Python behind the scenes #7: how Python attributes work

Published: Mon 28 December 2020
By Victor Skvortsov
tags: Python behind the scenes Python CPython

What happens when we get or set an attribute of a Python object? This question is not as simple as it may seem at first. It is true that any experienced Python programmer has a good intuitive understanding of how attributes work, and the documentation helps a lot to strengthen the understanding. Yet, when a really non-trivial question regarding attributes comes up, the intuition fails and the documentation can no longer help. To gain a deep understanding and be able to answer such questions, one has to study how attributes are implemented. That's what we're going to do today.

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.

A quick refresher

Last time we studied how the Python object system works. Some of the things we've learned in that part are crucial for our current discussion, so let's recall them briefly.

A Python object is an instance of a C struct that has at least two members:

- a reference count; and
- a pointer to the object's type.

Every object must have a type because the type determines how the object behaves. A type is also a Python object, an instance of the `PyTypeObject` struct:

```
// 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;
};
```

The members of a type are called slots. Each slot is responsible for a particular aspect of the object's behavior. For example, the `tp_call` slot of a type specifies what happens when we call the objects of that type. Some slots are grouped together in suites. An example of a suite is the "number" suite `tp_as_number`. Last time we studied its `nb_add` slot that specifies how to add objects. This and all other slots are very well described in the docs.

How slots of a type are set depends on how the type is defined. There are two ways to define a type in CPython:

- statically; or
- dynamically.

A statically defined type is just a statically initialized instance of `PyTypeObject`. All built-in types are defined statically. Here's, for example, the definition of 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 */
};
```

To dynamically allocate a new type, we call a metatype. A metatype is a type whose instances are types. It determines how types behave. In particular, it creates new type instances. Python has one built-in metatype known as `type`. It's the metatype of all built-in types. It's also used as the default metatype to create classes. When CPython executes the `class` statement, it typically calls `type()` to create the class. We can create a class by calling `type()` directly as well:

```
MyClass = type(name, bases, namespace)
```

The `tp_new` slot of `type` is called to create a class. The implementation of this slot is the `type_new()` function. This function allocates the type object and sets it up.

Slots of a statically defined type are specified explicitly. Slots of a class are set automatically by the metatype. Both statically and dynamically defined types can inherit some slots from its bases.

Some slots are mapped to special methods. If a class defines a special method that corresponds to some slot, CPython automatically sets the slot to the default implementation that calls the special method. This is why we can add objects whose class defines `__add__()`. CPython does the reverse for a statically defined type. If such a type implements a slot that corresponds to some special method, CPython sets the special method to the implementation that wraps the slot. This is how the `int` type gets its `__add__()` special method.

All types must be initialized by calling the `PyType_Ready()` function. This function does a lot of things. For example, it does slot inheritance and adds special methods based on slots. 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.

With this in mind, let's turn our attention to attributes.

Attributes and the VM

What is an attribute? We might say that an attribute is a variable associated with an object, but it's more than that. It's hard to give a definition that captures all important aspects of attributes. So, instead of starting with a definition, let's start with something we know for sure.

We know for sure that in Python we can do three things with attributes:

- get the value of an attribute: `value = obj.attr`
- set an attribute to some value: `obj.attr = value`
- delete an attribute: `del obj.attr`

What these operations do depends, like any other aspect of the object's behavior, on the object's type. A type has certain slots responsible for getting, setting and deleting attributes. The VM calls these slots to execute the statements like `value = obj.attr` and `obj.attr = value`. To see how the VM does that and what these slots are, let's apply the familiar method:

1. Write a piece of code that gets/sets/deletes an attribute.
2. Disassemble it to bytecode using the `dis` module.
3. Take a look at the implementation of the produced bytecode instructions in `ceval.c`.

Getting an attribute

Let's first see what the VM does when we get the value of an attribute. The compiler produces the `LOAD_ATTR` opcode to load the value:

```
$ echo 'obj.attr' | python -m dis
  1           0 LOAD_NAME                0 (obj)
              2 LOAD_ATTR                1 (attr)
...
```

And the VM executes this opcode as follows:

```
case TARGET(LOAD_ATTR): {
    PyObject *name = GETITEM(names, oparg);
    PyObject *owner = TOP();
    PyObject *res = PyObject_GetAttr(owner, name);
    Py_DECREF(owner);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

We can see that the VM calls the `PyObject_GetAttr()` function to do the job. Here's what this function does:

```
PyObject *
PyObject_GetAttr(PyObject *v, PyObject *name)
{
    PyTypeObject *tp = Py_TYPE(v);

    if (!PyUnicode_Check(name)) {
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     Py_TYPE(name)->tp_name);
        return NULL;
    }
    if (tp->tp_getattro != NULL)
        return (*tp->tp_getattro)(v, name);
    if (tp->tp_getattr != NULL) {
        const char *name_str = PyUnicode_AsUTF8(name);
        if (name_str == NULL)
            return NULL;
        return (*tp->tp_getattr)(v, (char *)name_str);
    }
    PyErr_Format(PyExc_AttributeError,
                 "'%.50s' object has no attribute '%U'",
                 tp->tp_name, name);
    return NULL;
}
```

It first tries to call the `tp_getattro` slot of the object's type. If this slot is not implemented, it tries to call the `tp_getattr` slot. If `tp_getattr` is not implemented either, it raises `AttributeError`.

A type implements `tp_getattro` or `tp_getattr` or both to support attribute access. According to the documentation, the only difference between them is that `tp_getattro` takes a Python string as the name of an attribute and `tp_getattr` takes a C string. Though the choice exists, you won't find types in CPython that implement `tp_getattr`, because it has been deprecated in favor of `tp_getattro`.

Setting an attribute

From the VM's perspective, setting an attribute is not much different from getting it. The compiler produces the `STORE_ATTR` opcode to set an attribute to some value:

```
$ echo 'obj.attr = value' | python -m dis
  1           0 LOAD_NAME                0 (value)
              2 LOAD_NAME                1 (obj)
              4 STORE_ATTR               2 (attr)
...
```

And the VM executes `STORE_ATTR` as follows:

```
case TARGET(STORE_ATTR): {
    PyObject *name = GETITEM(names, oparg);
    PyObject *owner = TOP();
    PyObject *v = SECOND();
    int err;
    STACK_SHRINK(2);
    err = PyObject_SetAttr(owner, name, v);
    Py_DECREF(v);
    Py_DECREF(owner);
    if (err != 0)
        goto error;
    DISPATCH();
}
```

We find that `PyObject_SetAttr()` is the function that does the job:

```
int
PyObject_SetAttr(PyObject *v, PyObject *name, PyObject *value)
{
    PyTypeObject *tp = Py_TYPE(v);
    int err;

    if (!PyUnicode_Check(name)) {
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     Py_TYPE(name)->tp_name);
        return -1;
    }
    Py_INCREF(name);

    PyUnicode_InternInPlace(&name);
    if (tp->tp_setattro != NULL) {
        err = (*tp->tp_setattro)(v, name, value);
        Py_DECREF(name);
        return err;
    }
    if (tp->tp_setattr != NULL) {
        const char *name_str = PyUnicode_AsUTF8(name);
        if (name_str == NULL) {
            Py_DECREF(name);
            return -1;
        }
        err = (*tp->tp_setattr)(v, (char *)name_str, value);
        Py_DECREF(name);
        return err;
    }
    Py_DECREF(name);
    _PyObject_ASSERT(name, Py_REFCNT(name) >= 1);
    if (tp->tp_getattr == NULL && tp->tp_getattro == NULL)
        PyErr_Format(PyExc_TypeError,
                     "'%.100s' object has no attributes "
                     "(%s .%U)",
                     tp->tp_name,
                     value==NULL ? "del" : "assign to",
                     name);
    else
        PyErr_Format(PyExc_TypeError,
                     "'%.100s' object has only read-only attributes "
                     "(%s .%U)",
                     tp->tp_name,
                     value==NULL ? "del" : "assign to",
                     name);
    return -1;
}
```

This function calls the `tp_setattro` and `tp_setattr` slots the same way as `PyObject_GetAttr()` calls `tp_getattro` and `tp_getattr`. The `tp_setattro` slot comes in pair with `tp_getattro`, and `tp_setattr` comes in pair with `tp_getattr`. Just like `tp_getattr`, `tp_setattr` is deprecated.

Note that `PyObject_SetAttr()` checks whether a type defines `tp_getattro` or `tp_getattr`. A type must implement attribute access to support attribute assignment.

Deleting an attribute

Interestingly, a type has no special slot for deleting an attribute. What then specifies how to delete an attribute? Let's see. The compiler produces the `DELETE_ATTR` opcode to delete an attribute:

```
$ echo 'del obj.attr' | python -m dis
  1           0 LOAD_NAME                0 (obj)
              2 DELETE_ATTR              1 (attr)
```

The way the VM executes this opcode reveals the answer:

```
case TARGET(DELETE_ATTR): {
    PyObject *name = GETITEM(names, oparg);
    PyObject *owner = POP();
    int err;
    err = PyObject_SetAttr(owner, name, (PyObject *)NULL);
    Py_DECREF(owner);
    if (err != 0)
        goto error;
    DISPATCH();
}
```

To delete an attribute, the VM calls the same `PyObject_SetAttr()` function that it calls to set an attribute, so the same `tp_setattro` slot is responsible for deleting attributes. But how does it know which of two operations to perform? The `NULL` value indicates that the attribute should be deleted.

As this section shows, the `tp_getattro` and `tp_setattro` slots determine how attributes of an object work. The next question that comes to mind is: How are these slots implemented?

Slots implementations

Any function of the appropriate signature can be an implementation of `tp_getattro` and `tp_setattro`. A type can implement these slots in an absolutely arbitrary way. Fortunately, we need to study only a few implementations to understand how Python attributes work. This is because most types use the same generic implementation.

The generic functions for getting and setting attributes are `PyObject_GenericGetAttr()` and `PyObject_GenericSetAttr()`. All classes use them by default. Most built-in types specify them as slots implementations explicitly or inherit them from `object` that also uses the generic implementation.

In this post, we'll focus on the generic implementation, since it's basically what we mean by Python attributes. We'll also discuss two important cases when the generic implementation is not used. The first case is `type`. It implements the `tp_getattro` and `tp_setattro` slots in its own way, though its implementation is quite similar to the generic one. The second case is any class that customizes attribute access and assignment by defining the `__getattribute__()`, `__getattr__()`, `__setattr__()` and `__delattr__()` special methods. CPython sets the `tp_getattro` and `tp_setattro` slots of such a class to functions that call those methods.

Generic attribute management

The `PyObject_GenericGetAttr()` and `PyObject_GenericSetAttr()` functions implement the behavior of attributes that we're all accustomed to. When we set an attribute of an object to some value, CPython puts the value in the object's dictionary:

```
$ python -q
>>> class A:
...     pass
...
>>> a = A()
>>> a.__dict__
{}
>>> a.x = 'instance attribute'
>>> a.__dict__
{'x': 'instance attribute'}
```

When we try to get the value of the attribute, CPython loads it from the object's dictionary:

```
>>> a.x
'instance attribute'
```

If the object's dictionary doesn't contain the attribute, CPython loads the value from the type's dictionary:

```
>>> A.y = 'class attribute'
>>> a.y
'class attribute'
```

If the type's dictionary doesn't contain the attribute either, CPython searches for the value in the dictionaries of the type's parents:

```
>>> class B(A): # note the inheritance
...     pass
...
>>> b = B()
>>> b.y
'class attribute'
```

So, an attribute of an object is one of two things:

- an instance variable; or
- a type variable.

Instance variables are stored in the object's dictionary, and type variables are stored in the type's dictionary and in the dictionaries of the type's parents. To set an attribute to some value, CPython simply updates the object's dictionary. To get the value of an attribute, CPython searches for it first in the object's dictionary and then in the type's dictionary and in the dictionaries of the type's parents. The order in which CPython iterates over the types when it searches for the value is the Method Resolution Order (MRO).

Python attributes would be as simple as that if there were no descriptors.

Descriptors

Technically, a descriptor is a Python object whose type implements certain slots: `tp_descr_get` or `tp_descr_set` or both. Essentially, a descriptor is a Python object that, when used as an attribute, controls what happens we get, set or delete it. If `PyObject_GenericGetAttr()` finds that the attribute value is a descriptor whose type implements `tp_descr_get`, it doesn't just return the value as it normally does but calls `tp_descr_get` and returns the result of this call. The `tp_descr_get` slot takes three parameters: the descriptor itself, the object whose attribute is being looked up and the object's type. It's up to `tp_descr_get` to decide what to do with the parameters and what to return. Similarly, `PyObject_GenericSetAttr()` looks up the current attribute value. If it finds that the value is a descriptor whose type implements `tp_descr_set`, it calls `tp_descr_set` instead of just updating the object's dictionary. The arguments passed to `tp_descr_set` are the descriptor, the object, and the new attribute value. To delete an attribute, `PyObject_GenericSetAttr()` calls `tp_descr_set` with the new attribute value set to `NULL`.

On one side, descriptors make Python attributes a bit complex. On the other side, descriptors make Python attributes powerful. As Python's glossary says,

> *Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.*

Let's revise one important use case of descriptors that we discussed in the previous part: methods.

A function put in the type's dictionary works not like an ordinary function but like a method. That is, we don't need to explicitly pass the first argument when we call it:

```
>>> A.f = lambda self: self
>>> a.f()
<__main__.A object at 0x108a20d60>
```

The `a.f` attribute not only works like a method, it is a method:

```
>>> a.f
<bound method <lambda> of <__main__.A object at 0x108a20d60>>
```

However, if we look up the value of `'f'` in the type's dictionary, we'll get the original function:

```
>>> A.__dict__['f']
<function <lambda> at 0x108a4ca60>
```

CPython returns not the value stored in the dictionary but something else. This is because functions are descriptors. The `function` type implements the `tp_descr_get` slot, so `PyObject_GenericGetAttr()` calls this slot and returns the result of the call. The result of the call is a method object that stores both the function and the instance. When we call a method object, the instance is prepended to the list of arguments and the function gets invoked.

Descriptors have their special behavior only when they are used as type variables. When they are used as instance variables, they behave like ordinary objects. For example, a function put in the object's dictionary does not become a method:

```
>>> a.g = lambda self: self
>>> a.g
<function <lambda> at 0x108a4cc10>
```

Apparently, the language designers haven't found a case when using a descriptor as an instance variable would be a good idea. A nice consequence of this decision is that instance variables are very straightforward. They are just data.

The `function` type is an example of a built-in descriptor type. We can also define our own descriptors. To do that, we create a class that implements the descriptor protocol: the `__get__()`, `__set__()` and `__delete__()` special methods:

```
>>> class DescrClass:
...     def __get__(self, obj, type=None):
...             print('I can do anything')
...             return self
...
```

```
>>> A.descr_attr = DescrClass()
>>> a.descr_attr
I can do anything
<__main__.DescrClass object at 0x108b458e0>
```

If a class defines `__get__()`, CPython sets its `tp_descr_get` slot to the function that calls that method. If a class defines `__set__()` or `__delete__()`, CPython sets its `tp_descr_set` slot to the function that calls `__delete__()` when the value is `NULL` and calls `__set__()` otherwise.

If you wonder why anyone would want to define their our descriptors in the first place, check out the excellent Descriptor HowTo Guide by Raymond Hettinger.

Our goal is to study the actual algorithms for getting and setting attributes. Descriptors is one prerequisite for that. Another is the understanding of what the object's dictionary and the type's dictionary really are.

Object's dictionary and type's dictionary

An object's dictionary is a dictionary in which instance variables are stored. Every object of a type keeps a pointer to its own dictionary. For example, every function object has the `func_dict` member for that purpose:

```
typedef struct {
    // ...
    PyObject *func_dict;        /* The __dict__ attribute, a dict or NULL */
    // ...
} PyFunctionObject;
```

To tell CPython which member of an object is the pointer to the object's dictionary, the object's type specifies the offset of this member using the `tp_dictoffset` slot. Here's how the `function` type does this:

```
PyTypeObject PyFunction_Type = {
    // ...
    offsetof(PyFunctionObject, func_dict),      /* tp_dictoffset */
    // ...
};
```

A positive value of `tp_dictoffset` specifies an offset from the start of the object's struct. A negative value specifies an offset from the end of the struct. The zero offset means that the objects of the type don't have dictionaries. Integers, for example, are such objects:

```
>>> (12).__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute '__dict__'
```

We can assure ourselves that `tp_dictoffset` of the `int` type is set to `0` by checking the `__dictoffset__` attribute:

```
>>> int.__dictoffset__
0
```

Classes usually have a non-zero `tp_dictoffset`. The only exception is classes that define the `__slots__` attribute. This attribute is an optimization. We'll cover the essentials first and discuss `__slots__` later.

A type's dictionary is a dictionary of a type object. Just like the `func_dict` member of a function points to the function's dictionary, the `tp_dict` slot of a type points to the type's dictionary. The crucial difference between the dictionary of an ordinary object and the dictionary of a type is that CPython knows about `tp_dict`, so it can avoid locating the dictionary of a type via `tp_dictoffset`. Handling the dictionary of a type in a general way would introduce an additional level of indirection and wouldn't brings much benefit.

Now, when we know what descriptors are and where attributes are stored, we're ready to see what the `PyObject_GenericGetAttr()` and `PyObject_GenericSetAttr()` functions do.

PyObject_GenericSetAttr()

We begin with `PyObject_GenericSetAttr()`, a function whose job is set an attribute to a given value. This function turns out to be a thin wrapper around another function:

```
int
PyObject_GenericSetAttr(PyObject *obj, PyObject *name, PyObject *value)
{
    return _PyObject_GenericSetAttrWithDict(obj, name, value, NULL);
}
```

And that function actually does the work:

```
int
_PyObject_GenericSetAttrWithDict(PyObject *obj, PyObject *name,
                                 PyObject *value, PyObject *dict)
{
    PyTypeObject *tp = Py_TYPE(obj);
    PyObject *descr;
    descrsetfunc f;
    PyObject **dictptr;
    int res = -1;

    if (!PyUnicode_Check(name)){
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     Py_TYPE(name)->tp_name);
        return -1;
    }

    if (tp->tp_dict == NULL && PyType_Ready(tp) < 0)
        return -1;

    Py_INCREF(name);

    // Look up the current attribute value
    // in the type's dict and in the parent's dicts using the MRO.
    descr = _PyType_Lookup(tp, name);

    // If found a descriptor that implements `tp_descr_set`, call this slot.
    if (descr != NULL) {
        Py_INCREF(descr);
        f = Py_TYPE(descr)->tp_descr_set;
        if (f != NULL) {
            res = f(descr, obj, value);
            goto done;
        }
    }

    // `PyObject_GenericSetAttr()` calls us with `dict` set to `NULL`.
    // So, `if` will be executed.
    if (dict == NULL) {
        // Get the object's dict.
        dictptr = _PyObject_GetDictPtr(obj);
        if (dictptr == NULL) {
            if (descr == NULL) {
                PyErr_Format(PyExc_AttributeError,
                             "'%.100s' object has no attribute '%U'",
                             tp->tp_name, name);
            }
            else {
                PyErr_Format(PyExc_AttributeError,
                             "'%.50s' object attribute '%U' is read-only",
                             tp->tp_name, name);
            }
            goto done;
        }
        // Update the object's dict with the new value.
        // If `value` is `NULL`, delete the attribute from the dict.
        res = _PyObjectDict_SetItem(tp, dictptr, name, value);
    }
    else {
        Py_INCREF(dict);
        if (value == NULL)
            res = PyDict_DelItem(dict, name);
        else
            res = PyDict_SetItem(dict, name, value);
        Py_DECREF(dict);
    }
    if (res < 0 && PyErr_ExceptionMatches(PyExc_KeyError))
        PyErr_SetObject(PyExc_AttributeError, name);

  done:
    Py_XDECREF(descr);
    Py_DECREF(name);
    return res;
}
```

Despite its length, the function implements a simple algorithm:

1. Search for the attribute value among type variables. The order of the search is the MRO.
2. If the value is a descriptor whose type implements the `tp_descr_set` slot, call the slot.
3. Otherwise, update the object's dictionary with the new value.

We haven't discussed the descriptor types that implement the `tp_descr_set` slot, so you may wonder why we need them at all. Consider Python's `property()`. The following example from the docs demonstrates its canonical usage to create a managed attribute:

```python
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

If c is an instance of C, `c.x` will invoke the getter, `c.x = value` will invoke the setter and `del c.x` the deleter.

How does `property()` work? The answer is simple: it's a descriptor type. It implements both the `tp_descr_get` and `tp_descr_set` slots that call the specified functions.

The example from the docs is only a framework and doesn't do much. However, it can easily be extended to do something useful. For example, we can write a setter that performs some validation of the new attribute value.

PyObject_GenericGetAttr()

Getting the value of an attribute is a bit more complicated than setting it. Let's see by how much. The `PyObject_GenericGetAttr()` function also delegates the work to another function:

```
PyObject *
PyObject_GenericGetAttr(PyObject *obj, PyObject *name)
{
    return _PyObject_GenericGetAttrWithDict(obj, name, NULL, 0);
}
```

And here's what that function does:

```
PyObject *
_PyObject_GenericGetAttrWithDict(PyObject *obj, PyObject *name,
                                 PyObject *dict, int suppress)
{
    /* Make sure the logic of _PyObject_GetMethod is in sync with
       this method.

       When suppress=1, this function suppress AttributeError.
    */

    PyTypeObject *tp = Py_TYPE(obj);
    PyObject *descr = NULL;
    PyObject *res = NULL;
    descrgetfunc f;
    Py_ssize_t dictoffset;
    PyObject **dictptr;

    if (!PyUnicode_Check(name)){
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     Py_TYPE(name)->tp_name);
        return NULL;
    }
    Py_INCREF(name);

    if (tp->tp_dict == NULL) {
        if (PyType_Ready(tp) < 0)
            goto done;
    }

    // Look up the attribute value
    // in the type's dict and in the parent's dicts using the MRO.
    descr = _PyType_Lookup(tp, name);

    // Check if the value is a descriptor that implements:
    // * `tp_descr_get`; and
    // * `tp_descr_set` (data descriptor)
    // In this case, call `tp_descr_get`
    f = NULL;
    if (descr != NULL) {
        Py_INCREF(descr);
        f = Py_TYPE(descr)->tp_descr_get;
        if (f != NULL && PyDescr_IsData(descr)) {
            res = f(descr, obj, (PyObject *)Py_TYPE(obj));
            if (res == NULL && suppress &&
                    PyErr_ExceptionMatches(PyExc_AttributeError)) {
                PyErr_Clear();
            }
            goto done;
        }
    }

    // Look up the attribute value in the object's dict
    // Return if found one
    if (dict == NULL) {
        /* Inline _PyObject_GetDictPtr */
        dictoffset = tp->tp_dictoffset;
        if (dictoffset != 0) {
            if (dictoffset < 0) {
                Py_ssize_t tsize = Py_SIZE(obj);
                if (tsize < 0) {
                    tsize = -tsize;
                }
                size_t size = _PyObject_VAR_SIZE(tp, tsize);
                _PyObject_ASSERT(obj, size <= PY_SSIZE_T_MAX);

                dictoffset += (Py_ssize_t)size;
                _PyObject_ASSERT(obj, dictoffset > 0);
                _PyObject_ASSERT(obj, dictoffset % SIZEOF_VOID_P == 0);
            }
            dictptr = (PyObject **) ((char *)obj + dictoffset);
            dict = *dictptr;
        }
    }
    if (dict != NULL) {
        Py_INCREF(dict);
        res = PyDict_GetItemWithError(dict, name);
        if (res != NULL) {
            Py_INCREF(res);
            Py_DECREF(dict);
            goto done;
        }
        else {
            Py_DECREF(dict);
            if (PyErr_Occurred()) {
                if (suppress && PyErr_ExceptionMatches(PyExc_AttributeError)) {
                    PyErr_Clear();
                }
                else {
                    goto done;
                }
            }
        }
    }

    // If _PyType_Lookup found a non-data desciptor,
    // call its `tp_descr_get`
    if (f != NULL) {
        res = f(descr, obj, (PyObject *)Py_TYPE(obj));
        if (res == NULL && suppress &&
                PyErr_ExceptionMatches(PyExc_AttributeError)) {
            PyErr_Clear();
        }
        goto done;
    }

    // If _PyType_Lookup found some value,
    // return it
    if (descr != NULL) {
        res = descr;
        descr = NULL;
        goto done;
    }

    if (!suppress) {
        PyErr_Format(PyExc_AttributeError,
                     "'%.50s' object has no attribute '%U'",
                     tp->tp_name, name);
    }
  done:
    Py_XDECREF(descr);
    Py_DECREF(name);
    return res;
}
```

The major steps of this algorithm are:

1. Search for the attribute value among type variables. The order of the search is the MRO.
2. If the value is a data descriptor whose type implements the `tp_descr_get` slot, call this slot and return the result of the call. Otherwise, remember the value and continue. A data descriptor is a descriptor whose type implements the `tp_descr_set` slot.
3. Locate the object's dictionary using `tp_dictoffset`. If the dictionary contains the value, return it.
4. If the value from step 2 is a descriptor whose type implements the `tp_descr_get` slot, call this slot and return the result of the call.
5. Return the value from step 2. The value can be `NULL`.

Since an attribute can be both an instance variable and a type variable, CPython must decide which one takes precedence over the other. What the algorithm does is essentially implement a certain order of precedence. This order is:

1. type data descriptors
2. instance variables
3. type non-data descriptors and other type variables.

The natural question to ask is: Why does it implement this particular order? More specifically, **why do data descriptors take precedence over instance variables but non-data descriptors don't?** First of all, note that some descriptors must take precedence over instance variables in order for attributes to work as expected. An example of such a descriptor is the `__dict__` attribute of an object. You won't find it in the object's dictionary, because it's a data descriptor stored in the type's dictionary:

```
>>> a.__dict__['__dict__']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '__dict__'
>>> A.__dict__['__dict__']
<attribute '__dict__' of 'A' objects>
>>> a.__dict__ is A.__dict__['__dict__'].__get__(a)
True
```

The `tp_descr_get` slot of this descriptor returns the object's dictionary located at `tp_dictoffset`. Now suppose that data descriptors don't take precedence over instance variables. What would happened then if we put `'__dict__'` in the object's dictionary and assigned it some other dictionary:

```
>>> a.__dict__['__dict__'] = {}
```

The `a.__dict__` attribute would return not the object's dictionary but the dictionary we assigned! That would be totally unexpected for someone who relies on `__dict__`. Fortunately, data descriptors do take precedence over instance variables, so we get the object's dictionary:

```
>>> a.__dict__
{'x': 'instance attribute', 'g': <function <lambda> at 0x108a4cc10>, '__dict__': {}}
```

Non-data descriptors don't take precedence over instance variables, so that most of the time instance variables have a priority over type variables. Of course, the existing order of precedence is one of many design choices. Guido van Rossum explains the reasoning behind it in PEP 252:

> *In the more complicated case, there's a conflict between names stored in the instance dict and names stored in the type dict. If both dicts have an entry with the same key, which one should we return? Looking at classic Python for guidance, I find conflicting rules: for class instances, the instance dict overrides the class dict, **except** for the special attributes (like `__dict__` and `__class__`), which have priority over the instance dict.*
>
> *I resolved this with the following set of rules, implemented in `PyObject_GenericGetAttr()`: ...*

Why is the `__dict__` attribute implemented as a descriptor in the first place? Making it an instance variable would lead to the same problem. It would be possible to override the `__dict__` attribute and hardly anyone wants to have this possibility.

We've learned how attributes of an ordinary object work. Let's see now how attributes of a type work.

Metatype attribute management

Basically, attributes of a type work just like attributes of an ordinary object. When we set an attribute of a type to some value, CPython puts the value in the type's dictionary:

```
>>> B.x = 'class attribute'
>>> B.__dict__
mappingproxy({'__module__': '__main__', '__doc__': None, 'x': 'class attribute'})
```

When we get the value of the attribute, CPython loads it from the type's dictionary:

```
>>> B.x
'class attribute'
```

If the type's dictionary doesn't contain the attribute, CPython loads the value from the metatype's dictionary:

```
>>> B.__class__
<class 'type'>
>>> B.__class__ is object.__class__
True
```

Finally, if the metatype's dictionary doesn't contain the attribute either, CPython searches for the value in the dictionaries of the metatype's parents...

The analogy with the generic implementation is clear. We just change the words "object" with "type" and "type" with "metatype". However, `type` implements the `tp_getattro` and `tp_setattro` slots in its own way. Why? Let's take a look at the code.

type_setattro()

We begin with the `type_setattro()` function, an implementation of the `tp_setattro` slot:

```c
static int
type_setattro(PyTypeObject *type, PyObject *name, PyObject *value)
{
    int res;
    if (!(type->tp_flags & Py_TPFLAGS_HEAPTYPE)) {
        PyErr_Format(
            PyExc_TypeError,
            "can't set attributes of built-in/extension type '%s'",
            type->tp_name);
        return -1;
    }
    if (PyUnicode_Check(name)) {
        if (PyUnicode_CheckExact(name)) {
            if (PyUnicode_READY(name) == -1)
                return -1;
            Py_INCREF(name);
        }
        else {
            name = _PyUnicode_Copy(name);
            if (name == NULL)
                return -1;
        }
        // ... ifdef
    }
    else {
        /* Will fail in _PyObject_GenericSetAttrWithDict. */
        Py_INCREF(name);
    }

    // Call the generic set function.
    res = _PyObject_GenericSetAttrWithDict((PyObject *)type, name, value, NULL);
    if (res == 0) {
        PyType_Modified(type);

        // If attribute is a special method,
        // add update the corresponding slots.
        if (is_dunder_name(name)) {
            res = update_slot(type, name);
        }
        assert(_PyType_CheckConsistency(type));
    }
    Py_DECREF(name);
    return res;
}
```

This function calls generic `_PyObject_GenericSetAttrWithDict()` to set the attribute value, but it does something else too. First, it ensures that the type is not a statically defined type, because such types are designed to be immutable:

```
>>> int.x = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'int'
```

It also checks whether the attribute is a special method. If the attribute is a special method, it updates the slots corresponding to that special method. For example, if we define the `__add__()` special method on an existing class, it will set the `nb_add` slot of the class to the default implementation that calls the method. Due to this mechanism, special methods and slots of a class are kept in sync.

type_getattro()

The `type_getattro()` function, an implementation of the `tp_getattro` slot, doesn't call the generic function but resembles it:

```
/* This is similar to PyObject_GenericGetAttr(),
   but uses _PyType_Lookup() instead of just looking in type->tp_dict. */
static PyObject *
type_getattro(PyTypeObject *type, PyObject *name)
{
    PyTypeObject *metatype = Py_TYPE(type);
    PyObject *meta_attribute, *attribute;
    descrgetfunc meta_get;
    PyObject* res;

    if (!PyUnicode_Check(name)) {
        PyErr_Format(PyExc_TypeError,
                     "attribute name must be string, not '%.200s'",
                     Py_TYPE(name)->tp_name);
        return NULL;
    }

    /* Initialize this type (we'll assume the metatype is initialized) */
    if (type->tp_dict == NULL) {
        if (PyType_Ready(type) < 0)
            return NULL;
    }

    /* No readable descriptor found yet */
    meta_get = NULL;

    /* Look for the attribute in the metatype */
    meta_attribute = _PyType_Lookup(metatype, name);

    if (meta_attribute != NULL) {
        Py_INCREF(meta_attribute);
        meta_get = Py_TYPE(meta_attribute)->tp_descr_get;

        if (meta_get != NULL && PyDescr_IsData(meta_attribute)) {
            /* Data descriptors implement tp_descr_set to intercept
             * writes. Assume the attribute is not overridden in
             * type's tp_dict (and bases): call the descriptor now.
             */
            res = meta_get(meta_attribute, (PyObject *)type,
                           (PyObject *)metatype);
            Py_DECREF(meta_attribute);
            return res;
        }
    }

    /* No data descriptor found on metatype. Look in tp_dict of this
     * type and its bases */
    attribute = _PyType_Lookup(type, name);
    if (attribute != NULL) {
        /* Implement descriptor functionality, if any */
        Py_INCREF(attribute);
        descrgetfunc local_get = Py_TYPE(attribute)->tp_descr_get;

        Py_XDECREF(meta_attribute);

        if (local_get != NULL) {
            /* NULL 2nd argument indicates the descriptor was
             * found on the target object itself (or a base)  */
            res = local_get(attribute, (PyObject *)NULL,
                            (PyObject *)type);
            Py_DECREF(attribute);
            return res;
        }

        return attribute;
    }

    /* No attribute found in local __dict__ (or bases): use the
     * descriptor from the metatype, if any */
    if (meta_get != NULL) {
        PyObject *res;
        res = meta_get(meta_attribute, (PyObject *)type,
                       (PyObject *)metatype);
        Py_DECREF(meta_attribute);
        return res;
    }

    /* If an ordinary attribute was found on the metatype, return it now */
    if (meta_attribute != NULL) {
        return meta_attribute;
    }

    /* Give up */
    PyErr_Format(PyExc_AttributeError,
                 "type object '%.50s' has no attribute '%U'",
                 type->tp_name, name);
    return NULL;
}
```

This algorithm indeed repeats the logic of the generic implementation but with three important differences:

- It gets the type's dictionary via `tp_dict`. The generic implementation would try to locate it using metatype's `tp_dictoffset`.
- It searches for the type variable not only in the type's dictionary but also in the dictionaries of the type's parents. The generic implementation would handle a type like an ordinary object that has no notions of inheritance.
- It supports type descriptors. The generic implementation would support only metatype descriptors.

As a result, we have the following order of precedence:

1. metatype data descriptors
2. type descriptors and other type variables
3. metatype non-data descriptors and other metatype variables.

That's how `type` implements the `tp_getattro` and `tp_setattro` slots. Since `type` is the metatype of all built-in types and the metatype of all classes by default, attributes of most types work according to this implementation. Classes themselves, as we've already said, use the generic implementation by default. If we want to change the behavior of attributes of a class instance or the behavior of attributes of a class, we need to define a new class or a new metaclass that uses a custom implementation. Python provides an easy way to do this.

Custom attribute management

The `tp_getattro` and `tp_setattro` slots of a class are initially set by the `type_new()` function that creates new classes. The generic implementation is its default choice. A class can customize attribute access, assignment and deletion by defining the `__getattribute__()`, `__getattr__()`, `__setattr__()` and `__delattr__()` special methods. When a class defines `__setattr__()` or `__delattr__()`, its `tp_setattro` slot is set to the `slot_tp_setattro()` function. When a class defines `__getattribute__()` or `__getattr__()`, its `tp_getattro` slot is set to the `slot_tp_getattr_hook()` function.

The `__setattr__()` and `__delattr__()` special methods are quite straightforward. Basically, they allow us to implement the `tp_setattro` slot in Python. The `slot_tp_setattro()` function simply calls `__delattr__(instance, attr_name)` or `__setattr__(instance, attr_name, value)` depending on whether the `value` is `NULL` or not:

```
static int
slot_tp_setattro(PyObject *self, PyObject *name, PyObject *value)
{
    PyObject *stack[3];
    PyObject *res;
    _Py_IDENTIFIER(__delattr__);
    _Py_IDENTIFIER(__setattr__);

    stack[0] = self;
    stack[1] = name;
    if (value == NULL) {
        res = vectorcall_method(&PyId___delattr__, stack, 2);
    }
    else {
        stack[2] = value;
        res = vectorcall_method(&PyId___setattr__, stack, 3);
    }
    if (res == NULL)
        return -1;
    Py_DECREF(res);
    return 0;
}
```

The `__getattribute__()` and `__getattr__()` special methods provide a way to customize attribute access. Both take an instance and an attribute name as their parameters and return the attribute value. The difference between them is when they get invoked.

The `__getattribute__()` special method is the analog of `__setattr__()` and `__delattr__()` for getting the value of an attribute. It's invoked instead of the generic function. The `__getattr__()` special method is used in tandem with `__getattribute__()` or the generic function. It's invoked when `__getattribute__()` or the generic function raise `AttributeError`. This logic is implemented in the `slot_tp_getattr_hook()` function:

```
static PyObject *
slot_tp_getattr_hook(PyObject *self, PyObject *name)
{
    PyTypeObject *tp = Py_TYPE(self);
    PyObject *getattr, *getattribute, *res;
    _Py_IDENTIFIER(__getattr__);

    getattr = _PyType_LookupId(tp, &PyId___getattr__);
    if (getattr == NULL) {
        /* No __getattr__ hook: use a simpler dispatcher */
        tp->tp_getattro = slot_tp_getattro;
        return slot_tp_getattro(self, name);
    }
    Py_INCREF(getattr);

    getattribute = _PyType_LookupId(tp, &PyId___getattribute__);
    if (getattribute == NULL ||
        (Py_IS_TYPE(getattribute, &PyWrapperDescr_Type) &&
         ((PyWrapperDescrObject *)getattribute)->d_wrapped ==
         (void *)PyObject_GenericGetAttr))
        res = PyObject_GenericGetAttr(self, name);
    else {
        Py_INCREF(getattribute);
        res = call_attribute(self, getattribute, name);
        Py_DECREF(getattribute);
    }
    if (res == NULL && PyErr_ExceptionMatches(PyExc_AttributeError)) {
        PyErr_Clear();
        res = call_attribute(self, getattr, name);
    }
    Py_DECREF(getattr);
    return res;
}
```

Let's translate the code to English:

1. If the class doesn't define `__getattr__()`, first set its `tp_getattro` slot to another function, `slot_tp_getattro()`, then call this function and return the result of the call.
2. If the class defines `__getattribute__()`, call it. Otherwise call generic `PyObject_GenericGetAttr()`.
3. If the call from the previous step raised `AttributeError`, call `___getattr__()`.
4. Return the result of the last call.

The `slot_tp_getattro()` function is an implementation of the `tp_getattro` slot that CPython uses when a class defines `__getattribute__()` but not `__getattr__()`. This function just calls `__getattribute__()`:

```
static PyObject *
slot_tp_getattro(PyObject *self, PyObject *name)
{
    PyObject *stack[2] = {self, name};
    return vectorcall_method(&PyId___getattribute__, stack, 2);
}
```

Why doesn't CPython set the `tp_getattro` slot to the `slot_tp_getattro()` function instead of `slot_tp_getattr_hook()` initially? The reason is the design of the mechanism that maps special methods to slots. It requires special methods that map to the same slot to provide the same implementation for that slot. And the `__getattribute__()` and `__getattr__()` special methods map to the same `tp_getattro` slot.

Even a perfect understanding of how the `__getattribute__()` and `__getattr__()` special methods work doesn't tell us why we need both of them. Theoretically, `__getattribute__()` should be enough to make attribute access work in any way we want. Sometimes, though, it's more convenient to define `__getattr__()`. For example, the standard `imaplib` module provides the `IMAP4` class that can be used to talk to a IMAP4 server. To issue the commands, we call the class methods. Both lowercase and uppercase versions of the commands work:

```
>>> from imaplib import IMAP4_SSL # subclass of IMAP4
>>> M = IMAP4_SSL("imap.gmail.com", port=993)
>>> M.noop()
('OK', [b'Nothing Accomplished. p11mb154389070lti'])
>>> M.NOOP()
('OK', [b'Nothing Accomplished. p11mb154389070lti'])
```

To support this feature, `IMAP4` defines `__getattr__()`:

```
class IMAP4:
    # ...

    def __getattr__(self, attr):
        #       Allow UPPERCASE variants of IMAP4 command methods.
        if attr in Commands:
            return getattr(self, attr.lower())
        raise AttributeError("Unknown IMAP4 command: '%s'" % attr)

    # ...
```

Achieving the same result with `__getattribute__()` would require us to explicitly call the generic function first: `object.__getattribute__(self, attr)`. Is this inconvenient enough to introduce another special method? Perhaps. The real reason, tough, why both `__getattribute__()` and `__getattr__()` exist is historical. The `__getattribute__()` special method was introduced in Python 2.2 when `__getattr__()` had already existed. Here's how Guido van Rossum explained the need for the new feature:

> *The `__getattr__()` method is not really the implementation for the get-attribute operation; it is a hook that only gets invoked when an attribute cannot be found by normal means. This has often been cited as a shortcoming - some class designs have a legitimate need for a get-attribute method that gets called for all attribute references, and this problem is solved now by making `__getattribute__()` available.*

What happens when we get or set an attribute of a Python object? I think we gave a detailed answer to this question. The answer, however, doesn't cover some important aspects of Python attributes. Let's discuss them as well.

Loading methods

We saw that a function object is a descriptor that returns a method object when we bound it to an instance:

```
>>> a.f
<bound method <lambda> of <__main__.A object at 0x108a20d60>>
```

But is it really necessary to create a method object if all we need to do is to call the method? Couldn't CPython just call the original function with the instance as the first argument? It could. In fact, this is exactly what CPython does.

When the compiler sees the method call with positional arguments like `obj.method(arg1,...,argN)`, it does not produce the `LOAD_ATTR` opcode to load the method and the `CALL_FUNCTION` opcode to call the method. Instead, it produces a pair of the `LOAD_METHOD` and `CALL_METHOD` opcodes:

```
$ echo 'obj.method()' | python -m dis
  1           0 LOAD_NAME                0 (obj)
              2 LOAD_METHOD              1 (method)
              4 CALL_METHOD              0
...
```

When the VM executes the `LOAD_METHOD` opcode, it calls the `_PyObject_GetMethod()` function to search for the attribute value. This function works just like the generic function. The only difference is that it checks whether the value is an unbound method, i.e. a descriptor that returns a method-like object bound to the instance. In this case, it doesn't call the `tp_descr_get` slot of the descriptor's type but returns the descriptor itself. For example, if the attribute value is a function, `_PyObject_GetMethod()` returns the function. The `function` type and other descriptor types whose objects act as unbound methods specify the `Py_TPFLAGS_METHOD_DESCRIPTOR` flag in their `tp_flags`, so it's easy to identify them.

It should be noted that `_PyObject_GetMethod()` works as described only when the object's type uses the generic implementation of `tp_getattro`. Otherwise, it just calls the custom implementation and doesn't perform any checks.

If `_PyObject_GetMethod()` finds an unbound method, the method must be called with the instance prepended to the list of arguments. If it finds some other callable that doesn't need to be bound to the instance, the list of arguments must be kept unchanged. Therefore, after the VM has executed `LOAD_METHOD`, the values on the stack can be arranged in one of two ways:

- an unbound method and a list of arguments including the instance: `(method | self | arg1 | ... | argN)`
- other callable and a list of arguments without the instance `(NULL | method | arg1 | ... | argN)`

The `CALL_METHOD` opcode exists to call the method appropriately in each of these cases.

To learn more about this optimization, check out the issue that originated it.

Listing attributes of an object

Python provides the built-in `dir()` function that can be used to view what attributes an object has. Have you ever wondered how this function finds the attributes? It's implemented by calling the `__dir__()` special method of the object's type. Types rarely define their own `__dir__()`, yet all the types have it. This is because the `object` type defines `__dir__()`, and all other types inherit from `object`. The implementation provided by `object` lists all the attributes stored in the object's dictionary, in the type's dictionary and in the dictionaries of the type's parents. So, `dir()` effectively returns all the attributes of an ordinary object. However, when we call `dir()` on a type, we don't get all its attributes. This is because `type` provides its own implementation of `__dir__()`. This implementation returns attributes stored in the type's dictionary and in the dictionaries of the type's parents. It, however, ignores attributes stored in the metatype's dictionary and in the dictionaries of the metatype's parents. The documentation explains why this is the case:

> *Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.*

Where attributes of types come from

Take any built-in type and list its attributes. You'll get quite a few:

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '_
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc
```

We saw last time that the special methods that correspond to slots are added automatically by the `PyType_Ready()` function that initializes types. But where do the rest attributes come from? They all must be specified somehow and then be set to something at some point. This is a vague statement. Let's make it clear.

The most straightforward way to specify attributes of a type is to create a new dictionary, populate it with attributes and set type's `tp_dict` to that dictionary. We cannot do that before built-in types are defined, so `tp_dict` of built-in types is initialized to `NULL`. It turns out that the `PyType_Ready()` function creates dictionaries of built-in types at runtime. It is also responsible for adding all the attributes.

First, `PyType_Ready()` ensures that a type has a dictionary. Then, it adds attributes to the dictionary. A type tells `PyType_Ready()` which attributes to add by specifying the `tp_methods`, `tp_members` and `tp_getset` slots. Each slot is an array of structs that describe different kinds of attributes.

tp_methods

The `tp_methods` slot is an array of the `PyMethodDef` structs that describe methods:

```
struct PyMethodDef {
    const char  *ml_name;   /* The name of the built-in function/method */
    PyCFunction ml_meth;    /* The C function that implements it */
    int         ml_flags;   /* Combination of METH_xxx flags, which mostly
                               describe the args expected by the C func */
    const char  *ml_doc;    /* The __doc__ attribute, or NULL */
};
typedef struct PyMethodDef PyMethodDef;
```

The `ml_meth` member is a pointer to a C function that implements the method. Its signature can be one of many. The `ml_flags` bitfield is used to tell CPython how exactly to call the function.

For each struct in `tp_methods`, `PyType_Ready()` adds a callable object to the type's dictionary. This object encapsulates the struct. When we call it, the function pointed by `ml_meth` gets invoked. This is basically how a C function becomes a method of a Python type.

The `object` type, for example, defines `__dir__()` and a bunch of other methods using this mechanism:

```
static PyMethodDef object_methods[] = {
    {"__reduce_ex__", (PyCFunction)object___reduce_ex__, METH_O, object___reduce_ex____doc__},
    {"__reduce__", (PyCFunction)object___reduce__, METH_NOARGS, object___reduce____doc__},
    {"__subclasshook__", object_subclasshook, METH_CLASS | METH_VARARGS,
     object_subclasshook_doc},
    {"__init_subclass__", object_init_subclass, METH_CLASS | METH_NOARGS,
     object_init_subclass_doc},
    {"__format__", (PyCFunction)object___format__, METH_O, object___format____doc__},
    {"__sizeof__", (PyCFunction)object___sizeof__, METH_NOARGS, object___sizeof____doc__},
    {"__dir__", (PyCFunction)object___dir__, METH_NOARGS, object___dir____doc__},
    {0}
};
```

The callable object added to the dictionary is usually a method descriptor. We should probably discuss what a method descriptor is in another post on Python callables, but essentially it is an object that behaves like a function object, i.e. it binds to instances. The major difference is that a function bound to an instance returns a method object, and a method descriptor bound to an instance returns a built-in method object. A method object encapsulates a Python function and an instance, and a built-in method object encapsulates a C function and an instance.

For example, `object.__dir__` is a method descriptor:

```
>>> object.__dir__
<method '__dir__' of 'object' objects>
>>> type(object.__dir__)
<class 'method_descriptor'>
```

If we bind `__dir__` to an instance, we get a built-in method object:

```
>>> object().__dir__
<built-in method __dir__ of object object at 0x1088cc420>
>>> type(object().__dir__)
<class 'builtin_function_or_method'>
```

If `ml_flags` flags specifies that the method is static, a built-in method object is added to the dictionary instead of a method descriptor straight away.

Every method of any built-in type either wraps some slot or is added to the dictionary based on `tp_methods`.

tp_members

The `tp_members` slot is an array of the `PyMemberDef` structs. Each struct describes an attribute that exposes a C member of the objects of the type:

```
typedef struct PyMemberDef {
    const char *name;
    int type;
    Py_ssize_t offset;
    int flags;
    const char *doc;
} PyMemberDef;
```

The member is specified by `offset`. Its type is specified by `type`.

For each struct in `tp_members`, `PyType_Ready()` adds a member descriptor to the type's dictionary. A member descriptor is a data descriptor that encapsulates `PyMemberDef`. Its `tp_descr_get` slot takes an instance, finds the member of the instance located at `offset`, converts it to a corresponding Python object and returns the object. Its `tp_descr_set` slot takes an instance and a value, finds the member of the instance located at `offset` and sets it to the C equivalent of the value. A member can be made read-only by specifying `flags`.

By this mechanism, for example, `type` defines `__dictoffset__` and other members:

```
static PyMemberDef type_members[] = {
    {"__basicsize__", T_PYSSIZET, offsetof(PyTypeObject,tp_basicsize),READONLY},
    {"__itemsize__", T_PYSSIZET, offsetof(PyTypeObject, tp_itemsize), READONLY},
    {"__flags__", T_ULONG, offsetof(PyTypeObject, tp_flags), READONLY},
    {"__weakrefoffset__", T_PYSSIZET,
     offsetof(PyTypeObject, tp_weaklistoffset), READONLY},
    {"__base__", T_OBJECT, offsetof(PyTypeObject, tp_base), READONLY},
    {"__dictoffset__", T_PYSSIZET,
     offsetof(PyTypeObject, tp_dictoffset), READONLY},
    {"__mro__", T_OBJECT, offsetof(PyTypeObject, tp_mro), READONLY},
    {0}
};
```

tp_getset

The `tp_getset` slot is an array of the [PyGetSetDef](https://docs.python.org/3/c-api/structures.html#c.PyGetSetDef) structs that desribe arbitrary data descriptors like `property()`:

```
typedef struct PyGetSetDef {
    const char *name;
    getter get;
    setter set;
    const char *doc;
    void *closure;
} PyGetSetDef;
```

For each struct in `tp_getset`, `PyType_Ready()` adds a getset descriptor to the type's dictionary. The `tp_descr_get` slot of a getset descriptor calls the specified `get` function, and the `tp_descr_set` slot of a getset descriptor calls the specified `set` function.

Types define the `__dict__` attribute using this mechanism. Here's, for example, how the `function` type does that:

```
static PyGetSetDef func_getsetlist[] = {
    {"__code__", (getter)func_get_code, (setter)func_set_code},
    {"__defaults__", (getter)func_get_defaults,
     (setter)func_set_defaults},
    {"__kwdefaults__", (getter)func_get_kwdefaults,
     (setter)func_set_kwdefaults},
    {"__annotations__", (getter)func_get_annotations,
     (setter)func_set_annotations},
    {"__dict__", PyObject_GenericGetDict, PyObject_GenericSetDict},
    {"__name__", (getter)func_get_name, (setter)func_set_name},
    {"__qualname__", (getter)func_get_qualname, (setter)func_set_qualname},
    {NULL} /* Sentinel */
};
```

The `__dict__` attribute is implemented not as a read-only member descriptor but as a geteset descriptor because it does more than simply return the dictionary located at `tp_dictoffset`. For instance, the descriptor creates the dictionary if it doesn't exist yet.

Classes also get the `__dict__` attribute by this mechanism. The `type_new()` function that creates classes specifies `tp_getset` before it calls `PyType_Ready()`. Some classes, though, don't get this attribute because their instances don't have dictionaries. These are the classes that define `__slots__`.

__slots__

The `__slots__` attribute of a class enumerates the attributes that the class can have:

```
>>> class D:
...     __slots__ = ('x', 'y')
...
```

If a class defines `__slots__`, the `__dict__` attribute is not added to the class's dictionary and `tp_dictoffset` of the class is set to `0`. The main effect of this is that the class instances don't have dictionaries:

```
>>> D.__dictoffset__
0
>>> d = D()
>>> d.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'D' object has no attribute '__dict__'
```

However, the attributes listed in `__slots__` work fine:

```
>>> d.x = 4
>>> d.x
4
```

How is that possible? The attributes listed in `__slots__` become members of class instances. For each member, the member descriptor is added to the class dictionary. The `type_new()` function specifies `tp_members` to do that.

```
>>> D.x
<member 'x' of 'D' objects>
```

Since instances don't have dictionaries, the `__slots__` attribute saves memory. According to Descriptor HowTo Guide,

> *On a 64-bit Linux build, an instance with two attributes takes 48 bytes with `__slots__` and 152 bytes without.*

The guide also lists other benefits of using `__slots__`. I recommend you check them out.

Summary

The compiler produces the `LOAD_ATTR`, `STORE_ATTR` and `DELETE_ATTR` opcodes to get, set, and delete attributes. To executes these opcodes, the VM calls the `tp_getattro` and `tp_setattro` slots of the object's type. A type may implement these slots in an arbitrary way, but mostly we have to deal with three implementations:

- the generic implementation used by most built-in types and classes
- the implementation used by `type`

- the implementation used by classes that define the `__getattribute__()`, `__getattr__()`, `__setattr__()` and `__delattr__()` special methods.

The generic implementation is straightforward once you understand what descriptors are. In a nutshell, descriptors are attributes that have control over attribute access, assignment and deletion. They allow CPython to implement many features including methods and properties.

Built-in types define attributes using three mechanisms:

- `tp_methods`
- `tp_members`; and
- `tp_getset`.

Classes also use these mechanisms to define some attributes. For example, `__dict__` is defined as a getset descriptor, and the attributes listed in `__slots__` are defined as member descriptors.

P.S.

This post closes the first season of the Python behind the scenes series. We've learned a lot over this time. A lot remains to be covered. The topics on my list include: CPython's memory management, the GIL, the implementation of built-in types, the import system, concurrency and the internals of the standard modules. You can tell me what you would like to read about next time. Send your ideas and preferences to *victor@tenthousandmeters.com*.

See you in 2021. Stay tuned!

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