Mémo Python

Table des matières

Intro

Permalien

Je rédige ce doc au fur et à mesure que j'en apprends sur Python… Bonne lecture ! (On peut rêver, peut-être ne suis-je pas le seul à le lire.)

Ne sont pas reprises les opérations « basiques » (boucles, manipulation des listes). On s'intéresse directement à la programmation orientée objet. Ensuite sont détaillés quelques fonctionnalités apportées par des modules. Une partie sur Django est particulièrement plus développée.

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

De nombreux liens sont disponibles. De manière générale :

1. Des liens vers les documentations sont fournis.
2. Des liens plus précis sont fournis en cliquant sur les termes dans la marge. <– En cours ! (càd incomplet)
3. Pour le reste, tout ce qui est en bleu = lien (sauf dans les cadres de code).

Nota Bene Ce document est encore incomplet, des sections sont susceptibles d'être vides.

Première partie

Programmation orientée objet en Python

1 Constantes et types natifs

Quelques constantes sont définies par Python comme `True` et `False`. De la même manière, quelques classes sont définies par défaut, appeléss types natifs, comme :

1. Types booléennes
   (a) Booléen (`bool`)
   (b) Opérations booléennes (`and, or, not`)
2. Types numériques
   (a) Entier (`int`)
   (b) Flottant (`float`)
   (c) Complexe (`complex`)
3. Types séquentiels
   (a) Liste (`list`)
   (b) Tuple (`tuple`)
   (c) Range (`range`)
4. Chaîne de caractères (`str`)
5. Séquences binaires (`bytes, bytearray, memoryview`)
6. Ensemble (`set` (muable), `frozenset` (non muable))
7. Dictionnaire (`dict`)

La liste n'est pas exhaustive, des compléments sur les types natifs sont disponibles dans la documentation.

Documentation Documentation Python 3 : constantes natives, types natifs

2 Classes

Les classes permettent des créer des objets appelés instances) qui partagent des caractéristiques de leur classe. Une classe correspond donc à un type, ou plutôt à un gabarit d'objet.

Documentation Introduction OpenClassrooms, Documentation Python 3, Wikilivres

2.1 Structure d'une classe

Les objets d'une classe partagent des caractéristiques communes à la classe : des attributs et des méthodes (des fonctions qui agissent sur leur attributs). Les objets sont créés gâce à une méthode spéciale appelée constructeur.

2.1.1 Création

Pour créer une classe, la syntaxe est la suivante :

```
class MaClasse:
    pass
```

Si la classe hérite d'une classe mère, alors il faut l'ajouter en argument :

```
class MaClasse(ClasseMere):
    pass
```

2.1.2 Initialiseur

L'initialiseur est une méthode spéciale appelée `__init__`, il prend en argument `self` (toutes les méthodes de la classe prennent en argument `self` qui est en fait l'instance en question) et tous les paramètres nécessaires à l'initialisation de l'instance. Il ne s'agit pas du constructeur (même si on l'appelle parfois ainsi par abus de langage) : il ne crée pas à proprement parler l'instance, mais agit sur celle-ci lorsqu'elle vient d'être créée. Le « véritable » constructeur est la méthode `__new__`, il n'est en général pas nécessaire de l'implémenter, sauf par exemple lorsque l'on crée des métaclasses ou si l'on veut créer des classes qui héritent des types natifs. L'initialiseur est appelé automatiquement lors que l'on crée l'objet (après le constructeur).

```
class MaClasse:
    CONSTANTE = ...
    def __init__(self, att1, att2):
        """Initialiseur"""
        self.attribut_1 = att1
        self.attribut_2 = att2
        self.attribut_3 = MaClasse.CONSTANTE
```

Ici, les deux premiers attributs sont personnalisables lors de la création des objets alors que le dernier est commun à tous. `MaClasse.CONSTANTE` est une variable de classe. Pour créer un objet on écrit simplement :

```
objet = MaClasse(att1, att2)
```

2.1.3 Méthodes

Les méthodes se définissent comme des fonctions, elles agissent en général sur les instances de la classe. Elles doivent prendre `self` en argument :

```
class MaClasse:
    def __init__(self):
         pass

    def methode(self, arg1, arg2):
         pass
```

Ensuite on les appelle de la manière suivante :

```
objet.methode(arg1, arg2)
```

2.2 Héritage

2.2.1 Principe

L'héritage est un moyen de créer des classes dérivées (classes filles) d'une classe source (classe mère). Une classe fille hérite de toutes les méthodes et variables de sa classe mère. Pour créer une classe fille, on utilise la syntaxe suivante.

```
class Mere:
    pass

class Fille(Mere):
    pass
```

Il est possible 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
```

Documentation OpenClassrooms, Documentation Python 3, Programiz

2.2.2 Ordre de résolution de méthode

2.2.3 Classe mère `object`

2.3 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. Les propriétés sont un cas particulier des descripteurs.

2.3.1 Définition d'une propriété

On crée les propriétés en utilisant des décorateurs. Elles contiennent un accesseur, un mutateur, un destructeur et une aide (docstring de l'accesseur).

Les propriétés sont aussi un moyen de simuler des attributs privés : pour simuler un attribut privé, on précède son nom d'un souligné. Ainsi, on appelle cet attribut sans le souligné dans le code grâce aux propriétés. Par convention, on n'agit pas sur les attributs qui commencent par un souligné en Python.

Exemple

```
class MaClasse:
    def __init__(self):
        self._attribut = 'Je suis un attribut'

    @property
    def attribut(self):
        """Propriété 'attribut'."""
        print("Accès à l'attribut")
        return self._attribut

    @attribut.setter
    def attribut(self, valeur):
        print("Modification de l'attribut")
        self._attribut = valeur

    @attribut.deleter
    def attribut(self):
        print('Adieu :(')
        del self._attribut
```

On utilise la propriété de la manière suivante :

```
>>> instance = MaClasse()
>>> instance.attribut
Accès à l'attribut
'Je suis un attribut'
>>> instance.attribut = 'Ah bon ?'
Modification de l'attribut
```

```
>>> del instance.attribut
Adieu :(
>>> help(MaClasse.attribut)
Help on property:

    Propriété 'attribut'.
```

Documentation Documentation Python 3, Priorités entre propriété et méthodes spéciales

2.3.2 Généralisation : les descripteurs

On dit qu'un objet est un descripteur s'il possède au moins une méthode `__get__` (accesseur), `__set__` (mutateur), ou `__delete__` (destructeur).

Exemple Exemple d'implémentation

```python
class Attribut:
    def __get__(self, inst, insttype):
        print("Accès à l'attribut")
        return inst._attribut

    def __set__(self, inst, valeur):
        print("Modification de l'attribut")
        inst._attribut = valeur

class MaClasse:
    def __init__(self):
        self._attribut = 'Je suis un attribut'

    attribut = Attribut()
```

Documentation Documentation Python 3

2.4 Méthodes statiques et méthodes de classes

2.4.1 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 renvoie à l'instance elle même. Lorsque l'on appelle une telle méthode sur une instance comme ceci : `instance.methode(<arguments>)`, Python exécute en fait `Classe.methode(instance, <argume`

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

Exemple

```python
class Maths:

    def addition(x, y):
        return x + y

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

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

On choisit ici de grouper trois fonctions car elles sont logiquement liées. Elles n'influent pas les instances donc elles ne prennent pas `self` en argument. Si l'on appelle ces méthodes sur une instance, une exception sera levée car Python entrera automatiquement l'argument `self` (donc en tout trois arguments) alors que les méthodes n'en prennent que deux. Pour remédier à cela, on les décore avec `@staticmethod`. On peut maintenant les appeler indifféremment sur la classe ou sur des instances.

2.4.2 Méthode de classe

Lorsque l'on veut manipuler des variables de classe et non des attributs d'instances, on crée des méthodes de classe. Celles-ci prennent la classe en premier argument, par convention on le note `cls` ; elles ne prennent logiquement pas `self` comme argument. Cette méthode est donc évaluée sur la classe. Sans autre modification, on ne peut peut appeler cette méthode que sur les instances car Python attend l'argument `cls`. Pour pouvoir appeler cette méthode sur la classe (logique car c'est une méthode de classe), on la décore avec `@classmethod`.

2.4.3 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 ses méthodes. 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.

Documentation Méthode statique sur Programiz, Méthode de classe sur Programiz, StackOverflow

2.5 Méthodes spéciales

Les méthodes spéciales sont déjà définies par défaut dans Python mais on peut les personnaliser. Elles sont reconnaissables par leur typographies : leur nom commence et se termine par deux soulignés. On en introduit quelques unes seulement ici.

Documentation Documentation Python 3, OpenClassrooms

2.5.1 Construction, initialisation et destruction

Le constructeur est la méthode `__new__`. C'est une méthode statique qui prend en argument la classe de l'objet à instancier (`cls`) et les autres arguments qui seront en paramètres de l'initialiseur ; il doit retourner un objet (l'instance à créer). `__init__` a déjà été décrit précédemment (contrairement au constructeur, cette méthode ne retourne rien). En pratique, on n'implémente pas la méthode `__new__` sauf dans certains cas.

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)
            return cls.instance
        else:
            raise TypeError(r"Cette classe singleton possède déjà une instance : {cls.instance}")

    def __del__(self):
        Singleton.instance = None

    def __init__(self, *args, **kwargs):
        pass
```

Pour détruire un objet, on définit la méthode `__del__`. On l'appelle comme ceci :

```
del objet
```

2.5.2 Représentation et chaine de caractère d'un objet

Il existe deux méthodes spéciales nommées `__repr__` et `__str__` qui sont appelées lorsque l'on exécute `repr(objet)` ou `return objet`, et quand on exécute `str(objet)` ou bien `print(objet)`. La fonction `__repr__` est donc utilisée lorsque l'on veut avoir accès à la représentation d'un objet, tandis que `__str__` permet de présenter l'objet de manière plus élégante en chaîne de caractères. Ces deux fonctions prennent en argument `self`. Lorsque la méthode `__str__` n'est pas définie, Python appelle la fonction de représentation à la place.

Exemple L'exemple suivant

```
class MaClasse:
    def __init__(self):
         self.attribut = 'Exemple'

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

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

permet de faire :

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

2.5.3 Accesseur et mutateur

Lorsque Python essaie d'accéder à un attribut, il appelle en premier la méthode spéciale `__getattribute__`, puis il appelle les descripteurs s'il sont définis. Lorsque l'on veut modifier un attribut, c'est la méthode spéciale `__setattr__` puis les descripteurs qui sont appelés. Si on essaie d'accéder à un attribut non défini, Python appelle en guise de dernière chance la méthode `__getattr__`. On peut personnaliser cette fonction de manière à ce qu'elle envoie une erreur, ou bien à ce qu'elle redirige vers un autre attribut ou effectue un calcul.

Exemple

```
class MaClasse:
    def __init__(self)
        self.a = int()

    def __getattribute__(self, attribut):
```

```
        print("J'accède à l'attribut {}...".format(attribut))
        return object.__getattribute__(self, attribut)

    def __getattr__(self, attribut):
        print("L'attribut {} est inaccessible !".format(attribut))

    def __setattr__(self, attribut, valeur):
        object.__setattr__(self, attribut, valeur)
        print("L'attribut a été changé !")
        # Il est nécessaire d'appeler la méthode par défaut, car appeler self.__setattr__
        # donnerait une récursivité infinie. En fait, on ne sait à ce stade pas comment
        # Python change concrètement la valeur de l'attribut.
```

On note que l'on utilise les méthodes spéciales de la classe `object` (méthodes par défaut) car appeler `self.__getattribute__` ou `self.__setattr__` donnent une récursivité sans fin ! Cela permet de faire :

```
>>> objet = MaClasse()
L'attribut a été changé !
>>> objet.b
L'attribut b est inaccessible !
>>> objet.a
J'accède à l'attribut a...
0
>>> objet.attribut = 1
L'attribut a été changé !
>>> objet.a
J'accède à l'attribut a...
1
```

Il existe aussi `__delattr__` qui prend en arguments `self` et le nom de l'attribut. Cette méthode est appelée lorsque l'on effectue `del objet.attribut`. Lors de l'écriture de la méthode, il faut utiliser `object.__delattr__` de la même manière que l'on utilise `object.__setattr__` pour `__setattr__` ou `object.__getattribute__` pour `__getattribute__`.

2.5.4 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__`).

3 Fonctions et objets exécutables

La façon la plus basique de définir une fonction est d'utiliser le mot-clé `def`. Lorsque l'on appelle, on spécifie des arguments aux paramètres de la fonction.

```
fonction(argument_de_paramètre_non_nommé, paramètre_nommé=argument)
```

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

- Lorsque l'on écrit les paramètres pour capter tous les paramètres possibles.
- Pour renseigner les arguments.

Exemple

```
# Cette fonction accepte tous les arguments
def fonction(*args, **kwargs):
    pass

# Celle-ci en accepte 4. On va tester sur celle-ci l'unpacking.
def fonction(par1, par2, par3, par4):
    for (arg, value) in locals().items():
        print(arg, ':', value)

arg1, arg2, arg3, arg4 = 1, 2, 3, 4
tuple_args = (arg1, arg2)
dict_args = {'par3': arg3, 'par4': arg4}

# Testons l'unpacking
fonction(*tuple_args, **dict_args)

# Sortie
# par4 : 4
# par3 : 3
# par2 : 2
# par1 : 1
```

4 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
```

Remarque La fonction `__contains__()` est définie chez les itérateurs, mais ceux-ci ne sont pas pas des conteneurs ! Implémenter cette méthode n'est donc pas une condition suffisante pour qu'un objet soit un conteneur (voir la section sur les itérateurs).

La plupart des conteneurs possède 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*).

4.1 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]
```

4.2 Objets séquentiels

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

5 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
```

6 Générateurs

6.1 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
```

6.2 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

7 Coroutines

8 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 retourner un objet que l'on peut appeler en écrivant `objet()` (avec d'éventuels arguments), on appelle ce type d'objet un « exécutable ». Le décorateur est bien sûr lui même un exécutable. Si on le définit comme une classe, on doit définir la méthode `__call__` qui permet de rendre ses instances exécutables.

Documentation 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 exécutable.

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 exécutable 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
```

Remarques On voit dans cet exemple que l'on peut définir des fonctions dans les définitions de fonctions. La mention `*args` fait référence à tous les arguments non nommés que l'on a entrés (c'est un tuple, par exemple `(arg1, arg2)`). La mention `**kwargs` fait référence aux arguments nommés (c'est un dictionnaire). Ainsi on est sûr de récupérer tous les arguments.

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.

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.

```python
class MaClasse:
    pass

print(type(MaClasse)) # <class 'type'>
```

10.2 La métaclasse `type`

10.3 Application pythonique : 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 popriété la future propriété de classe.

Exemple Un exemple simple

```python
class MaMetaclasse(type):
    @property
    def propriete(self):
        return self._propriete
    @propriete.setter
    def propriete(self, value):
        self._propriete = value

class MaClasse(metaclass=MaMetaclasse):
    _propriete = 5

print(MaClasse.propriete) #5
```

Deuxième partie
Librairie standard

11 Classes abstraites : abc, collections.abc

12 re

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

Documentation Documentation Python 3

12.1 Ecrire 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` » équivaut à `[a-zA-Z0-9_]`.
- « `\W` » désigne tout caractère non alpha-numérique.
- « `\d` » équivaut à `[0-9]`.
- « `\D` » désigne tout caractère non numérique.
- « `\s` » désigne un espace.

Place du motif dans la chaine

- « `^` » (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 apparaitre `n` fois.
- « `\{n,m\}` » indique que le caractère précédent doit apparaitre entre `n` et `m` fois.
- « * » indique que le caractère précédent n'apparait pas ou apparait sans maximum d'occurences (`ab*` correspond à `a`, `ab`, ou bien `abbbbbb`, etc.).
- « + » indique que le caractère précédent apparait au moins une fois (`ab+` correspond à `ab`, `abb`, ou bien `abbbbbb`, etc.).
- « ? » indique que le caractère précédent apparait 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 chaine 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.

12.2 Méthodes

re.compile

On compile une expression régulière en utilisant la fonction `compile`. Cette fonction retourne un objet expression régulière (regex) 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 chaine de caractère pour éviter d'avoir à écrire `\\` au lieu d'un unique `\`.

regex.finditer

On peut rechercher toutes les occurences 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.

Script

```
chaine = r"Je suis une phrase. Moi aussi"
regex = re.compile(r"[A-Z]\w*\s?(?:\w+\s?)*.")
resultats = regex.finditer(chaine)
while True:
    try:
        print(next(resultats).group(0))
    except:
        break
```

Sortie

```
Je suis une phrase.
Moi aussi.
```

regex.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'occurences 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>`.

Exemple

```
pass
```

13 datetime

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

14 turtle

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

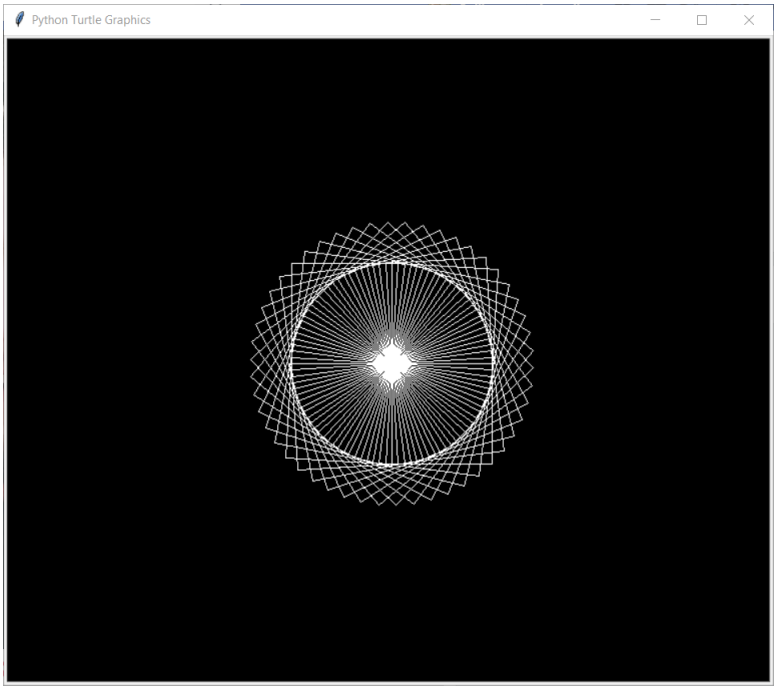
```
import turtle

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

Documentation Documentation Python 3, Wikilivres

15 ctypes

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

15.1 Boites de dialogue

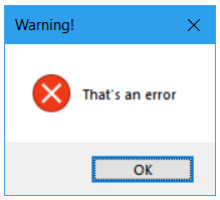
Le module ctypes peut servir à faire apparaître des boites de dialogue. On peut modifier le comportement du script Python en fonction du bouton appuyé car la fonction faisant apparaître ces boites renvoie un entier qui dépend du bouton appuyé. Diverses options sont disponibles :

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

# To also change icon, add these values to previous number
# 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

15.2 keyboard

15.3 os

15.4 sys

15.5 threading

indexthreading

Troisième partie
Modules à télécharger

16 virtualenv

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
```

Documentation Documentation de virtualenv, informations supplémentaires

17 django

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

Documentation Documentation officielle de Django 2.0 Tutoriel de la documentation

17.1 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.

17.2 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
```

17.2.1 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.

17.2.2 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
```

17.2.3 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

17.2.4 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.

17.2.5 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

17.2.6 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.

17.2.7 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.

17.2.8 Créer un modèle

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

```
from django.db import models

class Artiste(models.Model):
    nom = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    bio = models.TextField(max_length=1000)

class Chanson(models.Model):
    titre = models.CharField(max_length=200)
    annee = models.DateTimeField('année de sortie')
    album = models.CharField(max_length=200)
    artiste = models.ForeignKey(Artiste, on_delete=models.CASCADE)
```

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` :

```
INSTALLED_APPS = [
    'monapplication.apps.MonapplicationConfig',
    '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 est l'analogue d'un `git stage`, il faut exécuter ensuite la méthode `migrate` pour appliquer les migrations (analogue à `git commit`).

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

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

17.2.9 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 :

```
from django.contrib import admin
from .models import Chanson
from .models import Artiste

admin.site.register(Artiste)
admin.site.register(Chanson)
```

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 !
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 :

```
from django.db import models

class Artiste(models.Model):
    nom = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    bio = models.TextField(max_length=1000)

    def __str__(self):
        return self.nom

class Chanson(models.Model):
    titre = models.CharField(max_length=200)
```

FIGURE 1 – Administration avec les modèles créés

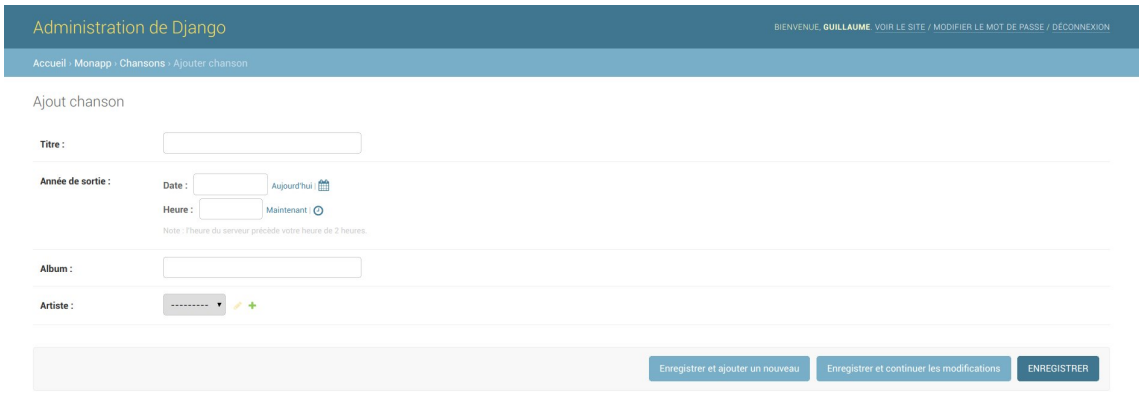


FIGURE 2 – Créer une chanson

```
    annee = models.DateTimeField('année de sortie')
    album = models.CharField(max_length=200)
    artiste = models.ForeignKey(Artiste, on_delete=models.CASCADE)

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

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

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

def chanson(request, chanson_id):
    return HttpResponse("Vous êtes sur la page de la chanson {}".format(chanson_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/<chanson_id>/', views.chanson),
    path('artiste/<artiste_id>/', views.artiste)
]
```

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 `artiste(request=<HttpRequest object>, question_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 Artiste
from .models import Chanson


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


def liste_chanson(request):
    liste_chansons = Chanson.objects.order_by('nom')
    context = {
        "liste_chansons": liste_chansons
    }
    return render(request, '/monapplication/chansons/index.html', context)


def liste_artiste(request):
    liste_artistes = Artiste.objects.order_by('nom')
    context = {
        "liste_artistes": liste_artistes
    }
    return render(request, '/monapplication/artistes/index.html', context)


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


def chanson(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/artistes/index.html
```

Voici un simple gabarit pour la liste des artistes :

```
{% if liste_artistes %}
    <ul>
    {% for artiste in liste_artistes %}
        <li><a href="/monapp/artiste/{{ artiste.id }}/">{{ artiste.nom }}</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('chanson/', views.liste_chanson),
    path('artiste/', views.liste_artiste),
    path('chanson/<chanson_id>/', views.chanson),
    path('artiste/<artiste_id>/', views.artiste)
]
```

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

17.2.11 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 !

17.2.12 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).

17.3 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.

Documentation Documentation Django 2 — Portail thématique sur les modèles

17.3.1 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)

17.3.2 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.

17.3.3 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 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 exécutable (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.

17.3.4 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`.

17.3.5 Les gestionnaires

`models`
`.manager`
`.Manager`

Le gestionnaire est l'interface par laquelle on fait des requêtes à la base de données avec l'API Django (voir le didacticiel pour un exemple, dans les vues `liste\_artistes` ou `liste\_chansons`. Le gestionnaire permet aussi bien d'inspecter la base de données que de la modifier.

class Manager
: Gestionnaire de la classe concernée. Par défaut, on a (on ne l'écrit pas mais c'est comme-ci) :

```
class Modèle:
    # ...
    objects = models.Manager()
```

On peut définir un gestionnaire personnalisé (par exemple pour une classe `Personne`, on peut le nommer `personnes`), dans ce cas, `Modèle.objects` produira une exception `AttributeError`. Si l'on veut définir un gestionnaire avec des méthodes personnalisées, il suffit de créer une classe héritant de `Manager`.

17.3.6 Modifier la 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 Chanson dans la base de données entraînera la création et sauvegarde de l'objet Artiste 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.

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. La première étape consiste à valider l'instance.

`Model.clean_fields(self, 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.clean(self)`

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.validate_unique(self, exclude=None)`

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

`Model.full_clean(self, exclude=None, validate_unique=True)`

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

17.3.7 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
liste_artistes = Artiste.objects.all() #  objects est le nom par défaut du gestionnaire
# on peut aussi appeler ces méthodes sur des QuerySet
liste_artistes_inversee = liste_artistes.reverse()
```

Voici quelques méthodes qui renvoient un `QuerySet` :

`gestionnaire.all()`

Renvoie un `QuerySet` contenant toutes les entrées de la table.

`gestionnaire.filter(**kwargs)`

Renvoie un `QuerySet` contenant tous les objets répondant aux paramètres rentrés.

`gestionnaire.exclude(**kwargs)`

Renvoie un `QuerySet` contenant tous les objets sauf ceux répondant aux paramètres rentrés.

`gestionnaire.reverse(**kwargs)`

Renvoie le `QuerySet` dans l'ordre inverse.

`gestionnaire.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
>>> liste_artistes
<QuerySet [<Artiste: Muse>, <Artiste: Keane>, <Artiste: Imagine Dragons>]>
>>> for artiste in liste_artistes:
...     print(artiste.nom)
...
Muse
Keane
Imagine Dragons
```

— On peut facilement récupérer le nombre d'éléments

```
>>> len(liste_artistes)
3
```

— On peut convertir le `QuerySet` en liste :

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

— Il existe des méthodes qui évaluent un `QuerySet` et qui renvoient autre chose qu'un `QuerySet`. En voici quelques unes.

`query_set.get(**kwargs)`
Renvoie *l'unique* objet répondant aux paramètres rentrés. S'il existe plusieurs objets possibles, ou zéro objet possible, cette fonction renvoie une erreur (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).

`query_set.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'avoir recours à `try: ... except: ...`. Les méthodes permettant d'agir sur la base de données sont détaillées plus loin.

`query_set.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.

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

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

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

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

17.4 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`.

Documentation Documentation Django 2 — Ecriture des vues — Distribution des URL

17.4.1 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 (voir la doc).

17.4.2 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))
```

17.4.3 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.

17.5 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.

Documentation Documentation Django 2

17.5.1 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 : `\{\{\} et \mintinline{python}{\}\}`. 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 exécutable), 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 \mintinline{python}{\%\}`. 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 liste_artistes %}
    <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 %}
```

17.5.2 Utiliser les gabarits dans les vues

17.6 Les formulaires

Documentation Documentation Django 2 — Les formulaires

18 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
```

18.1 Déploiement de Flask

18.1.1 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.

19 twilio

20 win10toast

21 splinter

22 pylint

22.1 autopy

Quatrième partie
Conventions des Python Enhancement Proposals

23 PEP 8 : Conventions de style du code Python

24 PEP 257 : Convention des docstrings