Argument Clinic Cómo Hacerlo

Versión 3.11.4

Guido van Rossum and the Python development team

agosto 24, 2023

Python Software Foundation
Email: docs@python.org

Índice general

autor Larry Hastings

Source code: Tools/clinic/clinic.py.

Resumen

Argument Clinic is a preprocessor for CPython C files. It was introduced in Python 3.4 with **PEP 436**, in order to provide introspection signatures, and to generate performant and tailor-made boilerplate code for argument parsing in CPython builtins, module level functions, and class methods. This document is divided in four major sections:

- *Background* talks about the basic concepts and goals of Argument Clinic.
- *Reference* describes the command-line interface and Argument Clinic terminology.
- *Tutorial* guides you through all the steps required to adapt an existing C function to Argument Clinic.
- *How-to guides* details how to handle specific tasks.

Nota: Argument Clinic is considered internal-only for CPython. Its use is not supported for files outside CPython, and no guarantees are made regarding backwards compatibility for future versions. In other words: if you maintain an external C extension for CPython, you're welcome to experiment with Argument Clinic in your own code. But the version of Argument Clinic that ships with the next version of CPython *could* be totally incompatible and break all your code.

1 Background

1.1 Basic concepts

When Argument Clinic is run on a file, either via the *Command-line interface* or via `make clinic`, it will scan over the input files looking for *start lines*:

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

When it finds one, it reads everything up to the *end line*:

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

Everything in between these two lines is Argument Clinic *input*. When Argument Clinic parses input, it generates *output*. The output is rewritten into the C file immediately after the input, followed by a *checksum line*. All of these lines, including the *start line* and *checksum line*, are collectively called an Argument Clinic *block*:

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

If you run Argument Clinic on the same file a second time, Argument Clinic will discard the old *output* and write out the new output with a fresh *checksum line*. If the *input* hasn't changed, the output won't change either.

Nota: You should never modify the output of an Argument Clinic block, as any change will be lost in future Argument Clinic runs; Argument Clinic will detect an output checksum mismatch and regenerate the correct output. If you are not happy with the generated output, you should instead change the input until it produces the output you want.

2 Reference

2.1 Terminology

start line The line `/*[clinic input]`. This line marks the beginning of Argument Clinic input. Note that the *start line* opens a C block comment.

end line The line `[clinic start generated code]*/`. The *end line* marks the _end_ of Argument Clinic *input*, but at the same time marks the _start_ of Argument Clinic *output*, thus the text *«clinic start start generated code»* Note that the *end line* closes the C block comment opened by the *start line*.

checksum A hash to distinguish unique *inputs* and *outputs*.

checksum line A line that looks like `/*[clinic end generated code: ...]*/`. The three dots will be replaced by a *checksum* generated from the *input*, and a *checksum* generated from the *output*. The checksum line marks the end of Argument Clinic generated code, and is used by Argument Clinic to determine if it needs to regenerate output.

input The text between the *start line* and the *end line*. Note that the start and end lines open and close a C block comment; the *input* is thus a part of that same C block comment.

output The text between the *end line* and the *checksum line*.

block All text from the *start line* to the *checksum line* inclusively.

2.2 Command-line interface

The Argument Clinic CLI (Command-Line Interface) is typically used to process a single source file, like this:

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

The CLI supports the following options:

-h, --help
Print CLI usage.

-f, --force
Force output regeneration.

-o, --output OUTPUT
Redirect file output to OUTPUT

-v, --verbose
Enable verbose mode.

--converters
Print a list of all supported converters and return converters.

--make
Walk *--srcdir* to run over all relevant files.

--srcdir SRCDIR
The directory tree to walk in *--make* mode.

FILE ...
The list of files to process.

2.3 Classes for extending Argument Clinic

class clinic.**CConverter**

The base class for all converters. See *How to write a custom converter* for how to subclass this class.

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

El valor predeterminado de Python para este parámetro, como un valor de Python. O el valor mágico `unspecified` si no hay ningún valor predeterminado.

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

El valor por defecto utilizado para inicializar la variable C cuando no hay un valor por defecto, pero no especificar un valor por defecto puede dar lugar a una advertencia de «variable no inicializada». Esto puede ocurrir fácilmente cuando se utilizan grupos de opciones—aunque un código bien escrito nunca utilizará este valor, la variable se pasa a la impl, y el compilador de C se quejará del «uso» del valor no inicializado. Este valor debe ser siempre una cadena no vacía.

converter

El nombre de la función de conversión de C, como una cadena de caracteres.

impl_by_reference

Un valor booleano. Si es verdadero, Argument Clinic agregará un `&` delante del nombre de la variable al pasarlo a la función impl.

parse_by_reference

Un valor booleano. Si es verdadero, Argument Clinic agregará un `&` delante del nombre de la variable al pasarlo a `PyArg_ParseTuple()`.

3 Tutorial

The best way to get a sense of how Argument Clinic works is to convert a function to work with it. Here, then, are the bare minimum steps you'd need to follow to convert a function to work with Argument Clinic. Note that for code you plan to check in to CPython, you really should take the conversion farther, using some of the *advanced concepts* you'll see later on in the document, like *How to use return converters* and *How to use the «self converter»*. But we'll keep it simple for this walkthrough so you can learn.

First, make sure you're working with a freshly updated checkout of the CPython trunk.

Next, find a Python builtin that calls either `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()`, and hasn't been converted to work with Argument Clinic yet. For this tutorial, we'll be using `_pickle.Pickler.dump`.

If the call to the `PyArg_Parse*()` function uses any of the following format units…:

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

… or if it has multiple calls to `PyArg_ParseTuple()`, you should choose a different function. (See *How to use advanced converters* for those scenarios.)

Also, if the function has multiple calls to `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()` where it supports different types for the same argument, or if the function uses something besides `PyArg_Parse*()` functions to parse its arguments, it probably isn't suitable for conversion to Argument Clinic. Argument Clinic doesn't support generic functions or polymorphic parameters.

Next, add the following boilerplate above the function, creating our input block:

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

Cut the docstring and paste it in between the `[clinic]` lines, removing all the junk that makes it a properly quoted C string. When you're done you should have just the text, based at the left margin, with no line wider than 80 characters. Argument Clinic will preserve indents inside the docstring.

If the old docstring had a first line that looked like a function signature, throw that line away; The docstring doesn't need it anymore — when you use `help()` on your builtin in the future, the first line will be built automatically based on the function's signature.

Example docstring summary line:

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

If your docstring doesn't have a «summary» line, Argument Clinic will complain, so let's make sure it has one. The «summary» line should be a paragraph consisting of a single 80-column line at the beginning of the docstring. (See **PEP 257** regarding docstring conventions.)

Our example docstring consists solely of a summary line, so the sample code doesn't have to change for this step.

Now, above the docstring, enter the name of the function, followed by a blank line. This should be the Python name of the function, and should be the full dotted path to the function — it should start with the name of the module, include any sub-modules, and if the function is a method on a class it should include the class name too.

In our example, `_pickle` is the module, `Pickler` is the class, and `dump()` is the method, so the name becomes `_pickle.Pickler.dump()`:

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

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

If this is the first time that module or class has been used with Argument Clinic in this C file, you must declare the module and/or class. Proper Argument Clinic hygiene prefers declaring these in a separate block somewhere near the top of the C file, in the same way that include files and statics go at the top. In our sample code we'll just show the two blocks next to each other.

El nombre de la clase y el módulo debe ser el mismo que el visto por Python. Compruebe el nombre definido en `PyModuleDef` o `PyTypeObject` según corresponda.

When you declare a class, you must also specify two aspects of its type in C: the type declaration you'd use for a pointer to an instance of this class, and a pointer to the `PyTypeObject` for this class:

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

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

(continué en la próxima página)

(proviene de la página anterior)

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

Declare each of the parameters to the function. Each parameter should get its own line. All the parameter lines should be indented from the function name and the docstring. The general form of these parameter lines is as follows:

```
name_of_parameter: converter
```

Si el parámetro tiene un valor predeterminado, agréguelo después del convertidor:

```
name_of_parameter: converter = default_value
```

Argument Clinic's support for «default values» is quite sophisticated; see *How to assign default values to parameter* for more information.

Next, add a blank line below the parameters.

What's a «converter»? It establishes both the type of the variable used in C, and the method to convert the Python value into a C value at runtime. For now you're going to use what's called a «legacy converter» — a convenience syntax intended to make porting old code into Argument Clinic easier.

For each parameter, copy the «format unit» for that parameter from the `PyArg_Parse()` format argument and specify *that* as its converter, as a quoted string. The «format unit» is the formal name for the one-to-three character substring of the *format* parameter that tells the argument parsing function what the type of the variable is and how to convert it. For more on format units please see arg-parsing.

Para unidades de formato de caracteres múltiples como `z#`, use la cadena completa de dos o tres caracteres.

Muestra:

```
/*[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]*/
```

If your function has `|` in the format string, meaning some parameters have default values, you can ignore it. Argument Clinic infers which parameters are optional based on whether or not they have default values.

Si su función tiene `$` en la cadena de caracteres de formato, lo que significa que toma argumentos de solo palabras clave, especifique `*` en una línea antes del primer argumento de solo palabras clave, con la misma indentación que las líneas de parámetros.

`_pickle.Pickler.dump()` has neither, so our sample is unchanged.

Next, if the existing C function calls `PyArg_ParseTuple()` (as opposed to `PyArg_ParseTupleAndKeywords()`), then all its arguments are positional-only.

To mark parameters as positional-only in Argument Clinic, add a `/` on a line by itself after the last positional-only parameter, indented the same as the parameter lines.

Muestra:

```
/*[clinic input]
module _pickle
```

(continué en la próxima página)

(proviene de la página anterior)

```
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]*/
```

It can be helpful to write a per-parameter docstring for each parameter. Since per-parameter docstrings are optional, you can skip this step if you prefer.

Nevertheless, here's how to add a per-parameter docstring. The first line of the per-parameter docstring must be indented further than the parameter definition. The left margin of this first line establishes the left margin for the whole per-parameter docstring; all the text you write will be outdented by this amount. You can write as much text as you like, across multiple lines if you wish.

Muestra:

```
/*[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]*/
```

Save and close the file, then run `Tools/clinic/clinic.py` on it. With luck everything worked—your block now has output, and a `.c.h` file has been generated! Reload the file in your text editor to see the 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]*/

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

Obviously, if Argument Clinic didn't produce any output, it's because it found an error in your input. Keep fixing your errors and retrying until Argument Clinic processes your file without complaint.

For readability, most of the glue code has been generated to a `.c.h` file. You'll need to include that in your original `.c` file, typically right after the clinic module block:

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

Vuelva a verificar que el código de análisis de argumentos generado por Argument Clinic se ve básicamente igual al código existente.

Primero, asegúrese de que ambos lugares usen la misma función de análisis de argumentos. El código existente debe llamar a `PyArg_ParseTuple()` o `PyArg_ParseTupleAndKeywords()`; asegúrese de que el código generado por Argument Clinic llame a la misma *exacta* función.

Second, the format string passed in to `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()` should be *exactly* the same as the hand-written one in the existing function, up to the colon or semi-colon.

Argument Clinic always generates its format strings with a `:` followed by the name of the function. If the existing code's format string ends with `;`, to provide usage help, this change is harmless — don't worry about it.

Third, for parameters whose format units require two arguments, like a length variable, an encoding string, or a pointer to a conversion function, ensure that the second argument is *exactly* the same between the two invocations.

Fourth, inside the output portion of the block, you'll find a preprocessor macro defining the appropriate static `PyMethodDef` structure for this builtin:

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

This static structure should be *exactly* the same as the existing static `PyMethodDef` structure for this builtin.

Si alguno de estos elementos difiere *de alguna manera*, ajuste la especificación de la función de Argument Clinic y vuelva a ejecutar `Tools/clinic/clinic.py` hasta que *sean* iguales.

Observe que la última línea de su salida es la declaración de su función «impl». Aquí es donde va la implementación incorporada. Elimine el prototipo existente de la función que está modificando, pero deje la llave de apertura. Ahora elimine su código de análisis de argumentos y las declaraciones de todas las variables en las que vierte los argumentos. Observe cómo los argumentos de Python ahora son argumentos para esta función implícita; si la implementación usó nombres diferentes para estas variables, corríjalo.

Let's reiterate, just because it's kind of weird. Your code should now look like this:

```
static return_type
your_function_impl(...)
/*[clinic end generated code: input=..., output=...]*/
{
...
```

Argument Clinic generated the checksum line and the function prototype just above it. You should write the opening and closing curly braces for the function, and the implementation inside.

Muestra:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/
/*[clinic end generated code: checksum=da39a3ee5e6b4b0d3255bfef95601890afd80709]*/

/*[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"
```

(continué en la próxima página)

(proviene de la página anterior)

```
...
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;
    }

    ...
```

Remember the macro with the `PyMethodDef` structure for this function? Find the existing `PyMethodDef` structure for this function and replace it with a reference to the macro. If the builtin is at module scope, this will probably be very near the end of the file; if the builtin is a class method, this will probably be below but relatively near to the implementation.

Note that the body of the macro contains a trailing comma; when you replace the existing static `PyMethodDef` structure with the macro, *don't* add a comma to the end.

Muestra:

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

Finally, compile, then run the relevant portions of the regression-test suite. This change should not introduce any new compile-time warnings or errors, and there should be no externally visible change to Python's behavior, except for one difference: `inspect.signature()` run on your function should now provide a valid signature!

¡Felicitaciones, ha adaptado su primera función para trabajar con Argument Clinic!

4 How-to guides

4.1 How to rename C functions and variables generated by Argument Clinic

Argument Clinic nombra automáticamente las funciones que genera para usted. Ocasionalmente, esto puede causar un problema, si el nombre generado choca con el nombre de una función C existente. Hay una solución sencilla: anule los nombres utilizados para las funciones de C. Simplemente agregue la palabra clave `"as"` a la línea de declaración de su función, seguida del nombre de la función que desea usar. Argument Clinic usará ese nombre de función para la función base (generada), luego agregará `"_impl"` al final y lo usará para el nombre de la función impl.

For example, if we wanted to rename the C function names generated for `pickle.Pickler.dump()`, it'd look like this:

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

(continué en la próxima página)

(proviene de la página anterior)

```
...
```

The base function would now be named `pickler_dumper()`, and the impl function would now be named `pickler_dumper_impl()`.

De manera similar, es posible que tenga un problema en el que desee asignar un nombre específico de Python a un parámetro, pero ese nombre puede ser inconveniente en C. Argument Clinic le permite asignar nombres diferentes a un parámetro en Python y en C, usando el mismo `"as"` como sintaxis:

```
/*[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`.

You can use this to rename the *self* parameter too!

4.2 How to convert functions using `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).

Actualmente, el código generado usará `PyArg_ParseTuple()`, pero esto cambiará pronto.

4.3 How to use optional groups

Algunas funciones heredadas tienen un enfoque complicado para analizar sus argumentos: cuentan el número de argumentos posicionales, luego usan una instrucción `switch` para llamar a una de varias llamadas diferentes `PyArg_ParseTuple()` dependiendo de cuántos argumentos posicionales existen. (Estas funciones no pueden aceptar argumentos de solo palabras clave). Este enfoque se usó para simular argumentos opcionales antes de que se creara `PyArg_ParseTupleAndKeywords()`.

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.)

En cualquier caso, el objetivo de Argument Clinic es admitir el análisis de argumentos para todas las incorporaciones CPython existentes sin cambiar su semántica. Por lo tanto, Argument Clinic admite este enfoque alternativo de análisis, utilizando lo que se denominan *grupos opcionales*. Los grupos opcionales son grupos de argumentos que deben pasarse todos juntos. Pueden estar a la izquierda o la derecha de los argumentos requeridos. *Solo* se pueden usar con parámetros de solo posición.

Nota: Los grupos opcionales *solo* están pensados para su uso al convertir funciones que realizan múltiples llamadas a `PyArg_ParseTuple()`! Las funciones que usan *cualquier* otro enfoque para analizar argumentos deben *casi*

nunca convertirse a Argument Clinic usando grupos opcionales. Las funciones que utilizan grupos opcionales actualmente no pueden tener firmas precisas en Python, porque Python simplemente no comprende el concepto. Evite el uso de grupos opcionales siempre que sea posible.

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.
    ]
    /

...
```

Notas:

- Para cada grupo opcional, se pasará un parámetro adicional a la función *impl* que representa al grupo. El parámetro será un int llamado `grupo_{direction}_{number}`, donde `{direction}` es `right` o `left` dependiendo de si el grupo está antes o después los parámetros requeridos, y `{number}` es un número que aumenta monótonamente (comenzando en 1) que indica qué tan lejos está el grupo de los parámetros requeridos. Cuando se llama a impl, este parámetro se establecerá en cero si este grupo no se usó, y se establecerá en un valor distinto de cero si se usó este grupo. (Por usado o no usado, me refiero a si los parámetros recibieron argumentos en esta invocación).
- Si no hay argumentos requeridos, los grupos opcionales se comportarán como si estuvieran a la derecha de los argumentos requeridos.
- En el caso de ambigüedad, el código de análisis de argumentos favorece los parámetros de la izquierda (antes de los parámetros requeridos).
- Los grupos opcionales solo pueden contener parámetros posicionales.
- Los grupos opcionales son *solo* destinados al código heredado. No utilice grupos opcionales para el código nuevo.

4.4 How to use real Argument Clinic converters, instead of «legacy converters»

Para ahorrar tiempo y minimizar cuánto necesita aprender para lograr su primer puerto a Argument Clinic, el tutorial anterior le indica que use «convertidores heredados». Los «convertidores heredados» son una conveniencia, diseñados explícitamente para facilitar la migración del código existente a Argument Clinic. Y para ser claros, su uso es aceptable al portar código para Python 3.4.

Sin embargo, a largo plazo probablemente queramos que todos nuestros bloques utilicen la sintaxis real de Argument Clinic para los convertidores. ¿Por qué? Un par de razones:

- Los convertidores adecuados son mucho más fáciles de leer y más claros en su intención.
- Hay algunas unidades de formato que no se admiten como «convertidores heredados», porque requieren argumentos y la sintaxis del convertidor heredado no admite la especificación de argumentos.
- En el futuro, es posible que tengamos una nueva biblioteca de análisis de argumentos que no esté restringida a lo que `PyArg_ParseTuple()` admite; esta flexibilidad no estará disponible para los parámetros que utilizan convertidores heredados.

Por lo tanto, si no le importa un poco de esfuerzo adicional, utilice los convertidores normales en lugar de los convertidores heredados.

En pocas palabras, la sintaxis de los convertidores de Argument Clinic (no heredados) parece una llamada a una función de Python. Sin embargo, si no hay argumentos explícitos para la función (todas las funciones toman sus valores predeterminados), puede omitir los paréntesis. Por tanto, `bool` y `bool()` son exactamente los mismos convertidores.

Todos los argumentos para los convertidores de Argument Clinic son solo de palabras clave. Todos los convertidores de Argument Clinic aceptan los siguientes argumentos:

c_default El valor predeterminado para este parámetro cuando se define en C. Específicamente, será el inicializador de la variable declarada en la «función de análisis». Consulte *la sección sobre valores predeterminados* para saber cómo usar esto. Especificado como una cadena de caracteres.

annotation El valor de anotación para este parámetro. Actualmente no es compatible, porque **PEP 8** exige que la biblioteca de Python no use anotaciones.

Además, algunos convertidores aceptan argumentos adicionales. Aquí hay una lista de estos argumentos, junto con sus significados:

accept Un conjunto de tipos de Python (y posiblemente pseudo-tipos); esto restringe el argumento permitido de Python a valores de estos tipos. (Esta no es una infraestructura de propósito general; por regla general, solo admite listas específicas de tipos como se muestra en la tabla de convertidores heredados).

Para aceptar `None`, agregue `NoneType` a este conjunto.

bitwise Solo se admite para enteros sin signo. El valor entero nativo de este argumento de Python se escribirá en el parámetro sin ninguna verificación de rango, incluso para valores negativos.

converter Solo compatible con el convertidor de `objetos`. Especifica el nombre de una «función de conversión» C para convertir este objeto en un tipo nativo.

encoding Solo compatible con cadenas de caracteres. Especifica la codificación que se utilizará al convertir esta cadena de un valor Python str (Unicode) en un valor `char *` de C.

subclass_of Solo compatible con el convertidor de `objetos`. Requiere que el valor de Python sea una subclase de un tipo de Python, como se expresa en C.

type Solo compatible con los convertidores de `object` y `self`. Especifica el tipo C que se utilizará para declarar la variable. El valor predeterminado es `"PyObject *"`.

zeroes Solo compatible con cadenas. Si es verdadero, se permiten bytes NUL incrustados (`'\\0'`) dentro del valor. La longitud de la cadena se pasará a la función impl, justo después del parámetro de cadena, como un parámetro llamado `<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.)

A continuación se muestra una tabla que muestra el mapeo de convertidores heredados en convertidores de Argument Clinic reales. A la izquierda está el convertidor heredado, a la derecha está el texto con el que lo reemplazaría.

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

Como ejemplo, aquí está nuestra muestra `pickle.Pickler.dump` usando el convertidor adecuado:

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

    obj: object
```

(continué en la próxima página)

(proviene de la página anterior)

```
        The object to be pickled.
    /

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

Una ventaja de los convertidores reales es que son más flexibles que los convertidores heredados. Por ejemplo, el convertidor `unsigned_int` (y todos los convertidores `unsigned_`) se pueden especificar sin `bitwise=True`. Su comportamiento predeterminado realiza una verificación de rango en el valor y no aceptarán números negativos. ¡No puedes hacer eso con un convertidor heredado!

Argument Clinic le mostrará todos los convertidores que tiene disponibles. Para cada convertidor, le mostrará todos los parámetros que acepta, junto con el valor predeterminado para cada parámetro. Simplemente ejecute `Tools/clinic/clinic.py --converters` para ver la lista completa.

4.5 How to use the `Py_buffer` converter

Cuando se utiliza el convertidor `Py_buffer` (o los convertidores heredados `'s*'`, `'w*'`, `'*y'` o `'z*'`), *no* debes llamar a `PyBuffer_Release()` en el búfer provisto. Argument Clinic genera código que lo hace por usted (en la función de análisis).

4.6 How to use advanced converters

¿Recuerda esas unidades de formato que omitió por primera vez porque eran avanzadas? Aquí le mostramos cómo manejarlas también.

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 to that call. 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`.

4.7 How to assign default values to parameter

Los valores predeterminados de los parámetros pueden ser cualquiera de varios valores. En su forma más simple, pueden ser literales string, int o float:

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

También pueden usar cualquiera de las constantes incorporadas de Python:

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

También hay soporte especial para un valor predeterminado de `NULL` y para expresiones simples, documentadas en las siguientes secciones.

El valor predeterminado `NULL`

Para los parámetros de cadena de caracteres y objeto, puede establecerlos en `None` para indicar que no hay ningún valor predeterminado. Sin embargo, eso significa que la variable C se inicializará en `Py_None`. Por conveniencia, hay un valor especial llamado `NULL` solo por esta razón: desde la perspectiva de Python se comporta como un valor predeterminado de `None`, pero la variable C se inicializa con `NULL`.

Valores predeterminados simbólicos

El valor predeterminado que proporcione para un parámetro no puede ser una expresión arbitraria. Actualmente, lo siguiente se admite explícitamente:

- Constantes numéricas (enteros y flotantes)
- Constantes de cadena de caracteres
- `True`, `False`, y `None`
- Simple symbolic constants like `sys.maxsize`, which must start with the name of the module

(En el futuro, esto puede necesitar ser aún más elaborado, para permitir expresiones completas como `CONSTANT - 1`.)

Expressions as default values

El valor predeterminado de un parámetro puede ser más que un valor literal. Puede ser una expresión completa, utilizando operadores matemáticos y buscando atributos en objetos. Sin embargo, este soporte no es exactamente simple, debido a una semántica no obvia.

Considere el siguiente ejemplo:

```
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
```

Otra complicación: Argument Clinic no puede saber de antemano si la expresión que proporciona es válida o no. Lo analiza para asegurarse de que parece legal, pero no puede *realmente* saberlo. ¡Debe tener mucho cuidado al usar expresiones para especificar valores que están garantizados para ser válidos en tiempo de ejecución!

Finalmente, dado que las expresiones deben ser representables como valores C estáticos, existen muchas restricciones sobre las expresiones legales. Aquí hay una lista de funciones de Python que no está autorizado a usar:

- Llamadas a funciones.
- Declaraciones if en línea (`3 if foo else 5`).
- Desempaque automático de secuencia (`*[1, 2, 3]`).
- Comprensiones de list/set/dict y expresiones generadoras.
- Literales tuple/list/set/dict.

4.8 How to use return converters

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, using `->` notation.

For example:

```
/*[clinic input]
add -> int

    a: int
    b: int
    /

[clinic start generated code]*/
```

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.

(Si utiliza tanto `"as"` *y* un convertidor de retorno para su función, el `"as"` debe aparecer antes del convertidor de retorno.)

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 (c:func:*PyErr_Occurred* returns a true value), then the generated code will propagate the error. Otherwise it will encode the value you return like normal.

Actualmente, Argument Clinic solo admite unos pocos convertidores de retorno:

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

None of these take parameters. For all of these, return `-1` to indicate error.

(También hay un convertidor experimental `NoneType`, que le permite retornar `Py_None` en caso de éxito o `NULL` en caso de falla, sin tener que incrementar el recuento de referencias en `Py_None`. seguro que agrega suficiente claridad para que valga la pena usarlo)

Para ver todos los convertidores retornados que admite Argument Clinic, junto con sus parámetros (si los hay), simplemente ejecute `Tools/clinic/clinic.py --converters` para ver la lista completa.

4.9 How to clone existing functions

Si tiene varias funciones que parecen similares, es posible que pueda utilizar la función «clone» de Clinic. Cuando clona una función existente, reutiliza:

- sus parámetros, incluyendo
  - sus nombres,
  - sus convertidores, con todos los parámetros,
  - sus valores predeterminados,
  - sus docstrings por parámetro,
  - su *kind* (ya sea solo posicional, posicional o por palabra clave, o solo por palabra clave), y
- su convertidor de retorno.

Lo único que no se ha copiado de la función original es su docstring; la sintaxis le permite especificar un nuevo docstring.

Aquí está la sintaxis para clonar una función:

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

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

(Las funciones pueden estar en diferentes módulos o clases. Escribí `module.class` en la muestra solo para ilustrar que debe usar la ruta completa a *ambas* funciones.)

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

Además, la función desde la que está clonando debe haberse definido previamente en el archivo actual.

4.10 How to call Python code

El resto de los temas avanzados requieren que escriba código Python que vive dentro de su archivo C y modifica el estado de ejecución de Argument Clinic. Esto es simple: simplemente define un bloque de Python.

Un bloque Python utiliza diferentes líneas delimitadoras que un bloque de función de la Argument Clinic. Se parece a esto:

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

Todo el código dentro del bloque de Python se ejecuta en el momento en que se analiza. Todo el texto escrito en stdout dentro del bloque se redirige a la «salida» después del bloque.

Como ejemplo, aquí hay un bloque de Python que agrega una variable entera estática al código C:

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

4.11 How to use the «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.

¿Qué sentido tiene ? Esto le permite anular el tipo de `self` o darle un nombre predeterminado diferente.

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

  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]*/
```

4.12 How to use the «defining class» converter

Argument Clinic facilita el acceso a la clase definitoria de un método. Esto es útil para método de tipo heap (heap type) que necesitan obtener el estado del nivel del módulo. Utilice `PyType_FromModuleAndSpec()` para asociar un nuevo tipo de pila con un módulo. Ahora puede usar `PyType_GetModuleState()` en la clase de definición para obtener el estado del módulo, por ejemplo, de un método de módulo.

Example from Modules/zlibmodule.c. First, `defining_class` is added to the clinic input:

```
/*[clinic input]
zlib.Compress.compress

  cls: defining_class
  data: Py_buffer
    Binary data to be compressed.
  /
```

Después de ejecutar la herramienta Argument Clinic, se genera la siguiente firma de función:

```
/*[clinic start generated code]*/
static PyObject *
zlib_Compress_compress_impl(compobject *self, PyTypeObject *cls,
                            Py_buffer *data)
/*[clinic end generated code: output=6731b3f0ff357ca6 input=04d00f65ab01d260]*/
```

El siguiente código ahora puede usar `PyType_GetModuleState(cls)` para obtener el estado del módulo:

```
zlibstate *state = PyType_GetModuleState(cls);
```

Each method may only have one argument using this converter, and it must appear after `self`, or, if `self` is not used, as the first argument. The argument will be of type `PyTypeObject *`. The argument will not appear in the `__text_signature__`.

The `defining_class` converter is not compatible with `__init__()` and `__new__()` methods, which cannot use the `METH_METHOD` convention.

It is not possible to use `defining_class` with slot methods. In order to fetch the module state from such methods, use `PyType_GetModuleByDef()` to look up the module and then `PyModule_GetState()` to fetch the module state. Example from the `setattro` slot method in Modules/_threadmodule.c:

```
static int
local_setattro(localobject *self, PyObject *name, PyObject *v)
{
    PyObject *module = PyType_GetModuleByDef(Py_TYPE(self), &thread_module);
    thread_module_state *state = get_thread_state(module);
    ...
}
```

Vea también **PEP 573**.

4.13 How to write a custom converter

A converter is a Python class that inherits from *`CConverter`*. The main purpose of a custom converter, is for parameters parsed with the `O&` format unit — parsing such a parameter means calling a `PyArg_ParseTuple()` «converter function».

Your converter class should be named *`ConverterName`*`_converter`. By following this convention, your converter class will be automatically registered with Argument Clinic, with its *converter name* being the name of your converter class with the `_converter` suffix stripped off.

Instead of subclassing `CConverter.__init__()`, write a `converter_init()` method. `converter_init()` always accepts a *self* parameter. After *self*, all additional parameters **must** be keyword-only. Any arguments passed to the converter in Argument Clinic will be passed along to your `converter_init()` method. See *`CConverter`* for a list of members you may wish to specify in your subclass.

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

Los convertidores personalizados más sofisticados pueden insertar código C personalizado para manejar la inicialización y la limpieza. Puede ver más ejemplos de convertidores personalizados en el árbol de fuentes de CPython; grep los archivos C para la cadena `CConverter`.

4.14 How to write a custom return converter

Escribir un convertidor de retorno personalizado es muy parecido a escribir un convertidor personalizado. Excepto que es algo más simple, porque los convertidores de retorno son en sí mismos mucho más simples.

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.

4.15 How to convert `METH_O` and `METH_NOARGS` functions

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`.

4.16 How to convert `tp_new` and `tp_init` functions

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

- El nombre de la función generado para `__new__` no termina en `__new__` como lo haría por defecto. Es solo el nombre de la clase, convertido en un identificador C válido.
- No `PyMethodDef #define` is generated for these functions.
- funciones `__init__` retornan `int`, no `PyObject *`.
- Utilice docstring como la clase de documentación.
- Aunque las funciones `__new__` y `__init__` siempre deben aceptar tanto los objetos `args` como los `kwargs`, al realizar la conversión puede especificar cualquier firma para estas funciones que desee. (Si su función no admite palabras clave, la función de análisis generada lanzará una excepción si recibe alguna).

4.17 How to change and redirect Clinic's output

Puede ser inconveniente tener la salida de Clinic intercalada con su código C convencional editado a mano. Afortunadamente, Clinic es configurable: puede almacenar en búfer su salida para imprimir más tarde (¡o antes!), O escribir su salida en un archivo separado. También puede agregar un prefijo o sufijo a cada línea del resultado generado por Clinic.

Si bien cambiar la salida de la Clínica de esta manera puede ser una bendición para la legibilidad, puede resultar en que el código de la Clínica utilice tipos antes de que se definan, o que su código intente utilizar el código generado por la Clínica antes de que se defina. Estos problemas pueden resolverse fácilmente reorganizando las declaraciones en su archivo o moviendo el código generado por Clinic a donde va. (Esta es la razón por la que el comportamiento predeterminado de Clinic es enviar todo al bloque actual; aunque muchas personas consideran que esto dificulta la legibilidad, nunca será necesario reorganizar su código para solucionar problemas de definición antes de su uso).

Comencemos por definir alguna terminología:

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
```

Todos los nombres tienen la forma `"<a>_<b>"`, donde `"<a>"` es el objeto semántico representado (la función de análisis, la función impl, el docstring o la estructura methoddef) y `"<b>"` representa qué tipo de declaración es el campo. Los nombres de campo que terminan en `"_prototype"` representan declaraciones hacia adelante de esa cosa, sin el cuerpo/datos reales de la cosa; los nombres de campo que terminan en `"_definition"` representan la definición real de la cosa, con el cuerpo/datos de la cosa. (`"methoddef"` es especial, es el único que termina con `"_define"`, lo que representa que es un preprocesador #define).

destination Un destino es un lugar en el que la Clínica puede escribir resultados. Hay cinco destinos incorporados:

`block` El destino predeterminado: impreso en la sección de salida del bloque Clínico actual.

`buffer` Un búfer de texto donde puede guardar texto para más tarde. El texto enviado aquí se agrega al final de cualquier texto existente. Es un error dejar texto en el búfer cuando Clinic termina de procesar un archivo.

`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`.)

Importante: Al usar un destino **"file", *debe registrar* el archivo generado!

`two-pass` Un búfer como `buffer`. Sin embargo, un búfer de dos pasadas solo se puede volcar una vez, e imprime todo el texto que se le envía durante todo el procesamiento, incluso desde los bloques de la Clínica *después* del punto de descarga.

`suppress` El texto se suprime — se tira.

Clinic define cinco nuevas directivas que le permiten reconfigurar su salida.

La primera nueva directiva es `dump`:

```
dump <destination>
```

Esto vuelca el contenido actual del destino nombrado en la salida del bloque actual y lo vacía. Esto solo funciona con destinos de `búfer` y de `dos pasadas`.

La segunda nueva directiva es `output`. La forma más básica de `output` es así:

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

Esto le dice a la Clínica que envíe *field* a *destination*. `output` también admite un metadestino especial, llamado `everything`, que le dice a Clinic que envíe *todos* los campos a ese *destination*.

`output` tiene una serie de otras funciones:

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

`output push` y `output pop` le permiten agregar y quitar configuraciones en una pila de configuración interna, para que pueda modificar temporalmente la configuración de salida, y luego restaurar fácilmente la configuración

anterior. Simplemente presione antes de su cambio para guardar la configuración actual, luego haga estallar cuando desee restaurar la configuración anterior.

`output preset` configura la salida de Clinic en una de varias configuraciones preestablecidas incorporadas, de la siguiente manera:

`block` Configuración inicial original de la clínica. Escribe todo inmediatamente después del bloque de entrada.

Suprime el `parser_prototype` y `docstring_prototype`, escribe todo lo demás en `block`.

`file` Diseñado para escribir todo lo que pueda en el «archivo clínico». Luego, `#include` este archivo cerca de la parte superior de su archivo. Es posible que deba reorganizar su archivo para que esto funcione, aunque generalmente esto solo significa crear declaraciones hacia adelante para varias definiciones de `typedef` y `PyTypeObject`.

Suprima `parser_prototype` y `docstring_prototype`, escriba la `impl_definition` en `block` y escriba todo lo demás en `file`.

El nombre de archivo predeterminado es `"{dirname}/clinic/{basename}.h"`.

`buffer` Guarde la mayor parte del resultado de Clinic para escribirlo en su archivo cerca del final. Para los archivos Python que implementan módulos o tipos incorporado, se recomienda que descargue el búfer justo encima de las estructuras estáticas para su módulo o tipo incorporado; estos suelen estar muy cerca del final. El uso de `buffer` puede requerir incluso más edición que `file`, si su archivo tiene arreglos estáticos `PyMethodDef` definidos en el medio del archivo.

Suprima el `parser_prototype`, `impl_prototype` y `docstring_prototype`, escriba `impl_definition` en `block` y escriba todo lo demás en `file`.

`two-pass` Similar al ajuste preestablecido de `buffer`, pero escribe declaraciones hacia adelante en el búfer de `dos pasadas` y definiciones en el `buffer`. Esto es similar al ajuste preestablecido de `buffer`, pero puede requerir menos edición que `buffer`. Vierta el búfer de `dos pasadas` cerca de la parte superior de su archivo y descargue el `buffer` cerca del final como lo haría cuando usa el ajuste preestablecido de `buffer`.

Suprime el `impl_prototype`, escribe `impl_definition` en `block`, escribe `docstring_prototype`, `methoddef_define` y `parser_prototype` en `two-pass`, escribe todo lo demás en `buffer`.

`partial-buffer` Similar al ajuste preestablecido de `buffer`, pero escribe más cosas en `block`, solo escribe los trozos realmente grandes de código generado en `buffer`. Esto evita el problema de definición antes del uso de `buffer` por completo, con el pequeño costo de tener un poco más de material en la salida del bloque. Vierta el `buffer` cerca del final, tal como lo haría cuando usa el ajuste predeterminado de `buffer`.

Suprime el `impl_prototype`, escribe `docstring_definition` y `parser_definition` en `buffer`, escribe todo lo demás en `block`.

La tercera nueva directiva es `destino`:

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

Esto realiza una operación en el destino llamado `name`.

Hay dos subcomandos definidos: `new` y `clear`.

El subcomando `new` funciona así:

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

Esto crea un nuevo destino con el nombre `<nombre>` y escribe `<tipo>`.

Hay cinco tipos de destinos:

`suppress` Tira el texto.

block Escribe el texto en el bloque actual. Esto es lo que hizo Clinic originalmente.

buffer Un búfer de texto simple, como el destino incorporado «búfer» anterior.

file Un archivo de texto. El destino del archivo toma un argumento adicional, una plantilla para usar para construir el nombre de archivo, así:

destino <name> nuevo <type> <file_template>

La plantilla puede usar tres cadenas internamente que serán reemplazadas por bits del nombre del archivo:

{path} La ruta completa al archivo, incluido el directorio y el nombre de archivo completo.

{dirname} El nombre del directorio en el que se encuentra el archivo.

{basename} Solo el nombre del archivo, sin incluir el directorio.

{basename_root} Nombre de base con la extensión recortada (todo hasta pero sin incluir el último ".").

{basename_extension} El último "." y todo lo que sigue. Si el nombre base no contiene un punto, esta será la cadena de caracteres vacía.

Si no hay puntos en el nombre del archivo, {basename} y {filename} son iguales, y {extension} está vacía. «{basename}{extension}» es siempre exactamente igual que «{filename}». «

two-pass Un búfer de dos pasadas (*two-pass*), como el destino incorporado de «dos pasadas» anterior.

El subcomando `clear` funciona así:

```
destination <name> clear
```

Elimina todo el texto acumulado hasta este punto en el destino. (No sé para qué necesitarías esto, pero pensé que tal vez sería útil mientras alguien está experimentando).

La cuarta nueva directiva está `set`:

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

`set` le permite configurar dos variables internas en la Clínica. `line_prefix` es una cadena que se antepondrá a cada línea de salida de la Clínica; `line_suffix` es una cadena de caracteres que se agregará a cada línea de salida de la Clínica.

Ambos admiten dos cadenas de caracteres de formato:

{block comment start} Se convierte en la cadena de caracteres `/*`, la secuencia de texto de inicio de comentario para archivos C.

{block comment end} Se convierte en la cadena `*/`, la secuencia de texto del comentario final para los archivos C.

La nueva directiva final es una que no debería necesitar usar directamente, llamada `preserve`:

```
preserve
```

Esto le dice a Clinic que el contenido actual de la salida debe mantenerse, sin modificaciones. La Clínica lo usa internamente cuando se descarga la salida en archivos de `file`; envolverlo en un bloque Clinic permite que Clinic use su funcionalidad de suma de comprobación existente para garantizar que el archivo no se modificó a mano antes de sobrescribirlo.

4.18 How to use the `#ifdef` trick

Si está convirtiendo una función que no está disponible en todas las plataformas, hay un truco que puede usar para hacer la vida un poco más fácil. El código existente probablemente se ve así:

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

Y luego, en la estructura `PyMethodDef` en la parte inferior, el código existente tendrá:

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

En este escenario, debe encerrar el cuerpo de su función *impl* dentro de `#ifdef`, así:

```
#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
```

(Puede encontrar el nombre real de esta macro dentro del código generado. O puede calcularlo usted mismo: es el nombre de su función tal como se define en la primera línea de su bloque, pero con puntos cambiados a guiones bajos, mayúsculas y `"_METHODDEF"` agregado al final.)

Quizás se esté preguntando: ¿qué pasa si `HAVE_FUNCTIONNAME` no está definido? ¡La macro `MODULE_FUNCTIONNAME_METHODDEF` tampoco se definirá!

Aquí es donde Argument Clinic se vuelve muy inteligente. De hecho, detecta que el bloqueo de Argument Clinic podría estar desactivado por el `#ifdef`. Cuando eso sucede, genera un pequeño código adicional que se ve así:

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

Eso significa que la macro siempre funciona. Si la función está definida, se convierte en la estructura correcta, incluida la coma al final. Si la función no está definida, esto se convierte en nada.

Sin embargo, esto causa un problema delicado: ¿dónde debería poner Argument Clinic este código adicional cuando se usa el ajuste preestablecido de salida «bloque»? No puede entrar en el bloque de salida, porque podría desactivarse con `#ifdef`. (¡Ese es todo el punto!)

En esta situación, Argument Clinic escribe el código adicional en el destino del «búfer». Esto puede significar que recibe una queja de 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.

4.19 How to use Argument Clinic in Python files

De hecho, es posible utilizar Argument Clinic para preprocesar archivos Python. Por supuesto, no tiene sentido usar bloques de Argument Clinic, ya que la salida no tendría ningún sentido para el intérprete de Python. ¡Pero usar Argument Clinic para ejecutar bloques de Python le permite usar Python como un preprocesador de Python!

Dado que los comentarios de Python son diferentes de los comentarios de C, los bloques de Argument Clinic incrustados en archivos de Python tienen un aspecto ligeramente diferente. Se ven así:

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

4.20 How to override the generated signature

You can use the `@text_signature` directive to override the default generated signature in the docstring. This can be useful for complex signatures that Argument Clinic cannot handle. The `@text_signature` directive takes one argument: the custom signature as a string. The provided signature is copied verbatim to the generated docstring.

Example from Objects/codeobject.c:

```
/*[clinic input]
@text_signature "($self, /, **changes)"
code.replace
    *
    co_argcount: int(c_default="self->co_argcount") = unchanged
    co_posonlyargcount: int(c_default="self->co_posonlyargcount") = unchanged
    # etc ...

    Return a copy of the code object with new values for the specified fields.
[clinic start generated output]*/
```

The generated docstring ends up looking like this:

```
replace($self, /, **changes)
--

Return a copy of the code object with new values for the specified fields.
```

Índice de Módulos Python

C

Índice

No alfabético