

df1 = make\_df('AB', [1, 2])

df2 = make\_df('AB', [3, 4])

pd.concat([df1, df2])

A B

1 A1 B1

2 A2 B2

3 A3 B3

4 A4 B4

Étudiez bien ce morceau de code. Fait-il ce que vous en attendiez ?

Par défaut, pd.concat assemble ses arguments "verticalement". Pour changer ce comportement, on peut utiliser l'argument axis.

#### Problèmes des index avec l'union

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.

C'est souvent un problème. Pour y pallier, on peut utiliser les index hiérarchiques.

x = make\_df('AB', [0, 1])

y = make\_df('AB', [2, 3])

y.index = x.index # Rend les index identiques

# Nous avons alors des index dupliqués

print(pd.concat([x, y]))

# Nous pouvons spécifier des index hiérarchiques

hdf = pd.concat([x, y], keys=['x', 'y'])

print(hdf)

A B

0 A0 B0

1 A1 B1

0 A2 B2

1 A3 B3

A B

x 0 A0 B0

1 A1 B1

y 0 A2 B2

1 A3 B3

Comment utiliser les index hiérarchiques ?

Pour accéder à un élément d'un objet Pandas avec un index hiérarchique, il suffit de spécifier plusieurs index. Dans notre exemple, essayez par exemple de voir le résultat de   hdf.loc[('x', 1),].

La fonction pd.concat est très puissante. Je vous invite à consulter la documentation de Pandas, en particulier [**cette page**](https://pandas.pydata.org/pandas-docs/stable/merging.html) pour en apprendre plus.

### La jointure

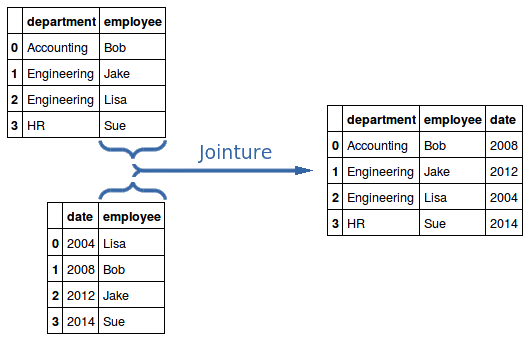
Une autre fonction très utile pour manipuler les Dataframe est pd.merge. Elle effectue une [jointure](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/liez-des-relations-grace-aux-jointures).

Une jointure, c'est assembler les informations d'un tableau A avec celles d'un autre tableau B selon un critère choisi. Ce critère s'appelle la **condition de jointure**. Cette condition est composée d’une ou plusieurs colonnes communes à A et B qui effectuent une correspondance entre les 2 tableaux.

Un petit exemple. Imaginons que nous disposons de deux Dataframes :

* df1  contenant une liste d'employés et le nom des départements dans lequel ils travaillent,
* df2  contenant la même liste d'employés et leurs dates d'entrée dans l'entreprise.

La fonction pd.merge nous permet de transformer ces deux Dataframes en un seul, contenant les deux informations.



Jointure entre deux dataframes selon la colonne "employee"

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)

La fonction pd.merge a automatiquement reconnu que la colonne employee était commune aux deux Dataframes, et l'a utilisée comme condition de jointure.

Mais si ces 2 colonnes avaient eu des noms différents (par exemple employee sur df1 et emp\_name sur df2), alors il aurait fallu écrire ceci :

df3 = pd.merge(df1, df2, left\_on= "employee", right\_on= "emp\_name")

#### La cardinalité

En effectuant une jointure, il est préférable de toujours faire attention à la **cardinalité** de la relation entre A et B afin d'évider certaines erreurs. Un exemple d'une telle erreur est donnée dans la section "Pourquoi les clés sont-elles si importantes ?" de [ce chapitre](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/ne-perdez-pas-de-vue-vos-cles).

Il y a 3 cardinalités différentes :

* **Un-à-un (one-to-one)** : lorsque qu'une ligne quelconque du tableau A correspond à une unique ligne du tableau B, et qu'une ligne de B correspond à une unique ligne du tableau A.
* **Un-à-plusieurs (one-to-many)** : quand une ligne quelconque de A correspond à une ligne de B, mais qu'une ligne de B peut correspondre à plusieurs lignes de A.
* **Plusieurs-à-plusieurs (many-to-many)** : quand une ligne de A peut correspondre à plusieurs lignes de B, et qu'une ligne de B peut correspondre à plusieurs lignes de A.

Un petit exemple à base de pommes ? Faites un tour [**par ici**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/utilisez-les-tables-dassociation) !

La fonction pd.merge ne fait pas la différence entre ces 3 cardinalités : elle s'utilise exactement de la même manière dans les 3 cas.

 Une petite précision destinée à ceux qui ont suivi le cours [**Initiez-vous à l'algèbre relationnelle avec le langage SQL**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql).

Il a été dit dans l'[**un des chapitres**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/utilisez-les-tables-dassociation) qu'en cas de cardinalité **plusieurs-à-plusieurs**, il est préférable d'utiliser une table d'association.

* Si on le fait, on "casse" le **plusieurs-à-plusieurs**, et on se retrouve alors avec 2 jointures de type **un-à-plusieurs** grâce à la table d'association. Les jointures sur des cardinalités **un-à-plusieurs** (et aussi un-à-un) sont celles auxquelles vous avez été habituées dans le cours sur l'algèbre relationnelle. Elles devraient donc ne pas vous causer de soucis !
* Mais si on ne le fait pas, alors on brise le principe énoncé dans [**ce chapitre de mise en garde**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/ne-perdez-pas-de-vue-vos-cles), qui dit "En général, on effectue une jointure d'une clé étrangère d'une table A vers une clé candidate d'une table B" . En effet, on se retrouve à faire une jointure sur une cardinalité **plusieurs-à-plusieurs** ; et dans ce cas, on ne joint sur aucune clé candidate ! La jointure **plusieurs-à-plusieurs** vous paraîtra donc un peu plus bizarre, mais vous devriez vous y faire rapidement ;)

Pour bien comprendre, déterminez dans le code qui suit les clés primaires et étrangères de chacun des dataframes  df1  à  df5  .

##### Cardinalité un-à-un

L'exemple que nous avons pris tout à l'heure correspond à une cardinalité un-à-un. En effet, un employé ne travaille que dans un seul département, et n'a qu'une seule année d'embauche.

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

##### Cardinalité un-à-plusieurs (ou 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.

Ici, c'est une cardinalité un-à-plusieurs. En effet, df3 représente des employés, et df4 représente des chefs. Un employé n'a qu'un seul chef, mais un chef peut diriger plusieurs employés.

df4 = pd.DataFrame({'department': ['Accounting', 'Engineering', 'HR'],

'supervisor': ['Carly', 'Guido', 'Steve']})

pd.merge(df3, df4)

Remarquez que Guido apparaît plusieurs fois dans le résultat.

##### Cardinalité plusieurs-à-plusieurs

Pour continuer avec notre exemple, supposons que nous disposions d'un autre Dataframe contenant les compétences nécessaires pour travailler dans chaque département :

df5 = pd.DataFrame({'department': ['Accounting', 'Accounting',

'Engineering', 'Engineering', 'HR', 'HR'],

'competence': ['math', 'spreadsheets', 'coding', 'linux',

'spreadsheets', 'organization']})

Maintenant, nous souhaitons associer à chaque employé les compétences qu'il doit posséder pour travailler dans son département. Ici, c'est une cardinalité plusieurs-à-plusieurs car un employé a besoin de plusieurs compétences, et une compétence peut être partagée par plusieurs employés.

pd.merge(df1, df5)

#### La jointure externe

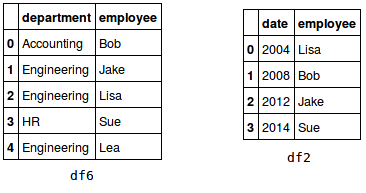
Reprenons df1 et ajoutons-y Lea, employée dans le département Engineering. Appelons ce nouveau dataframe df6. Cependant, nous n'ajoutons pas Lea à df2 :

df6 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue', 'Lea'],

'department': ['Accounting', 'Engineering', 'Engineering', 'HR', 'Engineering']})

df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],

'date': [2004, 2008, 2012, 2014]})

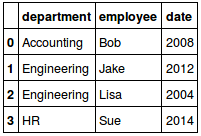


Ajoutons Lea !

Si on écrit cette jointure...

pd.merge(df6, df2)

... alors on obtient ceci :



pd.merge(df6, df2)

Lea a disparu !

Effectivement. Comme Lea est présente dans df6 mais pas dans df2, elle n’apparaît pas dans le résultat de la jointure... Pour spécifier que l'on veut garder tous les éléments de dataframe de gauche (ici df6), il faut alors écrire la ligne suivante :

pd.merge(df6, df2, how="left")

Ainsi, Lea est présente dans la table finale ! Bien entendu, pour garder toutes les lignes de la table de droite (df2), on écrit  how="right"  , et pour garder toutes les lignes à la fois à gauche et à droite, on écrit  how="outer"  . On appelle ces jointures respectivement : **jointure externe à gauche**, **jointure externe à droite** et **jointure externe totale**. Si ce n'est pas très clair, faites un petit tour dans la section "Jointure externe" de ce [chapitre](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/liez-des-relations-grace-aux-jointures#r-4569277).

### Le produit cartésien

Nous pouvons utiliser la jointure pour réaliser une autre opération d'algèbre relationnelle, le [produit cartésien](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/effectuez-un-produit-cartesien).

# 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, on='key')

# Effaçons la colonne key qui n'est plus utile

produit\_cartesien.drop('key',1, inplace=True)

Il est possible de condenser ces 4 lignes en une seule !

pd.merge(df1.assign(key=0), df2.assign(key=0), on='key').drop('key', axis=1)

Avec ce que vous avez vu sur la jointure plusieurs-à-plusieurs, comprenez-vous ce qui se passe ici ?

Le produit cartésien est composé de toutes les associations possibles entre les lignes de df1 et celles de df2. Ici, avec nos données, le produit cartésien n'a cependant pas beaucoup de sens...

### L'agrégation

Passons maintenant à l'[agrégation](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/noubliez-pas-lagregation) !

Comme les tableaux Numpy, nous pouvons facilement effectuer des opérations sur l'ensemble des éléments d'une Series ou un Dataframe, comme par exemple  sum  ou  mean  :

rng = np.random.RandomState(42)

# Une Series avec cinq nombres aléatoires

ser = pd.Series(rng.rand(5))

print(ser.sum())

print(ser.mean())

Pour un Dataframe, ces calculs sont aussi possibles, et ils sont réalisés par colonne :

df = pd.DataFrame({'A': rng.rand(5),

'B': rng.rand(5)})

# Par colonne

print(df.mean())

# Par ligne

print(df.mean(axis='columns'))

Ces fonctions sont des **fonctions d'agrégation**.

Pandas nous permet d'accomplir une agrégation par groupe, semblable à ce qu'on peut obtenir en utilisant le mot clé  [GROUP BY en SQL](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/agregez-vos-donnees-grace-au-group-by).

df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],

'data1': range(6),

'data2': [10,11,10,9,10,10]})

print(df)

Dans Pandas, cette opération se fait en deux étapes. Nous allons d'abord créer un objet de type DataFrameGroupBy  , que nous appelons  gb  :

gb = df.groupby('key')

La (les) colonne(s) données dans groupby s'appelle(nt) l'(les) **attribut(s) de partitionnement**.

Sur cet objet gb, on peut ensuite appliquer les fonctions d'agrégation :

print(gb.sum())

print(gb.mean())

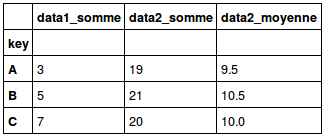
Si on veut par exemple calculer la somme des colonnes data1 et data2, et calculer la moyenne de data2 uniquement, sachez qu'il est possible de sélectionner les colonnes nécessaires sur notre objet gb :

s = gb['data1','data2'].sum()

m = gb['data2',].mean()

groupped = pd.concat([s,m], axis=1)

groupped.columns = ["data1\_somme","data2\_somme","data2\_moyenne"]



Résultat d'une agrégation

Si vous connaissez le SQL, la requête équivalente est :

SELECT sum(data1) as data1\_somme, sum(data2) as data2\_somme, mean(data2) as data2\_moyenne FROM df GROUP BY 'key';

Et voilà !

Comme pour les librairies précédentes, nous n'avons que frôlé toutes les possibilités offertes par Pandas. Je vous encourage vivement à consulter [**la documentation**](https://pandas.pydata.org/pandas-docs/stable/) de cette librairie pour en apprendre plus. Pour aller plus loin, reportez-vous à [**cette ressource**](https://jakevdp.github.io/PythonDataScienceHandbook/03.00-introduction-to-pandas.html).

## Entraînez-vous en générant des cartes de vœux avec Pandas

### À vous de jouer !

Pour vous entraîner, réalisez cet exercice étape par étape. Une fois terminé, vous pouvez comparer votre travail avec les pistes que je vous propose.

#### Contexte

Vous avez une idée de start-up ! Vous souhaitez vendre des cartes de vœux pour les anniversaires... Ca parait simple, mais attention, il y a un twist ! Vous voulez faire un type de cartes de vœux pour le jour de la semaine de naissance du destinataire. Vous aurez ainsi 7 types de cartes différents.

Vous aimeriez prévoir la quantité de cartes à créer. Malheureusement, vous ne savez pas quelle quantité commander pour chaque jour. Mais vous avez accès aux données de naissance aux États-Unis dans ce [fichier](https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv) (eh oui, vous partez directement à l'international !)

Pour mieux comprendre quelles cartes créer, vous allez donc étudier les jours de naissance des habitants des états-unis.

Cette tâche implique un peu de nettoyage de données, et un peu d’algèbre linéaire. Vous aurez certainement besoin d'un peu de traitement avec la fonction  to\_datetime  pour transformer les données en objets  datetime  (et un peu de connaissance en Python pour comprendre ce que c'est).

#### Consigne

1. Vous allez livrer un graphique avec trois courbes qui représentent le nombre de naissances pour chaque jour de la semaine pour les décennies 1960, 1970 et 1980.
2. Cette tâche implique un peu de nettoyage de données, et un peu d’algèbre linéaire. Vous aurez certainement besoin d'un peu de traitement avec la fonction  to\_datetime  pour transformer les données en objets  datetime  (et un peu de connaissance en Python pour comprendre ce que c'est). Pour cet exercice, faites attention à ces points :
   1. Nettoyage de la colonne births.
   2. Conversion des colonnes day, month et year en datatime.
   3. Conversion en jour de semaine.
   4. L'agrégation doit être faite grâce à librairies Pandas.

### Vérifiez votre travail

Alors, vous êtes allé au bout ? Suivez le guide pour vérifier votre travail !

[Voici](https://s3-eu-west-1.amazonaws.com/course.oc-static.com/courses/4452741/P4_fr.pdf) un exemple pour vous permettre de vérifier votre travail !