Ten thousand meters

Diving deep, flying high to see why

about blog materials

Python behind the scenes #6: how Python object system works

Published: Fri 04 December 2020
By Victor Skvortsov
tags: Python behind the scenes Python CPython

As we know from the previous parts of this series, the execution of a Python program consists of two major steps:

1. The CPython compiler translates Python code to bytecode.
2. The CPython VM executes the bytecode.

We've been focusing on the second step for quite a while. In part 4 we've looked at the evaluation loop, a place where Python bytecode gets executed. And in part 5 we've studied how the VM executes the instructions that are used to implement variables. What we haven't covered yet is how the VM actually computes something. We postponed this question because to answer it, we first need to understand how the most fundamental part of the language works. Today, we'll study the Python object system.

Note: In this post I'm referring to CPython 3.9. Some implementation details will certainly change as CPython evolves. I'll try to keep track of important changes and add update notes.

Motivation

Consider an extremely simple piece of Python code:

```
def f(x):
    return x + 7
```

To compute the function `f`, CPython must evaluate the expression `x + 7`. The question I'd like to ask is: How does CPython do that? Special methods such as `__add__()` and `__radd__()` probably come to your mind. When we define these methods on a class, the instances of that class can be added using the `+` operator. So, you might think that CPython does something like this:

1. It calls `x.__add__(7)` or `type(x).__add__(x, 7)`.
2. If `x` doesn't have `__add__()`, or if this method fails, it calls `(7).__radd__(x)` or `int.__radd__(7, x)`.

The reality, tough, is a bit more complicated. What really happens depends on what `x` is. For example, if `x` is an instance of a user-defined class, the algorithm described above resembles the truth. If, however, `x` is an instance of a built-in type, like `int` or `float`, CPython doesn't call any special methods at all.

To learn how some Python code is executed, we can do the following:

1. Disassemble the code into bytecode.

2. Study how the VM executes the disassembled bytecode instructions.

Let's apply this algorithm to the function `f`. The compiler translates the body of this function to the following bytecode:

```
$ python -m dis f.py
...
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (7)
              4 BINARY_ADD
              6 RETURN_VALUE
```

And here's what these bytecode instructions do:

1. `LOAD_FAST` loads the value of the parameter `x` onto the stack.
2. `LOAD_CONST` loads the constant `7` onto the stack.
3. `BINARY_ADD` pops two values from the stack, adds them and pushes the result back onto the stack.
4. `RETURN_VALUE` pops the value from the stack and returns it.

How does the VM add two values? To answer this question, we need to understand what these values are. For us, `7` is an instance of `int` and `x` is, well, anything. For the VM, though, everything is a Python object. All values the VM pushes onto the stack and pops from the stack are pointers to `PyObject` structs (hence the phrase "Everything in Python is an object").

The VM doesn't need to know how to add integers or strings, that is, how to do the arithmetic or concatenate sequences. All it needs to know is that every Python object has a type. A type, in turn, knows everything about its objects. For example, the `int` type knows how to add integers, and the `float` type knows how to add floats. So, the VM asks the type to perform the operation.

This simplified explanation captures the essence of the solution, but it also omits a lot of important details. To get a more realistic picture, we need to understand what Python objects and types really are and how they work.

Python objects and types

We've discussed Python objects a little in part 3. This discussion is worth repeating here.

We begin with the definition of the `PyObject` struct:

```c
typedef struct _object {
    _PyObject_HEAD_EXTRA // macro, for debugging purposes only
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;
```

It has two members:

- a reference count `ob_refcnt` that CPython uses for garbage collection; and
- a pointer to the object's type `ob_type`.

We said that the VM treats any Python object as `PyObject`. How is that possible? The C programming language has no notion of classes and inheritance. Nevertheless, it's possible to implement in C something that can be called a single inheritance. The C standard states that a pointer to any struct can be converted to a pointer to its first member and vice versa. So, we can "extend" `PyObject` by defining a new struct whose first member is `PyObject`.

Here's, for example, how the `float` object is defined:

```c
typedef struct {
    PyObject ob_base; // expansion of PyObject_HEAD macro
    double ob_fval;
} PyFloatObject;
```

A `float` object stores everything `PyObject` stores plus a floating-point value `ob_fval`. The C standard simply states that we can convert a pointer to `PyFloatObject` to a pointer to `PyObject` and vice versa:

```c
PyFloatObject float_object;
// ...
PyObject *obj_ptr = (PyObject *)&float_object;
PyFloatObject *float_obj_ptr = (PyFloatObject *)obj_ptr;
```

The reason why the VM treats every Python object as `PyObject` is because all it needs to access is the object's type. A type is also a Python object, an instance of the `PyTypeObject` struct:

```c
// PyTypeObject is a typedef for "struct _typeobject"

struct _typeobject {
    PyVarObject ob_base; // expansion of PyObject_VAR_HEAD macro
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;
```

```
    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;
};
```

By the way, note that the first member of a type is not `PyObject` but `PyVarObject`, which is defined as follows:

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;
```

Nevertheless, since the first member of `PyVarObject` is `PyObject`, a pointer to a type can still be converted to a pointer to `PyObject`.

So, what is a type and why does it have so many members? A type determines how the objects of that type behave. Each member of a type, called slot, is responsible for a particular aspect of the object's behavior. For example:

- `tp_new` is a pointer to a function that creates new objects of the type.
- `tp_str` is a pointer to a function that implements `str()` for objects of the type.
- `tp_hash` is a pointer to a function that implements `hash()` for objects of the type.

Some slots, called sub-slots, are grouped together in suites. A suite is just a struct that contains related slots. For example, the `PySequenceMethods` struct is a suite of sub-slots that implement the sequence protocol:

```
typedef struct {
    lenfunc sq_length;
    binaryfunc sq_concat;
    ssizeargfunc sq_repeat;
    ssizeargfunc sq_item;
    void *was_sq_slice;
    ssizeobjargproc sq_ass_item;
    void *was_sq_ass_slice;
    objobjproc sq_contains;

    binaryfunc sq_inplace_concat;
    ssizeargfunc sq_inplace_repeat;
} PySequenceMethods;
```

If you count all the slots and sub-slots, you'll get a scary number. Fortunately, each slot is very well documented in the Python/C API Reference Manual (I strongly recommend you to bookmark this link). Today we'll cover only a few slots. Nevertheless, it shall give us a general idea of how slots are used.

Since we're interested in how CPython adds objects, let's find the slots responsible for addition. There must be at least one such slot. After careful inspection of the `PyTypeObject` struct, we find that it has the "number" suite `PyNumberMethods`, and the first slot of this suite is a binary function called `nd_add`:

```
typedef struct {
    binaryfunc nb_add; // typedef PyObject * (*binaryfunc)(PyObject *, PyObject *)
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    // ... more sub-slots
} PyNumberMethods;
```

It seems that the `nb_add` slot is what we're looking for. Two questions naturally arise regarding this slot:

- What is it set to?
- How is it used?

I think it's better to start with the second. We should expect that the VM calls `nb_add` to execute the `BINARY_ADD` opcode. So, let's, for a moment, suspend our discussion about types and take a look at how the `BINARY_ADD` opcode is implemented.

BINARY_ADD

Like any other opcode, `BINARY_ADD` is implemented in the evaluation loop in `Python/ceval.c`:

```
case TARGET(BINARY_ADD): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    /* NOTE(haypo): Please don't try to micro-optimize int+int on
        CPython using bytecode, it is simply worthless.
        See http://bugs.python.org/issue21955 and
        http://bugs.python.org/issue10044 for the discussion. In short,
        no patch shown any impact on a realistic benchmark, only a minor
        speedup on microbenchmarks. */
    if (PyUnicode_CheckExact(left) &&
            PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(tstate, left, right, f, next_instr);
        /* unicode_concatenate consumed the ref to left */
    }
```

```
        else {
            sum = PyNumber_Add(left, right);
            Py_DECREF(left);
        }
        Py_DECREF(right);
        SET_TOP(sum);
        if (sum == NULL)
            goto error;
        DISPATCH();
    }
```

This code requires some comments. We can see that it calls `PyNumber_Add()` to add two objects, but if the objects are strings, it calls `unicode_concatenate()` instead. Why so? This is an optimization. Python strings seem immutable, but sometimes CPython mutates a string and thus avoids creating a new string. Consider appending one string to another:

```
output += some_string
```

If the `output` variable points to a string that has no other references, it's safe to mutate that string. This is exactly the logic that `unicode_concatenate()` implements.

It might be tempting to handle other special cases in the evaluation loop as well and optimize, for example, integers and floats. The comment explicitly warns against it. The problem is that a new special case comes with an additional check, and this check is only useful when it succeeds. Otherwise, it may have a negative effect on performance.

After this little digression, let's look at `PyNumber_Add()`:

```
PyObject *
PyNumber_Add(PyObject *v, PyObject *w)
{
    // NB_SLOT(nb_add) expands to "offsetof(PyNumberMethods, nb_add)"
    PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));
    if (result == Py_NotImplemented) {
        PySequenceMethods *m = Py_TYPE(v)->tp_as_sequence;
        Py_DECREF(result);
        if (m && m->sq_concat) {
            return (*m->sq_concat)(v, w);
        }
        result = binop_type_error(v, w, "+");
    }
    return result;
}
```

I suggest to step into `binary_op1()` straight away and figure out what the rest of `PyNumber_Add()` does later:

```
static PyObject *
binary_op1(PyObject *v, PyObject *w, const int op_slot)
{
    PyObject *x;
    binaryfunc slotv = NULL;
    binaryfunc slotw = NULL;

    if (Py_TYPE(v)->tp_as_number != NULL)
        slotv = NB_BINOP(Py_TYPE(v)->tp_as_number, op_slot);
    if (!Py_IS_TYPE(w, Py_TYPE(v)) &&
        Py_TYPE(w)->tp_as_number != NULL) {
        slotw = NB_BINOP(Py_TYPE(w)->tp_as_number, op_slot);
        if (slotw == slotv)
            slotw = NULL;
    }
    if (slotv) {
```

```
        if (slotw && PyType_IsSubtype(Py_TYPE(w), Py_TYPE(v))) {
            x = slotw(v, w);
            if (x != Py_NotImplemented)
                return x;
            Py_DECREF(x); /* can't do it */
            slotw = NULL;
        }
        x = slotv(v, w);
        if (x != Py_NotImplemented)
            return x;
        Py_DECREF(x); /* can't do it */
    }
    if (slotw) {
        x = slotw(v, w);
        if (x != Py_NotImplemented)
            return x;
        Py_DECREF(x); /* can't do it */
    }
    Py_RETURN_NOTIMPLEMENTED;
}
```

The `binary_op1()` function takes three parameters: the left operand, the right operand and an offset that identifies the slot. Types of both operands can implement the slot. Therefore, `binary_op1()` looks up both implementations. To calculate the result, it calls one implementation or another relying on the following logic:

1. If the type of one operand is a subtype of another, call the slot of the subtype.
2. If the left operand doesn't have the slot, call the slot of the right operand.
3. Otherwise, call the slot of the left operand.

The reason to prioritize the slot of a subtype is to allow the subtypes to override the behavior of their ancestors:

```
$ python -q
>>> class HungryInt(int):
...     def __add__(self, o):
...             return self
...
>>> x = HungryInt(5)
>>> x + 2
5
>>> 2 + x
7
>>> HungryInt.__radd__ = lambda self, o: self
>>> 2 + x
5
```

Let's turn back to `PyNumber_Add()`. If `binary_op1()` succeeds, `PyNumber_Add()` simply returns the result of `binary_op1()`. If, however, `binary_op1()` returns the `NotImplemented` constant, which means that the operation cannot be performed for a given combination of types, `PyNumber_Add()` calls the `sq_concat` "sequence" slot of the first operand and returns the result of this call:

```
PySequenceMethods *m = Py_TYPE(v)->tp_as_sequence;
if (m && m->sq_concat) {
    return (*m->sq_concat)(v, w);
}
```

A type can support the `+` operator either by implementing `nb_add` or `sq_concat`. These slots have different meanings:

- `nb_add` means algebraic addition with properties like `a + b = b + a`.

- `sq_concat` means the concatenation of sequences.

Built-in types such as `int` and `float` implement `nb_add`, and built-in types such as `str` and `list` implement `sq_concat`. Technically, there's no much difference. The main reason to choose one slot over another is to indicate the appropriate meaning. In fact, the `sq_concat` slot is so unnecessary that it's set to `NULL` for all user-defined types (i.e. classes).

We saw how the `nb_add` slot is used: it's called by the `binary_op1()` function. The next step is to see what it's set to.

What nb_add can be

Since addition is a different operation for different types, the `nb_add` slot of a type must be one of two things:

- it's either a type-specific function that adds object of that type; or
- it's a type-agnostic function that calls some type-specific functions, such as type's `__add__()` special method.

It's indeed one of these two, and which one depends on the type. For example, built-in types such as `int` and `float` have their own implementations of `nb_add`. In contrast, all classes share the same implementation. Fundamentally, built-in types and classes are the same thing – instances of `PyTypeObject`. The important difference between them is how they are created. This difference effects the way the slots are set, so we should discuss it.

Ways to create a type

There are two ways to create a type object:

- by statically defining it; or
- by dynamically allocating it.

Statically defined types

An example of a statically defined type is any built-in type. Here's, for instance, how CPython defines the `float` type:

```
PyTypeObject PyFloat_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "float",
    sizeof(PyFloatObject),
    0,
    (destructor)float_dealloc,                  /* tp_dealloc */
    0,                                          /* tp_vectorcall_offset */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_as_async */
    (reprfunc)float_repr,                       /* tp_repr */
    &float_as_number,                           /* tp_as_number */
    0,                                          /* tp_as_sequence */
    0,                                          /* tp_as_mapping */
    (hashfunc)float_hash,                       /* tp_hash */
    0,                                          /* tp_call */
    0,                                          /* tp_str */
    PyObject_GenericGetAttr,                    /* tp_getattro */
    0,                                          /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,   /* tp_flags */
    float_new__doc__,                           /* tp_doc */
    0,                                          /* tp_traverse */
    0,                                          /* tp_clear */
    float_richcompare,                          /* tp_richcompare */
    0,                                          /* tp_weaklistoffset */
    0,                                          /* tp_iter */
    0,                                          /* tp_iternext */
```

```
    float_methods,                              /* tp_methods */
    0,                                          /* tp_members */
    float_getset,                               /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    0,                                          /* tp_descr_get */
    0,                                          /* tp_descr_set */
    0,                                          /* tp_dictoffset */
    0,                                          /* tp_init */
    0,                                          /* tp_alloc */
    float_new,                                  /* tp_new */
};
```

The slots of a statically defined type are specified explicitly. We can easily see how the `float` type implements `nb_add` by looking at the "number" suite:

```
static PyNumberMethods float_as_number = {
    float_add,          /* nb_add */
    float_sub,          /* nb_subtract */
    float_mul,          /* nb_multiply */
    // ... more number slots
};
```

where we find the `float_add()` function, a straightforward implementation of `nb_add`:

```
static PyObject *
float_add(PyObject *v, PyObject *w)
{
    double a,b;
    CONVERT_TO_DOUBLE(v, a);
    CONVERT_TO_DOUBLE(w, b);
    a = a + b;
    return PyFloat_FromDouble(a);
}
```

The floating-point arithmetic is not that important for our discussion. This example demonstrates how to specify the behavior of a statically defined type. It turned out to be quite easy: just write the implementation of slots and point each slot to the corresponding implementation.

If you want to learn how to statically define your own types, check out [Python's tutorial for C/C++ programmers].

Dynamically allocated types

Dynamically allocated types are the types we define using the `class` statement. As we've already said, they are instances of `PyTypeObject`, just like statically defined types. Traditionally, we call them classes but we might call them user-defined types as well.

From the programmer's perspective, it's easier to define a class in Python than a type in C. This is because CPython does a lot of things behind the scenes when it creates a class. Let's see what's involved in this process.

If we wouldn't know where to start, we could apply the familiar method:

1. Define a simple class

```python
class A:
    pass
```

2. Run the disassembler:

```
$ python -m dis class_A.py
```

3. Study how the VM executes the produced bytecode instructions.

Feel free to do that if you find the time, or read [the article on classes](#) by Eli Bendersky. We'll take a shortcut.

An object is created by a call to a type, e.g. `list()` or `MyClass()`. A class is created by a call to a metatype. A metatype is just a type whose instances are types. Python has one built-in metatype called `PyType_Type`, which is known to us simply as `type`. Here's how it's defined:

```
PyTypeObject PyType_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "type",                                     /* tp_name */
    sizeof(PyHeapTypeObject),                   /* tp_basicsize */
    sizeof(PyMemberDef),                        /* tp_itemsize */
    (destructor)type_dealloc,                   /* tp_dealloc */
    offsetof(PyTypeObject, tp_vectorcall),      /* tp_vectorcall_offset */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_as_async */
    (reprfunc)type_repr,                        /* tp_repr */
    0,                                          /* tp_as_number */
    0,                                          /* tp_as_sequence */
    0,                                          /* tp_as_mapping */
    0,                                          /* tp_hash */
    (ternaryfunc)type_call,                     /* tp_call */
    0,                                          /* tp_str */
    (getattrofunc)type_getattro,                /* tp_getattro */
    (setattrofunc)type_setattro,                /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_HAVE_GC |
    Py_TPFLAGS_BASETYPE | Py_TPFLAGS_TYPE_SUBCLASS |
    Py_TPFLAGS_HAVE_VECTORCALL,                 /* tp_flags */
    type_doc,                                   /* tp_doc */
    (traverseproc)type_traverse,                /* tp_traverse */
    (inquiry)type_clear,                        /* tp_clear */
    0,                                          /* tp_richcompare */
    offsetof(PyTypeObject, tp_weaklist),        /* tp_weaklistoffset */
    0,                                          /* tp_iter */
    0,                                          /* tp_iternext */
    type_methods,                               /* tp_methods */
    type_members,                               /* tp_members */
    type_getsets,                               /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    0,                                          /* tp_descr_get */
    0,                                          /* tp_descr_set */
    offsetof(PyTypeObject, tp_dict),            /* tp_dictoffset */
    type_init,                                  /* tp_init */
    0,                                          /* tp_alloc */
    type_new,                                   /* tp_new */
    PyObject_GC_Del,                            /* tp_free */
    (inquiry)type_is_gc,                        /* tp_is_gc */
};
```

The type of all built-in types is `type`, and the type of all classes defaults to `type`. So, `type` determines how types behave. For example, what happens when we call a type, like `list()` or `MyClass()`, is specified by the `tp_call` slot of `type`. The implementation of the `tp_call` slot of `type` is the `type_call()` function. Its job is to create new objects. It calls two other slots to do that:

1. It calls `tp_new` of a type to create an object.

2. It calls `tp_init` of a type to initialize the created object.

The type of `type` is `type` itself. So, when we call `type()`, the `type_call()` function is invoked. It checks for the special case when we pass a single argument to `type()`. In this case, `type_call()` simply returns the type of the passed object:

```
$ python -q
>>> type(3)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type(type)
<class 'type'>
```

But when we pass three arguments to `type()`, `type_call()` creates a new type by calling `tp_new` and `tp_init` of `type` as described above. The following example demonstrates how to use `type()` to create a class:

```
$ python -q
>>> MyClass = type('MyClass', (), {'__str__': lambda self: 'Hey!'})
>>> instance_of_my_class = MyClass()
>>> str(instance_of_my_class)
Hey!
```

The arguments we pass to `type()` are:

1. the name of a class
2. a tuple of its bases; and
3. a namespace.

Other metatypes take arguments in this form as well.

We saw that we can create a class by calling `type()`, but that's not what we typically do. Typically, we use the `class` statement to define a class. It turns out that in this case too the VM eventually calls some metatype, and most often it calls `type()`.

To execute the `class` statement, the VM calls the `__build_class__()` function from the `builtins` module. What this function does can be summarized as follows:

1. Decide which metatype to call to create the class.
2. Prepare the namespace. The namespace will be used as a class's dictionary.
3. Execute the body of the class in the namespace, thus filling the namespace.
4. Call the metatype.

We can instruct `__build_class__()` which metatype it should call using the `metaclass` keyword. If no `metaclass` is specified, `__build_class__()` calls `type()` by default. It also takes metatypes of bases into account. The exact logic of choosing the metatype is nicely described in the docs.

Suppose we define a new class and do not specify `metaclass`. Where does the class actually get created? In this case, `__build_class__()` calls `type()`. This invokes the `type_call()` function that, in turn, calls the `tp_new` and `tp_init` slots of `type`. The `tp_new` slot of `type` points to the `type_new()` function. This is the function that creates classes. The `tp_init` slot of `type` points to the function that does nothing, so all the work is done by `type_new()`.

The `type_new()` function is nearly 500 lines long and probably deserves a separate post. Its essence, though, can be briefly summarized as follows:

1. Allocate new type object.
2. Set up the allocated type object.

To accomplish the first step, `type_new()` must allocate an instance of `PyTypeObject` as well as suites. Suites must be allocated separately from `PyTypeObject` because `PyTypeObject` contains only pointers to suites, not suites themselves. To handle this inconvenience, `type_new()` allocates an instance of the `PyHeapTypeObject` struct that extends `PyTypeObject` and contains the suites:

```
/* The *real* layout of a type object when allocated on the heap */
typedef struct _heaptypeobject {
    PyTypeObject ht_type;
    PyAsyncMethods as_async;
    PyNumberMethods as_number;
    PyMappingMethods as_mapping;
    PySequenceMethods as_sequence;
    PyBufferProcs as_buffer;
    PyObject *ht_name, *ht_slots, *ht_qualname;
    struct _dictkeysobject *ht_cached_keys;
    PyObject *ht_module;
    /* here are optional user slots, followed by the members. */
} PyHeapTypeObject;
```

To set up a type object means to set up its slots. This is what `type_new()` does for the most part.

Type initialization

Before any type can be used, it should be initialized with the `PyType_Ready()` function. For a class, `PyType_Ready()` is called by `type_new()`. For a statically defined type, `PyType_Ready()` must be called explicitly. When CPython starts, it calls `PyType_Ready()` for each built-in type.

The `PyType_Ready()` function does a number of things. For example, it does slot inheritance.

Slot inheritance

When we define a class that inherits from other type, we expect the class to inherit some behavior of that type. For example, when we define a class that inherits from `int`, we expect it to support the addition:

```
$ python -q
>>> class MyInt(int):
...     pass
...
>>> x = MyInt(2)
>>> y = MyInt(4)
>>> x + y
6
```

Does `MyInt` inherit the `nb_add` slot of `int`? Yes, it does. It's pretty straightforward to inherit the slots from a single ancestor: just copy those slots that the class doesn't have. It's a little bit more complicated when a class has multiple bases. Since bases, in turn, may inherit from other types, all these ancestor types combined form an hierarchy. The problem with the hierarchy is that it doesn't specify the order of inheritance. To solve this problem, `PyType_Ready()` converts this hierarchy into a list. The Method Resolution Order (MRO) determines how to perform this conversion. Once the MRO is calculated, it becomes easy to implement the inheritance in the general case. The `PyType_Ready()` function iterates over ancestors according to the MRO. From each ancestor, it copies those slots that haven't been set on the type before. Some slots support the inheritance and some don't. You can check in the docs whether a particular slot is inherited.

In contrast to a class, a statically defined type can specify at most one base. This is done by implementing the `tp_base` slot.

If no bases are specified, `PyType_Ready()` assumes that the `object` type is the only base. Every type directly or indirectly inherits from `object`. Why? Because it implements the slots that every type is expected to have. For example, it implements `tp_alloc`, `tp_init` and `tp_repr` slots.

The ultimate question

So far we've seen two ways in which a slot can be set:

- It can be specified explicitly (if a type is a statically defined type).
- It can be inherited from an ancestor.

It's still unclear how slots of a class are connected to its special methods. Moreover, we have a reverse problem for built-in types. How do they implement special methods? They certainly do:

```
$ python -q
>>> (3).__add__(4)
7
```

We come to the ultimate question of this post: What's the connection between special methods and slots?

Special methods and slots

The answer lies in the fact that CPython keeps a mapping between special methods and slots. This mapping is represented by the `slotdefs` array. It looks like this:

```
#define TPSLOT(NAME, SLOT, FUNCTION, WRAPPER, DOC) \
    {NAME, offsetof(PyTypeObject, SLOT), (void *)(FUNCTION), WRAPPER, \
     PyDoc_STR(DOC)}

static slotdef slotdefs[] = {
    TPSLOT("__getattribute__", tp_getattr, NULL, NULL, ""),
    TPSLOT("__getattr__", tp_getattr, NULL, NULL, ""),
    TPSLOT("__setattr__", tp_setattr, NULL, NULL, ""),
    TPSLOT("__delattr__", tp_setattr, NULL, NULL, ""),
    TPSLOT("__repr__", tp_repr, slot_tp_repr, wrap_unaryfunc,
           "__repr__($self, /)\n--\n\nReturn repr(self)."),
    TPSLOT("__hash__", tp_hash, slot_tp_hash, wrap_hashfunc,
           "__hash__($self, /)\n--\n\nReturn hash(self)."),
    // ... more slotdefs
}
```

Each entry of this array is a `slotdef` struct:

```
// typedef struct wrapperbase slotdef;

struct wrapperbase {
    const char *name;
    int offset;
    void *function;
    wrapperfunc wrapper;
    const char *doc;
    int flags;
    PyObject *name_strobj;
};
```

Four members of this struct are important for our discussion:

- `name` is a name of a special method.
- `offset` is an offset of a slot in the `PyHeapTypeObject` struct. It specifies the slot corresponding to the special method.
- `function` is an implementation of a slot. When a special method is defined, the corresponding slot is set to `function`. Typically, `function` calls special methods to do the work.
- `wrapper` is a wrapper function around a slot. When a slot is defined, `wrapper` provides an implementation for the corresponding special method. It calls the slot to do the work.

Here's, for example, an entry that maps `__add__()` special method to the `nb_add` slot:

- `name` is `"__add__"`.
- `offset` is `offsetof(PyHeapTypeObject, as_number.nb_add)`.
- `function` is `slot_nb_add()`.
- `wrapper` is `wrap_binaryfunc_l()`.

The `slotdefs` array is a many-to-many mapping. For example, as we'll see, both the `__add__()` and `__radd__()` special methods map to the same `nb_add` slot. Conversely, both the `mp_subscript` "mapping" slot and the `sq_item` "sequence" slot map to the same `__getitem__()` special method.

CPython uses the `slotdefs` array in two ways:

- to set slots based on special methods; and
- to set special methods based on slots.

Slots based on special methods

The `type_new()` function calls `fixup_slot_dispatchers()` to set slots based on special methods. The `fixup_slot_dispatchers()` function calls `update_one_slot()` for each slot in the `slotdefs` array, and `update_one_slot()` sets the slot to `function` if a class has the corresponding special method.

Let's take the `nb_add` slot as an example. The `slotdefs` array has two entries corresponding to that slot:

```
static slotdef slotdefs[] = {
    // ...
    BINSLOT("__add__", nb_add, slot_nb_add, "+"),
    RBINSLOT("__radd__", nb_add, slot_nb_add,"+"),
    // ...
}
```

`BINSLOT()` and `RBINSLOT()` are macros. Let's expand them:

```
static slotdef slotdefs[] = {
    // ...
    // {name, offset, function,
    //     wrapper, doc}
    //
    {"__add__", offsetof(PyHeapTypeObject, as_number.nb_add), (void *)(slot_nb_add),
        wrap_binaryfunc_l, PyDoc_STR("__add__" "($self, value, /)\n--\n\nReturn self" "+" "value.")},

    {"__radd__", offsetof(PyHeapTypeObject, as_number.nb_add), (void *)(slot_nb_add),
        wrap_binaryfunc_r, PyDoc_STR("__radd__" "($self, value, /)\n--\n\nReturn value" "+" "self.")},
```

```
        // ...
    }
```

What `update_one_slot()` does is look up `class.__add__()` and `class.__radd__()`. If either is defined, it sets `nb_add` of the class to `slot_nb_add()`. Note that both entries agree on `slot_nb_add()` as `function`. Otherwise, we would have a conflict when both are defined.

Now, what is `slot_nb_add()`, you ask? This function is defined with a macro that expands as follows:

```
static PyObject *
slot_nb_add(PyObject *self, PyObject *other) {
    PyObject* stack[2];
    PyThreadState *tstate = _PyThreadState_GET();
    _Py_static_string(op_id, "__add__");
    _Py_static_string(rop_id, "__radd__");
    int do_other = !Py_IS_TYPE(self, Py_TYPE(other)) && \
        Py_TYPE(other)->tp_as_number != NULL && \
        Py_TYPE(other)->tp_as_number->nb_add == slot_nb_add;
    if (Py_TYPE(self)->tp_as_number != NULL && \
        Py_TYPE(self)->tp_as_number->nb_add == slot_nb_add) {
        PyObject *r;
        if (do_other && PyType_IsSubtype(Py_TYPE(other), Py_TYPE(self))) {
            int ok = method_is_overloaded(self, other, &rop_id);
            if (ok < 0) {
                return NULL;
            }
            if (ok) {
                stack[0] = other;
                stack[1] = self;
                r = vectorcall_maybe(tstate, &rop_id, stack, 2);
                if (r != Py_NotImplemented)
                    return r;
                Py_DECREF(r); do_other = 0;
            }
        }
        stack[0] = self;
        stack[1] = other;
        r = vectorcall_maybe(tstate, &op_id, stack, 2);
        if (r != Py_NotImplemented || Py_IS_TYPE(other, Py_TYPE(self)))
            return r;
        Py_DECREF(r);
    }
    if (do_other) {
        stack[0] = other;
        stack[1] = self;
        return vectorcall_maybe(tstate, &rop_id, stack, 2);
    }
    Py_RETURN_NOTIMPLEMENTED;
}
```

You don't need to study this code carefully. Recall the `binary_op1()` function that calls the `nb_add` slot. The `slot_nb_add()` function basically repeats the logic of `binary_op1()`. The main difference is that `slot_nb_add()` eventually calls `__add__()` or `__radd__()`.

Setting special method on existing class

Suppose that we create a class without the `__add__()` and `__radd__()` special methods. In this case, the `nb_add` slot of the class is set to `NULL`. As expected, we cannot add instances of that class. If we, however, set `__add__()` or `__radd__()` after the class has been created, the addition works as if the method was a part of the class definition. Here's what I mean:

```
$ python -q
>>> class A:
...     pass
...
>>> x = A()
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'A' and 'int'
>>> A.__add__ = lambda self, o: 5
>>> x + 2
5
>>>
```

How does that work? To set an attribute on an object, the VM calls the `tp_setattro` slot of the object's type. The `tp_setattro` slot of `type` points to the `type_setattro()` function, so, when we set an attribute on a class, this function gets called. To set an attribute, it stores the value of the attribute in the class's dictionary. After this, it checks if the attribute is a special method. If it is a special method, the corresponding slots are set. The same `update_one_slot()` function is called to do that.

Before we can learn how CPython does the reverse, that is, how it adds special methods to built-in types, we need to understand what a method is.

Methods

A method is an attribute, but a peculiar one. When we call a method from an instance, the method implicitly receives the instance as its first parameter, which we usually denote `self`:

```
$ python -q
>>> class A:
...     def method(self, x):
...             return self, x
...
>>> a = A()
>>> a.method(1)
(<__main__.A object at 0x10d10bfd0>, 1)
```

But when we call the same method from a class, we have to pass all arguments explicitly:

```
>>> A.method(a, 1)
(<__main__.A object at 0x10d10bfd0>, 1)
```

In our example, the method takes one argument in one case and two arguments in another. How is that possible that the same attribute is a different thing depending on how we access it?

First of all, realize that a method we define on a class is just a function. A function accessed through an instance differs from the same function accessed through the instance's type because the `function` type implements the descriptor protocol. If you're unfamiliar with descriptors, I highly recommend you to read Descriptor HowTo Guide by Raymond Hettinger. In a nutshell, a descriptor is an object that, when used as an attribute, determines by itself how you get, set and delete it. Technically, a descriptor is an object that implements `__get__()`, `__set__()`, or `__delete__()` special methods.

The `function` type implements `__get__()`. When we look up some method, what we get is the result of a call to `__get__()`. Three arguments are passed to it:

- an attribute, i.e. a function

- an instance
- the instance's type.

If we look up a method on a type, the instance is `NULL`, and `__get__()` simply returns the function. If we look up a method on an instance, `__get__()` returns a method object:

```
>>> type(A.method)
<class 'function'>
>>> type(a.method)
<class 'method'>
```

A method object stores a function and an instance. When called, it prepends the instance to the list of arguments and calls the function.

Now we're ready to tackle the last question.

Special methods based on slots

Recall the `PyType_Ready()` function that initializes types and does slot inheritance. It also adds special methods to a type based on the implemented slots. `PyType_Ready()` calls `add_operators()` to do that. The `add_operators()` function iterates over the entries in the `slotdefs` array. For each entry, it checks whether the special method specified by the entry should be added to the type's dictionary. A special method is added if it's not already defined and if the type implements the slot specified by the entry. For example, if the `__add__()` special method is not defined on a type, but the type implements the `nb_add` slot, `add_operators()` puts `__add__()` in the type's dictionary.

What is `__add__()` set to? Like any other method, it must be set to some descriptor to behave like a method. While methods defined by a programmer are functions, methods set by `add_operators()` are wrapper descriptors. A wrapper descriptor is a descriptor that stores two things:

- It stores a wrapped slot. A wrapped slot "does the work" for a special method. For example, the wrapper descriptor of the `__add__()` special method of the `float` type stores `float_add()` as a wrapped slot.
- It stores a wrapper function. A wrapper function "knows" how to call the wrapped slot. It is `wrapper` of a `slotdef` entry.

When we call a special method that was added by `add_operators()`, we call a wrapper descriptor. When we call a wrapper descriptor, it calls a wrapper function. A wrapper descriptor passes to a wrapper function the same arguments that we pass to a special methods plus the wrapped slot. Finally, the wrapper function calls the wrapped slot.

Let's see how a built-in type that implements the `nb_add` slot gets its `__add__()` and `__radd__()` special methods. Recall the `slotdef` entries corresponding to `nb_add`:

```
static slotdef slotdefs[] = {
    // ...
    // {name, offset, function,
    //     wrapper, doc}
    //
    {"__add__", offsetof(PyHeapTypeObject, as_number.nb_add), (void *)(slot_nb_add),
        wrap_binaryfunc_l, PyDoc_STR("__add__" "($self, value, /)\n--\n\nReturn self" "+" "value.")},

    {"__radd__", offsetof(PyHeapTypeObject, as_number.nb_add), (void *)(slot_nb_add),
        wrap_binaryfunc_r, PyDoc_STR("__radd__" "($self, value, /)\n--\n\nReturn value" "+" "self.")},
    // ...
}
```

If a type implements the `nb_add` slot, `add_operators()` sets `__add__()` of the type to a wrapper descriptor with `wrap_binaryfunc_l()` as a wrapper function and `nb_add` as a wrapped slot. It similarly sets `__radd__()` of the type with one exception: a wrapper function is `wrap_binaryfunc_r()`.

Both `wrap_binaryfunc_l()` and `wrap_binaryfunc_r()` take two operands plus a wrapped slot as their parameters. The only difference is how they call the slot:

- `wrap_binaryfunc_l(x, y, slot_func)` calls `slot_func(x, y)`
- `wrap_binaryfunc_r(x, y, slot_func)` calls `slot_func(y, x)`.

The result of this call is what we get when we call the special method.

Summary

Today we've demystified perhaps the most magical aspect of Python. We've learned that the behavior of a Python object is determined by the slots of the object's type. The slots of a statically defined type can be specified explicitly, and any type can inherit some slots from its ancestors. The real insight was that the slots of a class are set up automatically by CPython based on the defined special methods. CPython does the reverse too. It adds special methods to the type's dictionary if the type implements the corresponding slots.

We've learned a lot. Nevertheless, the Python object system is such a vast subject that at least as much remains to be covered. For example, we haven't really discussed how attributes work. This is what we're going to do next time.

If you have any questions, comments or suggestions, feel free to contact me at victor@tenthousandmeters.com