Argument Clinic How-To

Release 3.9.4

Guido van Rossum
and the Python development team

April 04, 2021

Python Software Foundation
Email: docs@python.org

Contents

1 The Goals Of Argument Clinic **2**

2 Basic Concepts And Usage **2**

3 Converting Your First Function **3**

4 Advanced Topics **9**
4.1 Symbolic default values . . . 9
4.2 Renaming the C functions and variables generated by Argument Clinic . . . 9
4.3 Converting functions using PyArg_UnpackTuple . . . 10
4.4 Optional Groups . . . 10
4.5 Using real Argument Clinic converters, instead of "legacy converters" . . . 11
4.6 Py_buffer . . . 13
4.7 Advanced converters . . . 13
4.8 Parameter default values . . . 14
4.9 The `NULL` default value . . . 14
4.10 Expressions specified as default values . . . 14
4.11 Using a return converter . . . 15
4.12 Cloning existing functions . . . 16
4.13 Calling Python code . . . 16
4.14 Using a "self converter" . . . 17
4.15 Writing a custom converter . . . 17
4.16 Writing a custom return converter . . . 18
4.17 METH_O and METH_NOARGS . . . 18
4.18 tp_new and tp_init functions . . . 19
4.19 Changing and redirecting Clinic's output . . . 19
4.20 The #ifdef trick . . . 22
4.21 Using Argument Clinic in Python files . . . 23

Index **24**

author Larry Hastings

Abstract

Argument Clinic is a preprocessor for CPython C files. Its purpose is to automate all the boilerplate involved with writing argument parsing code for "builtins". This document shows you how to convert your first C function to work with Argument Clinic, and then introduces some advanced topics on Argument Clinic usage.

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

1 The Goals Of Argument Clinic

Argument Clinic's primary goal is to take over responsibility for all argument parsing code inside CPython. This means that, when you convert a function to work with Argument Clinic, that function should no longer do any of its own argument parsing—the code generated by Argument Clinic should be a "black box" to you, where CPython calls in at the top, and your code gets called at the bottom, with `PyObject *args` (and maybe `PyObject *kwargs`) magically converted into the C variables and types you need.

In order for Argument Clinic to accomplish its primary goal, it must be easy to use. Currently, working with CPython's argument parsing library is a chore, requiring maintaining redundant information in a surprising number of places. When you use Argument Clinic, you don't have to repeat yourself.

Obviously, no one would want to use Argument Clinic unless it's solving their problem—and without creating new problems of its own. So it's paramount that Argument Clinic generate correct code. It'd be nice if the code was faster, too, but at the very least it should not introduce a major speed regression. (Eventually Argument Clinic *should* make a major speedup possible—we could rewrite its code generator to produce tailor-made argument parsing code, rather than calling the general-purpose CPython argument parsing library. That would make for the fastest argument parsing possible!)

Additionally, Argument Clinic must be flexible enough to work with any approach to argument parsing. Python has some functions with some very strange parsing behaviors; Argument Clinic's goal is to support all of them.

Finally, the original motivation for Argument Clinic was to provide introspection "signatures" for CPython builtins. It used to be, the introspection query functions would throw an exception if you passed in a builtin. With Argument Clinic, that's a thing of the past!

One idea you should keep in mind, as you work with Argument Clinic: the more information you give it, the better job it'll be able to do. Argument Clinic is admittedly relatively simple right now. But as it evolves it will get more sophisticated, and it should be able to do many interesting and smart things with all the information you give it.

2 Basic Concepts And Usage

Argument Clinic ships with CPython; you'll find it in `Tools/clinic/clinic.py`. If you run that script, specifying a C file as an argument:

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

Argument Clinic will scan over the file looking for lines that look exactly like this:

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

When it finds one, it reads everything up to a line that looks exactly like this:

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

Everything in between these two lines is input for Argument Clinic. All of these lines, including the beginning and ending comment lines, are collectively called an Argument Clinic "block".

When Argument Clinic parses one of these blocks, it generates output. This output is rewritten into the C file immediately after the block, followed by a comment containing a checksum. The Argument Clinic block now looks like this:

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

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

You should never modify the output portion of an Argument Clinic block. Instead, change the input until it produces the output you want. (That's the purpose of the checksum—to detect if someone changed the output, as these edits would be lost the next time Argument Clinic writes out fresh output.)

For the sake of clarity, here's the terminology we'll use with Argument Clinic:

- The first line of the comment (`/*[clinic input]`) is the *start line*.
- The last line of the initial comment (`[clinic start generated code]*/`) is the *end line*.
- The last line (`/*[clinic end generated code: checksum=...]*/`) is the *checksum line*.
- In between the start line and the end line is the *input*.
- In between the end line and the checksum line is the *output*.
- All the text collectively, from the start line to the checksum line inclusively, is the *block*. (A block that hasn't been successfully processed by Argument Clinic yet doesn't have output or a checksum line, but it's still considered a block.)

3 Converting Your First Function

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

Let's dive in!

0. Make sure you're working with a freshly updated checkout of the CPython trunk.
1. Find a Python builtin that calls either `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()`, and hasn't been converted to work with Argument Clinic yet. For my example I'm using `_pickle.Pickler.dump()`.
2. If the call to the `PyArg_Parse` function uses any of the following format units:

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

   or if it has multiple calls to `PyArg_ParseTuple()`, you should choose a different function. Argument Clinic *does* support all of these scenarios. But these are advanced topics—let's do something simpler for your first function.

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

3. Add the following boilerplate above the function, creating our block:

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

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

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

   Sample:

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

5. If your docstring doesn't have a "summary" line, Argument Clinic will complain. So let's make sure it has one. The "summary" line should be a paragraph consisting of a single 80-column line at the beginning of the docstring.

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

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

   Sample:

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

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

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

   The name of the class and module should be the same as the one seen by Python. Check the name defined in the `PyModuleDef` or `PyTypeObject` as appropriate.

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

   Sample:

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

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

(continues on next page)

(continued from previous page)

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

8. Declare each of the parameters to the function. Each parameter should get its own line. All the parameter lines should be indented from the function name and the docstring.

   The general form of these parameter lines is as follows:

   ```
   name_of_parameter: converter
   ```

   If the parameter has a default value, add that after the converter:

   ```
   name_of_parameter: converter = default_value
   ```

   Argument Clinic's support for "default values" is quite sophisticated; please see *the section below on default values* for more information.

   Add a blank line below the parameters.

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

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

   For multicharacter format units like `z#`, use the entire two-or-three character string.

   Sample:

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

   If your function has `$` in the format string, meaning it takes keyword-only arguments, specify `*` on a line by itself before the first keyword-only argument, indented the same as the parameter lines.

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

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

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

    Currently this is all-or-nothing; either all parameters are positional-only, or none of them are. (In the future Argument Clinic may relax this restriction.)

    Sample:

```
/*[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. It's helpful to write a per-parameter docstring for each parameter. But per-parameter docstrings are optional; you can skip this step if you prefer.

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

    Sample:

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

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

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

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

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

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

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

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

13. Double-check that the argument-parsing code Argument Clinic generated looks basically the same as the existing code.

    First, ensure both places use the same argument-parsing function. The existing code must call either `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()`; ensure that the code generated by Argument Clinic calls the *exact* same function.

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

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

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

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

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

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

    If any of these items differ in *any way*, adjust your Argument Clinic function specification and rerun `Tools/clinic/clinic.py` until they *are* the same.

14. Notice that the last line of its output is the declaration of your "impl" function. This is where the builtin's implementation goes. Delete the existing prototype of the function you're modifying, but leave the opening curly brace. Now delete its argument parsing code and the declarations of all the variables it dumps the arguments into. Notice how the Python arguments are now arguments to this impl function; if the implementation used different names for these variables, fix it.

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

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

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

    Sample:

```
/*[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.
```

(continues on next page)

(continued from previous page)

```
    /

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

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

    Sample:

```
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.

    Well, except for one difference: `inspect.signature()` run on your function should now provide a valid signature!

    Congratulations, you've ported your first function to work with Argument Clinic!

4 Advanced Topics

Now that you've had some experience working with Argument Clinic, it's time for some advanced topics.

4.1 Symbolic default values

The default value you provide for a parameter can't be any arbitrary expression. Currently the following are explicitly supported:

- Numeric constants (integer and float)
- String constants
- `True`, `False`, and `None`
- Simple symbolic constants like `sys.maxsize`, which must start with the name of the module

In case you're curious, this is implemented in `from_builtin()` in `Lib/inspect.py`.

(In the future, this may need to get even more elaborate, to allow full expressions like `CONSTANT - 1`.)

4.2 Renaming the C functions and variables generated by Argument Clinic

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

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

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

...
```

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

Similarly, you may have a problem where you want to give a parameter a specific Python name, but that name may be inconvenient in C. Argument Clinic allows you to give a parameter different names in Python and in C, using the same `"as"` syntax:

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

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

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

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

4.3 Converting functions using PyArg_UnpackTuple

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

Currently the generated code will use `PyArg_ParseTuple()`, but this will change soon.

4.4 Optional Groups

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

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

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

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

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

```
/*[clinic input]

curses.window.addch

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

    ch: object
      Character to add.

    [
    attr: long
      Attributes for the character.
    ]
    /

...
```

Notes:

- For every optional group, one additional parameter will be passed into the impl function representing the group. The parameter will be an int named `group_{direction}_{number}`, where `{direction}` is either `right` or `left` depending on whether the group is before or after the required parameters, and `{number}` is a monotonically increasing number (starting at 1) indicating how far away the group is from the required parameters. When the impl is called, this parameter will be set to zero if this group was unused, and set to non-zero if this group was used. (By used or unused, I mean whether or not the parameters received arguments in this invocation.)
- If there are no required arguments, the optional groups will behave as if they're to the right of the required arguments.
- In the case of ambiguity, the argument parsing code favors parameters on the left (before the required parameters).
- Optional groups can only contain positional-only parameters.
- Optional groups are *only* intended for legacy code. Please do not use optional groups for new code.

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

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

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

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

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

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

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

`c_default` The default value for this parameter when defined in C. Specifically, this will be the initializer for the variable declared in the "parse function". See *the section on default values* for how to use this. Specified as a string.

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

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

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

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

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

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

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

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

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

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

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

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

`'B'`	`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})`

Continued on next page

Table 1 – continued from previous page

'y#'	str(accept={robuffer}, zeroes=True)
'y*'	Py_buffer
'Z'	Py_UNICODE(accept={str, NoneType})
'Z#'	Py_UNICODE(accept={str, NoneType}, zeroes=True)
'z'	str(accept={str, NoneType})
'z#'	str(accept={str, NoneType}, zeroes=True)
'z*'	Py_buffer(accept={buffer, str, NoneType})

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

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

    obj: object
        The object to be pickled.
    /

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

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

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

4.6 Py_buffer

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

4.7 Advanced converters

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

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

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

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

4.8 Parameter default values

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

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

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

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

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

4.9 The `NULL` default value

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

4.10 Expressions specified as default values

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

Consider the following example:

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

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

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

```
foo: Py_ssize_t = max_widgets
```

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

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

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

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

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

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

4.11 Using a return converter

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

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

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

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

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

Currently Argument Clinic supports only a few return converters:

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

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

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

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

4.12 Cloning existing functions

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

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

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

Here's the syntax for cloning a function:

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

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

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

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

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

4.13 Calling Python code

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

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

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

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

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

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

4.14 Using a "self converter"

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

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

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

```
/*[clinic input]

_pickle.Pickler.dump

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

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

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

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

/*[clinic input]

_pickle.Pickler.dump

  self: PicklerObject
  obj: object
  /

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

4.15 Writing a custom converter

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

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

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

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

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

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

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

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

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

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

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

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

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

```
/*[python input]

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

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

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

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

4.16 Writing a custom return converter

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

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

4.17 METH_O and METH_NOARGS

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

```
/*[clinic input]
meth_o_sample

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

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

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

4.18 tp_new and tp_init functions

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

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

4.19 Changing and redirecting Clinic's output

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

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

Let's start with defining some terminology:

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

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

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

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

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

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

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

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

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

suppress The text is suppressed—thrown away.

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

The first new directive is `dump`:

```
dump <destination>
```

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

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

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

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

`output` has a number of other functions:

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

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

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

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

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

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

4.20 The #ifdef trick

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

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

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

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

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

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

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

```
MODULE_FUNCTIONNAME_METHODDEF
```

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

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

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

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

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

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

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

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

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

4.21 Using Argument Clinic in Python files

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

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

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

Index

P

Python Enhancement Proposals
    PEP 8, 11