Your first C API extension module

- What we'll do
- Start with the headers
- Running your build tool
- Module export hook
- The slot table
- Exposing a function
  - Method definitions
- Returning an integer
- Accepting a string
- The result
- Appendix: Other build tools
  - Workaround for missing PyInit function
  - Compiling directly
    - Linux

This tutorial will take you through creating a simple Python extension module written in C or C++.

We will use the low-level Python C API directly. For easier ways to create extension modules, see the `recommended third party tools <c-api-tools>`.

The tutorial assumes basic knowledge about Python: you should be able to define functions in Python code before starting to write them in C. See `tutorial-index` for an introduction to Python

itself.

The tutorial should be approachable for anyone who can write a basic C library. While we will mention several concepts that a C beginner would not be expected to know, like `static` functions or linkage declarations, understanding these is not necessary for success.

We will focus on giving you a "feel" of what Python's C API is like. It will not teach you important concepts, like error handling and reference counting, which are covered in later chapters.

We will assume that you use a Unix-like system (including macOS and Linux), or Windows. On other systems, you might need to adjust some details -- for example, a system command name.

You need to have a suitable C compiler and Python development headers installed. On Linux, headers are often in a package like `python3-dev` or `python3-devel`.

You need to be able to install Python packages. This tutorial uses pip (`pip install`), but you can substitute any tool that can build and install `pyproject.toml`-based projects, like uv (`uv pip install`). Preferably, have a `virtual environment <venv-def>` activated.

> [!NOTE] This tutorial uses APIs that were added in CPython 3.15. To create an extension that's compatible with earlier versions of CPython, please follow an earlier version of this documentation.
>
> This tutorial uses C syntax added in C11 and C++20. If your extension needs to be compatible with earlier standards, please follow tutorials in documentation for Python 3.14 or below.

What we'll do

Let's create an extension module called `spam`[^1], which will include a Python interface to the C standard library function `system`. This function is defined in `stdlib.h`. It takes a C string as argument, runs the argument as a system command, and returns a result value as an integer. A manual page for `system` might summarize it this way:

```
#include <stdlib.h>
int system(const char *command);
```

Note that like many functions in the C standard library, this function is already exposed in Python. In production, use `os.system` or `subprocess.run` rather than the module you'll write here.

We want this function to be callable from Python as follows:

```
>>> import spam
>>> status = spam.system("whoami")
User Name
>>> status
0
```

> [!NOTE] The system command `whoami` prints out your username. It's useful in tutorials like this one because it has the same name on both Unix and Windows.

Start with the headers

Begin by creating a directory for this tutorial, and switching to it on the command line. Then, create a file named `spammodule.c` in your directory. [^2]

In this file, we'll include two headers: `Python.h` to pull in all declarations of the Python C API, and `stdlib.h` for the `system` function.[^3]

Add the following lines to `spammodule.c`:

../includes/capi-extension/spammodule-01.c

Be sure to put `stdlib.h`, and any other standard library includes, *after* `Python.h`. On some systems, Python may define some pre-processor definitions that affect the standard headers.

Running your build tool

With only the includes in place, your extension won't do anything. Still, it's a good time to compile it and try to import it. This will ensure that your build tool works, so that you can make and test incremental changes as you follow the rest of the text.

CPython itself does not come with a tool to build extension modules; it is recommended to use a third-party project for this. In this tutorial, we'll use meson-python. (If you want to use another one, see `first-extension-other-tools`.)

`meson-python` requires defining a "project" using two extra files.

First, add `pyproject.toml` with these contents:

```
[build-system]
build-backend = 'mesonpy'
requires = ['meson-python']

[project]
# Placeholder project information
# (change this before distributing the module)
name = 'sampleproject'
version = '0'
```

Then, create `meson.build` containing the following:

```
project('sampleproject', 'c')

py = import('python').find_installation(pure: false)

py.extension_module(
   'spam',          # name of the importable Python module
   'spammodule.c',  # the C source file
   install: true,
)
```

> [!NOTE] See the meson-python documentation for details on configuration.

Now, build install the *project in the current directory* (`.`) via `pip`:

```
python -m pip -v install .
```

The `-v` (`--verbose`) option causes `pip` to show the output from the compiler, which is often useful during development.

> [!TIP] If you don't have `pip` installed, run `python -m ensurepip`, preferably in a `virtual environment <venv-def>`. (Or, if you prefer another tool that can build and install `pyproject.toml`-based projects, use that.)

Note that you will need to run this command again every time you change your extension. Unlike Python, C has an explicit compilation step.

When your extension is compiled and installed, start Python and try to import it. This should fail with the following exception:

```
>>> import spam
Traceback (most recent call last):
   ...
ImportError: dynamic module does not define module export function (PyModExport_spam or
PyInit_spam)
```

Module export hook

The exception you got when you tried to import the module told you that Python is looking for a "module export function", also known as a `module export hook <extension-export-hook>`. Let's define one.

First, add a prototype below the `#include` lines:

../includes/capi-extension/spammodule-01.c

> [!TIP] The prototype is not strictly necessary, but some modern compilers emit warnings without it. It's generally better to add the prototype than to disable the warning.

The `PyMODEXPORT_FUNC` macro declares the function's return type, and adds any special linkage declarations needed to make the function visible and usable when CPython loads it.

After the prototype, add the function itself. For now, make it return `NULL`:

```
PyMODEXPORT_FUNC
PyModExport_spam(void)
{
   return NULL;
}
```

Compile and load the module again. You should get a different error this time.

```
>>> import spam
Traceback (most recent call last):
   ...
```

```
SystemError: module export hook for module 'spam' failed without setting an exception
```

Simply returning `NULL` is *not* correct behavior for an export hook, and CPython complains about it. That's good -- it means that CPython found the function! Let's now make it do something useful.

The slot table

Rather than `NULL`, the export hook should return the information needed to create a module. Let's start with the basics: the name and docstring.

The information should be defined in an array of `PySlot` entries, which are essentially key-value pairs. Define this array just before your export hook:

```
PyABIInfo_VAR(abi_info);

static PySlot spam_slots[] = {
   PySlot_STATIC_DATA(Py_mod_abi, &abi_info),
   PySlot_STATIC_DATA(Py_mod_name, "spam"),
   PySlot_STATIC_DATA(Py_mod_doc, "A wonderful module with an example function"),
   PySlot_END
};
```

The `PySlot_STATIC_DATA` macro is used when the slot value (here: `&abi_info`, `"spam"`, and the docstring) is a pointer to constant, statically allocated data.

The `PyABIInfo_VAR(abi_info);` macro and the `Py_mod_abi` slot are a bit of boilerplate that helps prevent extensions compiled for a different version of Python from crashing the interpreter.

For both `Py_mod_name` and `Py_mod_doc`, the values are C strings -- that is, NUL-terminated, UTF-8 encoded byte arrays.

Note `PySlot_END` sentinel entry at the end. This marks the end of the array. If you forget it, you'll trigger undefined behavior.

The array is defined as `static` -- that is, not visible outside this `.c` file. This will be a common theme. CPython only needs to access the export hook; all global variables and all other functions should generally be `static`, so that they don't clash with other extensions.

Return this array from your export hook instead of `NULL`:

```
PyMODEXPORT_FUNC
PyModExport_spam(void)
```

```
{
   return spam_slots;
}
```

Now, recompile and try it out:

```
>>> import spam
>>> print(spam)
<module 'spam' from '/home/encukou/dev/cpython/spam.so'>
```

You have an extension module! Try `help(spam)` to see the docstring.

The next step will be adding a function.

Exposing a function

To expose the `system` C function directly to Python, we'll need to write a layer of glue code to convert arguments from Python objects to C values, and the C return value back to Python.

One of the simplest ways to write glue code is a "`METH_O`" function, which takes two Python objects and returns one. All Python objects -- regardless of the Python type -- are represented in C as pointers to the `PyObject` structure.

Add such a function above the slots array:

```
static PyObject *
spam_system(PyObject *self, PyObject *arg)
{
   Py_RETURN_NONE;
}
```

For now, we ignore the arguments, and use the `Py_RETURN_NONE` macro, which expands to a `return` statement that properly returns a Python `None` object.

Recompile your extension to make sure you don't have syntax errors. We haven't yet added `spam_system` to the module, so you might get a warning that `spam_system` is unused.

Method definitions

To expose the C function to Python, you will need to provide several pieces of information in a structure called `PyMethodDef`[^4]:

- `ml_name`: the name of the Python function;
- `ml_doc`: a docstring;
- `ml_meth`: the C function to be called; and
- `ml_flags`: a set of flags describing details like how Python arguments are passed to the C function. We'll use `METH_O` here -- the flag that matches our `spam_system` function's signature.

Because modules typically create several functions, these definitions need to be collected in an array, with a zero-filled sentinel at the end. Add this array just below the `spam_system` function:

../includes/capi-extension/spammodule-01.c

As with module slots, a zero-filled sentinel marks the end of the array.

Next, we'll add the method to the module. Add a `Py_mod_methods` slot to your `PySlot` array:

../includes/capi-extension/spammodule-01.c

Recompile your extension again, and test it. Be sure to restart the Python interpreter, so that `import spam` picks up the new version of the module.

You should now be able to call the function:

```
>>> import spam
>>> print(spam.system)
<built-in function system>
>>> print(spam.system('whoami'))
None
```

Note that our `spam.system` does not yet run the `whoami` command; it only returns `None`.

Check that the function accepts exactly one argument, as specified by the `METH_O` flag:

```
>>> print(spam.system('too', 'many', 'arguments'))
Traceback (most recent call last):
   ...
TypeError: spam.system() takes exactly one argument (3 given)
```

Returning an integer

Now, let's take a look at the return value. Instead of `None`, we'll want `spam.system` to return a number -- that is, a Python `int` object. Eventually this will be the exit code of a system command,

but let's start with a fixed value, say, `3`.

The Python C API provides a function to create a Python `int` object from a C `int` value: `PyLong_FromLong`.[^5]

To call it, replace the `Py_RETURN_NONE` with the following 3 lines:

```
static PyObject *
spam_system(PyObject *self, PyObject *arg)
{
   int status = 3;
   PyObject *result = PyLong_FromLong(status);
   return result;
}
```

Recompile, restart the Python interpreter again, and check that the function now returns 3:

```
>>> import spam
>>> spam.system('whoami')
3
```

Accepting a string

Finally, let's handle the function argument.

Our C function, `!spam_system`, takes two arguments. The first one, `PyObject *self`, will be set to the `spam` module object. This isn't useful in our case, so we'll ignore it.

The other one, `PyObject *arg`, will be set to the object that the user passed from Python. We expect that it should be a Python string. In order to use the information in it, we will need to convert it to a C value -- in this case, a C string (`const char *`).

There's a slight type mismatch here: Python's `str` objects store Unicode text, but C strings are arrays of bytes. So, we'll need to *encode* the data, and we'll use the UTF-8 encoding for it. (UTF-8 might not always be correct for system commands, but it's what `str.encode` uses by default, and the C API has special support for it.)

The function to encode a Python string into a UTF-8 buffer is named `PyUnicode_AsUTF8AndSize`[^6]. Call it like this:

```
static PyObject *
spam_system(PyObject *self, PyObject *arg)
```

```
{
   const char *command = PyUnicode_AsUTF8AndSize(arg, NULL);
   int status = 3;
   PyObject *result = PyLong_FromLong(status);
   return result;
}
```

If `PyUnicode_AsUTF8AndSize` is successful, *command* will point to the resulting C string -- a zero-terminated array of bytes[^7]. This buffer is managed by the *arg* object, which means we don't need to free it, but we must follow some rules:

- We should only use the buffer inside the `spam_system` function. After `spam_system` returns, *arg* and the buffer it manages might be garbage-collected.
- We must not modify it. This is why we use `const`.

If `PyUnicode_AsUTF8AndSize` was *not* successful, it returns a `NULL` pointer. When calling *any* Python C API, we always need to handle such error cases. The way to do this in general is left for later chapters of this documentation. For now, be assured that we are already handling errors from `PyLong_FromLong` correctly.

For the `PyUnicode_AsUTF8AndSize` call, the correct way to handle errors is returning `NULL` from `spam_system`. Add an `if` block for this:

```
static PyObject *
spam_system(PyObject *self, PyObject *arg)
{
   const char *command = PyUnicode_AsUTF8AndSize(arg, NULL);
   if (command == NULL) {
      return NULL;
   }
   int status = 3;
   PyObject *result = PyLong_FromLong(status);
   return result;
}
```

To test that error handling works, compile again, restart Python so that `import spam` picks up the new version of your module, and try passing a non-string value to your function:

```
>>> import spam
>>> spam.system(3)
Traceback (most recent call last):
   ...
```

```
TypeError: bad argument type for built-in operation
```

Now, all that is left is calling the C library function `system` with the `char *` buffer, and using its result instead of the `3`:

```
static PyObject *
spam_system(PyObject *self, PyObject *arg)
{
   const char *command = PyUnicode_AsUTF8AndSize(arg, NULL);
   if (command == NULL) {
      return NULL;
   }
   int status = system(command);
   PyObject *result = PyLong_FromLong(status);
   return result;
}
```

Compile your module, restart Python, and test. This time, you should see your username -- the output of the `whoami` system command:

```
>>> import spam
>>> result = spam.system('whoami')
User Name
>>> result
0
```

You can also test with other commands, like `ls`, `dir`, or one that doesn't exist:

```
>>> import spam
>>> result = spam.system('nonexistent-command')
sh: line 1: nonexistent-command: command not found
>>> result
32512
```

The result

Congratulations! You have written a complete Python C API extension module, and completed this tutorial!

Here is the entire source file, for your convenience:

../includes/capi-extension/spammodule-01.c

Appendix: Other build tools

You should be able to follow this tutorial -- except the *Running your build tool* section itself -- with a build tool other than `meson-python`.

The Python Packaging User Guide has a list of recommended tools; be sure to choose one for the C language.

Workaround for missing PyInit function

If your build tool output complains about missing `PyInit_spam`, add the following function to your module for now:

```
// A workaround
void *PyInit_spam(void) { return NULL; }
```

This is a shim for an old-style `initialization function <extension-export-hook>`, which was required in extension modules for CPython 3.14 and below. Current CPython does not need it, but some build tools may still assume that all extension modules need to define it.

If you use this workaround, you will get the exception `SystemError: initialization of spam failed without raising an exception` instead of `ImportError: dynamic module does not define module export function`.

Compiling directly

Using a third-party build tool is heavily recommended, as it will take care of various details of your platform and Python installation, of naming the resulting extension, and, later, of distributing your work.

If you are building an extension for as *specific* system, or for yourself only, you might instead want to run your compiler directly. The way to do this is system-specific; be prepared for issues you will need to solve yourself.

Linux

On Linux, the Python development package may include a `python3-config` command that prints out the required compiler flags. If you use it, check that it corresponds to the CPython interpreter you'll

use to load the module. Then, start with the following command:

```
gcc --shared $(python3-config --cflags --ldflags) spammodule.c -o spam.so
```

This should generate a `spam.so` file that you need to put in a directory on `sys.path`.

Footnotes

[^1]: `spam` is the favorite food of Monty Python fans...

[^2]: The source file name is entirely up to you, though some tools can be picky about the `.c` extension. This tutorial uses the traditional `*module.c` suffix. Some people would just use `spam.c` to implement a module named `spam`, projects where Python isn't the primary language might use `py_spam.c`, and so on.

[^3]: Including `stdlib.h` is technically not necessary, since `Python.h` includes it and `several other standard headers <capi-system-includes>` for its own use or for backwards compatibility. However, it is good practice to explicitly include what you need.

[^4]: The `!PyMethodDef` structure is also used to create methods of classes, so there's no separate "`!PyFunctionDef`".

[^5]: The name `PyLong_FromLong` might not seem obvious. `PyLong` refers to a the Python `int`, which was originally called `long`; the `FromLong` refers to the C `long` (or `long int`) type.

[^6]: Here, `PyUnicode` refers to the original name of the Python `str` class: `unicode`.

```
The `AndSize` part of the name refers to the fact that this function can also retrieve the size of the buffer, using an output argument. We don't need this, so we set the second argument to NULL.
```

[^7]: We're ignoring the fact that Python strings can also contain NUL bytes, which terminate a C string. In other words, our function will treat `spam.system("foo\0bar")` as `spam.system("foo")`. This possibility can lead to security issues, so the real `os.system` function size checks for this case and raises an error.

□□□ #2
□ □□□ □□□ 2026-10-31 16:57:16 UTC
□ □□□ □□□ 2026-10-31 16:57:20 UTC