How-To Argument Clinic

Versión 3.10.5

Guido van Rossum
and the Python development team

agosto 01, 2022

Python Software Foundation
Email: docs@python.org

Índice general

autor *Larry Hastings*

Resumen

Argument Clinic es un preprocesador para archivos CPython C. Su propósito es automatizar todo el texto estándar involucrado con la escritura de código de análisis de argumentos para «incorporados». Este documento le muestra cómo convertir su primera función C para que funcione con Argument Clinic y luego presenta algunos temas avanzados sobre el uso de Argument Clinic.

Actualmente, Argument Clinic se considera solo interno para CPython. Su uso no es compatible con archivos fuera de CPython y no se ofrecen garantías con respecto a la compatibilidad con versiones anteriores. En otras palabras: si mantiene una extensión C externa para CPython, puede experimentar con Argument Clinic en su propio código. Pero la versión de Argument Clinic que se envía con la próxima versión de CPython *podría* ser totalmente incompatible y romper todo su código.

1 Los objetivos del Argument Clinic

El objetivo principal de Argument Clinic es asumir la responsabilidad de todo el código de análisis de argumentos dentro de CPython. Esto significa que, cuando convierte una función para que funcione con Argument Clinic, esa función ya no debería realizar ninguno de sus propios análisis de argumentos; el código generado por Argument Clinic debería ser una «caja negra» para usted, donde CPython llama al top, y su código se llama en la parte inferior, con `PyObject *args` (y tal vez `PyObject *kwargs`) convertido mágicamente en las variables y tipos C que necesita.

Para que Argument Clinic logre su objetivo principal, debe ser fácil de usar. Actualmente, trabajar con la biblioteca de análisis de argumentos de CPython es una tarea ardua que requiere mantener información redundante en un número sorprendente de lugares. Cuando usa Argument Clinic, no tiene que repetirse.

Obviamente, si Argument Clinic no produjo ningún resultado, es porque encontró un error en su entrada. Siga corrigiendo sus errores y vuelva a intentarlo hasta que Argument Clinic procese su archivo sin quejas.

Además, Argument Clinic debe ser lo suficientemente flexible como para trabajar con cualquier enfoque de análisis de argumentos. Python tiene algunas funciones con algunos comportamientos de análisis muy extraños; el objetivo de Argument Clinic es apoyarlos a todos.

Finalmente, la motivación original de Argument Clinic era proporcionar «signaturas» de introspección para las incorporaciones de CPython. Solía ser, las funciones de consulta de introspección lanzarían una excepción si pasaba un archivo incorporado. ¡Con Argument Clinic, eso es cosa del pasado!

Una idea que debe tener en cuenta al trabajar con Argument Clinic: cuanta más información le dé, mejor será su trabajo. Argument Clinic es ciertamente relativamente simple en este momento. Pero a medida que evolucione, se volverá más sofisticado y debería poder hacer muchas cosas interesantes e inteligentes con toda la información que le proporcione.

2 Conceptos básicos y uso

Argument Clinic se envía con CPython; lo encontrará en `Tools/clinic/clinic.py`. Si ejecuta ese script, especificando un archivo C como argumento:

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

Argument Clinic escaneará el archivo buscando líneas que se vean exactamente así:

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

Cuando encuentra uno, lee todo hasta una línea que se ve exactamente así:

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

Todo lo que se encuentra entre estas dos líneas es entrada para Argument Clinic. Todas estas líneas, incluidas las líneas de comentarios iniciales y finales, se denominan colectivamente un «bloque» de Argument Clinic.

Cuando Argument Clinic analiza uno de estos bloques, genera una salida. Esta salida se reescribe en el archivo C inmediatamente después del bloque, seguida de un comentario que contiene una suma de comprobación. El bloque Argument Clinic ahora tiene este aspecto:

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

Si ejecuta Argument Clinic en el mismo archivo por segunda vez, Argument Clinic descartará la salida anterior y escribirá la nueva salida con una nueva línea de suma de comprobación. Sin embargo, si la entrada no ha cambiado, la salida tampoco cambiará.

Nunca debe modificar la parte de salida de un bloque de Argument Clinic. En su lugar, cambie la entrada hasta que produzca la salida que desea. (Ese es el propósito de la suma de comprobación: detectar si alguien cambió la salida, ya que estas ediciones se perderían la próxima vez que Argument Clinic escriba una salida nueva).

En aras de la claridad, esta es la terminología que usaremos con Argument Clinic:

- La primera línea del comentario (`/*[clinic input]`) es la *línea de inicio*.
- La última línea del comentario inicial (`[clinic start generated code]*/`) es la *línea final*.
- La última línea (`/*[clinic end generated code: checksum=...]*/`) es la *línea de suma de comprobación* (*chemsum line*).
- Entre la línea de inicio y la línea final está el *input*.
- Entre la línea final y la línea de suma de comprobación se encuentra la *output*.
- Todo el texto colectivamente, desde la línea de inicio hasta la línea de suma de verificación inclusive, es el *bloque*. (Un bloque que no ha sido procesado con éxito por Argument Clinic todavía no tiene salida o una línea de suma de verificación, pero aún se considera un bloque).

3 Convirtiendo su primera función

La mejor manera de tener una idea de cómo funciona Argument Clinic es convertir una función para que funcione con ella. Aquí, entonces, están los pasos mínimos que debe seguir para convertir una función para que funcione con Argument Clinic. Tenga en cuenta que para el código que planea registrar en CPython, realmente debería llevar la conversión más lejos, utilizando algunos de los conceptos avanzados que verá más adelante en el documento (como «convertidores de retorno» y «convertidores automáticos»). Pero lo haremos simple para este tutorial para que pueda aprender.

¡Vamos a sumergirnos!

0. Asegúrese de estar trabajando con una versión recién actualizada de CPython.
1. Busca un incorporado de Python que llame a `PyArg_ParseTuple()` o `PyArg_ParseTupleAndKeywords()`, y que aún no se haya convertido para funcionar con Argument Clinic. Para mi ejemplo, estoy usando `_pickle.Pickler.dump()`.

2. Si la llamada a la función `PyArg_Parse` usa cualquiera de las siguientes unidades de formato:

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

   o si tiene múltiples llamadas a `PyArg_ParseTuple()`, debes elegir una función diferente. Argument Clinic *sí* admite todos estos escenarios. Pero estos son temas avanzados; hagamos algo más simple para su primera función.

   Además, si la función tiene múltiples llamadas a `PyArg_ParseTuple()` o `PyArg_ParseTupleAndKeywords()` donde admite diferentes tipos para el mismo argumento, o si la función usa algo además de las funciones PyArg_Parse para analizar sus argumentos, probablemente no sea adecuado para la conversión a Argument Clinic. Argument Clinic no admite funciones genéricas ni parámetros polimórficos.

3. Agrega la siguiente plantilla sobre la función, creando nuestro bloque

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

4. Corta el docstring y lo pega entre las líneas `[clinic]`, eliminando toda la basura que la convierte en una cadena C entre comillas. Cuando haya terminado, debería tener solo el texto, basado en el margen izquierdo, sin una línea de más de 80 caracteres. (Argument Clinic conservará las sangrías dentro del docstring).

   Si el docstring antiguo tenía una primera línea que parecía una firma de función, elimine esa línea. (El docstring ya no la necesita; cuando use `help()` en su incorporado en el futuro, la primera línea se creará automáticamente en función de la firma de la función).

   Muestra:

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

5. Si su cadena de documentos no tiene una línea de «resumen», Argument Clinic se quejará. Así que asegurémonos de que tenga uno. La línea de «resumen» debe ser un párrafo que consta de una sola línea de 80 columnas al comienzo de la cadena de documentos.

   (Nuestro docstring de ejemplo consiste únicamente en una línea de resumen, por lo que el código de muestra no tiene que cambiar para este paso.)

6. Sobre el docstring, ingrese el nombre de la función, seguido de una línea en blanco. Este debería ser el nombre de Python de la función, y debería ser la ruta de puntos completa a la función; debería comenzar con el nombre del módulo, incluir cualquier submódulo y, si la función es un método en una clase, debe incluir el nombre de la clase también.

   Muestra:

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

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

7. Si es la primera vez que ese módulo o clase se utiliza con Argument Clinic en este archivo C, debe declarar el módulo o la clase. La higiene de la clínica de argumentos apropiados prefiere declararlos en un bloque separado en algún lugar cerca de la parte superior del archivo C, de la misma manera que los archivos de inclusión y las

estadísticas van en la parte superior. (En nuestro código de muestra, solo mostraremos los dos bloques uno al lado del otro).

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.

Cuando declaras una clase, también debes especificar dos aspectos de su tipo en C: la declaración de tipo que usarías para un puntero a una instancia de esta clase y un puntero a `PyTypeObject` para esto clase.

Muestra:

```
/*[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. Declare cada uno de los parámetros a la función. Cada parámetro debe tener su propia línea. Todas las líneas de parámetros deben tener sangría del nombre de la función y el docstring.

   La forma general de estas líneas de parámetros es la siguiente:

   ```
   name_of_parameter: converter
   ```

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

   ```
   name_of_parameter: converter = default_value
   ```

   El soporte de Argument Clinic para «valores predeterminados» es bastante sofisticado; por favor vea *la sección a continuación sobre valores predeterminados* para más información.

   Agrega una línea en blanco debajo de los parámetros.

   ¿Qué es un «convertidor»? Establece tanto el tipo de variable utilizada en C como el método para convertir el valor de Python en un valor de C en tiempo de ejecución. Por ahora, va a utilizar lo que se llama un «convertidor heredado», una sintaxis conveniente destinada a facilitar la migración del código antiguo a Argument Clinic.

   Para cada parámetro, copie la «unidad de formato» para ese parámetro del argumento de formato `PyArg_Parse()` y especifique *eso* como su convertidor, como una cadena entre comillas. («unidad de formato» es el nombre formal de la subcadena de caracteres de uno a tres caracteres del parámetro `format` que le dice a la función de análisis de argumentos cuál es el tipo de variable y cómo convertirla. Para más información sobre las unidades de formato por favor vea 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
   ```

(continué en la próxima página)

(proviene de la página anterior)

```
    obj: 'O'

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

9. Si su función tiene `|` en la cadena de formato, lo que significa que algunos parámetros tienen valores predeterminados, puede ignorarlo. Argument Clinic infiere qué parámetros son opcionales en función de si tienen o no valores predeterminados.

   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` no tiene ninguno, por lo que nuestro ejemplo no ha cambiado.)

10. Si la función C existente llama a `PyArg_ParseTuple()` (a diferencia de `PyArg_ParseTupleAndKeywords()`), entonces todos sus argumentos son solo posicionales.

    Para marcar todos los parámetros como solo posicionales en Argument Clinic, agregue un `/` en una línea después del último parámetro, con la misma sangría que las líneas de parámetros.

    Actualmente esto es todo o nada; o todos los parámetros son solo posicionales o ninguno de ellos lo es. (En el futuro, Argument Clinic puede relajar esta restricción).

    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]*/
```

11. Es útil escribir una cadena de documentos por parámetro para cada parámetro. Pero los docstrings por parámetro son opcionales; puede omitir este paso si lo prefiere.

    A continuación, se explica cómo agregar un docstring por parámetro. La primera línea del docstring por parámetro debe tener más sangría que la definición del parámetro. El margen izquierdo de esta primera línea establece el margen izquierdo para todo el docstring por parámetro; todo el texto que escriba se verá afectado por esta cantidad. Puede escribir todo el texto que desee, en varias líneas si lo desea.

    Muestra:

```
/*[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)

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

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

12. Guarde y cierre el archivo, luego ejecute `Tools/clinic/clinic.py` en él. ¡Con suerte, todo funcionó — su bloque ahora tiene salida y se ha generado un archivo `.c.h` ! Vuelva a abrir el archivo en su editor de texto para ver:

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

Obviamente, si Argument Clinic no produjo ningún resultado, es porque encontró un error en su entrada. Siga corrigiendo sus errores y vuelva a intentarlo hasta que Argument Clinic procese su archivo sin quejas.

Para facilitar la lectura, la mayor parte del código de pegamento se ha generado en un archivo `.c.h`. Deberá incluir eso en su archivo `.c` original, generalmente justo después del bloque del módulo de la clínica:

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

13. 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.

    En segundo lugar, la cadena de formato pasada a `PyArg_ParseTuple()` o `PyArg_ParseTupleAndKeywords()` debe ser *exactamente* la misma que la escrita a mano en la función existente, hasta los dos puntos o punto y coma.

    (Argument Clinic siempre genera sus cadenas de caracteres de formato con un `:` seguido del nombre de la función. Si la cadena de caracteres de formato del código existente termina con `;`, para proporcionar ayuda de uso, este cambio es inofensivo; no se preocupe)

    En tercer lugar, para los parámetros cuyas unidades de formato requieren dos argumentos (como una variable de longitud, una cadena de codificación o un puntero a una función de conversión), asegúrese de que el segundo argumento sea *exactamente* el mismo entre las dos invocaciones.

    En cuarto lugar, dentro de la parte de salida del bloque, encontrará una macro de preprocesador que define la estructura static `PyMethodDef` apropiada para este incorporado:

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

Esta estructura estática debe ser *exactamente* la misma que la estructura estática existente `PyMethodDef` para este incorporado.

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.

14. 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.

    Reiteremos, solo porque es un poco extraño. Su código ahora debería verse así:

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

    Argument Clinic generó la línea de suma de comprobación y el prototipo de función justo encima de ella. Debe escribir las llaves de apertura (y cierre) para la función y la implementación en el interior.

    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"
...
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;
    }
```

(continué en la próxima página)

(proviene de la página anterior)

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


    ...
```

15. ¿Recuerda la macro con la estructura `PyMethodDef` para esta función? Busque la estructura existente `PyMethodDef` para esta función y la reemplaza con una referencia a la macro. (Si el incorporado está en el alcance del módulo, esto probablemente estará muy cerca del final del archivo; si el incorporado es un método de clase, probablemente estará debajo pero relativamente cerca de la implementación).

    Tenga en cuenta que el cuerpo de la macro contiene una coma al final. Entonces, cuando reemplace la estructura static `PyMethodDef` existente con la macro, *no* agregue una coma al final.

    Muestra:

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

16. 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.

    Bueno, excepto por una diferencia: `inspect.signature()` ejecutar en su función ahora debería proporcionar una firma válida!

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

4 Temas avanzados

Ahora que ha tenido algo de experiencia trabajando con Argument Clinic, es hora de algunos temas avanzados.

4.1 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`
- Constantes simbólicas simples como `sys.maxsize`, que debe comenzar con el nombre del módulo

En caso de que tenga curiosidad, esto se implementa en `from_builtin()` en `Lib/inspect.py`.

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

4.2 Cambiar el nombre de las funciones y variables C generadas por 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.

Por ejemplo, si quisiéramos cambiar el nombre de las funciones de C generadas para `pickle.Pickler.dump`, se vería así:

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

...
```

La función base ahora se llamaría `pickler_dumper()`, y la función implícita ahora se llamaría `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
```

Aquí, el nombre usado en Python (en la firma y el arreglo de `keywords`) sería `file`, pero la variable C se llamaría `file_obj`.

¡También puede usar esto para cambiar el nombre del parámetro `self`!

4.3 Convirtiendo funciones usando PyArg_UnpackTuple

Para convertir una función que analiza sus argumentos con `PyArg_UnpackTuple()`, simplemente escribe todos los argumentos, especificando cada uno como un `object`. Puede especificar el argumento `type` para convertir el tipo según corresponda. Todos los argumentos deben estar marcados como solo posicionales (agregue un `/` en una línea después del último argumento).

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

4.4 Grupos opcionales

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()`.

Si bien las funciones que utilizan este enfoque a menudo se pueden convertir para usar `PyArg_ParseTupleAndKeywords()`, argumentos opcionales y valores predeterminados, no siempre es posible. Algunas de estas funciones heredadas tienen comportamientos `PyArg_ParseTupleAndKeywords()` no admite directamente. El ejemplo más obvio es la función incorporada `range()`, que tiene un argumento opcional en el lado *izquierdo* de su argumento requerido. Otro ejemplo es `curses.window.addch()`, que tiene un grupo de dos argumentos que siempre deben especificarse juntos. (Los argumentos se denominan `x` e `y`; si llama a la función pasando `x`, también debe pasar `y`, y si no pasa `x` tampoco puede pasar `y`.)

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.

Para especificar un grupo opcional, agregue un `[` en una línea antes de los parámetros que desea agrupar y un `]` en una línea después de estos parámetros. Como ejemplo, así es como `curses.window.addch` usa grupos opcionales para hacer que los primeros dos parámetros y el último parámetro sean opcionales:

```
/*[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.

4.5 Usar convertidores de Argument Clinic reales, en lugar de «convertidores heredados»

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`.

Tenga en cuenta que no todas las combinaciones posibles de argumentos funcionarán. Por lo general, estos argumentos se implementan mediante *unidades de formato* `PyArg_ParseTuple` específicas, con un comportamiento específico. Por ejemplo, actualmente no puede llamar a `unsigned_short` sin especificar también `bitwise=True`. Aunque es perfectamente razonable pensar que esto funcionaría, esta semántica no se asigna a ninguna unidad de formato existente. Entonces, Argument Clinic no lo admite. (O, al menos, todavía no).

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')`

continué en la próxima página

Tabla 1 – proviene de la página anterior

'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
        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.

4.6 Py_buffer

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).

4.7 Convertidores avanzados

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

El truco es que todas esas unidades de formato toman argumentos, ya sean funciones de conversión o tipos, o cadenas que especifican una codificación. (Pero los «convertidores heredados» no admiten argumentos. Por eso los omitimos para su primera función). El argumento que especificó para la unidad de formato ahora es un argumento para el convertidor; este argumento es `converter` (para `O&`), `subclass_of` (para `O!`) o `encoding` (para todas las unidades de formato que comienzan con `e`).

Al usar `subclass_of`, es posible que también desee usar el otro argumento personalizado para `object():type`, que le permite establecer el tipo que realmente se usa para el parámetro. Por ejemplo, si desea asegurarse de que el objeto es una subclase de `PyUnicode_Type`, probablemente desee utilizar el convertidor `object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')`.

Un posible problema con el uso de Argument Clinic: elimina cierta flexibilidad posible para las unidades de formato que comienzan con `e`. Al escribir una llamada `PyArg_Parse` a mano, teóricamente podrías decidir en tiempo de ejecución qué cadena de codificación pasar a `PyArg_ParseTuple()`. Pero ahora esta cadena debe estar codificada en tiempo de preprocesamiento de Argument-Clinic. Esta limitación es deliberada; hizo que el soporte de esta unidad de formato fuera mucho más fácil y puede permitir futuras optimizaciones. Esta restricción no parece irrazonable; el propio CPython siempre pasa cadenas de codificación estáticas codificadas para parámetros cuyas unidades de formato comienzan con `e`.

4.8 Valores predeterminados de los parámetros

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.

4.9 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`.

4.10 Expresiones especificadas como valores por defecto

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` puede tener diferentes valores en diferentes plataformas. Por lo tanto, Argument Clinic no puede simplemente evaluar esa expresión localmente y codificarla en C. Por lo tanto, almacena el valor predeterminado de tal manera que se evaluará en tiempo de ejecución, cuando el usuario solicite la firma de la función.

¿Qué espacio de nombres está disponible cuando se evalúa la expresión? Se evalúa en el contexto del módulo del que procede el incorporado. Entonces, si su módulo tiene un atributo llamado «`max_widgets`», simplemente puede usarlo:

```
foo: Py_ssize_t = max_widgets
```

Si el símbolo no se encuentra en el módulo actual, falla para buscar en `sys.modules`. Así es como puede encontrar `sys.maxsize`, por ejemplo. (Dado que no sabe de antemano qué módulos cargará el usuario en su intérprete, es mejor limitarse a los módulos que están precargados por el propio Python).

La evaluación de los valores predeterminados solo en tiempo de ejecución significa que Argument Clinic no puede calcular el valor predeterminado de C equivalente correcto. Entonces necesita decirlo explícitamente. Cuando usa una expresión, también debe especificar la expresión equivalente en C, usando el parámetro `c_default` para el convertidor:

```
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.

4.11 Usando un convertidor de retorno

De forma predeterminada, la función implícita Argument Clinic genera para usted retorna `PyObject *`. Pero su función C a menudo calcula algún tipo de C, luego lo convierte en el `PyObject *` en el último momento. Argument Clinic se encarga de convertir sus entradas de tipos de Python en tipos C nativos; ¿por qué no convertir su valor de retorno de un tipo C nativo en un tipo Python también?

Eso es lo que hace un «convertidor de retorno». Cambia su función *impl* para retornar algún tipo de C, luego agrega código a la función generada (no implícita) para manejar la conversión de ese valor en el `PyObject *` apropiado.

La sintaxis de los convertidores de retorno es similar a la de los convertidores de parámetros. Especifica el convertidor de retorno como si fuera una anotación de retorno en la función en sí. Los convertidores de retorno se comportan de la misma manera que los convertidores de parámetros; aceptan argumentos, todos los argumentos son solo palabras clave y, si no está cambiando ninguno de los argumentos predeterminados, puede omitir los paréntesis.

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

Hay una complicación adicional al usar convertidores de retorno: ¿cómo indica que se ha producido un error? Normalmente, una función retorna un puntero válido (no `NULL`) para el éxito y `NULL` para el error. Pero si usa un convertidor de retorno de enteros, todos los enteros son válidos. ¿Cómo puede Argument Clinic detectar un error? Su solución: cada convertidor de retorno busca implícitamente un valor especial que indica un error. Si retorna ese valor y se ha establecido un error (`PyErr_Occurred()` retorna un valor verdadero), el código generado propagará el error. De lo contrario, codificará el valor que retorna como de costumbre.

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

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

Ninguno de estos toma parámetros. Para los tres primeros, retorna -1 para indicar error. Para `DecodeFSDefault`, el tipo de retorno es `const char *`; retorna un puntero `NULL` para indicar un 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.

4.12 Clonando funciones existentes

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.

4.13 Llamando código Python

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:...]*/
```

4.14 Usando un «auto convertidor»

Argument Clinic agrega automáticamente un parámetro «self» para usted usando un convertidor predeterminado. Establece automáticamente el `tipo` de este parámetro en el «puntero a una instancia» que especificó cuando declaró el tipo. Sin embargo, puede anular el convertidor de Argument Clinic y especificar uno usted mismo. Simplemente agregue su propio parámetro `self` como el primer parámetro en un bloque y asegúrese de que su convertidor sea una instancia de `self_converter` o una subclase del mismo.

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

¿Cómo especifica el tipo personalizado al que desea transmitir `self`? Si solo tiene una o dos funciones con el mismo tipo para `self`, puede usar directamente el convertidor `self` existente de Argument Clinic, pasando el tipo que desea usar como parámetro de `type`:

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

Por otro lado, si tiene muchas funciones que usarán el mismo tipo para `self`, es mejor crear su propio convertidor, subclasificando `self_converter` pero sobrescribiendo el miembro " type":

```
/*[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 Usando un convertidor de «clase definitoria» (*defining class*)

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.

Ejemplo de `Modules/zlibmodule.c`. Primero, se agrega `definition_class` a la entrada de la clínica:

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

```

(continué en la próxima página)

(proviene de la página anterior)

```
  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);
```

Cada método solo puede tener un argumento usando este convertidor, y debe aparecer después de `self` o, si no se usa `self`, como primer argumento. El argumento será de tipo `PyTypeObject *`. El argumento no aparecerá en el `__text_signature__`.

El convertidor `definition_class` no es compatible con los métodos `__init__` y `__new__`, que no pueden usar la convención `METH_METHOD`.

No es posible usar `defining_class` con métodos de ranura (*slot*). Para obtener el estado del módulo de dichos métodos, use `_PyType_GetModuleByDef` para buscar el módulo y luego `PyModule_GetState()` para buscar el estado del módulo. Ejemplo del método de ranura `setattro` en `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**.

4.16 Escribiendo un convertidor personalizado

Como dijimos en la sección anterior… ¡puedes escribir tus propios convertidores! Un convertidor es simplemente una clase de Python que hereda de `CConverter`. El propósito principal de un convertidor personalizado es si tiene un parámetro que usa la unidad de formato `O&`; analizar este parámetro significa llamar a `PyArg_ParseTuple()` «función de conversión».

Su clase de convertidor debe llamarse `*something*_converter`. Si el nombre sigue esta convención, entonces su clase de convertidor se registrará automáticamente con Argument Clinic; su nombre será el nombre de su clase con el sufijo `_converter` eliminado. (Esto se logra con una metaclase).

No debe subclasificar `CConverter.__init__`. En su lugar, debe escribir una función `converter_init()`. `converter_init()` siempre acepta un parámetro `self`; después de eso, todos los parámetros adicionales *deben* ser solo palabras clave. Cualquier argumento que se pase al convertidor en Argument Clinic se pasará a su `converter_init()`.

Hay algunos miembros adicionales de `CConverter` que tal vez desee especificar en su subclase. Aquí está la lista actual:

type El tipo C que se utilizará para esta variable. `type` debe ser una cadena de Python que especifique el tipo, por ejemplo `int`. Si se trata de un tipo de puntero, la cadena de tipo debe terminar con `'*'`.

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` como debería aparecer en el código Python, como una cadena. O `None` si no hay un valor predeterminado.

c_default `default` como debería aparecer en el código C, como una cadena de caracteres. O `None` si no hay un valor predeterminado.

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 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()`.

Aquí está el ejemplo más simple de un convertidor personalizado, de `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.

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`.

4.17 Escribiendo un convertidor de retorno personalizado

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.

Los convertidores de retorno deben tener una subclase de `CReturnConverter`. Todavía no hay ejemplos de convertidores de retorno personalizados, porque todavía no se utilizan ampliamente. Si desea escribir su propio convertidor de retorno, lea `Tools/clinic/clinic.py`, específicamente la implementación de `CReturnConverter` y todas sus subclases.

4.18 METH_O y METH_NOARGS

Para convertir una función usando `METH_O`, asegúrese de que el único argumento de la función esté usando el convertidor de `object` y marque los argumentos como solo posicional:

```
/*[clinic input]
meth_o_sample

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

Para convertir una función usando `METH_NOARGS`, simplemente no especifique ningún argumento.

Aún puede usar un autoconversor, un convertidor de retorno y especificar un argumento de `tipo` para el convertidor de objetos para `METH_O`.

4.19 funciones tp_new y tp_init

Puede convertir las funciones `tp_new` y `tp_init`. Simplemente nómbrelas `__new__` o `__init__` según corresponda. Notas:

- 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 se genera ningún `PyMethodDef #define` para estas funciones.
- 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).

4.20 Cambiar y redirigir la salida de Clinic

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 Un campo, en este contexto, es una subsección del resultado de la Clínica. Por ejemplo, el `#define` para la estructura `PyMethodDef` es un campo, llamado `methoddef_define`. La clínica tiene siete campos diferentes que puede generar por definición de función:

```
docstring_prototype
docstring_definition
methoddef_define
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 Un «archivo clínico» separado que Clinic creará automáticamente. El nombre de archivo elegido para el archivo es {basename}.clinic{extension}, donde a basename y extension se les asignó la salida de os.path.splitext() ejecutar en El archivo actual. (Ejemplo: el destino del file para _pickle.c se escribiría en _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.

4.21 El truco #ifdef

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 */
```

Luego, elimine esas tres líneas de la estructura `PyMethodDef`, reemplazándolas con la macro Argument Clinic generada:

```
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.
```

Cuando esto suceda, simplemente abra su archivo, busque el bloque `dump buffer` que Argument Clinic agregó a su archivo (estará en la parte inferior), luego muévalo arriba de la estructura `PyMethodDef` donde esa macro se utiliza.

4.22 Usando Argument Clinic en archivos Python

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:...]*/
```

Índice

P