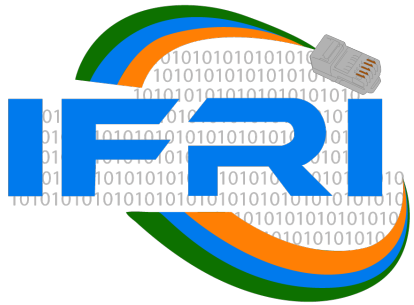
Programmation avancée en Python/R

Luc ATAKPA

Avril, 2024



DRAFT: EN COURS D'EDITION

Agenda

1. Objectifs du cours

2. Python avancé
   - Structures de données
   - Les fonctions comme objets
   - Classes et Protocoles
   - Structures de controle
   - Outils IA en Python

3. Introduction à la programmation en R

Objectifs du cours

- Approfondir vos bases du langage Python
- Introduire la programmation en R
- Introduire quelques outils Python pour l'IA

Agenda

1. Objectifs du cours

2. Python avancé
   - Structures de données
   - Les fonctions comme objets
   - Classes et Protocoles
   - Structures de controle
   - Outils IA en Python

3. Introduction à la programmation en R

Matériel

- Le cours Python est essentiellement basé sur l'ouvrage **Fluent Python**
- Nous utiliserons le dépôt GitHub fluentpython/example-code-2e
- Vous êtes amené à cloner le dépôt pour suivre les exemples pratiques
- Avoir Python installé est un prérequis
- Le cours sera un mixe de théorie et de sessions pratiques de codage
- Le but n'est pas d'apprendre des syntaxes, fonctions, méthodes etc... par cœur, mais de comprendre les aspects avancés de Python qui seront abordés tout au long du court.

Pourquoi Python Avancé

Un programmeur expérimenté peut commencer à écrire du code Python utile en quelques heures. Cependant, au fil des semaines et des mois de travail productif, de nombreux développeurs continuent d'écrire du code Python avec un fort accent hérité des langages appris auparavant. Même si Python est votre premier langage, il est souvent présenté dans les milieux académiques et les livres d'introduction en évitant soigneusement les fonctionnalités spécifiques au langage.

Luciano Ramalho, auteur de Fluent Python

Structures de données / Modèle de données

Le modèle de données Python fait référence aux éléments de base et aux mécanismes sous-jacents utilisés pour représenter et manipuler les données en Python. Il fournit un cadre qui nous permet de comprendre et d'interagir avec les objets intégrés au langage et ceux définis par l'utilisateur de manière **"Pythonique"**.

Vous pouvez considérer le modèle de données comme une description de Python en tant que framework. Il formalise les interfaces des éléments constitutifs du langage lui-même, tels que les séquences, les itérateurs, les fonctions, les classes, les gestionnaires de contexte, etc.

Éléments du modèle de données Python - I

- Tout est un objet en Python
- Chaque objet est caractérisé par son `identité`, son `type` et sa `valeur`
- L'identité et le type d'un objet Python ne peuvent être modifiés une fois l'objet créé.
- La valeur d'un objet peut changer si l'objet n'est pas immuable.
- L'identité d'un objet peut être perçue comme son adresse en mémoire (*CPython*)
- Les fonctions `id` et `type` renvoient respectivement l'id et le type d'un objet Python.

Éléments du modèle de données Python - II

Le modèle de données Python définit un ensemble de méthodes spéciales ou magiques (également appelées méthodes **dunder**, car elles commencent et se terminent par des doubles 'underscore', comme `__init__` ou `__str__`). L'implémentation de ces méthodes permet aux objets de supporter et d'interagir avec des éléments fondamentaux du langage tels que :

- Création et destruction d'objets (`__init__`, `__del__`)
- Représentation et formatage (`__repr__`, `__str__`, `__format__`)
- Comparaison et hachage (`__eq__`, `__lt__`, `__hash__`)
- Programmation asynchrone avec `await` (`__await__`)
- Collections (`__len__`, `__getitem__`, `__setitem__`, `__iter__`)
- Opérations numériques et autres (`__add__`, `__sub__`, `__call__`)
- etc...

Exemple de code

```
1  text = "Hi everyone"
2  num = 45.8
3  print(type(text), type(num))
4  print(id(text),id(num))
5  print(text, num)
```

Exercice - Complétez le Jeu de Cartes Pythonique

```
1  import collections
2  Card = collections.namedtuple('Card', ['rank', 'suit'])
3  class FrenchDeck:
4      ranks = [str(n) for n in range(2, 11)] + list('JQKA')
5      suits = 'spades diamonds clubs hearts'.split()
6      def __init__(self):
7          self._cards = [Card(rank, suit) for suit in self.suits
8                                          for rank in self.ranks]
9      def __len__(self):
10         return len(self._cards)
11     def __getitem__(self, position):
12         return self._cards[position]
```

Exercice - Complétez la classe Vecteur

```python
import math
class Vector:
    def __init__(self, x=0, y=0):
        self.x, self.y = x, y
    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'
    def __abs__(self):
        return math.hypot(self.x, self.y)
    def __bool__(self):
        return bool(abs(self))
    def __add__(self, other):
        ...
    ...
```

Éléments du modèle de données Python - III

Juste en implémentant les méthodes `__len__` et `__getitem__`, les objets d'une classe peuvent être utilisés comme une séquence avec d'autres constructions du langage Python, sans hériter d'une classe quelconque.

- Les utilisateurs de vos classes n'auront plus à chercher les méthodes utilisées pour des opérations standard (e.g. `.size()` ou `.length()`).
- C'est plus facile de bénéficier de la richesse de la bibliothèque standard de Python et ne réinventer la roue, comme la fonction `random.choice` .
- On a une intégration à bien d'autres constructions du langage qui fonctionnent avec les séquences (e.g. itération, recherche, slicing, etc...). (voir code).

Éléments du modèle de données Python - IV

- En général, les méthodes magiques sont censées être invoquées par l'interpréteur Python, pas par le programmeur.
- L'interpréteur invoque souvent les méthodes magiques de manière implicite. Comme `for card in deck` avec la classe `FrenchDeck`.
- Si vous avez besoin d'utiliser une méthode spéciale, il est préférable d'appeler la fonction native correspondante (e.g. `len`, `iter`, `str`).

Activité Pratique

Nous allons utiliser les fonctions spéciales pour la création d'une classe Python qui représente un vecteur bidimensionnel (dans un espace euclidien) et qui supporte les opérations suivantes : addition avec un autre vecteur, multiplication avec un scalaire, calcul de la magnitude du vecteur, support d'une valeur booléenne pour les vecteurs, affichage d'une représentation en chaine de caractères des objets de la classe. Utilisez les méthodes spéciales `__abs__`, `__add__`, `__mul__`, `__repr__`.

Extra: Rendre commutative la multiplication par un scalaire, et ajouter l'opération de soustraction entre deux vecteurs.

Structures de données / Modèle de données

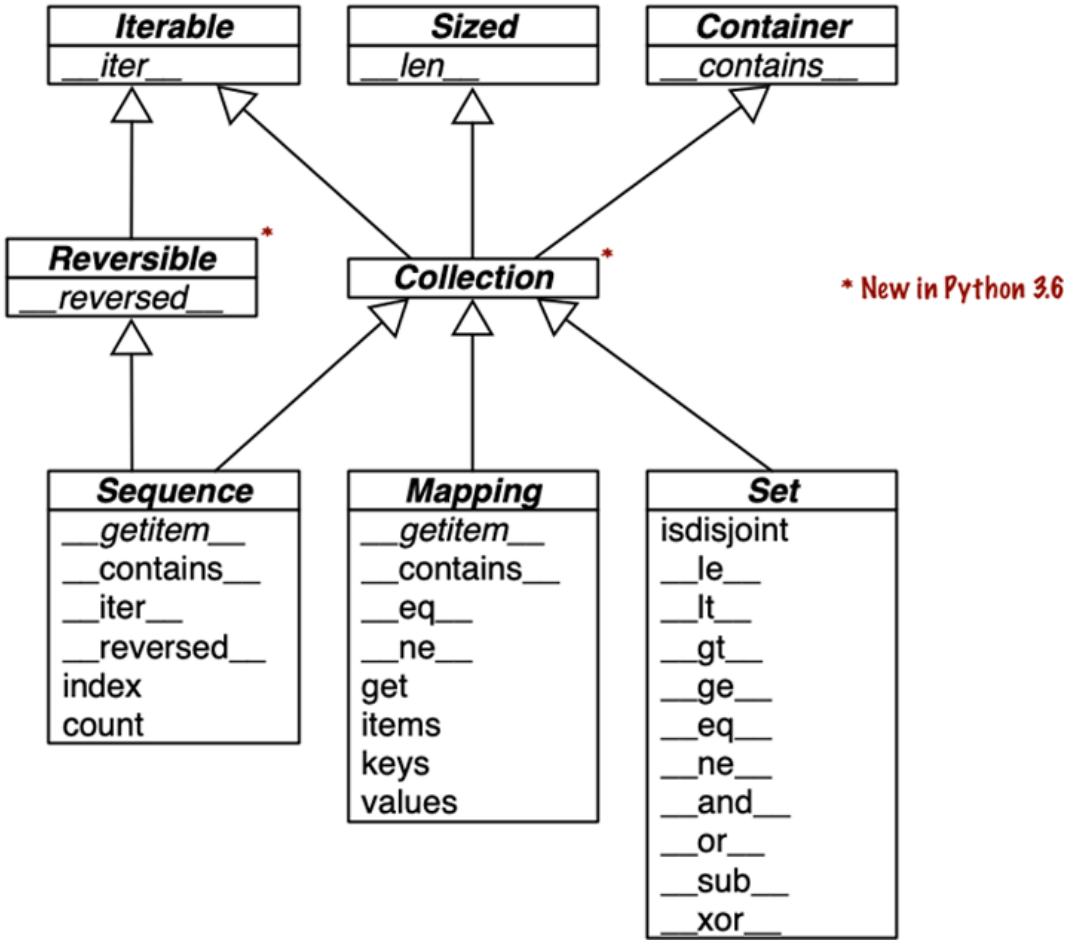


Diagramme UML de l'interface de certains types de collection (*Fluent Python*)

Aperçu des méthodes spéciales

Category	Method names
String/bytes representation	`__repr__`, `__str__`, `__format__`, `__bytes__`, `__fspath__`
Conversion to number	`__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
Emulating collections	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
Iteration	`__iter__`, `__aiter__`, `__next__`, `__anext__`, `__reversed__`
Callable or coroutine execution	`__call__`, `__await__`
Context management	`__enter__`, `__exit__`, `__aexit__`, `__aenter__`
Instance creation and destruction	`__new__`, `__init__`, `__del__`
Attribute management	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`

Table: Quelques méthodes spéciales

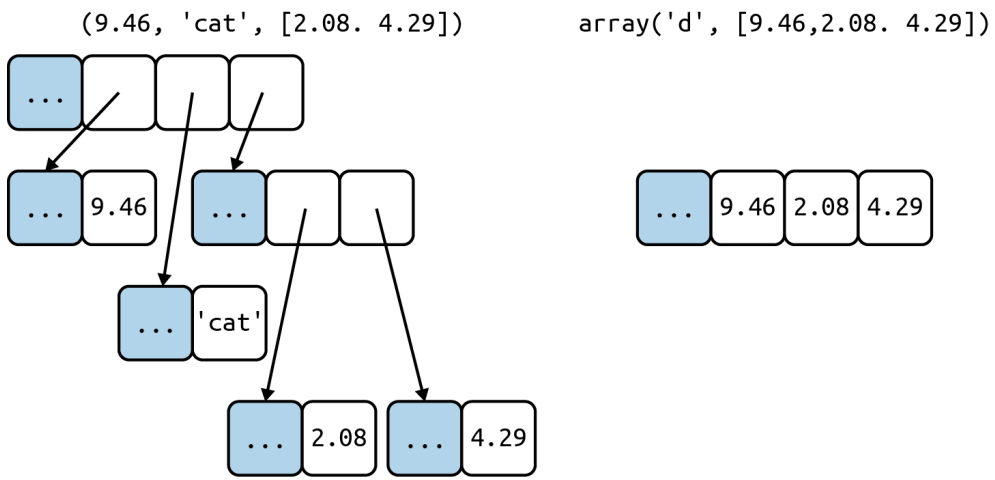
Operator category	Symbols	Method names
Unary numeric	`- + abs()`	`__neg__ __pos__ __abs__`
Rich comparison	`< <= != > >= ==`	`__lt__ __le__ __eq__ __ne__ __gt__ __ge__`
Arithmetic	`+ - * / // \% @ divmod() round() ** pow()`	`__add__ __sub__ __mul__ __truediv__ __floord __matmul__ __divmod__ __round__ __pow__`
Reversed arithmetic	(arithmetic operators with swapped operands)	`__radd__ __rsub__ __rmul__ __rtruediv__ __rf __rmod__ __rmatmul__ __rdivmod__ __rpow__`
Augmented assignment arithmetic	`+= -= *= /= //= \%= @= **=`	`__iadd__ __isub__ __imul__ __itruediv__ __if __imod__ __imatmul__ __ipow__`
Bitwise	`& | ^ << >> \~`	`__and__ __or__ __xor__ __lshift__ __rshift__`

Méthodes spéciales for opérateurs infixes

Structures de données / Séquences

La bibliothèque standard Python contient plusieurs types de séquences. On peut les diviser en deux catégories :

- Les **conteneurs**: Peuvent contenir des éléments de différents types, comme d'autres conteneurs. Les conteneurs gardent une référence des objets qu'ils contiennent. Exemples : `list`, `tuple`, `collections.deque`.
- Les **séquences plates**: Peuvent contenir des éléments d'un type simple. Les séquences plates gardent les 'valeurs' des éléments qu'elles continent de façon contiguë en mémoire dans la séquence elle-même (Pas de référence vers d'autres objets). Exemples: `str`, `bytes`, `array.array`



Organisation en mémoire des données d'un tuple et un 'array'

On peut aussi grouper les séquences en fonction de leur mutabilité :

- Les séquences **mutables**: Exemples : `list`, `bytearray`, `array.array`, `collections.deque`.
- Les séquences **immuables**: Exemples: `tuple`, `str`, `bytes`

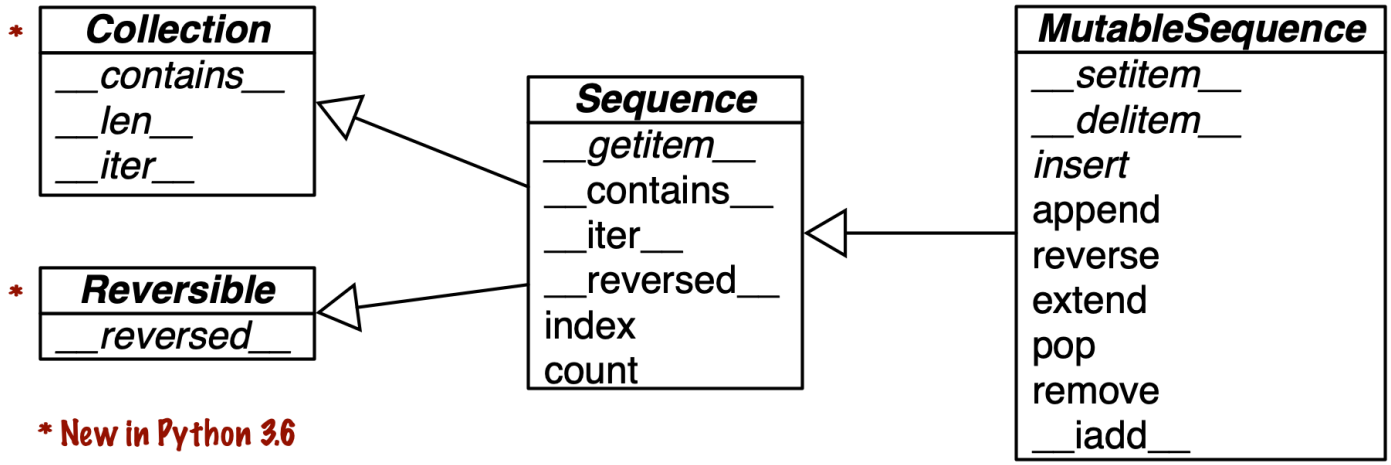


Diagramme de classes de certaines classes de `collections.abc`

Listes en compréhension et générateurs.

- Les listes en compréhension permettent de créer des listes avec une syntaxe concise, explicite, et sont souvent plus rapides que l'usage d'une boucle pour la création d'une liste. E.g.
  ```
  [i**2 for x in range(10) if x > 2]
  ```
- Les générateurs permettent de 'générer' des éléments d'une séquence un à un sans avoir à créer et à stocker toute la séquence en mémoire. Ils peuvent par exemple être utilisés pour la création de séquences autre que des listes. La syntaxe pour créer un générateur est similaire à celle des listes en compréhension, à l'exception qu'on utilise des parenthèses. E.g. `(i**2 for x in range(10) if x > 2)`.
  C'est juste une introduction aux générateurs, nous y reviendrons plus en détails.
- Nous avons aussi les ensembles et les dictionnaires en compréhension ('set comprehension' and 'dict comprehension').

Les tuples

- Les tuples sont des **listes immuables** et peuvent aussi être utilisés comme des enregistrements de données structurées.
- Lorsqu'ils sont utilisés en tant que listes immuables, l'immuabilité est définie sur la référence des objets du tuple. Les références ne peuvent être modifiées, mais si l'une d'elles pointe sur un objet mutable et cet objet est modifié, la valeur du tuple change aussi. Si le tuple contient une liste par exemple et la liste est modifiée, la valeur du tuple change aussi. Cependant, les tuples avec des objets mutables peuvent etre source de bugs. Si le tuple est utilisé par une fonction de **hachage** dans un contexte donné (e.g. dans un `set`, implicitement). Les fonctions de hachage ne fonctionnement que sur les objets immuables.
- Les tuples utilisent moins d'espace mémoire que les listes de la même taille et peuvent être optimisés par l'interpréteur Python.

L'unpacking L'unpacking permet de déballer les éléments d'un **itérable**. L'opérateur '*' peut être utilisé pour capturer tous les éléments restants en fonction du contexte d'usage.

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates  # unpacking
>>> latitude, longitude = longitude, latitude # swapping
>>> t = (20, 8)
>>> quotient, remainder = divmod(*t)
>>> a, b, *rest = range(5)
>>> func(*[1, 2], 3, *range(4, 7))
>>> *range(4), 4
...
```

Note: Les slides seront beaucoup plus compacts et minimalistes dorénavant, et serviront juste à annoncer les notions. Les notions seront évidemment expliquées durant le cours et les sessions de codage. L'une des raisons majeures, c'est pour permettre de passer plus de temps à pratiquer au fur et à mesure.

J'avais aussi la flemme de trop écrire dans les slides sachant que tout sera dument expliqué durant le cours de toute façon.

Autres concepts sur les séquences

- Le slicing: Utilisé pour d'extraire ou de modifier des éléments d'une séquence.
- Usage des opérateurs `+` et `*` avec les séquences.
- Quelques autres séquences et leurs pros/cons: `array.array`, `deque`, `set`, `memoryview`

Les Dictionnaires et Sets en Python

- Syntaxe moderne pour créer et gérer les dictionnaires et les mappings. Le 'dict comprehension', le 'dictionary unpacking', la fusions des types 'mappings', et le pattern matching sur les 'mappings'.
- L'importance des objets 'hachables' avec les types 'mapping'
- Méthodes courantes des types 'mapping'
- Traitement spécial des clés manquantes avec `defaultdict` vs `__missing__`.
- Les variantes du type `dict` dans la bibliothèque standard (e.g. `OrderedDict` `ChainMap`, `Counter`, `UserDict`)
- Les 'mappings' immuables avec `MappingProxyType`
- Les 'dictionary views'

Les Dictionnaires et Sets en Python - II

- Les types `set` et `frozenset`
- Les implications des tables de hachages dans l'usage des sets et les dictionnaires
- L'usage d'opération des 'set' sur les 'dict views'

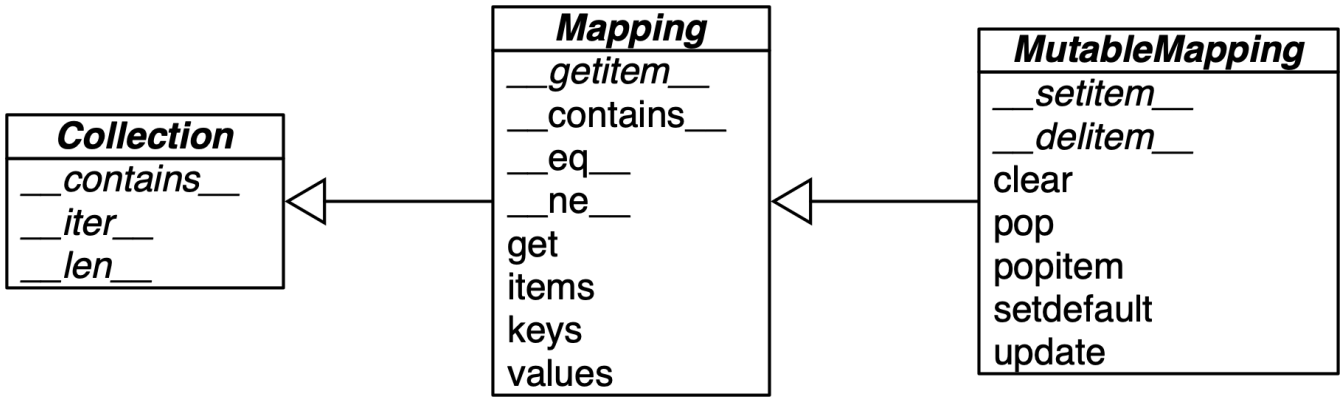


Diagramme des superclasses des types mappings

Les data classes

- Conçus pour créer des classes simples qui n'ont qu'une collection de champs, avec peu ou pas de fonctionnalités supplémentaires.
- `collections.namedtuple`
- `typing.NamedTuple`
- `@dataclasses.dataclass`. ('Fields options')
- Introduction aux 'Type hints'
- *Note: Les data classes ne devraient pas être abusées, et doivent être évité pour des classes complexes*

Références, Mutabilité et Recyclage

- Les variables en Python (étiquettes vs boîtes).
- Identité, égalité et aliases (== vs `is`)
- L'immutabilité relative des `tuple`
- Les copies sont superficielles 'shallow' par défaut
- Paramètres de fonctions par référence
- `del` et le garbage collection

Les fonctions sont des objets de premières classes en Python

Un objet de première classe est souvent défini comme une entité d'un programme qui peut

- Être créé
- Créé a l'exécution
- Attribué à une variable ou à un élément dans une structure de données
- Passé en argument à une fonction
- Renvoyé comme résultat d'une fonction
- Comme exemple, nous avons les entiers, les dictionnaires, et les ... fonctions.

Les fonctions en Python

- Les fonctions d'ordre supérieur
- Les fonctions anonymes
- Les 'Callable'
- Paramètres positionnels et 'keyword argument'.
- Quelques fonctions natives utiles (`functools.partial` , etc...)

Fonctions / Annotations, décorateurs, closures

Les annotations en Python : L'objectif est d'aider les outils de développement et les devs à trouver des bogues dans du code Python via une analyse statique, sans réellement exécuter le code. Surtout pour les projets de taille non négligeable. Les annotations sont progressives en Python (optionnelles et n'influencent pas la performance des programmes).

Les décorateurs et les closures : Les décorateurs de fonctions permettent de « marquer » les fonctions pour modifier leur comportement d'une manière ou d'une autre. Un 'closure' est une fonction qui conserve des variables libres afin qu'elles puissent être utilisées ultérieurement lorsque la portée de définition des variables n'est plus disponible.

Pratique: Portée des variables (`global` , `local` , `nonlocal`), les variables libres, Ordre de recherche des variables dans les fonctions, exemples de décorateurs (`functools.wraps` , `functools.cache` , `classmethod` , `staticmethod`)

Agenda

1. Objectifs du cours

2. Python avancé
   - Structures de données
   - Les fonctions comme objets
   - Classes et Protocoles
   - Structures de controle
   - Outils IA en Python

3. Introduction à la programmation en R