

---

# Argument Clinic How-To

Version 3.9.2

Guido van Rossum  
and the Python development team

avril 02, 2021

Python Software Foundation  
Email : docs@python.org

## Table des matières

<b>1</b>	<b>Les objectifs d'Argument Clinic</b>	<b>2</b>
<b>2</b>	<b>Concepts de base et utilisation</b>	<b>3</b>
<b>3</b>	<b>Convertissez votre première fonction</b>	<b>4</b>
<b>4</b>	<b>Sujets avancés</b>	<b>9</b>
4.1	Valeurs par défaut . . . . .	9
4.2	Renaming the C functions and variables generated by Argument Clinic . . . . .	9
4.3	Conversion des fonctions en utilisant <i>PyArg_UnpackTuple</i> . . . . .	10
4.4	Groupes optionnels . . . . .	10
4.5	Using real Argument Clinic converters, instead of "legacy converters" . . . . .	11
4.6	Py_buffer . . . . .	14
4.7	Advanced converters . . . . .	14
4.8	Parameter default values . . . . .	14
4.9	The <code>NULL</code> default value . . . . .	15
4.10	Expressions specified as default values . . . . .	15
4.11	Using a return converter . . . . .	16
4.12	Cloning existing functions . . . . .	16
4.13	Calling Python code . . . . .	17
4.14	Using a "self converter" . . . . .	17
4.15	Writing a custom converter . . . . .	18
4.16	Writing a custom return converter . . . . .	19
4.17	<code>METH_O</code> and <code>METH_NOARGS</code> . . . . .	19
4.18	<code>tp_new</code> and <code>tp_init</code> functions . . . . .	20
4.19	Changing and redirecting Clinic's output . . . . .	20
4.20	The <code>#ifdef</code> trick . . . . .	23
4.21	Using Argument Clinic in Python files . . . . .	24
	<b>Index</b>	<b>25</b>

---

## Résumé

Argument Clinic est un préprocesseur pour les fichiers C de CPython. Il permet d'automatiser les tâches répétitives lors de la rédaction du code d'analyse d'arguments pour les modules natifs. Ce document vous montre comment convertir votre première fonction C de façon à ce qu'elle fonctionne avec Argument Clinic, avant d'introduire des usages plus avancés d'Argument Clinic.

Argument Clinic est pour le moment considéré comme un outil interne à CPython. Il n'est pas conçu pour gérer des fichiers à l'extérieur de CPython, et sa compatibilité ascendante n'est pas garantie pour les versions futures. En d'autres termes, si vous êtes maintenant d'une extension C pour CPython, vous pouvez bien sûr expérimenter avec Argument Clinic sur votre propre code. Mais la version d'Argument Clinic livrée avec la prochaine version de CPython *pourrait* être totalement incompatible et casser tout votre code.

## 1 Les objectifs d'Argument Clinic

Le premier objectif d'Argument Clinic est de prendre en charge toute l'analyse d'arguments à l'intérieur de CPython. Cela signifie que si vous convertissez une fonction pour fonctionner avec Argument Clinic, cette fonction n'a plus du tout besoin d'analyser ses propres arguments. Le code généré par Argument Clinic doit être une « boîte noire » avec en entrée l'appel de CPython, et en sortie l'appel à votre code. Entre les deux, `PyObject *args` (et éventuellement `PyObject *kwargs`) sont convertis magiquement dans les variables et types C dont vous avez besoin.

Pour que le premier objectif d'Argument Clinic soit atteint, il faut qu'il soit facile à utiliser. Actuellement, travailler avec la bibliothèque d'analyse d'arguments de CPython est une corvée. Il faut maintenir une quantité surprenante d'informations redondantes. En utilisant Argument Clinic, il n'est plus nécessaire de se répéter.

Certainement, personne ne voudrait utiliser Argument Clinic s'il ne réglait pas leur problème -- sans en créer de nouveaux. Il est donc de la première importance qu'Argument Clinic génère du code correct. Il est aussi souhaitable que le code soit aussi plus rapide. Au minimum, il ne doit pas introduire de régression significative sur la vitesse d'exécution. (Au bout du compte, Argument Clinic *devrait* permettre une accélération importante -- on pourrait ré-écrire son générateur de code pour produire du code d'analyse d'argument adapté au mieux, plutôt que d'utiliser la bibliothèque d'analyse d'argument générique. On aurait ainsi l'analyse d'argument la plus rapide possible !)

De plus, Argument Clinic doit être suffisamment flexible pour s'accommoder d'approches différentes de l'analyse d'arguments. Il y a des fonctions dans Python dont le traitement des arguments est très étrange ; le but d'Argument Clinic est de les gérer toutes.

Finalement, la motivation première d'Argument Clinic était de fournir des « signatures » pour l'inspection des composants natifs de CPython. Précédemment, les fonctions d'inspection levaient une exception si vous passiez un composant natif. Grâce à Argument Clinic, ce comportement appartient au passé !

En travaillant avec Argument Clinic, il faut garder à l'esprit que plus vous lui donnez de détails, meilleur sera son boulot. Argument Clinic est bien sûr assez simple pour le moment. Mais à mesure qu'il évoluera et deviendra plus sophistiqué, il sera capable de faire beaucoup de choses intéressantes et intelligentes à partir de l'information que vous lui fournissez.

## 2 Concepts de base et utilisation

Argument Clinic est livré avec CPython ; vous le trouverez dans `Tools/clinic/clinic.py`. Si vous exécutez ce script, en spécifiant un fichier C comme argument :

```
$ python3 Tools/clinic/clinic.py foo.c
```

Argument Clinic va parcourir le fichier en cherchant cette ligne :

```
/*[clinic input]
```

Lorsqu'il en trouve une, il lit tout ce qui suit, jusqu'à cette ligne :

```
[clinic start generated code]*/
```

Tout ce qui se trouve entre ces deux lignes est une entrée pour Argument Clinic. Toutes ces lignes, en incluant les commentaires de début et de fin, sont appelées collectivement un « bloc » d'Argument Clinic.

Lorsque *Argument Clinic* analyse l'un de ces blocs, il produit une sortie. Cette sortie est réécrite dans le fichier C immédiatement après le bloc, suivie par un commentaire contenant une somme de contrôle. Le bloc Argument Clinic ressemble maintenant à cela :

```
/*[clinic input]
... clinic input goes here ...
[clinic start generated code]*/
... clinic output goes here ...
/*[clinic end generated code: checksum=...]*/
```

Si vous exécutez de nouveau Argument Clinic sur ce même fichier, Argument Clinic supprime la vieille sortie, et écrit la nouvelle sortie avec une ligne de somme de contrôle mise à jour. Cependant, si l'entrée n'a pas changé, la sortie ne change pas non plus.

Vous ne devez jamais modifier la sortie d'un bloc Argument Clinic. Changez plutôt l'entrée jusqu'à obtenir la sortie souhaitée (c'est précisément le but de la somme de contrôle, détecter si la sortie a été changée. En effet, ces modifications seront perdues après que Argument Clinic a écrit une nouvelle sortie).

Par souci de clarté, voilà la terminologie que nous emploierons :

- La première ligne du commentaire (`/*[clinic input]`) est la *ligne de début*.
- La dernière ligne du commentaire initial (`[clinic start generated code]*/`) est la *ligne de fin*.
- La dernière ligne (`/*[clinic end generated code: checksum=...]*/`) est la *ligne de contrôle*.
- On appelle *entrée* ce qui se trouve entre la ligne de début et la ligne de fin.
- Et on appelle *sortie* ce qui se trouve entre la ligne de fin et la ligne de contrôle.
- L'ensemble du texte, depuis la ligne de début jusqu'à la ligne de contrôle incluse s'appelle le *bloc*. (Un bloc qui n'a pas encore été traité avec succès par Argument Clinic n'a pas encore de sortie ni de ligne de contrôle mais est quand même considéré comme un bloc)

### 3 Convertissez votre première fonction

La meilleure manière de comprendre le fonctionnement d'Argument Clinic est de convertir une fonction. Voici donc les étapes minimales que vous devez suivre pour convertir une fonction de manière à ce qu'elle fonctionne avec Argument Clinic. Remarquez que pour du code que vous comptez inclure dans CPython, vous devrez certainement pousser plus loin la conversion, en utilisant les concepts avancés que nous verrons plus loin dans ce document (comme `return converters` et `self converters`). Mais concentrons nous pour le moment sur les choses simples.

En route !

0. Assurez-vous que vous travaillez sur une copie récemment mise à jour du code de CPython.
1. Trouvez une fonction native de Python qui fait appel à `PyArg_ParseTuple()` ou `PyArg_ParseTupleAndKeywords()`, et n'a pas encore été convertie par Argument Clinic. Pour cet exemple, j'utilise `_pickle.Pickler.dump()`.
2. Si l'appel à `PyArg_Parse` utilise l'un des formats suivants :

```
O&
O!
es
es#
et
et#
```

ou s'il y a de multiples appels à `PyArg_ParseTuple()`, choisissez une fonction différente. Argument Clinic gère tous ces scénarios, mais se sont des sujets trop avancés pour notre première fonction.

Par ailleurs, si la fonction a des appels multiples à `PyArg_ParseTuple()` ou `PyArg_ParseTupleAndKeywords()` dans lesquels elle permet différents types pour les mêmes arguments, il n'est probablement pas possible de la convertir pour Argument Clinic. Argument Clinic ne gère pas les fonctions génériques ou les paramètres polymorphes.

3. Ajoutez les lignes standard suivantes au-dessus de votre fonction pour créer notre bloc :

```
/*[clinic input]
[clinic start generated code]*/
```

4. Coupez la *docstring* et collez-la entre les lignes commençant par `[clinic]`, en enlevant tout le bazar qui en fait une chaîne de caractères correcte en C. Une fois que c'est fait, vous devez avoir seulement le texte, aligné à gauche, sans ligne plus longue que 80 caractères (Argument Clinic préserve l'indentation à l'intérieur de la *docstring*).

Si l'ancienne *docstring* commençait par une ligne qui ressemble à une signature de fonction, supprimez cette ligne. (Elle n'est plus nécessaire pour la *docstring*. Dans le futur, quand vous utiliserez `help()` pour une fonction native, la première ligne sera construite automatiquement à partir de la signature de la fonction.)

Échantillon :

```
/*[clinic input]
Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

5. Si votre *docstring* ne contient pas de ligne « résumé », Argument Clinic va se plaindre. Assurons-nous donc qu'il y en a une. La ligne « résumé » doit être un paragraphe consistant en une seule ligne de 80 colonnes au début de la *docstring*.

Dans notre exemple, la *docstring* est seulement composée d'une ligne de résumé, donc le code ne change pas à cette étape.

6. Au dessus de la *docstring*, entrez le nom de la fonction, suivi d'une ligne vide. Ce doit être le nom de la fonction en Python et être le chemin complet jusqu'à la fonction. Il doit commencer par le nom du module, suivi de tous les sous-modules, puis, si la fonction est une méthode de classe, inclure aussi le nom de la classe.

Échantillon :

```

/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

```

7. Si c'est la première fois que ce module ou cette classe est utilisée avec Argument Clinic dans ce fichier C, vous devez déclarer votre module et/ou votre classe. Pour suivre de bonnes pratiques avec Argument Clinic, il vaut mieux faire ces déclarations quelque part en tête du fichier C, comme les fichiers inclus et les statiques (dans cet extrait, nous montrons les deux blocs à côté l'un de l'autre).

Le nom de la classe et du module doivent être les mêmes que ceux vus par Python. Selon le cas, référez-vous à `PyModuleDef` ou `PyTypeObject`

Quand vous déclarez une classe, vous devez aussi spécifier deux aspects de son type en C : la déclaration de type que vous utiliseriez pour un pointeur vers une instance de cette classe et un pointeur vers le `PyTypeObject` de cette classe.

Échantillon :

```

/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

```

8. Déclarez chacun des paramètres de la fonction. Chaque paramètre doit être sur une ligne séparée. Tous les paramètres doivent être indentés par rapport au nom de la fonction et à la *docstring*.

La forme générale de ces paramètres est la suivante :

```
name_of_parameter: converter
```

Si le paramètre a une valeur par défaut, ajoutez ceci après le convertisseur :

```
name_of_parameter: converter = default_value
```

Argument Clinic peut traiter les « valeurs par défaut » de manière assez sophistiquée ; voyez [la section ci-dessous sur les valeurs par défaut](#) pour plus de détails.

Ajoutez une ligne vide sous les paramètres.

Que fait le « convertisseur » ? Il établit à la fois le type de variable utilisé en C et la méthode pour convertir la valeur Python en valeur C lors de l'exécution. Pour le moment, vous allez utiliser ce qui s'appelle un « convertisseur hérité » -- une syntaxe de convenance qui facilite le portage de vieux code dans Argument Clinic.

Pour chaque paramètre, copiez l'« unité de format » de ce paramètre à partir de l'argument de `PyArg_Parse()` et spécifiez *ça* comme convertisseur, entre guillemets. (l'« unité de format » est le nom formel pour la partie du paramètre *format*, de un à trois caractères, qui indique à la fonction d'analyse d'arguments quel est le type de la variable et comment la convertir. Pour plus d'information sur les unités de format, voyez `arg-parsing`.)

Pour des unités de format de plusieurs caractères, comme `z#`, utilisez l'ensemble des 2 ou 3 caractères de la chaîne.

Échantillon :

```

/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

```

(suite sur la page suivante)

```

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

```

9. Si votre fonction a le caractère | dans son format, parce que certains paramètres ont des valeurs par défaut, vous pouvez l'ignorer. Argument Clinic infère quels paramètres sont optionnels selon s'ils ont ou non une valeur par défaut.

Si votre fonction a le caractère \$ dans son format, parce qu'elle n'accepte que des arguments nommés, spécifiez \* sur une ligne à part, avant le premier argument nommé, avec la même indentation que les lignes de paramètres. (\_pickle.Pickler.dump n'a ni l'un ni l'autre, donc notre exemple est inchangé.)

10. Si la fonction C existante appelle PyArg\_ParseTuple() (et pas PyArg\_ParseTupleAndKeywords()), alors tous ses arguments sont uniquement positionnels.

Pour marquer tous les paramètres comme uniquement positionnels dans Argument Clinic, ajoutez / sur une ligne à part après le dernier paramètre, avec la même indentation que les lignes de paramètres.

Pour le moment, c'est tout ou rien ; soit tous les paramètres sont uniquement positionnels, ou aucun ne l'est. (Dans le futur, Argument Clinic supprimera peut être cette restriction.)

Échantillon :

```

/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

```

11. Il est utile d'ajouter une *docstring* pour chaque paramètre, mais c'est optionnel ; vous pouvez passer cette étape si vous préférez.

Voici comment ajouter la *docstring* d'un paramètre. La première ligne doit être plus indentée que la définition du paramètre. La marge gauche de cette première ligne établit la marge gauche pour l'ensemble de la *docstring* de ce paramètre ; tout le texte que vous écrivez sera indenté de cette quantité. Vous pouvez écrire autant de texte que vous le souhaitez, sur plusieurs lignes.

Échantillon :

```

/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'

```

```

        The object to be pickled.
    /

    Write a pickled representation of obj to the open file.
    [clinic start generated code]*/

```

12. Enregistrez puis fermez le fichier, puis exécutez `Tools/clinic/clinic.py` dessus. Avec un peu de chance tout a fonctionné, votre bloc a maintenant une sortie, et un fichier `.c.h` a été généré ! Ré-ouvrez le fichier dans votre éditeur pour voir :

```

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

    Write a pickled representation of obj to the open file.
    [clinic start generated code]*/

static PyObject *
_pickle_Pickler_dump(PicklerObject *self, PyObject *obj)
/*[clinic end generated code: output=87ecad1261e02ac7 input=552eb1c0f52260d9]*/

```

Bien sûr, si Argument Clinic n'a pas produit de sortie, c'est qu'il a rencontré une erreur dans votre entrée. Corrigez vos erreurs et réessayez jusqu'à ce qu'Argument Clinic traite votre fichier sans problème.

Pour plus de visibilité, la plupart du code a été écrit dans un fichier `.c.h`. Vous devez l'inclure dans votre fichier `.c` original, typiquement juste après le bloc du module `clinic` :

```
#include "clinic/_pickle.c.h"
```

13. Vérifiez bien que le code d'analyse d'argument généré par Argument Clinic ressemble bien au code existant. Assurez vous premièrement que les deux codes utilisent la même fonction pour analyser les arguments. Le code existant doit appeler soit `PyArg_ParseTuple()` soit `PyArg_ParseTupleAndKeywords()` ; assurez vous que le code généré par Argument Clinic appelle *exactement* la même fonction. Deuxièmement, la chaîne de caractère du format passée dans `PyArg_ParseTuple()` ou `PyArg_ParseTupleAndKeywords()` doit être *exactement* la même que celle écrite à la main, jusqu'aux deux points ou au point virgule. (Argument Clinic génère toujours ses chaînes de format avec : suivi du nom de la fonction. Si la chaîne de format du code existant termine par ;, pour fournir une aide sur l'utilisation, ce changement n'a aucun effet, ne vous en souciez pas.) Troisièmement, pour des paramètres dont l'unité de format nécessite deux arguments (comme une variable de longueur, ou une chaîne d'encodage, ou un pointeur vers une fonction de conversion), assurez vous que ce deuxième argument est *exactement* le même entre les deux invocations. Quatrièmement, à l'intérieur de la section sortie du bloc, vous trouverez une macro pré-processeur qui définit les structures statiques `PyMethodDef` appropriées pour ce module natif :

```

#define __PICKLE_PICKLER_DUMP_METHODDEF \
{"dump", (PyCFunction)__pickle_Pickler_dump, METH_O, __pickle_Pickler_dump__doc__
↪},

```

Cette structure statique doit être *exactement* la même que la structure statique `PyMethodDef` existante pour ce module natif.

Si l'un de ces éléments diffère *de quelque façon que se soit*, ajustez la spécification de fonction d'Argument Clinic et exécutez de nouveau `Tools/clinic/clinic.py` jusqu'à ce qu'elles soient identiques.

14. Notez que la dernière ligne de cette sortie est la déclaration de votre fonction `impl`. C'est là que va l'implémentation de la fonction native. Supprimez le prototype de la fonction que vous modifiez, mais laissez l'accolade ouverte. Maintenant, supprimez tout le code d'analyse d'argument et les déclarations de toutes les variables auxquelles il assigne les arguments. Vous voyez que désormais, les arguments Python sont ceux de cette fonction `impl`; si l'implémentation utilise des noms différents pour ces variables, corrigez-les.

Comme c'est un peu bizarre, ça vaut la peine de répéter. Votre fonction doit ressembler à ça :

```
static return_type
your_function_impl(...)
/*[clinic end generated code: checksum=...]*/
{
    ...
}
```

Argument Clinic génère une ligne de contrôle et la fonction prototype juste au dessus. Vous devez écrire les accolades d'ouverture (et de fermeture) pour la fonction, et l'implémentation à l'intérieur.

Échantillon :

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/
/*[clinic end generated code:↵
↵checksum=da39a3ee5e6b4b0d3255bfe95601890afd80709]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

PyDoc_STRVAR(__pickle_Pickler_dump__doc__,
"Write a pickled representation of obj to the open file.\n"
"\n"
"...
static PyObject *
_pickle_Pickler_dump_impl(PicklerObject *self, PyObject *obj)
/*[clinic end generated code:↵
↵checksum=3bd30745bf206a48f8b576a1da3d90f55a0a4187]*/
{
    /* Check whether the Pickler was initialized correctly (issue3664).
       Developers often forget to call __init__() in their subclasses, which
       would trigger a segfault without this check. */
    if (self->write == NULL) {
        PyErr_Format(PicklingError,
                     "Pickler.__init__() was not called by %s.__init__()",
                     Py_TYPE(self)->tp_name);
        return NULL;
    }

    if (_Pickler_ClearBuffer(self) < 0)
        return NULL;

    ...
}
```

15. Vous vous souvenez de la macro avec la structure `PyMethodDef` pour cette fonction? Trouvez la structure



`PyMethodDef` existante pour cette fonction et remplacez là par une référence à cette macro. (Si la fonction native est définie au niveau d'un module, vous le trouverez certainement vers la fin du fichier; s'il s'agit d'une méthode de classe, se sera sans doute plus bas, mais relativement près de l'implémentation.)

Notez que le corps de la macro contient une virgule finale. Donc, lorsque vous remplacez la structure statique `PyMethodDef` par la macro, *n'ajoutez pas* de virgule à la fin.

Échantillon :

```
static struct PyMethodDef Pickler_methods[] = {
    __PICKLE_PICKLER_DUMP_METHODDEF
    __PICKLE_PICKLER_CLEAR_MEMO_METHODDEF
    {NULL, NULL} /* sentinel */
};
```

16. Compilez, puis faites tourner les portions idoines de la suite de tests de régressions. Ce changement ne doit introduire aucun nouveau message d'erreur ou avertissement à la compilation, et il ne devrait y avoir aucun changement visible de l'extérieur au comportement de Python.

Enfin, à part pour une différence : si vous exécutez `inspect.signature()` sur votre fonction, vous obtiendrez maintenant une signature valide !

Félicitations, vous avez adapté votre première fonction pour qu'elle utilise `Argument Clinic` !

## 4 Sujets avancés

Maintenant que vous avez un peu d'expérience avec `Argument Clinic`, c'est le moment pour des sujets avancés.

### 4.1 Valeurs par défaut

La valeur par défaut que vous fournissez pour un paramètre ne peut pas être n'importe quelle expression. Actuellement, ce qui est géré :

- Constantes numériques (entier ou nombre flottant)
- Chaînes constantes
- `True`, `False` et `None`
- Constantes symboliques simples comme `sys.maxsize`, qui doivent commencer par le nom du module

Si par curiosité vous voulez lire l'implémentation, c'est `from_builtin()` dans `Lib/inspect.py`.

(Dans le futur, il est possible que l'on ait besoin de l'améliorer, pour autoriser les expressions complètes comme `CONSTANT - 1`.)

### 4.2 Renaming the C functions and variables generated by `Argument Clinic`

`Argument Clinic` automatically names the functions it generates for you. Occasionally this may cause a problem, if the generated name collides with the name of an existing C function. There's an easy solution : override the names used for the C functions. Just add the keyword `"as"` to your function declaration line, followed by the function name you wish to use. `Argument Clinic` will use that function name for the base (generated) function, then add `"_impl"` to the end and use that for the name of the impl function.

Par exemple, si nous voulons renommer les noms de fonction C générés pour `pickle.Pickler.dump`, ça ressemblerait à ça :

```
/*[clinic input]
pickle.Pickler.dump as pickler_dumper
...
```

La fonction de base sera maintenant nommée `pickler_dumper()`, et la fonction *impl* serait maintenant nommé `pickler_dumper_impl()`.

De même, vous pouvez avoir un problème quand vous souhaitez donner à un paramètre un nom spécifique à Python, mais ce nom peut être gênant en C. Argument Clinic vous permet de donner à un paramètre des noms différents en Python et en C :

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
    file as file_obj: object
    protocol: object = NULL
    *
    fix_imports: bool = True
```

Here, the name used in Python (in the signature and the `keywords` array) would be `file`, but the C variable would be named `file_obj`.

Vous pouvez utiliser ceci pour renommer aussi le paramètre `self`

### 4.3 Conversion des fonctions en utilisant *PyArg\_UnpackTuple*

To convert a function parsing its arguments with `PyArg_UnpackTuple()`, simply write out all the arguments, specifying each as an `object`. You may specify the `type` argument to cast the type as appropriate. All arguments should be marked positional-only (add a `/` on a line by itself after the last argument).

Actuellement, le code généré utilise `PyArg_ParseTuple()`, mais cela va bientôt changer.

### 4.4 Groupes optionnels

Some legacy functions have a tricky approach to parsing their arguments : they count the number of positional arguments, then use a `switch` statement to call one of several different `PyArg_ParseTuple()` calls depending on how many positional arguments there are. (These functions cannot accept keyword-only arguments.) This approach was used to simulate optional arguments back before `PyArg_ParseTupleAndKeywords()` was created.

While functions using this approach can often be converted to use `PyArg_ParseTupleAndKeywords()`, optional arguments, and default values, it's not always possible. Some of these legacy functions have behaviors `PyArg_ParseTupleAndKeywords()` doesn't directly support. The most obvious example is the builtin function `range()`, which has an optional argument on the *left* side of its required argument! Another example is `curses.window.addch()`, which has a group of two arguments that must always be specified together. (The arguments are called `x` and `y`; if you call the function passing in `x`, you must also pass in `y`—and if you don't pass in `x` you may not pass in `y` either.)

In any case, the goal of Argument Clinic is to support argument parsing for all existing CPython builtins without changing their semantics. Therefore Argument Clinic supports this alternate approach to parsing, using what are called *optional groups*. Optional groups are groups of arguments that must all be passed in together. They can be to the left or the right of the required arguments. They can *only* be used with positional-only parameters.

---

**Note :** Optional groups are *only* intended for use when converting functions that make multiple calls to `PyArg_ParseTuple()` ! Functions that use *any* other approach for parsing arguments should *almost never* be converted to Argument Clinic using optional groups. Functions using optional groups currently cannot have accurate signatures in Python, because Python just doesn't understand the concept. Please avoid using optional groups wherever possible.

---

To specify an optional group, add a `[` on a line by itself before the parameters you wish to group together, and a `]` on a line by itself after these parameters. As an example, here's how `curses.window.addch` uses optional groups to make the first two parameters and the last parameter optional :

```
/*[clinic input]

curses.window.addch

    [
    x: int
        X-coordinate.
    y: int
        Y-coordinate.
    ]

    ch: object
        Character to add.

    [
    attr: long
        Attributes for the character.
    ]
/

...
```

Notes :

- For every optional group, one additional parameter will be passed into the `impl` function representing the group. The parameter will be an int named `group_{direction}_{number}`, where `{direction}` is either `right` or `left` depending on whether the group is before or after the required parameters, and `{number}` is a monotonically increasing number (starting at 1) indicating how far away the group is from the required parameters. When the `impl` is called, this parameter will be set to zero if this group was unused, and set to non-zero if this group was used. (By used or unused, I mean whether or not the parameters received arguments in this invocation.)
- S'il n'y a pas d'arguments requis, les groupes optionnels se comportent comme s'ils étaient à droite des arguments requis.
- In the case of ambiguity, the argument parsing code favors parameters on the left (before the required parameters).
- Optional groups can only contain positional-only parameters.
- Les groupes optionnels sont *seulement* destinés au code hérité. Ne les utilisez pas dans du nouveau code.

## 4.5 Using real Argument Clinic converters, instead of "legacy converters"

To save time, and to minimize how much you need to learn to achieve your first port to Argument Clinic, the walkthrough above tells you to use "legacy converters". "Legacy converters" are a convenience, designed explicitly to make porting existing code to Argument Clinic easier. And to be clear, their use is acceptable when porting code for Python 3.4.

However, in the long term we probably want all our blocks to use Argument Clinic's real syntax for converters. Why? A couple reasons :

- The proper converters are far easier to read and clearer in their intent.
- There are some format units that are unsupported as "legacy converters", because they require arguments, and the legacy converter syntax doesn't support specifying arguments.
- In the future we may have a new argument parsing library that isn't restricted to what `PyArg_ParseTuple()` supports; this flexibility won't be available to parameters using legacy converters.

Therefore, if you don't mind a little extra effort, please use the normal converters instead of legacy converters.

In a nutshell, the syntax for Argument Clinic (non-legacy) converters looks like a Python function call. However, if there are no explicit arguments to the function (all functions take their default values), you may omit the parentheses. Thus

`bool` and `bool()` are exactly the same converters.

All arguments to Argument Clinic converters are keyword-only. All Argument Clinic converters accept the following arguments :

**c\_default** The default value for this parameter when defined in C. Specifically, this will be the initializer for the variable declared in the "parse function". See [the section on default values](#) for how to use this. Specified as a string.

**annotation** The annotation value for this parameter. Not currently supported, because [PEP 8](#) mandates that the Python library may not use annotations.

In addition, some converters accept additional arguments. Here is a list of these arguments, along with their meanings :

**accept** A set of Python types (and possibly pseudo-types) ; this restricts the allowable Python argument to values of these types. (This is not a general-purpose facility ; as a rule it only supports specific lists of types as shown in the legacy converter table.)

To accept `None`, add `NoneType` to this set.

**bitwise** Only supported for unsigned integers. The native integer value of this Python argument will be written to the parameter without any range checking, even for negative values.

**converter** Only supported by the `object` converter. Specifies the name of a C "converter function" to use to convert this object to a native type.

**encoding** Only supported for strings. Specifies the encoding to use when converting this string from a Python `str` (Unicode) value into a C `char *` value.

**subclass\_of** Only supported for the `object` converter. Requires that the Python value be a subclass of a Python type, as expressed in C.

**type** Only supported for the `object` and `self` converters. Specifies the C type that will be used to declare the variable. Default value is `"PyObject *"`.

**zeroes** Only supported for strings. If true, embedded NUL bytes (`'\0'`) are permitted inside the value. The length of the string will be passed in to the `impl` function, just after the string parameter, as a parameter named `<parameter_name>_length`.

Please note, not every possible combination of arguments will work. Usually these arguments are implemented by specific `PyArg_ParseTuple` *format units*, with specific behavior. For example, currently you cannot call `unsigned_short` without also specifying `bitwise=True`. Although it's perfectly reasonable to think this would work, these semantics don't map to any existing format unit. So Argument Clinic doesn't support it. (Or, at least, not yet.)

Below is a table showing the mapping of legacy converters into real Argument Clinic converters. On the left is the legacy converter, on the right is the text you'd replace it with.

'B'	<code>unsigned_char(bitwise=True)</code>
'b'	<code>unsigned_char</code>
'c'	<code>char</code>
'C'	<code>int(accept={str})</code>
'd'	<code>double</code>
'D'	<code>Py_complex</code>
'es'	<code>str(encoding='name_of_encoding')</code>
'es#'	<code>str(encoding='name_of_encoding', zeroes=True)</code>
'et'	<code>str(encoding='name_of_encoding', accept={bytes, bytearray, str})</code>
'et#'	<code>str(encoding='name_of_encoding', accept={bytes, bytearray, str}, zeroes=True)</code>
'f'	<code>float</code>
'h'	<code>short</code>
'H'	<code>unsigned_short(bitwise=True)</code>
'i'	<code>int</code>
'I'	<code>unsigned_int(bitwise=True)</code>

Suite sur la page suivante

Tableau 1 – suite de la page précédente

'k'	unsigned_long (bitwise=True)
'K'	unsigned_long_long (bitwise=True)
'l'	long
'L'	long long
'n'	Py_ssize_t
'O'	object
'O!'	object (subclass_of='&PySomething_Type')
'O&'	object (converter='name_of_c_function')
'p'	bool
'S'	PyBytesObject
's'	str
's#'	str (zeroes=True)
's*'	Py_buffer (accept={buffer, str})
'U'	unicode
'u'	Py_UNICODE
'u#'	Py_UNICODE (zeroes=True)
'w*'	Py_buffer (accept={rwbuffer})
'Y'	PyByteArrayObject
'y'	str (accept={bytes})
'y#'	str (accept={robuffer}, zeroes=True)
'y*'	Py_buffer
'Z'	Py_UNICODE (accept={str, NoneType})
'Z#'	Py_UNICODE (accept={str, NoneType}, zeroes=True)
'z'	str (accept={str, NoneType})
'z#'	str (accept={str, NoneType}, zeroes=True)
'z*'	Py_buffer (accept={buffer, str, NoneType})

As an example, here's our sample `pickle.Pickler.dump` using the proper converter :

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

One advantage of real converters is that they're more flexible than legacy converters. For example, the `unsigned_int` converter (and all the `unsigned_` converters) can be specified without `bitwise=True`. Their default behavior performs range checking on the value, and they won't accept negative numbers. You just can't do that with a legacy converter !

Argument Clinic will show you all the converters it has available. For each converter it'll show you all the parameters it accepts, along with the default value for each parameter. Just run `Tools/clinic/clinic.py --converters` to see the full list.

## 4.6 Py\_buffer

When using the `Py_buffer` converter (or the `'s*'`, `'w*'`, `'*y'`, or `'z*'` legacy converters), you *must* not call `PyBuffer_Release()` on the provided buffer. Argument Clinic generates code that does it for you (in the parsing function).

## 4.7 Advanced converters

Remember those format units you skipped for your first time because they were advanced? Here's how to handle those too.

The trick is, all those format units take arguments—either conversion functions, or types, or strings specifying an encoding. (But "legacy converters" don't support arguments. That's why we skipped them for your first function.) The argument you specified to the format unit is now an argument to the converter; this argument is either `converter` (for `O&`), `subclass_of` (for `O!`), or `encoding` (for all the format units that start with `e`).

When using `subclass_of`, you may also want to use the other custom argument for `object()` : `type`, which lets you set the type actually used for the parameter. For example, if you want to ensure that the object is a subclass of `PyUnicode_Type`, you probably want to use the converter `object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')`.

One possible problem with using Argument Clinic : it takes away some possible flexibility for the format units starting with `e`. When writing a `PyArg_Parse` call by hand, you could theoretically decide at runtime what encoding string to pass in to `PyArg_ParseTuple()`. But now this string must be hard-coded at Argument-Clinic-preprocessing-time. This limitation is deliberate; it made supporting this format unit much easier, and may allow for future optimizations. This restriction doesn't seem unreasonable; CPython itself always passes in static hard-coded encoding strings for parameters whose format units start with `e`.

## 4.8 Parameter default values

Default values for parameters can be any of a number of values. At their simplest, they can be string, int, or float literals :

```
foo: str = "abc"
bar: int = 123
bat: float = 45.6
```

They can also use any of Python's built-in constants :

```
yep: bool = True
nope: bool = False
nada: object = None
```

There's also special support for a default value of `NULL`, and for simple expressions, documented in the following sections.

## 4.9 The `NULL` default value

For string and object parameters, you can set them to `None` to indicate that there's no default. However, that means the C variable will be initialized to `Py_None`. For convenience's sake, there's a special value called `NULL` for just this reason : from Python's perspective it behaves like a default value of `None`, but the C variable is initialized with `NULL`.

## 4.10 Expressions specified as default values

The default value for a parameter can be more than just a literal value. It can be an entire expression, using math operators and looking up attributes on objects. However, this support isn't exactly simple, because of some non-obvious semantics.

Examinons l'exemple suivant :

```
foo: Py_ssize_t = sys.maxsize - 1
```

`sys.maxsize` can have different values on different platforms. Therefore Argument Clinic can't simply evaluate that expression locally and hard-code it in C. So it stores the default in such a way that it will get evaluated at runtime, when the user asks for the function's signature.

What namespace is available when the expression is evaluated? It's evaluated in the context of the module the builtin came from. So, if your module has an attribute called `"max_widgets"`, you may simply use it :

```
foo: Py_ssize_t = max_widgets
```

If the symbol isn't found in the current module, it fails over to looking in `sys.modules`. That's how it can find `sys.maxsize` for example. (Since you don't know in advance what modules the user will load into their interpreter, it's best to restrict yourself to modules that are preloaded by Python itself.)

Evaluating default values only at runtime means Argument Clinic can't compute the correct equivalent C default value. So you need to tell it explicitly. When you use an expression, you must also specify the equivalent expression in C, using the `c_default` parameter to the converter :

```
foo: Py_ssize_t(c_default="PY_SSIZE_T_MAX - 1") = sys.maxsize - 1
```

Another complication : Argument Clinic can't know in advance whether or not the expression you supply is valid. It parses it to make sure it looks legal, but it can't *actually* know. You must be very careful when using expressions to specify values that are guaranteed to be valid at runtime !

Finally, because expressions must be representable as static C values, there are many restrictions on legal expressions. Here's a list of Python features you're not permitted to use :

- Function calls.
- Inline if statements (`3 if foo else 5`).
- Automatic sequence unpacking (`*[1, 2, 3]`).
- List/set/dict comprehensions and generator expressions.
- Tuple/list/set/dict literals.

## 4.11 Using a return converter

By default the impl function Argument Clinic generates for you returns `PyObject *`. But your C function often computes some C type, then converts it into the `PyObject *` at the last moment. Argument Clinic handles converting your inputs from Python types into native C types—why not have it convert your return value from a native C type into a Python type too?

That's what a "return converter" does. It changes your impl function to return some C type, then adds code to the generated (non-impl) function to handle converting that value into the appropriate `PyObject *`.

The syntax for return converters is similar to that of parameter converters. You specify the return converter like it was a return annotation on the function itself. Return converters behave much the same as parameter converters; they take arguments, the arguments are all keyword-only, and if you're not changing any of the default arguments you can omit the parentheses.

(If you use both "as" *and* a return converter for your function, the "as" should come before the return converter.)

There's one additional complication when using return converters : how do you indicate an error has occurred ? Normally, a function returns a valid (non-NULL) pointer for success, and NULL for failure. But if you use an integer return converter, all integers are valid. How can Argument Clinic detect an error ? Its solution : each return converter implicitly looks for a special value that indicates an error. If you return that value, and an error has been set (`PyErr_Occurred()` returns a true value), then the generated code will propagate the error. Otherwise it will encode the value you return like normal.

Currently Argument Clinic supports only a few return converters :

```
bool
int
unsigned int
long
unsigned int
size_t
Py_ssize_t
float
double
DecodeFSDefault
```

None of these take parameters. For the first three, return -1 to indicate error. For `DecodeFSDefault`, the return type is `const char *`; return a NULL pointer to indicate an error.

(There's also an experimental `NoneType` converter, which lets you return `Py_None` on success or NULL on failure, without having to increment the reference count on `Py_None`. I'm not sure it adds enough clarity to be worth using.)

To see all the return converters Argument Clinic supports, along with their parameters (if any), just run `Tools/clinic/clinic.py --converters` for the full list.

## 4.12 Cloning existing functions

If you have a number of functions that look similar, you may be able to use Clinic's "clone" feature. When you clone an existing function, you reuse :

- its parameters, including
  - their names,
  - their converters, with all parameters,
  - their default values,
  - their per-parameter docstrings,
  - their *kind* (whether they're positional only, positional or keyword, or keyword only), and
- its return converter.



The only thing not copied from the original function is its docstring; the syntax allows you to specify a new docstring.

Here's the syntax for cloning a function :

```
/*[clinic input]
module.class.new_function [as c_basename] = module.class.existing_function

Docstring for new_function goes here.
[clinic start generated code]*/
```

(The functions can be in different modules or classes. I wrote `module.class` in the sample just to illustrate that you must use the full path to *both* functions.)

Sorry, there's no syntax for partially-cloning a function, or cloning a function then modifying it. Cloning is an all-or nothing proposition.

Also, the function you are cloning from must have been previously defined in the current file.

## 4.13 Calling Python code

The rest of the advanced topics require you to write Python code which lives inside your C file and modifies Argument Clinic's runtime state. This is simple : you simply define a Python block.

A Python block uses different delimiter lines than an Argument Clinic function block. It looks like this :

```
/*[python input]
# python code goes here
[python start generated code]*/
```

All the code inside the Python block is executed at the time it's parsed. All text written to stdout inside the block is redirected into the "output" after the block.

As an example, here's a Python block that adds a static integer variable to the C code :

```
/*[python input]
print('static int __ignored_unused_variable__ = 0;')
[python start generated code]*/
static int __ignored_unused_variable__ = 0;
/*[python checksum:...]*/
```

## 4.14 Using a "self converter"

Argument Clinic automatically adds a "self" parameter for you using a default converter. It automatically sets the `type` of this parameter to the "pointer to an instance" you specified when you declared the type. However, you can override Argument Clinic's converter and specify one yourself. Just add your own `self` parameter as the first parameter in a block, and ensure that its converter is an instance of `self_converter` or a subclass thereof.

What's the point? This lets you override the type of `self`, or give it a different default name.

How do you specify the custom type you want to cast `self` to? If you only have one or two functions with the same type for `self`, you can directly use Argument Clinic's existing `self` converter, passing in the type you want to use as the `type` parameter :

```
/*[clinic input]

_pickle.Pickler.dump
```

(suite sur la page suivante)

```

self: self(type="PicklerObject ")
obj: object
/

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/

```

On the other hand, if you have a lot of functions that will use the same type for `self`, it's best to create your own converter, subclassing `self_converter` but overwriting the `type` member :

```

/*[python input]
class PicklerObject_converter(self_converter):
    type = "PicklerObject "
[python start generated code]*/

/*[clinic input]

_pickle.Pickler.dump

self: PicklerObject
obj: object
/

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/

```

## 4.15 Writing a custom converter

As we hinted at in the previous section... you can write your own converters ! A converter is simply a Python class that inherits from `CConverter`. The main purpose of a custom converter is if you have a parameter using the `O&` format unit—parsing this parameter means calling a `PyArg_ParseTuple()` "converter function".

Your converter class should be named `*something*_converter`. If the name follows this convention, then your converter class will be automatically registered with Argument Clinic; its name will be the name of your class with the `_converter` suffix stripped off. (This is accomplished with a metaclass.)

You shouldn't subclass `CConverter.__init__`. Instead, you should write a `converter_init()` function. `converter_init()` always accepts a `self` parameter; after that, all additional parameters *must* be keyword-only. Any arguments passed in to the converter in Argument Clinic will be passed along to your `converter_init()`.

There are some additional members of `CConverter` you may wish to specify in your subclass. Here's the current list :

- type** The C type to use for this variable. `type` should be a Python string specifying the type, e.g. `int`. If this is a pointer type, the type string should end with `' * '`.
- default** The Python default value for this parameter, as a Python value. Or the magic value `unspecified` if there is no default.
- py\_default** `default` as it should appear in Python code, as a string. Or `None` if there is no default.
- c\_default** `default` as it should appear in C code, as a string. Or `None` if there is no default.
- c\_ignored\_default** The default value used to initialize the C variable when there is no default, but not specifying a default may result in an "uninitialized variable" warning. This can easily happen when using option groups—although properly-written code will never actually use this value, the variable does get passed in to the impl, and the C compiler will complain about the "use" of the uninitialized value. This value should always be a non-empty string.

**converter** The name of the C converter function, as a string.

**impl\_by\_reference** A boolean value. If true, Argument Clinic will add a & in front of the name of the variable when passing it into the impl function.

**parse\_by\_reference** A boolean value. If true, Argument Clinic will add a & in front of the name of the variable when passing it into `PyArg_ParseTuple()`.

Here's the simplest example of a custom converter, from `Modules/zlibmodule.c`:

```
/*[python input]

class ssize_t_converter(CConverter):
    type = 'Py_ssize_t'
    converter = 'ssize_t_converter'

[python start generated code]*/
/*[python end generated code: output=da39a3ee5e6b4b0d input=35521e4e733823c7]*/
```

This block adds a converter to Argument Clinic named `ssize_t`. Parameters declared as `ssize_t` will be declared as type `Py_ssize_t`, and will be parsed by the `'O&'` format unit, which will call the `ssize_t_converter` converter function. `ssize_t` variables automatically support default values.

More sophisticated custom converters can insert custom C code to handle initialization and cleanup. You can see more examples of custom converters in the CPython source tree; grep the C files for the string `CConverter`.

## 4.16 Writing a custom return converter

Writing a custom return converter is much like writing a custom converter. Except it's somewhat simpler, because return converters are themselves much simpler.

Return converters must subclass `CReturnConverter`. There are no examples yet of custom return converters, because they are not widely used yet. If you wish to write your own return converter, please read `Tools/clinic/clinic.py`, specifically the implementation of `CReturnConverter` and all its subclasses.

## 4.17 METH\_O and METH\_NOARGS

To convert a function using `METH_O`, make sure the function's single argument is using the `object` converter, and mark the arguments as positional-only:

```
/*[clinic input]
meth_o_sample

    argument: object
/
[clinic start generated code]*/
```

To convert a function using `METH_NOARGS`, just don't specify any arguments.

You can still use a self converter, a return converter, and specify a `type` argument to the object converter for `METH_O`.

## 4.18 tp\_new and tp\_init functions

You can convert `tp_new` and `tp_init` functions. Just name them `__new__` or `__init__` as appropriate. Notes :

- The function name generated for `__new__` doesn't end in `__new__` like it would by default. It's just the name of the class, converted into a valid C identifier.
- No `PyMethodDef #define` is generated for these functions.
- `__init__` functions return `int`, not `PyObject *`.
- Use the docstring as the class docstring.
- Although `__new__` and `__init__` functions must always accept both the `args` and `kwargs` objects, when converting you may specify any signature for these functions that you like. (If your function doesn't support key-words, the parsing function generated will throw an exception if it receives any.)

## 4.19 Changing and redirecting Clinic's output

It can be inconvenient to have Clinic's output interspersed with your conventional hand-edited C code. Luckily, Clinic is configurable : you can buffer up its output for printing later (or earlier!), or write its output to a separate file. You can also add a prefix or suffix to every line of Clinic's generated output.

While changing Clinic's output in this manner can be a boon to readability, it may result in Clinic code using types before they are defined, or your code attempting to use Clinic-generated code before it is defined. These problems can be easily solved by rearranging the declarations in your file, or moving where Clinic's generated code goes. (This is why the default behavior of Clinic is to output everything into the current block ; while many people consider this hampers readability, it will never require rearranging your code to fix definition-before-use problems.)

Let's start with defining some terminology :

**field** A field, in this context, is a subsection of Clinic's output. For example, the `#define` for the `PyMethodDef` structure is a field, called `methoddef_define`. Clinic has seven different fields it can output per function definition :

```
docstring_prototype
docstring_definition
methoddef_define
impl_prototype
parser_prototype
parser_definition
impl_definition
```

All the names are of the form "`<a>_<b>`", where "`<a>`" is the semantic object represented (the parsing function, the impl function, the docstring, or the methoddef structure) and "`<b>`" represents what kind of statement the field is. Field names that end in "`_prototype`" represent forward declarations of that thing, without the actual body/data of the thing; field names that end in "`_definition`" represent the actual definition of the thing, with the body/data of the thing. ("`methoddef`" is special, it's the only one that ends with "`_define`", representing that it's a preprocessor `#define`.)

**destination** A destination is a place Clinic can write output to. There are five built-in destinations :

**block** The default destination : printed in the output section of the current Clinic block.

**buffer** A text buffer where you can save text for later. Text sent here is appended to the end of any existing text. It's an error to have any text left in the buffer when Clinic finishes processing a file.

**file** A separate "clinic file" that will be created automatically by Clinic. The filename chosen for the file is `{basename}.clinic{extension}`, where `basename` and `extension` were assigned the output from `os.path.splitext()` run on the current file. (Example : the file destination for `_pickle.c` would be written to `_pickle.clinic.c`)

**Important : When using a file destination, you must check in the generated file !**

**two-pass** A buffer like `buffer`. However, a two-pass buffer can only be dumped once, and it prints out all text sent to it during all processing, even from Clinic blocks *after* the dumping point.

**suppress** The text is suppressed—thrown away.

Clinic defines five new directives that let you reconfigure its output.

The first new directive is `dump` :

```
dump <destination>
```

This dumps the current contents of the named destination into the output of the current block, and empties it. This only works with `buffer` and `two-pass` destinations.

The second new directive is `output`. The most basic form of `output` is like this :

```
output <field> <destination>
```

This tells Clinic to output *field* to *destination*. `output` also supports a special meta-destination, called `everything`, which tells Clinic to output *all* fields to that *destination*.

`output` has a number of other functions :

```
output push
output pop
output preset <preset>
```

`output push` and `output pop` allow you to push and pop configurations on an internal configuration stack, so that you can temporarily modify the output configuration, then easily restore the previous configuration. Simply push before your change to save the current configuration, then pop when you wish to restore the previous configuration.

`output preset` sets Clinic's output to one of several built-in preset configurations, as follows :

**block** Clinic's original starting configuration. Writes everything immediately after the input block.

Suppress the `parser_prototype` and `docstring_prototype`, write everything else to block.

**file** Designed to write everything to the "clinic file" that it can. You then `#include` this file near the top of your file. You may need to rearrange your file to make this work, though usually this just means creating forward declarations for various `typedef` and `PyObject` definitions.

Suppress the `parser_prototype` and `docstring_prototype`, write the `impl_definition` to block, and write everything else to file.

The default filename is "`{dirname}/clinic/{basename}.h`".

**buffer** Save up most of the output from Clinic, to be written into your file near the end. For Python files implementing modules or builtin types, it's recommended that you dump the buffer just above the static structures for your module or builtin type; these are normally very near the end. Using `buffer` may require even more editing than `file`, if your file has static `PyMethodDef` arrays defined in the middle of the file.

Suppress the `parser_prototype`, `impl_prototype`, and `docstring_prototype`, write the `impl_definition` to block, and write everything else to file.

**two-pass** Similar to the `buffer` preset, but writes forward declarations to the `two-pass` buffer, and definitions to the `buffer`. This is similar to the `buffer` preset, but may require less editing than `buffer`. Dump the `two-pass` buffer near the top of your file, and dump the `buffer` near the end just like you would when using the `buffer` preset.

Suppresses the `impl_prototype`, write the `impl_definition` to block, write `docstring_prototype`, `methoddef_define`, and `parser_prototype` to `two-pass`, write everything else to `buffer`.

**partial-buffer** Similar to the `buffer` preset, but writes more things to block, only writing the really big chunks of generated code to `buffer`. This avoids the definition-before-use problem of `buffer` completely, at the small cost of having slightly more stuff in the block's output. Dump the `buffer` near the end, just like you would when using the `buffer` preset.

Suppresses the `impl_prototype`, write the `docstring_definition` and `parser_definition` to `buffer`, write everything else to `block`.

The third new directive is `destination` :

```
destination <name> <command> [...]
```

This performs an operation on the destination named `name`.

There are two defined subcommands : `new` and `clear`.

The `new` subcommand works like this :

```
destination <name> new <type>
```

This creates a new destination with name `<name>` and type `<type>`.

There are five destination types :

**suppress** Throws the text away.

**block** Writes the text to the current block. This is what Clinic originally did.

**buffer** A simple text buffer, like the "buffer" builtin destination above.

**file** A text file. The file destination takes an extra argument, a template to use for building the filename, like so :

```
destination <name> new <type> <file_template>
```

The template can use three strings internally that will be replaced by bits of the filename :

**{path}** The full path to the file, including directory and full filename.

**{dirname}** The name of the directory the file is in.

**{basename}** Just the name of the file, not including the directory.

**{basename\_root}** Basename with the extension clipped off (everything up to but not including the last '.').

**{basename\_extension}** The last '.' and everything after it. If the basename does not contain a period, this will be the empty string.

If there are no periods in the filename, `{basename}` and `{filename}` are the same, and `{extension}` is empty. "{basename}{extension}" is always exactly the same as "{filename}".

**two-pass** A two-pass buffer, like the "two-pass" builtin destination above.

The `clear` subcommand works like this :

```
destination <name> clear
```

It removes all the accumulated text up to this point in the destination. (I don't know what you'd need this for, but I thought maybe it'd be useful while someone's experimenting.)

The fourth new directive is `set` :

```
set line_prefix "string"
set line_suffix "string"
```

`set` lets you set two internal variables in Clinic. `line_prefix` is a string that will be prepended to every line of Clinic's output; `line_suffix` is a string that will be appended to every line of Clinic's output.

Both of these support two format strings :

**{block comment start}** Turns into the string `/*`, the start-comment text sequence for C files.

**{block comment end}** Turns into the string `*/`, the end-comment text sequence for C files.

The final new directive is one you shouldn't need to use directly, called `preserve` :

```
preserve
```

This tells Clinic that the current contents of the output should be kept, unmodified. This is used internally by Clinic when dumping output into `file` files; wrapping it in a Clinic block lets Clinic use its existing checksum functionality to ensure the file was not modified by hand before it gets overwritten.

## 4.20 The `#ifdef` trick

If you're converting a function that isn't available on all platforms, there's a trick you can use to make life a little easier. The existing code probably looks like this :

```
#ifdef HAVE_FUNCTIONNAME
static module_functionname(...)
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

And then in the `PyMethodDef` structure at the bottom the existing code will have :

```
#ifdef HAVE_FUNCTIONNAME
{'functionname', ... },
#endif /* HAVE_FUNCTIONNAME */
```

In this scenario, you should enclose the body of your impl function inside the `#ifdef`, like so :

```
#ifdef HAVE_FUNCTIONNAME
/*[clinic input]
module.functionname
...
[clinic start generated code]*/
static module_functionname(...)
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

Then, remove those three lines from the `PyMethodDef` structure, replacing them with the macro Argument Clinic generated :

```
MODULE_FUNCTIONNAME_METHODDEF
```

(You can find the real name for this macro inside the generated code. Or you can calculate it yourself : it's the name of your function as defined on the first line of your block, but with periods changed to underscores, uppercased, and `"_METHODDEF"` added to the end.)

Perhaps you're wondering : what if `HAVE_FUNCTIONNAME` isn't defined? The `MODULE_FUNCTIONNAME_METHODDEF` macro won't be defined either !

Here's where Argument Clinic gets very clever. It actually detects that the Argument Clinic block might be deactivated by the `#ifdef`. When that happens, it generates a little extra code that looks like this :

```
#ifndef MODULE_FUNCTIONNAME_METHODDEF
#define MODULE_FUNCTIONNAME_METHODDEF
#endif /* !defined(MODULE_FUNCTIONNAME_METHODDEF) */
```

That means the macro always works. If the function is defined, this turns into the correct structure, including the trailing comma. If the function is undefined, this turns into nothing.

However, this causes one ticklish problem : where should Argument Clinic put this extra code when using the "block" output preset ? It can't go in the output block, because that could be deactivated by the `#ifdef`. (That's the whole point !)

In this situation, Argument Clinic writes the extra code to the "buffer" destination. This may mean that you get a complaint from Argument Clinic :

```
Warning in file "Modules/posixmodule.c" on line 12357:  
Destination buffer 'buffer' not empty at end of file, emptying.
```

When this happens, just open your file, find the `dump_buffer` block that Argument Clinic added to your file (it'll be at the very bottom), then move it above the `PyMethodDef` structure where that macro is used.

## 4.21 Using Argument Clinic in Python files

It's actually possible to use Argument Clinic to preprocess Python files. There's no point to using Argument Clinic blocks, of course, as the output wouldn't make any sense to the Python interpreter. But using Argument Clinic to run Python blocks lets you use Python as a Python preprocessor !

Since Python comments are different from C comments, Argument Clinic blocks embedded in Python files look slightly different. They look like this :

```
/*[python input]  
#print("def foo(): pass")  
#[python start generated code]*/  
def foo(): pass  
/*[python checksum:...]*/
```



## Index

### P

Python Enhancement Proposals

PEP 8, [12](#)