04 揭开对象神秘的面纱-慕课专栏

面向对象理论中“ **类** ”和“ **对象** ”这两个重要概念，在 *Python* 内部均以对象的形式存在。 “类”是一种对象，称为 **类型对象** ；“类”实例化生成的“对象”也是对象，称为 **实例对象** 。

根据对象不同特点还可进一步分类：

类别	特点
可变对象	对象创建后可以修改
不可变对象	对象创建后不能修改
定长对象	对象大小固定
变长对象	对象大小不固定

那么，对象在 *Python* 内部到底长啥样呢？

由于 *Python* 是由 *C* 语言实现的，因此 *Python* 对象在 *C* 语言层面应该是一个 **结构体** ，组织对象占用的内存。 不同类型的对象，数据及行为均可能不同，因此可以大胆猜测：不同类型的对象由不同的结构体表示。

对象也有一些共性，比如每个对象都需要有一个 **引用计数** ，用于实现 **垃圾回收机制** 。 因此，还可以进一步猜测：表示对象的结构体有一个 **公共头部** 。

到底是不是这样呢？——接下来在源码中窥探一二。

PyObject，对象的基石

在 *Python* 内部，对象都由 *PyObject* 结构体表示，对象引用则是指针 *PyObject* * 。 *PyObject* 结构体定义于头文件 *object.h* ，路径为 *Include/object.h* ，代码如下：

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

除了 *_PyObject_HEAD_EXTRA* 宏，结构体包含以下两个字段：

- **引用计数** (ob_refcnt)
- **类型指针** (ob_type)

引用计数 很好理解：对象被其他地方引用时加一，引用解除时减一； 当引用计数为零，便可将对象回收，这是最简单的垃圾回收机制。 **类型指针** 指向对象的 **类型对象** ，**类型对象** 描述 **实例对象** 的数据及行为。

回过头来看 *_PyObject_HEAD_EXTRA* 宏的定义，同样在 *Include/object.h* 头文件内：

```
#ifdef Py_TRACE_REFS

#define _PyObject_HEAD_EXTRA            \
    struct _object *_ob_next;           \
    struct _object *_ob_prev;

#define _PyObject_EXTRA_INIT 0, 0,

#else
#define _PyObject_HEAD_EXTRA
#define _PyObject_EXTRA_INIT
#endif
```

如果 *Py_TRACE_REFS* 有定义，宏展开为两个指针，看名字是用来实现 **双向链表** 的：

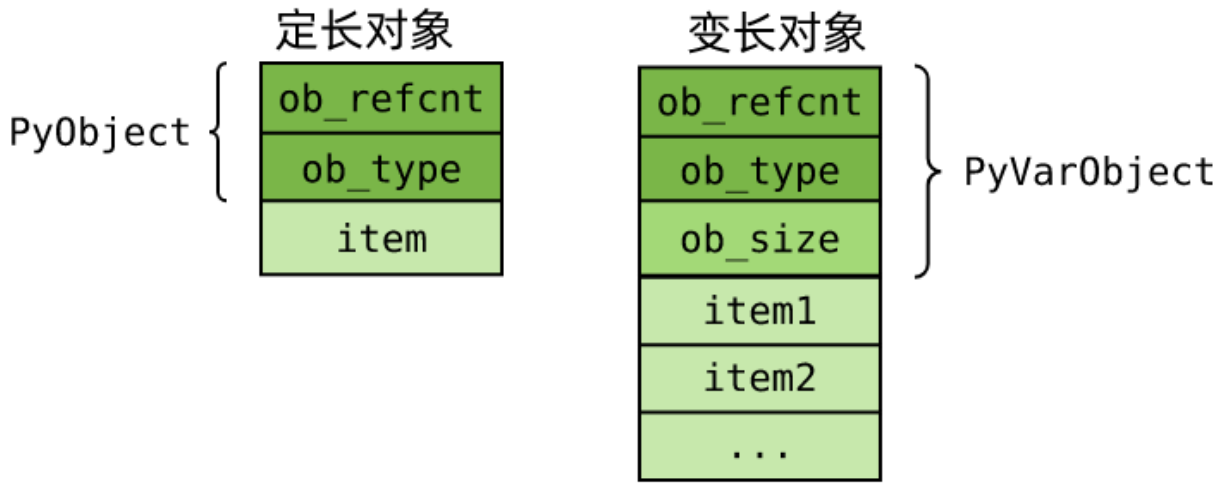
```
struct _object *_ob_next;
struct _object *_ob_prev;
```

结合注释，双向链表用于跟踪所有 **活跃堆对象** ，一般不启用，不深入介绍。

对于 **变长对象** ，需要在 *PyObject* 基础上加入长度信息，这就是 *PyVarObject* ：

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size;
} PyVarObject;
```

变长对象比普通对象多一个字段 *ob_size* ，用于记录元素个数：



定长对象变长对象

至于具体对象，视其大小是否固定，需要包含头部 *PyObject* 或 *PyVarObject* 。 为此，头文件

准备了两个宏定义，方便其他对象使用：

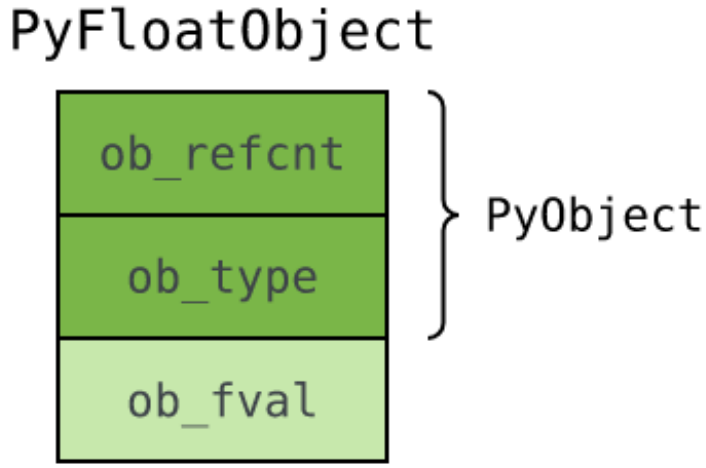
```
#define PyObject_HEAD         PyObject ob_base;
#define PyObject_VAR_HEAD     PyVarObject ob_base;
```

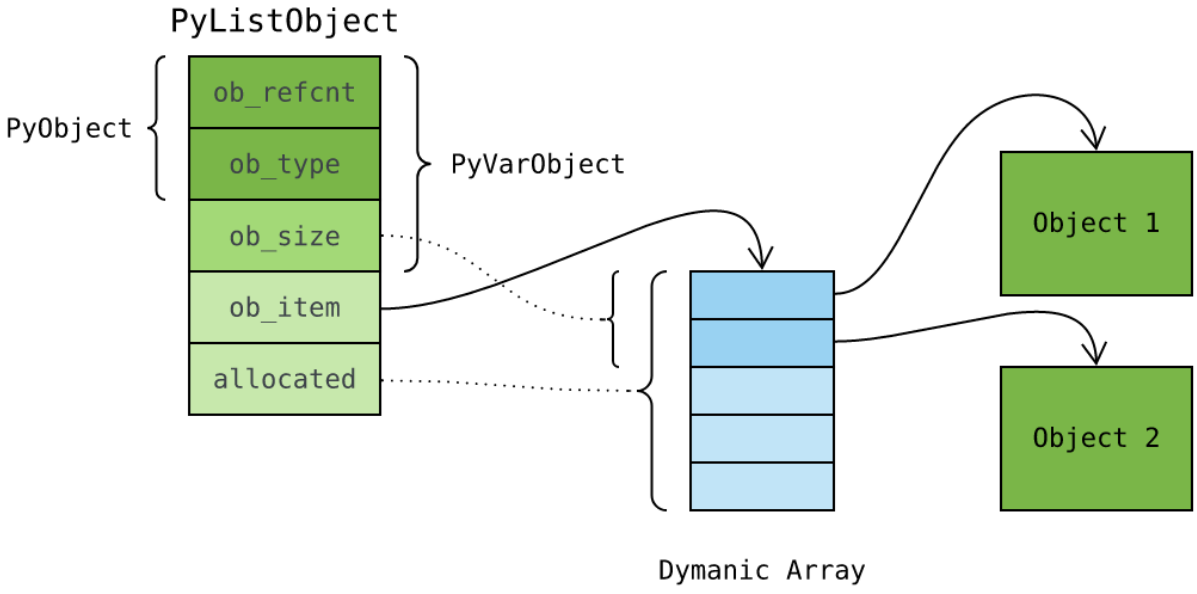
例如，对于大小固定的 **浮点对象** ，只需在 *PyObject* 头部基础上， 用一个 **双精度浮点数** *double* 加以实现：

```
typedef struct {
    PyObject_HEAD

    double ob_fval;
} PyFloatObject;
```

而对于大小不固定的 **列表对象** ，则需要在 *PyVarObject* 头部基础上，用一个动态数组加以实现，数组存储列表包含的对象，即 *PyObject* 指针：

```
typedef struct {
    PyObject_VAR_HEAD

    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```





如图， *PyListObject* 底层由一个数组实现，关键字段是以下 *3* 个：

- *ob_item* ，指向 **动态数组** 的指针，数组保存元素对象指针；
- *allocated* ，动态数组总长度，即列表当前的 **容量** ；
- *ob_size* ，当前元素个数，即列表当前的 **长度** ；

列表容量不足时， *Python* 会自动扩容，具体做法在讲解 *list* 源码时再详细介绍。

最后，介绍两个用于初始化对象头部的宏定义。 其中，*PyObject_HEAD_INIT* 一般用于 **定长对象** ，将引用计数 *ob_refcnt* 设置为 *1* 并将对象类型 *ob_type* 设置成给定类型：

```
#define PyObject_HEAD_INIT(type)        \
    { _PyObject_EXTRA_INIT              \
    1, type },
```

PyVarObject_HEAD_INIT 在 *PyObject_HEAD_INIT* 基础上进一步设置 **长度字段** *ob_size* ，一般用于 **变长对象** ：

```
#define PyVarObject_HEAD_INIT(type, size)       \
    { PyObject_HEAD_INIT(type) size },
```

后续在研读源码过程中，将经常见到这两个宏定义。

PyTypeObject，类型的基石

在 *PyObject* 结构体，我们看到了 *Python* 中所有对象共有的信息。 对于内存中的任一个对象，不管是何类型，它刚开始几个字段肯定符合我们的预期： **引用计数** 、 **类型指针** 以及变长对象特有的 **元素个数** 。

随着研究不断深入，我们发现有一些棘手的问题没法回答：

- 不同类型的对象所需内存空间不同，创建对象时从哪得知内存信息呢？
- 对于给定对象，怎么判断它支持什么操作呢？

对于我们初步解读过的 *PyFloatObject* 和 *PyListObject* ，并不包括这些信息。 事实上，这些作为对象的 **元信息** ，应该由一个独立实体保存，与对象所属 **类型** 密切相关。

注意到， *PyObject* 中包含一个指针 *ob_type* ，指向一个 **类型对象** ，秘密就藏在这里。类型对象 *PyTypeObject* 也在 *Include/object.h* 中定义，字段较多，只讨论关键部分：

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name;
    Py_ssize_t tp_basicsize, tp_itemsize;


    destructor tp_dealloc;
    printfunc tp_print;

    getattrfunc tp_getattr;
    setattrfunc tp_setattr;



    struct _typeobject *tp_base;


} PyTypeObject;
```

可见 **类型对象** *PyTypeObject* 是一个 **变长对象** ，包含变长对象头部。 专有字段有：

- **类型名称** ，即 *tp_name* 字段；
- 类型的继承信息，例如 *tp_base* 字段指向基类对象；
- 创建实例对象时所需的 **内存信息** ，即 *bp_basicsize* 和 *tp_itemsize* 字段；
- 该类型支持的相关 **操作信息** ，即 *tp_print* 、 *tp_getattr* 等函数指针；

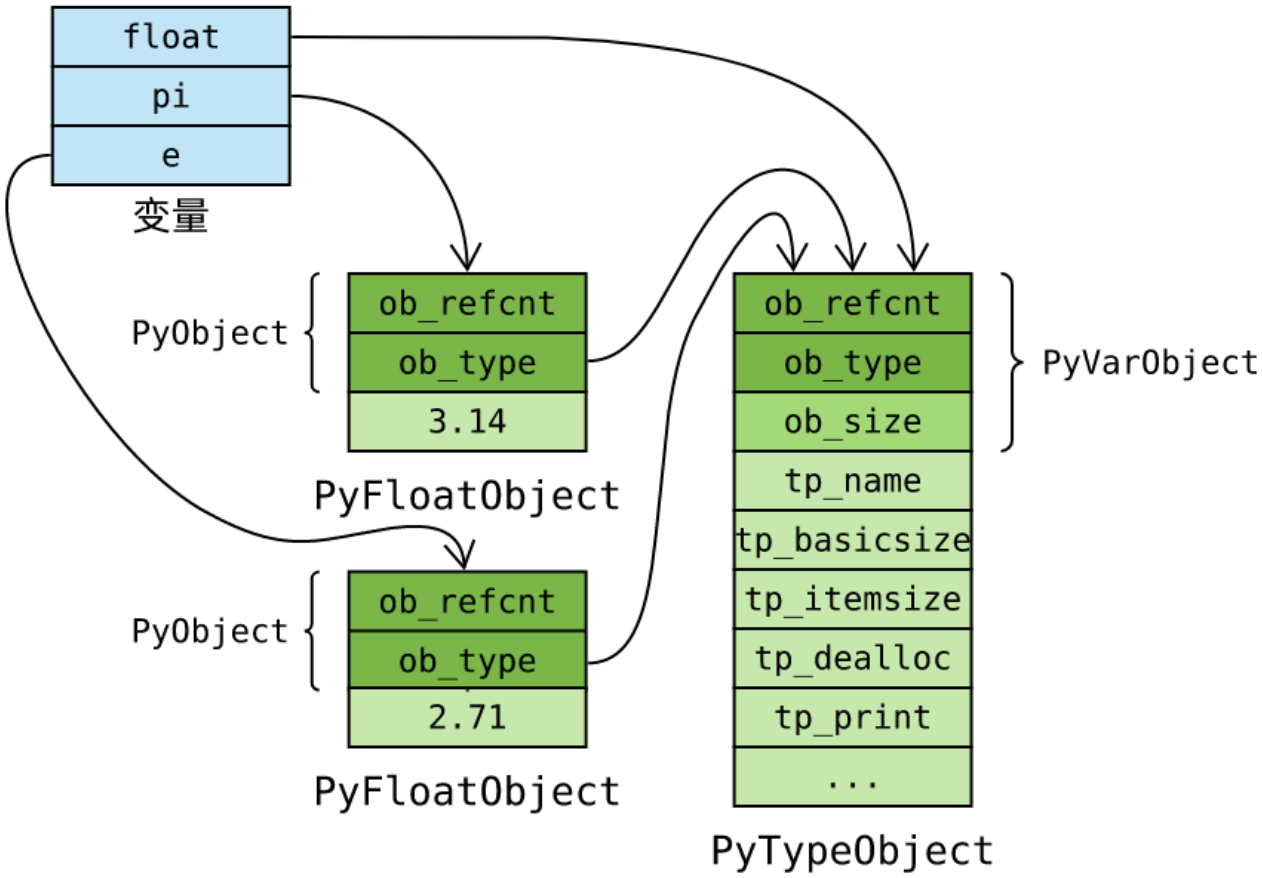
PyTypeObject 就是 **类型对象** 在 *Python* 中的表现形式，对应着面向对象中"**类**"的概念。 *PyTypeObject* 结构很复杂，但是我们不必在此刻完全弄懂它。 先有个大概的印象，知道 *PyTypeObject* 保存着对象的 **元信息** ，描述对象的 **类型** 即可。

接下来，以 **浮点** 为例，考察 **类型对象** 和 **实例对象** 在内存中的形态和关系：

```
>>> float
<class 'float'>
>>> pi = 3.14
>>> e = 2.71
>>> type(pi) is float
True
```

float 为浮点类型对象，系统中只有唯一一个，保存了所有浮点实例对象的元信息。 而浮点实例对象就有很多了，圆周率 *pi* 是一个，自然对数 *e* 是另一个，当然还有其他。

代码中各个对象在内存的形式如下图所示：



其中，两个浮点 **实例对象** 都是 *PyFloatObject* 结构体， 除了公共头部字段 *ob_refcnt* 和 *ob_type* ，专有字段 *ob_fval* 保存了对应的数值。 浮点 **类型对象** 是一个 *PyTypeObject* 结构体， 保存了类型名、内存分配信息以及浮点相关操作。 实例对象 *ob_type* 字段指向类型对象， *Python* 据此判断对象类型， 进而获悉关于对象的元信息，如操作方法等。 再次提一遍，*float* 、 *pi* 以及 *e* 等变量只是一个指向实际对象的指针。

由于浮点 **类型对象** 全局唯一，在 *C* 语言层面作为一个全局变量静态定义即可，*Python* 的确就这么做。 浮点类型对象就藏身于 *Object/floatobject.c* 中， *PyFloat_Type* 是也：

```
PyTypeObject PyFloat_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "float",
    sizeof(PyFloatObject),
    0,
    (destructor)float_dealloc,


    (reprfunc)float_repr,


};
```

其中，第 *2* 行初始化 *ob_refcnt* 、 *ob_type* 以及 *ob_size* 三个字段； 第 *3* 行将 *tp_name* 字段初始化成类型名称 *float* ；再往下是各种操作的函数指针。

注意到 *ob_type* 指针指向 *PyType_Type* ，这也是一个静态定义的全局变量。 由此可见，代表“ **类型的类型** ”即 *type* 的那个对象应该就是 *PyType_Type* 了。

PyType_Type，类型的类型

我们初步考察了 *float* 类型对象，知道它在 *C* 语言层面是 *PyFloat_Type* 全局静态变量。 类型是一种对象，它也有自己的类型，也就是 *Python* 中的 *type* ：

```
>>> float.__class__
<class 'type'>
```

自定义类型也是如此：

```
>>> class Foo(object):
...     pass
...
>>> Foo.__class__
<class 'type'>
```

那么， *type* 在 *C* 语言层面又长啥样呢？

围观 *PyFloat_Type* 时，我们通过 *ob_type* 字段揪住了 *PyType_Type* 。 的确，它就是 *type* 的肉身。 *PyType_Type* 在 *Object/typeobject.c* 中定义：

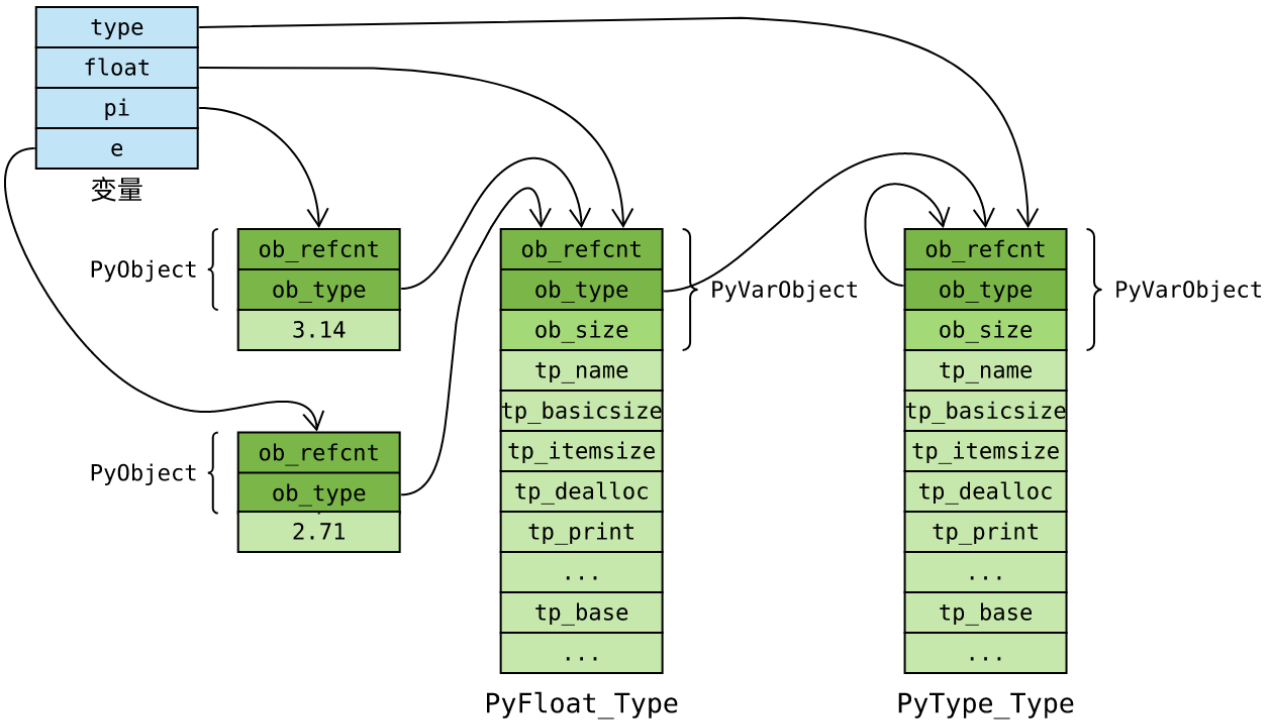
```
PyTypeObject PyType_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "type",
    sizeof(PyHeapTypeObject),
    sizeof(PyMemberDef),
    (destructor)type_dealloc,


    (reprfunc)type_repr,


};
```

内建类型和自定义类对应的 *PyTypeObject* 对象都是这个通过 *PyType_Type* 创建的。 *PyType_Type* 在 *Python* 的类型机制中是一个至关重要的对象，它是所有类型的类型，称为 **元类型** (*meta class*)。 借助元类型，你可以实现很多神奇的高级操作。

注意到， *PyType_Type* 将自己的 *ob_type* 字段设置成它自己(第 *2* 行)，这跟我们在 *Python* 中看到的行为是吻合的：

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

至此，元类型 type 在对象体系里的位置非常清晰了：



PyBaseObject_Type，类型之基

object 是另一个特殊的类型，它是所有类型的基类。 那么，怎么找到它背后的实体呢？ 理论上，通过 *PyFloat_Type* 中 *tp_base* 字段顺藤摸瓜即可。

然而，我们发现这个字段在并没有初始化：

```
0,
```

这又是什么鬼？

接着查找代码中 *PyFloat_Type* 出现的地方，我们在 *Object/object.c* 发现了蛛丝马迹：

```
if (PyType_Ready(&PyFloat_Type) < 0)
    Py_FatalError("Can't initialize float type");
```

敢情 *PyFloat_Type* 静态定义后还是个半成品呀！ *PyType_Ready* 对它做进一步加工，将 *PyFloat_Type* 中 *tp_base* 字段初始化成 *PyBaseObject_Type* ：

```
int
PyType_Ready(PyTypeObject *type)
{


    base = type->tp_base;
    if (base == NULL && type != &PyBaseObject_Type) {
        base = type->tp_base = &PyBaseObject_Type;
        Py_INCREF(base);
    }


}
```

PyBaseObject_Type 就是 *object* 背后的实体，先一睹其真容：

```
PyTypeObject PyBaseObject_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "object",
    sizeof(PyObject),
    0,
    object_dealloc,


    object_repr,
};
```

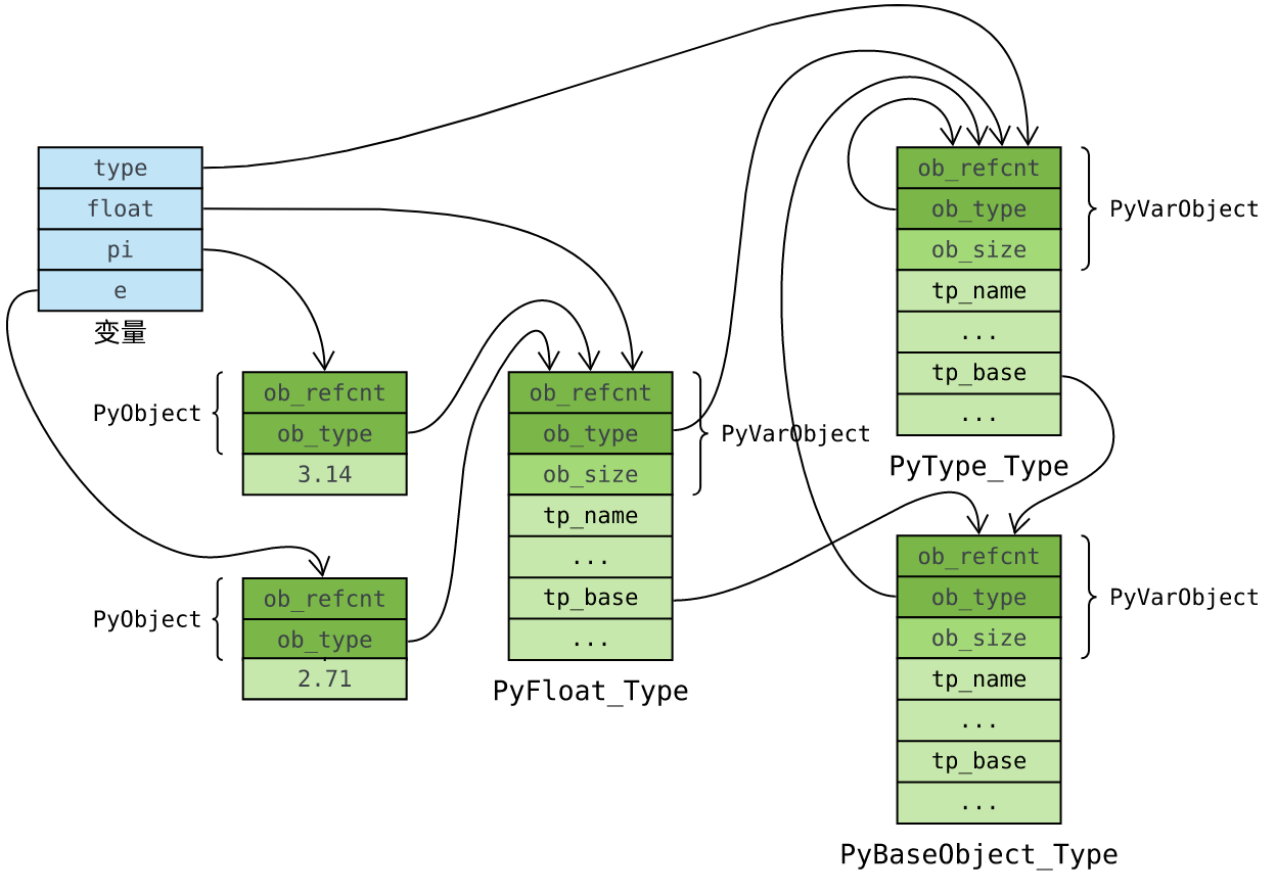
注意到， *ob_type* 字段指向 *PyType_Type* 跟 *object* 在 *Python* 中的行为时相吻合的：

```
>>> object.__class__
<class 'type'>
```

又注意到 *PyType_Ready* 函数初始化 *PyBaseObject_Type* 时，不设置 *tp_base* 字段。 因为继承链必须有一个终点，不然对象沿着继承链进行属性查找时便陷入死循环。

```
>>> print(object.__base__)
None
```

至此，我们完全弄清了 *Python* 对象体系中的所有实体以及关系，得到一幅完整的图画：



虽然很多细节还没来得及研究，这也算是一个里程碑式的胜利！让我们再接再厉！