Approfondissement : Python et science des données

Raphaël Fournier-S'niehotta

Sorbonne Université, raphael.fournier@lip6.fr

Mineure IASD
Python L3
2025-26

SCIENCES SORBONNE UNIVERSITÉ

Plan

Vous et moi

Élèves

Tour de salle :

- présentation rapide
- niveau en Python ?

coder avec une IA ?

Enseignant

Raphaël Fournier-S'niehotta

Maître de conférences LIP6 / Sorbonne Université

Sujets de recherche et d'enseignement :

- Apprentissage sur graphes (GraphML, IA)
- Fouille de données (data Mining)

Adresses utiles :

- mail : raphael.fournier@lip6.fr
- page du cours :
  `http://raphael.fournier-sniehotta.fr/enseignement/2526-python-intro`

Introduction

Introduction

Historique

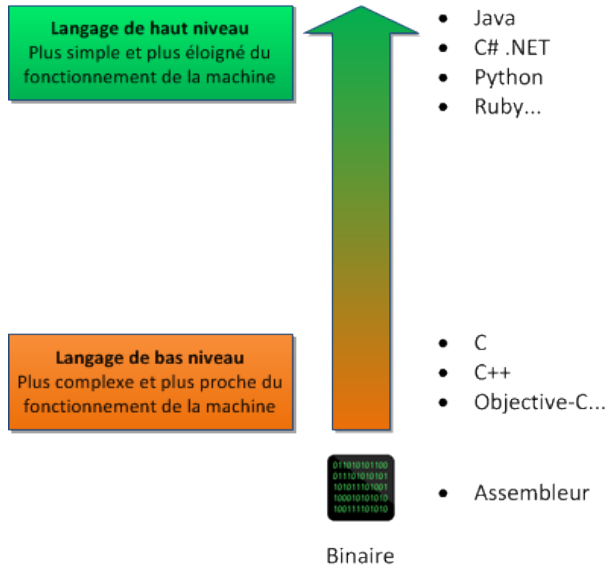
Le langage Python a été créé par Guido van Rossum en 1989 et rendu public en 1991. Le nom fait référence aux *Monty Python*.

G. van Rossum a été jusqu'à 2018 "Benevolent Dictator for Life".

`www.python.org`

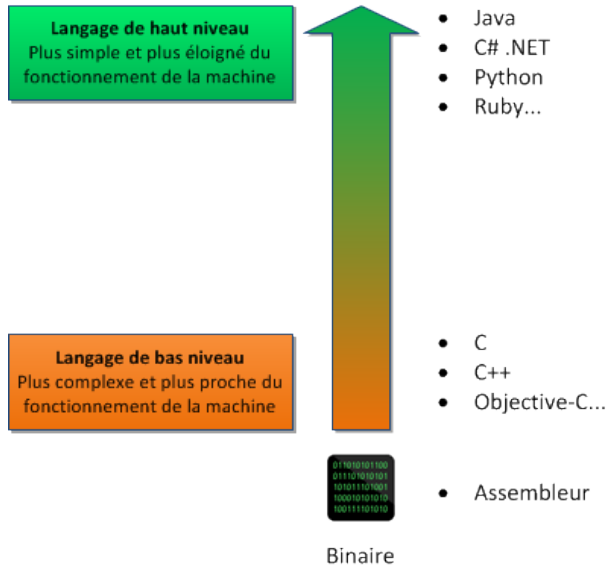
Un langage interprété...

- Simplificité d'écriture et flexibilité
- Exécution par un interpréteur ligne par ligne
- Exécution interactive ou par script
- Multiplateforme et open source
- Correction de bugs **relativement simple**

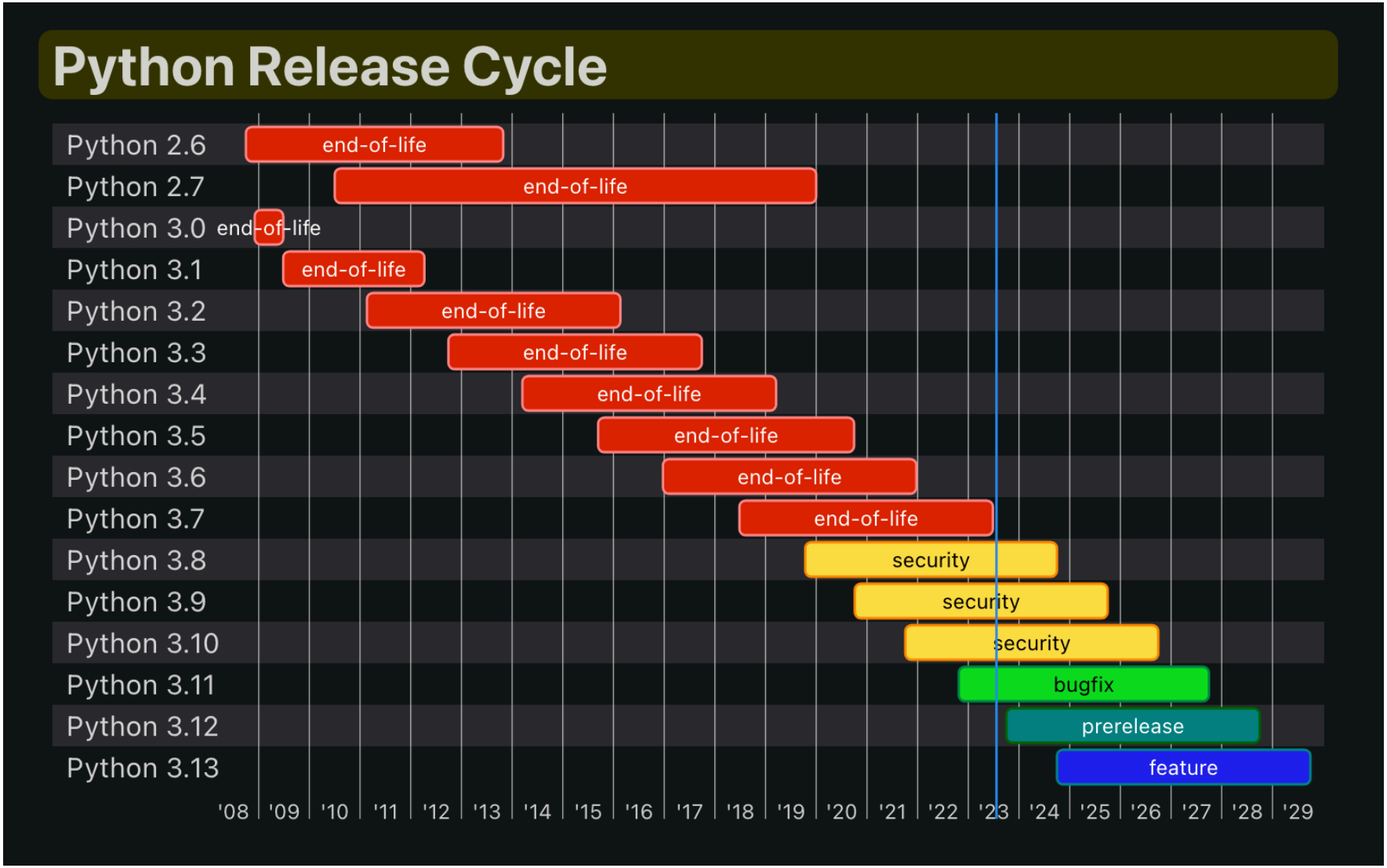


… mais relativement peu performant

- Interprétation vs compilation
- Gestion de données et bibliothèques de haut niveau
- Typage dynamique
- Gestion automatique de mémoire
- Traitement des boucles



Une histoire de versions



Installation et revue des outils

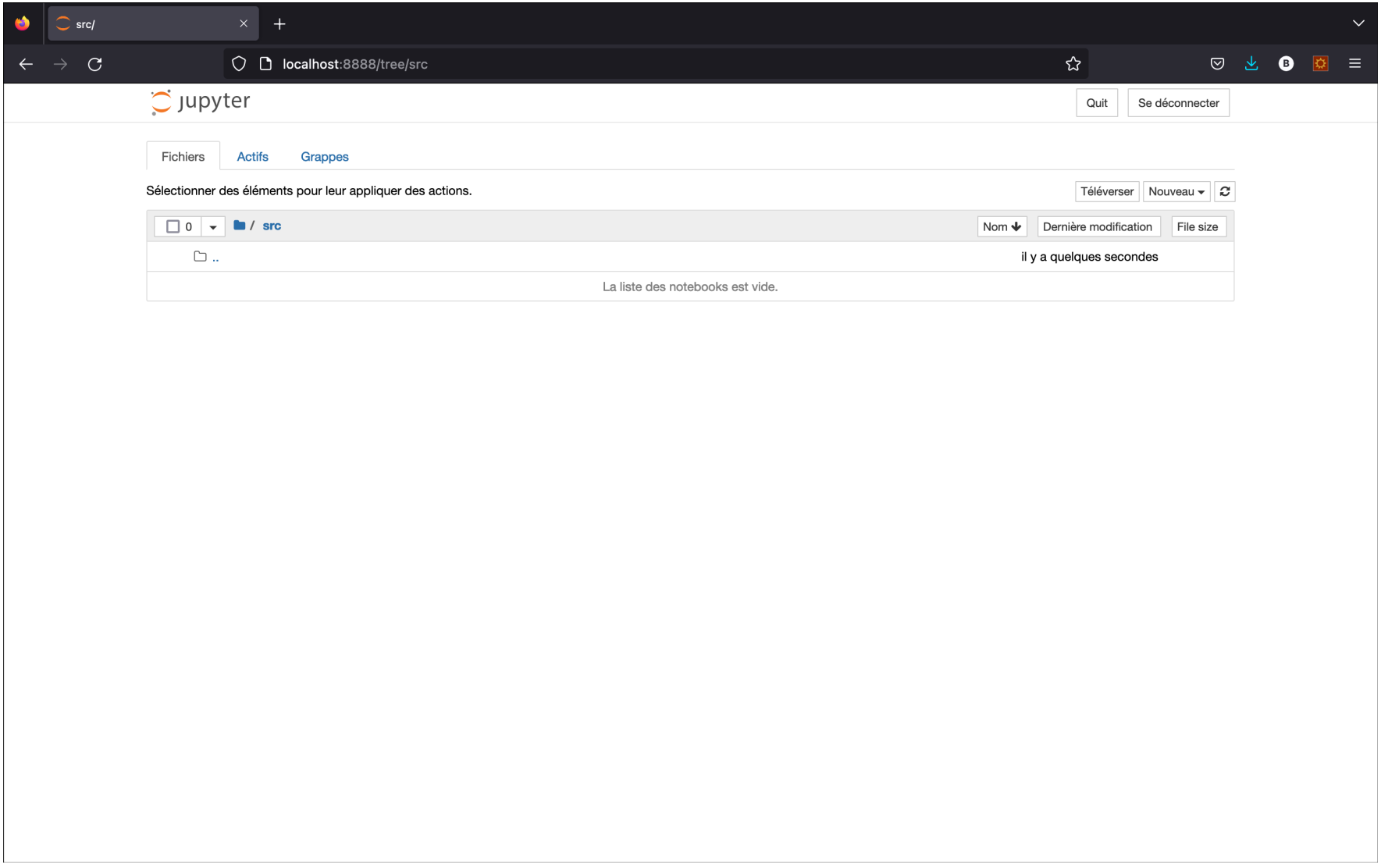
Pré-requis pour le cours

- Miniconda / Anaconda, python récent (cf versions)
- jupyter-lab ou jupyter-notebook
- Un IDE (Atom, VScode, etc.)
- Un terminal
  - Linux / MacOS : le terminal par défaut
  - Windows : privilégier Powershell, intégré à Anaconda

Commandes Linux à connaitre

Linux command	Description	Linux command example
cd	Change directory with a specified path	cd */path/directory1*
clear	Clear the screen	clear
cp	Copy file(s)	cp */path1/file1 /path2/file1*
diff	Compare the contents of files	diff *file1 file2*
exit	Log out of Linux	exit
grep	Find a string of text in a file	grep "word or phrase" *file1*
head	Display beginning of a file	head *file1*
less	View a file	less *file1*
ls	List contents of a directory	ls */path/directory1*
mv	Move file(s) or rename file(s)	mv */path1/file1 /path2/file2*
mkdir	Create a directory	mkdir *directory*
rm	Delete file(s)	rm *file1*
rmdir	Remove a directory	rmdir *directory*
tail	Display end of a file	tail *file1*
tar	Store, list or extract files in an archive	tar *file1*
vi	Edit file(s) with simple text editor	vi *file1*

Jupyter


src/
localhost:8888/tree/src
jupyter
Quit
Se déconnecter
Fichiers
Actifs
Grappes
Sélectionner des éléments pour leur appliquer des actions.
Téléverser
Nouveau
0
/ src
Nom
Dernière modification
File size
..
il y a quelques secondes
La liste des notebooks est vide.

Exécution d'un programme python : alternatives

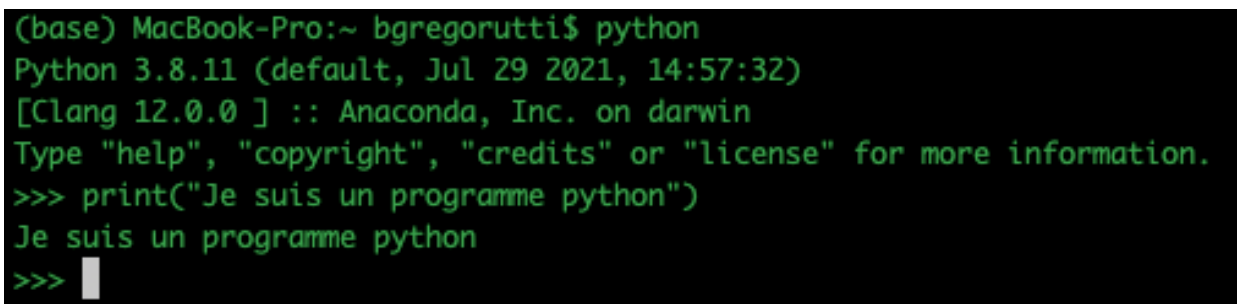
Cas 1 : via un shell python ou ipython

Exécution à la volée d'instructions python. Ouvrir un shell

```
$ python
```

Ecrire des instructions

```
>>> print("Je suis un programme python")
```



Exécution d'un programme python : alternatives

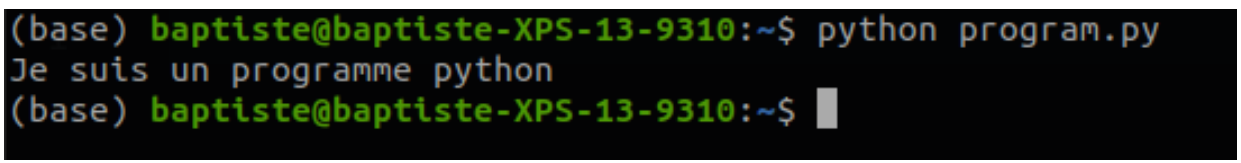
Cas 2 : via un terminal

Dans un éditeur de texte, créer un fichier `program.py` et écrire

```
print("Je suis un programme python")
```

Exécuter la ligne de commande

```
$ python program.py
```



Revue des outils

Outils de base :

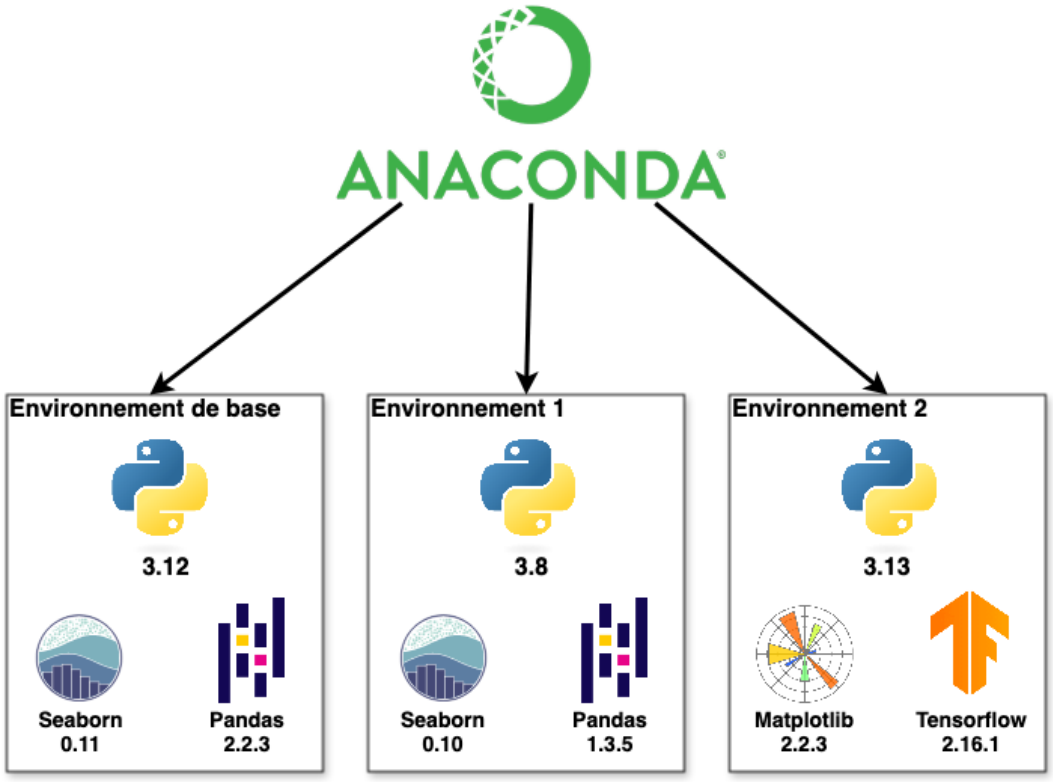
- python : exécuter du code à la volée
- pip : installer des packages

- pytest : tests unitaires
- pylint : vérification de la qualité du code source

Intégré au projet Anaconda :

- ipython : version interactive du shell python
- conda : installer des packages, gérer des environnements d'exécutions (voir ici), etc.
- Jupyter : environnement de développement interactif et flexible

Pour aller plus loin : environnements conda




```
$ conda create --name myenv python=3.9
$ conda activate myenv
$ conda install pip
$ pip install numpy
$ conda deactivate
```

Voir ici

Fondamentaux

Concepts de base : syntaxe et types

Syntaxe

- La syntaxe de Python repose sur une série d'instructions et des mots clés bien précis

```
>>> a = 1
>>> print(a)
1
```

- a est une **variable** et 1 est sa **valeur**
- la variable a est un objet
- la variable a a un type

```
>>> print(type(a))
int
```

Syntaxe

Les noms de variables sont libres à l'exception de certains mots réservés :

`def`, **`return`**, `list`, `type`, **`else`**, **`in`**

Liste complète ici

Conventions

- Ne pas mettre de caractère accentués ni de caractère non ASCII et préférer l'anglais
- Choisir des noms de variables qui soient compréhensibles
- Vous pouvez choisir des noms en plusieurs mots s'il n'est pas trop long, séparer les mots par un "_"

Syntaxe

Syntaxe en blocs indentés

- Indentation : décalage de lignes de code
- Délimiter des blocs logiques
- Convention : 4 espaces

```
def static_features(folder):

    # Headers
    header = 'filename chroma_stft spectral_centroid spectral_bandwidth rolloff zero_crossing_rate'
    for i in range(1, 21):
        header += ' mfcc{}'.format(i)
    header += ' label'
    header = header.split()

    features = []
    for k, filename in enumerate(os.listdir(folder)):

        if not k % 100:
            print('{} songs imported'.format(k))

        songname = os.path.join(folder, filename)
        genre = filename.split('_')[0]

        y, sr = librosa.load(songname, mono=True, duration=30)
        chroma_stft = librosa.feature.chroma_stft(y=y, sr=sr)
        spec_cent = librosa.feature.spectral_centroid(y=y, sr=sr)
        spec_bw = librosa.feature.spectral_bandwidth(y=y, sr=sr)
        rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr)
        zcr = librosa.feature.zero_crossing_rate(y)
        mfcc = librosa.feature.mfcc(y=y, sr=sr)

        current_song = [filename, np.mean(chroma_stft), np.mean(spec_cent), np.mean(spec_bw), np.mean(rolloff), np.mean(zcr)]
        for e in mfcc:
            current_song.append(np.mean(e))

        # Add the label
        current_song.append(genre)

        features.append(current_song)

    return DataFrame(features, columns=header)
```

Types de base : types numériques

- Entier, `int`

```
>>> a = 1
```

- Flottant, `float`

```
>>> a = 1.1
```

- Booléen, `bool`

```
>>> a = True
>>> b = False
```

Opérations sur les types numériques

Opérations élémentaires :

```
>>> 10 + 4
14
>>> 10 - 4
6
>>> 10 * 4
40
>>> 10 ** 4
10000
>>> 10 / 4
2.5
>>> 10 / float(4)
2.5
>>> 7 // 3
1
>>> 7 % 3
2
```

Opérations sur les booléens et comparaisons

Opérations :

```
>>> a = True
>>> b = a and False   # idem que b = a & False
>>> c = not a
>>> d = bool(0)
>>> e = bool(1)
```

Comparaisons :

```
>>> 5 > 3
>>> 5 >= 3
>>> 5 != 3
>>> 5 == 5
>>> 5 > 3 and 6 > 3
>>> 5 > 3 or 5 < 3
>>> not False
>>> False or not False and True
```

Types de base : types itérables

Types itérables, i.e. des séquences

- Liste, `list`

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

- Tuple, `tuple`

```
>>> a = (1, 2, 3)
```

- Dictionnaires, `dict`

```
>>> a = {"key1": 1, "key2": 2, "key3": 3}
```

Langage dynamiquement typé

Dynamique vs Statique

- Dynamique : le type est déterminé au **moment de l'exécution** et peu **changer**
- Statique : on fixe le type en début de programme

Inférence de type

Python détermine automatiquement le type d'une variable

```
>>> a = 1
>>> print(type(a))
int
>>> a = "hello"
>>> print(type(a))
str
```

Typage fort

Duck typing

- le type d'un objet est déterminé par l'ensemble de ses caractéristiques
- En particulier, une même opération peut fonctionner sur **objets de type différents**, si tant est que les opérations soient valables

```
>>> 5 + 4
9
>>> "titi" + "toto"
"tititoto"
```

Typage fort

Corollaire

Python **interdit** des opérations ayant peu de sens et **ne cherche pas à convertir** lui même.

Par exemple :

- **On ne peut pas** ajouter une chaîne de caractère et un entier
- **On peut** multiplier une chaîne de caractère et un entier

```
>>> "titi" * 2
"tititititi"

>>> "titi" + 2
TypeError: can only concatenate str (not "int") to str
```

Fonctions

Fonctions

Mots clés pour définir une fonction : `def` et `return`

```
>>> def fct(a, b, c):
...     d = (a + b) * c
...     return d

>>> fct(1, 2, 3)
```

- a, b et c sont les arguments (**nommés** ou **positionnels**)
- d est le retour de la fonction
- Le type des arguments n'est pas explicite mais peut l'être depuis la version 3.9 (PEP 3107)

Fonctions

Mots clés pour définir une fonction : `def` et `return`

```
>>> def fct(a=1, b=1, c=1):
...     d = (a + b) * c
...     return d

>>> fct()
```

- a, b et c sont les arguments (**nommés** ou **positionnels**)
- d est le retour de la fonction
- Le type des arguments n'est pas explicite mais peut l'être depuis la version 3.9 (PEP 3107)

Fonctions

Mots clés pour définir une fonction : `def` et `return`

```
>>> def fct(a, b, c=None):
...     if not c:
...         d = a + b
...     else:
...         d = (a + b) * c
...     return d

>>> fct(1, 2)
```

- a, b et c sont les arguments (**nommés** ou **positionnels**)
- d est le retour de la fonction
- Le type des arguments n'est pas explicite mais peut l'être depuis la version 3.9 (PEP 3107)

Fonction lambda

Une autre manière de définir une fonction :

```
>>> def sum(a, b):
...     return a + b
```

Avec `lambda` :

```
>>> sum = lambda a, b: a + b
```

Fonction lambda

Une autre manière de définir une fonction :

```
>>> def fct(a, b, c=None):
...     if not c:
...         d = a + b
...     else:
...         d = (a + b) * c
...     return d
```

Structures de contrôle

Structures de contrôle

Boucle for

```
>>> for item in iterable:
...     [instructions]
```

Exemple :

```
>>> for i in range(10):
...     print(i)
```

Mots clés spéciaux : `enumerate`, `break`, `pass` et `continue`

Structures de contrôle

Boucle while

```
>>> while [condition]:
...     [instructions]
```

Exemple :

```
>>> i = 0
>>> while i < 10:
...     print(i)
...     i += 1
```

Mots clés spéciaux : `break`, `pass` et `continue`

Structures de contrôle

Instruction if, elif, else

```
>>> if [condition1]:
...     [instructions]
... elif [condition2]:
...     [instructions]
... else:
...     [instructions]
```

Exemple :

```
>>> if i == 0:
...     print("i equals 0")
```

Remarque : `elif` et `else` sont optionnels

Structures de contrôle

Instruction if, elif, else

```
>>> if [condition1]:
...     [instructions]
... elif [condition2]:
...     [instructions]
... else:
...     [instructions]
```

Exemple :

```
>>> if not i:
...     print("i equals 0")
```

Remarque : `elif` et `else` sont optionnels

Structures de contrôle

Autre manière de faire

```
>>> if not i:
...     a = 1
... else:
...     a = 2
>>> a = 1 if not i else 2
```

Listes

Listes

Une liste est une séquence d'objets potentiellement de types différents :

```
>>> my_list = [True, 2, "3", 4]
```

Accès par indice :

```
my_list[start:stop:step]
```

Les indices commencent à 0 et peuvent être négatifs

Listes

Par exemple :

```
>>> my_list[0]
True

>>> my_list[0:1]
[True]

>>> my_list[0:2]
[True, 2]

>>> my_list[0:4:2]
[True, "3"]
```

Listes

Autres opérations utiles :

```
>>> print(2 in my_list)
True

>>> print([2, "3"] in my_list)
False

>>> list(range(4))
[0, 1, 2, 3]

>>> my_list + [10, 11]
[True, 2, "3", 4, 10, 11]

>>> my_list * 2
[True, 2, "3", 4, True, 2, "3", 4]
```

Opérations sur les listes

- Remplacer un élément
- Remplacer une sous-séquence (slicing)
- Supprimer des éléments
- Concaténer deux listes
- Répeter les élements d'une liste
- Ajout d'un élement en fin de liste

Méthodes et attributs d'une liste

Méthodes :

- append
- clear
- copy
- count
- extend
- index
- insert
- pop
- remove
- reverse
- sort

```
my_list = [1, 2, 3]
my_list.<nom de la methode>(<arguments>)
my_list.append(1)
```

Boucler sur une liste

```
>>> for item in my_list:
...     print(item)
```

Boucler sur une liste

```
>>> for item in my_list:
...     print(item)
```

Avec le mot clé `range`

```
>>> for i in range(len(my_list)):
...     print(my_list[i])
```

Boucler sur une liste

```
>>> for item in my_list:
...     print(item)
```

Avec le mot clé `range`

```
>>> for i in range(len(my_list)):
...     print(my_list[i])
```

Avec le mot clé `enumerate`

```
>>> for i, item in enumerate(my_list):
...     print(i, item)
```

Boucler sur une liste

Exemple : calculer le carré de chaque élément d'une liste d'entiers

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

Boucler sur une liste

Exemple : calculer le carré de chaque élément d'une liste d'entiers

```
>>> my_list = [1, 2, 3, 4]
>>> new_list = []
>>> for item in my_list:
...     new_list.append(item**2)
```

Boucler sur une liste

Exemple : calculer le carré de chaque élément d'une liste d'entiers

```
>>> my_list = [1, 2, 3, 4]
>>> new_list = []
>>> for item in my_list:
...     new_list.append(item**2)
```

Via une liste de compréhension :

```
>>> my_list = [1, 2, 3, 4]
>>> new_list = [item**2 for item in my_list]
```

Chaînes de caractères

Chaînes de caractères

Plusieurs manières d'écrire

```
>>> s = "une chaine de caracteres"
>>> s = 'une chaine de caracteres'
>>> s = """une chaine
de caracteres"""
```

Accès aux éléments

```
>>> s = "python"
>>> print(s[0])
"p"
>>> print(s[1:3])
"yt"
>>> print(s[1:6:2])
```

Chaînes de caractères

Boucler sur un chaine de caractères

```
>>> for item in "python":
...     print(item)
```

Concaténer plusieurs chaînes de caractères

```
>>> s = "une chaine" + "de" + "caracteres"
>>> print(s)
"une chainedecaracteres"
```

f-strings

Formatter une chaîne de caractères

```
>>> pi = 3.14159
>>> print(f"pi = {pi}")
pi = 3.14159
>>> print(f"pi = {pi:.2f}")
pi = 3.14
>>> print(f"pi = {pi:8.2f}")
pi =     3.14
```

Chaînes de caractères

Quelques méthodes utiles :

- `lower`
- `upper`
- `join`
- `replace`
- `split`

Tuples

Tuples

Un tuple est une séquence immutable d'objets potentiellement de types différents :

```
>>> my_tuple = (True, 2, "3", 4)
>>> my_tuple = (1,)
```

Accès par indice :

```
>>> my_tuple[0]
True
```

Comparaison avec les tuples

- ~~Remplacer un élément~~
- ~~Remplacer une sous-séquence (slicing)~~
- ~~Supprimer des éléments~~
- Concaténer deux tuples
- Répeter les élements d'un tuple
- ~~Ajout d'un élement en fin de tuple~~

Boucler sur un tuple

```
>>> for item in my_tuple:
...     print(item)
```

Boucler sur un tuple

```
>>> for item in my_tuple:
...     print(item)
```

Avec le mot clé `range`

```
>>> for i in range(len(my_tuple)):
...     print(my_tuple[i])
```

Boucler sur un tuple

```
>>> for item in my_tuple:
...     print(item)
```

Avec le mot clé `range`

```
>>> for i in range(len(my_tuple)):
...     print(my_tuple[i])
```

Avec le mot clé `enumerate`

```
>>> for i, item in enumerate(my_tuple):
...     print(i, item)
```

Dictionnaires

Dictionnaires

Un dictionnaire est une séquence mutable selon le paradigme clé/valeurs :

```
>>> my_dict = {"key1": 1, "key2": 2}
>>> my_dict = dict(key1=1, key2=2)
```

Accès par clé :

```
>>> my_dict["key1"]
1
>>> my_dict.get("key1")
1
```

Dictionnaires

Un dictionnaire est une séquence mutable selon le paradigme clé/valeurs :

```
>>> my_dict = {"key1": 1, "key2": 2}
>>> my_dict = dict(key1=1, key2=2)
```

Accès par clé :

```
>>> my_dict["key1"]
1
>>> my_dict.get("key1")
1
```

Que se passe-t-il si la clé n'existe pas ?

```
>>> my_dict["key3"]
?
>>> my_dict.get("key3")
?
```

Dictionnaires

Ajouter un élément :

```
>>> my_dict["new_key"] = new_value
```

Ou bien : utiliser la méthode `update`

Opérations sur les dictionnaires

Opérations

- Ajouter un élément
- Remplacer un élément
- Supprimer des éléments
- Concaténer deux dictionnaires

Méthodes associées

- `get`
- `keys`
- `values`
- `items`
- `clear`
- `pop`
- `update`

Boucler sur un dictionnaire

```
>>> for key in my_dict:
...     print(key, my_dict[key])
```

Boucler sur un dictionnaire

```
>>> for key in my_dict:
...     print(key, my_dict[key])
>>> for item in my_dict.items():
...     print(item)
```

Boucler sur un dictionnaire

```
>>> for key in my_dict:
...     print(key, my_dict[key])
>>> for item in my_dict.items():
...     print(item)
>>> for item in my_dict.values():
...     print(item)
```

Ensembles

Ensemble

Un ensemble est une séquence **mutable** contenant des éléments **ordonnés** et **uniques**. Un ensemble vide est créé par `set()`.

```
>>> a = {1, 2, 3, 3, 3, 3}
>>> print(a)
{1, 2, 3}
```

Intérêt des ensembles :

- Tests d'appartenance d'un élément à une séquence
- Suppression de doublons : `set([1, 1, 2, 2, 3])`
- Opérations mathématiques : unions, intersections, etc.

Mutabilité

Mutable vs immutable

Objet mutable

Un objet mutable peut être modifié après sa création

- list
- dict
- set

Objet immutable

Un objet immutable ne peut être modifié après sa création

- int, float, bool
- str
- tuple
- byte

Mutable vs immutable

Immutabilité : modification d'un entier

```
>>> a = 1
>>> id(a)
xxxxxxx560
>>> a += 1
>>> id(a)
xxxxxxx592
```

Que se passe-t-il ici ?

Mutable vs immutable

Immutabilité : modification d'un entier

```
>>> a = 1
>>> id(a)
xxxxxxx560
>>> a += 1
>>> id(a)
xxxxxxx592
```

Que se passe-t-il ici ?

Copie implicite

Mutable vs immutable

Mutabilité : modification d'une liste

```
>>> a = [1, 2]
>>> id(a)
xxxxxxx328
>>> a[0] += 1
>>> id(a)
xxxxxxx328
```

Mutable vs immutable

Autre exemple, `int`

```
>>> a = 1
>>> b = a
>>> b is a
True
>>> b += 1
>>> b
2


>>> b is a
False
```

Mutable vs immutable

Autre exemple, `list`

```
>>> a = [1, 2]
>>> b = a
>>> b is a
True
>>> b[0] += 1
>>> b
[2, 2]
>>> a
[2, 2]
>>> b is a
True
```

Mutable vs immutable

Cas d'objets mutables, dans une fonction

```
>>> def cast_list(l, idx):
...
...     l[idx] = str(l[idx])
...     return l

>>> l = [1, 2]

>>> print(cast_list(l, 0))
["1", 2]

>>> print(l)
???
```

Mutable vs immutable

Cas d'objets mutables, dans une fonction

```
>>> def cast_list(l, idx):
...     l = l.copy()
...     l[idx] = str(l[idx])
...     return l

>>> l = [1, 2]

>>> print(cast_list(l, 0))
["1", 2]

>>> print(l)
???
```

En bref

- Python gère les objets mutables et immutables différemment
- Les objets mutables sont très intéressants si on a besoin de changer leur structures (taille, type, etc.) mais cela peut être dangereux
- Les objets immutables sont plus rapides d'accès
- Les objets immutables doivent être préférés si on veux que l'objet reste le même tout au long de l'exécution
- Modifier un objet immutable est plus couteux car il nécessite une copie (explicite ou non)

Portée des variables

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...
...
...     val = a + b
...     return val

>>> sum(1, 2)
???

>>> print(val)
???
```

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...
...
...     val = a + b
...     return val

>>> sum(1, 2)
3


>>> print(val)
0
```

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...
...     val += 1
...     total_sum = a + b + val
...     return total_sum

>>> sum(1, 2)
UnboundLocalError: local variable "val" referenced before assignment

>>> print(val)
0
```

Portée des variables

Comme dans les autres langages, il existe deux types de variables en python :

- locales : des variables définies dans une fonction
- globales : des variables définies en dehors des fonctions

```
>>> val = 0
>>> def sum(a, b):
...     global val
...     val += 1
...     total_sum = a + b + val
...     return total_sum

>>> sum(1, 2)
4


>>> print(val)
1
```

Modules et imports

Modules et imports

Définition

Un module est un fichier python contenant un ensemble d'instructions (fonctions, classes, etc.).
Un module peut être :

- Créé localement, par exemple un fichier `mon_module.py`
- Inclus dans un package ou une librairie

Permet

1. Utiliser les fonctionnalités d'un module dans un autre
2. Structurer un programme python en plusieurs fichiers (implémentation **modulaire**)
3. Utiliser des packages open-source

Modules et imports

Exemple : my_module.py

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

Modules et imports

Exemple : my_module.py

```python
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> import my_module
>>> my_module.add(1, 2)
3
>>> my_module.prod(1, 2)
2
```

Modules et imports

Exemple : my_module.py

```
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> import my_module as mod
>>> mod.add(1, 2)
3
>>> mod.prod(1, 2)
2
```

Modules et imports

Exemple : my_module.py

```python
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> from my_module import add
>>> add(1, 2)
3
>>> prod(1, 2)
NameError: name "prod" is not defined
```

Modules et imports

Exemple : my_module.py

```python
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> from my_module import add, prod
>>> add(1, 2)
3
>>> prod(1, 2)
2
```

Modules et imports

Exemple : my_module.py

```python
def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

```
>>> from my_module import *    # NE SURTOUT PAS UTILISER
>>> add(1, 2)
3
>>> prod(1, 2)
2
```

Modules et imports

Pourquoi proscrire l'utilisation de `import *` ?

Exemple : la fonction `sqrt` existe dans plusieurs libraries :

- Dans `math` : calcule la racine carré pour un scalaire
- Dans `numpy` : calcule la racine carré pour un scalaire ou pour chaque élément d'un tableau

```
>>> from numpy import *
>>> from math import *
>>> print(sqrt([1, 2, 3]))
TypeError: must be real number, not list
```

Structurer un programme

Structure en plusieurs fichiers

- Un ou plusieurs modules contenant les fonctionnalités implémentées
- Un programme principal

```
my_project/
  - module1.py
  - module2.py
  - main.py
```

Structurer un programme

module1.py

```
import math
import sys

def prod(a, b):
    return a * b

def add(a, b):
    return a + b
```

main.py

```
from module1 import add, prod
print(add(1, 2))
print(prod(1, 2))
```