2. 自定义扩展类型：教程

Python 允许编写 C 扩展模块定义可以从 Python 代码中操纵的新类型，这很像内置的 `str` 和 `list` 类型。所有扩展类型的代码都遵循一个模式，但是在您开始之前，您需要了解一些细节。这份文件是对这个主题介绍。

2.1. 基础

CPython 运行时会将所有 Python 对象都视为 `PyObject*` 类型的变量，这是所有 Python 对象的"基础类型"。 `PyObject` 结构体本身只包含对象的 reference count 和指向对象的"类型对象"的指针。 这是动作所针对的目标。 类型对象决定解释器要调用哪些 (C) 函数，例如，在对象上查找一个属性，调用一个方法，或者与另一个对象相乘等。 这些 C 函数被称为"类型方法"。

所以，如果你想要定义新的扩展类型，需要创建新的类型对象。

这种事情只能通过例子来解释，下面是一个最小但完整的模块，它在 C 扩展模块 `custom` 中定义了一个名为 `Custom` 的新类型：

> **备注:** 这里展示的方法是定义 *static* 扩展类型的传统方法。可以适合大部分用途。C API也可以定义在堆上分配的扩展类型，使用 `PyType_FromSpec()` 函数，但不在本入门里讨论。

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* 这里添加类型专属的字段。 */
} CustomObject;

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *) &CustomType) < 0) {
        return -1;
    }
```

```
    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    // 使用静态类型时就使用这个
    {Py_mod_multiple_interpreters, Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

这部分很容易理解，这是为了跟上一章能对接上。这个文件定义了三件事：

1. 一个 `Custom` **对象** 包含的东西：这是 `CustomObject` 结构体，它会为每个 `Custom` 实例分配一次。
2. `Custom` **类型** 的行为：这是 `CustomType` 结构体，它定义了一组旗标和函数指针供解释器在收到特定操作请求时进行检查。
3. 如何定义和执行 `custom` 模块：这是 `PyInit_custom` 函数及所关联的用于定义该模块的 `custom_module` 结构体，以及用于设置新模块对象的 `custom_module_exec` 函数。

结构的第一块是

```
typedef struct {
    PyObject_HEAD
} CustomObject;
```

这就是一个自定义对象将会包含的内容。 `PyObject_HEAD` 是强制要求放在每个对象结构体之前并定义一个名为 `ob_base` 的 `PyObject` 类型的字段，其中包含一个指向类型对象和引用计数的指针（这两者可以分别使用宏 `Py_TYPE` 和 `Py_REFCNT` 来区分）。 使用宏的理由是将布局抽象出来并在 调试编译版中 中启用附加字段。

> **备注：** 注意在宏 `PyObject_HEAD` 后没有分号。意外添加分号会导致编译器提示出错。

当然，对象除了在 `PyObject_HEAD` 存储数据外，还有额外数据；例如，如下定义了标准的Python浮点数:

```
typedef struct {
    PyObject_HEAD
```

```
    double ob_fval;
} PyFloatObject;
```

第二个位是类型对象的定义:

```
static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};
```

备注: 推荐使用如上C99风格的初始化，以避免列出所有的 `PyTypeObject` 字段，其中很多是你不需要关心的，这样也可以避免关注字段的定义顺序。

在 `object.h` 中实际定义的 `PyTypeObject` 具有比如上定义更多的 字段。 剩余的字段会由 C 编译器用零来填充，通常的做法是不显式地指定它们，除非你确实需要它们。

我们先挑选一部分，每次一个字段:

```
.ob_base = PyVarObject_HEAD_INIT(NULL, 0)
```

这一行是强制的样板，用以初始化如上提到的 `ob_base` 字段:

```
.tp_name = "custom.Custom",
```

我们的类型的名称。 这将出现在我们的对象的默认文本表示形式和某些错误消息中，例如:

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Custom") to str
```

请注意此名称是一个带点号名称，它同时包括模块名称和模块中的类型名称。 本例中的模块是 `custom` 而类型是 `Custom`，因此我们将类型名称设为 `custom.Custom`。 使用真正的带点号的导入路径对于使你的类型与 `pydoc` 和 `pickle` 模块保持兼容是很重要的。

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

这样能让 Python 知道当创建新的 `Custom` 实例时需要分配多少内存。 `tp_itemsize` 仅用于可变大小的对象而在其他情况下都应为零。

备注: 如果你希望你的类型可在 Python 中被子类化，并且你的类型和它的基类型具有相同的 `tp_basicsize`，那么你可能会遇到多重继承问题。 你的类型的 Python 子类必须在其 `__bases__` 中将你的类型列在最前面，否则在调用你的类型的 `__new__()` 方法时将会出错。 你可以通过确

保你的类型具有比它的基类型更大的 `tp_basicsize` 值来避免这个问题。 在大多数时候，这都是可以的，因为要么你的类型是 `object`，要么你将为你的基类型添加数据成员，从而增加其大小。

我们将类旗标设为 `Py_TPFLAGS_DEFAULT`。

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

所有类型都应当在它们的旗标中包括此常量。 该常量将启用至少在 Python 3.3 之前定义的全部成员。 如果你需要更多的成员，你将需要对相应的旗标进行 OR 运算。

我们为 `tp_doc` 类型提供一个文档字符串.

```
.tp_doc = PyDoc_STR("Custom objects"),
```

要启用对象创建，我们必须提供一个 `tp_new` 处理器。 这等价于 Python 方法 `__new__()`，但必须显式地指定。 在这种情况下，我们可以使用 API 函数 `PyType_GenericNew()` 所提供的默认实现。

```
.tp_new = PyType_GenericNew,
```

除了 `custom_module_exec()` 中的某些代码以外，文件中的其他内容应该都很常见:

```
if (PyType_Ready(&CustomType) < 0) {
    return -1;
}
```

这将初始化 `Custom` 类型，为一些成员填充适当的默认值，包括我们在初始时设为 `NULL` 的 `ob_type`。

```
if (PyModule_AddObjectRef(m, "Custom", (PyObject *) &CustomType) < 0) {
    return -1;
}
```

这将把类型添加到模块字典中。 这样我们就能通过调用 `Custom` 类来创建 `Custom` 实例:

```
>>> import custom
>>> mycustom = custom.Custom()
```

就是这样！ 剩下的工作就是编译它；将上述代码放入名为 `custom.c` 的文件中，

```
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"

[project]
name = "custom"
version = "1"
```

名为 `pyproject.toml` 的文件中，并且

```
from setuptools import Extension, setup
setup(ext_modules=[Extension("custom", ["custom.c"])])
```

在名为 `setup.py` 的文件中；然后输入

```
$ python -m pip install .
```

在 shell 中应该会在子目录下产生一个文件 `custom.so` 并安装它；现在启动 Python --- 你应当能够执行 `import custom` 并尝试使用 `Custom` 对象。

这并不难，对吗?

当然，当前的自定义类型非常无趣。它没有数据，也不做任何事情。它甚至不能被子类化。

2.2. 向基本示例添加数据和方法

让我们通过添加一些数据和方法来扩展这个基本示例。 让我们再使该类型可以作为基类使用。 我们将创建一个新模块 `custom2` 来添加这些功能：

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free(self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
```

```
    }
    return (PyObject *) self;
}

static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        Py_XSETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_XSETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"first", Py_T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", Py_T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    CustomObject *self = (CustomObject *) op;
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
```

```
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = Custom_init,
    .tp_dealloc = Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *) &CustomType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    {Py_mod_multiple_interpreters, Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom2",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom2(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

该模块的新版本包含多处修改。

现在 `Custom` 类型的 C 结构体中有三个数据属性，*first*、*last* 和 *number*。 其中 *first* 和 *last* 变量是包含名字和姓氏的 Python 字符串。 *number* 属性是一个 C 整数。

对象的结构将被相应地更新:

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;
```

因为现在我们有数据需要管理，我们必须更加小心地处理对象的分配和释放。 至少，我们需要有一个释放方法:

```
static void
Custom_dealloc(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free(self);
}
```

它会被赋值给 tp_dealloc 成员:

```
.tp_dealloc = Custom_dealloc,
```

此方法会先清空两个 Python 属性的引用计数。 Py_XDECREF() 可以正确处理参数为 NULL 的情况（这可能在 tp_new 中途失败时发生）。 随后它将调用对象类型的 tp_free 成员（通过 Py_TYPE(self) 计算得到）来释放对象的内存。 请注意对象类型可以不是 CustomType，因为对象可能是一个子类的实例。

备注: 上面需要显式强制转换至 CustomObject * 是因为我们定义了 Custom_dealloc 接受一个 PyObject * 参数，由于 tp_dealloc 函数指针预期接受一个 PyObject * 参数。 通过向 tp_dealloc 槽位分配一个类型，我们声明它被调用时只能附带我们的 CustomObject 类的实例，因此强制转换至 (CustomObject *) 是安全的。 这就是 C 语言中面向对象的多态性!

在现有代码中，或是在本教程的之前版本中，你可能会看到类似的函数接受一个直接指向子类型对象结构体 (CustomObject*) 的指针，就像这样:

```
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
...
.tp_dealloc = (destructor) Custom_dealloc,
```

这将在所有 CPython 支持的架构上做同样的事，但是根据 C 标准，它会唤起未定义的行为。

我们希望确保头一个和末一个名称被初始化为空字符串，因此我们提供了一个 tp_new 实现:

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
```

```
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}
```

并在 tp_new 成员中安装它:

```
.tp_new = Custom_new,
```

tp_new 处理器负责创建（而不是初始化）该类型的对象。 它在 Python 中被暴露为 __new__() 方法。 它不需要定义 tp_new 成员，实际上许多扩展类型会简单地重用 PyType_GenericNew()，就像上面 Custom 类型的第一个版本所做的那样。 在此情况下，我们使用 tp_new 处理器来将 first 和 last 属性初始化为非 NULL 的默认值。

tp_new 将接受被实例化的类型（不要求为 CustomType，如果被实例化的是一个子类）以及在该类型被调用时传入的任何参数，并预期返回所创建的实例。 tp_new 处理器总是接受位置和关键字参数，但它们总是会忽略这些参数，而将参数处理留给初始化（即 C 中的 tp_init 或 Python 中的 __init__ 函数）方法来执行。

备注: tp_new 不应显式地调用 tp_init，因为解释器会自行调用它。

tp_new 实现会调用 tp_alloc 槽位来分配内存:

```
self = (CustomObject *) type->tp_alloc(type, 0);
```

由于内存分配可能会失败，我们必须在继续执行之前检查 tp_alloc 结果确认其不为 NULL。

备注: 我们没有自行填充 tp_alloc 槽位。 而是由 PyType_Ready() 通过从我们的基类继承来替我们填充它，其中默认为 object。 大部分类型都是使用默认的分配策略。

备注: 如果您要创建一个协作式 tp_new (它会调用基类型的 tp_new 或 __new__())，那么你 *不能* 在运行时尝试使用方法解析顺序来确定要调用的方法。 必须总是静态地确定你要调用的类型，并直接调用它的 tp_new，或是通过 type->tp_base->tp_new。 如果你不这样做，你的类型的同样继承自其它由 Python 定义的类的 Python 子类可能无法正常工作。 (具体地说，你可能无法创建这样的子类的实例而是会引发 TypeError。)

我们还定义了一个接受参数来为我们的实例提供初始值的初始化函数:

```
static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
```

```
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}
```

通过填充 tp_init 槽位。

```
.tp_init = Custom_init,
```

tp_init 槽位在 Python 中暴露为 __init__() 方法。 它被用来在创建对象后对其进行初始化。 初始化器总是接受位置和关键字参数，它们应当在成功时返回 0 而在出错时返回 -1。

不同于 tp_new 处理器，tp_init 不保证一定会被调用 (例如，在默认情况下 pickle 模块不会在未解封的实例上调用 __init__())。 它还可能被多次调用。 任何人都可以在我们的对象上调用 __init__() 方法。 因此，我们在为属性赋新值时必须格外小心。 例如像这样给 first 成员赋值:

```
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

但是这可能会有风险。 我们的类型没有限制 first 成员的类型，因此它可以是任何种类的对象。 它可以带有一个会执行尝试访问 first 成员的代码的析构器；或者该析构器可能会释放 线程状态 并让任意代码在其他线程中运行来访问和修改我们的对象。

为了保持谨慎并使我们避免这种可能性，我们几乎总是要在减少成员的引用计数之前给它们重新赋值。 什么时候我们可以不必再这样做?

- 当我们明确知道引用计数大于 1 的时候;
- 当我们知道对象的销毁 [1] 既不会释放 线程状态 也不会导致任何对我们的类型的代码的回调的时候;
- 当减少一个 tp_dealloc 处理器内不支持循环垃圾回收的类型的引用计数的时候 [2].

我们可能会想将我们的实例变量暴露为属性。 有几种方式可以做到这一点。 最简单的方式是定义成员的定义:

```
static PyMemberDef Custom_members[] = {
    {"first", Py_T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", Py_T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};
```

并将定义放置到 tp_members 槽位中:

```
.tp_members = Custom_members,
```

每个成员的定义都有成员名称、类型、偏移量、访问旗标和文档字符串。 请参阅下面的 泛型属性管理 小节来了解详情。section below for details.

此方式的缺点之一是它没有提供限制可被赋值给 Python 属性的对象类型的办法。 我们预期 first 和 last 的名称为字符串，但它们可以被赋值为任意 Python 对象。 此外，这些属性还可以被删除，并将 C 指针设为 NULL。 即使我们可以保证这些成员被初始化为非 NULL 值，如果这些属性被删除这些成员仍可被设为 NULL。

我们定义一个单独的方法，Custom.name()，它将对象名称输出为 first 和 last 名称的拼接。

```
static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    CustomObject *self = (CustomObject *) op;
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

该方法以的实现形式是一个接受 Custom (或 Custom 的子类) 实例作为第一个参数的 C 函数。 方法总是接受一个实例作为第一个参数。 方法往往也接受位置和关键字参数，但在本例中我们未接受任何参数也不需要接受位置参数元组或关键字参数字典。 该方法等价于以下 Python 方法:

```
def name(self):
    return "%s %s" % (self.first, self.last)
```

请注意我们必须检查 first 和 last 成员是否可能为 NULL。 这是因为它们可以被删除，在此情况下它们会被设为 NULL。 更好的做法是防止删除这些属性并将属性的值限制为字符串。 我们将在下一节了解如何做到这一点。

现在我们已经定义好了方法，我们需要创建一个方法定义数组:

```
static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};
```

（请注意我们使用了 `METH_NOARGS` 旗标来指明该方法不准备接受除 *self* 以外的任何参数）

并将其赋给 `tp_methods` 槽位:

```
.tp_methods = Custom_methods,
```

最后，我们将使我们的类型可被用作派生子类的基类。 我们精心地编写我们的方法以便它们不会随意假定被创建或使用的对象类型，所以我们需要做的就是将 `Py_TPFLAGS_BASETYPE` 添加到我们的类旗标定义中:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
```

我们将 `PyInit_custom()` 重命名为 `PyInit_custom2()`，更新 `PyModuleDef` 结构体中的模块名称，并更新 `PyTypeObject` 结构体中的完整类名。

最后，我们更新 `setup.py` 文件来包括新的模块，

```
from setuptools import Extension, setup
setup(ext_modules=[
    Extension("custom", ["custom.c"]),
    Extension("custom2", ["custom2.c"]),
])
```

然后我们重新安装以便能够 `import custom2`:

```
$ python -m pip install .
```

2.3. 提供对于数据属性的更精细控制

在本节中，我们将对 `Custom` 示例中 `first` 和 `last` 属性的设置进行更精细的控制。 在我们上一版本的模块中，实例变量 `first` 和 `last` 可以被设为非字符串值甚至被删除。 我们希望确保这些属性始终包含字符串。

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;
```

```c
static void
Custom_dealloc(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free(self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        Py_SETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_SETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_getfirst(PyObject *op, void *closure)
```

```
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->first);
}

static int
Custom_setfirst(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    Py_SETREF(self->first, Py_NewRef(value));
    return 0;
}

static PyObject *
Custom_getlast(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->last);
}

static int
Custom_setlast(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    Py_SETREF(self->last, Py_NewRef(value));
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", Custom_getfirst, Custom_setfirst,
     "first name", NULL},
    {"last", Custom_getlast, Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    CustomObject *self = (CustomObject *) op;
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

```
static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = Custom_init,
    .tp_dealloc = Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *) &CustomType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    {Py_mod_multiple_interpreters, Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom3",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom3(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

为了更好地控制 `first` 和 `last` 属性，我们将使用自定义的读取器和设置器函数。 以下就是用于读取和设置 `first` 属性的函数:

```
static PyObject *
Custom_getfirst(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}
```

读取器函数接受一个 Custom 对象和一个“闭包”，后者是一个空指针。 在本例中，该闭包将被忽略。 （闭包支持将定义数据传递给 读取器和设置器的高级用法。 例如，这可以被用来允许一组获取器和设置器函数根据闭包中的数据来决定要读取或设置的属性）。

设置器函数接受传入 Custom 对象、新值和闭包。 新值可能为 `NULL`，在这种情况下属性将被删除。 在我们的设置器中，如果属性被删除或者如果其新值不是字符串则会引发一个错误。

我们创建一个 PyGetSetDef 结构体的数组:

```
static PyGetSetDef Custom_getsetters[] = {
    {"first", Custom_getfirst, Custom_setfirst,
     "first name", NULL},
    {"last", Custom_getlast, Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};
```

并在 tp_getset 槽位中注册它:

```
.tp_getset = Custom_getsetters,
```

在 PyGetSetDef 结构体中的最后一项是上面提到的“闭包”。 在本例中，我们没有使用闭包，因此我们只传入 `NULL`。

我们还移除了这些属性的成员定义:

```
static PyMemberDef Custom_members[] = {
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};
```

我们还需要将 tp_init 处理器更新为只允许传入字符串 [3]:

```
static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}
```

通过这些更改，我们能够确保 `first` 和 `last` 成员一定不为 `NULL` 以便我们能在几乎所有情况下移除 `NULL` 值检查。 这意味着大部分 `Py_XDECREF()` 调用都可以被转换为 `Py_DECREF()` 调用。 我们不能更改这些调用的唯一场合是在 `tp_dealloc` 实现中，那里这些成员的初始化有可能在 `tp_new` 中失败。

我们还重命名了模块初始化函数和初始化函数中的模块名称，就像我们之前所做的一样，我们还向 `setup.py` 文件添加了一个额外的定义。

2.4. 支持循环垃圾回收

Python 具有一个可以标识不再需要的对象的 循环垃圾回收器 (GC) 即使它们的引用计数并不为零。 这种情况会在对象被循环引用时发生。 例如，设想:

```
>>> l = []
>>> l.append(l)
>>> del l
```

在这个例子中，我们创建了一个包含其自身的列表。 当我们删除它的时候，它将仍然具有一个来自其本身的引用。 它的引用计数并未降为零。 幸运的是，Python 的循环垃圾回收器将最终发现该列

表是无用的垃圾并释放它。

在 Custom 示例的第二个版本中，我们允许任意类型的对象存储到 first 或 last 属性中 [4]。 此外，在第二和第三个版本中，我们还允许子类化 Custom，并且子类可以添加任意属性。 出于这两个原因中的任何一个，Custom 对象都可以加入循环：

```
>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n
```

要允许一个加入引用循环的 Custom 实例能被循环 GC 正确检测和收集，我们的 Custom 类型需要填充两个额外的槽位并增加一个旗标来启用这些槽位：

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static int
Custom_traverse(PyObject *op, visitproc visit, void *arg)
{
    CustomObject *self = (CustomObject *) op;
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}

static int
Custom_clear(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(PyObject *op)
{
    PyObject_GC_UnTrack(op);
    (void)Custom_clear(op);
    Py_TYPE(op)->tp_free(op);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
```

```
    if (self != NULL) {
        self->first = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        Py_SETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_SETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_getfirst(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->first);
}

static int
Custom_setfirst(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
```

```
                        "The first attribute value must be a string");
        return -1;
    }
    Py_XSETREF(self->first, Py_NewRef(value));
    return 0;
}

static PyObject *
Custom_getlast(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->last);
}

static int
Custom_setlast(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    Py_XSETREF(self->last, Py_NewRef(value));
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", Custom_getfirst, Custom_setfirst,
     "first name", NULL},
    {"last", Custom_getlast, Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    CustomObject *self = (CustomObject *) op;
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

```
    .tp_new = Custom_new,
    .tp_init = Custom_init,
    .tp_dealloc = Custom_dealloc,
    .tp_traverse = Custom_traverse,
    .tp_clear = Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *) &CustomType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    {Py_mod_multiple_interpreters, Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

首先，遍历方法让循环 GC 知道能够参加循环的子对象:

```
static int
Custom_traverse(PyObject *op, visitproc visit, void *arg)
{
    CustomObject *self = (CustomObject *) op;
    int vret;
    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
```

```
            return vret;
    }
    return 0;
}
```

对于每个可以加入循环的子对象，我们都需要调用 `visit()` 函数，它会被传递给遍历方法。 `visit()` 函数接受该子对象和传递给遍历方法的额外参数 *arg* 作为其参数。 它返回一个在其为非零值时必须被返回的整数值。

Python 提供了一个可自动调用 visit 函数的 `Py_VISIT()` 宏。 使用 `Py_VISIT()`，我们可以最小化 `Custom_traverse` 中的准备工作量:

```
static int
Custom_traverse(PyObject *op, visitproc visit, void *arg)
{
    CustomObject *self = (CustomObject *) op;
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

备注: `tp_traverse` 实现必须将其参数准确命名为 *visit* 和 *arg* 以便使用 `Py_VISIT()`。

第二，我们需要提供一个方法用来清除任何可以参加循环的子对象:

```
static int
Custom_clear(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

请注意 `Py_CLEAR()` 宏的使用。 它是清除任意类型的数据属性并减少其引用计数的推荐的且安全的方式。 如果你要选择在将属性设为 `NULL` 之间在属性上调用 `Py_XDECREF()`，则属性的析构器有可能会回调再次读取该属性的代码 (*特别是* 如果存在引用循环的话)。

备注: 你可以通过以下写法来模拟 `Py_CLEAR()`:

```
PyObject *tmp;
tmp = self->first;
self->first = NULL;
Py_XDECREF(tmp);
```

无论如何，在删除属性时始终使用Nevertheless, it is much easier and less error-prone to always use `Py_CLEAR()` 都是更简单且更不易出错的。 请不要尝试以健壮性为代价的微小优化!

释放器 `Custom_dealloc` 可能会在清除属性时调用任意代码。 这意味着循环 GC 可以在函数内部被触发。 由于 GC 预期引用计数不为零，我们需要通过调用 `PyObject_GC_UnTrack()` 来让 GC 停止追

踪相关的对象。 下面是我们使用 PyObject_GC_UnTrack() 和 Custom_clear 重新实现的释放器:

```
static void
Custom_dealloc(PyObject *op)
{
    PyObject_GC_UnTrack(op);
    (void)Custom_clear(op);
    Py_TYPE(op)->tp_free(op);
}
```

最后，我们将 Py_TPFLAGS_HAVE_GC 旗标添加到类旗标中:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

这样就差不多了。 如果我们编写了自定义的 tp_alloc 或 tp_free 处理器，则我们需要针对循环垃圾回收来修改它。 大多数扩展都将使用自动提供的版本。

2.5. 子类化其他类型

创建派生自现有类型的新类型是有可能的。 最容易的做法是从内置类型继承，因为扩展可以方便地使用它所需要的 PyTypeObject。 在不同扩展模块之间共享这些 PyTypeObject 结构体则是困难的。

在本例中我们将创建一个继承自内置 list 类型的 SubList 类型。 这个新类型将完全兼容常规列表，但将拥有一个额外的 increment() 方法用于递增内部计数器的值:

```
>>> import sublist
>>> s = sublist.SubList(range(3))
>>> s.extend(s)
>>> print(len(s))
6
>>> print(s.increment())
1
>>> print(s.increment())
2
```

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

typedef struct {
    PyListObject list;
    int state;
} SubListObject;

static PyObject *
SubList_increment(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    SubListObject *self = (SubListObject *) op;
    self->state++;
    return PyLong_FromLong(self->state);
}

static PyMethodDef SubList_methods[] = {
    {"increment", SubList_increment, METH_NOARGS,
```

```
     PyDoc_STR("increment state counter")},
    {NULL},
};

static int
SubList_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    SubListObject *self = (SubListObject *) op;
    if (PyList_Type.tp_init(op, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}

static PyTypeObject SubListType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist.SubList",
    .tp_doc = PyDoc_STR("SubList objects"),
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_init = SubList_init,
    .tp_methods = SubList_methods,
};

static int
sublist_module_exec(PyObject *m)
{
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "SubList", (PyObject *) &SubListType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot sublist_module_slots[] = {
    {Py_mod_exec, sublist_module_exec},
    {Py_mod_multiple_interpreters, Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef sublist_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = sublist_module_slots,
};

PyMODINIT_FUNC
PyInit_sublist(void)
{
    return PyModuleDef_Init(&sublist_module);
}
```

如你所见，此源代码与之前几节中的 Custom 示例非常相似。 我们将逐一分析它们之间的主要区别。

```
typedef struct {
    PyListObject list;
    int state;
} SubListObject;
```

派生类型对象的主要差异在于基类型的对象结构体必须是第一个值。 基类型将已经在其结构体的开头包括了 PyObject_HEAD()。

当一个 Python 对象是 SubList 的实例时，它的 PyObject * 指针可以被安全地强制转换为 PyListObject * 和 SubListObject *:

```
static int
SubList_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    SubListObject *self = (SubListObject *) op;
    if (PyList_Type.tp_init(op, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}
```

我们可以在上面看到如何将调用传递到基类型的 __init__() 方法。

这个模式在编写具有自定义 tp_new 和 tp_dealloc 成员的类型时很重要。 tp_new 处理器不应为具有 tp_alloc 的对象实际分配内存，而是让基类通过调用自己的 tp_new 来处理它。

PyTypeObject 结构体支持用 tp_base 指定类型的实体基类。 由于跨平台编译器的问题，你无法以对 PyList_Type 的引用来直接填充该字段；它应当在 Py_mod_exec 函数中完成:

```
static int
sublist_module_exec(PyObject *m)
{
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "SubList", (PyObject *) &SubListType) < 0) {
        return -1;
    }

    return 0;
}
```

在调用 PyType_Ready() 之前，类型结构体必须已经填充 tp_base 槽位。 当我们从现有类型派生时，它不需要将 tp_alloc 槽位填充为 PyType_GenericNew() -- 来自基类型的分配函数将会被继承。

在那之后，调用 PyType_Ready() 并将类型对象添加到模块中的过程与基本的 Custom 示例是一样的。

脚注

[1] 当我们知道该对象属于基本类型，如字符串或浮点数时情况就是如此。

[2] 在本示例中我们需要 `tp_dealloc` 处理器中的这一机制，因为我们的类型不支持垃圾回收。

[3] 现在我们知道 first 和 last 成员都是字符串，因此也许我们可以对减少它们的引用计数不必太过小心，但是，我们还接受字符串子类的实例。 即使释放普通字符串不会对我们的对象执行回调，我们也不能保证释放一个字符串子类的实例不会对我们的对象执行回调。

[4] 而且，即使是将我们的属性限制为字符串实例，用户还是可以传入任意 `str` 子类因而仍能造成引用循环。