# Les filtres en DJANGO

[**Django**](https://www.djangoproject.com/)

* [**OVERVIEW**](https://www.djangoproject.com/start/overview/)
* [**DOWNLOAD**](https://www.djangoproject.com/download/)
* [**DOCUMENTATION**](https://docs.djangoproject.com/)
* [**NEWS**](https://www.djangoproject.com/weblog/)
* [**COMMUNITY**](https://www.djangoproject.com/community/)
* [**CODE**](https://github.com/django/django)
* [**ISSUES**](https://code.djangoproject.com/)
* [**ABOUT**](https://www.djangoproject.com/foundation/)
* [**♥ DONATE**](https://www.djangoproject.com/fundraising/)

# [**Documentation**](https://docs.djangoproject.com/fr/3.1/)

Haut du formulaire

Search:Rechercher

Bas du formulaire

* [**Getting Help**](https://docs.djangoproject.com/fr/3.1/faq/help/)
* Langue : **fr**
* Version de la documentation : **3.1**

# **Balises et filtres de gabarit personnalisés**[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#custom-template-tags-and-filters)

Le langage de gabarits de Django offre une large palette de [balises et filtres intégrés](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/) conçus pour répondre aux besoins de la logique de présentation de votre application. Néanmoins, il se peut que vous rencontriez des besoins non couverts par cet ensemble de fonctions de gabarits. Il est possible d’étendre le moteur de gabarit en définissant des balises et des filtres personnalisés en Python, puis en les rendant disponibles dans vos gabarits par la balise [**{% load %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-load).

## **Disposition du code**[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#code-layout)

L’endroit le plus souvent utilisé pour placer des balises et filtres de gabarit personnalisés est à l’intérieur d’une application Django. S’ils sont liés à une application existante, il est logique de les y intégrer ; sinon, ils peuvent être ajoutés à une nouvelle application. Lorsqu’une application Django est ajoutée à [**INSTALLED\_APPS**](https://docs.djangoproject.com/fr/3.1/ref/settings/#std:setting-INSTALLED_APPS), toute balise qu’elle définit à l’endroit conventionnel présenté ci-dessous est automatiquement mise à disposition du chargement depuis les gabarits.

L’application doit contenir un répertoire **templatetags** au même niveau que **models.py**, **views.py**, etc. S’il n’existe pas encore, créez-le, sans oublier le fichier **\_\_init\_\_.py** pour que le répertoire soit bien considéré comme un paquet Python.

**Le serveur de développement ne va pas redémarrer automatiquement**

Après avoir ajouté le module **templatetags**, vous devez redémarrer le serveur avant de pouvoir utiliser les balises ou les filtres dans les gabarits.

Vos balises et filtres personnalisés se trouveront dans un module à l’intérieur du répertoire **templatetags**. Le nom du fichier de module est le nom que vous utiliserez plus tard pour charger les balises, prenez donc soin de choisir un nom qui n’est pas déjà utilisé pour des balises et filtres d’une autre application.

Par exemple, si vos balises/filtres personnalisés se trouvent dans un fichier nommé **poll\_extras.py**, la disposition des fichiers de votre application pourrait ressembler à ceci :

polls/

\_\_init\_\_.py

models.py

templatetags/

\_\_init\_\_.py

poll\_extras.py

views.py

Et dans votre gabarit, voici ce qu’il faudrait écrire :

{% **load** poll\_extras %}

L’application qui contient les balises personnalisées doit apparaître dans [**INSTALLED\_APPS**](https://docs.djangoproject.com/fr/3.1/ref/settings/#std:setting-INSTALLED_APPS) pour que la balise [**{% load %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-load) fonctionne. C’est un élément de sécurité : cela permet d’accueillir du code Python de beaucoup de bibliothèques de gabarits sur une même machine sans devoir permettre l’accès à toutes ces bibliothèques pour chaque installation Django.

Il n’y a aucune limite sur le nombre de modules que l’on peut placer dans le paquet **templatetags**. Gardez simplement à l’esprit qu’une commande [**{% load %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-load) va charger les balises/filtres du nom de module Python indiqué, et non pas du nom de l’application.

Pour être valide, le module de la bibliothèque de balises doit contenir une variable au niveau module nommée **register** et qui doit être une instance de **template.Library**, dans laquelle toutes les balises et les filtres sont inscrits. Ainsi, dans la partie supérieure de votre module, écrivez ceci :

**from** **django** **import** template

register = template.Library()

Il est aussi possible d’inscrire les modules de balises de gabarit au moyen du paramètre **'libraries'** de **[DjangoTemplates](https://docs.djangoproject.com/fr/3.1/topics/templates/" \l "django.template.backends.django.DjangoTemplates" \o "django.template.backends.django.DjangoTemplates)**. Cette méthode est utile si vous souhaitez utiliser un nom autre que celui du module de balises de gabarit au moment de charger ces balises. Cela permet aussi d’inscrire des balises sans devoir installer une application.

**En coulisses**

Pour de multiples exemples, lisez le code source des filtres et balises par défaut de Django. Vous les trouverez respectivement dans **django/template/defaultfilters.py** et **django/template/defaulttags.py**.

Pour plus d’informations sur la balise **[load](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/" \l "std:templatetag-load)**, lisez sa documentation.

## **Écriture de filtres de gabarits personnalisés**[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-custom-template-filters)

Les filtres personnalisés sont des fonctions Python qui acceptent un ou deux paramètres :

* La valeur de la variable (en entrée), pas forcément une chaîne.
* La valeur du paramètre ; il peut avoir une valeur par défaut ou être simplement absent.

Par exemple, dans le filtre **{{ var|foo:"bar" }}**, le filtre **foo** reçoit la variable **var** et le paramètre **"bar"**.

Dans la mesure où le langage de gabarit ne s’occupe pas de la gestion des exceptions, toute exception générée dans un filtre de gabarit apparaît comme erreur de serveur. C’est pourquoi les fonctions de filtres devraient éviter de générer des exceptions quand il existe une valeur de repli raisonnable à renvoyer. Dans le cas d’une valeur d’entrée représentant clairement un bogue dans un gabarit, il se peut que la génération d’une exception soit toujours la meilleure solution plutôt que d’échouer silencieusement en masquant l’erreur.

Voici un exemple de définition de filtre :

**def** cut(value, arg):

*"""Removes all values of arg from the given string"""*

**return** value.replace(arg, '')

Et voici une exemple de la manière dont ce filtre pourrait être utilisé :

{{ somevariable|cut:"0" }}

La plupart des filtres n’acceptent pas de paramètre. Dans ce cas, ne rajoutez pas de paramètre à votre fonction

**def** lower(value): *# Only one argument.*

*"""Converts a string into all lowercase"""*

**return** value.lower()

### Inscription de filtres personnalisés[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#registering-custom-filters)

**django.template.Library.filter()**[**¶**](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#django.template.Library.filter)

Après avoir écrit la définition d’un filtre, il est nécessaire de l’inscrire avec votre instance de **Library** pour qu’il soit disponible dans le langage de gabarit de Django :

register.filter('cut', cut)

register.filter('lower', lower)

La méthode **Library.filter()** accepte deux paramètres :

1. Le nom du filtre, une chaîne.
2. La fonction de compilation, une fonction Python (et non pas le nom de la fonction sous forme de chaîne).

Vous pouvez aussi utiliser **register.filter()** sous forme de décorateur :

@register.filter(name='cut')

**def** cut(value, arg):

**return** value.replace(arg, '')

@register.filter

**def** lower(value):

**return** value.lower()

Si vous omettez le paramètre **name** comme dans le second exemple ci-dessus, Django utilisera le nom de la fonction comme nom de filtre.

Pour terminer, **register.filter()** accepte aussi trois paramètres nommés : **is\_safe**, **needs\_autoescape** et **expects\_localtime**. Ces paramètres sont décrits dans [filtres et échappement automatique](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#filters-auto-escaping) et [filtres et fuseaux horaires](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#filters-timezones) ci-dessous.

### Filtres de gabarits agissant sur des chaînes[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#template-filters-that-expect-strings)

**django.template.defaultfilters.stringfilter()**[**¶**](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#django.template.defaultfilters.stringfilter)

Si vous écrivez un filtre de gabarit qui s’attend uniquement à une chaîne comme premier paramètre, vous devriez utiliser le décorateur **stringfilter**. Celui-ci se chargera de convertir un objet dans son équivalent chaîne de caractères avant de le passer à votre fonction :

**from** **django** **import** template

**from** **django.template.defaultfilters** **import** stringfilter

register = template.Library()

@register.filter

@stringfilter

**def** lower(value):

**return** value.lower()

De cette façon, vous pourrez par exemple passer un nombre entier à ce filtre et il ne générera pas d’erreur **AttributeError** (les nombres entiers n’ont pas de méthode **lower()**).

### Les filtres et l’échappement automatique[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#filters-and-auto-escaping)

Quand vous écrivez un filtre personnalisé, réfléchissez sur la façon dont celui-ci va interagir avec le comportement d’échappement automatique de Django. Notez que deux types de chaînes peuvent circuler dans le code des gabarits :

* Des **chaînes brutes** sont les chaînes Python natives. En sortie, et si l’échappement automatique est en vigueur, ces chaînes seront échappées, sinon elles sont laissées telles quelles.
* Des **chaînes sûres** sont des chaînes qui ont été marquées comme sûres et par là-même qu’elles n’ont plus besoin d’être échappées au moment de l’affichage. Tout échappement nécessaire aura déjà été effectué. Elles sont habituellement utilisées pour des chaînes qui contiennent du HTML brut devant être interprété tel quel au niveau du client.

En interne, ces chaînes sont de type **[SafeString](https://docs.djangoproject.com/fr/3.1/ref/utils/" \l "django.utils.safestring.SafeString" \o "django.utils.safestring.SafeString)**. Vous pouvez les identifier en utilisant du code comme :

**from** **django.utils.safestring** **import** SafeString

**if** isinstance(value, SafeString):

*# Do something with the "safe" string.*

...

Le code des filtres de gabarits entre dans l’une des deux situations suivantes :

1. Votre filtre n’introduit aucun nouveau caractère HTML non sûr (**<**, **>**, **'**, **"** ou **&**) dans le résultat. Dans ce cas, vous pouvez laisser Django s’occuper de toute la gestion de l’échappement automatique. Tout ce que vous avez à faire est de définir le drapeau **is\_safe** à **True** lorsque vous inscrivez votre fonction de filtre, comme ceci :
2. @register.filter(is\_safe=**True**)
3. **def** myfilter(value):
4. **return** value

Ce drapeau indique à Django que si une chaîne « sûre » (safe) est passée à votre filtre, le résultat sera toujours « sûr » et que si une chaîne non sûre est passée, Django l’échappera automatiquement si nécessaire.

Vous pouvez considérer cela comme signifiant que « ce filtre est sûr, il n’introduit aucune éventualité de code HTML non sûr ».

La raison d’être de **is\_safe** est qu’il existe un grand nombre d’opérations normales sur les chaînes qui transforment un objet **SafeData** en un objet **str** normal, et qu’au lieu d’essayer d’identifier tous ces cas de figure, ce qui serait très difficile, Django répare les dommages après que le filtre ait fait son travail.

Par exemple, imaginons un filtre qui ajoute la chaîne **xx** à la fin de n’importe quelle variable d’entrée. Comme cela n’introduit aucun caractère HTML dangereux dans le résultat (sauf s’il y en avait déjà), vous devriez marquer ce filtre avec **is\_safe**:

@register.filter(is\_safe=**True**)

**def** add\_xx(value):

**return** '**%s**xx' % value

Lorsque ce filtre est utilisé dans un gabarit dans lequel l’échappement automatique est en vigueur, Django va échapper le contenu chaque fois que la variable dde départ n’est pas déjà marquée comme « sûre ».

Par défaut, **is\_safe** vaut **False**, et vous pouvez simplement l’omettre quand il n’est pas nécessaire.

Soyez prudent quand vous décidez si votre filtre conserve réellement des chaînes sûres. Si vous enlevez des caractères, il se peut que vous laissiez par mégarde des balises ou entités HTML mal équilibrées dans le résultat. Par exemple, en enlevant un **>** du contenu initial, cela pourrait transformer **<a>** en **<a** qui devrait alors être échappé à l’affichage pour éviter de poser des problèmes. De même, enlever un point-virgule (**;**) peut transformer **&amp;** en **&amp**, ce qui n’est plus une entité valide et devrait donc de nouveau être échappé. Ce ne sera pas si subtil dans la plupart des cas, mais gardez à l’esprit ce genre de problème potentiel lorsque vous relisez votre code.

Le marquage d’un filtre avec **is\_safe** force la valeur de retour du filtre en chaîne de caractères. Si votre filtre doit renvoyer un booléen ou une autre valeur non textuelle, le marquage avec **is\_safe** va probablement générer des conséquences inattendues (comme la conversion du booléen False en chaîne “False”).

1. Une autre option possible est de s’occuper manuellement dans le code du filtre d’effectuer d’éventuels échappements. C’est nécessaire quand vous introduisez de nouvelles balises HTML dans le résultat. Il faut alors marquer le contenu comme sûr et exempt de besoin d’échappement afin que le code HTML introduit ne soit pas échappé plus tard, ce qui implique donc que vous gériez vous-même ce contenu.

Pour marquer le résultat comme une chaîne sûre, utilisez **[django.utils.safestring.mark\_safe()](https://docs.djangoproject.com/fr/3.1/ref/utils/" \l "django.utils.safestring.mark_safe" \o "django.utils.safestring.mark_safe)**.

Mais restez prudent. Vous devez faire plus que simplement marquer le résultat comme sûr. Vous devez vous assurer qu’il soit réellement sûr et votre action dépend de l’activité de l’échappement automatique. L’idée est d’écrire des filtres qui peuvent agir dans des gabarits où l’échappement automatique est activé ou non, afin de faciliter la tâche aux rédacteurs de gabarits.

Pour que vos filtres sachent si l’échappement automatique est actuellement actif, définissez le drapeau **needs\_autoescape** à **True** lorsque vous inscrivez votre fonction de filtre (il vaut **False** par défaut). Ce drapeau indique à Django que votre fonction de filtre souhaite recevoir un paramètre mot-clé supplémentaire, **autoescape**. Ce paramètre vaudra **True** quand l’échappement automatique est en vigueur et **False** dans le cas contraire. Il est recommandé de définir la valeur par défaut du paramètre **autoescape** à **True**, pour que si la fonction est appelée depuis le code Python, l’échappement soit activé par défaut.

Par exemple, écrivons un filtre qui met en évidence le premier caractère d’une chaîne :

**from** **django** **import** template

**from** **django.utils.html** **import** conditional\_escape

**from** **django.utils.safestring** **import** mark\_safe

register = template.Library()

@register.filter(needs\_autoescape=**True**)

**def** initial\_letter\_filter(text, autoescape=**True**):

first, other = text[0], text[1:]

**if** autoescape:

esc = conditional\_escape

**else**:

esc = **lambda** x: x

result = '<strong>**%s**</strong>**%s**' % (esc(first), esc(other))

**return** mark\_safe(result)

Le drapeau **needs\_autoescape** et le paramètre nommé **autoescape** signifient que notre fonction saura si l’échappement automatique est en vigueur au moment où le filtre est appelé. Nous utilisons **autoescape** pour savoir si nous devons faire passer les données reçues par la fonction **django.utils.html.conditional\_escape** (sinon, nous utilisons la fonction neutre comme fonction d’échappement). La fonction **conditional\_escape()** est semblable à **escape()**, sauf que l’échappement n’a lieu que si la valeur d’entrée n’est **pas** déjà une instance de **SafeData**. Si une instance de **SafeData** est transmise à **conditional\_escape()**, les données sont renvoyées sans être modifiées.

Finalement, dans l’exemple ci-dessus, nous n’oublions pas de marquer le résultat comme sûr afin que notre code HTML soit inséré directement dans le gabarit sans échappement supplémentaire.

Il n’y a pas besoin de se préoccuper du drapeau **is\_safe** dans ce cas (même si on pourrait le définir sans dommages). Chaque fois que vous gérez manuellement la question de l’échappement automatique et que vous renvoyez une chaîne sûre, le drapeau **is\_safe** ne change rien, quelle que soit sa valeur.

**Avertissement**

Risque de vulnérabilités XSS lors de la réutilisation de filtres intégrés

Les filtres intégrés de Django contiennent **autoescape=True** par défaut afin d’obtenir le bon comportement d’échappement automatique et d’éviter ainsi une éventuelle vulnérabilité de script inter-site.

Dans les versions plus anciennes de Django, réutilisez prudemment les filtres intégrés de Django car la valeur par défaut de **autoescape** est **None**. Vous devrez passer explicitement **autoescape=True** pour que le résultat soit automatiquement échappé.

Par exemple, si vous voulez écrire un filtre personnalisé nommé **urlize\_and\_linebreaks** qui combine les filtres **[urlize](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/" \l "std:templatefilter-urlize)** et **[linebreaksbr](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/" \l "std:templatefilter-linebreaksbr)**, le filtre pourrait ressembler à ceci :

**from** **django.template.defaultfilters** **import** linebreaksbr, urlize

@register.filter(needs\_autoescape=**True**)

**def** urlize\_and\_linebreaks(text, autoescape=**True**):

**return** linebreaksbr(

urlize(text, autoescape=autoescape),

autoescape=autoescape

)

Puis :

{{ comment|urlize\_and\_linebreaks }}

serait équivalent à :

{{ comment|urlize|linebreaksbr }}

### Filtres et fuseaux horaires[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#filters-and-time-zones)

Si vous écrivez un filtre personnalisé qui opère sur des objets **[datetime](https://docs.python.org/3/library/datetime.html" \l "datetime.datetime" \o "(disponible dans Python v3.9))**, vous allez généralement l’inscrire avec le drapeau **expects\_localtime** défini à **True**:

@register.filter(expects\_localtime=**True**)

**def** businesshours(value):

**try**:

**return** 9 <= value.hour < 17

**except** **AttributeError**:

**return** ''

Lorsque ce drapeau est défini et si le premier paramètre du filtre est un objet date/heure sensible aux fuseaux horaires, Django le convertit au besoin dans le fuseau horaire actuel avant de le passer à votre filtre, selon les [règles de conversion de fuseaux horaires dans les gabarits](https://docs.djangoproject.com/fr/3.1/topics/i18n/timezones/#time-zones-in-templates).

## **Écriture de balises de gabarits personnalisées**[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-custom-template-tags)

Les balises sont plus complexes que les filtres, car les balises peuvent tout faire. Django fournit un certain nombre de raccourcis qui facilitent l’écriture de la plupart des types de balises. Nous allons commencer par explorer ces raccourcis, puis expliquer comment écrire une balise à partir de rien pour les cas où les raccourcis ne conviennent pas au besoin.

### Balises simples[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#simple-tags)

**django.template.Library.simple\_tag()**[**¶**](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#django.template.Library.simple_tag)

Bien des balises de gabarit acceptent des paramètres (chaînes ou variables de gabarit) et renvoient un résultat après avoir effectué quelques opérations sur la base des paramètres initiaux et de certaines informations externes. Par exemple, une balise **current\_time** pourrait accepter une chaîne de formatage et renvoyer l’heure sous forme de chaîne dans le bon format.

Pour faciliter la création de ce type de balise, Django fournit une fonction utilitaire, **simple\_tag**. Cette fonction, qui est une méthode de **django.template.Library**, accepte elle-même une fonction qui accepte autant de paramètres que nécessaire, l’enveloppe dans une fonction **render** ainsi que les autres éléments nécessaires tels que mentionnés ci-dessus et l’inscrit dans le système de gabarits.

Notre fonction **current\_time** pourrait donc être écrite comme ceci :

**import** **datetime**

**from** **django** **import** template

register = template.Library()

@register.simple\_tag

**def** current\_time(format\_string):

**return** datetime.datetime.now().strftime(format\_string)

Quelques petites choses à signaler au sujet de la fonction utilitaire **simple\_tag**:

* Le contrôle du nombre de paramètres requis, etc., a déjà été effectué au moment où notre fonction est appelée, nous n’avons donc pas à le faire.
* Les guillemets autour du paramètre (le cas échéant) ont déjà été enlevés, nous recevons donc une chaîne normale.
* Si le paramètre était une variable de gabarit, notre fonction reçoit la valeur réelle de la variable et non pas la variable elle-même.

Au contraire des autres utilitaires de balises, **simple\_tag** passe son résultat par **[conditional\_escape()](https://docs.djangoproject.com/fr/3.1/ref/utils/" \l "django.utils.html.conditional_escape" \o "django.utils.html.conditional_escape)** si le contexte de gabarit est en mode échappement automatique, pour garantir du HTML correct et vous protéger d’éventuelles vulnérabilités XSS.

Si l’échappement complémentaire n’est pas souhaité, il est nécessaire d’utiliser **[mark\_safe()](https://docs.djangoproject.com/fr/3.1/ref/utils/" \l "django.utils.safestring.mark_safe" \o "django.utils.safestring.mark_safe)** pour autant que vous soyez absolument sûr que votre code ne contienne pas de vulnérabilité XSS. Pour la construction de petits bouts de HTML, il est fortement recommandé d’utiliser **[format\_html()](https://docs.djangoproject.com/fr/3.1/ref/utils/" \l "django.utils.html.format_html" \o "django.utils.html.format_html)** au lieu de **mark\_safe()**.

Si votre balise de gabarit a besoin d’accéder au contexte en cours, vous pouvez utiliser le paramètre **takes\_context** lors de l’inscription de la balise :

@register.simple\_tag(takes\_context=**True**)

**def** current\_time(context, format\_string):

timezone = context['timezone']

**return** your\_get\_current\_time\_method(timezone, format\_string)

Notez que le premier paramètre doit être nommé **context**.

Pour plus d’informations sur le fonctionnement de l’option **takes\_context**, consultez la section sur les [balises d’inclusion](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#howto-custom-template-tags-inclusion-tags).

Si vous avez besoin de renommer votre balise, vous pouvez lui donner un nom personnalisé :

register.simple\_tag(**lambda** x: x - 1, name='minusone')

@register.simple\_tag(name='minustwo')

**def** some\_function(value):

**return** value - 2

Les fonctions **simple\_tag** peuvent accepter n’importe quel nombre de paramètres positionnels ou nommés. Par exemple :

@register.simple\_tag

**def** my\_tag(a, b, \*args, \*\*kwargs):

warning = kwargs['warning']

profile = kwargs['profile']

...

**return** ...

Puis, dans le gabarit, n’importe quel nombre de paramètres séparés par des espaces peuvent être transmis à la balise de gabarit. Comme en Python, les valeurs des paramètres nommés sont définis par le signe égal (**=**) et doivent être placés après les paramètres positionnels. Par exemple :

{% **my\_tag** 123 "abcd" book.title warning=message|lower profile=user.profile %}

Il est possible de stocker le résultat de la balise dans une variable de gabarit au lieu de l’afficher directement. Cela se fait en utilisant le paramètre **as** suivi du nom de variable. Ce faisant, vous pouvez ensuite afficher ce contenu à l’endroit où vous le souhaitez :

{% **current\_time** "%Y-%m-%d %I:%M %p" **as** the\_time %}

<**p**>The time is {{ the\_time }}.</**p**>

### Balises d’inclusion[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#inclusion-tags)

**django.template.Library.inclusion\_tag()**[**¶**](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#django.template.Library.inclusion_tag)

Un autre type de balises de gabarit fréquent est le type qui affiche certaines données en effectuant le rendu d’un autre gabarit. Par exemple, l’interface d’administration de Django utilise des balises de gabarit personnalisées pour afficher les boutons au bas des pages de formulaires d’ajout/édition. Ces boutons ont toujours une même apparence, mais les cibles de leurs liens changent en fonction de l’objet en cours d’édition, ils représentent donc un cas typique d’utilisation d’un petit gabarit complété par certains détails sur l’objet en cours (dans le cas précis de l’interface d’administration, il s’agit de la balise **submit\_row**).

Ces types de balises sont appelées des « balises d’inclusion ».

Un exemple sera le meilleur moyen d’illustrer l’écriture de balises d’inclusion. Écrivons une balise qui affiche une liste de choix pour un objet **Poll** donné, comme celui qui a été créé dans les [tutoriels](https://docs.djangoproject.com/fr/3.1/intro/tutorial02/#creating-models). Nous utiliserons la balise comme ceci :

{% **show\_results** poll %}

… et cela donnera le résultat suivant :

<**ul**>

<**li**>First choice</**li**>

<**li**>Second choice</**li**>

<**li**>Third choice</**li**>

</**ul**>

Pour commencer, définissez la fonction qui accepte le paramètre et qui produit un dictionnaire de données comme résultat. Le point important ici est que nous devons uniquement renvoyer un dictionnaire, rien d’autre de plus complexe. Il sera utilisé comme contexte de gabarit pour le fragment de gabarit. Exemple :

**def** show\_results(poll):

choices = poll.choice\_set.all()

**return** {'choices': choices}

Ensuite, créez le gabarit utilisé pour faire le rendu du résultat de la balise. Ce gabarit est une fonctionnalité figée de la balise : c’est le rédacteur de la balise qui le définit, pas le concepteur des gabarits. Suivant notre exemple, le gabarit est très court :

<**ul**>

{% **for** choice **in** choices %}

<**li**> {{ choice }} </**li**>

{% **endfor** %}

</**ul**>

Maintenant, créez et inscrivez la balise d’inclusion en appelant la méthode **inclusion\_tag()** d’un objet **Library**. Suivant notre exemple, si le gabarit ci-dessus se trouve dans un fichier nommé **results.html** dans un répertoire parcouru par le chargeur de gabarits, voici comment nous inscrivons la balise :

*# Here, register is a django.template.Library instance, as before*

@register.inclusion\_tag('results.html')

**def** show\_results(poll):

...

Il est également possible d’inscrire la balise d’inclusion en utilisant une instance de **[django.template.Template](https://docs.djangoproject.com/fr/3.1/ref/templates/api/" \l "django.template.Template" \o "django.template.Template)**:

**from** **django.template.loader** **import** get\_template

t = get\_template('results.html')

register.inclusion\_tag(t)(show\_results)

… lorsque nous avons initialement créé la fonction.

Il peut arriver que vos balises d’inclusion nécessitent un grand nombre de paramètres, ce qui rend pénible pour les auteurs des gabarits la transmission de tous les paramètres en se souvenant de l’ordre exigé. Pour résoudre ce problème, Django fournit l’option **takes\_context** pour les balises d’inclusion. Si vous définissez **takes\_context** lors de la création d’une balise de gabarit, celle-ci n’aura aucun paramètre obligatoire et la fonction Python sous-jacente recevra un paramètre, le contexte du gabarit au moment où la balise a été appelée.

Par exemple, disons que vous écrivez une balise d’inclusion qui sera toujours utilisée dans un contexte contenant les variables **home\_link** et **home\_title** pointant sur la page principale. Voici à quoi la fonction Python ressemblerait :

@register.inclusion\_tag('link.html', takes\_context=**True**)

**def** jump\_link(context):

**return** {

'link': context['home\_link'],

'title': context['home\_title'],

}

Notez que le premier paramètre de la fonction doit être nommé **context**.

Dans la ligne **register.inclusion\_tag()**, nous précisons **takes\_context=True** ainsi que le nom du gabarit. Voici à quoi pourrait ressembler le gabarit **link.html**:

Jump directly to <**a** href="{{ link }}">{{ title }}</**a**>.

Puis, chaque fois que vous souhaitez utiliser cette balise personnalisée, chargez sa bibliothèque et appelez-la sans paramètre, comme ceci :

{% **jump\_link** %}

Notez que lorsque vous utilisez **takes\_context=True**, il n’y a pas besoin de transmettre de paramètres à la balise de gabarit. Cette dernière a automatiquement accès au contexte.

Le paramètre **takes\_context** vaut **False** par défaut. Lorsqu’il est défini à **True**, la balise reçoit l’objet contexte, comme dans cet exemple. C’est la seule différence entre ce cas et l’exemple **inclusion\_tag** précédent.

Les fonctions **inclusion\_tag** acceptent n’importe quel nombre de paramètres positionnels ou nommés. Par exemple :

@register.inclusion\_tag('my\_template.html')

**def** my\_tag(a, b, \*args, \*\*kwargs):

warning = kwargs['warning']

profile = kwargs['profile']

...

**return** ...

Puis, dans le gabarit, n’importe quel nombre de paramètres séparés par des espaces peuvent être transmis à la balise de gabarit. Comme en Python, les valeurs des paramètres nommés sont définis par le signe égal (**=**) et doivent être placés après les paramètres positionnels. Par exemple :

{% **my\_tag** 123 "abcd" book.title warning=message|lower profile=user.profile %}

### Balises de gabarits personnalisées (niveau avancé)[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#advanced-custom-template-tags)

Il peut arriver que les fonctionnalités de base de la création d’une balise de gabarit personnalisée ne suffisent pas. Ne vous en faites pas. Django vous offre un accès complet à l’interface interne nécessaire pour construire une balise de gabarit à partir de rien.

### Aperçu rapide[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#a-quick-overview)

Le système de gabarits fonctionne par un processus en deux étapes : la compilation et le rendu. Pour créer une balise de gabarit personnalisée, il s’agit de définir à la fois le comportement de la compilation et celui du rendu.

Quand Django compile un gabarit, il divise le texte brut du gabarit en « nœuds ». Chaque nœud est une instance de **django.template.Node** et possède une méthode **render()**. Un gabarit compilé est une liste d’objets **Node**. Lorsque vous appelez **render()** sur un objet gabarit compilé, le gabarit appelle **render()** pour chaque **Node** de sa liste de nœuds en leur passant le contexte. Les résultats sont tous concaténés pour former le rendu du gabarit.

Ainsi, pour créer une balise de gabarit personnalisée, vous définissez la manière de convertir la balise de gabarit brute en un **Node** (fonction de compilation) et ce que fait la méthode **render()** du nœud.

### Écriture de la fonction de compilation[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-the-compilation-function)

Pour chaque balise de gabarit que l’analyseur de gabarit rencontre, il appelle une fonction Python avec le contenu de la balise et l’objet analyseur lui-même. Cette fonction est responsable de renvoyer une instance de **Node** en fonction du contenu de la balise.

Par exemple, écrivons une implémentation complète de notre balise de gabarit, **{% current\_time %}**, qui affiche l’heure et la date courantes formatées en fonction d’un paramètre donné à la balise, en respectant la syntaxe **[strftime()](https://docs.python.org/3/library/time.html" \l "time.strftime" \o "(disponible dans Python v3.9))**. Il est conseillé de définir la syntaxe de la balise avant toute autre chose. Dans notre cas, disons que la balise devrait être utilisée comme ceci :

<**p**>The time is {% **current\_time** "%Y-%m-%d %I:%M %p" %}.</**p**>

L’analyseur de cette fonction doit capturer le paramètre et créer un objet **Node**:

**from** **django** **import** template

**def** do\_current\_time(parser, token):

**try**:

*# split\_contents() knows not to split quoted strings.*

tag\_name, format\_string = token.split\_contents()

**except** **ValueError**:

**raise** template.TemplateSyntaxError(

"**%r** tag requires a single argument" % token.contents.split()[0]

)

**if** **not** (format\_string[0] == format\_string[-1] **and** format\_string[0] **in** ('"', "'")):

**raise** template.TemplateSyntaxError(

"**%r** tag's argument should be in quotes" % tag\_name

)

**return** CurrentTimeNode(format\_string[1:-1])

Notes :

* **parser** est l’objet analyseur de gabarit. Nous n’en avons pas besoin dans cet exemple.
* **token.contents** est une chaîne formée du contenu brut de la balise. Dans notre exemple, il s’agit de **'current\_time "%Y-%m-%d %I:%M %p"'**.
* La méthode **token.split\_contents()** sépare les paramètres en fonction des espaces tout en gardant groupées les chaînes entre guillemets. La méthode plus directe **token.contents.split()** ne serait pas aussi robuste, car elle couperait naïvement à chaque espace, y compris ceux à l’intérieur des chaînes entre guillemets. Il est conseillé de toujours utiliser **token.split\_contents()**.
* Cette fonction est responsable de lever l’exception **django.template.TemplateSyntaxError** à chaque erreur de syntaxe, avec des messages utiles.
* Les exceptions **TemplateSyntaxError** utilisent la variable **tag\_name**. Ne codez pas en dur le nom de la balise dans les messages d’erreur, car cela lie le nom de la balise à votre fonction. **token.contents.split()[0]** sera toujours le nom de la balise, même lorsque celle-ci n’accepte aucun paramètre.
* La fonction renvoie un **CurrentTimeNode** avec tout ce que le nœud a besoin de connaître au sujet de la balise. Dans ce cas, il passe le paramètre (**"%Y-%m-%d %I:%M %p"**). Les guillemets ouvrant et fermant de la balise de gabarit sont enlevés par **format\_string[1:-1]**.
* L’analyse est de très bas niveau. Les développeurs Django ont expérimenté l’écriture de petites infrastructures au-dessus de ce système d’analyse, en utilisant des techniques comme les grammaires EBNF, mais ces expériences ont rendu le moteur de gabarit trop lent. L’analyse est de bas niveau car c’est la solution la plus rapide.

### Écriture de la fonction de rendu[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-the-renderer)

La seconde étape dans l’écriture de balises de gabarit est définir une sous-classe de **Node** possédant une méthode **render()**.

Pour continuer l’exemple ci-dessus, nous devons définir **CurrentTimeNode**:

**import** **datetime**

**from** **django** **import** template

**class** **CurrentTimeNode**(template.Node):

**def** \_\_init\_\_(self, format\_string):

self.format\_string = format\_string

**def** render(self, context):

**return** datetime.datetime.now().strftime(self.format\_string)

Notes :

* **\_\_init\_\_()** reçoit la chaîne **format\_string** de **do\_current\_time()**. Passez toujours toute option ou paramètre à un **Node** par l’intermédiaire de sa méthode **\_\_init\_\_()**.
* C’est dans la méthode **render()** que se passe réellement le travail.
* **render()** doit généralement échouer silencieusement, particulièrement en environnement de production. Cependant, dans certains cas, spécialement lorsque **context.template.engine.debug** vaut **True**, cette méthode peut générer une exception pour faciliter le débogage. Par exemple, plusieurs balises intégrées génèrent **django.template.TemplateSyntaxError** si elles reçoivent le mauvais nombre de paramètres ou des paramètres du mauvais type.

Au final, cette distinction entre compilation et rendu aboutit à un système de gabarit efficace, car un gabarit peut produire un rendu avec plusieurs contextes différents sans devoir être analysé plusieurs fois.

### Considérations sur l’échappement automatique[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#auto-escaping-considerations)

Le résultat des balises de gabarit ne passe **pas** automatiquement par les filtres d’échappement automatique (à l’exception de **[simple\_tag()](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/" \l "django.template.Library.simple_tag" \o "django.template.Library.simple_tag)** comme expliqué plus haut) . Toutefois, il y a quand même deux ou trois choses à garder à l’esprit en écrivant des balises de gabarit.

Si la méthode **render()** de votre balise de gabarit conserve le résultat dans une variable de contexte (plutôt que de renvoyer le résultat dans une chaîne), elle devrait alors s’occuper d’appeler **mark\_safe()** si nécessaire. Lorsque la variable sera finalement affichée, elle sera affectée par le réglage d’échappement automatique en vigueur à ce moment, il faut donc que les contenus qui ne doivent plus être échappés soient marqués comme tels.

De même, si votre balise de gabarit crée un nouveau contexte pour procéder à certains sous-rendus, définissez l’attribut d’échappement automatique pour la valeur du contexte en cours. La méthode **\_\_init\_\_** de la classe **Context** accepte un paramètre nommé **autoescape** et qui est destiné à cet effet. Par exemple :

**from** **django.template** **import** Context

**def** render(self, context):

*# ...*

new\_context = Context({'var': obj}, autoescape=context.autoescape)

*# ... Do something with new\_context ...*

Ce n’est pas une situation très courante, mais c’est utile si vous faites vous-même le rendu d’un gabarit. Par exemple :

**def** render(self, context):

t = context.template.engine.get\_template('small\_fragment.html')

**return** t.render(Context({'var': obj}, autoescape=context.autoescape))

Si nous avions négligé de passer la valeur actuelle de **context.autoescape** au nouveau **Context** de cet exemple, les résultats auraient toujours subi un échappement automatique, ce qui pourrait ne pas correspondre au comportement attendu si la balise de gabarit est utilisée à l’intérieur d’un bloc [**{% autoescape off %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-autoescape).

### Considérations sur la concurrence entre « threads »[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#thread-safety-considerations)

Dès qu’un nœud est analysé, sa méthode **render** peut être appelée indéfiniment. Comme Django est parfois lancé dans un environnement avec plusieurs fils d’exécution parallèles (multi-thread), un même nœud peut être rendu plusieurs fois simultanément avec différents contextes en réponse à des requêtes séparées. Il est donc important de s’assurer que vos balises de gabarit sont robustes à la concurrence.

Pour que vos balises de gabarit sachent gérer la concurrence, il ne faut jamais stocker des informations d’état dans le nœud lui-même. Par exemple, Django contient une balise de gabarit [**cycle**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-cycle) qui parcourt une liste de chaînes données à chaque rendu :

{% **for** o **in** some\_list %}

<**tr** class="{% **cycle** 'row1' 'row2' %}">

...

</**tr**>

{% **endfor** %}

Une implémentation naïve de **CycleNode** pourrait ressembler à ceci :

**import** **itertools**

**from** **django** **import** template

**class** **CycleNode**(template.Node):

**def** \_\_init\_\_(self, cyclevars):

self.cycle\_iter = itertools.cycle(cyclevars)

**def** render(self, context):

**return** next(self.cycle\_iter)

Mais supposons que deux gabarits soient rendus en parallèle et qu’ils contiennent l’extrait de gabarit ci-dessus :

1. Le fil d’exécution 1 effectue sa première itération de boucle, **CycleNode.render()** renvoie « row1 »
2. Le fil d’exécution 2 effectue sa première itération de boucle, **CycleNode.render()** renvoie « row2 »
3. Le fil d’exécution 1 effectue sa seconde itération de boucle, **CycleNode.render()** renvoie « row1 »
4. Le fil d’exécution 2 effectue sa seconde itération de boucle, **CycleNode.render()** renvoie « row2 »

Le nœud CycleNode effectue son itération, mais globalement. En ce qui concerne les fils d’exécution 1 et 2, ils renvoient toujours la même valeur. Ce n’est évidemment pas ce que nous voulons !

Pour résoudre ce problème, Django met à disposition un **render\_context** qui est associé au **context** du gabarit qui est en cours de rendu. **render\_context** se comporte comme un dictionnaire Python et doit être utilisé pour stocker l’état de **Node** entre les invocations de la méthode **render**.

Révisons notre implémentation de **CycleNode** en utilisant **render\_context**:

**class** **CycleNode**(template.Node):

**def** \_\_init\_\_(self, cyclevars):

self.cyclevars = cyclevars

**def** render(self, context):

**if** self **not** **in** context.render\_context:

context.render\_context[self] = itertools.cycle(self.cyclevars)

cycle\_iter = context.render\_context[self]

**return** next(cycle\_iter)

Notez qu’il est totalement correct de stocker sous forme d’attribut de l’information globale qui ne changera pas durant le cycle de vie de **Node**. Dans le cas de **CycleNode**, le paramètre **cyclevars** ne change plus après que **Node** a été instancié, nous n’avons donc pas besoin de le placer dans **render\_context**. Mais l’information d’état qui est spécifique au gabarit en cours de rendu, comme l’état d’itération de **CycleNode**, doit être stockée dans **render\_context**.

**Note**

Remarquez la manière dont nous avons utilisé **self** pour délimiter l’information spécifique de **CycleNode** dans **render\_context**. Il peut y avoir plusieurs **CycleNodes** dans un gabarit donné, il faut donc être prudent de ne pas empiéter sur l’information d’état d’un autre nœud. La façon la plus simple de faire cela est de toujours utiliser **self** comme clé dans **render\_context**. Si vous conservez la trace de plusieurs variables d’état, faites de **render\_context[self]** un dictionnaire.

### Inscription de la balise[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#registering-the-tag)

Pour terminer, inscrivez la balise dans l’instance **Library** de votre module, comme expliqué dans la section sur l”[écriture des balises de gabarit personnalisées](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#howto-writing-custom-template-tags) ci-dessus. Exemple :

register.tag('current\_time', do\_current\_time)

La méthode **tag()** accepte deux paramètres :

1. Le nom de la balise de gabarit (une chaîne). En cas d’omission, c’est le nom de la fonction de compilation qui sera utilisé.
2. La fonction de compilation, une fonction Python (et non pas le nom de la fonction sous forme de chaîne).

Comme pour l’inscription de filtre, il est aussi possible d’utiliser une syntaxe de décorateur :

@register.tag(name="current\_time")

**def** do\_current\_time(parser, token):

...

@register.tag

**def** shout(parser, token):

...

Si vous omettez le paramètre **name** comme dans le second exemple ci-dessus, Django utilise le nom de la fonction comme nom de balise.

### Transmission de variables de gabarit à la balise[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#passing-template-variables-to-the-tag)

Même si vous pouvez transmettre autant de paramètres que souhaité à la balise de gabarit en utilisant **token.split\_contents()**, les paramètres sont alors tous identifiés comme chaînes de caractères. Pour pouvoir passer du contenu dynamique (une variable de gabarit) comme paramètre d’une balise de gabarit, cela demande un petit peu plus d’effort.

Dans les exemples précédents, l’heure actuelle a été formatée en chaîne et celle-ci a été renvoyée en tant que chaîne. Supposons que nous souhaitions passer un **[DateTimeField](https://docs.djangoproject.com/fr/3.1/ref/models/fields/" \l "django.db.models.DateTimeField" \o "django.db.models.DateTimeField)** d’un objet pour que la balise de gabarit mette en forme cette date :

<**p**>This post was last updated at {% **format\_time** blog\_entry.date\_updated "%Y-%m-%d %I:%M %p" %}.</**p**>

Initialement, **token.split\_contents()** renverra trois valeurs :

1. Le nom de la balise
2. La chaîne **'blog\_entry.date\_updated'** (sans les guillemets).
3. La chaîne de formatage **'"%Y-%m-%d %I:%M %p"'**. La valeur de retour de **split\_contents()** contiendra les guillemets ouvrant et fermant pour de telles chaînes constantes.

Votre balise devrait maintenant commencer à ressembler à ceci :

**from** **django** **import** template

**def** do\_format\_time(parser, token):

**try**:

*# split\_contents() knows not to split quoted strings.*

tag\_name, date\_to\_be\_formatted, format\_string = token.split\_contents()

**except** **ValueError**:

**raise** template.TemplateSyntaxError(

"**%r** tag requires exactly two arguments" % token.contents.split()[0]

)

**if** **not** (format\_string[0] == format\_string[-1] **and** format\_string[0] **in** ('"', "'")):

**raise** template.TemplateSyntaxError(

"**%r** tag's argument should be in quotes" % tag\_name

)

**return** FormatTimeNode(date\_to\_be\_formatted, format\_string[1:-1])

Vous devez aussi modifier la fonction de rendu pour récupérer le contenu réel de la propriété **date\_updated** de l’objet **blog\_entry**. Cela peut se faire en utilisant la classe **Variable()** de **django.template**.

Pour utiliser la classe **Variable**, il faut l’instancier avec le nom de la variable à résoudre puis appeler **variable.resolve(context)**. Ainsi, par exemple :

**class** **FormatTimeNode**(template.Node):

**def** \_\_init\_\_(self, date\_to\_be\_formatted, format\_string):

self.date\_to\_be\_formatted = template.Variable(date\_to\_be\_formatted)

self.format\_string = format\_string

**def** render(self, context):

**try**:

actual\_date = self.date\_to\_be\_formatted.resolve(context)

**return** actual\_date.strftime(self.format\_string)

**except** template.VariableDoesNotExist:

**return** ''

La résolution de variable lèvera une exception **VariableDoesNotExist** si elle ne peut pas résoudre la chaîne qui lui a été passée dans le contexte actuel de la page.

### Définition d’une variable dans le contexte[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#setting-a-variable-in-the-context)

Les exemples ci-dessus ne font qu’afficher une valeur. Il est généralement plus souple de définir des variables de gabarit dans les balises de gabarit que d’afficher des valeurs. De cette façon, les auteurs de gabarits peuvent réutiliser les valeurs créées par vos balises de gabarit.

Pour définir une variable dans le contexte, attribuez des valeurs à l’objet dictionnaire du contexte dans la méthode **render()**. Voici une version mise à jour de **CurrentTimeNode** qui définit une variable de gabarit **current\_time** au lieu de l’afficher :

**import** **datetime**

**from** **django** **import** template

**class** **CurrentTimeNode2**(template.Node):

**def** \_\_init\_\_(self, format\_string):

self.format\_string = format\_string

**def** render(self, context):

context['current\_time'] = datetime.datetime.now().strftime(self.format\_string)

**return** ''

Notez que **render()** renvoie la chaîne vide. **render()** doit toujours renvoyer une chaîne. Si la balise de gabarit ne fait que définir une variable, **render()** doit renvoyer la chaîne vide.

Voici comment on pourrait utiliser cette nouvelle version de la balise :

{% **current\_time** "%Y-%m-%d %I:%M %p" %}<**p**>The time is {{ current\_time }}.</**p**>

**Portées des variables dans le contexte**

Toute variable définie dans le contexte ne sera disponible que dans le même « bloc » de gabarit dans lequel elle a été définie. Ce comportement est voulu ; il définit la portée des variables afin qu’elles n’entrent pas en conflit avec le contexte d’autres blocs.

Mais il reste un problème avec **CurrentTimeNode2**: le nom de variable **current\_time** est codé en dur. Cela veut dire que vous devrez vous assurer que votre gabarit n’utilise pas **{{ current\_time }}** ailleurs, parce que **{% current\_time %}** va écraser de manière aveugle la valeur de cette éventuelle variable. Une solution plus propre est de permettre à la balise de gabarit de définir le nom de la variable produite, comme ceci :

{% **current\_time** "%Y-%m-%d %I:%M %p" **as** my\_current\_time %}

<**p**>The current time is {{ my\_current\_time }}.</**p**>

Pour faire cela, il faudra modifier à la fois la fonction de compilation et la classe **Node**, comme ceci :

**import** **re**

**class** **CurrentTimeNode3**(template.Node):

**def** \_\_init\_\_(self, format\_string, var\_name):

self.format\_string = format\_string

self.var\_name = var\_name

**def** render(self, context):

context[self.var\_name] = datetime.datetime.now().strftime(self.format\_string)

**return** ''

**def** do\_current\_time(parser, token):

*# This version uses a regular expression to parse tag contents.*

**try**:

*# Splitting by None == splitting by spaces.*

tag\_name, arg = token.contents.split(**None**, 1)

**except** **ValueError**:

**raise** template.TemplateSyntaxError(

"**%r** tag requires arguments" % token.contents.split()[0]

)

m = re.search(r'(.\*?) as (\w+)', arg)

**if** **not** m:

**raise** template.TemplateSyntaxError("**%r** tag had invalid arguments" % tag\_name)

format\_string, var\_name = m.groups()

**if** **not** (format\_string[0] == format\_string[-1] **and** format\_string[0] **in** ('"', "'")):

**raise** template.TemplateSyntaxError(

"**%r** tag's argument should be in quotes" % tag\_name

)

**return** CurrentTimeNode3(format\_string[1:-1], var\_name)

La différence ici est que **do\_current\_time()** capture la chaîne de format et le nom de la variable, les transmettant les deux à **CurrentTimeNode3**.

Finalement, si vous n’avez besoin que d’une syntaxe simple pour votre balise de gabarit de mise à jour du contexte, envisagez l’utilisation du raccourci **[simple\_tag()](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/" \l "django.template.Library.simple_tag" \o "django.template.Library.simple_tag)** qui permet d’attribuer le résultat de la balise à une variable de gabarit.

### Analyse jusqu’à la prochaine balise de bloc[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#parsing-until-another-block-tag)

Les balises de gabarit peuvent travailler en tandem. Par exemple, la balise [**{% comment %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-comment) standard masque tout jusqu’à la prochaine occurrence de **{% endcomment %}**. Pour créer une balise de gabarit semblable, utilisez **parser.parse()** dans votre fonction de compilation.

Voici comment une balise **{% comment %}** simplifiée pourrait être écrite :

**def** do\_comment(parser, token):

nodelist = parser.parse(('endcomment',))

parser.delete\_first\_token()

**return** CommentNode()

**class** **CommentNode**(template.Node):

**def** render(self, context):

**return** ''

**Note**

L’implémentation effective de [**{% comment %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-comment) est légèrement différente dans la mesure où elle autorise des balises incorrectes à apparaître entre **{% comment %}** et **{% endcomment %}**. Elle fait cela en appelant **parser.skip\_past('endcomment')** au lieu de **parser.parse(('endcomment',))**, suivi par **parser.delete\_first\_token()**, ce qui évite de générer une liste de nœuds.

**parser.parse()** accepte un tuple de noms de balises bloc définissant les limites d’analyse. Elle renvoie une instance de **django.template.NodeList** contenant une liste de tous les objets **Node** que l’analyseur a rencontré avant que n’apparaisse l’une des balises indiquées dans le tuple.

Dans **nodelist = parser.parse(('endcomment',))** de l’exemple ci-dessus, **nodelist** est une liste de tous les nœuds entre **{% comment %}** et **{% endcomment %}**, à l’exclusion de **{% comment %}** et **{% endcomment %}**.

Après l’appel à **parser.parse()**, l’analyseur n’a pas encore « consommé » la balise **{% endcomment %}**, c’est pourquoi le code doit explicitement appeler **parser.delete\_first\_token()**.

**CommentNode.render()** renvoie une chaîne vide. Tout ce qui se trouve entre **{% comment %}** et **{% endcomment %}** est ignoré.

### Analyse jusqu’à la prochaine balise de bloc et enregistrement du contenu[¶](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#parsing-until-another-block-tag-and-saving-contents)

Dans l’exemple prédédent, **do\_comment()** a ignoré tout ce qui apparaissait entre **{% comment %}** et **{% endcomment %}**. Au lieu de faire cela, il est possible de faire quelque chose avec le code situé entre les balises de bloc.

Par exemple, voici une balise de gabarit personnalisée, **{% upper %}**, qui met en majuscules tout ce qui se trouve entre elle et **{% endupper %}**.

Utilisation :

{% **upper** %}This will appear in uppercase, {{ your\_name }}.{% **endupper** %}

Comme dans l’exemple précédent, nous utilisons **parser.parse()**. Mais cette fois-ci, nous passons le contenu de **nodelist** à **Node**:

**def** do\_upper(parser, token):

nodelist = parser.parse(('endupper',))

parser.delete\_first\_token()

**return** UpperNode(nodelist)

**class** **UpperNode**(template.Node):

**def** \_\_init\_\_(self, nodelist):

self.nodelist = nodelist

**def** render(self, context):

output = self.nodelist.render(context)

**return** output.upper()

Le seul nouveau concept ici est l’appel à **self.nodelist.render(context)** dans **UpperNode.render()**.

Pour plus d’exemples de rendus complexes, consultez le code source des balises [**{% for %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-for) dans **django/template/defaulttags.py** et [**{% if %}**](https://docs.djangoproject.com/fr/3.1/ref/templates/builtins/#std:templatetag-if) dans **django/template/smartif.py**.

[**Moteur de gabarit personnalisé**](https://docs.djangoproject.com/fr/3.1/howto/custom-template-backend/)

[**Écriture d’un système de stockage personnalisé**](https://docs.djangoproject.com/fr/3.1/howto/custom-file-storage/)

# **Informations supplémentaires**

## **Support Django!**

* [PyLab donated to the Django Software Foundation to support Django development. Donate today!](https://www.djangoproject.com/fundraising/)

## **Sommaire**

* [Balises et filtres de gabarit personnalisés](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/)
  + [Disposition du code](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#code-layout)
  + [Écriture de filtres de gabarits personnalisés](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-custom-template-filters)
    - [Inscription de filtres personnalisés](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#registering-custom-filters)
    - [Filtres de gabarits agissant sur des chaînes](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#template-filters-that-expect-strings)
    - [Les filtres et l’échappement automatique](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#filters-and-auto-escaping)
    - [Filtres et fuseaux horaires](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#filters-and-time-zones)
  + [Écriture de balises de gabarits personnalisées](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-custom-template-tags)
    - [Balises simples](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#simple-tags)
    - [Balises d’inclusion](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#inclusion-tags)
    - [Balises de gabarits personnalisées (niveau avancé)](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#advanced-custom-template-tags)
    - [Aperçu rapide](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#a-quick-overview)
    - [Écriture de la fonction de compilation](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-the-compilation-function)
    - [Écriture de la fonction de rendu](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#writing-the-renderer)
    - [Considérations sur l’échappement automatique](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#auto-escaping-considerations)
    - [Considérations sur la concurrence entre « threads »](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#thread-safety-considerations)
    - [Inscription de la balise](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#registering-the-tag)
    - [Transmission de variables de gabarit à la balise](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#passing-template-variables-to-the-tag)
    - [Définition d’une variable dans le contexte](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#setting-a-variable-in-the-context)
    - [Analyse jusqu’à la prochaine balise de bloc](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#parsing-until-another-block-tag)
    - [Analyse jusqu’à la prochaine balise de bloc et enregistrement du contenu](https://docs.djangoproject.com/fr/3.1/howto/custom-template-tags/#parsing-until-another-block-tag-and-saving-contents)

## **Parcourir**

* Précédent : [Moteur de gabarit personnalisé](https://docs.djangoproject.com/fr/3.1/howto/custom-template-backend/)
* Suivant : [Écriture d’un système de stockage personnalisé](https://docs.djangoproject.com/fr/3.1/howto/custom-file-storage/)
* [Table des matières](https://docs.djangoproject.com/fr/3.1/contents/)
* [Index général](https://docs.djangoproject.com/fr/3.1/genindex/)
* [Index des modules Python](https://docs.djangoproject.com/fr/3.1/py-modindex/)

## **Vous êtes ici :**

* [Documentation de Django 3.1](https://docs.djangoproject.com/fr/3.1/)
  + [Guides pratiques](https://docs.djangoproject.com/fr/3.1/howto/)
    - Balises et filtres de gabarit personnalisés

## **Obtenir de l'aide**

[FAQ](https://docs.djangoproject.com/fr/3.1/faq/)

Essayez la FAQ, vous y trouverez des réponses à de nombreuses questions courantes.

[Index](https://docs.djangoproject.com/en/stable/genindex/), [Index des modules](https://docs.djangoproject.com/en/stable/py-modindex/), or [Sommaire](https://docs.djangoproject.com/en/stable/contents/)

Pratique lorsqu'on cherche des informations précises.

[django-users mailing list](https://groups.google.com/group/django-users/)

Pour chercher des informations dans les archives de la liste de diffusion django-users, ou pour poser une question.

[#django IRC channel](irc://irc.freenode.net/django)

Posez une question dans le canal IRC #django (#django-fr pour le français), ou cherchez dans les journaux IRC pour trouver une éventuelle réponse existante.

[Ticket tracker](https://code.djangoproject.com/)

Signalez des bogues de Django ou de sa documentation dans notre système de suivi de tickets.

## **Télécharger :**

Hors ligne (Django 3.1) : [HTML](https://media.djangoproject.com/docs/django-docs-3.1-fr.zip) | [PDF](https://media.readthedocs.org/pdf/django/3.1.x/django.pdf) | [ePub](https://media.readthedocs.org/epub/django/3.1.x/django.epub)  
Offert par [Read the Docs](https://readthedocs.org/).

# **Django Links**

## Learn More

* [About Django](https://www.djangoproject.com/start/overview/)
* [Getting Started with Django](https://www.djangoproject.com/start/)
* [Team Organization](https://docs.djangoproject.com/en/dev/internals/organization/)
* [Django Software Foundation](https://www.djangoproject.com/foundation/)
* [Code of Conduct](https://www.djangoproject.com/conduct/)
* [Diversity Statement](https://www.djangoproject.com/diversity/)

## Get Involved

* [Join a Group](https://www.djangoproject.com/community/)
* [Contribute to Django](https://docs.djangoproject.com/en/dev/internals/contributing/)
* [Submit a Bug](https://docs.djangoproject.com/en/dev/internals/contributing/bugs-and-features/)
* [Report a Security Issue](https://docs.djangoproject.com/en/dev/internals/security/#reporting-security-issues)

## Follow Us

* [GitHub](https://github.com/django)
* [Twitter](https://twitter.com/djangoproject)
* [News RSS](https://www.djangoproject.com/rss/weblog/)
* [Django Users Mailing List](https://groups.google.com/forum/#!forum/django-users)

## Support Us

* [Sponsor Django](https://www.djangoproject.com/fundraising/)
* [Official merchandise store](https://django.threadless.com/)
* [Amazon Smile](https://docs.djangoproject.com/foundation/donate/#amazon-smile)
* [Benevity Workplace Giving Program](https://docs.djangoproject.com/foundation/donate/#benevity-giving)

[**Django**](https://www.djangoproject.com/)

* Hosting by[Rackspace](https://www.rackspace.com/)
* Design by[Threespot](https://www.threespot.com/) &

© 2005-2021 [Django Software Foundation](https://www.djangoproject.com/foundation/) and individual contributors. Django is a [registered trademark](https://www.djangoproject.com/trademarks/) of the Django Software Foundation.