Mémo Python

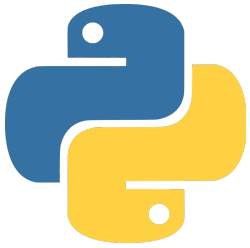


Table des matières

Avant de commencer (sauf pour les impatients)

permalien

Je rédige ce pdf au fur et à mesure que j'en apprends sur Python… Bonne lecture !

Le document considère que le lecteur a un niveau « prépa » en Python : la définition de fonctions, les boucles, les conditions et la manipulation des listes est supposée connue. La première partie concerne la programmation orientée objet et le modèle de données de Python. Les deux parties suivantes concernent des modules de la bibliothèque standard et des modules communautaires. Il y a de nombreux liens en bleu, ne pas hésiter à cliquer dessus ! Les liens « Plus d'informations » pointent vers les sources utilisées.

Sauf exception, on considère qu'on travaille sur **Python 3.7** et sur une distribution GNU/Linux (par exemple : Linux Mint, Fedora, Solus, Debian, Arch Linux).

Ce document est écrit au format TeX puis compilé avec XeLaTeX (et minted pour les sections de code) tantôt sur Windows, tantôt sur Fedora (ça ne change pas le résultat). Il est disponible au téléchargement (pour mettre le document à jour) en cliquant sur ce lien : python.pycolore.fr.

Notation Toutes les fonctions sont notées avec des parenthèses : `fonction()`.

Notation Pour la documentation des méthodes :

- on note `Classe.methode(self, *args, **kwargs)` (avec le nom de classe en `PascalCase`) ;
- on note `obj.methode(*args, **kwargs)` (avec le nom de l'objet en `snake_case`).

Auteur Guillaume Fayard.

Code source GitHub.

Première partie

Introduction à la programmation orientée objet en Python

Dans cette partie, nous allons modéliser un monde peuplé de fourmis.

1 Objets

Tout est objet en Python, des nombres aux listes en passant par les modules et les exceptions. Tout objet possède un type, un autre objet qui est responsable de sa création. On peut le récupérer grâce à la fonction native `type()` :

```
>>> type(1)
<class 'int'>
>>> type([1, 2, 3])
<class 'list'>
>>> import os # on importe un module
>>> type(os) # et on lui demande son type
<class 'module'>
```

On peut demander le type de leur type :

```
>>> type(type(1))
<class 'type'>
>>> type(type([1, 2, 3]))
<class 'type'>
>>> type(type(os))
<class 'type'>
```

Nativement, tous les types sont créés par `type`. On peut créer nos objets nous-mêmes grâce à `type()` : il suffit de lui donner un nom de type, un tuple contenant ses parents et un dictionnaire d'attributs.

Commençons par créer un objet élémentaire sans attribut ni parent.

```
>>> Fourmi = type("Fourmi", (), {})
>>> Fourmi
<class '__main__.Fourmi'>
```

On vient de créer notre premier type d'objet, il s'agit d'une classe. Les objets créés à partir d'une classe sont appelés instances. Créons maintenant notre première instance.

```
>>> fourmi = Fourmi() # on appelle la classe pour l'instancier
>>> fourmi
<__main__.Fourmi object at 0x7f593a678550>
>>> type(fourmi)
<class '__main__.Fourmi'>
>>> fourmi.role = "ouvrière" # on peut lui donner un attribut
>>> fourmi.role # puis y accéder
'ouvrière'
>>> vars(fourmi) # on peut aussi demander tous les attributs qu'il possède
{'role': 'ouvrière'}
```

En pratique, on ne crée pas nos classes avec la fonction `type()`, ce n'est pas très pratique si on veut que notre classe ait un comportement plus complexe. En effet, une classe peut posséder des méthodes, des propriétés, que l'on préfère écrire en utilisant la définition de classes.

2 Classes

2.1 Structure d'une classe

Les classes permettent des créer des objets appelés instances qui partagent des caractéristiques communes. Une classe est en fait un gabarit qui nous permet de créer un certain type d'objets. Si on réécrit notre exemple précédent de max, ça donne ça :

```python
class Fourmi:
    pass # car la classe est vide
```

Il n'y a pas grand chose dedans pour l'instant. Mais ça s'utilise pareil.

Les objets instanciés par une classe partagent des caractéristiques communes à la classe :

1. des attributs, des variables propres aux instances ;
2. des méthodes, des fonctions propres aux instances et qui agissent par exemple sur leurs attributs.

Les classes sont des gabarits qui permettent de créer un même type d'objet.

2.1.1 Attributs

Nous allons créer la classe représentant le monde dans lequel vont évoluer les fourmis. On considère qu'il s'agit d'une grille ayant une certaine hauteur et une certaine largeur :

```python
class Monde:
    hauteur = 32
    largeur = 32
```

Les variables `hauteur` et `largeur` sont appelées attributs de classe. Chaque objet de cette classe y aura accès :

```
>>> monde1 = Monde()
>>> monde1.hauteur
32
>>> monde2 = Monde()
>>> monde2.largeur
32
```

Nos mondes ont une largeur et une hauteur, mais comme ce sont des attributs de classe, ils ont tous la même taille ; ce serait plus intéressant de créer des mondes de taille différentes. On peut parfaitement surcharger les attributs de classe pour en faire des attributs d'instance :

```
>>> monde1.largeur = 64
>>> vars(monde1)
{'largeur': 64}
```

On remarque que les attributs de classe ne sont pas renvoyées par `vars()`. Cela est dû au fait que les attributs de classe sont gardés uniquement dans la classe ; ainsi une modification d'attribut de classe impactera toutes les instances.

2.1.2 Initialisation

Maintenant, on veut pouvoir initialiser automatiquement des variables. Définir la taille des mondes après leur création n'est pas gênant pour l'instant, mais quand l'initialisation comprend de nombreux attributs, cela devient fastidieux. On crée pour cela une fonction dans la classe appelée initialiseur.

```python
class Monde:
    def __init__(self, hauteur, largeur):
        """Initialiseur de la classe Monde."""
        self.hauteur = hauteur
        self.largeur = largeur
```

On peut alors passer directement la hauteur et la largeur lors de l'appel de la classe pour l'instanciation. Ces arguments sont automatiquement passés à `__init__()` :

```
>>> monde1 = Monde(32, 32)
>>> monde2 = Monde(largeur=64, hauteur=128)
>>> vars(monde1)
{'hauteur': 32, 'largeur': 32}
>>> vars(monde2)
{'hauteur': 128, 'largeur': 64}
```

2.1.3 Méthodes

Les fonctions définies dans les classes sont appelées méthodes, c'est le cas de l'initialiseur `__init__()`. On peut en définir d'autres pour implémenter des comportements aux instances.

```
class Fourmi:
    def __init__(self, role, x, y):
        self.role = role
        self.x = x
        self.y = y

    def move(self, x, y):
        self.x += x
        self.y += y
```

Une méthode s'utilise comme ceci :

```
>>> fourmi = Fourmi('ouvrière', 0, 0)
>>> fourmi.move(1, 1)
>>> vars(fourmi)
{'role: 'ouvrière', 'x': 1, 'y': 1}
```

Lorsque l'on évalue une méthode sur une instance, Python lui passe automatiquement en premier paramètre l'instance en question. Par convention on nomme donc toujours le premier paramètre des méthodes `self` qui fait référence à l'instance en cours.

2.1.4 Encapsulation : les propriétés

On a défini une classe Monde et une classe Foumi qui peut se déplacer. Maintenant, on veut que les fourmis ne puissent pas sortir du monde. On serait tenté d'utiliser des getters et setters (ou accesseurs et mutateurs) :

```
class Fourmi:
    def __init__(self, role, x, y, monde):
        self.role = role
        self.set_x(x)
        self.set_y(y)
        self.monde = monde

    def set_x(self, x):
        if x > self.monde.largeur:
            raise ValueError("{} est trop grand.".format(x))
        self.x = x

    def set_y(self, y):
        if y > self.monde.hauteur:
            raise ValueError("{} est trop grand.".format(y))
        self.y = y

    def move(self, x, y):
        self.set_x(x)
        self.set_y(y)
```

Python possède un mécanisme d'encapsulation sympa qui s'appelle les propriétés, elles permettent d'avoir le même genre de comportement, mais de manière transparente car la modification de valeurs d'attributs garde la même syntaxe :

```
class Fourmi:
    def __init__(self, role, x, y, monde):
        self.role = role
        self.monde = monde
        self.x = x
        self.y = y

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        if not 0 <= value < self.monde.largeur:
            raise ValueError("{} n'est pas compris entre 0 et {}.".format(value, self.monde.largeur))
```

```
        self._x = value

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, value):
        if not 0 <= value < self.monde.hauteur:
            raise ValueError("{} n'est pas compris entre 0 et {}.".format(value, self.monde.hauteur))
        self._y = value

    def move(self, x, y):
        self.x = x
        self.y = y
```

Une mauvaise affectation de `x` ou `y` engendrera une erreur :

```
>>> monde = Monde(32, 32)
>>> fourmi = Fourmi('ouvrière', 0, 0, monde)
>>> fourmi.x = -1
File "<stdin>", line 1, in <module>
    fourmi.x = -1
File "<stdin", line 15, in x
    raise ValueError("{} est trop grand.".format(value))
ValueError: -1 n'est pas compris en tre 0 et 32.
```

On utilise ici les propriétés pour l'encapsulation d'attributs. Mais elles sont également utiles pour des attributs calculés :

```
class Fourmi:
    # contenu précédent

    @property
    def distance_origine(self):
        return math.sqrt(self.x**2 + self.y**2)
```

On fait appel à cette propriété comme à un attribut classique :

```
>>> fourmi.move(3, 4)
>>> fourmi.distance_origine
5.0
```

2.1.5 Héritage

On peut créer des fourmis et leur associer un rôle. Cependant, un rôle différent engendrera un comportement différent (donc des méthodes différentes). Pour illustrer cela, on peut utiliser l'héritage :

```
class Fourmi:
    def __init__(self, x, y, monde):
        self.x = x
        self.y = y
        self.monde = monde

    # le reste de la classe fourmi précédente


class Ouvriere(Fourmi):
    role = "ouvrière"


class Reine(Fourmi):
    role = "reine"
```

Ici, les nouvelles classes `Ouvriere` et `Reine` héritent de la classe `Fourmi` : elles héritent donc de tout le contenu de cette dernière. Autrement dit, tout ce qui est défini dans la classe `Fourmi` l'est aussi pour `Ouvriere` et `Reine`. Comme on sait que le rôle sera le même pour les fourmis instanciées par une même classe, on peut en faire un attribut de classe.

Deuxième partie

Programmation orientée objet en Python

1 Classes

Les classes permettent des créer des objets appelés instances qui partagent des caractéristiques communes. Une classe est en fait un gabarit qui nous permet de créer un certain type d'objets.

Plus d'informations Introduction OpenClassrooms, Documentation Python 3, Wikilivres

1.1 Structure d'une classe

Les objets d'une classe partagent des caractéristiques communes à la classe :

1. des attributs, des variables propres à aux instances ;
2. des méthodes, des fonctions propres aux instances et qui agissent par exemple sur leurs attributs.

1.1.1 Définition d'une classe et création d'objets

Pour définir une classe, la syntaxe est la suivante :

```
class MaClasse:
    pass
```

Lorsqu'on lance l'exécution d'un module Python, l'interpréteur exécute le contenu du corps de toutes les classes qu'il rencontre pour la première fois. Ainsi si l'on exécute :

```
class Foo:
    print('bar')
```

La console affichera `bar`. Pour créer une instance de classe, il faut appeler la classe. Appeler une classe revient à appeler son constructeur, composé d'un créateur d'objet et d'un initialiseur. En reprenant la classe `Foo` :

```
>>> Foo # évaluation de la classe Foo
<class '__main__.Foo'>
>>> Foo() # création d'un objet Foo
<__main__.Foo object at 0x7f193dd99630>
```

1.1.2 Attributs

Créons un objet :

```
>>> foo = Foo()
```

Cet objet est pour l'instant vide, on peut lui donner un attribut :

```
>>> foo.attr = 1
>>> foo.attr
1
```

Les informations concernant les attributs d'une instance sont stockées dans le dictionnaire de l'instance, un attribut spécial nommé `__dict__`.

```
>>> foo.__dict__
{'attr': 1}
```

Lorsqu'on accède un attribut avec la syntaxe `foo.attr`, cette syntaxe est par défaut équivalente à :

```
>>> foo.__dict__['attr']
1
```

Pour modifier la valeur d'un attribut, on lui assigne tout simplement une nouvelle valeur :

```
>>> foo.attr = 456
>>> foo.attr
456
>>> foo.__dict__['attr'] = 789 # équivalent
>>> foo.attr
789
```

Une classe peut également définir des attributs de classe :

```
class Foo:
    class_attr = "I'm a class attribute."
```

Toutes les instances y ont accès :

```
>>> Foo.class_attr
"I'm a class attribute."
>>> foo = Foo()
>>> foo.class_attr
"I'm a class attribute."
```

Pourtant, cet attribut n'est pas dans `foo.__dict__`. En effet, lorsqu'on accède à un attribut, Python va le rechercher dans le dictionnaire de l'instance, mais aussi de sa classe s'il ne l'a pas trouvé. Ainsi :

— Modifier un attribut de classe pour une instance ajoutera une entrée dans son dictionnaire (on n'accède ainsi par la suite plus à l'attribut de classe mais au nouvel attribut d'instance, on peut dire qu'on l'a surchargé).

— Modifier un attribut de classe via la classe le modifie pour toutes les instances qui ne l'ont pas surchargé, ainsi que pour toutes les instances qui seront créées ensuite.

```
>>> foo.class_attr = 'New value'
>>> objet.__dict__
{'class_attr': 'New value'}
>>> bar = Foo()
>>> bar.__dict__
{}
>>> Foo.class_attr = "I just got a new value."
>>> bar.class_attr
"I just got a new value."
>>> foo.class_attr
'New value'
```

1.1.3 Méthodes

Les méthodes se définissent comme des fonctions dans le corps de la classe, elles agissent en général sur les instances de la classe. Python leur passe *toujours* l'instance sur laquelle elles sont appliquées en premier paramètre. Par convention, il est noté `self`.

```
class MaClasse:

    def methode(self, arg1, arg2):
        print(locals())
```

Ensuite on les appelle de la manière suivante :

```
>>> objet = MaClasse()
>>> objet
<__main__.MaClasse object at 0x7f13337e50f0>
>>> objet.methode(arg1, arg2)
{'self': <__main__.MaClasse object at 0x7f13337e50f0>, 'arg1': 123, 'arg2': 'ABC'}
```

C'est grâce à cette variable `self` que l'on peut agir sur l'instance.

1.1.4 Initialiseur

L'initialiseur est une méthode spéciale appelée `__init__()`, il est appelé lorsqu'une instance vient d'être créée et permet d'en initialiser les attributs. L'exemple suivant permet d'initialiser deux attributs :

```
class MaClasse:

    def __init__(self, att1, att2):
        """Initialiseur"""
        self.attribut1 = att1
        self.attribut2 = att2
```

Ces deux attributs sont initialisés lorsqu'on crée un nouvel objet. Les valeurs initiales des attributs sont automatiquement passés à `__init__()` lorsqu'on appelle la classe pour instancier :

```
>>> objet = MaClasse(123, 'ABC')
>>> objet.__dict__
{'attribut1': 123, 'attribut2': 'ABC'}
```

1.2 Héritage

1.2.1 Principe

L'héritage est un moyen de créer des classes dérivées (classes filles) d'une classe de base (classe mère). Une classe fille hérite de toutes les méthodes et attributs de sa classe mère. Pour indiquer les classes parentes d'une nouvelle classe, on les indique en paramètres lors de sa définition. L'héritage en action dans un exemple on ne peut plus simple :

```
>>> class Mere:
...     attr = 1
...
>>> class Fille(Mere):
...     pass
...
>>> Fille().attr
1
```

Il est possible de surcharger (d'écraser) une méthode héritée en la redéfinissant dans la classe fille. Si on veut accéder à une méthode héritée alors qu'on l'a redéfinie dans la classe fille, on utilise la fonction `super()` qui permet d'appeler la méthode de la classe mère de la classe présente (sans l'argument `self`).

Exemple

```
class Meuble:
    def __init__(self, couleur, materiau):
        self.couleur = couleur
        self.materiau = materiau

class Bibliotheque(Meuble):
    def __init__(self, couleur, materiau, n):
        super().__init__(couleur, materiau)
        self.nb_livres = n
```

On peut utiliser deux fonctions pour vérifier l'héritage : `isinstance` renvoie `True` si l'objet est une instance de la classe ou de ses classes filles ; `issubclass` permet de voir si une classe est fille d'une autre.

```
>>> bibli = Bibliotheque('blanc', 'vert', 150)
>>> bibli.__dict__
{'couleur': 'blanc', 'materiau': 'vert', 'nb_livres': 150}
>>> isinstance(bibli, Meuble)
True
>>> isinstance(bibli, Bibliotheque)
True
>>> issubclass(Bibliotheque, Meuble)
True
>>> issubclass(Meuble, Bibliotheque)
False
>>> isinstance(bibli, int)
False
>>> isinstance(bibli, object)
True
```

Plus d'informations OpenClassrooms, Documentation Python 3, Programiz

1.2.2 Ordre de résolution de méthode

1.2.3 Classe mère `object`

On a dit précédemment que le constructeur était composé d'un initialiseur et d'un créateur d'instance ; cependant l'exemple ne définissait pas de créateur d'instance : c'est parce qu'il est défini dans une classe `object`. Toutes les classes en Python 3 héritent implicitement de cette classe. Elle définit de nombreuses méthodes, notamment les méthodes dites spéciales, que l'on peut surcharger pour les personnaliser.

```
>>> object
<class 'object'>
>>> help(object)
Help on class object in module builtins:

class object
 |  The most base type

>>> class Foo:
...     pass
...
>>> issubclass(Foo, object)
True
```

1.3 Propriétés

Les propriétés représentent en Python le principe d'encapsulation. Elles sont utiles si on souhaite contrôler l'accès à un attribut ou si on veut que le changement d'une valeur d'un attribut engendre des modifications sur d'autres attributs. Du côté utilisateur, ce mécanisme est complètement transparent car il permet de garder la syntaxe classique `inst.attr = valeur`. Les propriétés sont un cas particulier des descripteurs.

On crée les propriétés en utilisant des décorateurs. Elles contiennent un accesseur, un mutateur, un destructeur et une aide (docstring de l'accesseur). Dans certains cas, il n'est pas nécessaire d'avoir un attribut associé, un simple calcul suffit :

```
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    @property
    def fahrenheit(self):
        """Propriété 'fahrenheit'."""
        return self.celsius * 1.8 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) / 1.8
```

Ainsi, on peut écrire :

```
>>> temp = Temperature(20)
>>> temp.fahrenheit
68.0
>>> temp.fahrenheit = 69
>>> temp.celsius
20.555555555555554
```

Parfois, il est nécessaire d'ajouter des attributs « cachés », par exemple si l'on veut aussi contrôler le changement de température en degrés Celsius, pour éviter une récursivité infinie :

```
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    @property
    def celsius(self):
        """Propriété 'celsius'.

        Vérifie si la température est supérieure à -273°C avant d'assigner."""
```

```
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273:
            raise ValueError("Une température en degrés Celsius doit être supérieure à -273°C.")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Propriété 'fahrenheit'."""
        return self.celsius * 1.8 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) / 1.8
```

Dans d'autres cas, on peut stocker le résultat de calculs dans un attribut caché. Ici, le calcul des degrés Fahrenheit est rapide, mais il peut s'avérer utile de stocker le résultat pour ne pas avoir à recalculer à chaque fois.

On utilise la propriété de la manière suivante :

```
>>> help(Temperature.celsius)
Help on property:

    Propriété 'celsius'.

    Vérifie si la température est supérieure à -273°C avant d'assigner.

>>> temp.celsius = -300
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../*.py", line 18, in celsius
    raise ValueError("Une température en degrés Celsius doit être supérieure à -273°C.")
ValueError: Une température en degrés Celsius doit être supérieure à -273°C.
>>> temp.fahrenheit = -462 # ça marche aussi car cette propriété fait appel à celle des celsius !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../*.py", line 18, in celsius
    raise ValueError("Une température en degrés Celsius doit être supérieure à -273°C.")
ValueError: Une température en degrés Celsius doit être supérieure à -273°C.
```

Plus d'informations Documentation Python 3, Priorités entre propriété et méthodes spéciales

1.4 Méthodes statiques et méthodes de classes

1.4.1 Méthode statique

Les méthodes que l'on a vues jusqu'à maintenant agissent sur les instances des classes : elles prennent toujours en premier argument le mot clé `self` qui correspond à l'instance elle même. Lorsque l'on appelle une telle méthode sur une instance comme ceci :

```
instance.methode(*args, **kwargs)
```

Python exécute en fait :

```
type(instance).methode(instance, *args, **kwargs)
```

Évaluer `type()` sur un objet renvoie sa classe.

En fait, ces deux objets sont différents. `Classe.methode` est une simple fonction, alors que `instance.methode` est une méthode partiellement évaluée sur l'instance (méthode liée, en anglais « bound method »), c'est-à-dire que l'instance est mise en premier argument.

Parfois, on écrit des méthodes qui n'ont pas d'incidence sur les instances de la classe. Imaginons une classe `Maths` qui regrouperait des opérations basiques :

```
class Maths:
    def addition(x, y):
        return x + y
```

```
    def multiplication(x, y):
        return x * y

    def division(x, y):
        return x / y
```

```
>>> Math.addition(1, 3)
4
```

Jusqu'ici tout va bien. Mais imaginons que la classe se développe et que l'on instancie des objets `Maths`. On aura un soucis :

```
>>> Maths().addition(1, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: addition() takes 2 positional arguments but 3 were given
```

Le problème étant que Python a en fait exécuté :

```
Maths.addition(Maths(), x, y)
```

Pour remédier à cela, il existe le décorateur `@staticmethod`. On doit écrire alors :

```
class Maths:
    @staticmethod
    def addition(x, y):
        return x + y

    @staticmethod
    def multiplication(x, y):
        return x * y

    @staticmethod
    def division(x, y):
        return x / y
```

On peut ainsi écrire sans crainte :

```
>>> Maths().addition(1, 3)
4
```

1.4.2 Méthode de classe

Parfois, on veut pouvoir agir sur la classe et non sur l'instance. Dans ce cas, la méthode de classe prend en premier paramètre `cls` (la classe) au lieu de `self` (l'instance).

```
>>> class Foo:
...     CONSTANT = "Bar"
...     def print_constant(cls):
...         print(cls.CONSTANT)
...
>>> Foo().print_constant()
Bar
```

Deux problèmes surviennent. Tout d'abord, même si le premier paramètre s'appelle `cls`, c'est encore l'instance qui est mise en paramètre.

```
>>> foo = Foo()
>>> foo.CONSTANT = "Baz"
>>> foo.print_constant()
Baz # On veut Bar !
```

Deuxième problème : on ne peut pas appeler la méthode de classe sur la classe (même problème que pour les méthodes statiques) :

```
>>> Foo.print_constant()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print_constant() missing 1 required positional argument: 'cls'
```

Pour remédier à cela, on utilise le décorateur `@classmethod`.

```
>>> class Foo:
...     CONSTANT = "Bar"
...     @classmethod
...     def print_constant(cls):
...         print(cls.CONSTANT)
...
>>> foo = Foo()
>>> foo.CONSTANT = "Baz"
>>> foo.print_constant()
Bar # Youpi !
>>> Foo.print_constant()
Bar # Youyoupi !
```

1.4.3 Cas de l'héritage

En résumé :

1. Les méthodes statiques sont des fonctions reliées à des classes, mais qui n'agissent pas sur celles-ci.
2. Les méthodes de classe sont des fonctions qui prennent la classe en paramètre.

Une classe qui hérite d'une classe mère hérite de toutes les méthodes de celle-ci. Les méthodes statiques restent donc inchangées, tandis que les méthodes de classe s'adaptent à la nouvelle classe, car elles la prennent en premier argument.

Exemple Un exemple d'utilisation de méthodes statiques et de classe sont la création de constructeurs alternatifs. On s'aperçoit de la différence des deux notions.

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    @staticmethod
    def par_date_de_naissance(nom, date):
        return Personne(nom, 2018-date)

    @classmethod
    def par_date_de_naissance2(cls, nom, date):
        return cls(nom, 2018-date)

class Homme(Personne):
    sexe = 'homme'
```

```
>>> homme1 = Homme.par_date_de_naissance('Jean', 1997)
>>> homme2 = Homme.par_date_de_naissance2('Jean', 1997)
>>> type(homme1)
<class '__main__.Personne'>
>>> type(homme2)
<class '__main__.Homme'>
```

Pour avoir `homme1` de type `Homme`, il faut redéfinir la méthode statique dans la classe fille.

Plus d'informations Méthode statique sur Programiz, Méthode de classe sur Programiz, StackOverflow

2 Méthodes spéciales

Python est un langage avec une syntaxe de haut niveau, c'est-à-dire facilement compréhensible par l'utilisateur humain. Derrière cette syntaxe se cachent des méthodes appelées méthodes spéciales. On les reconnaît par la présence de soulignés qui encadrent leur nom, c'est le cas de `__new__()` et `__init__()`.

Exemples

```
## Syntaxe de haut niveau   ## Méthode spéciale correspondante
# Listes
L = [1, 2, 3]
2 in L                     # L.__contains__(2)
len(L)                     # L.__len__()

# Opérations
objet1 + objet2            # objet1.__add__(objet2)
objet1 == objet2           # objet1.__eq__(objet2)

# Conversion en str ou affichage
print(objet)               # objet.__str__()
str(objet)                 # objet.__str__()

# Appel d'une fonction
fonction()                 # fonction.__call__() <- intéressant celui-là !
```

Plus d'informations Documentation Python 3, OpenClassroom

2.1 Création, initialisation et finalisation

Objet.__new__(cls[, *args, **kwargs])

Créateur de l'instance. C'est une méthode statique qui prend en premier paramètre obligatoire la classe de l'instance à créer. Les arguments suivants sont ceux passés à l'initialiseur. Cette méthode doit renvoyer une nouvelle instance ; L'initialiseur est alors appelé. En général, on n'implémente pas cette méthode, sauf dans certains cas particuliers.

Cette méthode statique est un cas particulier qui ne nécessite pas de décorateur `@staticmethod`.

Objet.__init__(self[, *args, **kwargs])

Initialiseur de l'instance. C'est ici qu'on initialise les attributs de l'instance.

Ces deux méthodes forment le constructeur de l'instance, elles sont appelées lorsqu'on appelle la classe pour construire un objet. Si on surcharge ces méthodes, il ne faut pas oublier d'appeler les méthodes héritées grâce à `super()`. Dans le cas où la classe hérite uniquement d'`object`, il n'y a pas besoin d'appeler `super().__init__()` car les instances d'`object` n'ont aucun attribut (donc il n'y a pas d'initialisation).

Exemple On veut définir une classe « singleton » qui ne peut créer qu'une instance.

```
class Singleton:
    """Classe qui ne peut instancier qu'une fois."""

    instance = None

    def __new__(cls, *args, **kwargs):
        if not cls.instance:
            cls.instance = super().__new__(cls) # object.__new__() ne prend que le type à instancier comme paramètre
            return cls.instance
        else:
            raise TypeError(f"Cette classe singleton possède déjà une instance : {cls.instance}")

    def __del__(self):
        Singleton.instance = None
```

Objet.__del__(self)

Finaliseur de l'instance. Cette méthode est appelée lorsqu'un objet est sur le point d'être détruit, mais n'est pas responsable de sa destruction. Lorsqu'on hérite uniquement d'`object`, il n'est pas nécessaire d'appeler `super().__del__()` car elle ne fait rien. La syntaxe

```
del variable
```

décrémente le nombre de références vers l'`objet` correspondant. Si celui-ci atteint zéro, alors `objet.__del__()` est appelée.

2.2 Représentation et chaîne de caractères d'un objet

Si on crée une instance de la classe `Singleton` précédente et qu'on demande sa représentation dans l'interpréteur, il nous renvoie quelque chose de pas très explicite :

```
>>> inst = Singleton()
>>> inst
<__main__.Singleton object at 0x000002B07DB73898>
```

On doit donc définir une méthode de représentation.

`Objet.__repr__(self)`

Appelée par `repr()`, ou bien lorsqu'on demande l'objet dans l'interpréteur. Cette méthode calcule et renvoie une chaîne de caractères compréhensible « canonique », c'est-à-dire qu'elle doit ressembler à une expression Python à partir de laquelle on doit pouvoir recréer un objet de la même valeur, c'est à dire telle que `repr(objet) == objet`. Si ce n'est pas possible, elle devrait renvoyer une description entre chevrons `"<description>"`.

Parfois, on veut quelque chose de plus joli destiné à un véritable affichage (quand on appelle `print()`). On peut vouloir aussi avoir la capacité de convertir un objet en une chaîne de caractère avec `str()`. On doit alors définir une autre méthode spéciale :

`Objet.__str__(self)`

Appelée par `str()`, `print()` et `format()`. Cette méthode renvoie une chaîne de caractère correspondant à la représentation informelle de l'objet. Si cette méthode n'est pas définie, alors `__repr__()` est utilisée à la place.

Exemple L'exemple suivant

```
class MaClasse:
    def __init__(self, attr):
         self.attribut = attr

    def __repr__(self):
        return "MaClasse(attribut={})".format(self.attribut)

    def __str__(self):
        return "Instance de MaClasse ayant comme attribut {}"
              .format(self.attribut)
```

permet de faire :

```
>>> obj = MaClasse("Exemple")
>>> obj
MaClasse(attribut=Exemple)
>>> print(obj)
Instance de MaClasse ayant comme attribut Exemple.
```

2.3 Accès et modification des attributs

On a vu précédemment les propriétés qui permettent une sorte d'encapsulation des attributs. Lorsqu'on veut accéder à un attribut par la syntaxe

```
instance.attr
```

Python appelle en premier une méthode spéciale.

`Objet.__getattribute__(self, name)`

Appelée en premier lorsque l'on veut accéder à un attribut par les syntaxes

```
objet.attr             # objet.__getattribute__('attr')
getattr(objet, 'attr') # objet.__getattribute__('attr')
```

Cette méthode doit renvoyer l'attribut `name` demandé s'il existe (ou calculer sa valeur) ou lever une exception `AttributeError` sinon. Dans ce cas, la méthode `__getattr__()` est appelée.

Cette méthode est définie dans la classe `object`, le mécanisme d'accès par défaut aux attributs est donc le suivant :

1. `object.__getattribute__()` commence par rechercher `name` sous forme de descipteur dans le dictionnaire `__dict__` de la classe de l'instance (et de ses classes parentes s'il ne trouve pas).

```
type(objet).__dict__[name].__get__(objet, type(objet))
```

2. `object.__getattribute__()` recherche ensuite `name` sous forme de simple variable dans le dictionnaire `__dict__` de l'instance, puis dans celui de sa classe si elle ne le trouve pas, ainsi que dans celui de chaque classe parente jusqu'à le trouver.

```
objet.__dict__[name]
```

3. Si `object.__getattribute__()` n'a pas trouvé `name` dans aucun `__dict__`, elle lève une exception `AttributeError`.

Surcharger cette méthode va donc modifier le mécanisme par défaut. Cependant, pour les attributs dont on ne veut pas modifier l'accès, il faut penser à appeler la méthode `__getattribute__()` d'`object` ou de la classe parente. Pour les attributs dont on veut modifier le comportement d'accès, il ne faut pas utiliser la syntaxe classique `objet.attr` car cela va créer une récursivité infinie, il faut avoir recours à `super().__getattribute__()` ou bien accéder directement aux descripteurs ou clé du dictionnaire.

`Objet.__getattr__(self, name)`

Appelée si `__getattribute__()` lève une exception `AttributeError`. Par défaut, cette méthode fait la même chose, mais c'est ici que l'on peut calculer des attributs dynamiques : des attributs qui ne sont pas initialisés mais dont on veut pouvoir calculer la valeur.

`Objet.__setattr__(self, name, value)`

Appelée lorsque l'on assigne une valeur à un attribut :

```
objet.attr = value            # objet.__setattr__('attr', value)
setattr(objet, 'attr', value) # objet.__setattr__('attr', value)
```

Cette méthode est définie dans la classe `object`, l'ordre de recherche de l'attribut à modifier est le suivant :

1. Si un descripteur est trouvé, il est utilisé pour modifier la valeur de `name`.

```
type(objet).__dict__[name].__set__(objet, value)
```

2. Sinon une nouvelle clé est créée dans le `__dict__` de l'instance avec pour valeur `value`.

```
objet.__dict__[name] = value
```

Surcharger cette méthode va donc modifier le mécanisme par défaut. Cependant, pour les attributs dont on ne veut pas modifier le comportement de modification, il faut penser à appeler la méthode `__setattr__()` d'`object` ou de la classe parente. Pour les attributs dont on veut surcharger le mécanisme de modification, il y a le même problème que pour `__getattribute__()`, attention de ne pas créer de récursivité infinie.

`Objet.__delattr__(self, name)`

Appelée lorsque l'on veut détruire un attribut :

```
del objet.attr         # objet.__delattr__('attr')
delattr(objet, 'attr') # objet.__delattr__('attr')
```

Finalise l'attribut avant sa suppression s'il existe et lève une exception `AttributeError` sinon. Cette méthode doit appeler `super().__delattr__()` pour éviter une récursivité infinie lors de l'appel à la suppression de l'attribut.

Par exemple, en reprenant la classe `Temperature` vue précédemment, on voudrait que l'attribut `_celsius` ne soit pas accessible (même par si convention, on n'utilise pas directement les attributs commençant par _ en Python) : cela casserait tout le contrôle que l'on essaie d'avoir ! Une solution à cela serait le recours à `__getattribute__()` et `__setattr__()`. Ci-dessous un exemple de classe qui possède une propriété qui utilise un attribut que l'on désire « privé » :

```
class ClassWithAPrivateAttribute:
    def __init__(self, attr):
        self.attr = attr

    # la propriété qui sera parfaitement accessible
    @property
    def attr(self):
        """Accès public à attr."""
        # on cherche directement dans __dict__ pour ne pas appeler __getattribute__()
        return self.__dict__['_attr']
```

```
    # son setter
    @attr.setter
    def attr(self, value):
        print("Calculs et autres logiques...")
        # idem, on modifie directement __dict__ pour ne pas appeler __setattr__()
        self.__dict__['_attr'] = value

    # c'est ici que l'on va restreindre l'accès à _attr
    def __getattribute__(self, name):
        """Bloque l'accès aux attributs préfixés de '_'."""
        if name[0] == '_' and name[1] != '_':
            raise AttributeError("'{}' est un attribut privé.".format(name))
        return super().__getattribute__(name)

    # et ici que l'on bloque les modifs
    def __setattr__(self, name, value):
        """Bloque la modification des attributs préfixés de '_'."""
        if name[0] == '_' and name[1] != '_':
            raise AttributeError("'{}' est un attribut privé.".format(name))
        super().__setattr__(name, value)
```

L'accès à `_attr` est restreint :

```
>>> instance = ClassWithAPrivateAttribute(123)
>>> instance.attr
123
>>> instance.attr = 456
'Calculs et autres logiques...'
>>> instance.attr
456
>>> instance._attr = 123 # tentative d'accès direct à l'attribut caché
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    instance._attr = 123
  File "....py", line 28, in __setattr__
    raise AttributeError("'{}' est un attribut privé.".format(name))
AttributeError: '_attr' est un attribut privé.
>>> instance.__dict__['_attr']
456
```

Cet exemple peut paraître incomplet puisqu'on peut encore modifier les attributs en passant par `__dict__`. Tout dépend à quel point on veut restreindre l'accès.

2.4 Surcharges d'opérateur

Les surcharges d'opérateur permettent de faire des opérations arithmétiques avec des objets, c'est-à-dire d'indiquer à Python ce qu'il faut faire lorsque l'on exécute `objet1 + objet2`. Ces méthodes prennent en arguments `self` (l'objet 1) et l'objet 2.

Méthode	Appel
`__add__()`	`objet1 + objet2`
`__sub__()`	`objet1 - objet2`
`__mul__()`	`objet1 * objet2`
`__truediv__()`	`objet1 / objet2`
`__floordiv__()`	`objet1 // objet2`
`__mod__()`	`objet1 \% objet2`

Les deux objets ne sont pas nécessairement du même type ! Cependant, cette opération n'est pas symétrique : le code `objet + 5` par exemple exécute `objet.__add__(5)`, alors que `5 + objet` exécute `int.__add__(5)`. Pour que l'opération soit symétrique, il faut aussi définir ces fonctions avec le préfixe `r` (par exemple `__radd__()`).

2.5 Duck typing

On appelle *duck typing* , en français « typage canard », le fait de reconnaître un type d'objets grâce à leurs méthodes et attributs. Cela est utile lorsque l'on veut qu'une fonction puisse prendre en paramètre une certaine catégorie d'objets qui ne sont pas forcément du même type ; ils partageront cependant des caractéristiques communes qui leur permettront d'être traités par la fonction.

Par exemple, on peut récupérer la longueur d'une liste avec `len()` :

```
>>> len([1, 2, 3])
3
```

Mais on peut récupérer la longueur de plein d'autres choses :

```
>>> len("abc")
3
>>> len((1, 2))
2
>>> len(range(5))
5
```

Tous ses objets sont de types différents : liste, chaîne de caractères, tuple, `range` ; ils partagent cependant quelque chose :

```
>>> for obj in [[1, 2, 3], "abc", (1, 2), range(5)]:
...     print(obj.__len__())
...
3
3
2
5
```

Ainsi, `len()` accepte tout objet possédant une taille, c'est-à-dire tout objet implémentant la méthode spéciale `__len__()`. Les méthodes spéciales permettent de cette manière d'implémenter une multitude de comportements aux objets et de les rendre compatibles avec des API Python : on peut très bien créer un objet sur lequel on peut itérer avec une boucle `for` en implémentant le protocole d'itération, mais aussi le rendre appelable comme une fonction grâce à la méthode `__call__()`.

Les parties suivantes décrivent ces catégories d'objets définies selon les méthodes spéciales implémentées. On appelle parfois protocole l'ensemble des méthodes spéciales à implémenter pour une catégorie (exemples : protocole d'itérateur ou protocole de descripteur).

3 Fonctions et objets *callable*

Un objet est dit *callable* (que l'on peut traduire par « appelable ») si lui appliquer des parenthèses déclenche quelque chose.

```
obj() # on appelle l'objet obj
obj(arg) # on lui passe un argument positionnel
obj(arg1, param=arg2) # on lui passe aussi un argument nommé
result = obj()
# En général appeler un objet retourne quelque chose. En fait, un callable renvoie toujours
# au moins None.
```

Plus d'informations

- les fonctions (documentation Python 3) ;
- déballage et callables (Sam & Max) ;
- fonctionnement interne des callables (StackOverflow).

3.1 Créer un objet *callable*

Pour qu'une classe puisse instancier des objets *callable*, il suffit qu'elle définisse une méthode `__call__()`.

`Callable.__call__(self[, *args, **kwargs])`

Cette méthode est appelée lorsqu'on appelle un objet. Tout objet possédant cette méthode est *callable*.

```
>>> class HelloCallable:
...     def __call__(self):
...         return "Hello"
...
>>> HelloCallable()() # () pour instancier puis () pour appeler
Hello
```

Les classes sont *callable* : on les appelle pour instancier.

```
>>> class Bar: pass
...
>>> hasattr(Bar, "__call__")
True
```

3.2 Les objets fonctions

Les fonctions et les méthodes sont des objets *callable*, elles sont créées avec le mot-clé `def`. On peut utiliser des annotations pour indiquer les types des paramètres et le type de la valeur renvoyée (les annotations peuvent être plus généralement n'importe quelle expression Python).

```
>>> def func(arg1: int = 1) -> int:
...     """Docstring."""
...     return arg1 # l'interpréteur s'arrête au premier return rencontré
```

Les fonctions sont des objets ; à ce titre, on peut parfaitement leur définir des attributs de fonction. Un exemple farfelu pour montrer que ça existe :

```
>>> def incrementor():
...     incrementor.n += 1
...     return incrementor.n
...
>>> incrementor.n = -1
>>> incrementor()
0
>>> incrementor()
1
```

Ces objets sont des instances de `function`, ils possèdent quelques attributs spéciaux :

```
>>> type(func)
<class 'function'>
>>> func.__name__
'func'
>>> func.__module__
'main'
>>> func.__annotations__
{'arg1': <class 'int'>, 'return': <class 'int'>}
>>> func.__defaults__
(1,)
>>> func.__doc__
'Docstring.'
```

Les méthodes ont en plus un attribut spécial `__self__` et un attribut spécial `__func__`. On peut aussi créer une fonction anonyme avec `lambda` :

```
>>> func = lambda x: x # fonction identité (syntaxe -> lambda <args>: <expression à renvoyer>)
>>> type(func)
<class 'function'>
>>> func.__name__
'<lambda>'
```

Ce mécanisme est utile si l'on veut créer une fonction à la volée, par exemple quand on utilise `map()`.

3.3 Gérer les paramètres d'une fonction

On peut assigner à certains paramètres une valeur par défaut :

```
>>> def function(arg1, arg2=5):
...     return arg1 + arg2
...
>>> function(1)
6
```

Les paramètres ayant une valeur par défaut doivent être placé *après* ceux qui n'en ont pas.

Remarque Les valeurs par défaut sont assignées lors de la définition de la fonction. Si l'objet associé est muable, la valeur par défaut restera toujours celle d'origine même si la fonction est appelée alors que l'objet associé a mué.

On peut passer deux types d'arguments à une fonction :

1. les arguments dits *positionnels* : ils sont interprétés par la fonction grâce à leur position dans la liste des arguments passés à la fonction ;
2. les arguments dits *nommés* : on spécifie le nom de leur paramètre quand on les passe à la fonction. On doit avoir passé tous les arguments positionnels avant de passer des arguments nommés.

On peut utiliser les opérateurs d'*unpacking* (« déballage ») :

— Lorsque l'on écrit les paramètres pour capter tous les paramètres possibles.

— Pour renseigner les arguments.

Cette fonction prend tous les arguments qui lui sont passés :

```
>>> def fonction(*args, **kwargs):
...     print(locals())
...
>>> fonction(1, 2, 3, 'A', 'B', 'C', kwarg1=4, kwarg2='D')
{'args': (1, 2, 3, 'A', 'B', 'C'), 'kwargs': {'kwarg1': 4, 'kwarg2': 'D'}}
```

`*args` va contenir tous les arguments positionnels dans un tuple et `**kwargs` tous les arguments nommés dans un dictionnaire. Cette syntaxe est utilisée quand on définit une fonction qui en enrobe une autre pour lui transmettre tous les arguments (voir les décorateurs).

On peut aussi utiliser le déballage lors d'appels de fonctions :

```
>>> def fonction(par1, par2, par3, par4):
...     print(locals())
...
>>> arg1, arg2, arg3, arg4 = 1, 2, 3, 4
>>> tuple_args = (arg1, arg2)
>>> dict_args = {'par3': arg3, 'par4': arg4}
>>> fonction(*tuple_args, **dict_args)
{'par1': 1, 'par2': 2, 'par3': 3, 'par4': 4}
```

Remarque Le déballage, ce n'est pas que pour les paramètres de fonctions :

```
>>> L = [1, 2, 3]
>>> a, b, c = L # on déballe L
>>> print(a, b, c)
1 2 3
>>> char1, char2 = "12" # ça fonctionne pour tout itérable
>>> print(char1, char2)
1 2
>>> a, b = (b, a)
>>> a, b = b, a # équivalent à la ligne précédente, l'assignation simultanée est en fait du déballage
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for k, v in d.items():
...     print(k, ":", v)
...
a : 1
b : 2
c : 3
```

4 Descripteurs

Les descripteurs sont une généralisation des propriétés (plus précisément, les propriétés sont des descripteurs), ce sont des objets qui doivent implémenter au moins une des méthodes spéciales suivantes :

`Descripteur.__get__(self, inst, owner)`

Appelée lorsqu'on essaie d'accéder à l'attribut :

```
inst.descripteur
```

`inst` est l'instance sur laquelle on essaie d'accéder à l'attribut et `owner` est le type de `inst`, la classe propriétaire du descripteur. Dans le cas particulier où l'on accède à un attribut par la classe :

```
Class.descripteur
```

`inst` vaut `None` et `owner` est la classe en question.

Cette méthode doit retourner l'attribut en question ou le calcul de sa valeur, ou bien lever une exception `AttributeError`.

`Descripteur.__set__(self, inst, value)`

Appelée lorsqu'on essaie de modifier la valeur d'un attribut d'`inst` de la classe propriétaire à `value` :

```
inst.descripteur = value
```

`Descripteur.__delete__(self, inst)`

Appelée lorsqu'on essaie de supprimer l'attribut de `inst` de la classe propriétaire :

```
del inst.descripteur
```

Les descripteurs sont utiles pour créer des « propriétés générales » qui n'ont pas besoin d'en savoir beaucoup sur les classes propriétaires des attributs et qui peuvent être utilisées par différents types d'objets.

Exemple Un descripteur d'entier borné

```
class BoundedIntDescriptor:
    """Entier borné.

    Vérifie que l'attribut est compris entre mini et maxi.
    """

    def __init__(self, mini, maxi, doc=None):
        self.min = mini
        self.max = maxi
        self.__doc__ = doc

    def __get__(self, inst, owner):
        print("--> __get__ du descripteur appelé.")
        return getattr(inst, '_' + self.name)

    def __set__(self, inst, value):
        print("--> __set__ du descripteur appelé.")
        if not self.min <= value <= self.max:
            raise ValueError("{} doit être compris entre {} et {} ({} donné).".format(
                self.name, self.min, self.max, value
            ))
        setattr(inst, '_' + self.name, value)

    def __set_name__(self, owner, name):
        self.name = name


class Time:
    def __init__(self, h, m, s):
        self.h = h
        self.m = m
        self.s = s

    h = BoundedIntDescriptor(0, 23, "Entier compris entre 0 et 23 représentant les heures.")
    m = BoundedIntDescriptor(0, 59, "Entier compris entre 0 et 59 représentant les minutes.")
    s = BoundedIntDescriptor(0, 59, "Entier compris entre 0 et 59 représentant les secondes.")

    def __repr__(self):
        return "{} h {} min {} s".format(self._h, self._m, self._s)
```

Le recours aux fonctions `getattr()` et `setattr()` permet d'accéder aux attributs des instances concernées. Si on avait stocké l'attribut dans l'instance du descripteur par exemple avec un `self.attr` au lieu de `inst._attr`, changer par exemple la valeur de `h` pour `t1` (cf. l'exemple précédent) aurait affecté `t2` ! En effet, le descripteur est instancié *au moment où la classe est définie et non à l'initialisation des instances de celle-ci.* En résumé, tous les attributs `h` pointent vers la même instance de `BoundedIntDescriptor`, de même pour `m` et `s`.

Pour récupérer le nom de l'attribut afin d'afficher une erreur explicite, j'ai utilisé la méthode `__set_name__()`.

Descripteur.__set_name__(self, owner, name)

Appelée lors de la création de la classe, c'est-à-dire quand le descripteur est instancié. Le nom de l'instance de `Descripteur` est assigné à `name`.

Jouons avec notre descripteur :

```
>>> t1 = Time(1, 2, 3)
--> __set__ du descripteur appelé.
--> __set__ du descripteur appelé.
--> __set__ du descripteur appelé.
>>> t1
1 h 2 min 3 s
>>> t1.h
--> __get__ du descripteur appelé.
1
>>> t1.m
--> __get__ du descripteur appelé.
2
>>> t1.s
--> __get__ du descripteur appelé.
3
>>> t1.h = 6
--> __set__ du descripteur appelé.
>>> t1
6 h 2 min 3 s
>>> t2 = Time(10, 11, 12)
--> __set__ du descripteur appelé.
--> __set__ du descripteur appelé.
--> __set__ du descripteur appelé.
>>> t2
10 h 11 min 12 s
>>> t1
6 h 2 min 3 s
>>> help(t1)
--> __get__ du descripteur appelé.
--> __get__ du descripteur appelé.
--> __get__ du descripteur appelé.

Help on Time in module __main__ object:

class Time(builtins.object)
 |  Time(h, m, s)
 |
 |  Methods defined here:
 |
 |  __init__(self, h, m, s)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  h
 |      Entier compris entre 0 et 23 représentant les heures.
 |
 |  m
 |      Entier compris entre 0 et 59 représentant les minutes.
 |
 |  s
 |      Entier compris entre 0 et 59 représentant les secondes.

>>> t1.m = -1
--> __set__ du descripteur appelé.
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<string>", line 19, in __set__
ValueError: m doit être compris entre 0 et 59 (-1 donné).
```

À ce stade, on peut se demander quelle solution choisir pour contrôler les accès aux attributs ou faire des attributs dynamiques : propriétés, descripteurs, méthodes spéciales ? La réponse est : si on peut faire ce que l'on veut facilement, ne pas choisir une solution compliquée inutilement ! Une réponse sur StackOverflow (voir « Plus d'informations ») propose ceci :

— La première solution la plus simple : utiliser le mécanisme par défaut qui utilise l'attribut spécial `__dict__` de l'instance.

— Si ce mécanisme est insuffisant, par exemple si on veut déclencher des calculs ou ajouter de la logique, utiliser les propriétés.

— Si ce mécanisme est insuffisant, écrire des descripteurs adaptés ; on vient de voir qu'ils sont utiles pour des propriétés génériques par exemple.

— Si ce mécanisme est inadapté, utiliser `__getattr__()` ; cette méthode est utile pour simuler la présence d'attributs. Cette méthode n'agit pas sur les attributs existants.

— En dernier recours, utiliser `__getattribute__()` ; la différence avec la solution précédente est que cette méthode est la première appelée lorsque l'on accède à un attribut. Ainsi, on a la main sur absolument tous les attributs, ce qui peut engendrer des comportements indésirables… À utiliser avec précaution, donc.

Plus d'informations

— Guide sur les descripteurs (documentation officielle).

— Quand utiliser les propriétés et les méthodes spéciales

5 Conteneurs

Remarque préléminaire Pour les méthodes spéciales spécifiques aux types des parties suivantes, je me base sur le module `collections.abc`

Les conteneurs sont des objets voués à contenir d'autres objets. Les principaux exemples de conteneurs sont les listes, les chaînes de caractères, les tuples, les dictionnaires ou encore les ensembles. Un objet est dit conteneur s'il possède la méthode spéciale `__contains__()`.

`conteneur.__contains__(objet)`

Retourne `True` si `objet` est présent dans `conteneur`, `False` sinon. On appelle cette méthode spéciale comme ceci :

```
objet in conteneur
```

La plupart des conteneurs possèdent une taille. Elle calculée par la méthode spéciale `__len__()`.

`conteneur.__len__()`

Retourne la taille du conteneur. Devrait retourner un entier positif ou nul. Si cette fonction retourne autre chose, une exception est levée. On l'appelle comme ceci :

```
len(conteneur)
```

Exemples Des « sized » capricieux

```
class StrSized:
    def __len__(self):
        return 'Ma taille !'

class NegativeSized:
    def __len__(self):
        return -1

len(StrSized()) # TypeError: 'str' object cannot be interpreted as an integer
len(NegativeSized()) # ValueError: __len__() should return >= 0
```

Notons qu'un objet peut avoir une taille sans pour autant être un conteneur (on appelle ça un *sized*).

5.1 Conteneurs indexables

Parmi ces conteneurs se distinguent les conteneurs que l'on peut indexer. Ce sont les conteneurs sur lesquels on peut utiliser les crochets `[]` pour faire référence à un objet dans le conteneur ; on parle de table de correspondance (*mapping*, non détaillé ici), et de séquence lorsque les index sont des entiers. Parmi les exemples cités précédemment, seuls les ensembles ne sont pas indexables. On rend un objet indexable en implémentant une méthode spéciale.

`indexable.__getitem__(index)`

Retourne l'objet référencé par `index` (cet indice peut être n'importe quel objet). Appelée en faisant :

```
indexable[index]
```

On peut rendre mutables les objets indexables grâce à deux méthodes spéciales.

`indexable.__setitem__(index, valeur)`

Assigne la nouvelle `valeur` à l'objet référencé par `index`. Appel :

```
indexable[index] = valeur
```

`indexable.__delitem__(index)`

Détruit l'objet référencé par `index`. Appel :

```
del indexable[index]
```

5.2 Objets séquentiels

Les objets séquentiels sont des objets indexables qui n'acceptent que des entiers comme index. <a venir : Slices>

6 Itérateurs

Les itérateurs sont des objets incontournables en Python, ils sont notamment utilisés lorsque l'on fait une boucle `for`. Les objets itérateurs peuvent être créés par des objets itérables. Des itérables connus sont les listes, les dictionnaires, les tuples, les `range()`.

L'intérêt principal des itérateurs est leur faible consommation mémoire : contrairement à un objet conteneur qui prend autant d'espace que d'objets qu'il contient, un itérateur calcule chaque élément lorsqu'il est appelé.

Exemple Ce qu'il se passe lorsque l'on fait une boucle `for`

```
>>> L = [0, 1, 2, 3, 4] # les listes sont des objets itérables
>>> for element in L # appelle l'itérateur de l'itérable L
...     print(element) # à chaque ligne, appelle l'élément suivant de l'itérateur
0
1
2
3
4
```

Les itérateurs sont implémentés sous forme de classes et doivent respecter le protocole d'itérateur : deux méthodes spéciales doivent être implémentées.

`iterateur.__iter__()`

Cette méthode doit retourner l'itérateur lui-même, on peut auparavant y effectuer quelques opérations d'initialisation. Appel :

```
iter(iterateur)
```

`iterateur.__next__()`

Cette méthode retourne l'élément suivant dans la séquence de l'itérateur. Une fois que le dernier élément a été appelé, lève une exception `StopIteration`. Appel :

```
next(iterateur)
```

Les itérables doivent quant à eux implémenter la méthode `__iter__()` qui appelle l'itérateur associé. On l'appelle en faisant `iter(objet_iterable)`.

Exemple Un incrémenteur

```
class Incrementor:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self. n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

On peut maintenant utiliser l'itérateur.

```
>>> inc = Incrementor(2)
>>> iterator = iter(inc)
>>> next(inc)
0
>>> next(inc)
1
>>> next(inc)
2
>>> next(inc)
Traceback (most recent call last):
  File "<stdin>", line XX, in <module>
    print(next(iterator))
  File "<stdin>", line XX, in __next__
    raise StopIteration
StopIteration
```

On peut aussi utiliser une boucle `for` pour itérer notre itérateur.

```
>>> for i in Incrementor(5):
...     print(i)
0
1
2
3
4
5
```

7 Générateurs

Plus d'informations L'excellent cours d'Antoine Rozo

7.1 Fonction génératrice et mot-clé `yield`

Les générateurs sont une façon plus simple d'implémenter les itérateurs. Au lieu de créer une classe avec les deux méthodes du protocole d'itération, on définit une fonction qui retourne les résultats avec le mot clé `yield`. Lorsqu'une fonction possède ce mot-clé, l'appeler crée un générateur (rien d'autre n'est exécuté). Un générateur est un itérateur qui possède quelques méthodes supplémentaires (cf. plus bas) ; ce sont en quelque sorte des itérateurs upgradés. On appelle par abus de langage les fonctions qui retournent des générateurs (des « fonctions génératrices ») des générateurs aussi (alors que ce sont de simples fonctions).

Pour faire le lien avec les itérateurs, on peut réécrire l'exemple précédent à l'aide d'un générateur.

```
def incrementor(max):
    n = 0
    while n <= max:
        yield n
        n += 1
```

On utilise les générateurs comme des itérateurs (les générateurs sont des itérateurs). Lorsque l'on évalue la méthode `__next__()` sur un générateur (en faisant `next(generateur)`), celui-ci parcourt la fonction génératrice jusqu'au premier `yield` qu'il rencontre, puis s'arrête. Lorsque la méthode `__next__()` est de nouveau appelée, le générateur continue le parcours jusqu'au `yield` suivant, et ainsi de suite. Lorsqu'il n'y en a plus, le générateur lève une exception `StopIteration`.

```
>>> gen = incrementor(2)
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen) # Erreur
StopIteration
>>> for i in incrementor(2):
...     print(i)
0
1
2
```

On n'appelle pas la méthode `iter()` sur un générateur. Ainsi, pour réinitialiser le générateur, on doit le recréer en appelant une nouvelle fois la fonction génératrice.

Remarque Distinction entre fonction génératrice et générateur

```
>>> incrementor
<function incrementor at 0x00000240AA034D08> # simple fonction
>>> incrementor(1)
<generator object incrementor at 0x00000240A9FC74F8> # générateur
```

7.2 Fonctions supplémentaires

En plus du mot-clé `yield`, on peut utiliser des fonctions supplémentaires dans les générateurs :

`generateur.send()`

Cette méthode permet de communiquer avec le générateur en lui envoyant une valeur. Lorsqu'elle est appelée avec un argument, celui-ci est envoyé au `yield` actuellement atteint, et le générateur reprend le parcours jusqu'au `yield` suivant. Ainsi, appeler cette méthode consomme une itération ! Lorsqu'on utilise cette méthode, il ne faut pas oublier d'affecter `yield` à une variable, sinon l'argument donné par `send` sera perdu. Séquentiellement, cela donne :

1. Le générateur vient d'être créé. On appelle `__next__()`, le générateur parcourt la fonction jusqu'au premier `yield`.
2. Le générateur rencontre un `yield`. Il envoie ce que le `yield` lui fournit et se met en pause.
3. Le générateur est à nouveau appelé. Si c'est avec un `send()`, il passe son argument au `yield` qui la donne à la variable à laquelle il est affecté.
4. Une fois que c'est fait, le générateur reprend le parcours de la fonction jusqu'au `yield` suivant, (retour à l'étape 2).

Exemple On reprend l'incrémenteur et on veut pourvoir l'étendre, c'est à dire lui envoyer un nombre et l'ajouter au maximum initial. Pour cela, on modifie la fonction génératrice et on ajoute des `print()` pour voir comment fonctionne `send()` (on utilise les f-strings).

```
def incrementor(max):
    n = 0
    while n <= max:
        print(f"max_pre : {max}")
        add_max = yield n
        print(f"max_post : {max}")
        print(f"add_max : {add_max}")
        n += 1
        max = max + add_max if add_max else max
```

Lorsque l'on appelle `send`, la valeur est stockée dans `add_max`. On peut alors étendre l'incrémenteur.

```
>>> gen = incrementor(2)
>>> next(gen)   # Le générateur est appelé, il commence le parcours
max_pre : 2     # Il rencontre le premier print()
0               # et un yield : il envoie ce que celui-ci lui fournit...
>>> next(gen)   # et attend qu'on le rappelle !
max_post : 2    # il reprend son parcours
add_max : None  # on voit bien que l'on a rien envoyé au générateur
max_pre : 2
1
>>> next(gen)
max_post : 2
add_max : None  # toujours rien...
max_pre : 2
2
>>> gen.send(3) # Le yield retourne à add_max la valeur de send()
max_post : 2    # le générateur reprend le parcours...
add_max : 3     # On a bien un add_max de 3 !
max_pre : 5     # arrivé au while, le max a donc changé, la boucle peut donc continuer !
3               # et on atteint bien le yield suivant
>>> next(gen)
max_post : 5
add_max : None
max_pre : 5
4
```

En fait, on s'aperçoit que `next(gen)` et `gen.send(None)` sont équivalents. On ne peut pas appeler `send()` avec autre chose que `None` en paramètre avant d'avoir appelé au moins une fois `__next__()`. En effet, l'affectation se fait par l'intermédiaire du `yield` actuel.

`generateur.throw(type[, value, traceback])`
: Envoie une exception au générateur. Si celui-ci l'attrape, alors retourne également la valeur suivante du générateur.

`generateur.close()`
: Envoie une exception au générateur

8 Coroutines

8.1 Objets *awaitables*

9 Décorateurs

Les décorateurs sont des fonctions ou des classes qui permettent de modifier le comportement d'une autre fonction (ou classe). Les décorateurs sont utiles lorsque l'on souhaite qu'un certain nombre de fonctions effectuent des tâches communes comme par exemple donner leur temps d'exécution. On appelle un décorateur de la manière suivante.

```
@decorateur
def fonction():
    pass
```

Le code précédent a le même comportement que le code suivant :

```
def fonction():
    pass

fonction = decorateur(fonction)
```

Ainsi, `fonction` devient l'objet retournée par `decorateur(fonction)`. Le décorateur doit donc être un callable.

Plus d'informations Stack Overflow

9.1 En tant que classe

Une façon d'implémenter un décorateur est d'utiliser les classes. La fonction décorée deviendra alors une instance de la classe de ce décorateur. Il faut obligatoirement définir la méthode `__call__()` pour pouvoir rendre cette instance *callable*.

Exemple On considère ici un décorateur qui compte le nombre d'appels de la fonction décorée.

```
class Compteur:
    def __init__(self, f):
        self.call = 0
        self.f = f

    def __call__(self, *args, **kwargs):
        self.call += 1
        print("La fonction {} a été appelée {} fois.".format(self.f.__name__, self.call))
        return self.f(*args, **kwargs)
```

9.2 En tant que fonction

Comme un décorateur est un objet *callable* qui n'a d'autre utilité que d'être appelé, il est aussi logique de le définir en tant que fonction.

Exemple Même décorateur que précédemment mais en l'implémentant en tant que fonction.

```
def compteur(f):
    def wrapper(*args, **kwargs):
        wrapper.call += 1
        print("La fonction {} a été appelée {} fois.".format(f.__name__, wrapper.call))
        return f(*args, **kwargs)
    wrapper.call = 0
    return wrapper
```

Dans cet exemple, on assigne à `wrapper` un attribut de fonction (on peut le faire, puisqu'une fonction est un objet – de la classe `function`). On le définit après avoir défini cette fonction.
Les deux décorateurs précédents donnent ce comportement :

```
>>> @compteur
... def func():
...     return "func got called!"
...
>>> func()
La fonction func a été appelée 1 fois.
'func got called!'
>>> func()
La fonction func a été appelée 2 fois.
'func got called!'
```

9.3 Décorateurs à paramètres

On peut faire en sorte que le décorateur prenne un ou plusieurs paramètres. Dans ce cas, il faut définir le décorateur à l'intérieur d'une clôture qui prend en argument ces différents paramètres.

Exemple On veut retourner une erreur quand la fonction retourne une valeur trop élevée.

```
def depasse_max(max):
    def deco(f):
        def wrapper(*args, **kwargs):
            n = f(*args, **kwargs)
            if n > max:
                print("Maximum {} dépassé.".format(max))
                return
            return n
        return wrapper
    return deco
```

Ces deux syntaxes sont équivalentes :

```
# Syntaxe 1
@depasse_max(10)
def demande_nombre():
    n = int(input("Entrer un nombre : "))
    return n

# Syntaxe 2
```

```
def demande_nombre():
    n = int(input("Entrer un nombre : "))
    return n

demande_nombre = depasse_max(10)(demande_nombre)
```

Cela permet de faire

```
>>> demande_nombre()
Entrer un nombre : 11
Maximum 10 dépassé.
```

10 Métaclasses

10.1 Principe

Les métaclasses sont les classes qui instancient d'autres classes. Par défaut, une seule métaclasse est définie : la métaclasse `type`. On s'en rend compte en demandant le type des classes que l'on crée.

```
>>> class MaClasse:
...     pass
...
>>> type(MaClasse)
<class 'type'>
```

10.2 La métaclasse `type`

On sait que passer à `type()` un objet renvoie son type, c'est-à-dire l'objet qui l'a instancié. Mais `type()` permet aussi des créer des types (des classes) à la volée si on lui passe plus d'arguments : un nom, un itérable contenant ses classes parentes et un dictionnaire contenant ses attributs.

```
>>> MyType = type("MyType", (), {}) # une classe on ne peut plus basique
>>> MyType()
<__main__.MyType object at 0x7ff838fb7518>
>>> issubclass(MyType, object)
True
>>> isisntance(MyType, type)
True
```

On peut donc créer des fonctions qui créent des classes.

```
def class_creator(name, bases=(), attrs={}): # il faut garder la signature de type()
    """Créateur de classe personnalisé.

    Celui ajoute à chaque classe créée un identifiant de classe correspondant au
    nombre de classes créées au moment de l'appel de class_creator().
    """
    if not hasattr(class_creator, "increment"):
        class_creator.increment = 0 # on utilise un attribut de fonction
    attrs["class_id"] = class_creator.increment
    class_creator.increment += 1
    return type(name, bases, attrs)
```

```
>>> first = class_creator("first")
>>> second = class_creator("second")
>>> first.class_id
0
>>> second.class_id
1
>>> class Third(metaclass=class_creator):
...     pass
...
>>> Third.class_id
2
```

Ou plutôt des générateurs de classes :

```
def class_generator_function(bases=(), attrs={}):
    """Générateur de classe."""
    increment = 0
    name = yield # initialisation
    while True:
        attrs["class_id"] = increment
        name = yield type(name, bases, attrs)
        increment += 1
```

```
>>> class_generator = class_generator_function()
>>> next(class_generator) # initialisation
>>> first = class_generator.send("first")
>>> second = class_generator.send("second")
>>> first.class_id
0
>>> second.class_id
1
```

10.3 Ecrire une métaclasse

On est un peu limité dans le cas de fonctions qui créent des classes. Pour des choses plus complexes, on peut écrire des classes qui instancient d'autres classes : des métaclasses.

Pour qu'une classe puisse instancier d'autres classes, il faut hériter de `type`. Cela permet notamment d'hériter de sa fonction `__new__()`. D'habitude (c'est-à-dire quand on hérite simplement de `object`), cette fonction ne fait rien de spécial, elle retourne simplement un objet vide que l'on initialise dans `__init__()`. Dans le cas de `type`, c'est cette fonction qui est chargée de créer la classe. Pour indiquer que l'on est en train de définir une métaclasse, on écrit `cls` au lieu de `self` pour faire référence à l'objet instancié, et `mcls` au lieu de `cls` pour faire référence au type.

Cette métaclasse ne fait rien de plus que `type` :

```
class SimpleMeta(type):
    def __new__(mcls, name, bases, attrs):
        print(name, "was created.")
        return super().__new__(mcls, name, bases, attrs)

class ClassUsingSimpleMeta(metaclass=SimpleMeta):
    pass
```

A l'exécution on verra :

```
ClassUsingSimpleMeta was created.
```

Si l'on veut le même comportement que les exemples précédents :

```
class SimpleMeta(type):
    class_id = 0
    def __init__(cls, name, bases, attrs): # on récupère les arguments de __new__
        super().__init__(name, bases, attrs)
        cls.class_id = cls.class_id
        type(cls).class_id += 1
```

```
>>> class Class0(metaclass=SimpleMeta):
...     pass
...
>>> Class2 = SimpleMeta("Class2", (), {})
>>> Class0.class_id
0
>>> Class1.class_id
1
```

On peut ajouter des paramètres lors de la déclaration d'une classe :

```
class AbstractoOrNotAbstractMeta(type):
    def __new__(mcls, name, bases, attrs, abstract=False):
        cls = super().__new__(mcls, name, bases, attrs)
        print(name)
        if abstract:
```

```
            def new(cls, *args, **kwargs):
                raise TypeError("This class is abstract.")
            cls.__new__ = new
        else:
            cls.__new__ = object.__new__
        return cls


class Abstract(metaclass=AbstractoOrNotAbstractMeta, abstract=True):
    pass

class Concrete(Abstract):
    pass

Concrete()
Abstract() # TypeError
```

10.4 Application des métaclasses : propriété de classe

On pourrait imaginer des propriétés de classes afin d'ajouter une couche de logique sur une simple variable de classe. Au lieu de définir un descripteur générique, on créer une métaclasse qui aura comme propriété la future propriété de classe.

Exemple Un exemple simple

```
class ClassPropertyMeta(type):
    def __new__(mcs, name, bases, attrs):
        """Créateur personnalisé.

        On redéfinit __new__ pour s'assurer que les éventuels setters des
        propriétés soient appelés.
        """
        cls = super().__new__(mcs, name, bases, {})
        for attr, value in attrs.items():
            setattr(cls, attr, val)
        return cls

    @property
    def some_positive_attr(self):
        return self._propriete

    @some_positive_attr.setter
    def some_positive_attr(self, value):
        if value < 0:
            raise ValueError("some_positive_integer must be > 0.")
        self._propriete = value


class ClassPropertyOwner(metaclass=ClassPropertyMeta):
    some_positive_attr = -1
```

Cette définition de classe va lever une exception `ValueError`. Oui, une déclaration de classe peut lever une exception.

Troisième partie
Bibliothèque standard

1 Fonctions et types natifs

builtins

Cette section présente des fonctions et types accessibles dans l'interpréteur sans rien importer. Ce sont les `builtins`.

Plus d'informations et listes exhaustives

- Fonctions et constructeurs natifs
- Constantes natives
- Types natifs ou types standard

Remarque Parfois, on est amemnés (même si ce n'est pas forcément recommandé) à nommer un variable d'après un objet natif. Pour retrouver ledit objet natif, on peut explicitement passer par le module `builtins`.

1.1 Interactions avec l'utilisateur

`print(*objects, sep=' ', end="\n ", file=sys.stdout, flush=False)`

Affiche les `objects` passés en paramètre sous forme de chaîne de caractères grâce au constructeur `str()`. Si plusieurs objets sont passés en paramètre, ils sont séparés par `sep`. La sortie se termine par `end`. Ces deux derniers paramètres doivent être nommés, tous les arguments positionnels sont considérés objets à afficher. Si aucun objet n'est fourni, l'affiche que `end`.

```
>>> print("Python", "OCaml", "C", "Java", sep=" > ")
Python > OCaml > C > Java
```

Afficher revient en fait à écrire dans un flux texte. Par défaut, il s'agit de la sortie standard (ce qui s'affiche sur la console). On peut changer la cible de la sortie avec le paramètre `file` ; l'objet passé en paramètre doit avoir une méthode `write(chaine)`. On peut forcer l'effacement du tampon de flux en mettant le paramètre `flush` à `True`.

Pour afficher des progressions, on peut utiliser le retour chariot `"\r"` comme chaîne finale :

```
>>> import time
>>> for i in range(100):
...     print(f"{i+1} %", end="\r")
...     time.sleep(0.1)
... else:
...     print("\nTerminé !")
...
100%
Terminé !
```

`input(prompt="")`

Affiche `prompt` et lit une ligne sur l'entrée standard qui est convertie en chaîne de caractères (en supprimant le retour à la ligne final).

```
>>> print(input("Glou glou ?\n"))
Glou glou ?
Ouaf
Ouaf
>>> def interactive_add():
...     first = int(input("Entrer un premier entier : "))
...     second = int(input("Entrer un second entier : "))
...     result = first + second
...     print(f"{first} + {second} = {result}")
...     return result
...
>>> result = interactive_add()
Entrer un premier entier : 1846
Entrer un second entier : 898
1846 + 898 = 2744
>>> result
2744
```

1.2 Interactions avec le développeur et débogage

help([objet])

Invoque le système d'aide de Python. Si aucun paramètre n'est donné, lance l'aide interactive de Python. Le paramètre objet peut être tout objet importé ou un mot-clé sous forme de chaîne de caractères.

breakpoint(*args, **kwargs)

Sans paramètre, place dans le débogueur. Cette fonction appelle en fait `sys.breakpointhook()`. Il est possible de passer des paramètres à cette fonction, cf. la documentation.

globals()

Renvoie un dictionnaire contenant les variables globales du module courrant. Si appelé dans une fonction ou une méthode importée, renvoie les variables globales du module de provenance.

locals()

Renvoie un dictionnaire contenant les variables locales. Si appelé au niveau d'un module, a le même comportement que `globals()`.

vars([object])

Renvoie `objet.__dict__` si `objet` est renseigné, et `locals()` sinon.

dir([object])

Renvoie la liste des variables l'espace de nom courant, et si `object` est renseigné, tente de renvoyer la liste de ces attributs valides.

1.3 Types utilitaires pour les boucles

class range

Les objets `range` sont des objets séquentiels (donc itérables) dont les éléments sont calculés à la volée. Il s'agit d'un outil indispensable pour faire des boucles. Il y a deux manières de construire un objet `range` :

range(fin) Construit un objet `range` qui permet d'itérer sur tous les entiers de 0 à `fin`.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

range(debut, fin, pas=1) Construit un objet `range` permettant d'itérer sur les entiers de `debut` à `fin` avec un pas de `pas`. Lève une exception `ValueError` si `pas` vaut 0.

```
>>> for i in range(6, -1, -2):
...     print(i)
...
6
4
2
0
```

Un objet `range` n'est pas un itérateur.

```
>>> next(range(1))
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'range' object is not an iterator
```

Il s'agit d'un objet séquentiel itérable qui gère toutes les opérations communes sur de tels objets (indices, tranches, etc.). Les valeurs sont calculées à la volée grâce à la formule `range_object[i] = debut + pas*i`, avec comme contraintes `i >= 0` et `range_object[i] < fin` si `pas > 0` sinon `range_object[i] > fin`.

Attributs définis :

range.start Valeur du paramètre `debut` passé au constructeur (0 par défaut).

range.stop Valeur du paramètre `fin` passé au constructeur.

range.step Valeur du paramètre `pas` passé au constructeur (1 par défaut).

class zip

Les objets `zip` permettent de construire des objets itérables agrégeant d'autres itérables.

zip(*iterables) Construit un objet `zip` permettant d'itérer sur l'ensemble des itérables passés en paramètres à la fois. Le i-ème élément de cet itérateur est un tuple contenant les i-èmes éléments de chaque itérable. L'itérateur est épuisé lorsque dès qu'un des itérables en paramètre est épuisé.

```
>>> for t in zip("python", "java", "cpp", "ocaml"):
...     print(*t, sep=", ")
...
p, j, c, o
y, a, p, c
t, v, p, a
```

class enumerate

Les objets `enumerate` permettent d'itérer et d'énumérer à la fois.

enumerate(iterable, start=0) Construit un objet `enumerate` générant des tuples contenant un compte (démarrant à `start`), et les objets résultant de l'itération sur l'itérable passé en paramètre.

```
>>> for i, c in enumerate("abc"):
...     print(f"{i} : {c}")
0 : a
1 : b
2 : c
```

En quelque sorte, `enumerate(iterable)` est l'équivalent de `zip(range(len(iterable)), iterable)`.

1.4 Manipulation des attributs d'un objet

Les fonctions natives suivantes sont utiles pour manipuler dynamiquement les attributs d'un objet.

getattr(obj, name[, default])

Renvoie la valeur de l'attribut nommé `name` de l'objet `obj` : `getattr(obj, "x")` renvoie `obj.x`. Si `name` n'est pas pris en charge par `obj` (c'est-à-dire si `obj.__getattribute__()` et `obj.__getattr__()` ne parviennent pas à calculer `name`), renvoie `default` s'il est renseigné, ou lève une exception `AttributeError` sinon.

hasattr(obj, name)

Permet de tester l'existence l'attribut nommé `name` par `obj` (`hasattr(obj, "x")` teste l'existence de `obj.x`) : renvoie `False` si `getattr(obj, name)` lève une exception `AttributeError`, et `True` sinon.

setattr(obj, name, value)

Assigne `value` à l'attribut nommé `name` de `obj` : `setattr(obj, "x", 1)` est équivalent à `obj.x = 1`.

Ces fonctions sont équivalentes aux syntaxes classiques d'accès aux attributs et et de leur modificaion (i.e. `obj.attr`, `obj.attr = val`), elles ne permettent pas de d'éviter le mécanisme de descripteur ou les méthodes spéciales `__getattribute__()`, `__getattr__()` ou `__setattr__()`.

```
>>> class Foo:
...     @property
...     def a(self):
...         print("prop access")
...         return 1
...
>>> getattr(Foo(), "a")
prop access
1
>>> setattr(Foo(), "a", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

1.5 Exécuter, évaluer du code

exec(instruction)

Exécute la chaîne de caractères `instruction` en argument.

eval(expression)

Évalue la chaîne de caractères `expression` en argument et renvoie le résultat.

1.6 Héritage et typage

super(type_obj, obj)

isinstance(obj, classinfo)

issubclass(obj, classinfo)

1.7 Le père et le créateur

class object

Classe parente de toutes les classes. Toutes les classes héritent implicitement d'`object`.

object() Constructeur de la classe `object`. Construit un objet vide auquel on ne peut assigner d'attribut et qui possède les méthodes communes à tous les objets, documentées ci-dessous.

object.__new__(classe) Crée un objet vide. `object.__new__()` n'accepte qu'un type comme argument, sauf quand `__init__()` est définie dans une classe, au quel cas `object.__new__()` accepte tous les arguments de `__init__()`. Si on veut surcharger `object.__new__()`, il faudra cependant penser à créer l'objet vide en appelant `object.__new__()` avec la classe comme unique paramètre.

```
>>> class ClassWithoutInit:
...     pass
...
>>> class ClassWithInit:
...     def __init__(self, param):
...         self.attr = param
...
>>> class ClassWithNew:
...     def __new__(cls, param):
...         instance = super().__new__(cls)
...         instance.attr = param
...         return instance
...
>>> object.__new__(ClassWithOutInit)
<__main__.ClassWithoutInit object at 0x7f11b9448cc0>
>>> object.__new__(ClassWithInit, 1)
<__main__.ClassWithInit object at 0x7f11b933f400>
>>> object.__new__(ClassWithNew, 1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: object.__new__() takes exactly one argument (the type to instantiate)
```

object.__repr__(self) Renvoie une chaîne de caractère indiquant le type de l'objet ainsi que son adresse mémoire.

object.__str__(self) Renvoie `object.__repr__(self)`.

object.__dir__(self) Appelée quand `dir()` est appelée sur un objet. Renvoie un itérable des attributs de l'objet.

object.__eq__(self, other) Appelée lors de l'opération `objet == other`. Compare les valeurs de hachage et renvoie `True` si elles sont égales.

object.__ne__(self, other) Appelée lors de l'opération `objet != other`. Renvoie l'opposé de `object.__eq__(self, other)`.

object.__ge__(self, other) Appelée lors de l'opération `objet >= other`. Renvoie `NotImplemented`.

object.__gt__(self, other) Appelée lors de l'opération `objet > other`. Renvoie `NotImplemented`.

object.__le__(self, other) Appelée lors de l'opération `objet <= other`. Renvoie `NotImplemented`.

object.__lt__(self, other) Appelée lors de l'opération `objet < other`. Renvoie `NotImplemented`.

object.__getattribute__(self, name) Appelée lorsqu'on accède à l'attribut `name` avec la syntaxe `self.name`. Cette méthode va chercher `name` dans le dictionnaire de l'instance `self.__dict__`. S'il est introuvable, elle va chercher successivement dans le dictionnaire du type de l'instance, puis dans toutes les classes parentes. Dans le cas où l'attribut est trouvé dans le dictionnaire d'une classe, s'il possède une méthode `__get__()` (c'est-à-dire s'il s'agit d'un descripteur ou par extension d'une propriété), alors celle-ci sera appelée.

object.__setattr__(self, name, value) Appelée lorsque l'on écrit `self.name = value`. Le mécanisme de recherche d'attribut est le même que pour `object.__getattribute__()`. Dans le cas d'un descripteur, c'est `__set__()` qui est appelée.

object.__sizeof__(self) Renvoie la taille occupée en mémoire de `self` en octets.

class type

Unique métaclasse (créateur de type) native.

type(objet) Passer un objet à `type` renvoie la classe qui l'a instancié.

type(name, bases, attrs) Pour créer un type, on appelle le constructeur avec les paramètres suivant :

1. `name` : le nom du type à créer ;
2. `bases` : les classes parentes du type à créer (`object` est implicite) dans un itérable ;
3. `attrs` : un dictionnaire contenant des attributs à affecter au type.

2 `abc`, `collections.abc` — Classes mères abstraites

3 `re` — Expressions régulières

re

Le module `re` permet d'utiliser les expressions régulières en Python.

Plus d'informations Documentation Python 3

3.1 Écrire une expression régulière

Les expressions régulières sont un excellent moyen de retrouver des motifs complexes dans une chaîne de caractères. On écrit les motifs à rechercher grâce à plusieurs caractères spéciaux :

Spécification du caractère

- « `.` » désigne n'importe quel caractère.
- « `[]` » permet de dire quels caractères on veut trouver (`[a-e]` : `a`, `b`, `c`, `d` ou `e` ; `[a-eA-E]` idem avec les majuscule comprises ; `[+-*]` : soit `*` soit `+` soit `-`).
- « `\W` » désigne tout caractère non alpha-numérique.
- « `\d` » équivaut à `[0-9]`.
- « `\D` » désigne tout caractère non numérique.
- « `\w` » équivaut à `[a-zA-Z0-9_]`.
- « `\s` » désigne un espace.

Place du motif dans la chaîne

- « `^` » (se place au début) signifie que le début de la chaine doit correspondre au motif.
- « `\$` » (se place à la fin) signifie que la fin de la chaine doit correspondre au motif.

Nombre d'apparition(s) consécutive(s)

- « `{n}` » indique que le caractère précédent doit apparaître `n` fois.
- « `{n,m}` » indique que le caractère précédent doit apparaître entre `n` et `m` fois.
- « `*` » indique que le caractère précédent n'apparaît pas ou apparaît sans maximum d'occurrences (`ab*` correspond à `a`, `ab`, ou bien `abbbbbb`, etc.).
- « `+` » indique que le caractère précédent apparaît au moins une fois (`ab+` correspond à `ab`, `abb`, ou bien `abbbbbb`, etc.).
- « `?` » indique que le caractère précédent apparaît au plus une fois (équivalent à `{0,1}`).

Les quatre derniers qualificateurs sont dits gourmands : ils valident autant de caractères que possible. Par exemple pour `"aaaaa"`, `a{3,5}` validera la chaîne en entier. Pour une version non gourmande, on suit le qualificateur d'un `?` : `*?`, `+?`, `??` et `{n,m}?`. Un qualificateur non gourmand valide le moins de caractères possibles.

Pour contrôler le nombre d'apparitions d'un groupe de caractères, on met ceux-ci entre parenthèses (`(abc)+` : `abc`, `abcabc`, etc.). Cela crée un groupe de caractères, on peut le nommer en suivant la parenthèses ouvrante de `?P<nom>`. Cela est utile par exemple quand on veut remplacer des caractères. On peut séparer des expression régulières par un `|` afin d'indiquer que plusieurs possibilités sont possibles.

3.2 Méthodes

re.compile

On compile une expression régulière en utilisant la fonction `compile` . Cette fonction retourne un objet expression régulière (un objet Pattern) sur lequel on peut évaluer diverses méthodes. Si l'on cherche une phrase, la syntaxe sera :

```
import re

regex = re.compile(r"[A-Z]\w*\s?(\w+\s?)*.")
```

Remarque On utilise le préfixe `r` devant la chaîne de caractère pour éviter d'avoir à écrire `\\` au lieu d'un unique `\`.

Pattern.finditer

On peut rechercher toutes les occurrences du motif grâce à la méthode `re.finditer(motif, chaine)` . Cela retourne un objet iterable. On accède aux objets en appelant `next(iterable)`, qui retourne un objet expression rationnelle. Celui-ci contient plusieurs chaînes de caractères (une pour chaque groupe du motif), on y accède en appelant les différents groupes : `objet.group(numéro ou nom)`.

Exemple On veut extraire les phrases d'une chaîne de caractères.

```
>>> chaine = r"Je suis une phrase. Moi aussi"
>>> regex = re.compile(r"[A-Z]\w*\s?(?:\w+\s?)*.")
>>> resultats = regex.finditer(chaine)
>>> for phrase in resultats:
...     print(phrase.group(0))
...
Je suis une phrase.
Moi aussi.
```

Pattern.sub

On peut remplacer les motifs par d'autres motifs en utilisant la méthode `re.sub`. Elle prend en paramètres :

1. le motif (chaîne de caractères ou objet expression rationnelle.)
2. le remplacement (peut être une fonction)
3. la chaine à traiter
4. `count`=le nombre d'occurrences à remplacer

et renvoie la chaine de caractères modifiée. Lorsque l'on veut appeler un groupe de caractères nommé avec `(?P<nom>)`, on y fait référence dans la chaine de remplacement par `\g<nom>`.

4 `datetime` — Objets dates

datetime

Le module `datetime` permet de créer des objets représentant des dates et de faire des opérations. La classe `datetime.date` représente une date par son année, son mois et son jour : `jour = datetime.date(2017, 1, 1)` correspond à la date 1er janvier 2017. La classe `datetime.timedelta` permet de faire des opérations sur les dates. Ses objets sont représentés par un nombre de jours (on peut construire un `timedelta` avec des semaines/mois/années, le constructeur convertit en jours). Le module `datetime` peut ausi être utilisé pour utiliser des durées plus réduites, i.e. secondes, minutes, heures, etc.

Exemple

```
>>> import datetime
>>> j1 = datetime.date(2017, 1, 1)
>>> j2 = j1 + datetime.timedelta(30)
>>> j2
datetime.date(2017, 1, 31)
```

Documentation Documentation Python 3

5 `functools` — Outils pour la programmation fonctionnelle

functools

Ce module fournit des outils pour la manipulation de fonctions

5.1 Préservation des attributs d'une fonction décorée

`@functools.wraps` Décorer une fonction change ses attributs (en effet, on retourne un wrapper en général) : en particulier son nom et sa docstring :

```
>>> def f():
...     return "Hello!"
...
>>> f()
"Hello!"
>>> f
<function f at 0x0000023CECFFC1E0>
>>> def deco(f):
...     def wrapper():
...         print(f())
...         print("Done.")
...     return wrapper
...
>>> @deco
... def f():
...     return "Hello!"
...
>>> f()
Hello!
Done.
>>> f
<function deco.<locals>.wrapper at 0x0000023CEDB2EF28>
```

Le décorateur `@wraps` du module `functools` permet d'y remédier.

```
>>> from functools import wraps
>>> def deco(f):
...     @wraps(f)
...     def wrapper(*args, **kwargs):
...         print(f())
...         print("Done.")
...     return wrapper
...
>>> @deco
... def f():
...     return "Hello!"
...
>>> f()
Hello!
Done.
>>> f
<function f at 0x00000223E312DEA0>
```

6 `turtle` — Dessins basiques

Contient des classes pour dessiner des formes simples en faisant avancer des tortues. Elles peuvent avancer, reculer, tourner d'un certain angle. La classe `Turtle` permet de créer des objets tortues qui peuvent :

1. Avancer : `Turtle.forward(<nb de pixels>)`
2. Reculer : `Turtle.backward(<nb de pixels>)`
3. Tourner à droite ou à gauche (ex : `Turtle.right(<degrés>)`)
4. Changer de couleur (`Turtle.color(<couleur>)`) ou de forme (`Turtle.shape(<forme>)`).

Exemple

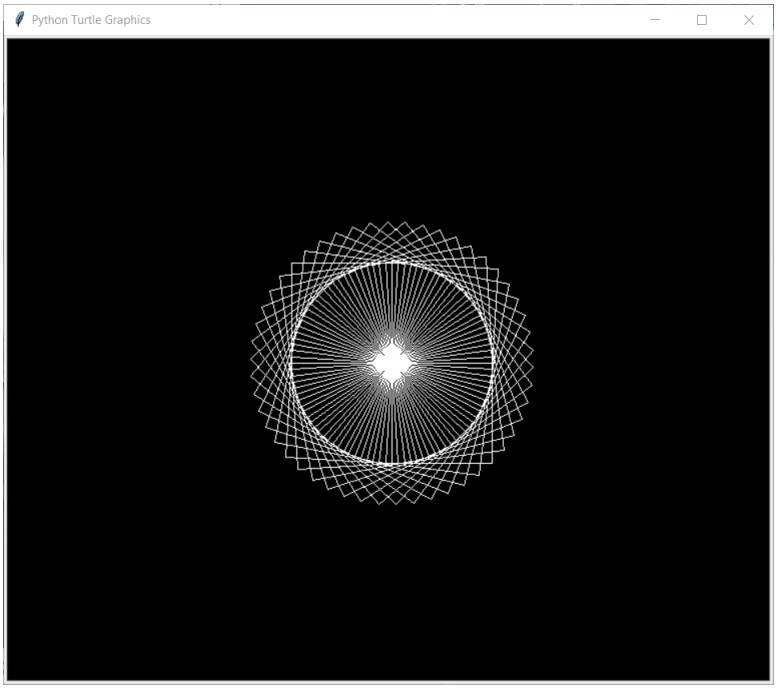
```
import turtle

Terrain = turtle.Screen()
Terrain.bgcolor("black")

Tortue = turtle.Turtle()
Tortue.speed(3)
Tortue.shape("turtle")
Tortue.color("white")
```

```
for i in range(50):
    for e in range(4):
        Tortue.forward(100)
        Tortue.right(90)
    Tortue.right(360/50)

Terrain.exitonclick()
```



Résultat

Plus d'informations Documentation Python 3, Wikilivres

7 ctypes — Appeler des fonctions en C

Ce module sert à appeler des fonctions écrites en langage C dans des librairies DLL par exemple.

7.1 Boites de dialogue

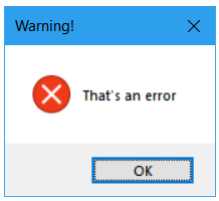
Le module ctypes peut servir à faire apparaître des boites de dialogue. On peut modifier le comportement du script Python en fonction du bouton appuyé car la fonction faisant apparaître ces boites renvoie un entier qui dépend du bouton appuyé. Diverses options sont disponibles :

```
# Boutons disponibles :
# 0 : OK
# 1 : OK | Annuler
# 2 : Abandonner | Recommencer | Ignorer
# 3 : Oui | Non | Annuler
# 4 : Oui | Non
# 5 : Recommencer | Annuler
# 6 : Annuler | Recommencer | Continuer

# Icone
# 16 Icone erreur
# 32 Icone question
# 48 Icone attention
# 64 Icone information
```

Exemple

```
ctypes.windll.user32.MessageBoxW(0, "That's an error", "Warning!", 16)
```



Résultat

8 keyboard — Manipulation du clavier

9 os — Diverses interfaces avec le système d'exploitation

10 sys — Fonctions et paramètres spécifiques au système

11 threading — Parallélisme avec les threads

12 asyncio — Programmation asynchrone

Nota Bene Pour l'instant, cette section sort un peu de nulle part, mais tout s'éclaircira quand la syntaxe des `async` et `await` sera expliquée.

Cette bibliothèque permet de facilement effectuer de la programmation asynchrone (à ne pas confondre avec la programmation en parallèle). L'intérêt est de rendre les tâches non-bloquantes, typiquement les requêtes à des serveurs qui peuvent mettre du temps à obtenir une réponse. Cette librairie n'est pas adaptée pour les calculs longs, car elle ne permet pas de calculer plus rapidement !

Exemple Voici un code synchrone que l'on peut rendre asynchrone.

```
import time


def sync_get_response(id, temps_de_reponse):
    time.sleep(temps_de_reponse/1000)  # simulation temps de reponse
    print("Réponse {} reçue !".format(id))


def sync_main():
    sync_get_response(1, 50)
    sync_get_response(2, 50)
    sync_get_response(3, 4000)
    sync_get_response(4, 50)
    sync_get_response(5, 50)
    sync_get_response(6, 6000)
    sync_get_response(7, 50)
    sync_get_response(8, 50)
    sync_get_response(9, 50)

beginning = time.time()
sync_main()
print('Durée', time.time()-beginning)
```

```
Réponse 1 reçue !
Réponse 2 reçue !
Réponse 3 reçue !
Réponse 4 reçue !
Réponse 5 reçue !
Réponse 6 reçue !
Réponse 7 reçue !
Réponse 8 reçue !
```

```
Réponse 9 reçue !
Durée 10.367376565933228
```

On remarque que les requêtes longues ralentissent l'exécution du programme. De plus, les requêtes sont effectuées dans l'ordre.

```
import asyncio
import time


async def get_response(id, temps_de_reponse)
    await asyncio.sleep(temps_de_reponse/1000)
    print("Réponse {} reçue !".format(id))


async def main():
    req1 = loop.create_task(get_response(1, 50))
    req2 = loop.create_task(get_response(2, 50))
    req3 = loop.create_task(get_response(3, 1000)) # simulons une requete longue
    req4 = loop.create_task(get_response(4, 50))
    req5 = loop.create_task(get_response(5, 50))
    req6 = loop.create_task(get_response(6, 1000)) # une autre
    req7 = loop.create_task(get_response(7, 50))
    req8 = loop.create_task(get_response(8, 50))
    req9 = loop.create_task(get_response(9, 50))
    await asyncio.wait([req1, req2, req3, req4, req5, req6, req7, req8, req9])

if __name__ == '__main__':
    beginning = time.time()
    try:
        loop = asyncio.get_event_loop()
        loop.run_until_complete(main())
    except:
        pass
    finally:
        loop.close()
        print('Durée', time.time()-beginning)
```

```
Réponse 1 reçue !
Réponse 7 reçue !
Réponse 9 reçue !
Réponse 8 reçue !
Réponse 2 reçue !
Réponse 5 reçue !
Réponse 4 reçue !
Réponse 3 reçue !
Réponse 6 reçue !
Durée 5.994143724441528
# asyncio.sleep() resquillerait-il ? On lui a pourtant demandé de dormir 6 secondes !
```

Dans le 2e exemple, le programme n'attend pas de recevoir la réponse pour envoyer les autres requêtes.

Quatrième partie

Modules à télécharger

1 `virtualenv` — Environnements virtuels

`virtualenv`

Les environnement virtuels sont un bon moyen pour :

1. Installer des modules sans avoir besoin des droits administrateurs
2. Avoir plusieurs environnements de travail avec des modules Python de versions différentes. Exemple, j'ai un projet Django 2 et je veux créer un site avec Django-CMS, qui requiert Django 1 ! Je suis obligé de recourir aux environnements virtuels.

Pour une utilisation basique, on commence par installer `virtualenv` avec `pip`.

```
$ pip install virtualenv # ou pip3 selon votre version de Python
```

Puis on se place dans le dossier où l'on veut placer les environnements virtuels, par exemple sous Linux dans `/home/votre_nom/python_env/`, et on crée notre environnement !

```
$ virtualenv env
```

Python y place alors les exécutables fondamentaux et quelques modules basiques. Ensuite, pour travailler dans l'environnement créé, il faut lancer la commande :

```
$ source /home/votre_nom/python_env/env/bin/activate
```

L'environnement apparaît maintenant entre parenthèses dans la console. Pour désactiver cet environnement, on lance simplement la commande :

```
(env) $ deactivate
```

`virtualenvwrapper` Il existe le module `virtualenvwrapper` qui permet de naviguer facilement entre les environnements. Après avoir installé ce paquet, il faut ajouter dans le path une variable `WORKON_HOME` qui correspond au répertoire où seront stockés les environnement virtuels. Ensuite on pourra utiliser les commandes

```
$ mkvirtualenv env # creation d'un environnement virtuel
$ workon # visualisation des environnements existants
env
(env) $ workon env # selection d'un environnement
(env) $ deactivate # quitter cet environnement
```

Plus d'informations Documentation de virtualenv, informations supplémentaires

2 Flask — Microframework Web

Ce framework minimaliste permet de faire des sites Web sans pour autant imposer une façon de développer. Plugins utiles :

SQLAlchemy un ORM pour gérer plus facilement la bdd

Marsmallow un serializer pour Python

3 `django` — Framework Web Full Stack

`django` Ce module permet de créer des sites web en Python. *Il est question ici de la version 2.*

Plus d'informations Documentation officielle de Django 2.2 Tutoriel de la documentation

3.1 Fonctionnement

Django fonctionne selon l'architecture Model-View-Template (MVT) que l'on peut traduire par Modèle-Vue-Gabarit. Celle-ci s'appuie sur l'architecture Model-View-Controler (MVC) :

- Les modèles structurent de la base de données, là où sont stockées toutes les informations. Ici, ce sont des classes Python dont les attributs correspondent à des champs dans la base de données. On n'écrit jamais de SQL avec Django !
- Les vues représentent les pages web : elles présentent les informations aux utilisateurs et récupèrent leurs actions. Ici, ce sont des fonctions Python qui prennent en argument la requête (HTTP par exemple) et des informations sur l'URL et qui renvoie, en utilisant les gabarits, la bonne page à l'utilisateur (la bonne réponse HTTP).
- Les gabarits permettent de structurer facilement les vues. Ce sont des fichiers HTML avec un peu de syntaxe de gabarit Django.
- Le contrôleur fait l'interface entre les vues et les modèles : il récupère et renvoie les informations nécessaires. Cette partie est gérée de manière autonome par Django.

3.2 Didacticiel

Cette partie s'appuie sur le tutoriel de la documentation Django, ne pas hésiter à s'y rendre pour plus d'infos. Concertant l'installation, il est conseillé d'installer Django dans un environnement virtuel. Dans cet environnement, on utilise l'installateur autonome `pip`.

```
$ pip install Django
```

3.2.1 Créer un projet

```
$ django-admin startproject nom_du_projet
```

Un dossier est créé, avec trois sous-dossiers (un nommé d'après le projet, un dossier media, et un dossier static) et trois fichiers (une base de données, un fichier python et un fihcier requirements.txt). Pour lancer une première fois le projet sur un serveur local, on utilise la commande (il faut être dans le dossier du projet) :

```
$ python manage.py runserver # on peut remplacer python par python3
```

En se rendant sur l'URL indiquée, ou plus simplement `localhost:8000` (on peut modifier le port si l'on veut : on écrit le port souhaité à la suite de la commande précédente), on tombe sur une page nous disant que l'installation de Django a réussi.

3.2.2 Créer une application

Une fois le projet créé, on crée une première application (cela peut être un sondage, un blog, etc., les applications sont les blocs du site). Une application peut être réutilisée pour d'autres projets. On crée une application par la commande (en étant dans le répertoire du projet) :

```
$ python manage.py startapp nom_de_l_application
```

3.2.3 Le fichier `settings.py`

Il comporte les principaux paramètres du projet. On y renseigne notamment le type de base de données que l'on utilise ; si on utilise SQLite, tout est géré automatiquement. On y gère aussi le fuseau horaire, les langues, les applications installées, parmi les suivantes, installées par défaut :

- `django.contrib.admin` : l'interface d'administration
- `django.contrib.auth` : un système d'authentification
- `django.contrib.contenttypes` : une structure pour les types de contenu
- `django.contrib.sessions` : un cadre pour les sessions
- `django.contrib.messages` : un cadre pour l'envoi de messages
- `django.contrib.staticfiles` : une structure pour la prise en charge des fichiers statiques

3.2.4 Migrations

Ces applications nécessitent des tables dans la base de données. Elles ne sont pas créées lors de la création du projet (d'où un probable message d'erreur lors du premier lancement), on crée les tables nécessaires grâce à la commande :

```
$ python manage.py migrate
```

Il faut relancer cette commande lorsque l'on doit mettre à jour la base de données, typiquement lorsque l'on crée ou modifie des modèles, ou que l'on importe ou crée des applications.

3.2.5 Structure des fichiers

La structure des fichiers est la suivante, pour un projet appelé monsite et une application nommée monapplication.

```
monsite/
    manage.py
    monsite/
        __init__.py
        settings.py
```

```
        urls.py
        wsgi.py
    monapplication/
        __init__.py
        admin.py
        migrations/
            __init__.py
        models.py
        tests.py
        views.py
```

On s'intéresse maintenant à cette application

3.2.6 Ecrire une vue

Les vues s'écrivent dans le fichier `views.py`, ce sont des fonctions. On peut commencer par écrire une première vue basique :

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello world!")
```

Cette fonction récupère une requette HTTP et renvoie une réponse HTTP. Celle-ci est écrite en HTML ici directement en argument de `HttpResponse()`, en général on n'utilise pas cette façon de faire, on utilise les modèles et les gabarits.

3.2.7 Lui associer une url

Il faut associer à la vue que l'on vient de créer une URL, c'est-à-dire la requête associée. On crée donc un fichier `urls.py` dans le répertoire de l'application :

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

La page « index » est par convention (je crois) la page affichée lorsque l'on appelle la racine du projet ou d'une application, c'est pour cela que le premier argument de la fonction `path()` est une chaîne vide. Il faut maintenant relier les URL de l'application aux URL du projet, en modifiant `urls.py` du répertoire racine du projet :

```
from django.contrib import admin
from django.urls import include
from django.urls import path

urlpatterns = [
    path('monapplication/', include('monapplication.urls')),
    path('admin/', admin.site.urls),
]
```

La fonction `include()` permet de faire appel aux autres fichiers d'URL que l'on a créés, il faut toujours utiliser cette fonction, la seule exception étant l'administration. On peut tester en lançant un `runserver`. Si on va sur `localhost:8000`, on a une erreur 404 ! En se rendant à l'URL `localhost:8000/monapplication/`, Hello world ! apparaît.

3.2.8 Créer un modèle

Les modèles structurent la base de données et contiennent des métadonnées. Prenons un exemple musical et créons un modèle `Artist`, un modèle `Album` et un modèle `Song`. On les implémente en tant que classes dans le fichier `models.py` :

```
    class Artist(models.Model):
    name = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    bio = models.TextField()
```

```python
class Album(models.Model):
    name = models.CharField(max_length=100)
    artist = models.ForeignKey(Artist, on_delete=models.CASCADE)
    year = models.DateField()


class Song(models.Model):
    track_number = models.IntegerField()
    title = models.CharField(max_length=200)
    album = models.ForeignKey(Album, on_delete=models.CASCADE)

    @property
    def artist(self):
        return self.album.artist

    @property
    def year(self):
        return self.album.year
```

Les champs sont représentés par des différentes instance de classe `Field`, il en existe divers types. Le premier paramètre non nommé de ces instances permet sert à donner un nom plus lisible à ces champs (ici on l'a utilisé pour `annee`).

Une fois ces modèles créés, il faut les activer dans la base de données. Pour cela, il faut commencer par indiquer dans le fichier `settings.py` que l'on a créé une nouvelle application. On ajoute dans `INSTALLED_APPS` une référence vers la classe de configuration de l'application (qui se trouve dans le fichier `apps.py`). On se trouve donc avec, dans `settings.py` :

```python
INSTALLED_APPS = [
    'monapplication',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

On indique alors à Django que les modèles ont été modifiés :

```
$ python manage.py makemigrations monapplication # on n'est pas obligé de mettre le nom de
                                                 # l'application
```

Cette instruction crée les fichiers de migration qui contiennent les instructions des changements à faire en base de données. Il faut ensuite appliquer ces changements sur la base en appliquant ces migrations.

```
$ python manage.py migrate
```

Remarque Les deux étapes précédentes sont à répéter à chaque fois que l'on a modifié les modèles.

3.2.9 Interface administrateur

Il y a deux manières d'interagir avec la base de données :

1. Avec l'API Django (non développé ici) à travers le shell Python.
2. Avec l'interface graphique administrateur de Django.

L'interface administrateur est créée automatiquement. Pour y accéder, il faut commencer par créer un super-utilisateur.

```
$ python manage.py createsuperuser
```

Il suffit ensuite de suivre la procédure. Une fois cela fini, on peut se rendre (après un `runserver`) sur l'interface à l'adresse `localhost:8000/admin`. Une page de connexion apparaît, on se connecte avec les identifiants du compte super-utilisateur créé précédemment. Après connexion, on arrive sur la page d'administration. Cependant, nous n'avons toujours pas accès aux modèles que l'on a créés. Pour cela, il faut modifier le fichier `admin.py` de l'application :

```python
from django.contrib import admin
from .models import Song, Album, Artist

admin.site.register(Artist)
admin.site.register(Album)
admin.site.register(Song)
```

Ainsi, les modèles apparaissent dans un bloc correspondant à l'application concernée (figure 1). On peut donc créer une chanson, par exemple (figure 2). On voit que l'on peut renseigner tous les champs que l'on a créés dans nos modèles. L'outil d'administration est donc un outil très puissant qui nous permet d'agir sur la base de données graphiquement !

FIGURE 1 – Administration avec les modèles créés

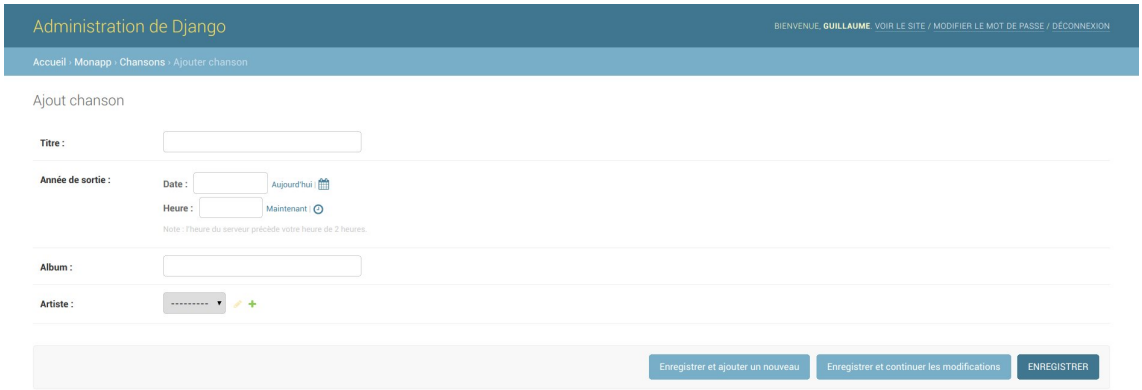


FIGURE 2 – Créer une chanson

Si l'on crée une chanson ou un artiste, on peut voir que dans la liste des objets, apparaît la mention "Chanson object" ou bien "Artiste object". En effet, on n'a pas défini de méthode de représentation dans nos modèles, on peut le faire comme suit :

```python
from django.db import models

class Artist(models.Model):
    name = models.CharField(max_length=100)
```

```
    genre = models.CharField(max_length=100)
    bio = models.TextField()

    def __str__(self):
        return self.name

class Album(models.Model):
    name = models.CharField(max_length=100)
    artist = models.ForeignKey(Artist, on_delete=models.CASCADE)
    year = models.DateField()

    def __str__(self):
        return self.name

class Song(models.Model):
    track_number = models.IntegerField()
    title = models.CharField(max_length=200)
    album = models.ForeignKey(Album, on_delete=models.CASCADE)

    @property
    def artist(self):
        return self.album.artist

    @property
    def year(self):
        return self.album.year

    def __str__(self):
        return self.title
```

En actualisant la page, les noms des artistes et titres de chansons apparaissent bien.

3.2.10 Introduction aux vues et gabarits

Créons plus de vues dans le fichier `views.py`. Par exemple des vues qui affichent des artistes et leurs chansons, des vues qui affichent des chansons et leurs paroles. On commence simplement :

```
def artist(request, artist_id):
    return HttpResponse("Vous êtes sur la page de l'artiste {}".format(artist_id))

def song(request, song_id):
    return HttpResponse("Vous êtes sur la page de la chanson {}".format(song_id))
```

Il faut ensuite aller renseigner les URL dans `urls.py`

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index),
    path('chanson/<song_id>/', views.song),
    path('artiste/<artist_id>/', views.artist)
]
```

Si on va sur la page `localhost:8000/monapplication/artiste/1`, on voit : « Vous êtes sur la page de l'artiste 1 ». En effet, Django analyse l'URL de la manière suivante :

1. `monapplication/` il va dans les URL de l'application monapplication
2. `artiste/1/` il cherche la ligne correspondante dans le fichier `urls.py`. Il trouve alors la ligne `artiste/<artiste_id>`, il appelle donc la vue `artist(request=<HttpRequest object>, artist_id=1)`.

On peut aussi créer des vues qui interagissent avec la base de données en utilisant l'API Django. Par exemple les pages racines d'artistes et de chansons pourraient les afficher dans l'ordre alphabétique. On aura finalement le fichier `views.py` suivant.

```
from django import HttpResponse
from django.shortcuts import render
from .models import Artist, Album, Song

def index(request):
    return HttpResponse("Hello world!")
```

```
def songs_list(request):
    songs_list = Song.objects.order_by('name')
    context = {
        "liste_chansons": songs_list
    }
    return render(request, 'songs/index.html', context)


def artists_list(request):
    artists_list = Artist.objects.order_by('nom')
    context = {
        "artists_list": artists_list
    }
    return render(request, 'artists/index.html', context)


def artist(request, artist_id):
    return HttpResponse("Vous êtes sur la page de l'artiste {}"
                        .format(artist_id))


def song(request, chanson_id):
    return HttpResponse("Vous êtes sur la page de la chanson {}"
                        .format(chanson_id))
```

On va utiliser des gabarits pour les deux premières vues. La fonction `render()` est un raccourci qui permet de renvoyer une réponse HTTP avec un gabarit. Les gabarits sont des fichiers HTML rangés dans le répertoire `templates` de l'application. Par exemple pour la liste d'artistes, on aura

```
monapplication/templates/monapplication/artists/index.html
```

Voici un simple gabarit pour la liste des artistes :

```
{% if artists_list %}
    <ul>
    {% for artist in artists_list %}
        <li><a href="/monapp/artist/{{ artist.id }}/">{{ artist.name }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>Aucun artiste.</p>
{% endif %}
```

Remarque Même si dans nos modèles, on ne crée pas d'attribut `id`, celui-ci est créé automatiquement.

Il ne faut pas oublier de mettre à jour `urls.py` :

```
urlpatterns = [
    path('', views.index),
    path('song/', views.songs_list),
    path('artist/', views.artists_list),
    path('song/<song_id>/', views.song),
    path('artist/<artist_id>/', views.artist)
]
```

Ainsi, si vous allez sur `localhost:8000/artist/`, la liste de vos artistes s'affichera, ou bien « Aucun artiste. » sinon.

3.2.11 Fichiers statiques

Les fichiers statiques sont rangés dans un répertoire nommé `static`, l'architecture est similaire à celle des gabarits. Imaginons que l'on veuille tout mettre en vert. On crée un fichier `style.css` dans le répertoire assoicé à l'application.

```
html {
    color: green;
}
```

On modifie ensuite par exemple le gabarit de la liste des artistes en ajoutant ce code au début :

```
{% load static %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />
```

La balise de gabarit `{% load static %}` génère l'URL absolue des fichiers statiques. Si on se rend à la page des artistes, tout est vert !

3.2.12 Thèmes abordés ici

Cela marque la fin du didacticiel. On s'intéresse maintenant aux différents aspects de Django :

1. Les modèles
2. Les vues
3. Les gabarits
4. Les formulaires
5. L'administration
6. Le déploiement

Ce n'est pas exhaustif, la meilleure façon de se documenter reste la documentation officielle (qui est d'ailleurs très bien faite).

3.3 Les modèles et les opérations sur la base de données

`django.db.models`

Comme indiqué dans le didacticiel :

1. Les modèles sont des classes filles de `models.Model` que l'on écrit dans le fichier `models.py` de l'application concernée.
2. Cette application doit être mentionnée dans la liste `INSTALLED_APPS` du fichier `settings.py`
3. Un modèle correspond à une table de la base de données. Les champs sont les attributs de la classe du modèle.

Plus d'informations Documentation Django 2 — Portail thématique sur les modèles

3.3.1 Les champs : attributs des modèles

`models.Field`

Un champ de modèle doit être une instance de la classe `Field` (où l'une de ses dérivées). Le choix du type de champ détermine le genre de donnée à stocker (par exemple des nombres ou du texte), le composants HTML qui sera utilisé dans le formulaire utilisé pour renseigner ce champs dans l'administration, et enfin les exigences minimales de validation de ce champ. Se référer aux liens dans la marge pour une documentation complète. Quelques types de champs génériques :

```
class CharField(max_length=None, **options)
```

Un champ pour une chaîne de caractère (courte ou longue). Le paramètre `max_length` règle la taille maximale de ce champ. Il en existe de plus précis pour les mails ou les URL, cf. la doc.

```
class DateField(auto_now=False, auto_now_add=False, **options)
```

Une date, représentée par la classe Python `datetime.date`. Le paramètre `auto_now` permet d'assigner automatiquement la date du jour à chaque enregistrement de l'objet, tandis que `auto_now_add` enregistre la date du jour à la création de l'objet.

```
class DateTimeField(auto_now=False, auto_now_add=False, **options)
```

Une heure, représentée par la classe Python `datetime.datetime`

```
class IntegerField(**options)
```

Un nombre entier compris entre -2147483648 et 2147483647.

```
class TextField(**options)
```

Un champ de texte, plus adapté que `CharField` pour les longs textes, car la zone de saisie est plus importante dans le formulaire (on ne détaille pas ici les composants HTML de formulaires, cf. la doc)

3.3.2 Les relations entre les modèles

models.ForeignKey

On peut aussi renseigner les relations entre les modèles (donc entre les tables de la base de données).

class ForeignKey(to, on_delete, **options)

Une relation plusieurs-à-un, (cf. le didacticiel, exemple des chansons qui ont l'artiste en `ForeignKey`). Cette classe exige la classe à laquelle le modèle et relié, et l'option `on_delete` : `models.CASCADE` si l'on veut que lorsque l'on supprime la `ForeignKey`, que tous les objets associés du modèle concerné soient supprimés, ou bien `SET_NULL` si l'on veut que les objets aient la valeur `null` à la place de la `ForeignKey` supprimée (dans ce cas il faut aussi renseigner `null=True`). Il y a d'autres possibilités (cf. la doc), voir des exemples.

class OneToOneField(to, on_delete, parent_link=False, **options)

Une relation un-à-un, dont le fonctionnement est similaire à `ForeignKey` ; voir des exemples.

class ManyToManyField(to, **options)

Une relation plusieurs-à-plusieurs, qui fonctionne de la même manière que `ForeignKey` (avec d'autres paramètres supplémentaires, cf. la doc) ; voir des exemples.

3.3.3 Les options des champs

**options

Les champs acceptent des `options`, en voici quelques unes (on note après un signe = la valeur par défaut) :

null=False

Si la valeur est `True`, alors Django stocke les valeurs vides dans la base de données avec `NULL`.

blank=False

Si la valeur est `True`, alors on peut laisser ce champ vide (cette option agit lors de la validation, ne pas confondre avec le paramètre précédent).

choices

C'est un itérable (tuple ou liste par exemple) constitué de couples `(A, B)` où `A` est la valeur réelle pour le modèle (valeur stockée en base) et `B` le texte affiché à l'utilisateur. On peut organiser en sous groupe comme dans cet exemple :

```
choix_media = [
    ('Audio', [('vinyl', 'Vinyl'), ('cd', 'CD')]),
    ('Vidéo', [('vhs', 'Cassette VHS'), ('dvd', 'DVD')]),
    ('unknown', 'Unknown'),
]
```

default

C'est la valeur par défaut du champ, cela peut être un objet ou un objet appelable (dans ce cas, il est appelé lors de la création de l'objet). Il ne peut pas s'agir d'un objet muable ! En effet, le système de noms de Python ferait que plusieurs instances de modèles seraient référencés vers une même instance de cet objet. Au lieu de cela, on crée une fonction qui retourne cet objet muable.

help_text

C'est une chaîne de caractère qui décrit le champ concerné, utilise lorsque l'on utilise la documentation générée automatiquement par Django.

primary_key

Si la valeur est `True`, alors ce champ représentera une clé primaire du modèle. Si aucun champ n'est renseigné, Django en crée un automatiquement : `id`.

verbose_name

Chaîne de caractère qui est le « nom verbeux » de l'attribut, c'est-à-dire un nom humainement compréhensible pour cet attribut. Il sera affiché à la place du nom de l'attribut dans le formulaire de l'administration (Django l'utilise en convertissant les soulignés en espaces). A l'exception des champs de relations, ce nom verbeux peut-être renseigné en tant que premier paramètre non nommé du champ. Pour ces exceptions, on doit nommer cette option.

3.3.4 Les métadonnées

Meta

On peut attribuer des métadonnées à un modèle grâce à une classe `Meta` incorporée dans la classe du modèle. C'est une classe facultative. Elle permet d'enrichir l'interface administrateur. On y renseigne plusieurs options, en voici quelques unes :

ordering="-order_date"

Définit une méthode de tri des instances d'un modèle. C'est une liste ou un tuple de chaîne de caractères. Chaque chaîne correspond à un nom de champ, préfixé par un - si l'on veut que le tri soit descendant (on ne met rien pour un tri ascendant). Les tris sont rangés dans la liste par ordre de priorité (Django trie par rapport au premier critère, puis second, etc.)

verbose_name, **verbose_name_plural**

Noms verbeux (même principe que pour les champs) respectivement dans le cas du singulier et dans le cas du pluriel.

db_table

Nom de la table dans la base de données. Par défaut, Django la nomme `application_modèle`.

3.3.5 Enregistrer des instances de modèles en base de données

`models.base.Model`

L'administration permet de facilement modifier la base de données à la main, mais on doit utiliser l'API Django si on veut modifier la base de données à partir des vues ou des modèles eux-mêmes (par exemple, en reprenant l'exemple du didacticiel, on peut imaginer que la sauvegarde d'un objet `Song` dans la base de données entraînera la création et sauvegarde de l'objet `Artist` associé s'il n'existe pas).

Pour insérer un objet dans la table de données, on commence déjà par l'instancier. Comme indiqué dans le didacticiel, tous les modèles héritent de la classe `Model`.

class Model

Tous les modèles doivent hériter de cette classe ! Ainsi on a accès à toutes les méthodes définies par défaut. Il est déconseillé de surcharger l'initialiseur `__init__()`, car cela pourrait entraîner des erreurs. Il est conseillé de créer un gestionnaire personnalisé (une classe qui hérite de `Manager`) et d'y écrire la méthode personnalisée.

Le constructeur accepte en paramètres nommés les différents champs qui le caractérisent. Une fois les objets créés, on peut modifier leurs attributs (donc leurs futurs champs). Pour les inclure dans la base de données, il faut les sauvegarder. On peut éventuellement commencer par valider les instances à enregistrer ; ce mécanisme est utilisé par les formulaires.

model_inst.clean_fields(exclude=None)

Cette méthode valide les champs de l'instance (typiquement, lève une erreur si un champ est vide, alors qu'on n'a pas le paramètre `blank=True`). L'option `exclude` permet d'indiquer des champs à ignorer lors de la validation. Si la validation échoue, lève une exception `ValidationError`.

model_inst.clean()

Une méthode à personnaliser pour effectuer des méthodes personnalisées sur notre modèle (effectuer automatiquement des valeurs à des champs, effectuer des validations qui demandent de vérifier plusieurs champs simultanément par exemple). Devrait lever une exception `ValidationError` si échoue.

model_inst.validate_unique(exclude=None)

Vérifie les contraintes d'unicité du modèle et lève une `ValidationError` si échoue.

model_inst.full_clean(exclude=None, validate_unique=True)

Exécute les trois méthodes précédentes (exécute `validate_unique` si le paramètre correspondant est `True`).

Pour sauvegarder un objet en base, on appelle la méthode `save()` :

model_inst.save(force_insert=False, force_update=False, using=DEFAULT_DB_ALIAS, update_fields=None)

Sauvegarde l'objet en base. Le paramètre `update_fields` permet de spécifier les champs à mettre à jour pour gagner en performance. Pour une explication complète de cette méthode, voir la documentation officielle.

3.3.6 Les gestionnaires

`models.manager.Manager`

Le gestionnaire est l'interface par laquelle on fait des requêtes à la base de données grâce à l'API des QuerySet (voir le didacticiel pour un exemple, dans les vues `artists_list` ou `liste_chansons`). Le gestionnaire permet aussi bien d'inspecter la base de données que de la modifier.

class Manager

Gestionnaire par défaut. Si aucun manager n'est défini explicitement dans un modèle, il sera automatiquement assigné à l'attribut de classe `objects`.

Le rôle du gestionnaire est de renvoyer un objet QuerySet adapté au modèle attaché. On peut appeler toutes les méthodes des QuerySet sur un gestionnaire.

```
    MyModel.objects # manager par défaut
MyModel.objects.all() # QuerySet contenant tous les objets MyModel
```

Il peut être utile d'écrire un gestionnaire personnalisé lorsque l'on effectue de manière récurrente un certain filtrage sur les objets. Imaginons que l'on veuille récupérer toutes les chansons d'un artiste, triées selon les albums et les numéros de pistes :

```
muse = Artist.objects.get(name="Muse")
songs_from_muse = Song.objects.filter(artist=muse).order_by("album__year", "track_number")
```

On peut définir une méthode spécifique pour cette requête dans un manager dérivé de `models.Manager` :

```
class SongManager(models.Manager):
    def from_artist(self, artist):
        return self.get_queryset().filter(artist=artist).order_by("album__year", "track_number")

class Song(models.Model):
    # corps de la classe

    # on associe le manager SongManager à ce modèle
    objects = SongManager()
```

Ainsi on peut retrouver les chansons triées avec :

```
muse = Artist.objects.get(name="Muse")
songs_from_muse = Song.objects.from_artist(muse)
```

Cela permet d'améliorer la maintenabilité du code.

3.3.7 Récupérer des informations de la base de données

`models.query.QuerySet`

Lorsque l'on veut récupérer des informations de la base de données, on utilise l'API Django. Différentes méthodes appliquées sur les gestionnaires des modèles permettent d'obtenir des objets `QuerySet` qui contiennent les informations désirées. En résumé, on utilise la syntaxe :

```
# schéma
query_set = Modèle.gestionnaire.methode()
# exemple
artists_list = Artiste.objects.all() #  objects est le nom par défaut du gestionnaire
# on peut aussi appeler ces méthodes sur des QuerySet
reversed_artists_list = artists_list.reverse()
```

Voici quelques méthodes qui renvoient un `QuerySet` :

`queryset.all()`
: Renvoie un `QuerySet` contenant toutes les entrées de la table.

`queryset.filter(**kwargs)`
: Renvoie un `QuerySet` contenant tous les objets répondant aux paramètres rentrés.

`queryset.exclude(**kwargs)`
: Renvoie un `QuerySet` contenant tous les objets sauf ceux répondant aux paramètres rentrés.

`queryset.reverse(**kwargs)`
: Renvoie le `QuerySet` dans l'ordre inverse.

`queryset.distinct(**kwargs)`
: Renvoie un `QuerySet` sans doublon.

Il y a plusieurs façons d'exploiter un `QuerySet` :

— Ils sont itérables :

```
# On imagine qu'on a déjà un QuerySet, on reprend le modèle du didacticiel
>>> artists_list
<QuerySet [<Artist: Muse>, <Artist: Keane>, <Artist: Imagine Dragons>]>
>>> for artiste in artists_list:
...     print(artiste.nom)
...
Muse
Keane
Imagine Dragons
```

— On peut facilement récupérer le nombre d'éléments

```
>>> len(artists_list)
3
```

— On peut convertir le `QuerySet` en liste :

```
>>> L = list(artists_list)
>>> L
[<Artist: Muse>, <Artist: Keane>, <Artist: Imagine Dragons]
```

Il existe des méthodes de `QuerySet` renvoyant autre chose qu'un `QuerySet`. En voici quelques unes.

`queryset.get(**kwargs)`
Renvoie *l'unique* objet répondant aux paramètres rentrés. S'il existe plusieurs objets possibles, ou zéro objet possible, cette méthode lève une exception (respectivement `MultipleObjectsReturned` et `DoesNotExist`). Si une requête renvoie un `QuerySet` singleton, on peut directement récupérer l'objet avec cette méthode sans paramètre (c'est risqué).

`queryset.get_or_create(defaults=None, **kwargs)`
Même comportement que ci-dessus, sauf que si l'objet n'existe pas, il est créé. Renvoie un tuple `objet, créé` où `objet` est l'objet créé ou charge, `créé` un booléen : `True` si l'objet a été créé et `False` sinon. Cette méthode permet d'alléger la syntaxe et d'éviter d'effectuer diverses vérifications. Les méthodes permettant d'agir sur la base de données sont détaillées plus loin.

`queryset.update_or_create(defaults=None, **kwargs)`
Essaie de trouver un objet correspondant aux paramètres et lui assigne les nouvelles valeurs rentrées, et crée l'objet s'il n'existe pas. Renvoie la même chose que la méthode précédente.

`queryset.last()`
Renvoie le dernier objet d'un `QuerySet` (si ce dernier n'est pas trié, il est automatiquement trié selon la clé primaire).

`queryset.first()`
Idem que la méthode précédente mais renvoie le premier objet.

`queryset.latest(*fields)`
Renvoie l'objet le plus récent selon le champ indiqué (on les indique de la même manière que pour `ordering`).

`queryset.earliest(*fields)`
Idem que la méthode précédente mais renvoie le plus ancien.

Remarque Toutes les méthodes de QuerySet sont disponibles sur les objets Manager.

3.4 Les requêtes HTTP : vues et URL

`django.http`

Comme indiqué dans le didacticiel :

1. Les vues sont des fonctions, rangées dans le fichier `views.py` de l'application.
2. Elles prennent en paramètre obligatoirement une requête Web (à laquelle peuvent s'ajouter des paramètres facultatifs) et renvoient une réponse Web.
3. La gestion des URL associées aux vues se fait dans le fichier `urls.py`.

Plus d'informations Documentation Django 2.2 — Ecriture des vues — Distribution des URL

3.4.1 Requêtes HTTP

`http.HttpRequest`

Les vues manipulent des requêtes HTTP et renvoient une réponse HTTP en utilisant les modèles et les gabarits. Elles prennent en paramètres une requête HTTP et d'éventuels paramètres supplémentaires dans l'URL (cf. le didacticiel). On commence par décrire ce qu'est une requête HTTP pour Django.

`class HttpRequest`
Lorsque Django reçoit une requête HTTP, il crée une instance de cette classe contenant les métadonnées associées à la requête. Elle est ensuite mise en premier paramètre de la vue appropriée (ce paramètre est par convention nommé `request`, cf. les exemples dans le didacticiel). Cette classe présente plusieurs attributs et méthodes dont voici une petite sélection :

request.path
Chaîne de caractères représentant le chemin vers la page demandée, sans `<protocole>://<hote>`. Par exemple : `/python/classes.html` pour `https://pycolore.fr/python/classes.html`.

request.method
Caine de caractères en majuscule représentant la méthode HTTP utilisée dans la requête. Cet attribut est utile dans les vues implémentées en tant que fonction :

```
def some_view(request):
    if request.method == "GET":
        # do something
    elif request.method == "POST":
        # do something else
```

request.GET
Dictionnaire contenant tous les données HTTP GET de la requête.

request.POST
Dictionnaire contenant toutes les données passées à la requête HTTP par la méthode POST.

3.4.2 Réponse HTTP

http.HttpResponse

On s'intéresse maintenant à ce que les vues renvoient : les réponses HTTP.

class HttpResponse
Cette classe hérite de `HttpResponseBase`. Les réponses HTTP ne sont pas créées automatiquement par Django, ce sont les vues qui les créent. *Une vue se doit de retourner une réponse HTTP!* Typiquement, on peut créer une réponse HTTP avec comme unique paramètre une chaîne de caractère qui sera le contenu de la page HTML retournée.

```
response = HttpResponse("Voici du texte de page Web.")
```

Quelques attributs :

HttpResponse.content
Une chaîne de caractères qui représente le contenu de la réponse.

HttpResponse.status_code
Code HTTP de la réponse, 200 par défaut (succès de la réponse). Des classes filles de `HttpResponse` ont une valeur par défaut différente.

Il existe aussi diverses méthodes (cf. la doc).

class HttpResponseNotFound(HttpResponse)
Exemple de classe fille de `HttpResponse`, identique à sa classe mère à l'exception de son code HTTP, ici, 404. Il en existe d'autres (voir la doc).

django.shortcuts.render

Comme premier paramètre (c'est-à-dire `content`), on peut utiliser une méthode de gabarit, la méthode `render`, qui permet de renvoyer du HTML en utilisant les gabarits. Il existe le raccourci `render` pour alléger le code :

render(request, template_name, context=None, content_type=None, status=None, using=None)
Fonction qui combine un gabarit avec dictionnaire de contexte et renvoie une `HttpResponse` avec le texte résultant. Deux paramètres obligatoires : `request` et `template_name`, le nom complet du gabarit à utiliser.

Les deux vues suivantes sont équivalentes (issus de la doc Django) :

```
from django.shortcuts import render
from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)
```

```
from django.http import HttpResponse
from django.template import loader
from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponse(template.render(context, request))
```

3.4.3 La gestion des URL

django.urls

Les URL (Uniforme Resource Locators) sont gérés dans les différents fichiers `urls.py`. Il y en a un dans chaque application et un dans le répertoire racine. Les URL sont configurées dans la liste `urlpatterns`. Voici les principales fonctions à utiliser.

`path(route, view, kwargs=None, name=None)`

Cette fonction est utilisée dans la liste `urlpatterns`. Elle prend deux paramètres obligatoires : la route, une chaîne de caractère qui correspond à une URL, et une vue (ou bien la fonction `include` qui appelle d'autres URL). La route peut contenir des éléments entre chevrons `<paramètre>` qui servent de paramètres pour la vue (rappel : les vues sont des fonctions).

`include(module, namespace=None)`

Cette fonction, en général utilisée comme second paramètre de la fonction `path()` prend en argument un module d'URL qu'il faut inclure après l'URL mise en premier paramètre.

3.5 Les gabarits

Un gabarit Django est un fichier texte ou une chaîne de caractères Python balisée en utilisant le langage de gabarit Django. Certaines expressions (étiquettes et variables) sont reconnues et inerprétées par le moteur de gabarit. Pour rendre un gabarit, celui-ci a besoin d'un dictionnaire de contexte : il remplace les variables par leur valeur et exécute les étiquettes. Le reste est maintenu tel quel.

Plus d'informations Documentation Django 2

3.5.1 La syntaxe des gabarits

Les variables Elles utilisent le dictionnaire de contexte pour afficher leur valeur correspondante. Les noms des variables sont les clés du dictionnaire. Dans le gabarit, les variables sont entourées de doubles accolades. Par exemple, le gabarit

```
La chanson {{ chanson }} a été écrite par {{ artiste }}.
```

avec le dictionnaire `{{'artiste': 'Muse', 'chanson': 'Starlight'}}` donnera :

```
La chanson Starlight a été écrite par Muse.
```

On accèdes aux attributs d'instances, aux indices de listes, aux clés de dictionnaires par une notation pointée.

```
{{ dico.clé }}
{{ objet.attribut }}
{{ liste.indice }}
```

Si la valeur de la variable est une fonction (ou n'importe quel objet appealble), il sera appelé sans paramètre et le résultat retourné sera utilisé.

Balises intégrées

Les balises Elles permettent de faire diverses choses, comme utiliser des boucles logiques ou insérer d'autres gabarits. Leur nom sont entourés de `{%` et `%}`. Certaines balises sont orphelines, les autres s'utilisent comme ceci :

```
<!-- Exemple de balise nommée balise. -->
{% balise %}
<!-- Contenu -->
{% endbalise %}
```

`block` (oprheline)

Définit un bloc pouvant être surchargé par des gabarits enfants.

`comment`

Ignore ce qui est compris entre `{% comment %}` et `{% endcomment %}`.

`if`

Evalue une variable et, si celle-ci vaut **`True`** (ie est différent de **`False`**, **`''`** ou **`None`**), affiche le bloc correspondant.

```
{% if var_1 %}
    <!-- contenu -->
{% elif var_2 > var_3 %}
    <!-- contenu -->
{% elif var_4 and var_5 or var_6 %}
    <!-- OR est prioritaire sur AND.
    Utiliser des parenthèses est une erreur de syntaxe,
    utiliser des IF imbriqués si nécessaire. -->
{% elif var_7 in var_8 %}
    <!-- contenu -->
{% elif var_9 is not var_10 %}
    <!-- contenu -->
{% endif %}
```

`firstof` (orpheline)

Affiche le premier paramètre qui ne vaut pas **`False`** et rien dans le cas où aucun paramètre n'est vrai. On peut ajouter un dernier paramètre si aucun n'est validé. On peut utiliser le mot clé **`as`** pour stocker la variable (voir `cycle` un peu plus bas).

```
{% firstof var1 var2 var3 "dernier recours" %}
<!-- est l'équivalent de -->
{% if var1 %}
    {{ var1 }}
{% elif var2 %}
    {{ var2 }}
{% elif var3 %}
    {{ var3 }}
{% else %}
    "dernier recours"
{% endif %}
```

`for`

Effectue une boucle sur chaque élément d'une liste. On peut ensuite utiliser cet élément comme variable. Exemple :

```
<ul>
{% for artiste in artists_list %}
    <li>{{ artiste.nom }}</li>
{% endfor %}
</ul>
```

On peut ajouter une balise `{% empty %}` pour afficher du contenu lorsque la liste est vide (ou n'existe pas).

`cycle` (orpheline)

Affiche un de ses paramètres à chaque apparition de la balise : le premier, puis le deuxième, et ainsi de suite ; et revient au début lorsque tous les paramètres ont été utilisés. On peut mélanger variables et chaînes de caractères, par exemple :

```
{% for elemt in liste %}
    <div class="{% cycle 'chaine_1' variable_de_la_chaine_2 'chaine_3' %}">
        <!-- contenu -->
    </div>
{% endfor %}
```

A la première itération, `"chaine_1"` sera utilisé, puis la chaîne contenu dans `variable_de_la_chaine_2`, puis `"chaine_3"`. Il est également possible de sauvegarder temporairement le paramètre dans une variable que l'on peut réutiliser plus loin.

```
{% for elemt in liste %}
    <div class="{% cycle 'chaine_1' 'chaine_2' as chaine %}">
        <!-- contenu -->
    </div>
    <div class="{{ chaine }}">
    </div>
{% endfor %}
```

3.5.2 Utiliser les gabarits dans les vues

3.6 Les formulaires

Les formulaires en Django facilitent la création de formulaires HTML et peuvent facilement se relier aux modèles. Ils contiennent le traitement des données reçues lorsqu'on reçoit une requête `POST` d'un formulaire.

Plus d'informations Documentation Django 2 — Les formulaires

3.6.1 Requêtes GET et requêtes POST

3.6.2 Construction d'un formulaire

`forms.Form` Un formulaire doit hériter de la classe `forms.Form`. De la même manière que pour les modèles, on y renseigne un certain nombres de champs (ce sont les mêmes champs que pour les modèles).

3.6.3 Utilisation du formulaire

4 WSGI

Ce module permet de faire tourner Python sur un serveur comme Apache. On voit ici comment déployer une application Flask ou Django avec Apache 2.4 sur un système d'exploitation Debian 9 (Stretch). On considère qu'Apache est connu. Premièrement, installer le module d'Apache pour Python 3 :

```
$ sudo apt install libapache2-mod-wsgi-py3
```

4.1 Déploiement de Flask

4.1.1 Hôte virtuel

Notre application respecte l'arborescence :

```
app/
|---flaskapp.wsgi
|---FlaskApp/
    |---__init__.py
    ...
```

On crée un hôte virtuel pour notre application Flask.

```
$ cd /etc/apache2/sites-availables
```

```
<VirtualHost *:80>
    ServerName domain.com
    ServerAdmin youremail@email.com
    WSGIScriptAlias / /var/www/chemin/votre/app/flaskapp.wsgi

    <Directory /var/www/chemin/votre/app/FlaskApp>
        Require all granted # signifie que toute requête est acceptée
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/FlaskApp-error.log
    LogLevel warn
    CustomLog ${APACHE_LOG_DIR}/FlaskApp-access.log combined
</VirtualHost>
```

On peut prendre n'importe quel nom pour les logs.
Voici le contenu de notre fichier WSGI :

```
#!/usr/bin/python
# on dit à Debian d'utiliser python3

import sys
import logging

logging.basicConfig(stream=sys.stderr)
sys.path.insert(0,"/var/www/chemin/votre/app/")

# en assumant que l'on a app=Flask(__name__)
from FlaskApp import app as application
```

Voilà ! Normalemnt ça marche :) À tester avec un Hello World.

5 `win10toast` — Notifications Windows 10

6 `pytaglib` — Tags de fichiers audio

Ce module premet l'accès et l'écriture aux métadonnées (tags) de fichiers audio. Ce module utilise `taglib`.

6.1 Installation

Il faut au préalable posséder le paquet de développement de Python, `python3-devel` sur Fedora et le paquet de développement de taglib, `taglib-devel` sur Fedora. (Pour Debian, il suffit de remplacer `devel` par `dev`.)

6.2 Utilisation

```
>>> import taglib
>>> song = taglib.File("chemin/vers/le/fichier.mp3")
>>> song.tags
{'ARTIST': ['Foo'], 'TITLE': ['Bar']}
>>> song.length # durée en secondes
60
>>> song.tags["ARTIST"] = ["Baz"] # toujours sous forme de liste
>>> del song.tags["TITLE"]
>>> song.save()
>>> song.tags
{'ARTIST': ['Baz']}
```

Principaux tags :

- `TITLE` : Titre du morceau
- `ARTIST` : Artsite du morceau
- `ALBUM` : Album
- `COMPOSER` : Compositeur (utilse pour la musique classique)
- `TRACKNUMBER` : Numéro de piste
- `DATE` : Année de sortie

7 `autopy` — Tâches automatiques

Cinquième partie
Conventions des Python Enhancement Proposals

1 PEP 8 : Conventions de style du code Python

2 PEP 257 : Convention des docstrings

Index