

Introduction au Langage Python pour la Data Analyse

Angelo.Steffenel@univ-reims.fr

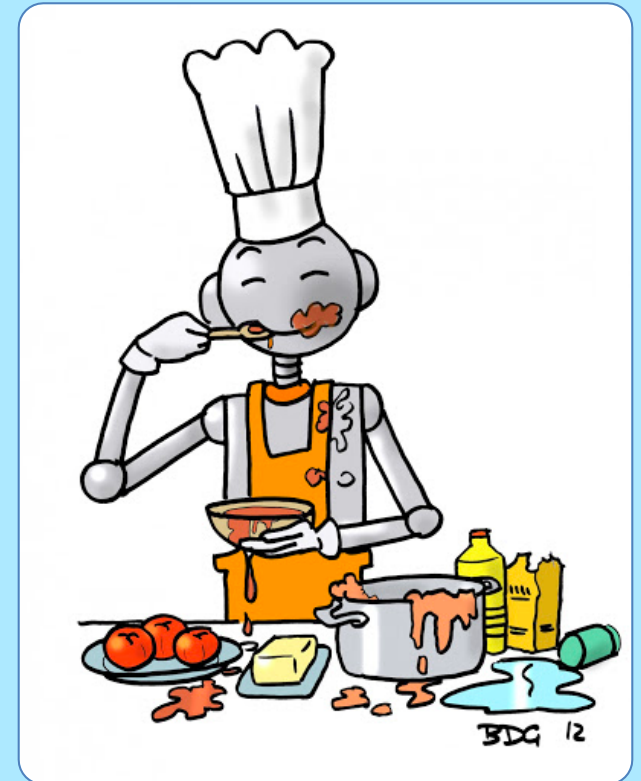
Algorithmes vs Langages de Programmation

- La base d'un programme d'ordinateur est l'algorithme
- Un algorithme n'est qu'une "recette de cuisine"
 - Décrit les ingrédients et les étapes de préparation
- Passer de la recette au gâteau c'est toute une autre histoire



Les Langages de Programmation

- Pour réussir une recette il faut
 - **Des instructions précises** (actions, ordre, etc.)
 - Des instructions **adaptées au cuisinier**
- Un ordinateur n'est rien d'autre qu'un cuisinier très spécialisé
 - Ne connaît que le langage machine (0s et 1s)
- Un langage de programmation permet **d'exprimer nos ordres** pour ensuite les traduire en langage machine



Langages Haut Niveau

- Au fil du temps les langages sont devenus plus proches de notre langue naturelle
- Quelques exemples
 - C et C++
 - Java
 - Javascript
 - Visual Basic
 - Ruby
 - Python

```
1  #!/usr/bin/env python
2  import sys
3  import os
4  import simpleknn
5  from bigfile import BigFile
6
7  if __name__ == "__main__":
8      trainCollection = 'toydata'
9      nimages = 2
10     feature = 'f1'
11     dim = 3
12
13     testCollection = trainCollection
14     testset = testCollection
15
16     featureDir = os.path.join(rootpath, trainCollect:
17     searcher = simpleknn.load_model(os.path.join(fe
```

C'est quoi le langage Python ?

- Python a été créé à la fin des années 1980
 - Son auteur est un hollandais qui a travaillé pour des compagnies telles que Google et Dropbox
- **Langage "compact"** par rapport à d'autres langages populaires (Java, C/C++)
- C'est un langage **simple pour démarrer et très puissant**
 - Dispose de bibliothèques très sophistiquées
 - La gestion de ces bibliothèques est très facile
 - Donc **rapidement adopté par des non-informaticiens**
- Devenu l'un des langages incontournables pour la **data analyse et l'intelligence artificielle**

Trois façons de tester Python

1. **Installer** le langage sur son ordinateur

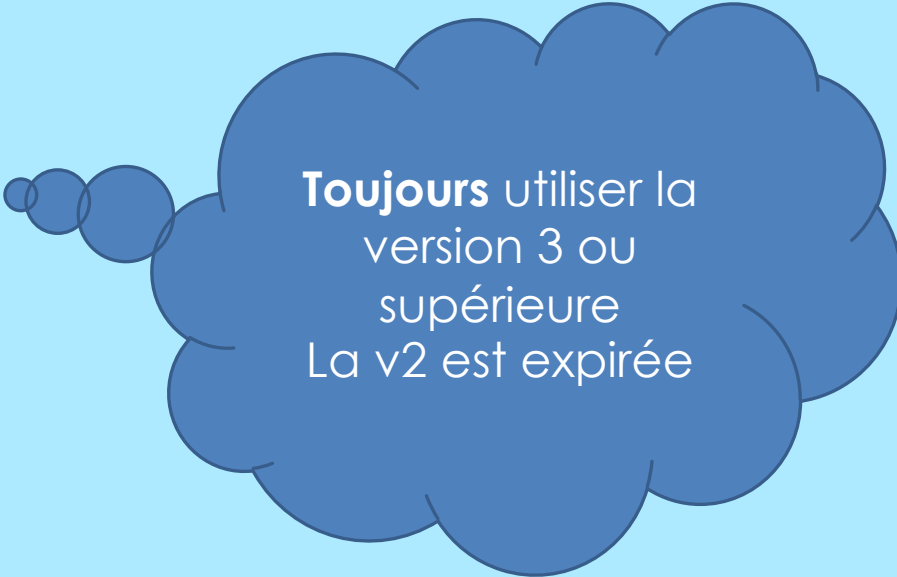
- <https://www.python.org/downloads/>
- Normalement cela inclut l'environnement de développement IDLE
- Certains OS (Mac, Linux) incluent déjà une version de Python

2. Utiliser un **environnement en ligne**

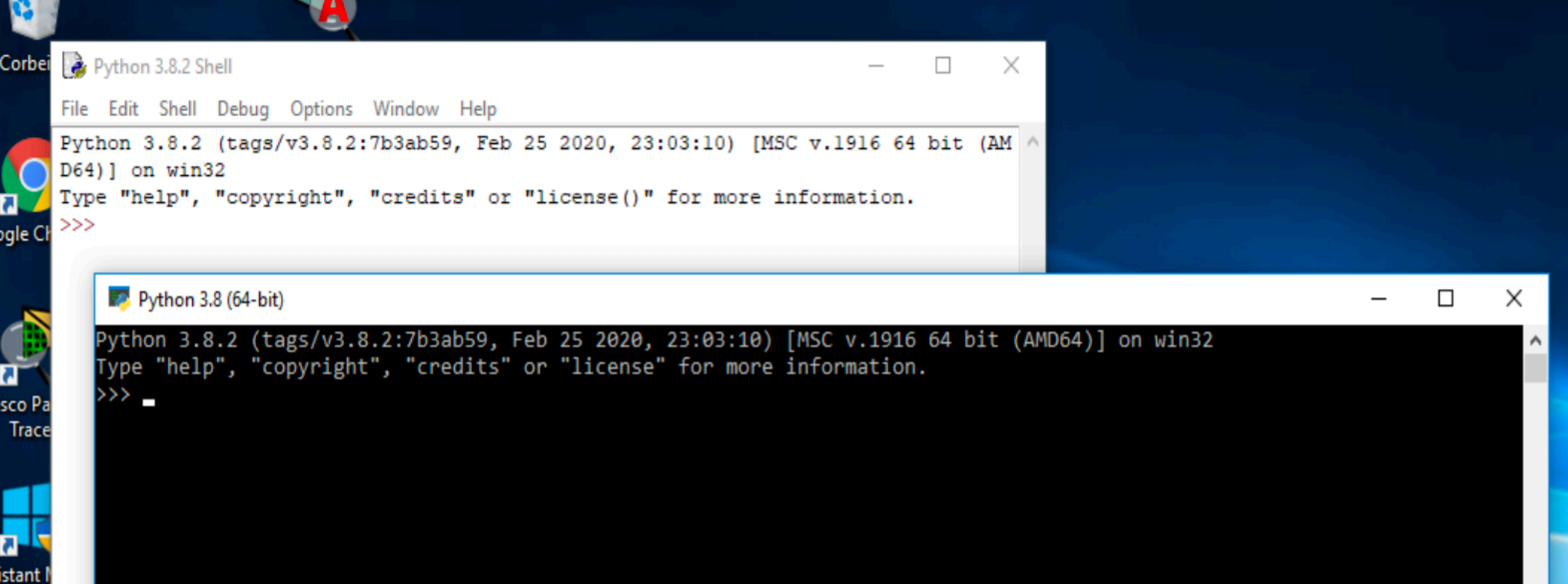
- Par exemple, <https://repl.it/languages/python3>

3. Utiliser un "**notebook**"

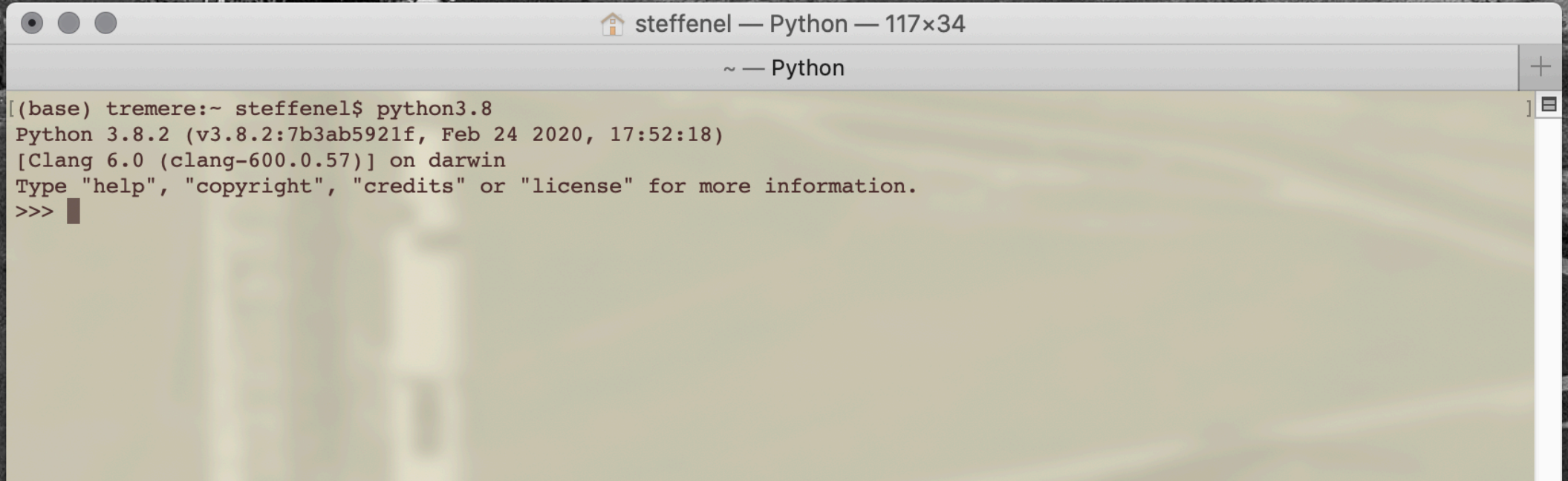
- Par exemple, <https://colab.research.google.com/>



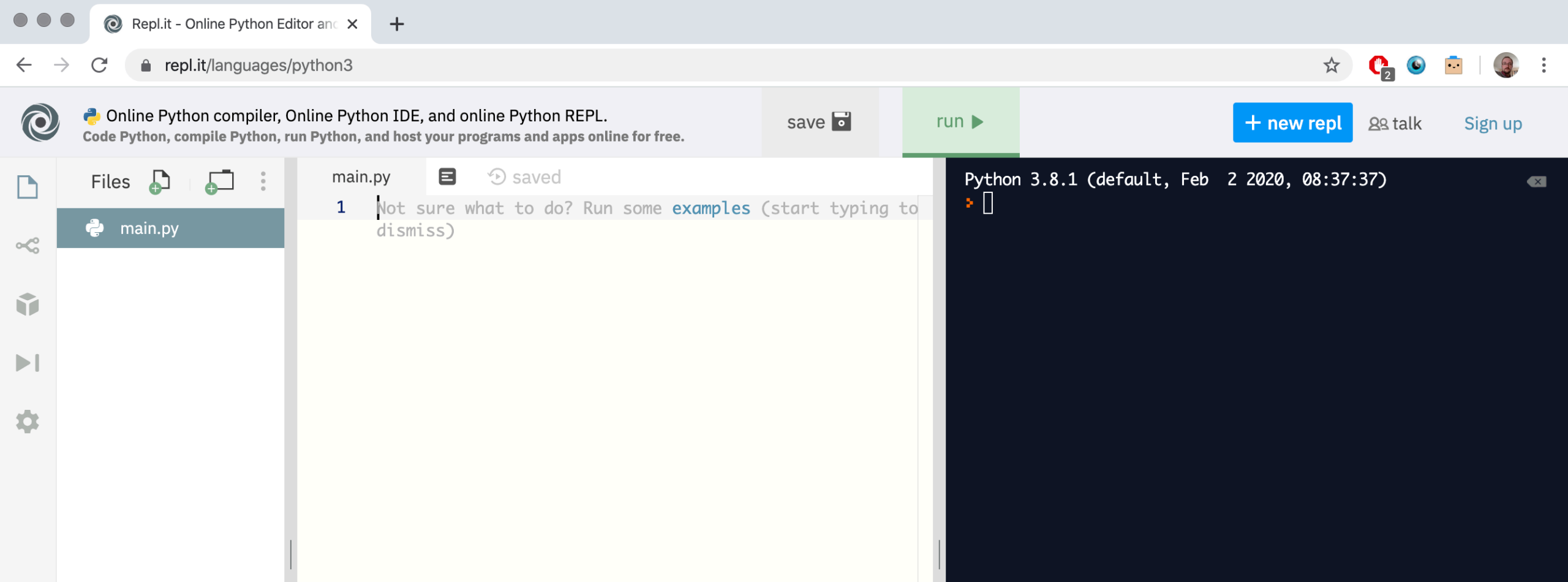
Toujours utiliser la
version 3 ou
supérieure
La v2 est expirée




IDLE et Terminal sous Windows






Terminal MacOS



<https://repl.it/languages/python3>

 **Bienvenue dans Colaboratory**

FichierModifierAffichageInsérerExécutionOutilsAide

 Partager

Sommaire

<>Présentation de Colaboratory

Premiers pas

Autres ressources

Exemples de machine learning : projet Seedbank

+ Section

+ Code+ Texte

Copier sur Drive

Connecter

Modification

▼ Premiers pas

Le document que vous consultez est un [notebook Jupyter](#), hébergé dans Colaboratory. Il ne s'agit pas d'une page statique, mais d'un environnement interactif qui vous permet d'écrire et d'exécuter du code en Python ou dans un autre langage.

Voici par exemple une **cellule de code** avec un bref script en Python qui calcule une valeur, l'enregistre dans une variable et imprime le résultat :

▶

```
seconds_in_a_day = 24 * 60 * 60
seconds_in_a_day
```

↑↓↻⚙️🗑️⋮

👤

86400

Pour exécuter le code dans la cellule ci-dessus, sélectionnez-le en cliquant dessus, puis cliquez sur le bouton de lecture à gauche du code, ou utilisez le raccourci clavier Commande/Ctrl+Entrée.

Toutes les cellules modifient le même état général. Les variables que vous définissez en exécutant une cellule peuvent donc être utilisées dans d'autres cellules :

<https://colab.research.google.com/>

Démarrer doucement

- Un code python peut être exécuté **directement à partir du clavier** ou grâce à un **programme écrit dans un fichier texte**
 - Un programme python est identifié par **l'extension .py**
- Pour des opérations simples, on peut utiliser l'interpréteur ou de l'environnement de développement (IDLE ou repl.it)
- Les environnements de type Notebook sont très utilisés pour des tutoriaux en ligne ("texte + code + exercices")

print() : imprimer un message sur l'écran

- Commande de base pour **afficher des messages sur l'écran**
 - Ex : `print('Bonjour tout le monde')`
- Permet la concaténation de messages
 - Ex: `print('Bonjour', 'tout', 'le', 'monde')`
 - Par défaut le séparateur est un espace
 - L'option **sep** permet de modifier le séparateur
- Attention avec les caractères spéciaux

```
Python 3.8.1 (default, Feb  2 2020, 08:37:37)
print('Bonjour tout le monde')
Bonjour tout le monde
>
> print('Bonjour', 'tout', 'le', 'monde')
Bonjour tout le monde
>
> print('Comment imprimer un apostrophe ' ?')
File "<stdin>", line 1
    print('Comment imprimer un apostrophe ' ?')
                                         ^
SyntaxError: invalid syntax
> print('Comment imprimer un apostrophe \' ?')
Comment imprimer un apostrophe ' ?
> 
```


Opérateurs arithmétiques

- **Opérations arithmétiques** simples

- + (somme), - (soustraction), * (multiplication), / (division)
- ** (exponentiation)
- // (division entière), % (reste de la division entière)

- **Précédence des opérations**

- Comme en mathématique
- Utilisation de parenthèses () mais pas de crochets ou accolades
 - $(2+5) * ((3/2) - (1/2))$

PEMDAS



Parenthèses
Exposants
Multiplication
Division
Addition
Soustraction

Les variables

- Même idée que les x et y en mathématique
 - **Une variable est un " tiroir "** dont la valeur qu'il contient peut varier
- **Attribution d'une valeur à une variable avec =**
 - `x = 10`
 - `pi = 3.14159`
 - `prenom = 'Angelo'`
- Les variables ont des **types**
 - Entier → classe 'int'
 - Réel → classe 'float'
 - Chaîne de caractères → classe 'str'

```
>>> x = 10
>>> pi = 3.14159
>>> prenom = 'Angelo'
>>>
>>> type(x)
<class 'int'>
>>> type(pi)
<class 'float'>
>>> type(prenom)
<class 'str'>
>>>
>>> len (prenom)
6
>>> len (pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'float' has no len()
```

- Détermine le type d'une variable
 - `type(x)`
- La longueur d'une variable
 - `len(prenom)`

input() : lire à partir du clavier

- Très utile pour **récupérer la valeur d'un paramètre au moment de l'exécution**
 - Évite d'écrire la valeur "en dur" sur le code
 - On peut donner un **message d'attente**
- Attention : **la valeur lue** est de type **str**
 - Selon le besoin **il faut la convertir**
- Conversion de types
 - `int(variable)`
 - `float(variable)`
 - `str(variable)`

```
main.py  saving...
1  print('quel est votre prenom ?')
2  prenom = input()
3  print('salut', prenom)
4
5  age = input('et quelle est votre age ? ')
6  bientot = int(age)+1
7  print(prenom, 'l\'année prochaine tu feras', bientot)
```

```
quel est votre prenom ?
Angelo
salut Angelo
et quelle est votre age ? 42
Angelo l'année prochaine tu feras 43
>
```

Les tests conditionnels

- Un test conditionnel **compare des valeurs**
 - Plusieurs types de comparateurs
 - `==` (égalité)
 - `>`, `<`, `<=`, `>=`
 - `!=` (différent)
- Le résultat final est **True** ou **False** (classe bool)
 - Possibilité d'associer les tests
 - **and** (ET logique), **or** (ou logique)
 - Souvent utilisé avec **if** ou les boucles **while**

```
permisdeconduire = False
age = input('quelle est votre age ? ')

if int(age) >= 18 and permisdeconduire:
    print('vous pouvez conduire et boire')
else:
    print('vous ne pouvez pas conduire')
```

```
quelle est votre age ? 42
vous ne pouvez pas conduire
```



Blocs d'instructions

- les blocs d'instructions ont toujours la même structure
 - Une **entête** terminant par : **(deux points)**
 - Des **instructions indentées** sous cette ligne d'en-tête
- Utiliser toujours le même type d'indentation
 - Soit **tabs**, soit un **numéro d'espaces** prédéfini
 - Le mélange donne droit à des **messages d'erreur**
- On peut écrire des **blocs imbriqués**
 - Chaque recul est un nouveau niveau

```
main.py  saving...
1  permisdeconduire = False
2  age = input('quelle est votre age ? ')
3
4  if int(age) >= 18 and permisdeconduire:
5      print('vous pouvez conduire')
6      print('vous pouvez boire aussi')
7  else:
8      print('vous ne pouvez pas conduire')
9
```

```
File "main.py", line 6
    print('vous pouvez boire aussi')
    ^
IndentationError: unexpected indent
> |
```

Séquences et Listes

```
> s = [1,7,2,4]
> len(s)
4
> 3 in s
False
> s[2]
2
> s + [5,3]
[1, 7, 2, 4, 5, 3]
> s
[1, 7, 2, 4]
```

- Python inclut 3 types de séquences
 - Chaînes (ex chaînes de caractères)
 - Listes → ce sont des tableaux
 - N-uplets
- Les **listes sont des séquences modifiables** d'éléments
 - Peuvent être multidimensionnelles (tableaux, matrices, etc.)
- Opérations typiques
 - Inclusion
 - `x in s`
 - Concatenation `s1 + s2`
 - Accès à une position
 - `s[i]` - accès à l'élément d'indice i (l'origine à l'indice 0)
 - `s[i:j]` accès aux éléments entre les indices i (inclus) et j (exclus)
 - Longueur – `len(s)`
 - Valeurs Min et max dans la liste - `min(s)`, `max(s)`

Boucles

- Boucles **while**
- Le bloc démarre avec l'instruction **while** et un test logique

```
while (solde_compte > 0):  
    solde_compte = solde_compte - achats
```
- Il se répète **tant que le test est True**

- Boucles **for**
 - Utilisé pour parcourir une séquence
- Format

```
for variable in séquence:  
    instruction
```
- Exemple

```
for i in [1,2,3,5,7]:  
    print ('nombre premier', i)
```

Fonctions et appels à fonctions

- Une fonction est un **bloc d'instructions nommé** censé être appelé plusieurs fois
 - Exemples de fonctions "natives" : `print()`, `len()`
- La définition d'une fonction se fait avec le mot clé **def** :

```
def nom_fonction ([paramètre1, ...]):  
    instructions  
    [return variables]
```
- Le plus souvent, vous allez appeler de fonctions issues de bibliothèques
 - Ex : bibliothèques mathématiques, de traitement de données, d'affichage graphique

Utiliser une bibliothèque (module)

- `import module` : charge un module appelé module.py
 - `import module as alias` : permet d'utiliser un alias plus explicite ou plus court
- Quelques bibliothèques utiles
 - **math** : opérations mathématiques
 - **random** : valeurs aléatoires
 - **numbers** : numéros complexes, intégrales, réel, ...
 - **time**, **datetime**, **calendar** : opérations, conversion et arithmétique avec du temps
- La commande "import" cherchera le fichier du module dans le répertoire courant ou dans le répertoire des bibliothèques du système

Autres bibliothèques intéressantes

- **Numpy** : bibliothèque pour la manipulation d'arrays et matrices
- **Scipy** : inclut des fonctions d'algèbre linéaire statistique, interpolation de données
- **Pandas** : pour la manipulation et l'analyse des données, les séries temporelles
- **Matplotlib** : pour l'affichage graphique des données
- **Scikit-learn** : pour le machine learning (classification, clustering)

Exercices Python

- Exercices de la Partie 1
- L'énoncé se trouve à l'adresse
 - <https://github.com/lsteffene1/IntroPythonData/blob/master/Exercices.pdf>

Bases de la bibliothèque Pandas

- Bibliothèque **manipulation de données tabulaires** (tableaux, matrices, etc.)
- Pour démarrer, on fait habituellement appel à deux modules

```
import numpy as np
import pandas as pd
```
- Deux structures de données : **DataFrame** et Series

DataFrames

- Les **DataFrame** (DF) sont des **structures de données tabulaires "riches"**
 - Nom et type des colonnes
 - Indexation ("clé primaire")
- On peut créer des DF manuellement ou à partir de fichiers (csv, excel, etc.)

Exemple : DataFrame avec les actions en bourse d'Apple

Date	Open	High	Low	Close	Volume	Adj Close
2014-09-16	99.80	101.26	98.89	100.86	66818200	100.86
2014-09-15	102.81	103.05	101.44	101.63	61216500	101.63
2014-09-12	101.21	102.19	101.08	101.66	62626100	101.66
...

Exemple manipulation de fichier CSV

- Pour créer un DataFrame à partir d'un **fichier csv**, il suffit de faire
- `df = pd.read_csv('nomfichier')`
- Des options pour gérer les entêtes, la transformation des données, etc.
 - `pd.read_csv('tmp.csv', index_col=[0], parse_dates=[0], header=None)`
 - `pd.read_csv('tmp.csv', index_col='Date', parse_dates=True)`
- On peut aussi **exporter sur un fichier** avec `to_csv('nomfichier')`
 - On peut aussi lire/exporter Excel, JSON, HTML, HDF5, ...

Des informations

Date	Open	High	Low	Close	Volume	Adj Close
2014-09-16	99.80	101.26	98.89	100.86	66818200	100.86
2014-09-15	102.81	103.05	101.44	101.63	61216500	101.63
2014-09-12	101.21	102.19	101.08	101.66	62626100	101.66
...

- On peut obtenir des **informations sur les datasets** avec `info()` et `describe()`

```
import pandas as pd
```

```
aapl = pd.read_csv('aapl.csv',  
index_col='Date', parse_dates=True)
```

```
print(aapl.info())
```

```
print(aapl.describe())
```

```
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 6081 entries, 2008-10-14 to 1984-09-07  
Data columns (total 6 columns):  
#   Column      Non-Null Count  Dtype  
---  ---  
0   Open        6081 non-null   float64  
1   High        6081 non-null   float64  
2   Low         6081 non-null   float64  
3   Close       6081 non-null   float64  
4   Volume      6081 non-null   int64  
5   Adj Close   6081 non-null   float64  
dtypes: float64(5), int64(1)  
memory usage: 332.6 KB  
None
```

info()

describe()

```
None  
      Open      High      Low      Close      Volume      Adj Close  
count  6081.000000  6081.000000  6081.000000  6081.000000  6.081000e+03  6081.000000  
mean    46.823511   47.681506   45.913595   46.798619  1.363986e+07   23.529794  
std     33.993517   34.578077   33.273106   33.947235  1.352107e+07   37.375601  
min     12.880000   13.190000   12.720000   12.940000  8.880000e+04    1.650000  
25%     24.730000   25.010000   24.200000   24.690000  5.530000e+06    7.380000  
50%     38.250000   38.880000   37.460000   38.130000  8.976400e+06    9.910000  
75%     53.500000   54.550000   52.500000   53.610000  1.631920e+07   14.360000  
max     200.590000  202.960000   197.800000   199.830000  2.650690e+08   199.830000
```

Recherche par position

```
AAPL.iloc[0:3, :]
```

Format : **[ligne_{init}:ligne_{fin}, col_{init}:col_{fin}]**

	Open	High	Low	Close	Volume	Adj Close
Date						
2014-09-16	99.80	101.26	98.89	100.86	66818200	100.86
2014-09-15	102.81	103.05	101.44	101.63	61216500	101.63
2014-09-12	101.21	102.19	101.08	101.66	62626100	101.66

Que ferait la commande

```
AAPL.iloc[-5: , :]
```

 ?

Recherche par position

Même chose que
AAPL.head(3)

AAPL.iloc[0:3, :]

	Open	High	Low	Close	Volume	Adj Close
Date						
2014-09-16	99.80	101.26	98.89	100.86	66818200	100.86
2014-09-15	102.81	103.05	101.44	101.63	61216500	101.63
2014-09-12	101.21	102.19	101.08	101.66	62626100	101.66

Que ferait la commande **AAPL.iloc[-5: , :]** ?

Même chose que
AAPL.tail(5)

Recherche par position

- Ça marche aussi avec les colonnes

```
➤ aapl.iloc[0:3 , : ]
```

	Open	High	Low	Close	Volume	Adj Close
Date						
2008-10-14	116.26	116.40	103.14	104.08	70749800	104.08
2008-10-13	104.55	110.53	101.02	110.26	54967000	110.26
2008-10-10	85.70	100.00	85.00	96.80	79260700	96.80

```
➤
```



```
➤ print(aapl.iloc[0:3,2:4])
```

	Low	Close
Date		
2008-10-14	103.14	104.08
2008-10-13	101.02	110.26
2008-10-10	85.00	96.80

Recherche par index

- Si on peut rechercher **par position** avec **iloc[]**, on peut aussi entrer une clé pour **rechercher par l'index avec loc[]**

```
aapl.iloc['2004-12-23']
```

- Dans certains cas (dates, par exemple) Python peut adapter la recherche
 - Ex : donner juste l'année (2004)

```
aapl.iloc['2004']
```

```
print(aapl.loc['2004-12-23'])
```

	Open	High	Low	Close	Volume	Adj Close
Date						
2004-12-23	63.75	64.25	63.6	64.01	8783200	32.01

```
✚
```

```
✚
```

```
print(aapl.loc['2004'].head(5))
```

	Open	High	Low	Close	Volume	Adj Close
Date						
2004-12-31	64.89	65.00	64.03	64.40	9949600	32.20
2004-12-30	64.81	65.03	64.22	64.80	12333600	32.40
2004-12-29	63.81	64.98	63.57	64.44	16055800	32.22
2004-12-28	63.30	64.25	62.05	64.18	21848400	32.09
2004-12-27	64.80	65.15	62.88	63.16	19981800	31.58

```
✚
```


Exercices Manipulation Pandas

- Exercices de la Partie 2
- L'énoncé se trouve à l'adresse
 - <https://github.com/lsteffene1/IntroPythonData/blob/master/Exercices.pdf>

Utilisation de DataFrames comme SQL

- Les DataFrames permettent des **opérations proches de celles du SQL**
 - **SELECT, WHERE**
 - **GROUP BY**
 - JOIN
 - UPDATE, DELETE
- De plus, les DataFrames Panda sont **facilement transposables sur des framework BigData**
 - Ex : **Apache Spark** contient aussi des DataFrame

SELECT

- Dans SQL, **l'opération SELECT sélectionne une liste de colonnes** (ou * pour toutes les colonnes) :

```
SELECT total_bill, tip, smoker, time  
FROM tips
```

- En Pandas, nous pouvons **passer une liste avec des noms de colonnes**

```
tips = pd.load_csv('tips.csv')  
select = tips[['total_bill', 'tip', 'smoker', 'time']]
```

WHERE

- La clause **WHERE** en SQL permet d'indiquer une **condition pour faire un filtrage** :

```
SELECT * FROM tips
```

```
WHERE time = 'Dinner'
```

- En Pandas, il est possible de faire ce filtrage de **plusieurs manières**. Voici un exemple :

```
dejeuner = tips[tips['time'] == 'Lunch']
```

- Les conditions peuvent se cumuler avec **& (ET logique)** et **| (OU logique)**

```
tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
```

GROUP BY

- L'opération **GROUP BY** regroupe des données en fonction d'un paramètre afin de faire une opération d'agrégation. En SQL :

```
SELECT sex, count(*) FROM tips  
GROUP BY sex;
```

```
Female 87  
Male 157
```

- En pandas, la méthode **groupby()** a la même fonction :

```
tips.groupby('sex').size()
```

- Il est possible d'utiliser **count()** mais ça affiche la somme par colonne

```
sex  
Female 87  
Male 157  
dtype: int64
```

```
total_bill  tip  smoker  day  time  size  
sex  
Female  87   87      87    87    87    87  
Male   157  157     157   157   157   157
```

Exercices Pandas SQL

- Exercices de la Partie 3
- L'énoncé se trouve à l'adresse
 - <https://github.com/lsteffene/IntroPythonData/blob/master/Exercices.pdf>

Visualisation avec Matplotlib

- Matplotlib est une bibliothèque pour **l'élaboration rapide de graphiques**
 - Très utilisé pour l'exploration des données
- Pandas s'intègre très facilement avec Matplotlib
 - Plusieurs possibilités d'affichage de DataFrames
- Ex de départ :

```
import pandas as pd  
import matplotlib.pyplot as plt  
aapl = pd.read_csv('aapl.csv',  
                  index_col='Date', parse_dates=True)
```

	adj_close	close	high	low	open	volume
date						
2000-03-01	31.68	130.31	132.06	118.50	118.56	38478000
2000-03-02	29.66	122.00	127.94	120.69	127.00	11136800
2000-03-03	31.12	128.00	128.23	120.00	124.87	11565200
2000-03-06	30.56	125.69	129.13	125.00	126.00	7520000
2000-03-07	29.87	122.87	127.44	121.12	126.44	9767600
2000-03-08	29.66	122.00	123.94	118.56	122.87	9690800

Visualisation d'un DataFrame

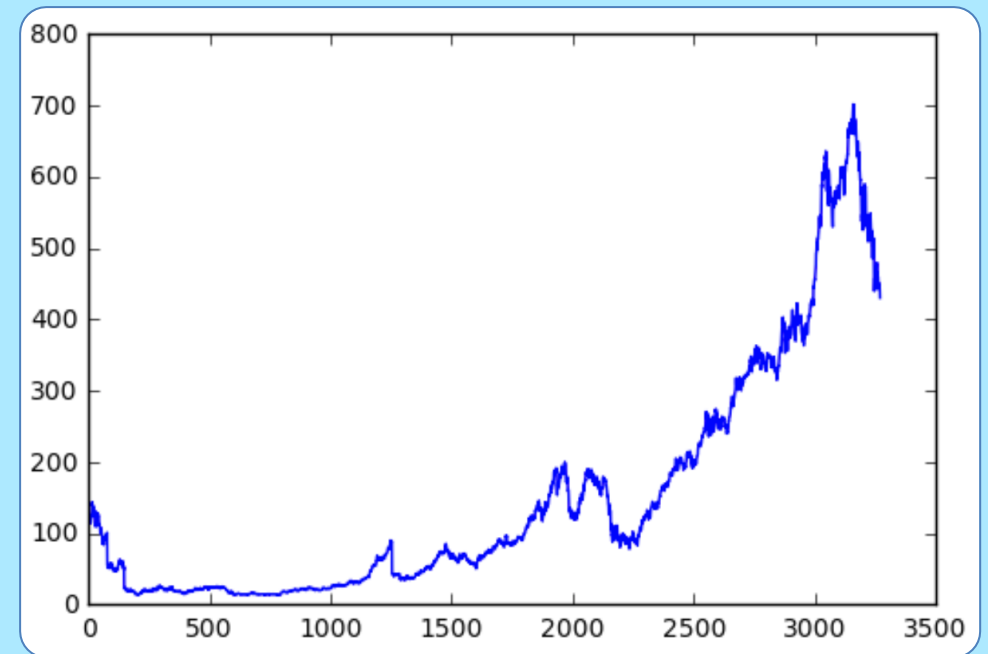
- On peut afficher le DataFrame en passant les données à matplotlib

```
close_arr = aapl['Close'].values
```

```
plt.plot(close_arr)
```

```
plt.show()
```

- L'inconvénient est qu'on perd certaines informations
 - Quelles sont les dates (notre index) ?



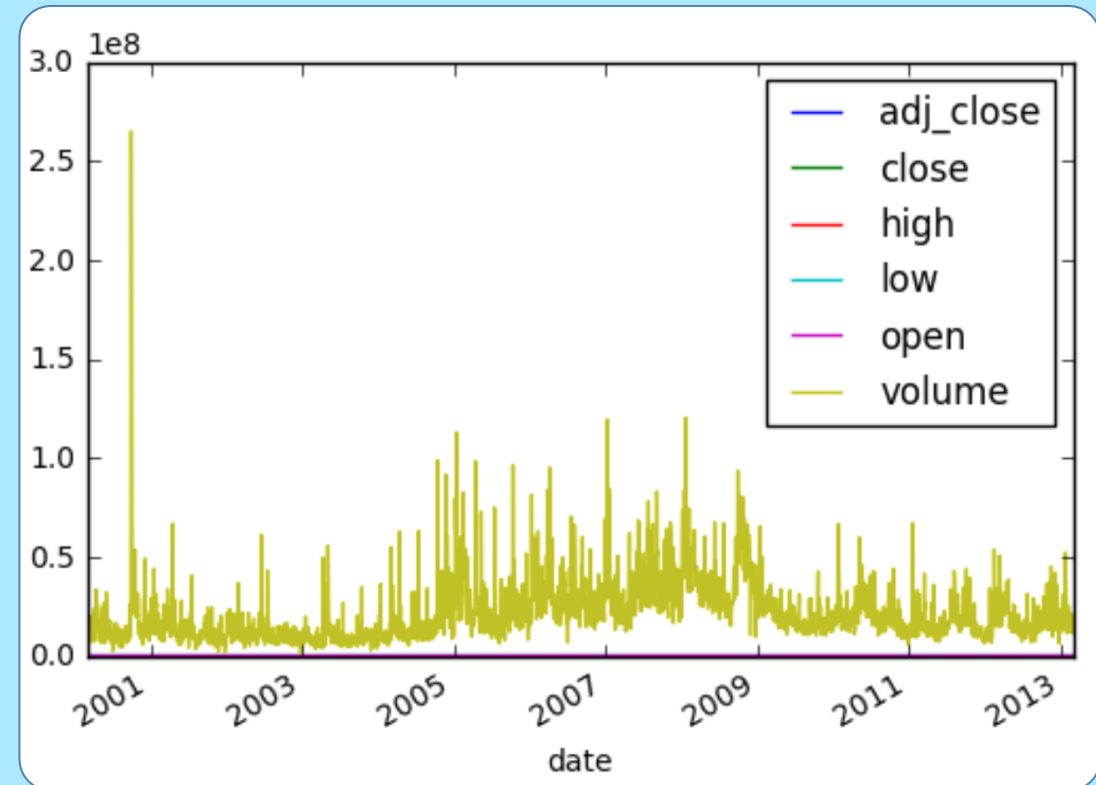
Visualisation d'un DataFrame

- On peut afficher directement le DataFrame

```
aapl.plot()
```

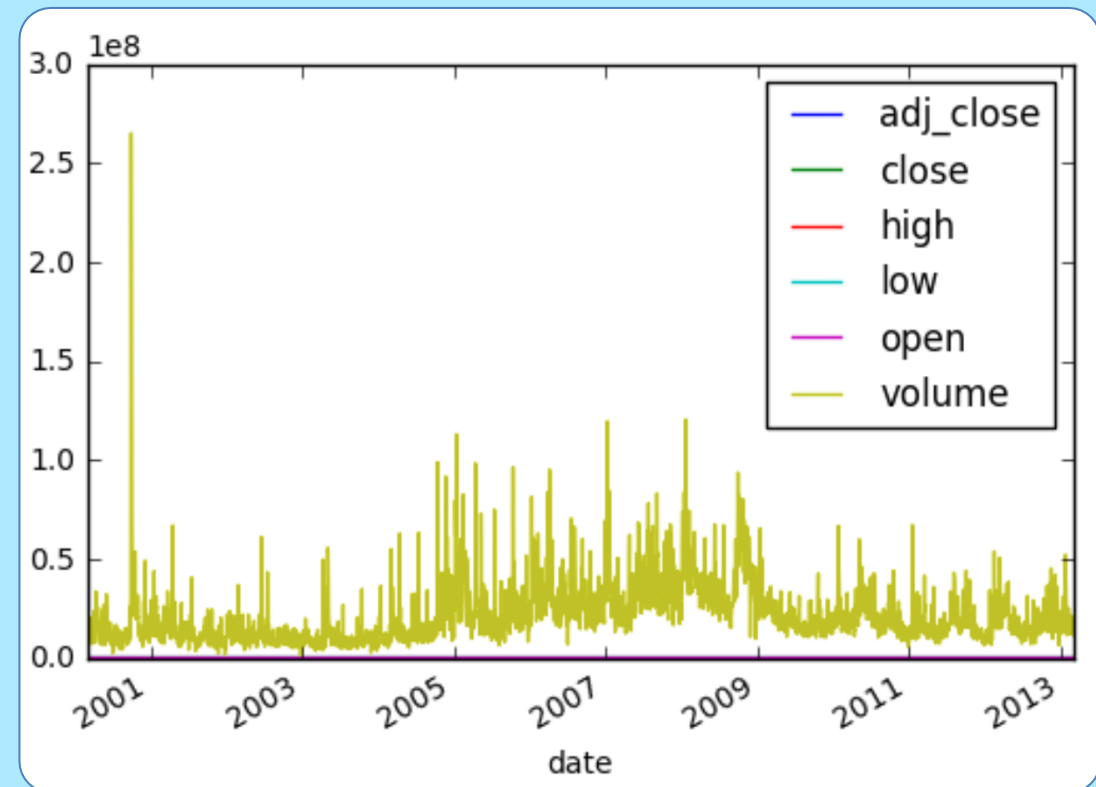
```
plt.show()
```

- Ici toutes les colonnes sont affichées
 - Nom des colonnes, des axes, tout y est !
 - Mais problème avec l'échelle des valeurs
 - Volume est trop grand, ça cache les autres



Visualisation d'un DataFrame

- On doit modifier l'échelle
- On peut aussi "filtrer"
 - les colonnes à afficher
 - les intervalles



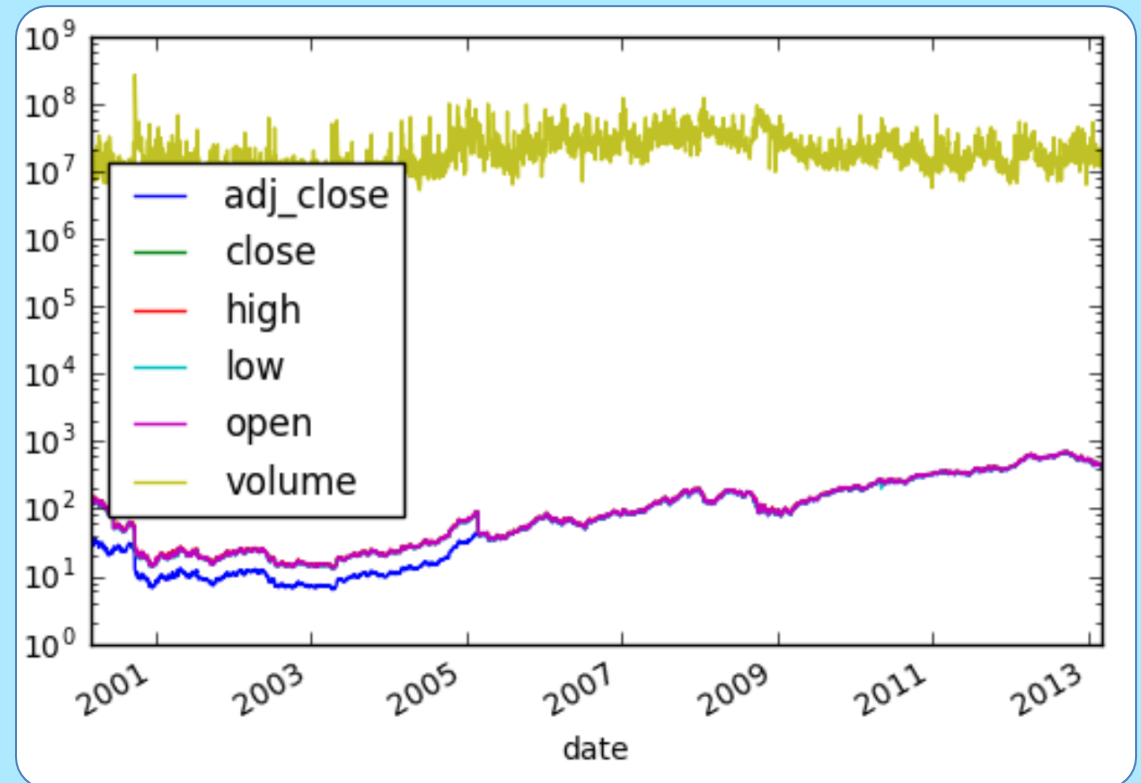
Modifier l'échelle

- Nous allons mettre une **échelle logarithmique sur l'axe Y**

```
aapl.plot()
```

```
plt.yscale('log')
```

```
plt.show()
```

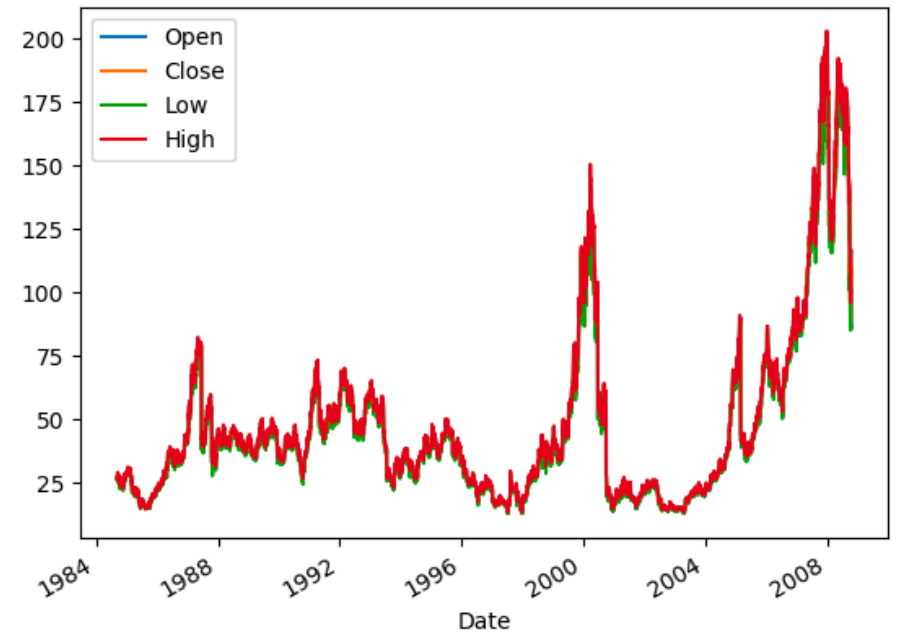


Customiser le graphique

- Nous allons **filtrer uniquement les colonnes** open, close, high et low

```
detail = aapl.loc[:,['Open', 'Close', 'High',  
'Low']]
```

```
detail.plot()
```



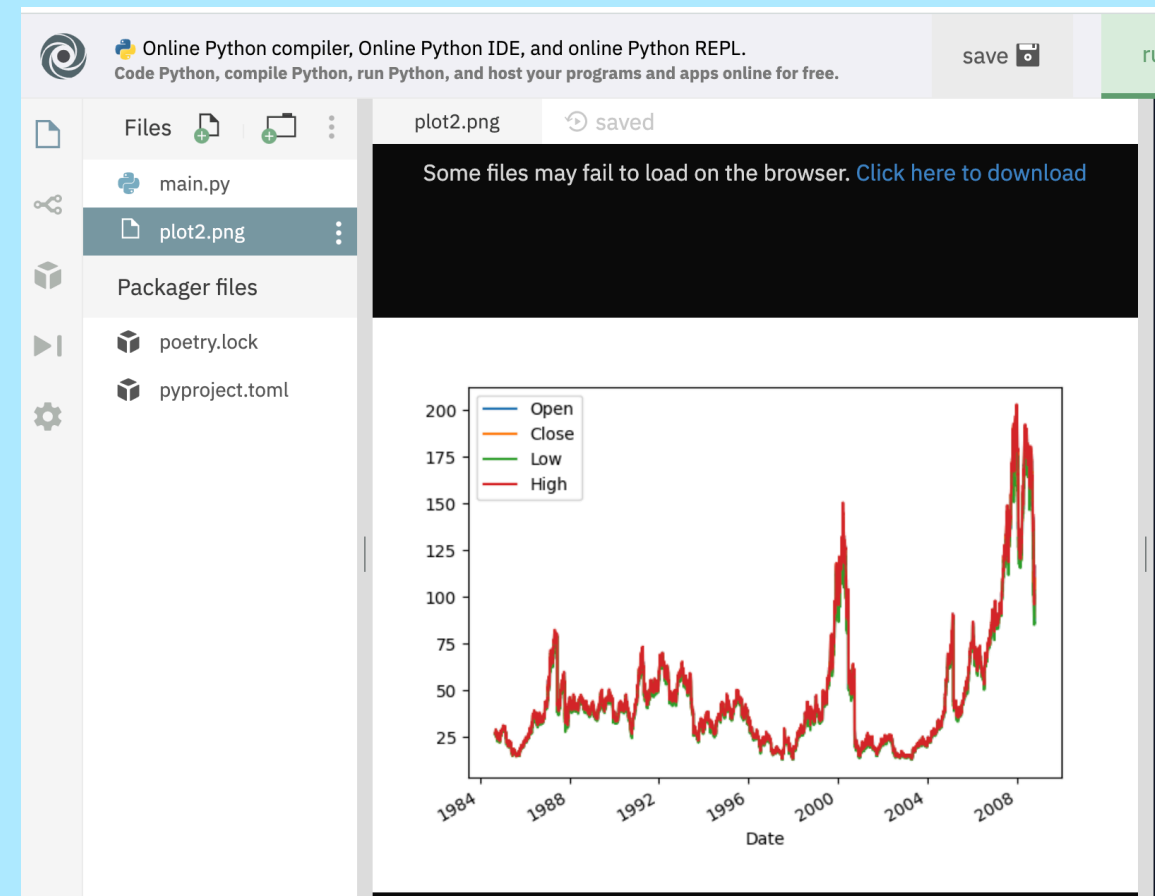
Enregistrer les graphiques

- La méthode **show()** fait l'affichage sur l'écran
 - **Ne marche pas sur repl.it, il faut utiliser savefig()**
- On peut enregistrer l'image dans un fichier
 - Plusieurs formats supportés (png, jpg, pdf)

```
detail = aapl.loc[:,['Open','Close','Low','High']]
```

```
detail.plot()
```

```
plt.savefig('plot.png')
```



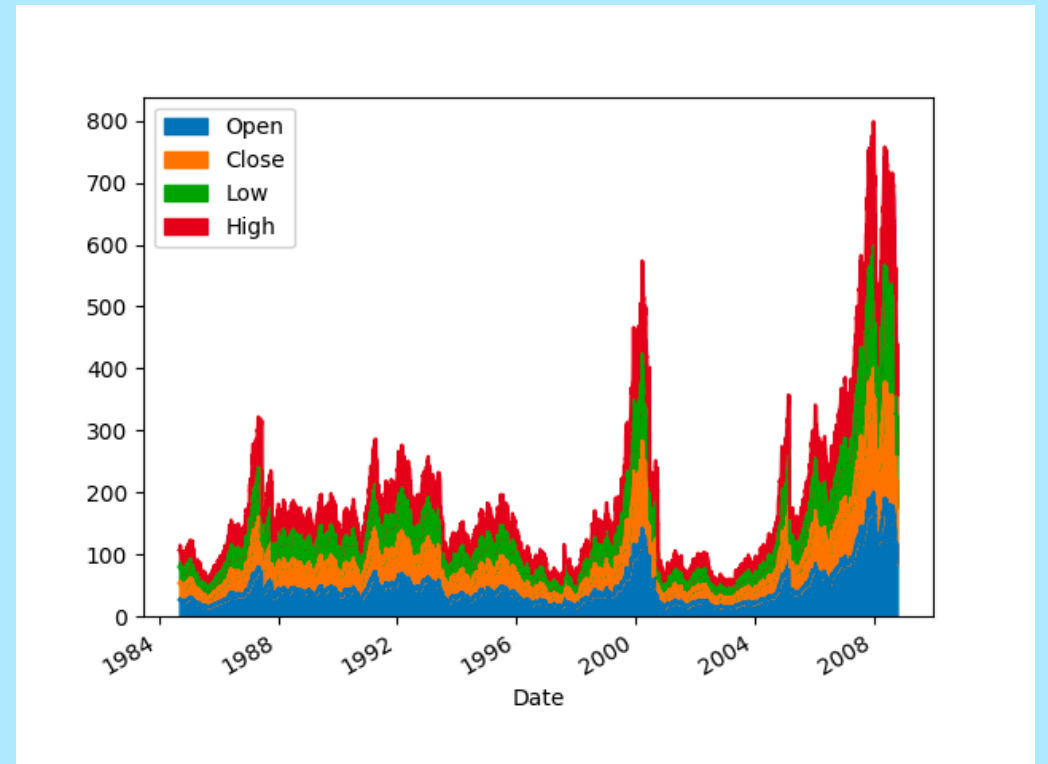
Enregistrer les graphiques

- La méthode `show()` fait l'affichage sur l'écran
 - **Ne marche pas sur repl.it, il faut utiliser `savefig()`**
- On peut enregistrer l'image dans un fichier
 - Plusieurs formats supportés (png, jpg, pdf)

```
detail = aapl.loc[:,['Open','Close','Low','High']]
```

```
detail.plot(kind='area')
```

```
plt.savefig('plot.png')
```



Exercices Pandas + Matplotlib

- Exercices de la Partie 4
- L'énoncé se trouve à l'adresse
 - <https://github.com/lsteffene1/IntroPythonData/blob/master/Exercices.pdf>