

CPython Internals 学习笔记

<http://pgbovine.net/cpython-internals.htm>

Based on CPython v3.6.1

2017.05 (v1)

目录

简介	4
版权声明	4
Lecture 1 Interpreter and source code overview-----	5
修改 PYTHON 解释器源码	6
Lecture 2 Opcodes and main interpreter loop-----	7
PYTHON 字节码与反汇编	7
主循环函数中是如何执行 PYTHON 代码的	9
小结	10
参考链接	10
Lecture 3 Frames, function calls and scope -----	11
可视化 PYTHON 代码执行过程	11
FRAME、FUNCTION、CODE OBJECT、BYTECODE	12
CALL_FUNCTION	14
小结	14
Lecture 4 PyObject the core Python object -----	15
PYTHON OBJECT PROTOCOL.....	15
PYOBJECT	16
小结	18
参考链接	18
Lecture 5 Example Python data types -----	19
PYTHON SEQUENCE TYPES.....	19
PYUNICODEOBJECT.....	19
小结	22
Lecture 6 Code objects, function objects, and closures---	23
FUNCTION OBJECT	23
CLOSURE	24
参考链接	25
Lecture 7 Iterators -----	26
ITERATOR.....	26
GET_ITER & FOR_ITER	26

小结	30
Lecture 8 User-defined classes and objects-----	31
NEW STYLE CLASSES VS OLD STYLE CLASSES	31
LOAD_BUILD_CLASS	31
参考链接	33
Lecture 9 - Generators-----	34
YIELD_VALUE	34
PYGENOBJECT	35
小结	36

简介

本课程《CPython internals: A ten-hour codewalk through the Python interpreter source code》共包含九个章节，基于 Python 官方实现 CPython，概述了 Python 源码的基本原理。原课程为 UCSD 的 Philip Guo 在 2014 年开授的 CSC253 Dynamic Languages and Software Development 中的一部分，课程在线视频地址为：

- YouTube: https://www.youtube.com/watch?v=Lhadel7_EIU
- 优酷: http://v.youku.com/v_show/id_XMTQ0NzY5ODcyOA==.html

原课程基于 Python2.7.8/Windows，我学习使用的系统环境为 Python3.6.1/Ubuntu 14.04。课程内容如下：

- Lecture 1 - Interpreter and source code overview
- Lecture 2 - Opcodes and main interpreter loop
- Lecture 3 - Frames, function calls, and scope
- Lecture 4 - PyObject: The core Python object
- Lecture 5 - Example Python data types
- Lecture 6 - Code objects, function objects, and closures
- Lecture 7 - Iterators
- Lecture 8 - User-defined classes and objects
- Lecture 9 - Generators

版权声明

全文内容基于开源课程、项目，**切勿用于任何商业用途**；

欢迎转载、分享，请**保持署名**：<http://blog.rainy.im>；

如发现错误或意见，欢迎发送邮件至：me@rainy.im；

如果本书内容对您有所帮助，欢迎扫码打赏支持~ 



LECTURE 1 INTERPRETER AND SOURCE CODE OVERVIEW

首先下载 Python3.6.1 源码并编译（为了不覆盖系统自带的 Python，只编译不安装）：

```
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tgz  
tar zxvf Python-3.6.1.tgz  
cd Python-3.6.1/  
../configure  
make
```

第一节主要是介绍 Python 解释器的由来以及 CPython 源码的目录结构。CPython 是 Python 官方的实现版本，也就是由 C 语言写成，其源码（也是 CPython Internals 课程主要学习的内容）都是 *.c 和 *.h。

从 CPython 源码，经过 gcc（或其它）编译器编译得到可执行程序 python，既是 Python 脚本的解释器，也是实时交互的 REPL 程序。源码目录如下图所示：

```
→ Python-3.6.1    tree -d -L 1  
.  
| -- build  
| -- Doc  
| -- Grammar  
| -- Include  
| -- Lib  
| -- Mac  
| -- Misc  
| -- Modules  
| -- Objects  
| -- Parser  
| -- PC  
| -- PCbuild  
| -- Programs  
| -- Python  
`-- Tools  
  
15 directories
```

其中我们关心的有：

- `Include/`, 相关头文件;
- `Objects/`, 定义 Python 内置类型;
- `Python/`, Python 解释器运行时程序（包括 REPL 主循环程序）。

其它如 **Modules**、**Lib** 是与 **Python** 标准库相关的代码，则不在本课程考虑范围，因为它们通常是为 **Python** 提供额外功能的支持，对 **Python** 核心程序影响不大。

修改 **Python** 解释器源码

执行 **make** 命令会将所有源码进行编译，这个过程可能会比较漫长（取决于机器性能），但是我们在调试过程中修改了某个源文件的代码，再进行重新编译就不需要花那么久的时间。例如，修改 **Python REPL** 解释器主循环程序的代码：

```
1108     for (;;) {
1109         printf("Hello Python3.6");
1110         assert(stack_pointer >= f->f_valuestack); /* else underflow */
1111         assert(STACK_LEVEL() <= co->co_stacksize); /* else overflow */
1112         assert(!PyErr_Occurred());
1113
1114         /* Do periodic things. Doing this every time through
1115            the loop would add too much overhead, so we do it
1116            only every Nth instruction. We also do it if
1117            ``pendingcalls_to_do'' is set, i.e. when an asynchronous
1118            event needs attention (e.g. a signal handler or
1119            async I/O handler); see Py_AddPendingCall() and
1120            Py_MakePendingCalls() above. */
1121
1122         if (_Py_atomic_load_relaxed(&eval_breaker)) {
1123             if (_Py_OPCODE(*next_instr) == SETUP_FINALLY) {
1124                 /* Make the last opcode before
1125                    a try: finally: block uninterruptible. */
1126                 goto fast_next_opcode;
1127             }
1128             if (_Py_atomic_load_relaxed(&pendingcalls_to_do)) {
1129                 if (Py_MakePendingCalls() < 0)
1130                     goto error;
1131             }
1132 #ifdef WITH_THREAD
1133 }
```

[cpython] 1:vim* 2:/-

这时再执行 **make** 命令就会快得多。此时执行 **./python -c “print(123)”**，将会看到满屏幕的 Hello Python3.6 中间夹着一个 123。

LECTURE 2 OPCODES AND MAIN INTERPRETER LOOP

Python 源码（代码字符串或 `*.py` 文件）首先被解释器编译成字节码（bytecode），本节课从探索字节码开始，引出对解释器主循环函数的基本认识。Python 3.6 中的解释器主循环函数相比 2.7 已经有了较大的改动，但是基本的框架还是一样，只是需要稍加探索。另外，课程中教授讲解字节码的例子直接使用文件：

```
c = compile("test.py", "test.py", "exec")
list(c.co_code)

./python -m dis test.py
```

这导致 `compile` 返回的字节码与 `dis` 输出的指令代码无法一一对应，对于这个问题的原因可能是从源文件直接 `compile`，其操作过程是不同的，因为我更改了 `test.py` 的内容，其字节码是不变的。因此，我采用直接在解释器内定义的方式，结果可以更准确地与 `dis` 模块反汇编的结果一一对应。

Python 字节码与反汇编

```
→ Python-3.6.1 ./python
Python 3.6.1 (default, May 10 2017, 10:49:45)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def foo():
...     x = 1
...     y = 2
...     return x + y
...
>>> bc = foo.__code__
>>> bc
<code object foo at 0x7f6235207ae0, file "<stdin>", line 1>
>>> list(bc.co_code)
[100, 1, 125, 0, 100, 2, 125, 1, 124, 0, 124, 1, 23, 0, 83, 0]
>>> █
```

上面的代码中，首先定义了一个函数 `foo()`，然后可以通过 `__code__` 获取该函数的代码对象（`code object`），而代码对象中的 `co_code` 属性即为这段代码的字节码。在 Python 3.6 中，字节码是以 `bytes` 类型存储的（在 2.7 还是 `str`），因此可以直接通过 `list()` 转换为整数列表：

```
[100, 1,  
 125, 0,  
 100, 2,  
 125, 1,  
 124, 0,  
 124, 1,  
 23, 0,  
 83, 0]
```

这一串数字（十进制）就是 `foo` 函数经过编译之后在 **Python** 虚拟机中的样子。我将其按照两两一对的形式排列，是为了与 `dis` 反汇编的结果进行比较（右图）。

`dis.dis(bc)` 将该字节码进行反编译，打印出对应的操作。其中第一列是代码的行号（我们的 `foo` 函数只有

2、3、4三行）；第二列为指令的偏移量，在 **Python3.6** 中每条指令使用两个字节存储，因此所有的偏移量都是 2；第三列为具体指令，例如代码中的第2行 `x = 1`，被编译成两条指令：`LOAD_CONST` 和 `STORE_FAST`；第四列为该条指令的参数（如果有的话）；最后括号里是说明信息。

比较上图中的反汇编结果与最上面的字节码列表，可以发现反汇编结果的第四列与上面数组的第二列相同（这就是我将它成对排列的原因）；这样一来，自然会想到，反汇编结果中的第三列是否也对应字节码中的第一列？事实上就是如此，我们可以从 `Include/opcode.h` 头文件中找到所有这些指令对应的操作码：

```
#define STORE_GLOBAL 97  
#define DELETE_GLOBAL 98  
#define LOAD_CONST 100  
#define LOAD_NAME 101  
#define BUILD_TUPLE 102  
#define BUILD_LIST 103  
#define BUILD_SET 104  
#define BUILD_MAP 105  
#define LOAD_ATTR 106  
#define COMPARE_OP 107  
#define IMPORT_NAME 108  
#define IMPORT_FROM 109  
#define JUMP_FORWARD 110  
#define JUMP_IF_FALSE_OR_POP 111  
#define JUMP_IF_TRUE_OR_POP 112  
#define JUMP_ABSOLUTE 113  
#define POP_JUMP_IF_FALSE 114  
#define POP_JUMP_IF_TRUE 115  
#define LOAD_GLOBAL 116  
#define CONTINUE_LOOP 119  
#define SETUP_LOOP 120  
#define SETUP_EXCEPT 121  
#define SETUP_FINALLY 122  
#define LOAD_FAST 124  
#define STORE_FAST 125
```

主循环函数中是如何执行 Python 代码的

根据 PEP 523，对 frame evaluation 的 API 进行了更新 ([Python/ceval.c](#))，首先我们找到 PyEval_EvalFrameEx 函数的定义：

```
714 PyObject *
715 PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
716 {
717     PyThreadState *tstate = PyThreadState_GET();
718     return tstate->interp->eval_frame(f, throwflag);
719 }
```

所谓的 frame（具体将在下一节展开），或是 execution frame，它用于存储当前执行的代码对象（code object）相关的信息，例如局部作用域（local scope）、流程控制的 blockstack、字节码中的指令所操作的 value stack 等。其中 value stack 就是用于存储上面 dis 展示的反汇编结果中的第四列参数。

根据上面的定义，我们需要找到 eval_frame 函数，定义在 [Python/pystate.c](#)（右图）：

```
69 PyInterpreterState *
70 PyInterpreterState_New(void)
71 {
72     PyInterpreterState *interp = (PyInterpreterState *) PyMem_RawMalloc(sizeof(PyInterpreterState));
73
74     if (interp != NULL) {
75         HEAD_INIT();
76 #ifdef WITH_THREAD
77         if (head_mutex == NULL)
78             Py_FatalError("Can't initialize threads for interpreter");
79 #endif
80         interp->modules = NULL;
81         interp->modules_by_index = NULL;
82         interp->sysdict = NULL;
83         interp->builtins = NULL;
84         interp->builtins_copy = NULL;
85         interp->tstate_head = NULL;
86         interp->codec_search_path = NULL;
87         interp->codec_search_cache = NULL;
88         interp->codec_error_registry = NULL;
89         interp->codecs_initialized = 0;
90         interp->fscodec_initialized = 0;
91         interp->importlib = NULL;
92         interp->import_func = NULL;
93         interp->eval_frame = _PyEval_EvalFrameDefault;
94     }
95 #ifdef HAVE_DLOPEN
```

根据红色标记的那一行，可以发现 eval_frame 实际指向的是 _PyEval_EvalFrameDefault，再回到 ceval.c，可以找到这个函数的定义（右图）。这个函数很长，一直到第 3701 行，其中 主循环函数（1108行） 也在这个函数范围之内。虽然 Python 3.6

```
721 PyObject *
722 _PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
723 {
724 #ifdef DXPAIRS
725     int lastopcode = 0;
726 #endif
727     PyObject **stack_pointer; /* Next free slot in value stack */
728     const _Py_CODEUNIT *next_instr;
729     int opcode; /* Current opcode */
730     int oparg; /* Current opcode argument, if any */
731     enum why_code why; /* Reason for block stack unwind */
732     PyObject **fastlocals, **freevars;
733     PyObject *retval = NULL; /* Return value */
734     PyThreadState *tstate = PyThreadState_GET();
735     PyCodeObject *co;
```

API 有所升级，但是这个函数对应 2.7 版本中的 `PyEval_EvalFrameEx`，其主要框架是不变的。

再看上图中红色标记的两行，`opcode` 和 `oparg` 就是分别对应 `dis` 反汇编中的操作码和参数（第三列与第四列），也就是说 `Python` 主函数（不停循环）依次读取字节码，并通过一个巨大的 `switch-case` 分支来将操作码和参数匹配到对应的操作，为方便理解，我们可以将主函数简写为：

```
1 PyObject *
2 _PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
3 {
4     int opcode;
5     int oparg;
6     co = f->f_code;
7     first_instr = (_Py_CODEUNIT *) PyBytes_AS_STRING(co->co_code);
8     next_instr = first_instr;
9
10    for (;;){
11        NEXTOPARG()
12
13        switch (opcode) {
14            case ...
15        }
16    }
17 }
```

其中 `NEXTOPARG()` 是定义在 [869](#) 行的宏：

```
869 #define NEXTOPARG() do { \
870     _Py_CODEUNIT word = *next_instr; \
871     opcode = _Py_OPCODE(word); \
872     oparg = _Py_OPARG(word); \
873     next_instr++; \
874 } while (0)
```

小结

到目前为止，我们已经从宏观的角度大致了解了 `CPython` 是如何执行 `Python` 的字节码的，后面的课程将会在此基础之上探讨更多细节问题。

参考链接

- [探索 Python 代码对象](#)
- [How should I understand the output of dis.dis?](#)
- [Patching Function Bytecode in Python](#)
- [PEP 523 -- Adding a frame evaluation API to CPython](#)

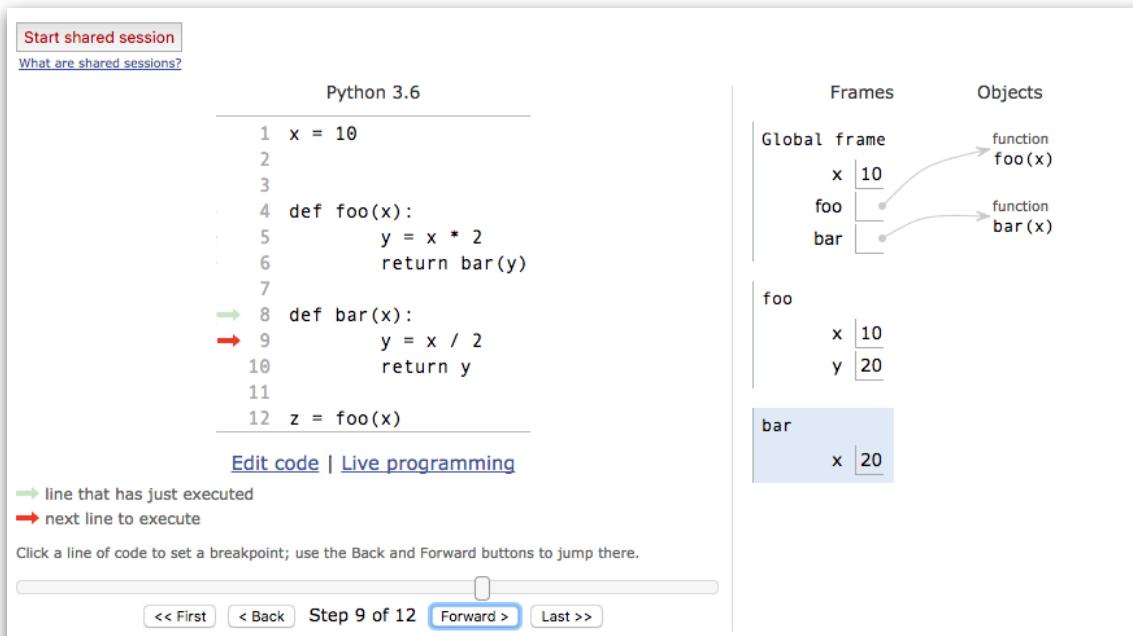
LECTURE 3 FRAMES, FUNCTION CALLS AND SCOPE

首先纠正上一节课的一个错误，使用 `compile` 函数编译源文件时，应该首先读取文件内容，因为 `compile` 的第一个参数应该是字符串。因此从源码文件编译应该用如下方式：

```
c = compile(open("test.py").read(), "test.py", "exec")
```

可视化 Python 代码执行过程

教授本人制作的网站：<http://www.pythontutor.com/visualize.html> 可用于可视化 Python 代码的执行过程，例如：



这里可视化的是 Python 代码执行过程中的 `frame` 与作用域范围内的值或对象。在一节中描述了 `frame` (`execution frame`) 的含义，稍后将会看到它在源码中的定义，在此之前，先来看一下上面这段包含了多个函数调用的反汇编结果：

```
1 x = 10 | ➔ Python-3.6.1 ./python -m dis test.py
2           0 LOAD_CONST               0 (10)
3           2 STORE_NAME                0 (x)
4 def foo(x):
5     y = x * 2
6     return bar(y)
7
8 def bar(x):
9     y = x / 2
10    return y
11
12 z = foo(i)
~           4 LOAD_CONST               1 (<code object foo at 0x7f83dcd36150, file "test.py", line 4>)
~           6 LOAD_CONST               2 ('foo')
~           8 MAKE_FUNCTION
~           10 STORE_NAME               1 (foo)
~           12 LOAD_CONST               3 (<code object bar at 0x7f83dcd3c660, file "test.py", line 8>)
~           14 LOAD_CONST               4 ('bar')
~           16 MAKE_FUNCTION
~           18 STORE_NAME               2 (bar)
~           20 LOAD_NAME                1 (foo)
~           22 LOAD_NAME                3 (i)
~           24 CALL_FUNCTION            1
~           26 STORE_NAME               4 (z)
~           28 LOAD_CONST               5 (None)
~           30 RETURN_VALUE

➔ Python-3.6.1
```

首先，在第4行和第8行定义了两个函数 `foo`、`bar`，在编译过程中并不会执行函数，因此在此处只是将函数体放入 `code object` 中，并与变量名 `foo`、`bar` 相关联（从上面的可视化图中也可以看到）；到了第12行，执行 `z = foo(x)` 时，对应的指令为：

`CALL_FUNCTION`，这里并没有展示调用 `foo` 函数之后发生的指令，只有单独针对 `foo` 函数的 `code object` 才能看到反汇编结果。想要真正理解这段代码在 `Python` 解释器中的执行过程，我们需要回到 `ceval.c` 的主循环函数中，不过在此之前，我们需要先弄清楚以下几个概念：`frame`、`function`、`code object`、`bytecode`。

Frame、Function、Code Object、Bytecode

在上一节中已经提到，`bytecode` 是 `Python` 虚拟机中对程序的表示，即由操作码和参数所组成；`code object` 除了包含 `bytecode` 之外（在 `co_code` 属性中），还包括一些其它信息，定义在 [Include/code.h](#)：

```
20  /* Bytecode object */
21  typedef struct {
22      PyObject_HEAD
23      int co_argcount;           /* #arguments, except *args */
24      int co_kwonlyargcount;    /* #keyword only arguments */
25      int co_nlocals;           /* #local variables */
26      int co_stacksize;         /* #entries needed for evaluation stack */
27      int co_flags;             /* CO_..., see below */
28      int co_firstlineno;       /* first source line number */
29      PyObject *co_code;        /* instruction opcodes */
30      PyObject *co_consts;       /* list (constants used) */
31      PyObject *co_names;        /* list of strings (names used) */
32      PyObject *co_varnames;     /* tuple of strings (local variable names) */
33      PyObject *co_freevars;     /* tuple of strings (free variable names) */
34      PyObject *co_cellvars;     /* tuple of strings (cell variable names) */
35      /* The rest aren't used in either hash or comparisons, except for co_name,
36       used in both. This is done to preserve the name and line number
37       for tracebacks and debuggers; otherwise, constant de-duplication
38       would collapse identical functions/lambdas defined on different lines.
39 */
40      unsigned char *co_cell2arg; /* Maps cell vars which are arguments. */
41      PyObject *co_filename;     /* unicode (where it was loaded from) */
42      PyObject *co_name;          /* unicode (name, for reference) */
43      PyObject *co_lnotab;        /* string (encoding addr<->lineno mapping) See
44                               Objects/lnotab_notes.txt for details. */
45      void *co_zombieframe;      /* for optimization only (see frameobject.c) */
46      PyObject *co_weakreflist;   /* to support weakrefs to code objects */
47      /* Scratch space for extra data relating to the code object.
48       Type is a void* to keep the format private in codeobject.c to force
49       people to go through the proper APIs. */
50      void *co_extra;
51  } PyCodeObject;
```

从上面的定义可以发现，除了 `co_code` 之外，`code object` 还包含了很多其它信息，包括变量名、文件名（可用于 `debug`）等。`frame` 定义在 [Include/frameobject.h](#)（下图，省略了中间部分），可以看出 `PyFrameObject` 是通过链表（lined list）实现的：

```
struct _frame *f_back
```

```

17  typedef struct _frame {
18      PyObject_VAR_HEAD
19      struct _frame *f_back;      /* previous frame, or NULL */
20      PyObject *f_code;          /* code segment */
21      PyObject *f_builtins;       /* builtin symbol table (PyDictObject) */
22      PyObject *f_globals;        /* global symbol table (PyDictObject) */
23      PyObject *f_locals;         /* local symbol table (any mapping) */
24      PyObject **f_valuestack;    /* points after the last local */
25      /* Next free slot in f_valuestack. Frame creation sets to f_valuestack.
26      Frame evaluation usually NULLS it, but a frame that yields sets it
27      to the current stack top. */
28      PyObject **f_stacktop;
29      PyObject *f_trace;          /* Trace function */

30      ...
31      int f_lineno;              /* Current line number */
32      int f_iblock;               /* index in f_blockstack */
33      char f_executing;          /* whether the frame is still executing */
34      PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
35      PyObject *f_localsplus[1];   /* locals+stack, dynamically sized */
36  } PyFrameObject;

```

它除了保存 **code object** (***f_code**, 见上一节中简写主函数的第6行)，还有当前的环境信息，包括 **f_globals**、**f_builtins**、**f_locals**，以及存储操作码参数的 **f_valuestack**，等等。

frame 中包含了 **code object** 以及相关环境变量，这与 **function** 很像。只不过 **function** 是我们在 **Python** 程序中定义的对象，而是 **frame** 是运行时 (**runtime**) 产生的对当前执行代码的表征，可以通过下面的可视化图来说明两者的区别：



这里我们定义了一个递归函数 **fact**，在执行过程中递归地调用它自己，每一次递归调用时都会产生一个新的 **frame**，这一 **frame** 包含了当前的一些环境变量和 **code object**（与 **fact** 函数的 **code object** 相同），而我们定义的 **function** 永远只有一个。

CALL_FUNCTION

在本节最开始的反汇编结果中，我们只看到最后出现了一次 **CALL_FUNCTION** 指令，为了探究这一函数执行的过程，我们需要去主函数的 **switch-case** 中寻找对应的操作：

```
3280     TARGET(CALL_FUNCTION) {
3281         PyObject **sp, *res;
3282         PCALL(PCALL_ALL);
3283         sp = stack_pointer;
3284         res = call_function(&sp, oparg, NULL);
3285         stack_pointer = sp;
3286         PUSH(res);
3287         if (res == NULL) {
3288             goto error;
3289         }
3290         DISPATCH();
3291     }
```

而真正的 `call_function` 定义在第4779行，并且在经过了一系列的条件检查之后，调用第4889行的 `fast_function`，最终，返回调用 `_PyEval_EvalCodeWithName`（第3863行）。在 `_PyEval_EvalCodeWithName` 函数的开始，首先创建了新的 `frame`：

```
3888     /* Create the frame */
3889     tstate = PyThreadState_GET();
3890     assert(tstate != NULL);
3891     f = PyFrame_New(tstate, co, globals, locals);
3892     if (f == NULL) {
3893         return NULL;
3894     }
3895     fastlocals = f->f_localsplus;
3896     freevars = f->f_localsplus + co->co_nlocals;
```

这就能说明上面递归函数的可视化结果的运行过程了，同时也清楚地展示了 `frame` 和 `function` 的关系。这一函数一直到4142行结束，中间完成了 Python 函数调用过程中参数的解析、堆栈操作、`generator/coroutine` 等处理过程。

小结

这一节在主函数的基础上进一步深入介绍了运行时的 `frame` 以及函数调用过程的操作，厘清了 `frame`、`function`、`code object` 以及 `bytecode` 之间的关系。

LECTURE 4 PYOBJECT THE CORE PYTHON OBJECT

在 Python 中“一切皆为对象”，这句话的意思是：对象是 Python 对数据的抽象，Python 程序中所有的数据都是通过对对象或对象之间的关系来表示的（引用自 [Python Data Model](#)）。本节主要讨论 Python 中对象的实现，主要包括的文件有：Include/object.h、Objects/object.c。

Python Object protocol

Python 中的一切对象都拥有如下几个属性：

- ID，通过内置方法 `id()` 可以查看一个对象的 ID（通常是内存地址），一旦创建，对象的 ID 就不会改变；
- 类型，通过内置方法 `type()` 返回一个对象的类型（返回结果也是一个对象），一旦创建，对象的类型就不会改变；
- 值，有些对象的值是可变的（**mutable**），如列表；有些对象的值是不可变的（**immutable**），如元组。

```
→ Python-3.6.1 ./python
Python 3.6.1 (default, May 10 2017, 10:49:45)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> s = 'hello'
>>> id(s)
140293964064168
>>> t = type(s)
>>> t
<class 'str'>
>>> help(t)

>>> print(t.__doc__)
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

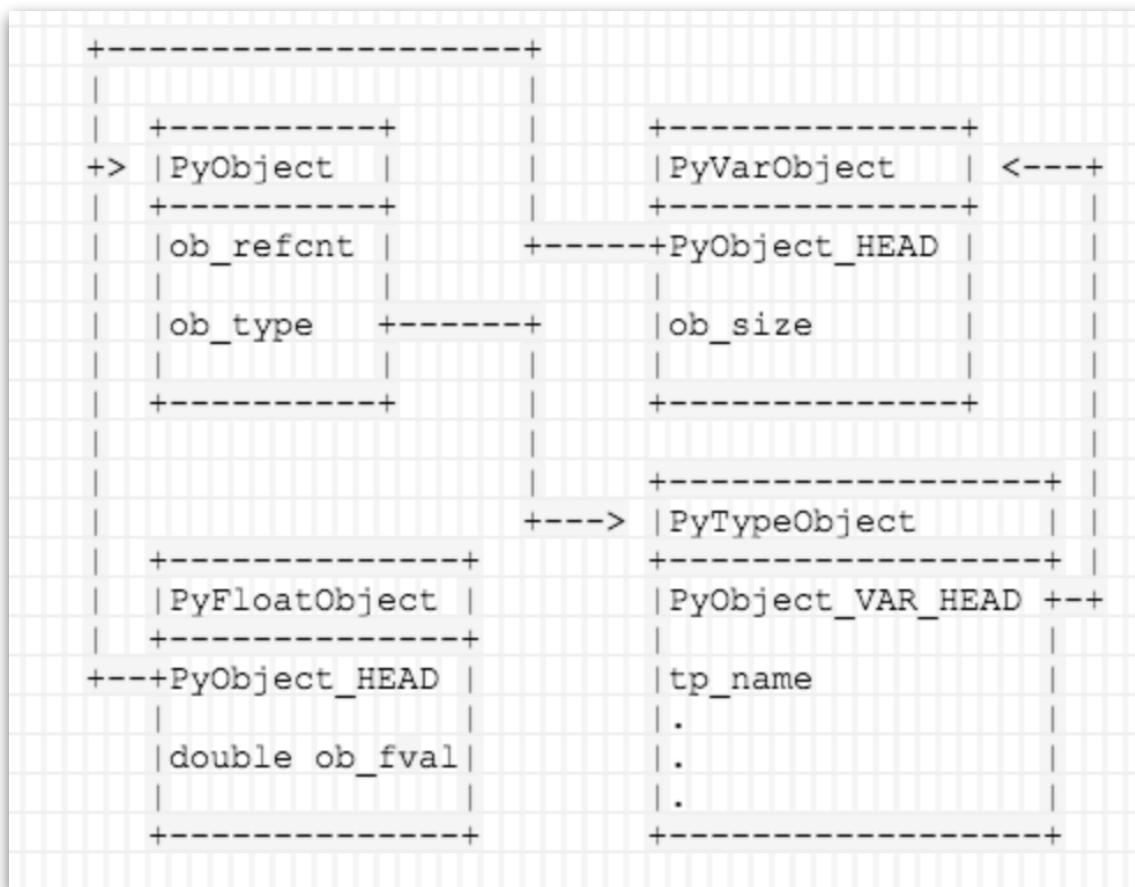
Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
>>> t(123)
'123'
>>> s = 'world'
>>> id(s)
140293931392168
>>> id(t)
8984096
>>> █
```

稍后我们将看到 Python 是如何实现这样的对象机制的。除了以上属性，Python 对象创建之后通常不需要手动删除，而是通过垃圾回收（gc）机制自动清除。CPython 通过引用计数实现垃圾回收。

PyObject

PyObject 是一切对象的基础，定义在 `Include/code.h`，其中除了包含一个引用计数（用于 gc）和指向类型信息的 `struct _typeobject *ob_type` 指针之外，没有任何内容。但是所有 Python 对象的指针都可以强制转换为 PyObject*，因为所有对象（包括 PyTypeObject）结构定义的第一个成员都通过宏 #define PyObject_HEAD PyObject ob_base 指向 PyObject，这种用法称为 structural subtype。

以浮点数对象的定义为例，在 PyFloatObject 结构中，第一个成员就是 PyObject ob_base 的，另一个成员是 double 类型，存储浮点数的值。关于 Python 内置基本类型的定义将在后面的课程进行学习，这里需要提到的是 PyTypeObject 也通过这种方式“继承”了 PyObject，只不过是通过中间人 PyVarObject，而 PyVarObject 只是对 PyObject 扩展了一个 ob_size 字段，用于那些有长度（len）的对象。



Include/object.h 定义了 PyObject 的结构，而 Object/object.c 则实现了 Python 对象的通用方法。例如，所有对象都可以通过 str() 返回该对象的字符串结果。在 Python 层面上，如果某对象定义（重载）了 __str__ 特殊方法，str() 会采用该特殊方法的结果。而对于默认情况下，Object/object.c 定义该函数的原始实现过程（第 502 行），去掉一系列检查与 DEBUG 选项之后的简化版本如下：

```
1 PyObject *
2 PyObject_Str(PyObject *v)
3 {
4     PyObject *res;
5     res = (*Py_TYPE(v)->tp_str)(v);
6     return res;
7 }
```

这里实现了根据不同对象类型动态调用 tp_str 函数，其中

```
#define Py_TYPE(ob) (((PyObject*)(ob))->ob_type)
```

定义了 Py_TYPE 为访问 PyObject 的 ob_type 成员的宏，不同类型的对象会有不同的方法来实现 tp_str，我们还是以浮点数对象为例，看看其实现过程（Object/floatobject.c）：

```
317 static PyObject *
318 float_repr(PyFloatObject *v)
319 {
320     PyObject *result;
321     char *buf;
322
323     buf = PyOS_double_to_string(PyFloat_AS_DOUBLE(v),
324                                 'r', 0,
325                                 Py_DTSF_ADD_DOT_0,
326                                 NULL);
327     if (!buf)
328         return PyErr_NoMemory();
329     result = _PyUnicode_FromASCII(buf, strlen(buf));
330     PyMem_Free(buf);
331     return result;
332 }
```

具体实现过程不在这里深入探究了，只是为了说明通过这种方法实现了不同类型对象通用方法的动态加载。

小结

本节课主要概述了 Python 对象的实现基础: `PyObject`, 以及非常重要的类型对象 `PyTypeObject`。为后面的课程中挑选几个基本类型进行深入探讨做准备。

参考链接

- [Python/C API Reference Manual » Object Implementation Support](#)

LECTURE 5 EXAMPLE PYTHON DATA TYPES

本节课在 `PyObject` 的基础上，以 `str` 类型为例，进一步讨论 `Python` 数据类型的内部实现。不过由于 `str` 是 `Python 2.x` 到 `3.x` 最大的变化之一，甚至没有 `PyStringObject`，取而代之的是 `PyUnicodeObject`。好在我们并不是想要从头重写一遍 `C``Python`，只是了解其大概的实现过程，因而和第一节课提到的一样，总体的架构是不变的。

Python sequence types

在上一节课的 `Python Object protocol` 中提到，`Python` 中的对象按照是否可变可以分为可变与不可变对象。例如，列表是可变的，而字符串和元组是不可变的：

```
>>> x = 'hello'
>>> x[0]
'h'
>>> x[0] = 'H'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> y = list(x)
>>> y
['h', 'e', 'l', 'l', 'o']
>>> y[0] = 'H'
>>> y
['H', 'e', 'l', 'l', 'o']
>>> z = (x, y)
>>> z
('hello', ['H', 'e', 'l', 'l', 'o'])
>>> z[0] = None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> z[1][0] = 'h'
>>> z
('hello', ['h', 'e', 'l', 'l', 'o'])
>>> y
['h', 'e', 'l', 'l', 'o']
>>> |
```

以上面的例子作为说明，`Python` 中的变量存储的都是指向对象（扩展自 `PyObject`）的指针，因此对于不可变对象来说，只是存储的指针地址不变，而指针所指向的对象内容是否可变则与其无关。

序列类型的共同特征是无论是否可变，都可以通过 `for...in...` 语法进行遍历。

PyUnicodeObject

Python 2 到 3 最大的升级之一就是默认使用 **Unicode** 作为字符串，这一点对于用户来说绝对是友好的，我们不需要再去考虑字符串中出现的各种奇怪的 \xe9 和 \u96e8，只需要根据所见即所得的原理去看待字符串即可：

不过这就让 **CPython** 的实现过程变得稍微曲折一些，**str** 类的结构定义在 [Include/unicodeobject.h](#)：

```
337 /* Non-ASCII strings allocated through PyUnicode_New use the
338     PyCompactUnicodeObject structure. state.compact is set, and the data
339     immediately follow the structure. */
340 typedef struct {
341     PyASCIIObject _base;
342     Py_ssize_t utf8_length;      /* Number of bytes in utf8, excluding the
343                                * terminating \0. */
344     char *utf8;                /* UTF-8 representation (null-terminated) */
345     Py_ssize_t wstr_length;    /* Number of code points in wstr, possible
346                                * surrogates count as two code points. */
347 } PyCompactUnicodeObject;
348
349 /* Strings allocated through PyUnicode_FromUnicode(NULL, len) use the
350     PyUnicodeObject structure. The actual string data is initially in the wstr
351     block, and copied into the data block using _PyUnicode_Ready. */
352 typedef struct {
353     PyCompactUnicodeObject _base;
354     union {
355         void *any;
356         Py_UCS1 *latin1;
357         Py_UCS2 *ucs2;
358         Py_UCS4 *ucs4;
359     } data;                  /* Canonical, smallest-form Unicode buffer */
360 } PyUnicodeObject;
361 #endif
```

在这里就不深入探究如何实现、存储 **UTF-8** 编码了。接下来以字符串操作的反汇编结果，查看其内部实现机制：

```
→ Python-3.6.1 ./python -m dis test.py
  0 LOAD_CONST      0 ('hello')
  2 STORE_NAME       0 (x)
  4 LOAD_CONST      1 ('world')
  6 STORE_NAME       1 (y)
  8 LOAD_NAME        0 (x)
 10 LOAD_NAME        1 (y)
 12 COMPARE_OP      2 (==)
 14 POP_TOP
 16 LOAD_NAME        0 (x)
 18 LOAD_NAME        1 (y)
 20 COMPARE_OP      8 (is)
 22 POP_TOP
 24 LOAD_CONST      2 (None)
 26 RETURN_VALUE

→ Python-3.6.1 |
```

再次回到解释器主循环程序 [Python/ceval.c](#)，我们发现 `==` 和 `is` 两个比较语法编译得到的操作指令是相同的，都是 `COMPARE_OP`，但是参数不同，前者是 `2` 后者为 `8`。我们找到主循环函数中的 `COMPARE_OP` 分支：

```
2793     TARGET(COMPARE_OP) {
2794         PyObject *right = POP();
2795         PyObject *left = TOP();
2796         PyObject *res = cmp_outcome(oparg, left, right);
2797         Py_DECREF(left);
2798         Py_DECREF(right);
2799         SET_TOP(res);
2800         if (res == NULL)
2801             goto error;
2802         PREDICT(POP_JUMP_IF_FALSE);
2803         PREDICT(POP_JUMP_IF_TRUE);
2804         DISPATCH();
2805     }
```

```
4950 static PyObject *
4951 cmp_outcome(int op, PyObject *v, PyObject *w)
4952 {
4953     int res = 0;
4954     switch (op) {
4955     case PyCmp_IS:
4956         res = (v == w);
4957         break;
```

可以预料，`PyCmp_IS` 一定是定义在 `Include/opcode.h` 的操作码参数：

```
139 enum cmp_op {PyCmp_LT=Py_LT, PyCmp_LE=Py_LE, PyCmp_EQ=Py_EQ, PyCmp_NE=Py_NE,
140                 PyCmp_GT=Py_GT, PyCmp_GE=Py_GE, PyCmp_IN, PyCmp_NOT_IN,
141                 PyCmp_IS, PyCmp_IS_NOT, PyCmp_EXC_MATCH, PyCmp_BAD};
```

也就是说，`is` 比较的是两个 `PyObject` 指针参数 `*v` 和 `*w`；在 Python 层面，相当于 `id(x) == id(y)`。对于 `PyCmp_EQ`（`2`）则稍微复杂一些，在上面的 `cmp_outcome` 函数中，默认情况将会调用 `return PyObject_RichCompare(v, w, op)`，和上一节课中提到的浮点数的例子相似，这一函数定义在 `Objects/object.c`（[第706行](#)），最终调用 `res = do_richcompare(v, w, op)`。可以预料，`do_richcompare` 是根据对象类型不同动态加载的，于是我们找到 `Objects/unicodeobject.c`（[第11150行](#)），并最终找到真正实现字符串比较的函数（[第10842行](#)）：`static int unicode_compare(PyObject *str1, PyObject *str2)`。

小结

本节课首先简单介绍了 Python 中的序列数据类型（List, Tuple, String），从 Python 层面理解它们的引用机制（特别是可变与不可变之分）；然后以 str 类型为例，查看其在运行过程中的实现机制。

LECTURE 6 CODE OBJECTS, FUNCTION OBJECTS, AND CLOSURES

本节课继续探讨 Python 中的函数对象。在前面的课程中我们已经讨论过函数的调用该过程，这次主要探讨函数对象的实现以及一种特殊的函数应用：Closure。

Function Object

函数对象的结构定义在 `Include/funcobject.h`，源码中的注释信息可以帮助我们更好地理解其实现机制：

```
21  typedef struct {
22      PyObject_HEAD
23      PyObject *func_code;           /* A code object, the __code__ attribute */
24      PyObject *func_globals;        /* A dictionary (other mappings won't do) */
25      PyObject *func_defaults;       /* NULL or a tuple */
26      PyObject *func_kwdefaults;     /* NULL or a dict */
27      PyObject *func_closure;        /* NULL or a tuple of cell objects */
28      PyObject *func_doc;           /* The __doc__ attribute, can be anything */
29      PyObject *func_name;          /* The __name__ attribute, a string object */
30      PyObject *func_dict;          /* The __dict__ attribute, a dict or NULL */
31      PyObject *func_weakreflist;    /* List of weak references */
32      PyObject *func_module;         /* The __module__ attribute, can be anything */
33      PyObject *func_annotations;    /* Annotations, a dict or NULL */
34      PyObject *func_qualname;       /* The qualified name */
35
36      /* Invariant:
37      *     func_closure contains the bindings for func_code->co_freevars, so
38      *     PyTuple_Size(func_closure) == PyCode_GetNumFree(func_code)
39      *     (func_closure may be NULL if PyCode_GetNumFree(func_code) == 0).
40      */
41 } PyFunctionObject;
```

在 Python 3.x 中，函数的一些属性由 `func_x` 改名为 `__x__`，例如在 [Lecture 3](#) 提到的 `code object`，即存储在 `__code__`，也就是上图结构中的 `PyObject *func_code`。函数中除了包含代码对象之外，还存储了函数所在的环境中的全局变量，保存在 `__globals__`：

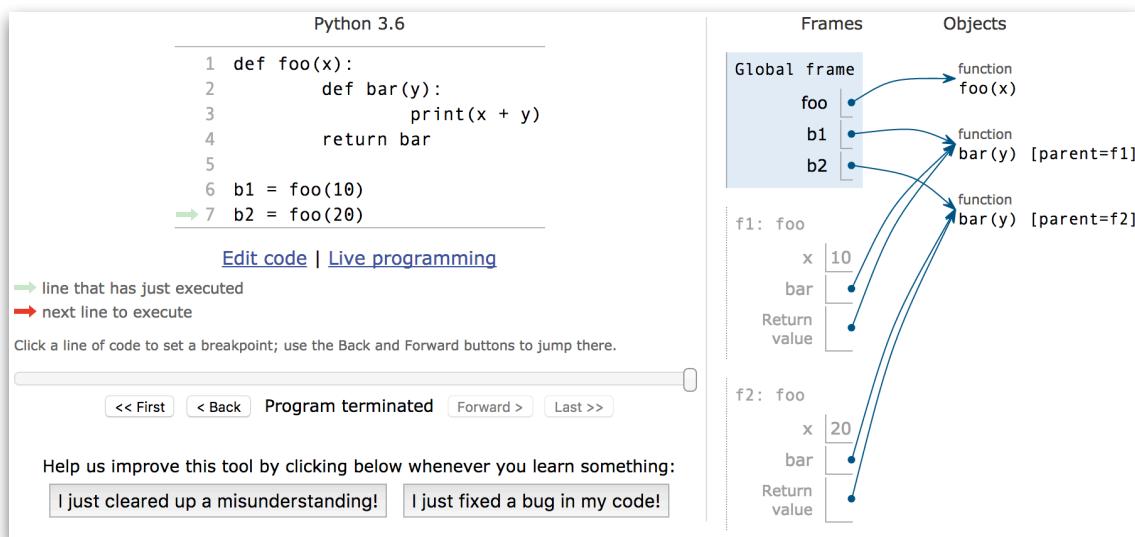
```
→ Python-3.6.1 ./python
Python 3.6.1 (default, May 10 2017, 10:49:45)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def foo(x, y):
...     return x + y
...
>>> foo.__globals__
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'foo': <function foo at 0x7f7046326e18>}
>>> PI = 3.14
>>> foo.__globals__
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'foo': <function foo at 0x7f7046326e18>, 'PI': 3.14}
>>> █
```

因为函数执行过程中除了函数体中的局部变量，还可能会用到全局变量。这就涉及到 Python 中作用域的问题，而在 Python 的函数中，除了函数外的全局变量和函数内的局部变量，还可能因为函数的嵌套而产生其它的作用域范围，这就引出闭包的概念，接下来就来探讨一下闭包在 Python 中的实现。

Closure

关于 Python 层面的作用域与闭包，可以查看参考链接中的文章。

首先来看一个闭包可视化的例子：



可以看出，每一次调用 `foo` 都会创建一个新的 `bar`，而 `bar` 中存储了来自 `foo` 的参数 `x`。

PyFunctionObject 结构中的：

```
PyObject *func_closure; /* NULL or a tuple of cell objects */
```

存储了闭包中所必须的环境变量：

```
→ Python-3.6.1 ./python -i test.py
>>> b1.__closure__
(<cell at 0x7fe05026b858: int object at 0x8e21a0>,)
>>> b1.__closure__[0].cell_contents
10
>>> b1.__code__.co_freevars
('x',)
>>> b1.__code__.co_varnames
('y',)
>>> 
```

`b1` 从“父函数”中“继承”的变量（上例中是 `x`）存储在 `b1.__closure__` 中；而对应的变量名则存储在 `b1` 的 `code object` 中，其中 `x` 是“自由变量”，而 `y` 则是绑定到 `b1` 所接收的参数上。

参考链接

- [PyTips 0x04-作用域与闭包](#)

LECTURE 7 ITERATORS

本节课主要关于 Python 中的 **Iterator** 对象以及 **for** 循环语句的实现机制。

Iterator

Iterator 是 Python 中的一种对象，用于表示一串数据，通过重复调用 `iterator.__next__()`（Python 2.x 中该方法名为 `next()`）方法（或内置的 `next(iterator)` 方法）可以逐个读取每个数据，直到读取完所有数据时，**Iterator** 对象会抛出 **StopIteration** 异常。

一些内置的容器类型（如 `list`、`tuple` 等）可以通过 `iter()` 方法创建一个新的 **Iterator**；`iter()` 方法调用容器类型的 `__iter__()` 方法来获取 **Iterator**，因此自定义的类也可以通过实现 `__iter__()` 和 `__next__()` 方法使其满足 **Iterator Protocol**。

GET_ITER & FOR_ITER

Python 中最常用的 **for** 循环语句，就是通过创建一个 **Iterator** 并重复调用 **next** 方法，直到捕捉到 **StopIteration** 异常来结束循环。反汇编下面这个简单的 **for** 循环语句：

```
1 x = ['a', 'b', 'c']
2 for e in x:
3     print(e)

→ Python-3.6.1 ./python -m dis iter_test.py
 1          0 LOAD_CONST              0 ('a')
 2          2 LOAD_CONST              1 ('b')
 4          4 LOAD_CONST              2 ('c')
 6          6 BUILD_LIST               3
 8          8 STORE_NAME              0 (x)

 2         10 SETUP_LOOP             20 (to 32)
 12        12 LOAD_NAME               0 (x)
 14        14 GET_ITER
>>    16 FOR_ITER                12 (to 30)
 18        18 STORE_NAME              1 (e)

 3         20 LOAD_NAME               2 (print)
 22        22 LOAD_NAME               1 (e)
 24        24 CALL_FUNCTION            1
 26        26 POP_TOP
 28        28 JUMP_ABSOLUTE           16
>>    30 POP_BLOCK
>>    32 LOAD_CONST               3 (None)
 34        34 RETURN_VALUE
```

从第10行 `SETUP_LOOP` 开始进入 `for` 语句，`GET_ITER` 相当于 Python 中的 `iter(x)` 方法，获取 `Iterator` 对象；然后会重复执行16~28行，直到循环结束，跳到第30行（第16行的操作码参数 `to 30`）。

通过第14行的操作码 `GET_ITER`，我们可以从 `ceval.c` 中找到该操作码（3014行）：

```
3014     TARGET(GET_ITER) {
3015         /* before: [obj]; after [getiter(obj)] */
3016         PyObject *iterable = TOP();
3017         PyObject *iter = PyObject_GetIter(iterable);
3018         Py_DECREF(iterable);
3019         SET_TOP(iter);
3020         if (iter == NULL)
3021             goto error;
3022         PREDICT(FOR_ITER);
3023         PREDICT(CALL_FUNCTION);
3024         DISPATCH();
3025     }
```

通过调用 `PyObject_GetIter` 函数获取 `Iterator`，该函数定义在 [Objects/abstract.c#L3110](#)：

```
3110     PyObject *
3111     PyObject_GetIter(PyObject *o)
3112     {
3113         PyTypeObject *t = o->ob_type;
3114         getiterfunc f = NULL;
3115         f = t->tp_iter;
3116         if (f == NULL) {
3117             if (PySequence_Check(o))
3118                 return PySeqIter_New(o);
3119             return type_error("'%.*.200s' object is not iterable", o);
3120         }
3121         else {
3122             PyObject *res = (*f)(o);
3123             if (res != NULL && !PyIter_Check(res)) {
3124                 PyErr_Format(PyExc_TypeError,
3125                             "iter() returned non-iterator "
3126                             "of type '%.100s'",
3127                             res->ob_type->tp_name);
3128                 Py_DECREF(res);
3129                 res = NULL;
3130             }
3131             return res;
3132         }
3133     }
```

首先通过 `o->ob_type` 获取传入对象的类型对象 (`PyTypeObject`)，然后查询该对象是否定义了 `tp_iter` 方法，在 Python2.x 中的查询方法稍有不同，是通过一个 `FLAG` 参数来查询的，但原理是一致的。在 Python 3 的实现中，`list` 对象已经有了 `tp_iter` 方法，可以在 `Objects/listobject.c` 中找到定义：

```

2731 static PyObject *
2732 list_iter(PyObject *seq)
2733 {
2734     listiterobject *it;
2735
2736     if (!PyList_Check(seq)) {
2737         PyErr_BadInternalCall();
2738         return NULL;
2739     }
2740     it = PyObject_GC_New(listiterobject, &PyListIter_Type);
2741     if (it == NULL)
2742         return NULL;
2743     it->it_index = 0;
2744     Py_INCREF(seq);
2745     it->it_seq = (PyListObject *)seq;
2746     _PyObject_GC_TRACK(it);
2747     return (PyObject *)it;
2748 }

```

首先声明了一个 `listiterobject` 类型的指针，然后通过 `PyObject_GC_New` 赋予其 `PyListIter_Type` 这一 `PyTypeObject`。`listiterobject` 和 `PyListIter_Type` 都定义在 `listobject.c` 中：

```

2671 typedef struct {
2672     PyObject_HEAD
2673     Py_ssize_t it_index;
2674     PyListObject *it_seq; /* Set to NULL when iterator is exhausted */
2675 } listiterobject;

```

```

2697 PyTypeObject PyListIter_Type = {
2698     PyVarObject_HEAD_INIT(&PyType_Type, 0)           <list_iterator at 0x1039ca438>
2699     "list_iterator",                                /* tp_name */
2700     sizeof(listiterobject),                         /* tp_basicsize */
2701     0,                                              /* tp_itemsize */
2702     /* methods */
2703     (destructor)listiter_dealloc,                  /* tp_dealloc */
2704     0,                                              /* tp_print */

```



在 Python 层面，列表对象对应的 Iterator 确实类型名称为 `list_iterator`：

```

→ Python-3.6.1 ./python
Python 3.6.1 (default, May 10 2017, 10:49:45)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = [1,2,3]
>>> x.__iter__()
<list_iterator object at 0x7f0e5b39c6d8>
>>> 

```

根据 `listiterator` 结构的定义，它存储了一个当前位置的 `it_index` 属性和一个指向列表对象的指针，由此可以在每次调用 `next()` 方法时递增 `it_index` 来获取下一个数据。我们可以将 `for` 循环理解为 `it_index++` 的操作。

回到上面的反汇编的结果，`GET_ITER` 指令获取了 `Iterator` 之后，接下来进入 `FOR_ITER`，也就是真正的循环过程，从 `ceval.c` 中找到找一操作码：

```
3032 TARGET(FOR_ITER) {
3033     /* before: [iter]; after: [iter, iter()] *or* [] */
3034     PyObject *iter = TOP();
3035     PyObject *next = (*iter->ob_type->tp_iternext)(iter);
3036     if (next != NULL) {
3037         PUSH(next);
3038         PREDICT(STORE_FAST);
3039         PREDICT(UNPACK_SEQUENCE);
3040         DISPATCH();
3041     }
3042     if (PyErr_Occurred()) {
3043         if (!PyErr_ExceptionMatches(PyExc_StopIteration))
3044             goto error;
3045         else if (tstate->c_tracefunc != NULL)
3046             call_exc_trace(tstate->c_tracefunc, tstate->c_traceobj, tstate, f);
3047         PyErr_Clear();
3048     }
3049     /* iterator ended normally */
3050     STACKADJ(-1);
3051     Py_DECREF(iter);
3052     JUMPBY(oparg);
3053     PREDICT(POP_BLOCK);
3054     DISPATCH();
3055 }
```

The diagram shows two blue arrows. One arrow points from the 'FOR_ITER 12 (to 30)' label back up to the 'PREDICT(POP_BLOCK);' line in the assembly code. Another arrow points from the 'PREDICT(POP_BLOCK);' line down to the 'DISPATCH();' line.

首先从 `TOP()` 中获取 `GET_ITER` 指令所返回的 `Iterator`，然后调用 `*iter->ob_type->tp_iternext` 方法，该方法仍然可以从 `listobject.c` 文件中找到：

```
2765 static PyObject *
2766 listiter_next(listiterobject *it)
2767 {
2768     PyListObject *seq;
2769     PyObject *item;
2770
2771     assert(it != NULL);
2772     seq = it->it_seq;
2773     if (seq == NULL)
2774         return NULL;
2775     assert(PyList_Check(seq));
2776
2777     if (it->it_index < PyList_GET_SIZE(seq)) {
2778         item = PyList_GET_ITEM(seq, it->it_index);
2779         ++it->it_index;
2780         Py_INCREF(item);
2781         return item;
2782     }
2783
2784     it->it_seq = NULL;
2785     Py_DECREF(seq);
2786     return NULL;
2787 }
```

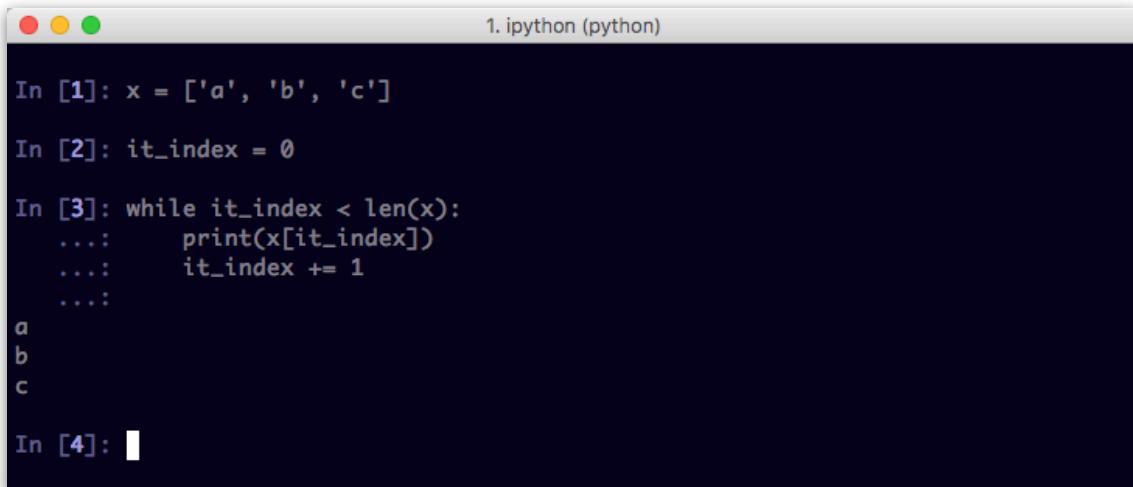
正如上文中所提到的，`listiter_next` 方法就是通过 `++it->it_index` 操作来增加Iterator当前位置，并通过 `PyList_GET_ITEM(seq, it->it_index)` 获取当前位置上数据，`PyList_GET_ITEM` 宏自然定义在 `Include/listobject.h` 中：

```
72 #define PyList_GET_ITEM(op, i) (((PyListObject *)(op))->ob_item[i])
```

以上就是 `for` 循环语句对 `list` 对象的遍历过程。除了 `list`，其它可以转化为 `Iterator` 对象的类型可能有各自不同的实现方式（包括自定义类），但原理上是相同的。

小结

本节开始用到的 `Python` 文件实际上也可以写作：



```
In [1]: x = ['a', 'b', 'c']
In [2]: it_index = 0
In [3]: while it_index < len(x):
...:     print(x[it_index])
...:     it_index += 1
...:
a
b
c
In [4]:
```

这样似乎更符合一般语言的习惯，不过 `Python` 通过 `Iterator Protocol`，将一般的循环流程简化为 `for ... in ...`，将面向机器的思维转化为面向人类语言的思维，这体现了 `Python` 简洁易懂的语法背后的设计思路。

LECTURE 8 USER-DEFINED CLASSES AND OBJECTS

有了前面6节课的基础，第7、8、9节课则本别以特定的 Python 对象、语法为例，探讨 CPython 的实现机制。上一节课讨论的是 **Iterator**，本节课讨论 Python 中用户定义的类和对象。

New style classes VS Old style classes

本节课所用的例子是在上一节课基础上自定义的 **Iterator** 类：

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high
    def __iter__(self):
        return self
    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

c = Counter(5, 7)
```

其中有一点区别，在上一节课中也提到了，Python 3 中迭代器协议采用 **__next__** 方法，而不是 **next**。此外需要说明的是，在 Python 2 中，使用 **class** 定义类时存在新、旧两种风格，取决于是否继承 **object** 对象，上面的例子如果可以运行在 Python 2 中则是一种旧风格的类。既然有风格新旧之分，那自然是采用新的风格更好（如果想要具体了解两种定义的区别，可以查看后面的参考链接），不过在 Python 3 中，已经不存在所谓的旧风格了，因此上面的例子在 Python 3 中是没有歧义的。

LOAD_BUILD_CLASS

除了新旧风格的差别，Python 3 与 2 在类的实现上也存在较大差异，接下来的内容将以 Python 3.6.1 为准。

接下来还是执行反汇编指令：

```

→ Python-3.6.1 ./python -m dis count.py
  0 LOAD_BUILD_CLASS
  1           0 (code object Counter at 0x7
f3e49a00780, file "count.py", line 1>)
  2 LOAD_CONST          0 (<code object Counter at 0x7
f3e49a00780, file "count.py", line 1>)
  3 MAKE_FUNCTION       1 ('Counter')
  4 LOAD_CONST          0
  5 LOAD_CONST          1 ('Counter')
  6 LOAD_CONST          2
  7 CALL_FUNCTION        0 (Counter)
  8 STORE_NAME           0 (Counter)

 14 LOAD_NAME             0 (Counter)
 15 LOAD_CONST            2 (5)
 16 LOAD_CONST            3 (7)
 17 CALL_FUNCTION         2
 18 STORE_NAME             1 (c)
 19 LOAD_CONST            4 (None)
 20 RETURN_VALUE

```

我们还是去 `ceval.c` 中找到 `LOAD_BUILD_CLASS` 操作码，在2110行：

```

2110 TARGET(LOAD_BUILD_CLASS) {
2111     _Py_IDENTIFIER(__build_class__);
2112
2113     PyObject *bc;
2114     if (_PyDict_CheckExact(f->f_builtins)) {
2115         bc = _PyDict_GetItemId(f->f_builtins, &PyId__build_class__);

        ● ● ●

2135     PUSH(bc);
2136     DISPATCH();
2137 }

```

`__build_class__` 是 Python 3 中的一个内置方法，用于定义一个类：

```

Help on built-in function __build_class__ in module builtins:

__build_class__(...)

    __build_class__(func, name, *bases, metaclass=None, **kwds) -> class

        Internal helper function used by the class statement.
(END)

```

```

class C(A, B, metaclass=M, other=42, *more_bases, *more_kwds):
    pass

# would translate into this:
C = __build_class__(<func>,
                    'C',
                    A,
                    B,
                    metaclass=M,
                    other=42, *more_bases, *more_kwds)

```

因此 LOAD_BUILD_CLASS 指令是从内置方法中取出 `_build_class` 并压入栈，之后的步骤和一般的函数构造过程一致。

上面的反汇编过程省去了类内的方法定义过程，可以通过 `dis.dis` 查看：

```
→ Python-3.6.1 ./python
Python 3.6.1 (default, May 10 2017, 10:49:45)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information
>>> import count
>>> import dis
>>> dis.dis(count.Counter)
Disassembly of __init__:
  3      0 LOAD_FAST              1 (low)
         2 LOAD_FAST              0 (self)
         4 STORE_ATTR             0 (current)

  4      6 LOAD_FAST              2 (high)
         8 LOAD_FAST              0 (self)
        10 STORE_ATTR             1 (high)
        12 LOAD_CONST             0 (None)
        14 RETURN_VALUE

Disassembly of __iter__:
  6      0 LOAD_FAST              0 (self)
         2 RETURN_VALUE

Disassembly of __next__:
  8      0 LOAD_FAST              0 (self)
         2 LOAD_ATTR               0 (current)
         4 LOAD_FAST              0 (self)
         6 LOAD_ATTR               1 (high)
         8 COMPARE_OP              4 (>)
        10 POP_JUMP_IF_FALSE       18

  9      12 LOAD_GLOBAL             2 (StopIteration)
        14 RAISE_VARARGS            1
        16 JUMP_FORWARD             24 (to 42)
```

参考链接

1. [What does Python's builtin build class do? - Stack Overflow](#)
2. [class - What is the difference between old style and new style classes in Python? - Stack Overflow](#)

LECTURE 9 - GENERATORS

上一节课中自定义了一个符合 **Iterator Protocol** 的 **Counter** 类，本节采用 Python 中一种更加简洁优雅的形式来实现这一功能，即生成器（generator）：

```
"""
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high
    def __iter__(self):
        return self
    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

def Counter(low, high):
    current = low
    while current <= high:
        yield current
        current += 1

c = Counter(5, 7)
for elt in c:
    print(elt)
```

通过定义一个 **generator** 函数，完全不需要修改原有 **Counter** 类的接口，即实现了同样的功能。**generator** 的定义和普通函数一样，唯一的区别就在于 **yield** 关键词。为了查看 **generator** 内部的反汇编结果，和上一节一样，我们需要使用 **dis.dis** 方法。

YIELD_VALUE

```
>>> import dis
>>> dis.dis(count.Counter)
 17      0 LOAD_FAST              0 (low)
          2 STORE_FAST             2 (current)

 18      4 SETUP_LOOP              26 (to 32)
      >>  6 LOAD_FAST              2 (current)
          8 LOAD_FAST              1 (high)
         10 COMPARE_OP             1 (<=)
         12 POP_JUMP_IF_FALSE       30

 19      14 LOAD_FAST              2 (current)
         16 YIELD_VALUE
         18 POP_TOP
```

从 `ceval.c` 中找到 `YIELD_VALUE` 指令，实际上只是将 `stack_pointer` 指针保存在 `f->f_stacktop` 中，以便再次执行 `generator` 时回到 `yield` 语句所在的位置，继续执行：

```
2057     TARGET(YIELD_VALUE) {
2058         retval = POP();
2059
2060         if (co->co_flags & CO_ASYNC_GENERATOR) {
2061             PyObject *w = _PyAsyncGenValueWrapperNew(retval);
2062             Py_DECREF(retval);
2063             if (w == NULL) {
2064                 retval = NULL;
2065                 goto error;
2066             }
2067             retval = w;
2068         }
2069
2070         f->f_stacktop = stack_pointer;
2071         why = WHY_YIELD;
2072         goto fast_yield;
2073     }
```

PyGenObject

不管是 `generator` 还是上一节中的 `__build_class__` 方法，都是不同类型的 `Function`，如果你还记得，在前文中的 [Lecture 3](#) 已经提到过函数的编译过程。在 `_PyEval_EvalCodeWithName` 函数中，包含了处理 `generator` 的部分：

```
/* Handle generator/coroutine/asynchronous generator */
gen = PyGen_NewWithQualName(f, name, qualname);
```

不用说，生成 `PyGenObject` 的 `PyGen_NewWithQualName` 肯定定义在 `Objects/genobject.c`（真正的定义是 `gen_new_with_qualname`）。

`PyGenObject` 的类型对象定义为：

```
745 PyTypeObject PyGen_Type = {
746     PyVarObject_HEAD_INIT(&PyType_Type, 0)
747     "generator",                                     /* tp_name */
748     sizeof(PyGenObject),                            /* tp_basicsize */
749     0,                                              /* tp_itemsize */
750     /* methods */

    ●    ●    ●

774     (iternextfunc)gen_iternext,                      /* tp_iternext */
775     gen_methods,                                    /* tp_methods */
776     gen_memberlist,                                /* tp_members */
777     gen_getsetlist,                               /* tp_getset */
778     0,                                              /* tp_base */
779     0,                                              /* tp_dict */
```

在 Lecture 7 中我们已经看到，遍历 **Iterator** 时所使用的 **next()** 方法，实际上是调用该对象（类型）的 **tp_iternext** 方法，从上图我们可以看出，**generator** 对象的 **tp_iternext** 方法为 **gen_iternext**：

```
560 static PyObject *
561 gen_iternext(PyGenObject *gen)
562 {
563     return gen_send_ex(gen, NULL, 0, 0);
564 }
```

由此可以知道，对于 **generator** 来说，**next()** 方法调用的实际上是 **gen.send(None)**。从 **Python** 层面上来看：

```
>>> c = Counter(1, 6)
>>> next(c)
1
>>> next(c)
2
>>> c.send(None)
3
>>> c.send(None)
4
>>> next(c)
5
```

小结

以上就是本教程的全部内容。本课程并非想要从头实现一次 **Python**，而是借助 **dis** 模块反汇编 **Python** 代码，进而从源码中找出关于 **C****Python** 实现的线索。

在 **Python** 中，一切皆为对象。因此在寻找 **Python** 的实现机制时，最先需要找到要分析的对象，及其类型。**Python** 中的内置对象定义在 **Objects/xxx.c** 和 **Include/xxx.h** 中；**xxx.h** 中定义了该对象的基本属性，通常名为 **PyXxxObject**，而 **xxx.c** 中则定义了该对象的类型、方法等信息，其中类型定义为 **PyXxx_Type**。其中，一些通用的方法抽象至 **Objects/abstract.c** 中，例如 **PyObject_Length** 等。

通过 **dis** 模块获得 **Python** 代码的反汇编操作码及参数，可以在 **Python/ceval.c** 这一解释器主程序所在源文件中找到对应的处理。

C**Python** 代码库已经托管到 [Github](#) 上，配合 **Chrome Octotree** 插件，可以快速浏览不同版本的 **C****Python** 源码。