Argument Clinic 的用法

发布 *3.10.5*

Guido van Rossum
and the Python development team

六月 14, 2022

Python Software Foundation
Email: docs@python.org

Contents

作者 Larry Hastings

摘要

Argument Clinic 是 CPython 的一个 C 文件预处理器。旨在自动处理所有与“内置”参数解析有关的代码。本文展示了将 C 函数转换为配合 Argument Clinic 工作的做法，还介绍了一些关于 Argument Clinic 用法的进阶内容。

目前 Argument Clinic 视作仅供 CPython 内部使用。不支持在 CPython 之外的文件中使用，也不保证未来版本会向下兼容。换句话说：如果维护的是 CPython 的外部 C 语言扩展，欢迎在自己的代码中试用 Argument Clinic。但 Argument Clinic 与新版 CPython 中的版本 可能完全不兼容，且会打乱全部代码。

1 Argument Clinic 的设计目标

Argument Clinic 的主要目标，是接管 CPython 中的所有参数解析代码。这意味着，如果要把某个函数转换为配合 Argument Clinic 一起工作，则该函数不应再作任何参数解析工作——Argument Clinic 生成的代码应该是个“黑盒”，CPython 会在顶部发起调用，底部则调用自己的代码，`PyObject *args`（也许还有 `PyObject *kwargs`）会神奇地转换成所需的 C 变量和类型。

Argument Clinic 为了能完成主要目标，用起来必须方便。目前，使用 CPython 的参数解析库是一件苦差事，需要在很多地方维护冗余信息。如果使用 Argument Clinic，则不必再重复代码了。

显然，除非 Argument Clinic 解决了自身的问题，且没有产生新的问题，否则没有人会愿意用它。所以，Argument Clinic 最重要的事情就是生成正确的代码。如果能加速代码的运行当然更好，但至少不应引入明显的减速。（最终 Argument Clinic 应该可以实现较大的速度提升——代码生成器可以重写一下，以产生量身定做的参数解析代码，而不是调用通用的 CPython 参数解析库。这会让参数解析达到最佳速度！）

此外，Argument Clinic 必须足够灵活，能够与任何参数解析的方法一起工作。Python 有一些函数具备一些非常奇怪的解析行为；Argument Clinic 的目标是支持所有这些函数。

最后，Argument Clinic 的初衷是为 CPython 内置程序提供内省“签名”。以前如果传入一个内置函数，内省查询函数会抛出异常。有了 Argument Clinic，再不会发生这种问题了！

在与 Argument Clinic 合作时，应该牢记一个理念：给它的信息越多，它做得就会越好。诚然，Argument Clinic 现在还比较简单。但会演变得越来越复杂，应该能够利用给出的全部信息干很多聪明而有趣的事情。

2 基本概念和用法

Argument Clinic 与 CPython 一起提供，位于 `Tools/clinic/clinic.py`。若要运行它，请指定一个 C 文件作为参数。

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

Argument Clinic 会扫描 C 文件，精确查找以下代码：

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

一旦找到一条后，就会读取所有内容，直至遇到以下代码：

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

这两行之间的所有内容都是 Argument Clinic 的输入。所有行，包括开始和结束的注释行，统称为 Argument Clinic “块”。

Argument Clinic 在解析某一块时，会生成输出信息。输出信息会紧跟着该块写入 C 文件中，后面还会跟着包含校验和的注释。现在 Argument Clinic 块看起来应如下所示：

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

如果对同一文件第二次运行 Argument Clinic，则它会丢弃之前的输出信息，并写入带有新校验行的输出信息。不过如果输入没有变化，则输出也不会有变化。

不应去改动 Argument Clinic 块的输出部分。而应去修改输入，直到生成所需的输出信息。(这就是校验和的用途——检测是否有人改动了输出信息，因为在 Argument Clinic 下次写入新的输出时，这些改动都会丢失)。

为了清晰起见，下面列出了 Argument Clinic 用到的术语：

- 注释的第一行 `/*[clinic input]` 是 *起始行*。
- 注释（`[clinic start generated code]*/`）的最后一行是 *结束行*。
- 最后一行（`/*[clinic end generated code: checksum=...]*/`）是 *校验和行*。
- 在起始行和结束行之间是 *输入数据*。
- 在结束行和校验和行之间是 *输出数据*。
- 从开始行到校验和行的所有文本，都是 *块*。(Argument Clinic 尚未处理成功的块，没有输出或校验和行，但仍视作一个块)。

3 函数的转换

要想了解 Argument Clinic 是如何工作的，最好的方式就是转换一个函数与之合作。下面介绍需遵循的最基本步骤。请注意，若真的准备在 CPython 中进行检查，则应进行更深入的转换，使用一些本文后续会介绍到的高级概念（比如“返回转换”和“自转换”）。但以下例子将维持简单，以供学习。

就此开始

0. 请确保 CPython 是最新的已签出版本。
1. 找到一个调用 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()`，且未被转换为采用 Argument Clinic 的 Python 内置程序。这里用了 `_pickle.Pickler.dump()`。
2. 如果对 `PyArg_Parse` 函数的调用采用了以下格式化单元：

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

   或者多次调用 `PyArg_ParseTuple()`，则应再选一个函数。Argument Clinic *确实* 支持上述这些状况。但这些都是高阶内容——第一次就简单一些吧。

   此外，如果多次调用 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()` 且同一参数需支持不同的类型，或者用到 PyArg_Parse 以外的函数来解析参数，则可能不适合转换为 Argument Clinic 形式。Argument Clinic 不支持通用函数或多态参数。
3. 在函数上方添加以下模板，创建块：

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

4. 剪下文档字符串并粘贴到 `[clinic]` 行之间，去除所有的无用字符，使其成为一个正确引用的 C 字符串。最有应该只留下带有左侧缩进的文本，且行宽不大于 80 个字符。(参数 Clinic 将保留文档字符串中的缩进。)

   如果文档字符串的第一行看起来像是函数的签名，就把这一行去掉吧。((文档串不再需要用到它——将来对内置函数调用 `help()` 时，第一行将根据函数的签名自动建立。)

   示例：

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

5. 如果文档字符串中没有“摘要”行，Argument Clinic 会报错。所以应确保带有摘要行。“摘要”行应为在文档字符串开头的一个段落，由一个 80 列的单行构成。

   (示例的文档字符串只包括一个摘要行，所以示例代码这一步不需改动)。

6. 在文档字符串上方，输入函数的名称，后面是空行。这应是函数的 Python 名称，而且应是句点分隔的完整路径——以模块的名称开始，包含所有子模块名，若函数为类方法则还应包含类名。

   示例：

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

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

7. 如果是第一次在此 C 文件中用到 Argument Clinic 的模块或类，必须对其进行声明。清晰的 Argument Clinic 写法应于 C 文件顶部附近的某个单独块中声明这些，就像 include 文件和 statics 放在顶部一样。(在这里的示例代码中，将这两个块相邻给出。)

   类和模块的名称应与暴露给 Python 的相同。请适时检查 `PyModuleDef` 或 `PyTypeObject` 中定义的名称。

   在声明某个类时，还必须指定其 C 语言类型的两个部分：用于指向该类实例的指针的类型声明，和指向该类 `PyTypeObject` 的指针。

   示例：

```
/*[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. 声明函数的所有参数。每个参数都应另起一行。所有的参数行都应对齐函数名和文档字符串进行缩进。

   这些参数行的常规形式如下：

```
name_of_parameter: converter
```

   如果参数带有缺省值，请加在转换器之后：

```
name_of_parameter: converter = default_value
```

   Argument Clinic 对“缺省值”的支持方式相当复杂；更多信息请参见关于缺省值的部分 。

   在参数行下面添加一个空行。

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.

每个参数都要从 "PyArg_Parse()" 格式参数中复制其 "格式单元"，并以带引号字符串的形式指定其转换器。("格式单元" 是 `format` 参数的 1-3 个字符的正式名称，用于让参数解析函数知晓该变量的类型及转换方法。关于格式单位的更多信息，请参阅 arg-parsing)。

对于像 `z#` 这样的多字符格式单元，要使用 2-3 个字符组成的整个字符串。

示例：

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

9. 如果函数的格式字符串包含 `|`，意味着有些参数带有缺省值，这可以忽略。Argument Clinic 根据参数是否有缺省值来推断哪些参数是可选的。

   如果函数的格式字符串中包含 `$`，意味着只接受关键字参数，请在第一个关键字参数之前单独给出一行 `*`，缩进与参数行对齐。

   (`_pickle.Pickler.dump` 两种格式字符串都没有，所以这里的示例不用改动。)

10. 如果 C 函数调用的是 `PyArg_ParseTuple()` (而不是 `PyArg_ParseTupleAndKeywords()`)，那么其所有参数均是仅限位置参数。

    若要在 Argument Clinic 中把所有参数都标记为只认位置，请在最后一个参数后面一行加入一个 `/`，缩进程度与参数行对齐。

    目前这个标记是全体生效；要么所有参数都是只认位置，要么都不是。(以后 Argument Clinic 可能会放宽这一限制。)

    示例：

```
/*[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. 为每个参数都编写一个文档字符串，这很有意义。但这是可选项；可以跳过这一步。

    下面介绍如何添加逐参数的文档字符串。逐参数文档字符串的第一行必须比参数定义多缩进一层。第一行的左边距即确定了所有逐参数文档字符串的左边距；所有文档字符串文本都要同等缩进。文本可以用多行编写。

    示例：

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

/*[clinic input]
_pickle.Pickler.dump

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

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

12. 保存并关闭该文件，然后运行 `Tools/clinic/clinic.py` 。运气好的话就万事大吉——程序块现在有了输出信息，并且生成了一个 `.c.h` 文件！在文本编辑器中重新打开该文件，可以看到：

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

显然，如果 Argument Clinic 未产生任何输出，那是因为在输入信息中发现了错误。继续修正错误并重试，直至 Argument Clinic 正确地处理好文件。

为了便于阅读，大部分“胶水”代码已写入 `.c.h` 文件中。需在原 `.c` 文件中包含这个文件，通常是在 clinic 模块之后：

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

13. 请仔细检查 Argument Clinic 生成的参数解析代码，是否与原有代码基本相同。

首先，确保两种代码使用相同的参数解析函数。原有代码必须调用 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()` ；确保 Argument Clinic 生成的代码调用 完全相同的函数。

其次，传给 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()` 的格式字符串应该 完全与原有函数中的相同，直到冒号或分号为止。

（Argument Clinic 生成的格式串一定是函数名后跟着 `:` 。如果现有代码的格式串以 `;` 结尾，这种改动不会影响使用，因此不必担心。）

第三，如果格式单元需要指定两个参数（比如长度、编码字符串或指向转换函数的指针），请确保第二个参数在两次调用时 完全相同。

第四，在输出部分会有一个预处理器宏，为该内置函数定义合适的静态 `PyMethodDef` 结构：

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

此静态结构应与本内置函数现有的静态结构 `PyMethodDef` 完全相同。

只要上述这几点存在不一致，请调整 Argument Clinic 函数定义，并重新运行 `Tools/clinic/clinic.py` ，直至 完全相同。

14. 注意，输出部分的最后一行是“实现”函数的声明。也就是该内置函数的实现代码所在。删除需要修改的函数的现有原型，但保留开头的大括号。再删除其参数解析代码和输入变量的所有声明。注意现在 Python 所见的参数即为此实现函数的参数；如果实现代码给这些变量采用了不同的命名，请进行修正。

    因为稍显怪异，所以还是重申一下。现在的代码应该如下所示：

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

    上面是 Argument Clinic 生成的校验值和函数原型。函数应该带有闭合的大括号，实现代码位于其中。

    示例：

```
/*[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;
    }

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

    ...
```

15. 还记得用到 `PyMethodDef` 结构的宏吧？找到函数中已有的 `PyMethodDef` 结构，并替换为宏的引用。（如果函数是模块级的，可能会在文件的末尾附近；如果函数是个类方法，则可能会在靠近实现代码的下方。）

    注意，宏尾部带有一个逗号。所以若用宏替换已有的静态结构 `PyMethodDef` 时，请勿在结尾添加

逗号了。

示例：

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

16. 编译，然后运行回归测试套件中的有关测试程序。不应引入新的编译警告或错误，且对 Python 也不应有外部可见的变化。

    差别只有一个，即 `inspect.signature()` 运行于新的函数上，现在应该新提供一个有效的签名！

    祝贺你，现在已经用 Argument Clinic 移植了第一个函数。

4 进阶

现在 Argument Clinic 的使用经验已具备了一些，该介绍一些高级内容了。

4.1 Symbolic default values

提供给参数的默认值不能是表达式。目前明确支持以下形式：

- 数值型常数（整数和浮点数）。
- 字符串常量
- `True` 、`False` 和 `None` 。
- 以模块名开头的简单符号常量，如 `sys.maxsize`。

如果你感到好奇，这是在 `from_builtin()` (`Lib/inspect.py`) 中实现的。

（未来可能需要加以细化，以便可以采用 `CONSTANT - 1` 之类的完整表达式。）

4.2 对 Argument Clinic 生成的 C 函数和变量进行重命名

Argument Clinic 会自动为其生成的函数命名。如果生成的名称与现有的 C 函数冲突，这偶尔可能会造成问题，有一个简单的解决方案：覆盖 C 函数的名称。只要在函数声明中加入关键字 `"as"` ，然后再加上要使用的函数名。Argument Clinic 将以该函数名为基础作为（生成的）函数名，然后在后面加上 `"_impl"`，并用作实现函数的名称。

例如，若对 `pickle.Pickler.dump` 生成的 C 函数进行重命名，应如下所示：

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

...
```

原函数会被命名为 `pickler_dumper()`，而实现函数现在被命名为 “pickler_dumper_impl()“。

同样的问题依然会出现：想给某个参数取个 Python 用名，但在 C 语言中可能用不了。Argument Clinic 允许在 Python 和 C 中为同一个参数取不同的名字，依然是利用 `"as"` 语法：

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

    obj: object
    file as file_obj: object
```

（下页继续）

(续上页)

```
    protocol: object = NULL
    *
    fix_imports: bool = True
```

这里 Python（签名和 keywords 数组中）中用的名称是 file，而 C 语言中的变量命名为 file_obj。

self 参数也可以进行重命名。

4.3 函数转换会用到 PyArg_UnpackTuple

若要将函数转换为采用 PyArg_UnpackTuple() 解析其参数，只需写出所有参数，并将每个参数定义为 object。可以指定 type 参数，以便能转换为合适的类型。所有参数都应标记为只认位置（在最后一个参数后面加上 /）。

目前，所生成的代码将会用到 PyArg_ParseTuple()，但很快会做出改动。

4.4 可选参数组

有些过时的函数用到了一种让人头疼的函数解析方式：计算位置参数的数量，据此用 switch 语句进行各个不同的 PyArg_ParseTuple() 调用。（这些函数不能接受只认关键字的参数。）在没有 PyArg_ParseTupleAndKeywords() 之前，这种方式曾被用于模拟可选参数。

虽然这种函数通常可以转换为采用 PyArg_ParseTupleAndKeywords()、可选参数和默认值的方式，但并不是全都可以做到。这些过时函数中，PyArg_ParseTupleAndKeywords() 并不能直接支持某些功能。最明显的例子是内置函数 range()，它的必需参数的 左边存在一个可选参数！另一个例子是 curses.window.addch()，它的两个参数是一组，必须同时指定。（参数名为 x 和 y；如果调用函数时传入了 x，则必须同时传入 “y“；如果未传入 x，则也不能传入 y）。

不管怎么说，Argument Clinic 的目标就是在不改变语义的情况下支持所有现有 CPython 内置参数的解析。因此，Argument Clinic 采用所谓的 可选组方案来支持这种解析方式。可选组是必须一起传入的参数组。他们可以在必需参数的左边或右边，只能用于只认位置的参数。

注解： 可选组 仅适用于多次调用 PyArg_ParseTuple() 的函数！采用 任何其他方式解析参数的函数，应该 几乎不采用可选组转换为 Argument Clinic 解析。目前，采用可选组的函数在 Python 中无法获得准确的签名，因为 Python 不能理解这个概念。请尽可能避免使用可选组。

若要定义可选组，可在要分组的参数前面加上 [，在这些参数后加上 “]“，要在同一行上。举个例子，下面是 curses.window.addch 采用可选组的用法，前两个参数和最后一个参数可选：

```
/*[clinic input]

curses.window.addch

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

    ch: object
      Character to add.

    [
    attr: long
      Attributes for the character.
    ]
```

(下页继续)

(续上页)

```
    /

...
```

注：

- 每一个可选组，都会额外传入一个代表分组的参数。参数为 int 型，名为 `group_{direction}_{number}`，其中 `{direction}` 取决于此参数组位于必需参数 `right` 还是 `left`，而 `{number}` 是一个递增数字（从 1 开始），表示此参数组与必需参数之间的距离。在调用函数时，若未用到此参数组则此参数将设为零，若用到了参数组则该参数为非零。所谓的用到或未用到，是指在本次调用中形参是否收到了实参。
- 如果不存在必需参数，可选组的行为等同于出现在必需参数的右侧。
- 在模棱两可的情况下，参数解析代码更倾向于参数左侧（在必需参数之前）。
- 可选组只能包含只认位置的参数。
- 可选组 仅限用于过时代码。请勿在新的代码中使用可选组。

4.5 采用真正的 Argument Clinic 转换器，而不是“传统转换器”

为了节省时间，尽量减少要学习的内容，实现第一次适用 Argument Clinic 的移植，上述练习简述的是“传统转换器”的用法。“传统转换器”只是一种简便用法，目的就是更容易地让现有代码移植为适用于 Argument Clinic 。说白了，在移植 Python 3.4 的代码时，可以考虑采用。

不过从长远来看，可能希望所有代码块都采用真正的 Argument Clinic 转换器语法。原因如下：

- 合适的转换器可读性更好，意图也更清晰。
- 有些格式单元是“传统转换器”无法支持的，因为这些格式需要带上参数，而传统转换器的语法不支持指定参数。
- 后续可能会有新版的参数解析库，提供超过 `PyArg_ParseTuple()` 支持的功能；而这种灵活性将无法适用于传统转换器转换的参数。

因此，若是不介意多花一点精力，请使用正常的转换器，而不是传统转换器。

简而言之，Argument Clinic（非传统）转换器的语法看起来像是 Python 函数调用。但如果函数没有明确的参数（所有函数都取默认值），则可以省略括号。因此 `bool` 和 `bool()` 是完全相同的转换器。

Argument Clinic 转换器的所有参数都只认关键字。所有 Argument Clinic 转换器均可接受以下参数：

`c_default` 该参数在 C 语言中的默认值。具体来说，将是在“解析函数”中声明的变量的初始化器。用法参见*the section on default values* 。定义为字符串。

`annotation` 参数的注解值。目前尚不支持，因为 **PEP 8** 规定 Python 库不得使用注解。

此外，某些转换器还可接受额外的参数。下面列出了这些额外参数及其含义：

`accept` 一些 Python 类型的集合（可能还有伪类型）；用于限制只接受这些类型的 Python 参数。（并非通用特性；只支持传统转换器列表中给出的类型）。

若要能接受 `None`，请在集合中添加 `NoneType`。

`bitwise` 仅用于无符号整数。写入形参的将是 Python 实参的原生整数值，不做任何越界检查，即便是负值也一样。

`converter` 仅用于 `object` 转换器。为某个 C 转换函数指定名称，用于将对象转换为原生类型。

`encoding` 仅用于字符串。指定将 Python str(Unicode) 转换为 C 语言的 `char *` 时应该采用的编码。

`subclass_of` 仅用于 `object` 转换器。要求 Python 值是 Python 类型的子类，用 C 语言表示。

type 仅用于 object 和 self 转换器。指定用于声明变量的 C 类型。默认值是 "PyObject *"。

zeroes 仅用于字符串。如果为 True，则允许在值中嵌入 NUL 字节（'\\0'）。字符串的长度将通过名为 <parameter_name>_length 的参数传入，跟在字符串参数的后面。

请注意，并不是所有参数的组合都能正常生效。通常这些参数是由相应的 PyArg_ParseTuple 格式单元实现的，行为是固定的。比如目前不能不指定 bitwise=True 就去调用 unsigned_short。虽然完全有理由认为这样可行，但这些语义并没有映射到任何现有的格式单元。所以 Argument Clinic 并不支持。（或者说，至少目前还不支持。）

下表列出了传统转换器与真正的 Argument Clinic 转换器之间的映射关系。左边是传统的转换器，右边是应该换成的文本。

'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})

举个例子，下面是采用合适的转换器的例子 pickle.Pickler.dump：

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

真正的转换器有一个优点，就是比传统的转换器更加灵活。例如，unsigned_int 转换器（以及所有 unsigned_ 转换器）可以不设置 bitwise=True 。他们默认会对数值进行范围检查，而且不会接受负数。用传统转换器就做不到这一点。

Argument Clinic 会列明其全部转换器。每个转换器都会给出可接受的全部参数，以及每个参数的默认值。只要运行 Tools/clinic/clinic.py --converters 就能得到完整的列表。

4.6 Py_buffer

在使用 Py_buffer 转换器（或者 's*'、'w*'、'*y' 或 'z*' 传统转换器）时，不可在所提供的缓冲区上调用 PyBuffer_Release()。Argument Clinic 生成的代码会自动完成此操作（在解析函数中）。

4.7 高级转换器

还记得编写第一个函数时跳过的那些格式单元吗，因为他们是高级内容？下面就来介绍这些内容。

其实诀窍在于，这些格式单元都需要给出参数——要么是转换函数，要么是类型，要么是指定编码的字符串。（但“传统转换器”不支持参数。这就是为什么第一个函数要跳过这些内容）。为格式单元指定的参数于是就成了转换器的参数；参数可以是 converter``（对于 ``O&）、subclass_of``（对于 ``O!），或者是 encoding （对于 e 开头的格式单元）。

在使用 subclass_of 时，可能还需要用到 object() 的另一个自定义参数：type，用于设置参数的实际类型。例如，为了确保对象是 PyUnicode_Type 的子类，可能想采用转换器 object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')。

Argument Clinic 用起来可能存在一个问题：丧失了 e 开头的格式单位的一些灵活性。在手工编写 PyArg_Parse 调用时，理论上可以在运行时决定传给 PyArg_ParseTuple() 的编码字符串。但现在这个字符串必须在 Argument-Clinic 预处理时进行硬编码。这个限制是故意设置的；以便简化对这种格式单元的支持，并允许以后进行优化。这个限制似乎并不合理；CPython 本身总是为 e 开头的格式单位参数传入静态的硬编码字符串。

4.8 参数的默认值

参数的默认值可以是多个值中的一个。最简单的可以是字符串、int 或 float 字面量。

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

还可以使用 Python 的任何内置常量。

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

对默认值 NULL 和简单表达式还提供特别的支持，下面将一一介绍。

4.9 默认值 NULL

对于字符串和对象参数而言，可以设为 None，表示没有默认值。但这意味着会将 C 变量初始化为 Py_None。为了方便起见，提供了一个特殊值 “NULL“，目的就是为了让 Python 认为默认值就是 None，而 C 变量则会初始化为 NULL。

4.10 设为默认值的表达式

参数的默认值不仅可以是字面量。还可以是一个完整的表达式，可采用数学运算符及对象的属性。但这种支持并没有那么简单，因为存在一些不明显的语义。

请考虑以下例子：

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

sys.maxsize 在不同的系统平台可能有不同的值。因此，Argument Clinic 不能简单地在本底环境对表达式求值并用 C 语言硬编码。所以默认值将用表达式的方式存储下来，运行的时候在请求函数签名时会被求值。

在对表达式进行求值时，可以使用什么命名空间呢？求值过程运行于内置模块的上下文中。因此，如果模块带有名为 max_widgets 的属性，直接引用即可。

```
foo: Py_ssize_t = max_widgets
```

如果表达式不在当前模块中，就会去 sys.modules 查找。比如 sys.maxsize 就是如此找到的。（因为事先不知道用户会加载哪些模块到解释器中，所以最好只用到 Python 会预加载的模块。）

仅当运行时才对缺省值求值，意味着 Argument Clinic 无法计算出正确的 C 缺省值。所以需显式给出。在使用表达式时，必须同时用转换器的 “c_default“ 参数指定 C 语言中的等价表达式。

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

还有一个问题也比较复杂。Argument Clinic 无法事先知道表达式是否有效。解析只能保证看起来是有效值，但无法 实际知晓。在用表达式时须十分小心，确保在运行时能得到有效值。

最后一点，由于表达式必须能表示为静态的 C 语言值，所以存在许多限制。以下列出了不得使用的 Python 特性：

- 功能
- 行内 if 语句（3 if foo else 5）
- 序列类自动解包（*[1, 2, 3]）
- 列表、集合、字典的解析和生成器表达式。
- 元组、列表、集合、字典的字面量

4.11 返回值转换器

Argument Clinic 生成的植入函数默认会返回 PyObject *。但是通常 C 函数的任务是要对某些 C 类型进行计算，然后将其转换为 PyObject * 作为结果。Argument Clinic 可以将输入参数由 Python 类型转换为本地 C 类型——为什么不让它将返回值由本地 C 类型转换为 Python 类型呢？

这就是 “返回值转换器” 的用途。它将植入函数修改成返回某种 C 语言类型，然后在生成的（非植入）函数中添加代码，以便将 C 语言值转换为合适的 PyObject *。

返回值转换器的语法与参数转换器的类似。返回值转换器的定义方式，类似于函数返回值的注解。返回值转换器的行为与参数转换器基本相同，接受参数，参数只认关键字，如果不修改默认参数则可省略括号。

（如果函数同时用到了 "as" 和返回值转换器，"as" 应位于返回值转换器之前。）

返回值转换器还存在一个复杂的问题：出错信息如何表示？通常函数在执行成功时会返回一个有效（非 NULL）指针，失败则返回 NULL。但如果使用了整数的返回值转换器，所有整数都是有效值。Argument Clinic 怎么检测错误呢？解决方案是：返回值转换器会隐含寻找一个代表错误的特殊值。如果返回该特殊值，且设置了出错标记（PyErr_Occurred() 返回 True），那么生成的代码会传递该错误。否则，会对返回值进行正常编码。

目前 Argument Clinic 只支持少数几种返回值转换器。

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

这些转换器都不需要参数。前 3 个转换器如果返回 -1 则表示出错。DecodeFSDefault 的返回值类型是 const char *；若返回 NULL 指针则表示出错。

（还有一个 NoneType 转换器是实验性质的，成功时返回 Py_None ，失败则返回 NULL，且不会增加 Py_None 的引用计数。此转换器是否值得适用，尚不明确）。

只要运行 Tools/clinic/clinic.py --converters ，即可查看 Argument Clinic 支持的所有返回值转换器，包括其参数。

4.12 克隆已有的函数

如果已有一些函数比较相似，或许可以采用 Clinic 的“克隆”功能。克隆之后能够复用以下内容：

- 参数，包括：
  - 名称
  - 转换器（带有全部参数）
  - 默认值
  - 参数前的文档字符串
  - 类别（只认位置、位置或关键字、只认关键字）
- 返回值转换器

唯一不从原函数中复制的是文档字符串；这样就能指定一个新的文档串。

下面是函数的克隆方法：

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

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

（原函数可以位于不同的模块或类中。示例中的 module.class 只是为了说明，两个函数都必须使用全路径）。

对不起，没有什么语法可对函数进行部分克隆或克隆后进行修改。克隆要么全有要么全无。

另外，要克隆的函数必须在当前文件中已有定义。

4.13 调用 Python 代码

下面的高级内容需要编写 Python 代码，存于 C 文件中，并修改 Argument Clinic 的运行状态。其实很简单：只需定义一个 Python 块。

Python 块的分隔线与 Argument Clinic 函数块不同。如下所示：

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

Python 块内的所有代码都会在解析时执行。块内写入 stdout 的所有文本都被重定向到块后的“输出”部分。

以下例子包含了 Python 块，用于在 C 代码中添加一个静态整数变量：

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

4.14 self 转换器的用法

Argument Clinic 用一个默认的转换器自动添加一个“self”参数。自动将 self 参数的 `type` 设为声明类型时指定的“指向实例的指针”。不过 Argument Clinic 的转换器可被覆盖，也即自己指定一个转换器。只要将自己的 `self` 参数作为块的第一个参数即可，并确保其转换器是 `self_converter` 的实例或其子类。

这有什么用呢？可用于覆盖 `self` 的类型，或为其给个不同的默认名称。

如何指定 `self` 对应的自定义类型呢？如果只有 `self` 类型相同的一两个函数，可以直接使用 Argument Clinic 现有的 `self` 转换器，把要用的类型作为 `type` 参数传入：

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

如果有很多函数将使用同一类型的 `self`，则最好创建自己的转换器，继承自 `self_converter` 类但要覆盖其 `type` 成员：

```
/*[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 “定义类”转换器

Argument Clinic 为访问方法的定义类提供了便利。这对 heap type 方法十分有用，因为它需要获取模块级的运行状态。用 `PyType_FromModuleAndSpec()` 将新的堆类型与模块联系起来。现在可以在定义类上用 `PyType_GetModuleState()` 获取模块状态了，例如从模块方法中获取。

示例来自 `Modules/zlibmodule.c`。首先，在 clinic 的输入块添加 `defining_class`：

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

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

运行 Argument Clinic 工具后，会生成以下函数签名：

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

现在，以下代码可以用 `PyType_GetModuleState(cls)` 获取模块状态了：

```
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);
    ...
}
```

See also **PEP 573**.

4.16 Writing a custom converter

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

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

You shouldn’t subclass `CConverter.__init__`. Instead, you should write a `converter_init()` function. `converter_init()` always accepts a `self` parameter; after that, all additional parameters *must* be

keyword-only. Any arguments passed in to the converter in Argument Clinic will be passed along to your `converter_init()`.

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

`type` The C type to use for this variable. `type` should be a Python string specifying the type, e.g. `int`. If this is a pointer type, the type string should end with `' *'`.

`default` The Python default value for this parameter, as a Python value. Or the magic value `unspecified` if there is no default.

`py_default` `default` as it should appear in Python code, as a string. Or `None` if there is no default.

`c_default` `default` as it should appear in C code, as a string. Or `None` if there is no default.

`c_ignored_default` The default value used to initialize the C variable when there is no default, but not specifying a default may result in an "uninitialized variable" warning. This can easily happen when using option groups—although properly-written code will never actually use this value, the variable does get passed in to the impl, and the C compiler will complain about the "use" of the uninitialized value. This value should always be a non-empty string.

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

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

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

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

```
/*[python input]

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

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

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

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

4.17 Writing a custom return converter

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

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

4.18 METH_O and METH_NOARGS

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

```
/*[clinic input]
meth_o_sample

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

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

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

4.19 tp_new and tp_init functions

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

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

4.20 Changing and redirecting Clinic's output

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

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

Let's start with defining some terminology:

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

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

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

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

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

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

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

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

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

suppress The text is suppressed—thrown away.

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

The first new directive is dump:

```
dump <destination>
```

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

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

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

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

output has a number of other functions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The third new directive is `destination`:

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

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

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

The `new` subcommand works like this:

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

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

There are five destination types:

suppress Throws the text away.

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

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

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

destination <name> new <type> <file_template>

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

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

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

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

{basename_root} Basename with the extension clipped off (everything up to but not including the last '.').

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

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

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

The `clear` subcommand works like this:

```
destination <name> clear
```

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

The fourth new directive is `set`:

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

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

Both of these support two format strings:

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

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

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

```
preserve
```

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

4.21 The #ifdef trick

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

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

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

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

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

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

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

```
MODULE_FUNCTIONNAME_METHODDEF
```

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

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

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

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

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

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

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

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

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

4.22 Using Argument Clinic in Python files

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

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

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

索引

P