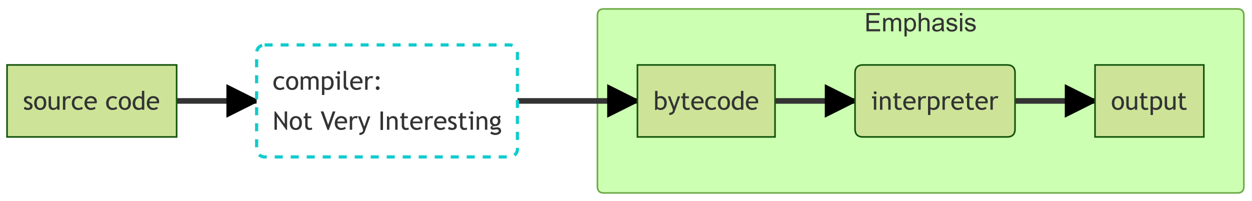
Cpython Internals笔记

Lecture 1 - Interpreter and source code overview

1. Interpreter and source code overview.



Lecture 2 - Opcodes and main interpreter loop

1. 获取bytecode用到的主要命令:

```
# ------------------------------------------
import dis
dis.dis(mod)    # 使用dis模块获取mod的bytecode
# ------------------------------------------
c = compile(open('test.py').read(), 'test.py', 'exec')  # 返回code object
# 返回bytecode的数字编码 [100, 0, 0, 90, 0, 0, 100,....]
[ord(byte) for byte in c.co_code]
dis.dis(c)      # 返回bytecode
dir(c) # 返回c中主要信息
c.co_code # 返回汇编代码
c.co_consts # 返回常量元组
```

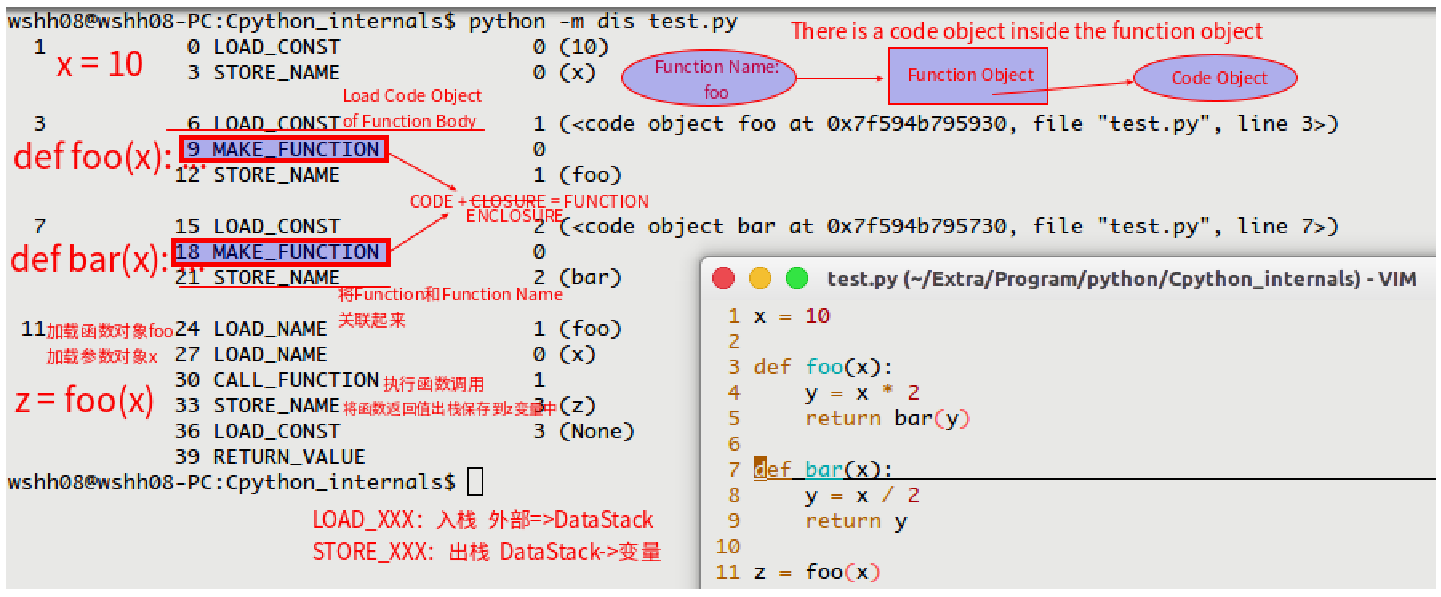
```
python -m dis test.py
```

   *可用byteplay包替代上述命令获得更好的反编译效果
2. 关键代码注释
   1. **Python/ceval.c**
      - ines 693-3021: main interpreter, operate one frame(equivalent one function), returns a python object to whoever calls this function.

```
PyObject *PyEval_EvalFrameEx(PyFrameObject *f, int throwflag){
/* line 689-730: define locale variables to store the locale states
 * line 698: The pointer to the value stack
 */
register PyObject **stack_pointer; /* Next free slot in value stack */
/* line 825-858: stack manipulation macros
 * line 919-945: grab everything out from code(frame)
 * line 964: a giant infinite loop: go through the bytecode one byte a time
 */
for (;;) {
        // line 1080: Extract the next opcode
        opcode = NEXTOP();
        // line 1083: Extract args if have arg
        if (HAS_ARG(opcode))
        oparg = NEXTARG();
        }
        // line 1112: Execute the opcode by switch.
        switch (opcode) {
                /* Py_DECREF/Py_INCREF is used to support the reference count
                   Garbage Collection: like Py_DECREF(v) in POP_TOP.
                */
                CASE LOAD_FAST: ... break;
                CASE LOAD_CONST: ... break;
                ...
                }
        // line 2959-2960: kick out the infinite loop
        if (why != WHY_NOT)
        break;
        // clean up
        }
// line 3020: return the result value
return retval;
}
```

Lecture 3 - Frames, function calls, and scope

1. 函数调用相关指令预览

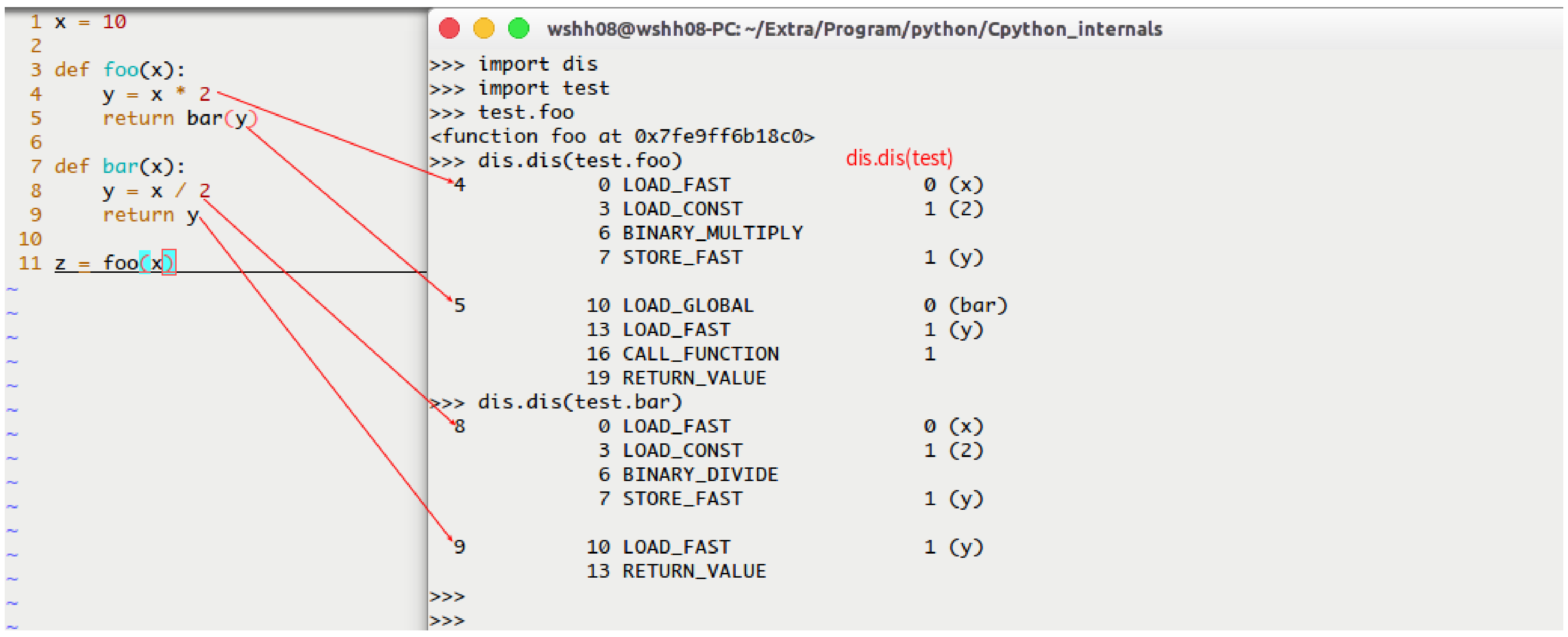


There is a code object inside the function object
Function Name: foo
Function Object
Code Object
x = 10
def foo(x):
def bar(x):
z = foo(x)
Load Code Object of Function Body
CODE + ~~CLOSURE~~ ENCLOSURE = FUNCTION
将Function和Function Name关联起来
加载函数对象foo
加载参数对象x
执行函数调用
将函数返回值出栈保存到z变量中
LOAD_XXX: 入栈 外部=>DataStack
STORE_XXX: 出栈 DataStack->变量

```
wshh08@wshh08-PC:Cpython_internals$ python -m dis test.py
  1           0 LOAD_CONST               0 (10)
              3 STORE_NAME               0 (x)

  3           6 LOAD_CONST               1 (<code object foo at 0x7f594b795930, file "test.py", line 3>)
              9 MAKE_FUNCTION            0
             12 STORE_NAME               1 (foo)

  7          15 LOAD_CONST               2 (<code object bar at 0x7f594b795730, file "test.py", line 7>)
             18 MAKE_FUNCTION            0
             21 STORE_NAME               2 (bar)

 11          24 LOAD_NAME                1 (foo)
             27 LOAD_NAME                0 (x)
             30 CALL_FUNCTION            1
             33 STORE_NAME               3 (z)
             36 LOAD_CONST               3 (None)
             39 RETURN_VALUE
wshh08@wshh08-PC:Cpython_internals$
```

test.py (~/Extra/Program/python/Cpython_internals) - VIM

```
x = 10

def foo(x):
    y = x * 2
    return bar(y)

def bar(x):
    y = x / 2
    return y

z = foo(x)
```

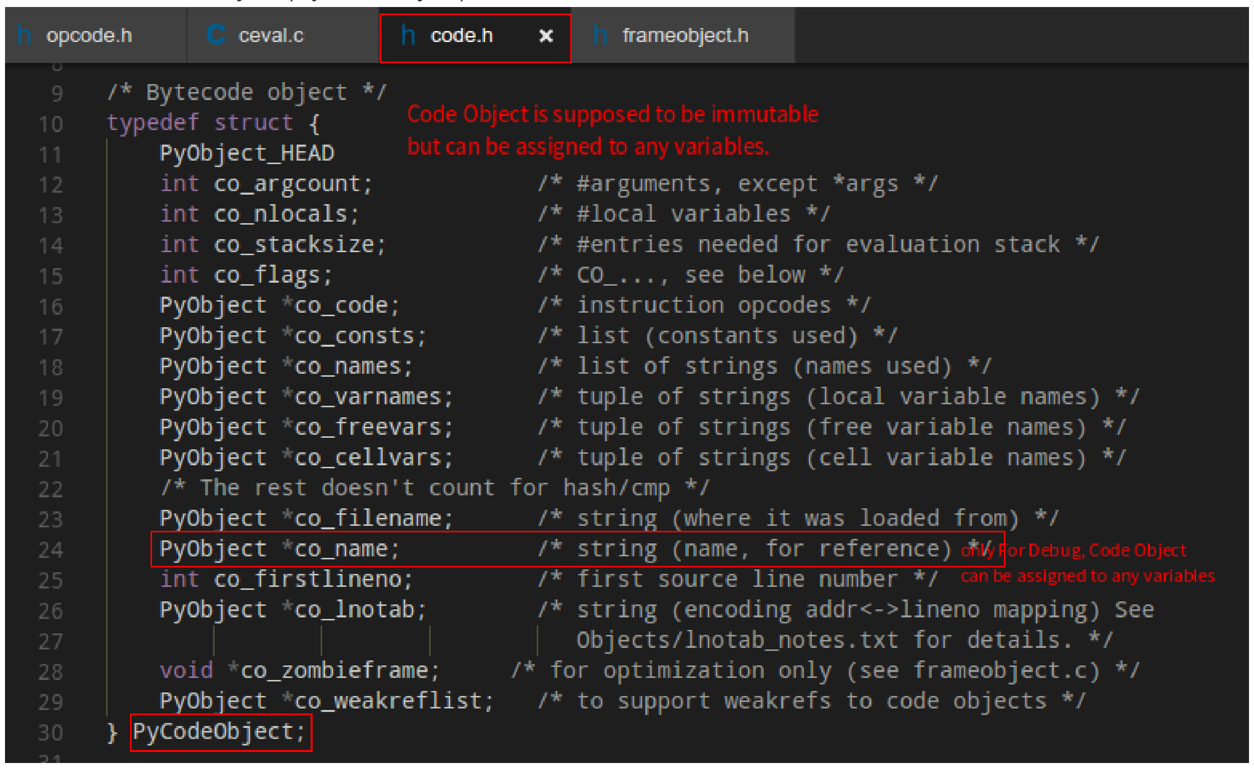


```
x = 10

def foo(x):
    y = x * 2
    return bar(y)

def bar(x):
    y = x / 2
    return y

z = foo(x)
```

wshh08@wshh08-PC: ~/Extra/Program/python/Cpython_internals

```
>>> import dis
>>> import test
>>> test.foo
<function foo at 0x7fe9ff6b18c0>
>>> dis.dis(test.foo)
  4           0 LOAD_FAST                0 (x)
              3 LOAD_CONST               1 (2)
              6 BINARY_MULTIPLY
              7 STORE_FAST               1 (y)

  5          10 LOAD_GLOBAL              0 (bar)
             13 LOAD_FAST                1 (y)
             16 CALL_FUNCTION            1
             19 RETURN_VALUE
>>> dis.dis(test.bar)
  8           0 LOAD_FAST                0 (x)
              3 LOAD_CONST               1 (2)
              6 BINARY_DIVIDE
              7 STORE_FAST               1 (y)

  9          10 LOAD_FAST                1 (y)
             13 RETURN_VALUE
>>>
>>>
```

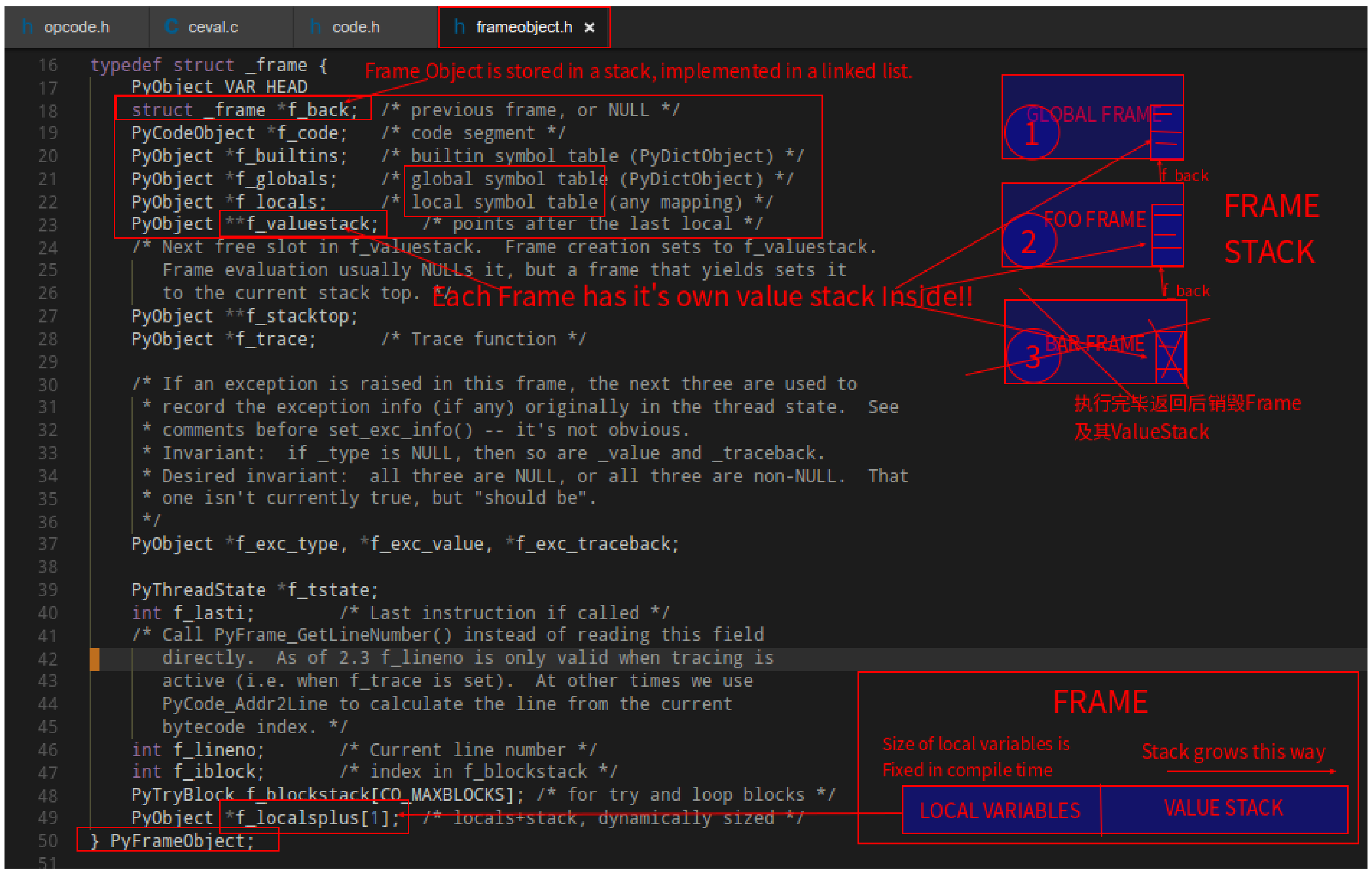
2. Definition of code object (PyCodeObject) - **Include/code.h**



opcode.h
ceval.c
code.h
frameobject.h
Code Object is supposed to be immutable but can be assigned to any variables.
only for Debug, Code Object can be assigned to any variables

```c
/* Bytecode object */
typedef struct {
    PyObject_HEAD
    int co_argcount;            /* #arguments, except *args */
    int co_nlocals;             /* #local variables */
    int co_stacksize;           /* #entries needed for evaluation stack */
    int co_flags;               /* CO_..., see below */
    PyObject *co_code;          /* instruction opcodes */
    PyObject *co_consts;        /* list (constants used) */
    PyObject *co_names;         /* list of strings (names used) */
    PyObject *co_varnames;      /* tuple of strings (local variable names) */
    PyObject *co_freevars;      /* tuple of strings (free variable names) */
    PyObject *co_cellvars;      /* tuple of strings (cell variable names) */
    /* The rest doesn't count for hash/cmp */
    PyObject *co_filename;      /* string (where it was loaded from) */
    PyObject *co_name;          /* string (name, for reference) */
    int co_firstlineno;         /* first source line number */
    PyObject *co_lnotab;        /* string (encoding addr<->lineno mapping) See
                                   Objects/lnotab_notes.txt for details. */
    void *co_zombieframe;     /* for optimization only (see frameobject.c) */
    PyObject *co_weakreflist;   /* to support weakrefs to code objects */
} PyCodeObject;
```

3. Definition of the frame object(PyFrameObject) - **Include/frameobject.h**

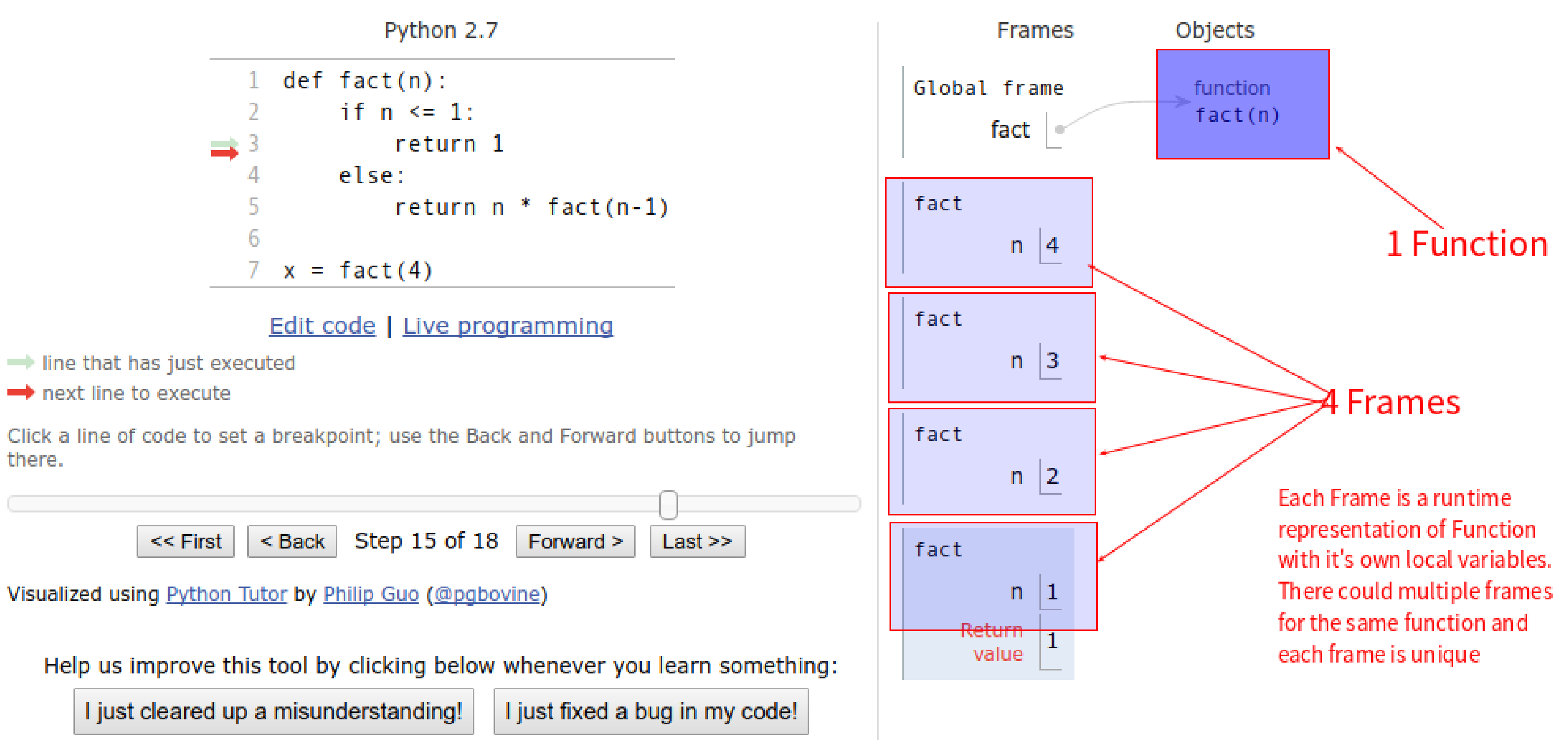


opcode.h
ceval.c
code.h
frameobject.h
Frame Object is stored in a stack, implemented in a linked list.
Each Frame has it's own value stack Inside!!
GLOBAL FRAME
1
f_back
FOO FRAME
2
f_back
BAR FRAME
3
FRAME STACK
执行完毕返回后销毁Frame 及其ValueStack
FRAME
Size of local variables is Fixed in compile time
Stack grows this way
LOCAL VARIABLES
VALUE STACK

```c
typedef struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;  /* previous frame, or NULL */
    PyCodeObject *f_code;   /* code segment */
    PyObject *f_builtins;   /* builtin symbol table (PyDictObject) */
    PyObject *f_globals;    /* global symbol table (PyDictObject) */
    PyObject *f_locals;     /* local symbol table (any mapping) */
    PyObject **f_valuestack;    /* points after the last local */
    /* Next free slot in f_valuestack.  Frame creation sets to f_valuestack.
       Frame evaluation usually NULLs it, but a frame that yields sets it
       to the current stack top. */
    PyObject **f_stacktop;
    PyObject *f_trace;      /* Trace function */

    /* If an exception is raised in this frame, the next three are used to
     * record the exception info (if any) originally in the thread state.  See
     * comments before set_exc_info() -- it's not obvious.
     * Invariant:  if _type is NULL, then so are _value and _traceback.
     * Desired invariant:  all three are NULL, or all three are non-NULL.  That
     * one isn't currently true, but "should be".
     */
    PyObject *f_exc_type, *f_exc_value, *f_exc_traceback;

    PyThreadState *f_tstate;
    int f_lasti;        /* Last instruction if called */
    /* Call PyFrame_GetLineNumber() instead of reading this field
       directly.  As of 2.3 f_lineno is only valid when tracing is
       active (i.e. when f_trace is set).  At other times we use
       PyCode_Addr2Line to calculate the line from the current
       bytecode index. */
    int f_lineno;       /* Current line number */
    int f_iblock;       /* index in f_blockstack */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks */
    PyObject *f_localsplus[1];  /* locals+stack, dynamically sized */
} PyFrameObject;
```

4. Terminology Clarity:
    - **Code**: The most primitive thing, a bunch of bytecode. A Code Object has bytecode, and also has some extra semantic information like constants, variable names.
    - **Function**: A function has a code object, and also has a environment pointer to where was defined.
    - **Frame**: Also has a code object, also has a environment pointer, it's the representation of code at runtime while running it.

How do I use this?



```python
1  def fact(n):
2      if n <= 1:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  x = fact(4)
```

Edit code | Live programming

line that has just executed
next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

Visualized using Python Tutor by Philip Guo (@pgbovine)

Help us improve this tool by clicking below whenever you learn something:

Each Frame is a runtime representation of Function with it's own local variables. There could multiple frames for the same function and each frame is unique

5. Detail about the CALL_FUNCTION instruction:
    - line 2671-2686 in **ceval.c**: instruction **CALL_FUNCTION(argc)**，CALL_FUNCTION instruction only need get the argument argc(the low byte of argc indicates the number of positional arguments, high byte number of keyword parameters) from the stack, and interpreter will get proper parameters from stack automatically.

```c
case CALL_FUNCTION:
    {
        PyObject **sp;
        PCALL(PCALL_ALL);
        sp = stack_pointer;
                #ifdef WITH_TSC
        x = call_function(&sp, oparg, &intr0, &intr1);
                #else
        x = call_function(&sp, oparg);
                #endif
        stack_pointer = sp;
        PUSH(x);
        if (x != NULL)
            continue;
        break;
    }
```

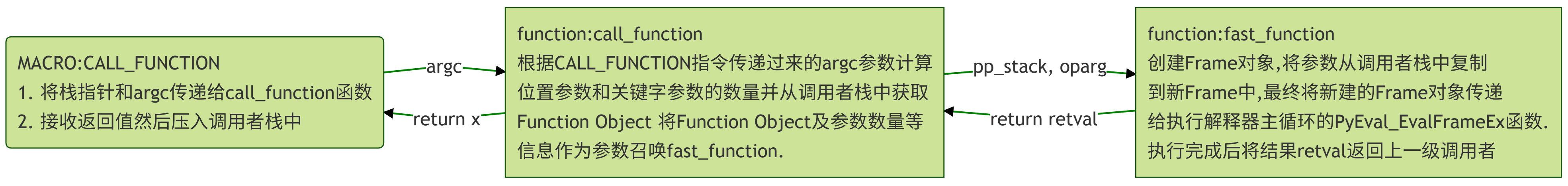
    - line 3991-4071 in **ceval.c**: **call_function(栈指针, 参数数量oparg)**,只需传递参数数量，具体参数从栈中获取，返回值最终被PUSH(x)语句压入当前栈中。

```
static PyObject *call_function(PyObject ***pp_stack, int oparg)
{
  // na: number of positional parameters
  int na = oparg & 0xff;
  // nk: number of keyword parameters
  int nk = (oparg>>8) & 0xff;
  int n = na + 2 * nk;
  PyObject **pfunc = (*pp_stack) - n - 1;
  PyObject *func = *pfunc;
  PyObject *x, *w;

  /* Always dispatch PyCFunction first, because these are
     presumed to be the most frequent callable object.
  */
  if (PyCFunction_Check(func) && nk == 0) {
   ... handle a C function
  } else {
          /* If it's a regular Python Function, call the fast_function(...) */
      if (PyFunction_Check(func))
          x = fast_function(func, pp_stack, n, na, nk);
      else
          /* Go this way when call the CALL_FUNCTION instruction
           * to make instance of a class, at this time 'func' is
               * a Class Objection instead a true Function Object
           */
          x = do_call(func, pp_stack, na, nk);
          Py_DECREF(func);
    }
  /* Clear the stack of the function object.  Also removes
     the arguments in case they weren't consumed already
     (fast_function() and err_args() leave them on the stack).
  */
  while ((*pp_stack) > pfunc) {
      w = EXT_POP(*pp_stack);
      Py_DECREF(w);
      PCALL(PCALL_POP);
  }
  return x;
}
```

- line 4082-4133 in ceval.c: fast_function(函数对象指针, 栈指针, 所有参数在栈中占据空间, 位置参数数量，键参数数量):根据传入的参数从栈中获得参数执行函数对象返回PyObject对象

```
static PyObject *
fast_function(PyObject *func, PyObject ***pp_stack, int n, int na, int nk)
{
        PyCodeObject *co = (PyCodeObject *)PyFunction_GET_CODE(func);
        PyObject *globals = PyFunction_GET_GLOBALS(func);
        PyObject *argdefs = PyFunction_GET_DEFAULTS(func);
        // Create a new frame and assign to f
        PyFrameObject *f;
        f = PyFrame_New(tstate, co, globals, NULL);
        /* Copy arguments from the stack into the new frame
         * f_localsplus is the storage for localvariables and
         * value stacks in the new frame(see last line of the definition of PyFrameObject above)
        */
        fastlocals = f->f_localsplus;
        // stack is the old stack from calling function
    stack = (*pp_stack) - n;
        /* copy arguments from old frame to the new one (n = na + 2 * nk),
         * this operation implements the passing of parameters from caller to the calee.
         */
    for (i = 0; i < n; i++) {
        Py_INCREF(*stack);
        fastlocals[i] = *stack++;
        }
        /* Call the function PyEval_EvalFrameEx(See Lecture2.2.1) that execute
           the interpreter main loop on the new frame we just created.
        */
    retval = PyEval_EvalFrameEx(f,0);
    ++tstate->recursion_depth;
    Py_DECREF(f);
    --tstate->recursion_depth;
        /* 返回执行结果，该结果经过上述几个函数层层返回后
           最终被压入调用了CALL_FUNCTION指令的FRAME的VALUE STACK中
        */
    return retval;
}
```

- 流程图表述整个CALL_FUNCTION指令执行过程：


MACRO:CALL_FUNCTION
1. 将栈指针和argc传递给call_function函数
2. 接收返回值然后压入调用者栈中
argc
return x
function:call_function
根据CALL_FUNCTION指令传递过来的argc参数计算
位置参数和关键字参数的数量并从调用者栈中获取
Function Object 将Function Object及参数数量等
信息作为参数召唤fast_function.
pp_stack, oparg
return retval
function:fast_function
创建Frame对象,将参数从调用者栈中复制
到新Frame中,最终将新建的Frame对象传递
给执行解释器主循环的PyEval_EvalFrameEx函数.
执行完成后将结果retval返回上一级调用者

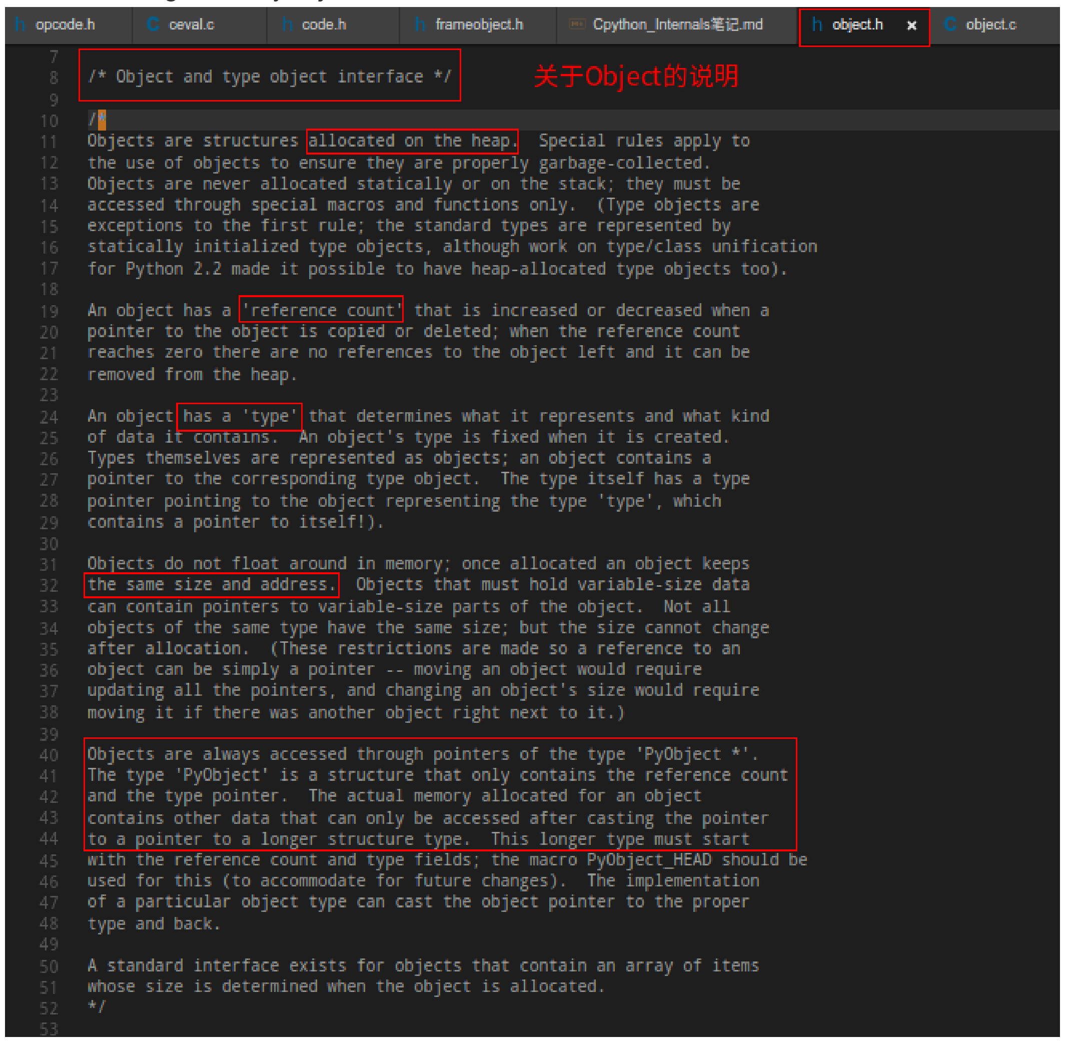
Lecture 4 - PyObject: The core Python object

1. Overview
   - Inspect an object in Python: `dir(obj)` -show everyting(Methods, Properties and so on) inside the mode.

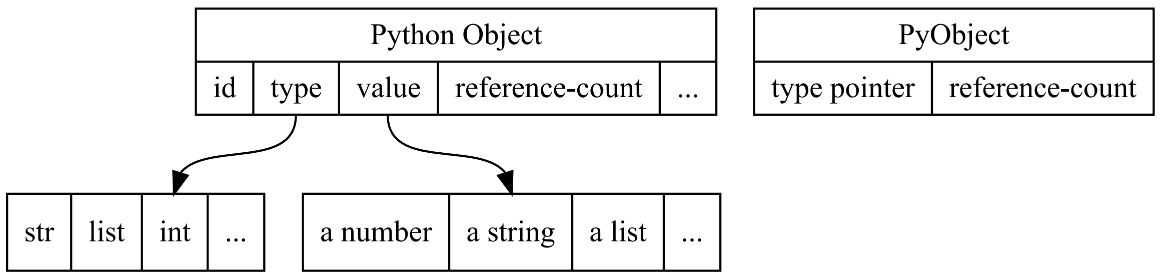
- Everthing in python is an object even a number like 123(instance of PyIntObject), thist is less efficient than C or Java which store numbers directly, but Override an object to add extra features is more flexible.
- Interface of Python always work with PyObjects, everthing should be wrapped in a python object to be handled in Python. There is implemention of every signle type object(see source code in Object/xxxobject.c).
- `id(x)` return memory location of an object. In python once an object is evaluated, it's memory location won't be changed.
- `type(x)` return type information of an object, every object not only has a id, but also has type information.
- 查看对象引用数的语句:

```
from sys import getrefcount
x = ['a', 'b', 'c']
getrefcount(x)
```

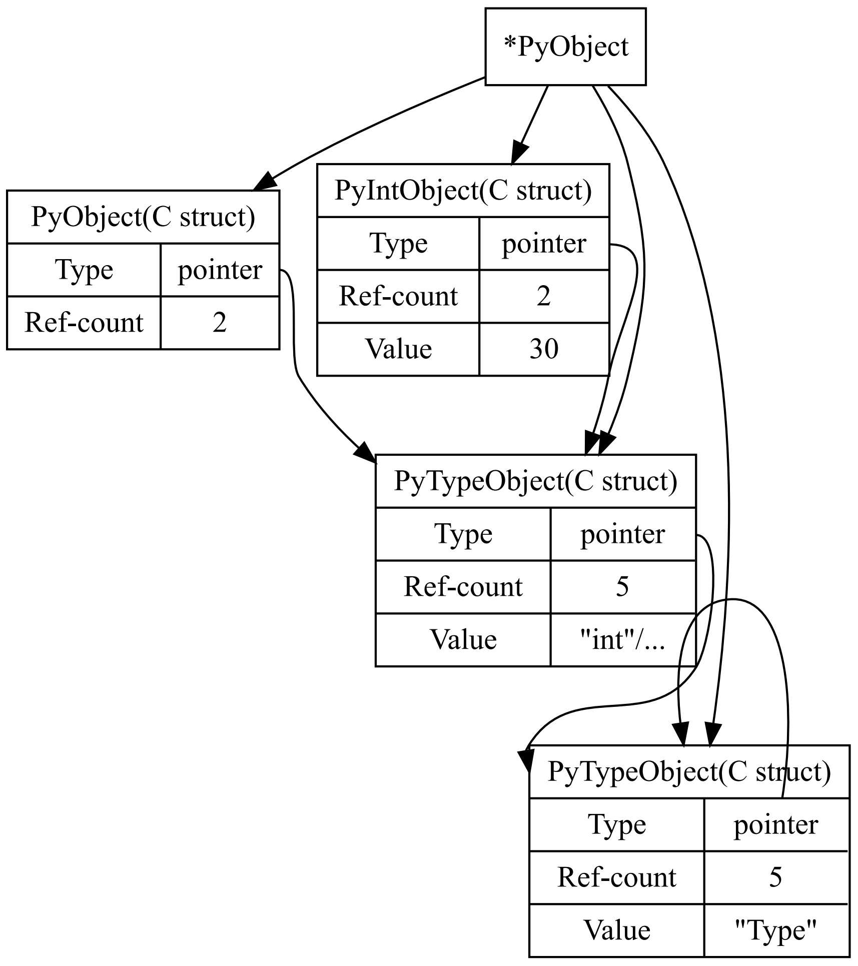
2. Contents of a generic PyObject:



```
/* Object and type object interface */

/*
Objects are structures allocated on the heap.  Special rules apply to
the use of objects to ensure they are properly garbage-collected.
Objects are never allocated statically or on the stack; they must be
accessed through special macros and functions only.  (Type objects are
exceptions to the first rule; the standard types are represented by
statically initialized type objects, although work on type/class unification
for Python 2.2 made it possible to have heap-allocated type objects too).

An object has a 'reference count' that is increased or decreased when a
pointer to the object is copied or deleted; when the reference count
reaches zero there are no references to the object left and it can be
removed from the heap.

An object has a 'type' that determines what it represents and what kind
of data it contains.  An object's type is fixed when it is created.
Types themselves are represented as objects; an object contains a
pointer to the corresponding type object.  The type itself has a type
pointer pointing to the object representing the type 'type', which
contains a pointer to itself!).

Objects do not float around in memory; once allocated an object keeps
the same size and address.  Objects that must hold variable-size data
can contain pointers to variable-size parts of the object.  Not all
objects of the same type have the same size; but the size cannot change
after allocation.  (These restrictions are made so a reference to an
object can be simply a pointer -- moving an object would require
updating all the pointers, and changing an object's size would require
moving it if there was another object right next to it.)

Objects are always accessed through pointers of the type 'PyObject *'.
The type 'PyObject' is a structure that only contains the reference count
and the type pointer.  The actual memory allocated for an object
contains other data that can only be accessed after casting the pointer
to a pointer to a longer structure type.  This longer type must start
with the reference count and type fields; the macro PyObject_HEAD should be
used for this (to accommodate for future changes).  The implementation
of a particular object type can cast the object pointer to the proper
type and back.

A standard interface exists for objects that contain an array of items
whose size is determined when the object is allocated.
*/
```



3. Python Objects in the C world(C structual subtype: utilize the same field in C struct to create a type hierarchical in C)



4. 代码解析
   - line 77-81 and line 101-108 and line 114-116 in **Include/object.h**

```
/* PyObject_HEAD defines the initial segment of every PyObject.
 * _PyObject_HEAD_EXTRA is always empty
 */
#define PyObject_HEAD                   \
        _PyObject_HEAD_EXTRA                \
        Py_ssize_t ob_refcnt;               \
        struct _typeobject *ob_type;
/* Nothing is actually declared to be a PyObject, but every pointer to
         * a Python object can be cast to a PyObject*.  This is inheritance built
         * by hand.  Similarly every pointer to a variable-size Python object can,
         * in addition, be cast to PyVarObject*.
         */
typedef struct _object {
        PyObject_HEAD   // subtype
} PyObject;

/* Macros to get reference count, type and size from Python Objects */
#define Py_REFCNT(ob)           (((PyObject*)(ob))->ob_refcnt)
#define Py_TYPE(ob)             (((PyObject*)(ob))->ob_type)
#define Py_SIZE(ob)             (((PyVarObject*)(ob))->ob_size)
```

- line 25-26 in **Include/intobject.h**

```
typedef struct {
    PyObject_HEAD       // Type & Reference Count
    long ob_ival;       // Value
} PyIntObject;
```

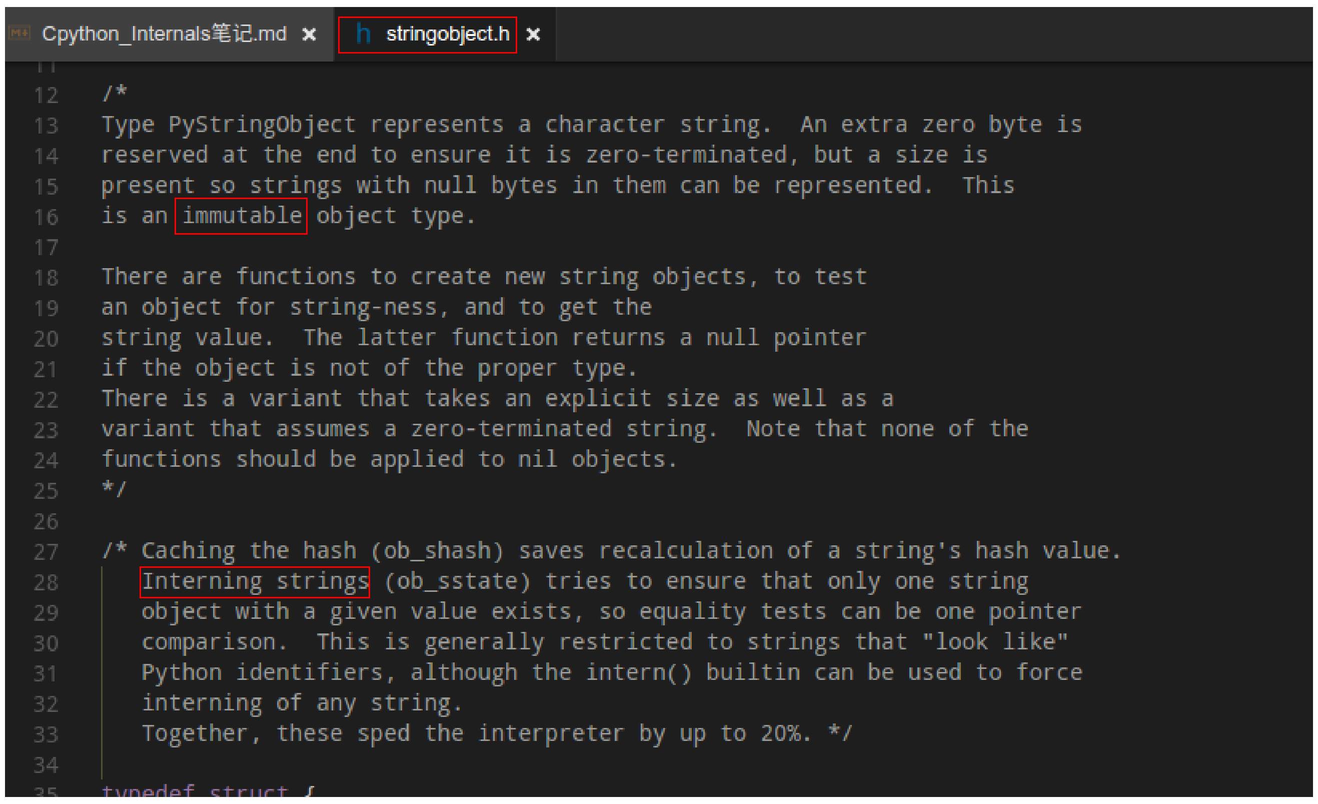
- **Objects/object.c**: generic thing can be done to an object

```
// line 240-248: How was an object created
PyObject *
_PyObject_New(PyTypeObject *tp)
{
    PyObject *op;
        /* Allocate new memory and put it to a PyObject pointer 'op' */
    op = (PyObject *) PyObject_MALLOC(_PyObject_SIZE(tp));
    if (op == NULL)
        return PyErr_NoMemory();
        /* Call the constructor to initialize the object */
    return PyObject_INIT(op, tp);
}
// line 448-467 and line 406-446: How was an object converted to a string
PyObject *
PyObject_Str(PyObject *v)
{
    PyObject *res = _PyObject_Str(v);
    if (res == NULL)
        return NULL;
    assert(PyString_Check(res));
    return res;
}

PyObject *
_PyObject_Str(PyObject *v)
{
    PyObject *res;
    int type_ok;
    if (v == NULL)
        return PyString_FromString("<NULL>");
    if (PyString_CheckExact(v)) {
        Py_INCREF(v);
        return v;
    }
    if (Py_TYPE(v)->tp_str == NULL)
        return PyObject_Repr(v);

    /* It is possible for a type to have a tp_str representation that loops
       infinitely. */
    if (Py_EnterRecursiveCall(" while getting the str of an object"))
        return NULL;
        /* Every type that can be converted to String should have
           implemented the tp_str method. This is where the
           dynamic(One Interface call tp_str, and each type
           implements it's own version of tp_str) comes in.
           This is the thing that jump into the specific code
           for each type.
         */
    res = (*Py_TYPE(v)->tp_str)(v);
    if (res == NULL)
        return NULL;
    type_ok = PyString_Check(res);
    if (!type_ok) {
                ...
        return NULL;
    }
    return res;
}
```

Lecture 5 - Example Python data types

1. Objective of this lecture: To understand how Python data types are "subtypes" of the core PyObject.
    - Intro to Python sequence types -- tuples, lists, strings
    - Abstract object interface: **Objects/abstract.c**

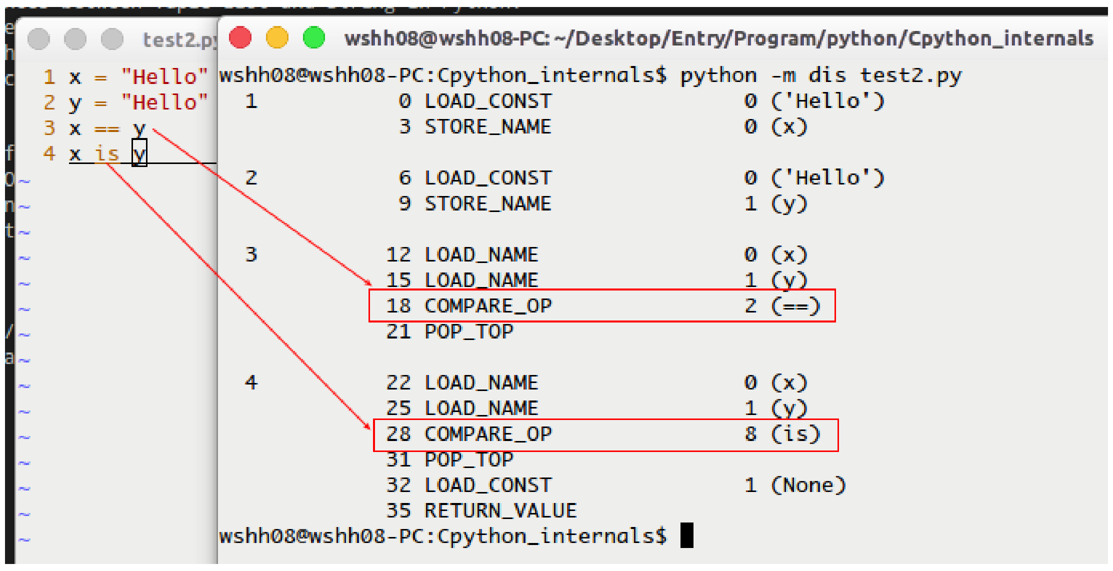
- String type: **Objects/stringobject.c**

2. Differences between Tuple List and String in Python:
    - `Tuple` is immutable, `List` is mutable and variable length, `String` is a sequence of characters(can be accessed by index), it's immutable.
3. About the String object:
    - 

```
/*
Type PyStringObject represents a character string.  An extra zero byte is
reserved at the end to ensure it is zero-terminated, but a size is
present so strings with null bytes in them can be represented.  This
is an immutable object type.

There are functions to create new string objects, to test
an object for string-ness, and to get the
string value.  The latter function returns a null pointer
if the object is not of the proper type.
There is a variant that takes an explicit size as well as a
variant that assumes a zero-terminated string.  Note that none of the
functions should be applied to nil objects.
*/

/* Caching the hash (ob_shash) saves recalculation of a string's hash value.
   Interning strings (ob_sstate) tries to ensure that only one string
   object with a given value exists, so equality tests can be one pointer
   comparison.  This is generally restricted to strings that "look like"
   Python identifiers, although the intern() builtin can be used to force
   interning of any string.
   Together, these sped the interpreter by up to 20%. */

typedef struct {
```

    - line 35-49 in **Include/stringobject.h**: definition of PyStringObject

```
typedef struct {
    PyObject_VAR_HEAD   // ref-count & type pointer & size
    long ob_shash;      // Hash cache
    int ob_sstate;      // String interned or not
        /* Memmory for storing characters will be allocated by malloc() right after ob_sval
           Once an PyObject struct is casted to a PyStringObject, we can access
           characters by offeset to the ob_sval.
         */
    char ob_sval[1];

    /* Invariants(不变的):
     *     ob_sval contains space for 'ob_size+1' elements.
     *     ob_sval[ob_size] == 0.
     *     ob_shash is the hash of the string or -1 if not computed yet.
     *     ob_sstate != 0 iff the string object is in stringobject.c's
     *       'interned' dictionary; in this case the two references
     *       from 'interned' to this object are *not counted* in ob_refcnt.
     */
} PyStringObject;
```

4. In Python `x == y` means content of x and y is equal. but `x is y` means id(x) equals to id(y), so to speak the variable x and y is associated to the same object, and address of this object must be equal.
    - 

test2.py
wshh08@wshh08-PC: ~/Desktop/Entry/Program/python/Cpython_internals

```
x = "Hello"
y = "Hello"
x == y
x is y
```

```
wshh08@wshh08-PC:Cpython_internals$ python -m dis test2.py
  1           0 LOAD_CONST               0 ('Hello')
              3 STORE_NAME               0 (x)

  2           6 LOAD_CONST               0 ('Hello')
              9 STORE_NAME               1 (y)

  3          12 LOAD_NAME                0 (x)
             15 LOAD_NAME                1 (y)
             18 COMPARE_OP               2 (==)
             21 POP_TOP

  4          22 LOAD_NAME                0 (x)
             25 LOAD_NAME                1 (y)
             28 COMPARE_OP               8 (is)
             31 POP_TOP
             32 LOAD_CONST               1 (None)
             35 RETURN_VALUE
wshh08@wshh08-PC:Cpython_internals$
```

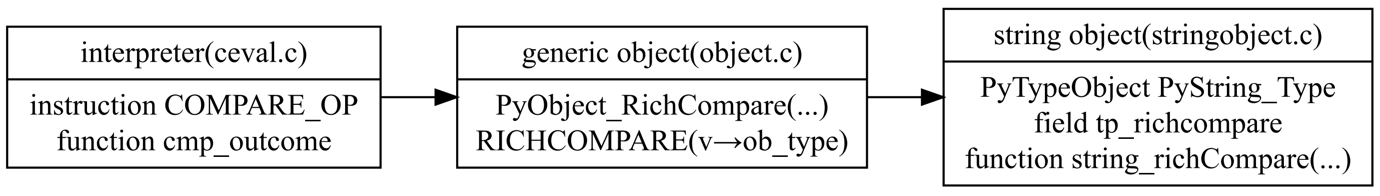
    - line 2275-2308 and line 4463-4543 in **Include/ceval.c**: Implementation of instruction `COMPARE_OP` in python

```
case COMPARE_OP:
    w = POP();
    v = TOP();
        // if both integers, go inside this.
    if (PyInt_CheckExact(w) && PyInt_CheckExact(v)) {
                ...
    }
    else {
      slow_compare:
                /* oparg: argument of instruction, "==", ">", "<=" or "is" and so on */
        x = cmp_outcome(oparg, v, w);
    }
        ...
    continue;

static PyObject *
cmp_outcome(int op, register PyObject *v, register PyObject *w)
{
    int res = 0;
        /* switch based on the operation */
    switch (op) {
    case PyCmp_IS:
                /* what an 'is' operator does in python is
                   just checking two pointers make sure it's equal,
                   it can be very fast
                */
        res = (v == w);
        break;
    case PyCmp_IS_NOT:
        res = (v != w);
        break;
    case PyCmp_IN:
        res = PySequence_Contains(w, v);
        if (res < 0)
            return NULL;
        break;
    case PyCmp_NOT_IN:
        res = PySequence_Contains(w, v);
        if (res < 0)
            return NULL;
        res = !res;
        break;
    case PyCmp_EXC_MATCH:
                ...
    default:
                /* the most general compare function PyObject_RichCompare(...) defined in Objects/object.c */
        return PyObject_RichCompare(v, w, op);
    }
    v = res ? Py_True : Py_False;
    Py_INCREF(v);
    return v;
}
```

- line 593-595 943-979 in **Objects/object.c**: PyObject_RichCompare(...), the most general compare function.

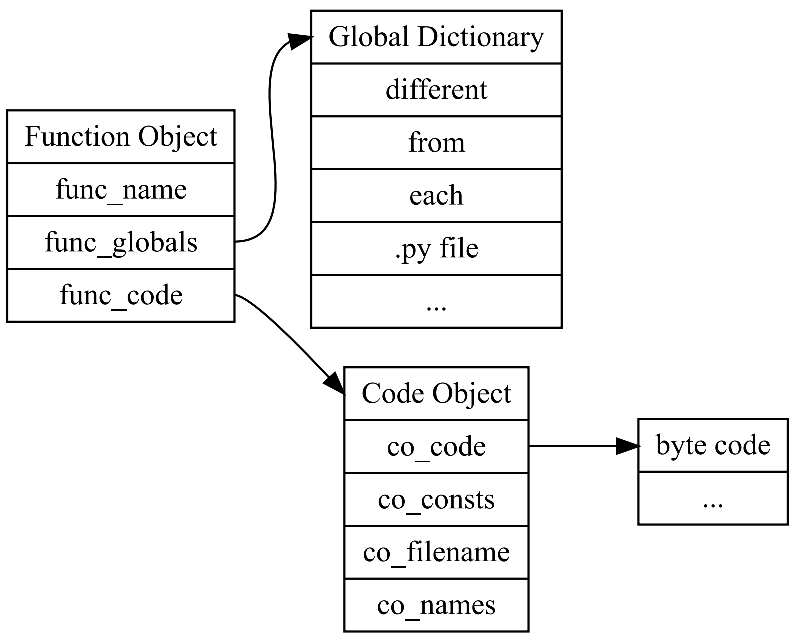
```c
/* Macro to get the tp_richcompare field of a type if defined */
#define RICHCOMPARE(t) (PyType_HasFeature((t), Py_TPFLAGS_HAVE_RICHCOMPARE) \
             ? (t)->tp_richcompare : NULL)
/* tp_richcompare is a field in struct PyString_Type(defined in line 3838 of 'Objects/stringobject.c'):
 * PyTypeObject PyString_Type = {
 * ...
 * * function string_richcompare implemented in line 1191 of 'Objects/stringobject.c'
 * (richcmpfunc)string_richcompare,            /* tp_richcompare
 * ...
 * };
 *
 * definition of function pointer type 'richcmpfunc' in Objects/object.h
 * typedef PyObject *(*richcmpfunc) (PyObject *, PyObject *, int);
 */

PyObject *
PyObject_RichCompare(PyObject *v, PyObject *w, int op
{
    PyObject *res;

    assert(Py_LT <= op && op <= Py_GE);
    if (Py_EnterRecursiveCall(" in cmp"))
        return NULL;

    /* If the types are equal, and not old-style instances, try to
       get out cheap (don't bother with coercions etc.). */
    if (v->ob_type == w->ob_type && !PyInstance_Check(v)) {
        cmpfunc fcmp;
                /* grab the field 'tp_richcompare'(a function pointer) out from the type object and then apply it */
        richcmpfunc frich = RICHCOMPARE(v->ob_type);
        /* If the type has richcmp, try it first.  try_rich_compare
           tries it two-sided, which is not needed since we've a
           single type only. */
        if (frich != NULL) {
            res = (*frich)(v, w, op);
            if (res != Py_NotImplemented)
                goto Done;
            Py_DECREF(res);
        }
        /* No richcmp, or this particular richmp not implemented.
           Try 3-way cmp. */
        fcmp = v->ob_type->tp_compare;
        if (fcmp != NULL) {
            int c = (*fcmp)(v, w);
            c = adjust_tp_compare(c);
            if (c == -2) {
                res = NULL;
                goto Done;
            }
            res = convert_3way_to_object(op, c);
            goto Done;
        }
    }
```

- 流程图总结：



Lecture 6 - Code objects, function objects, and closures

1. Objective: To understand how Python functions are simply PyObject structures
   - Code objects
   - Function objects
   - Closures
2. Contents of a function object:
   - foo.func_name: Function name
   - foo.func_dict
   - foo.func_globals: pointer to the global dictionary, when you want to access a global variable in your function code, you can know where to access. As if there is only one global dictionary, why does each function object need a pointer to it? Why don't use just one global pointer that everyone can use it to access global dictionary? Because, if your project has 10 different files, each file would have a different global, each function need to track its own globals.
   - foo.func_code: pointer of a code object, holding the byte coode of the function.



3. **NOTE**: the function object isn't created until you actually execute the line of code that define the function. But all the code objects have actually been precompiled, the code object is made during the compilation stage.
4. 代码解析:
    - line 10-30 in **Include/code.h**: Definition of code object, these information can be accessed by `foo.co_code.xxxx` , the code object doesn't contain any pointer to the global variables.

```
/* Bytecode object */
typedef struct {
    PyObject_HEAD
    int co_argcount;            /* #arguments, except *args */
    int co_nlocals;             /* #local variables */
    int co_stacksize;           /* #entries needed for evaluation stack */
    int co_flags;               /* CO_..., see below */
    PyObject *co_code;          /* instruction opcodes */
    PyObject *co_consts;        /* list (constants used) */
    PyObject *co_names;         /* list of strings (names used) */
    PyObject *co_varnames;      /* tuple of strings (local variable names) */
    PyObject *co_freevars;      /* tuple of strings (free variable names) */
    PyObject *co_cellvars;      /* tuple of strings (cell variable names) */
    /* The rest doesn't count for hash/cmp */
    PyObject *co_filename;      /* string (where it was loaded from) */
    PyObject *co_name;          /* string (name, for reference) */
    int co_firstlineno;         /* first source line number */
    PyObject *co_lnotab;        /* string (encoding addr<->lineno mapping) See
                                   Objects/lnotab_notes.txt for details. */
    void *co_zombieframe;     /* for optimization only (see frameobject.c) */
    PyObject *co_weakreflist;   /* to support weakrefs to code objects */
} PyCodeObject;
```

    - line 43-109 in **Objects/codeobject.c**: Constructor of the code object.

```
PyCodeObject *
PyCode_New(int argcount, int nlocals, int stacksize, int flags,
           PyObject *code, PyObject *consts, PyObject *names,
           PyObject *varnames, PyObject *freevars, PyObject *cellvars,
           PyObject *filename, PyObject *name, int firstlineno,
           PyObject *lnotab)
{
    /* Check argument types */
        ...
    /* Intern selected string constants */
        ...
    co = PyObject_NEW(PyCodeObject, &PyCode_Type);

    return co;
}
```

    - line 472-511 in **Objects/codeobject.c**: Type Object of code object, it contains slots for all functions you can call on the code object.

```
PyTypeObject PyCode_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "code",
        ...
        (cmpfunc)code_compare,              /* tp_compare */
        ...
}
```

    - line 21-38 in ***Include/funcobject.h***: Definition of the Function Object.

```
/* Function objects and code objects should not be confused with each other:
 *
 * Function objects are created by the execution of the 'def' statement.
 * They reference a code object in their func_code attribute, which is a
 * purely syntactic object, i.e. nothing more than a compiled version of some
 * source code lines.  There is one code object per source code "fragment",
 * but each code object can be referenced by zero or many function objects
 * depending only on how many times the 'def' statement in the source was
 * executed so far.
 */

typedef struct {
    PyObject_HEAD
    PyObject *func_code;        /* A code object */
    PyObject *func_globals;     /* A dictionary (other mappings won't do) */
    PyObject *func_defaults;    /* NULL or a tuple(default parameters) */
    PyObject *func_closure;     /* NULL or a tuple of cell objects */
    PyObject *func_doc;         /* The __doc__ attribute, can be anything */
    PyObject *func_name;        /* The __name__ attribute, a string object */
    PyObject *func_dict;        /* The __dict__ attribute, a dict or NULL */
    PyObject *func_weakreflist; /* List of weak references */
    PyObject *func_module;      /* The __module__ attribute, can be anything */

    /* Invariant:
     *     func_closure contains the bindings for func_code->co_freevars, so
     *     PyTuple_Size(func_closure) == PyCode_GetNumFree(func_code)
     *     (func_closure may be NULL if PyCode_GetNumFree(func_code) == 0).
     */
} PyFunctionObject;
```

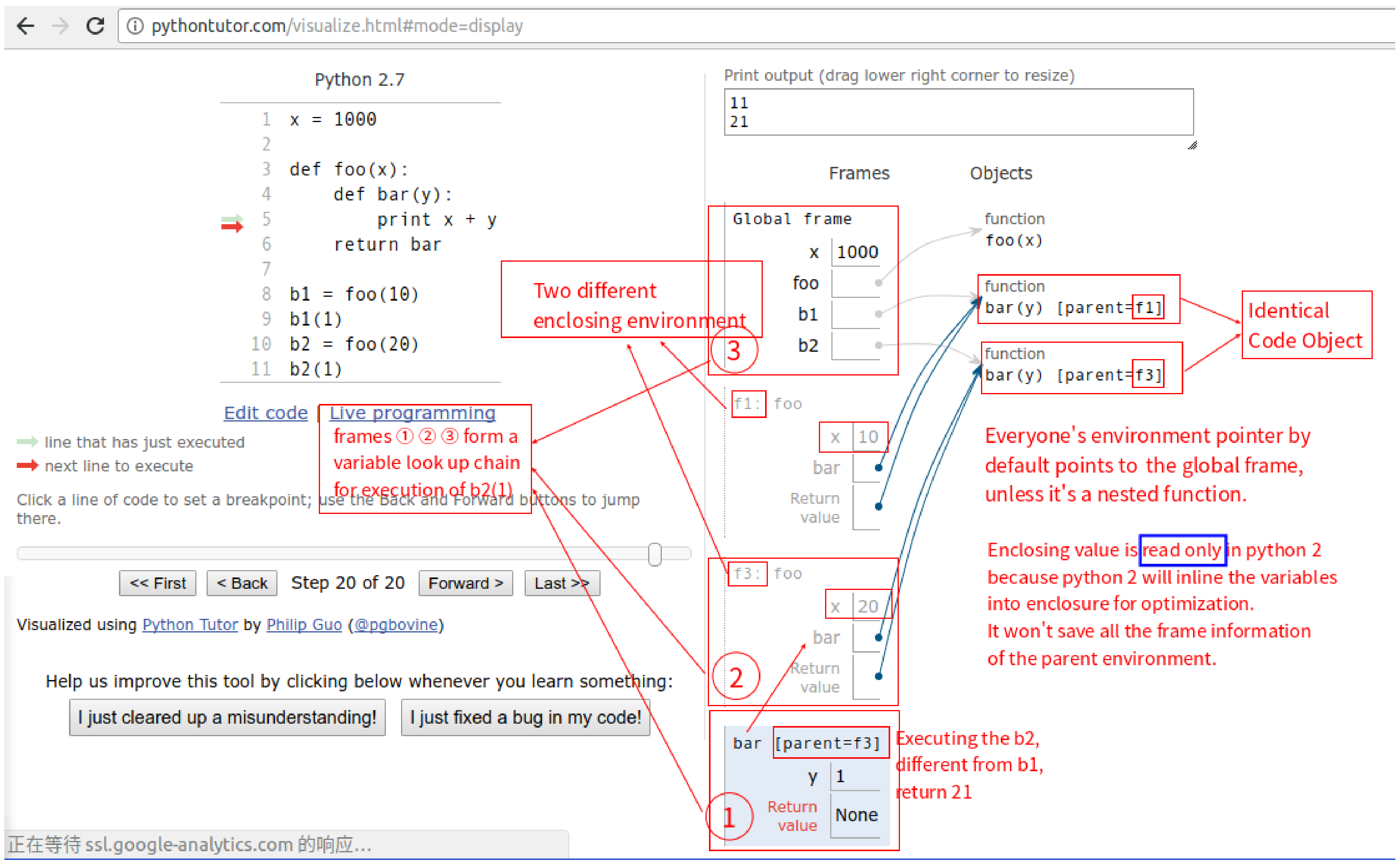
- line 9-61 in **Objects/funcobject.c**: Function Object = Code Object + Globals

```
PyObject *
PyFunction_New(PyObject *code, PyObject *globals)
{
        ...
}
```

- line 159-174 and line 332-344 in **Objects/funcobj.c**: Maps python name(like `func_closure` , `__closure__`) to actual fields in C struct, to allow python code access directly variables defined in C.

```
#define OFF(x) offsetof(PyFunctionObject, x)

/* Read Only */
static PyMemberDef func_memberlist[] = {
    {"func_closure",  T_OBJECT,     OFF(func_closure),
     RESTRICTED|READONLY},
    {"__closure__",  T_OBJECT,      OFF(func_closure),
     RESTRICTED|READONLY},
    {"func_doc",      T_OBJECT,     OFF(func_doc), PY_WRITE_RESTRICTED},
    {"__doc__",       T_OBJECT,     OFF(func_doc), PY_WRITE_RESTRICTED},
    {"func_globals",  T_OBJECT,     OFF(func_globals),
     RESTRICTED|READONLY},
    {"__globals__",  T_OBJECT,      OFF(func_globals),
     RESTRICTED|READONLY},
    {"__module__",    T_OBJECT,     OFF(func_module), PY_WRITE_RESTRICTED},
    {NULL}  /* Sentinel */
};
/* Read and Set */
static PyGetSetDef func_getsetlist[] = {
    {"func_code", (getter)func_get_code, (setter)func_set_code},
    {"__code__", (getter)func_get_code, (setter)func_set_code},
    {"func_defaults", (getter)func_get_defaults,
     (setter)func_set_defaults},
    {"__defaults__", (getter)func_get_defaults,
     (setter)func_set_defaults},
    {"func_dict", (getter)func_get_dict, (setter)func_set_dict},
    {"__dict__", (getter)func_get_dict, (setter)func_set_dict},
    {"func_name", (getter)func_get_name, (setter)func_set_name},
    {"__name__", (getter)func_get_name, (setter)func_set_name},
    {NULL} /* Sentinel */
};
/* line 547-586: definition of Type Object of the function object */
PyTypeObject PyFunction_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "function",
        ...
        /*function_call is used to call the function*/
        function_call,                              /* tp_call */
        ...
}
/* line 487-524 definition of the method function_call */
static PyObject *
function_call(PyObject *func, PyObject *arg, PyObject *kw)
{
        ...
        /* function PyEval_EvalCodeEx actually get all information
         * out from the Function Object and create the frame then
         * pass to the interpreter main loop function
         * PyEval_EvalFrameEx to execute the function
         * 与前面CALL_FUNCTION指令中调用的fast_function应用场景
         * 有何不同?
         */
        result = PyEval_EvalCodeEx(
            (PyCodeObject *)PyFunction_GET_CODE(func),
            PyFunction_GET_GLOBALS(func), (PyObject *)NULL,
            &PyTuple_GET_ITEM(arg, 0), PyTuple_GET_SIZE(arg),
            k, nk, d, nd,
            PyFunction_GET_CLOSURE(func));

        return result;
}
```

5. Closure



pythontutor.com/visualize.html#mode=display

Python 2.7

```
x = 1000

def foo(x):
    def bar(y):
        print x + y
    return bar

b1 = foo(10)
b1(1)
b2 = foo(20)
b2(1)
```

Edit code | Live programming

line that has just executed
next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First | < Back | Step 20 of 20 | Forward > | Last >>

Visualized using Python Tutor by Philip Guo (@pgbovine)

Help us improve this tool by clicking below whenever you learn something:

I just cleared up a misunderstanding! | I just fixed a bug in my code!

正在等待 ssl.google-analytics.com 的响应...

Print output (drag lower right corner to resize)

```
11
21
```

Two different enclosing environment

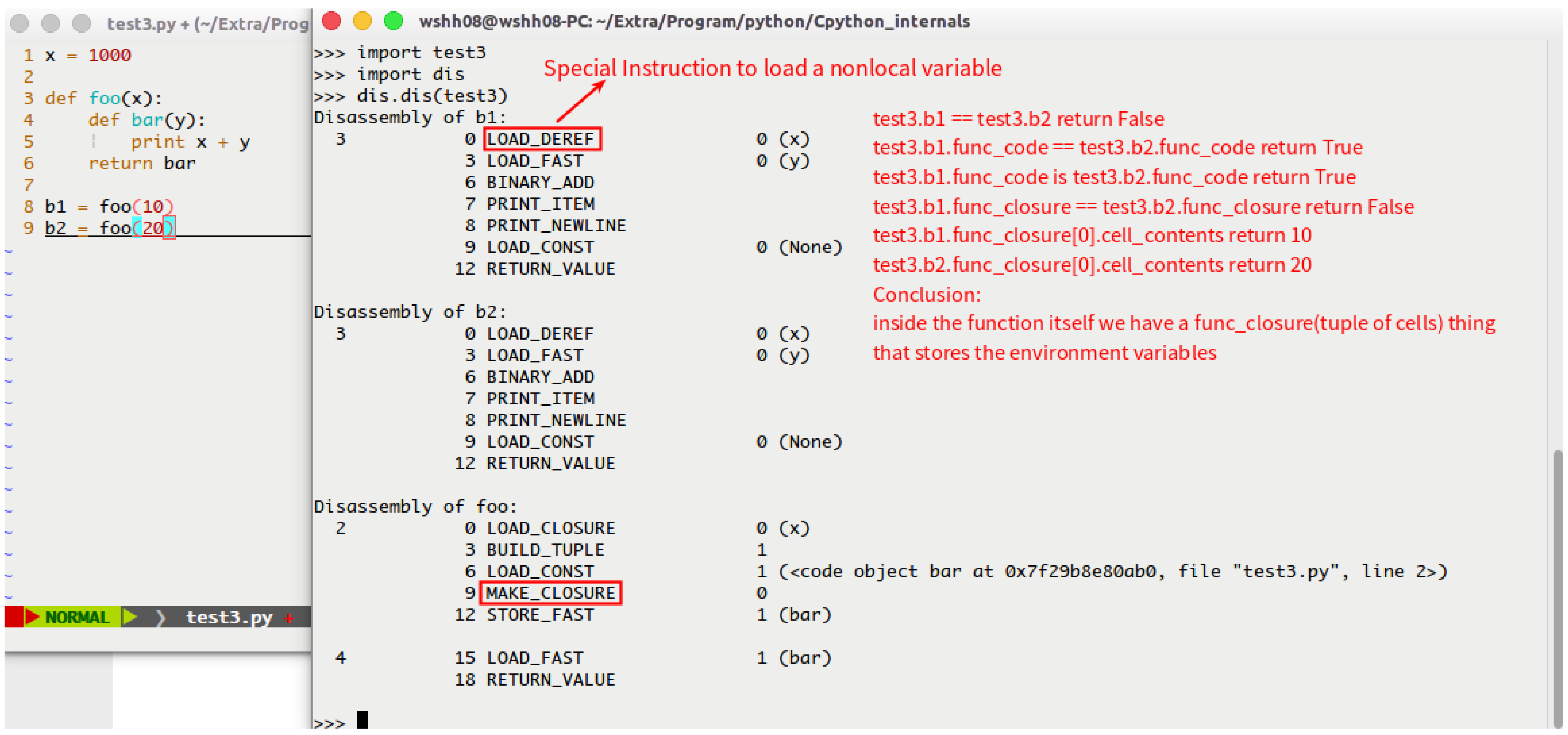
Live programming frames ① ② ③ form a variable look up chain for execution of b2(1)

Identical Code Object

Everyone's environment pointer by default points to the global frame, unless it's a nested function.

Enclosing value is read only in python 2 because python 2 will inline the variables into enclosure for optimization. It won't save all the frame information of the parent environment.

Executing the b2, different from b1, return 21



test3.py + (~/Extra/Prog

```
x = 1000

def foo(x):
    def bar(y):
        print x + y
    return bar

b1 = foo(10)
b2 = foo(20)
```

NORMAL test3.py

wshh08@wshh08-PC: ~/Extra/Program/python/Cpython_internals

```
>>> import test3
>>> import dis
>>> dis.dis(test3)
Disassembly of b1:
  3           0 LOAD_DEREF               0 (x)
              3 LOAD_FAST                0 (y)
              6 BINARY_ADD
              7 PRINT_ITEM
              8 PRINT_NEWLINE
              9 LOAD_CONST               0 (None)
             12 RETURN_VALUE

Disassembly of b2:
  3           0 LOAD_DEREF               0 (x)
              3 LOAD_FAST                0 (y)
              6 BINARY_ADD
              7 PRINT_ITEM
              8 PRINT_NEWLINE
              9 LOAD_CONST               0 (None)
             12 RETURN_VALUE

Disassembly of foo:
  2           0 LOAD_CLOSURE             0 (x)
              3 BUILD_TUPLE              1
              6 LOAD_CONST               1 (<code object bar at 0x7f29b8e80ab0, file "test3.py", line 2>)
              9 MAKE_CLOSURE             0
             12 STORE_FAST               1 (bar)

  4          15 LOAD_FAST                1 (bar)
             18 RETURN_VALUE
>>>
```

Special Instruction to load a nonlocal variable

test3.b1 == test3.b2 return False
test3.b1.func_code == test3.b2.func_code return True
test3.b1.func_code is test3.b2.func_code return True
test3.b1.func_closure == test3.b2.func_closure return False
test3.b1.func_closure[0].cell_contents return 10
test3.b2.func_closure[0].cell_contents return 20
Conclusion:
inside the function itself we have a func_closure(tuple of cells) thing that stores the environment variables

6. Source Code about instruction LOAD_DEREF
    - line 2157-2181 in **Python/ceval.c**

    ```
    case LOAD_DEREF:
                /* get name of the environment variables, can get by:
                 * test3.b2.func_code.co_freevars
                 */
        x = freevars[oparg];
        w = PyCell_Get(x);
                ...
    ```

    - line 26 and line 33-37 in **Include/funcobject.h**

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

/* Invariant:
 *     func_closure contains the bindings for func_code->co_freevars, so
 *     PyTuple_Size(func_closure) == PyCode_GetNumFree(func_code)
 *     (func_closure may be NULL if PyCode_GetNumFree(func_code) == 0).
 */
```

Lecture 7 - Iterators

1. Objective: To understand how iterators and 'for' loops work under the hood.
    - Python 'for' loops over a list: ceval.c, abstract.c, listobject.c
    - Defining a class with a custom iterator: iterator.h, iterobject.c
2. Iterator: A object that encapsulate the act of iterating. Need have one method called `next()` to get the next elememt of iterating.
    - To make a iterator from a list: `i = iter(lst)` or `i = lst.__iter__()` , then the `i` is a iterator object, it's not actually the `lst` , but it holds a pointer(the pointer can only advance forward, can't go backwards, and iterators in python don't support modification) points to the first element of the list. Everytime call the `next()` method on `i` , will return the current element of the pointer and advance the pointer to the next element in the list. At the very end of the list, call the `next()` function will raise a `StopIteration`

exception.

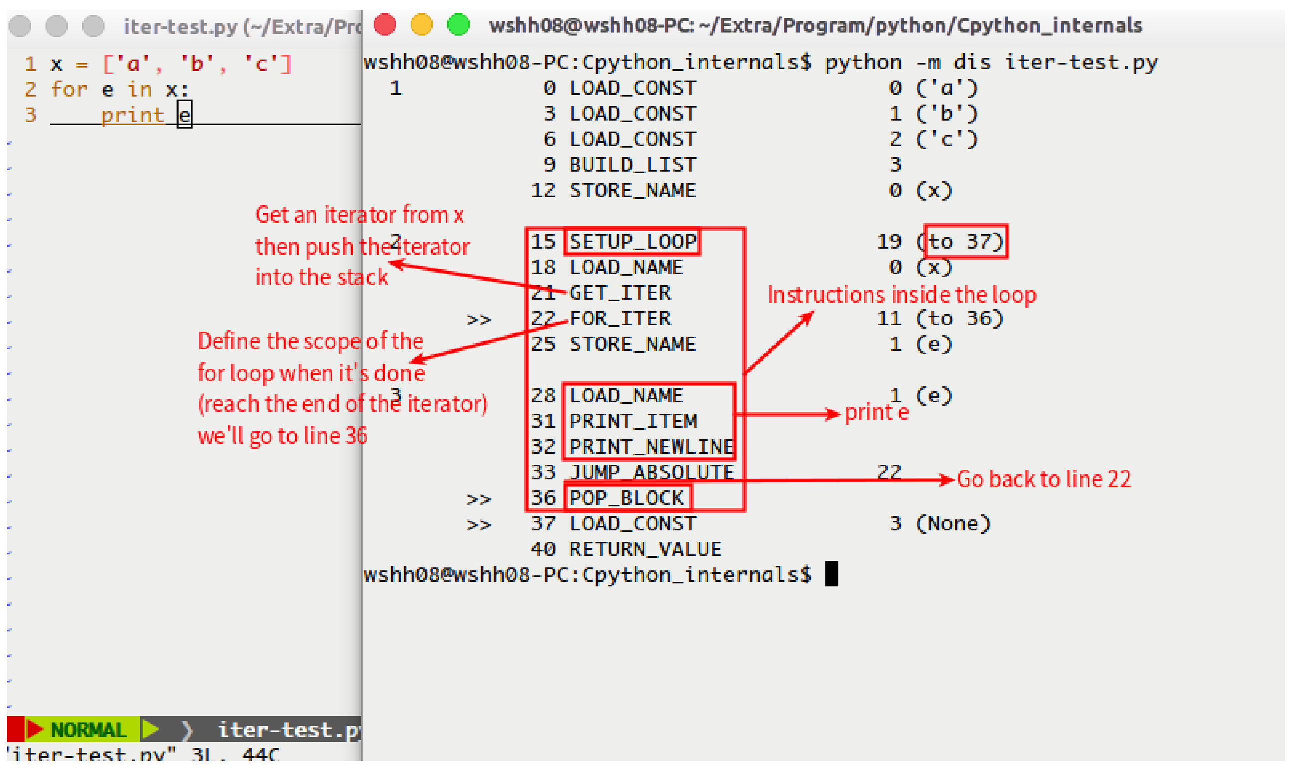
- Why do we need iterator instead of accessing elements by index directly?
    1. You can make code shorter by iterator.
    2. You don't need to worry about bound of the index.
    3. You can define a iterator for a complex data struct(like a tree) and make it retrieve data in the struct in a determined order without worry about the index and its bound.
- Python code of using a iterator to print all elements in a list:

```python
lst = ['a', 'b', 'c']
i = lst.__iter__() # 或者 i = iter(lst)
while True:
    try:
        print i.next()
    except StopIteration:
        break
# This for loop implicitly make an iterator to iterate on the list
# Code behind a for loop is kind of similar to code above
for elt in lst:
    print elt
```

- Using a for loop with builtin iterator to print elements in a dictionary you can only get the keys out in a undeterminaed order.
- A `String` object also has a builtin iterator that can put characters in the string out orderly while calling `next()` on it.

3. Instructions of 'for' loop:

-



```
iter-test.py (~/Extra/Pr
x = ['a', 'b', 'c']
for e in x:
    print e
```

```
wshh08@wshh08-PC: ~/Extra/Program/python/Cpython_internals
wshh08@wshh08-PC:Cpython_internals$ python -m dis iter-test.py
  1           0 LOAD_CONST               0 ('a')
              3 LOAD_CONST               1 ('b')
              6 LOAD_CONST               2 ('c')
              9 BUILD_LIST               3
             12 STORE_NAME               0 (x)

  2          15 SETUP_LOOP              19 (to 37)
             18 LOAD_NAME                0 (x)
             21 GET_ITER
        >>   22 FOR_ITER                11 (to 36)
             25 STORE_NAME               1 (e)

  3          28 LOAD_NAME                1 (e)
             31 PRINT_ITEM
             32 PRINT_NEWLINE
             33 JUMP_ABSOLUTE           22
        >>   36 POP_BLOCK
        >>   37 LOAD_CONST               3 (None)
             40 RETURN_VALUE
wshh08@wshh08-PC:Cpython_internals$
```

Get an iterator from x then push the iterator into the stack
Define the scope of the for loop when it's done (reach the end of the iterator) we'll go to line 36
Instructions inside the loop
print e
Go back to line 22
NORMAL
iter-test.py
"iter-test.py" 3L, 44C

4. Diving into the code of `GET_ITER` and `FOR_ITER` :
    - line 2493-2528 in **Python/ceval.c**

```
case GET_ITER:
    /* before: [obj]; after [getiter(obj)] */
    v = TOP();
    x = PyObject_GetIter(v);
    Py_DECREF(v);
    if (x != NULL) {
        SET_TOP(x);
        /* as GET_ITER is always followed by FOR_ITER,
         * we can jump to FOR_ITER without go back to the
         * main interpreter loop
         */
        PREDICT(FOR_ITER);
        continue;
    }
    STACKADJ(-1);
    break;

PREDICTED_WITH_ARG(FOR_ITER);
case FOR_ITER:
    /* before: [iter]; after: [iter, iter()] *or* [] */
    v = TOP();
    /* call the next() method on the iterator object */
    x = (*v->ob_type->tp_iternext)(v);
    if (x != NULL) {
        PUSH(x);
        /* push the value gotten back to the stack */
        PREDICT(STORE_FAST);
        PREDICT(UNPACK_SEQUENCE);
        /* go back to the main iterpreter loop and execute the next opcode */
        continue;
    }
    if (PyErr_Occurred()) {
        if (!PyErr_ExceptionMatches(
                        PyExc_StopIteration))
            break;
        PyErr_Clear();
    }
    /* iterator ended normally
     * once x == NULL and no exception happend, we'll come here
     * jump to the instruction of POP_BLOCK and then go back to the
     */
    x = v = POP();
    Py_DECREF(v);
    // #define JUMPBY(x)       (next_instr += (x))
    JUMPBY(oparg);
    continue;
```

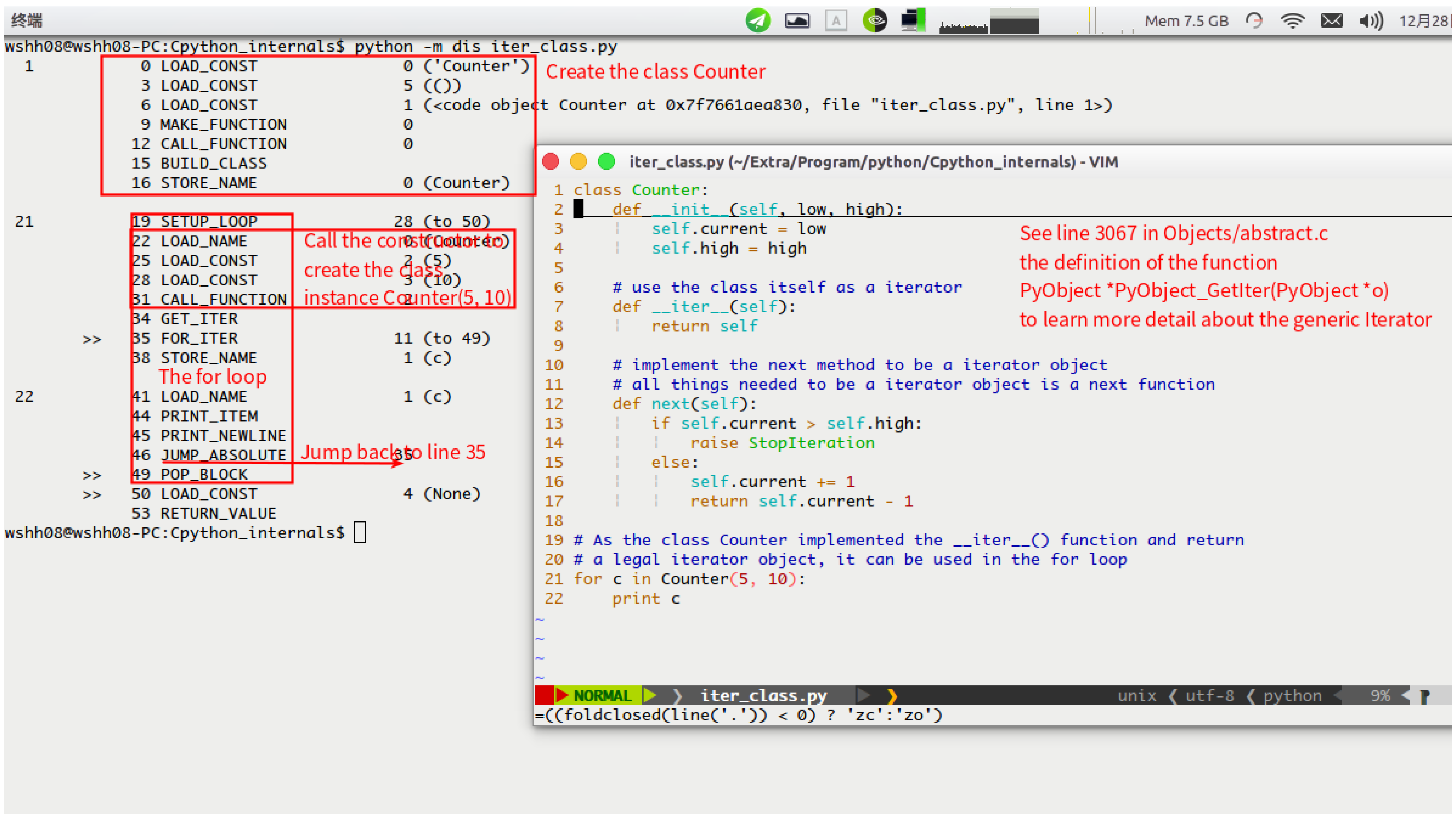
- line 3066-3090 in **Objects/abstract.c**: definition of the function `PyObject_GetIter` - how to get a iterator from a generic object, used by struction `GET_ITER`. There may be 2 kinds of iterators in Python generic iterator defined by the Type Object and the sequence Iterator for Sequence Objects.

```
PyObject *
PyObject_GetIter(PyObject *o)
{
    PyTypeObject *t = o->ob_type;
    getiterfunc f = NULL;
    /* if type of the object has a flag py_TPFLAGS_HAVE_ITER
     * grab the tp_iter method from the type object
     */
    if (PyType_HasFeature(t, Py_TPFLAGS_HAVE_ITER))
        f = t->tp_iter;
    if (f == NULL) {
        /* if there is no tp_iter method, see if it's a sequence */
        if (PySequence_Check(o))
            /* if it's a sequence object, create a sequence iterator
             * from the sequence and then return it
             */
            return PySeqIter_New(o);
        return type_error("'%.200s' object is not iterable", o);
    }
    else {
        PyObject *res = (*f)(o);
        if (res != NULL && !PyIter_Check(res)) {
            PyErr_Format(PyExc_TypeError,
                         "iter() returned non-iterator "
                         "of type '%.100s'",
                         res->ob_type->tp_name);
            Py_DECREF(res);
            res = NULL;
        }
        return res;
    }
}
```

- line 11-28 in **Objects/iterobject.c**: definition of the function PySeqIter_New - how to create an iterator for a sequence object.

```c
PyObject *
PySeqIter_New(PyObject *seq)
{
    seqiterobject *it;

    if (!PySequence_Check(seq)) {
        PyErr_BadInternalCall();
        return NULL;
    }
    /* create the iterator object */
    it = PyObject_GC_New(seqiterobject, &PySeqIter_Type);
    if (it == NULL)
        return NULL;
    /* set index of the iterator 0 */
    it->it_index = 0;
    Py_INCREF(seq);
    /* keep the original sequence object in the iterator for track */
    it->it_seq = seq;
    _PyObject_GC_TRACK(it);
    return (PyObject *)it;
}
/* line 5-9 in Objects/iterobject.c: definition of a sequence iterator object */
typedef struct {
    PyObject_HEAD
    long      it_index;
    PyObject *it_seq; /* Set to NULL when iterator is exhausted */
} seqiterobject;
/* line 96-127 in Objects/iterobject.c:
 * definition of Type Object of sequence iterator object
 */
PyTypeObject PySeqIter_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "iterator",                                 /* tp_name */
    ...
    /* method to get next element from the iterator */
    iter_iternext,                              /* tp_iternext */
    ...
}
/* line 45-71 in Object/iterobject.c: implemention of function iter_iternext */
static PyObject *
iter_iternext(PyObject *iterator)
{
    seqiterobject *it;
    PyObject *seq;
    PyObject *result;

    assert(PySeqIter_Check(iterator));
    it = (seqiterobject *)iterator;
        /* get the sequence object */
    seq = it->it_seq;
    if (seq == NULL)
        return NULL;

    /* get the element with it_inddex to return */
    result = PySequence_GetItem(seq, it->it_index);
    if (result != NULL) {
                /* increase the index by 1: core part of the iterating */
        it->it_index++;
        return result;
    }
    if (PyErr_ExceptionMatches(PyExc_IndexError) ||
        PyErr_ExceptionMatches(PyExc_StopIteration))
    {
        PyErr_Clear();
        Py_DECREF(seq);
        it->it_seq = NULL;
    }
    return NULL;
}
```

- About the generic(non-sequence object) iterator in python
    -

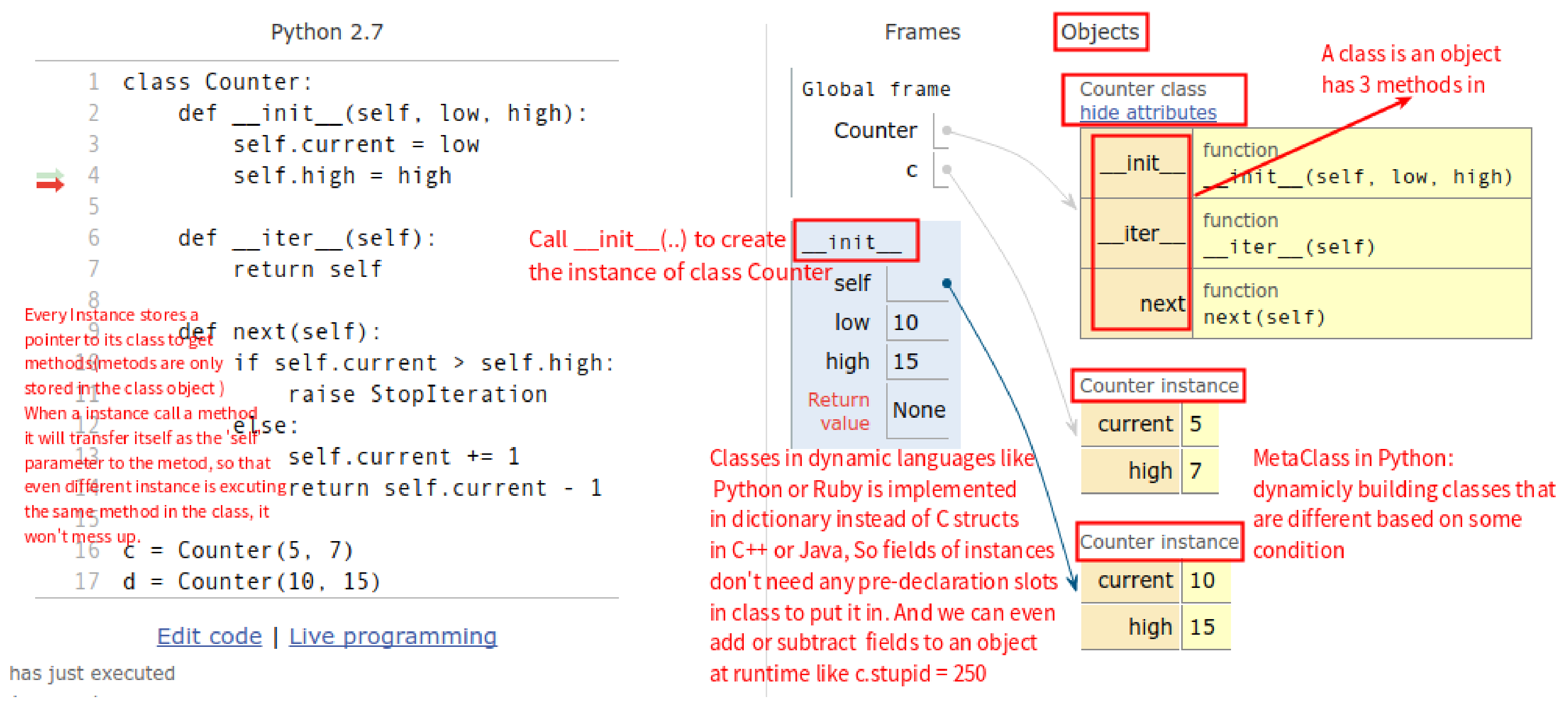

终端
Mem 7.5 GB
12月28
Create the class Counter
Call the constructor to create the class instance Counter(5, 10)
The for loop
Jump back to line 35
iter_class.py (~/Extra/Program/python/Cpython_internals) - VIM
See line 3067 in Objects/abstract.c the definition of the function PyObject *PyObject_GetIter(PyObject *o) to learn more detail about the generic Iterator

```
wshh08@wshh08-PC:Cpython_internals$ python -m dis iter_class.py
  1           0 LOAD_CONST               0 ('Counter')
              3 LOAD_CONST               5 (())
              6 LOAD_CONST               1 (<code object Counter at 0x7f7661aea830, file "iter_class.py", line 1>)
              9 MAKE_FUNCTION            0
             12 CALL_FUNCTION            0
             15 BUILD_CLASS
             16 STORE_NAME               0 (Counter)

 21          19 SETUP_LOOP              28 (to 50)
             22 LOAD_NAME                0 (Counter)
             25 LOAD_CONST               2 (5)
             28 LOAD_CONST               3 (10)
             31 CALL_FUNCTION            2
             34 GET_ITER
        >>   35 FOR_ITER                11 (to 49)
             38 STORE_NAME               1 (c)

 22          41 LOAD_NAME                1 (c)
             44 PRINT_ITEM
             45 PRINT_NEWLINE
             46 JUMP_ABSOLUTE           35
        >>   49 POP_BLOCK
        >>   50 LOAD_CONST               4 (None)
             53 RETURN_VALUE
wshh08@wshh08-PC:Cpython_internals$
```

```python
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    # use the class itself as a iterator
    def __iter__(self):
        return self

    # implement the next method to be a iterator object
    # all things needed to be a iterator object is a next function
    def next(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

# As the class Counter implemented the __iter__() function and return
# a legal iterator object, it can be used in the for loop
for c in Counter(5, 10):
    print c
```

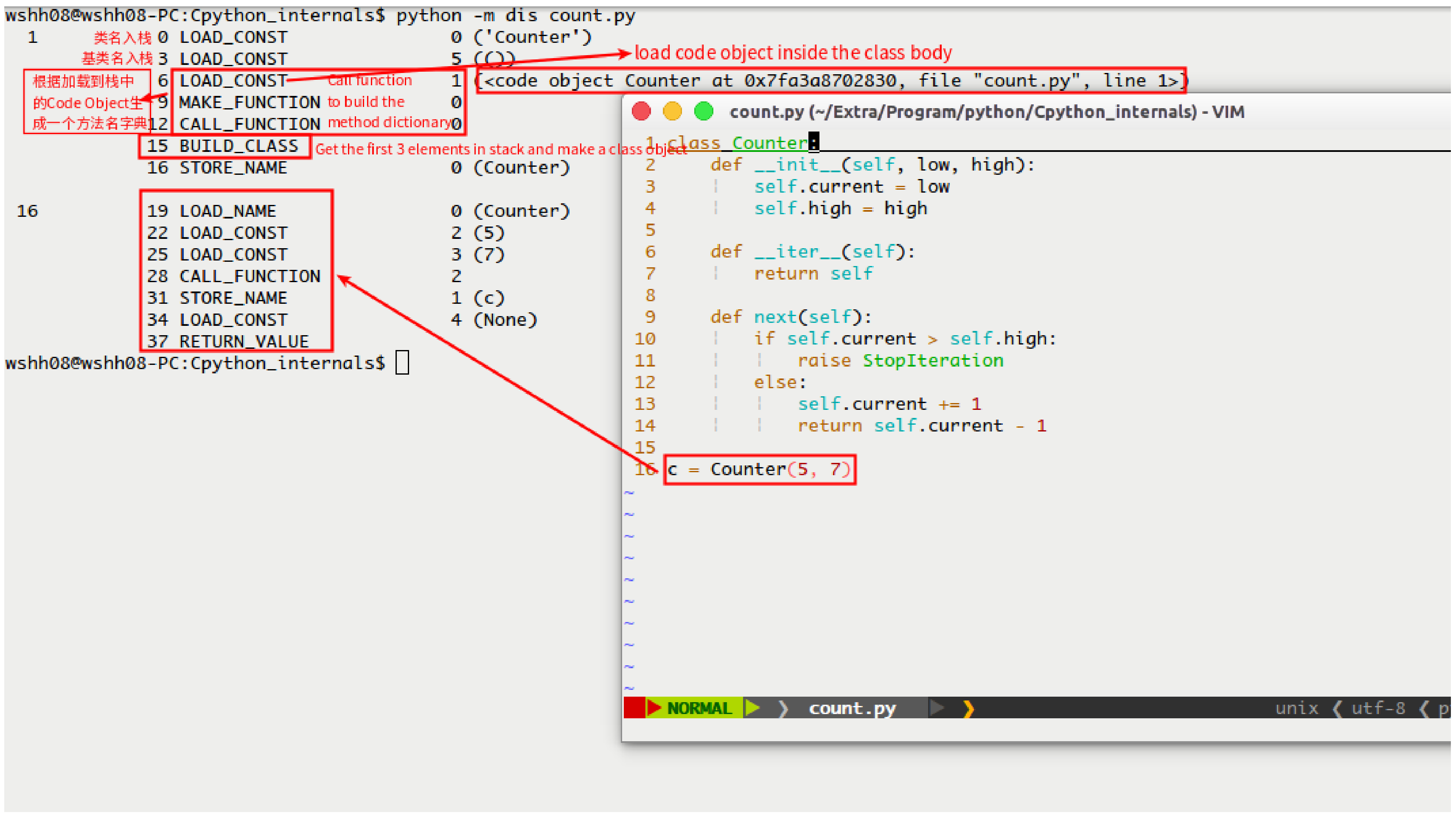
```
 NORMAL   iter_class.py                                   unix < utf-8 < python < 9%
=((foldclosed(line('.')) < 0) ? 'zc':'zo')
```

Lecture 8 - User-defined classes and objects

1. Objective: To understand how Python implements classes and objects
    - Classes and objects in Python
    - Include/classobject.h, Objects/classobject.c
2. Overview:
    - 
Python 2.7
Frames
Objects
Global frame
Counter
c
Counter class
hide attributes
__init__
__iter__
next
function
__init__(self, low, high)
function
__iter__(self)
function
next(self)
__init__
self
low
10
high
15
Return value
None
Counter instance
current
5
high
7
Counter instance
current
10
high
15
A class is an object has 3 methods in
Call __init__(..) to create the instance of class Counter
Every Instance stores a pointer to its class to get methods(metods are only stored in the class object)
When a instance call a method it will transfer itself as the 'self' parameter to the metod, so that even different instance is excuting the same method in the class, it won't mess up.
Classes in dynamic languages like Python or Ruby is implemented in dictionary instead of C structs in C++ or Java, So fields of instances don't need any pre-declaration slots in class to put it in. And we can even add or subtract fields to an object at runtime like c.stupid = 250
MetaClass in Python: dynamicly building classes that are different based on some condition
Edit code
Live programming
has just executed

```python
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)
d = Counter(10, 15)
```

    -



```
wshh08@wshh08-PC:Cpython_internals$ python -m dis count.py
  1           0 LOAD_CONST               0 ('Counter')
              3 LOAD_CONST               5 (())
              6 LOAD_CONST               1 (<code object Counter at 0x7fa3a8702830, file "count.py", line 1>)
              9 MAKE_FUNCTION            0
             12 CALL_FUNCTION            0
             15 BUILD_CLASS
             16 STORE_NAME               0 (Counter)

 16          19 LOAD_NAME                0 (Counter)
             22 LOAD_CONST               2 (5)
             25 LOAD_CONST               3 (7)
             28 CALL_FUNCTION            2
             31 STORE_NAME               1 (c)
             34 LOAD_CONST               4 (None)
             37 RETURN_VALUE
wshh08@wshh08-PC:Cpython_internals$
```

count.py (~/Extra/Program/python/Cpython_internals) - VIM

```
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)
```

```
NORMAL  count.py                    unix  utf-8
```

类名入栈
基类名入栈
根据加载到栈中的Code Object生成一个方法名字典
Call function to build the method dictionary
load code object inside the class body
Get the first 3 elements in stack and make a class object

3. Source Code
    - line 1936-1946 in **Python/ceval.c**: about the instruction BUILD_CLASS

```c
case BUILD_CLASS:
    /* get the method dictionary */
    u = TOP();
    /* get the tuple of the names of the base classes */
    v = SECOND();
    /* get the class name */
    w = THIRD();
    STACKADJ(-2);
    /* build the class object */
    x = build_class(u, v, w);
    /* push the new class object into the stack */
    SET_TOP(x);
    Py_DECREF(u);
    Py_DECREF(v);
    Py_DECREF(w);
    break;
```

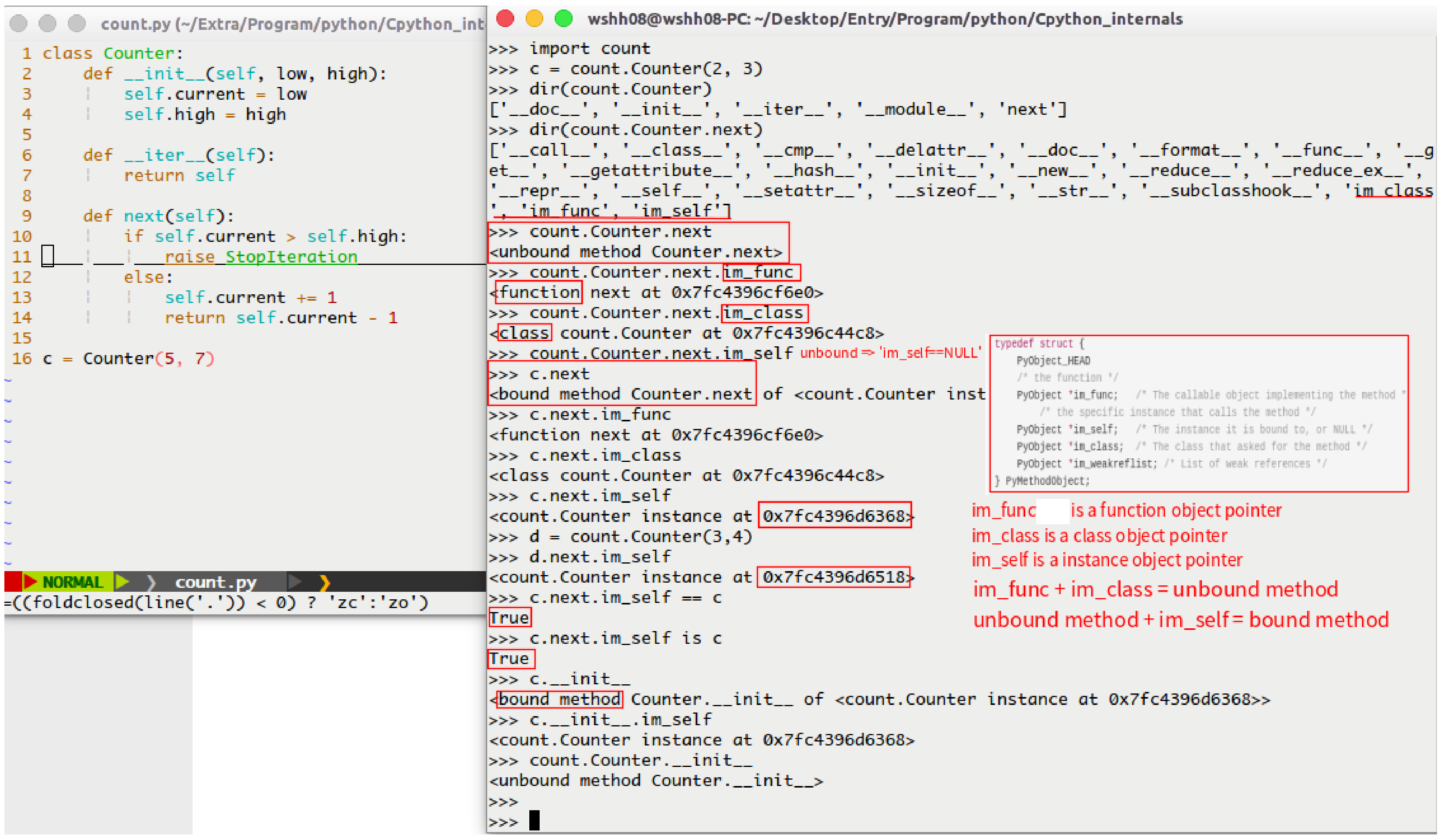
    - line 4618-4670 in **Python/ceval.c**: definition of function `build_class(..)`

```c
static PyObject *
build_class(PyObject *methods, PyObject *bases, PyObject *name)
{
    ...
    /* Just image metaclass as a function that
     * take name, bases, and methods to build a
     * class object. Interpreter will call
     * a default metaclass to create a class
     * without a specific metaclass.
     */
    result = PyObject_CallFunctionObjArgs(metaclass, name, bases, methods, NULL);
    ...
    return result
}
```

    - line 12-37 in **Include/classobject.h**: Data struct associated with Class Object and Instance of Class.

```c
/* Class Object */
typedef struct {
    PyObject_HEAD
    PyObject    *cl_bases;      /* A tuple of class objects */
    PyObject    *cl_dict;       /* A dictionary */
    PyObject    *cl_name;       /* A string */
    /* The following three are functions or NULL */
    PyObject    *cl_getattr;
    PyObject    *cl_setattr;
    PyObject    *cl_delattr;
    PyObject    *cl_weakreflist; /* List of weak references */
} PyClassObject;

/* Instance Object: instance doesn't store any methods itself */
typedef struct {
    PyObject_HEAD
    /* Pointer to class of the instance to get methods(only stored in class) */
    PyClassObject *in_class;    /* The class object */
    /* values of fields in the instance */
    PyObject      *in_dict;     /* A dictionary */
    PyObject      *in_weakreflist; /* List of weak references */
} PyInstanceObject;

/* Method Object, Method is just a function wrapped with
 * a 'self' pointer and a 'class' pointer
 */
typedef struct {
    PyObject_HEAD
    /* the function */
    PyObject *im_func;   /* The callable object implementing the method */
        /* the specific instance that calls the method */
    PyObject *im_self;   /* The instance it is bound to, or NULL */
    PyObject *im_class;  /* The class that asked for the method */
    PyObject *im_weakreflist; /* List of weak references */
} PyMethodObject;
```

- Relation between Method and Function



count.py (~/Extra/Program/python/Cpython_int

```python
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)
```

```
=((foldclosed(line('.')) < 0) ? 'zc':'zo')
```

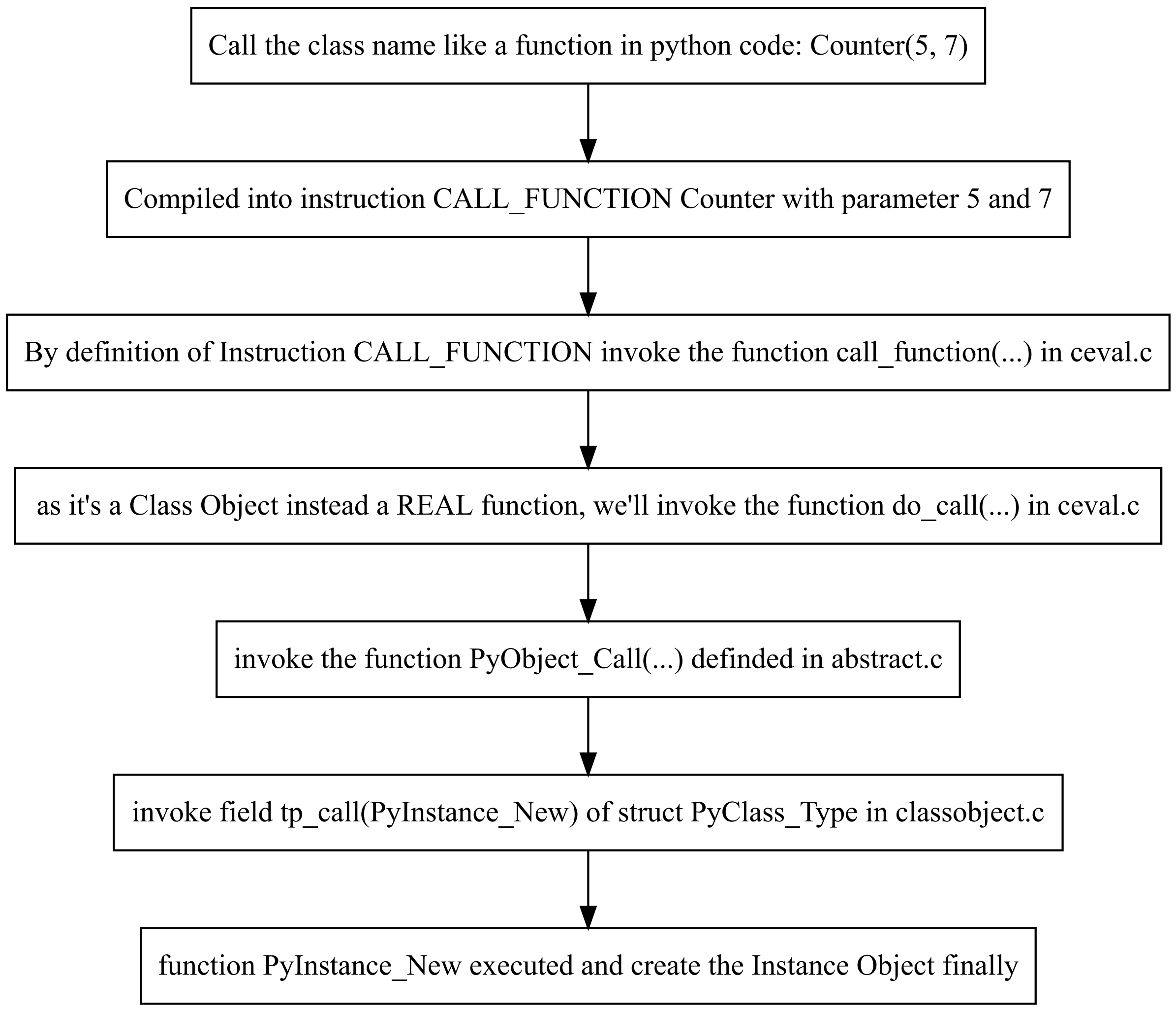
wshh08@wshh08-PC: ~/Desktop/Entry/Program/python/Cpython_internals

```
>>> import count
>>> c = count.Counter(2, 3)
>>> dir(count.Counter)
['__doc__', '__init__', '__iter__', '__module__', 'next']
>>> dir(count.Counter.next)
['__call__', '__class__', '__cmp__', '__delattr__', '__doc__', '__format__', '__func__', '__g
et__', '__getattribute__', '__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__self__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'im_class
', 'im_func', 'im_self']
>>> count.Counter.next
<unbound method Counter.next>
>>> count.Counter.next.im_func
<function next at 0x7fc4396cf6e0>
>>> count.Counter.next.im_class
<class count.Counter at 0x7fc4396c44c8>
>>> count.Counter.next.im_self unbound => 'im_self==NULL'
>>> c.next
<bound method Counter.next of <count.Counter inst
>>> c.next.im_func
<function next at 0x7fc4396cf6e0>
>>> c.next.im_class
<class count.Counter at 0x7fc4396c44c8>
>>> c.next.im_self
<count.Counter instance at 0x7fc4396d6368>
>>> d = count.Counter(3,4)
>>> d.next.im_self
<count.Counter instance at 0x7fc4396d6518>
>>> c.next.im_self == c
True
>>> c.next.im_self is c
True
>>> c.__init__
<bound method Counter.__init__ of <count.Counter instance at 0x7fc4396d6368>>
>>> c.__init__.im_self
<count.Counter instance at 0x7fc4396d6368>
>>> count.Counter.__init__
<unbound method Counter.__init__>
>>>
>>>
```

```c
typedef struct {
    PyObject_HEAD
    /* the function */
    PyObject *im_func;   /* The callable object implementing the method *
        /* the specific instance that calls the method */
    PyObject *im_self;   /* The instance it is bound to, or NULL */
    PyObject *im_class;  /* The class that asked for the method */
    PyObject *im_weakreflist; /* List of weak references */
} PyMethodObject;
```

im_func is a function object pointer
im_class is a class object pointer
im_self is a instance object pointer
im_func + im_class = unbound method
unbound method + im_self = bound method

- Every Instance stores a pointer to its class to get methods(metods are only stored in the class object) When a instance call a method it will transfer itself as the 'self' parameter to the metod, so that even different instance is excuting the same method in the class, it won't mess up.
- Process of creating instance of a class:


Call the class name like a function in python code: Counter(5, 7)
Compiled into instruction CALL_FUNCTION Counter with parameter 5 and 7
By definition of Instruction CALL_FUNCTION invoke the function call_function(...) in ceval.c
as it's a Class Object instead a REAL function, we'll invoke the function do_call(...) in ceval.c
invoke the function PyObject_Call(...) definded in abstract.c
invoke field tp_call(PyInstance_New) of struct PyClass_Type in classobject.c
function PyInstance_New executed and create the Instance Object finally

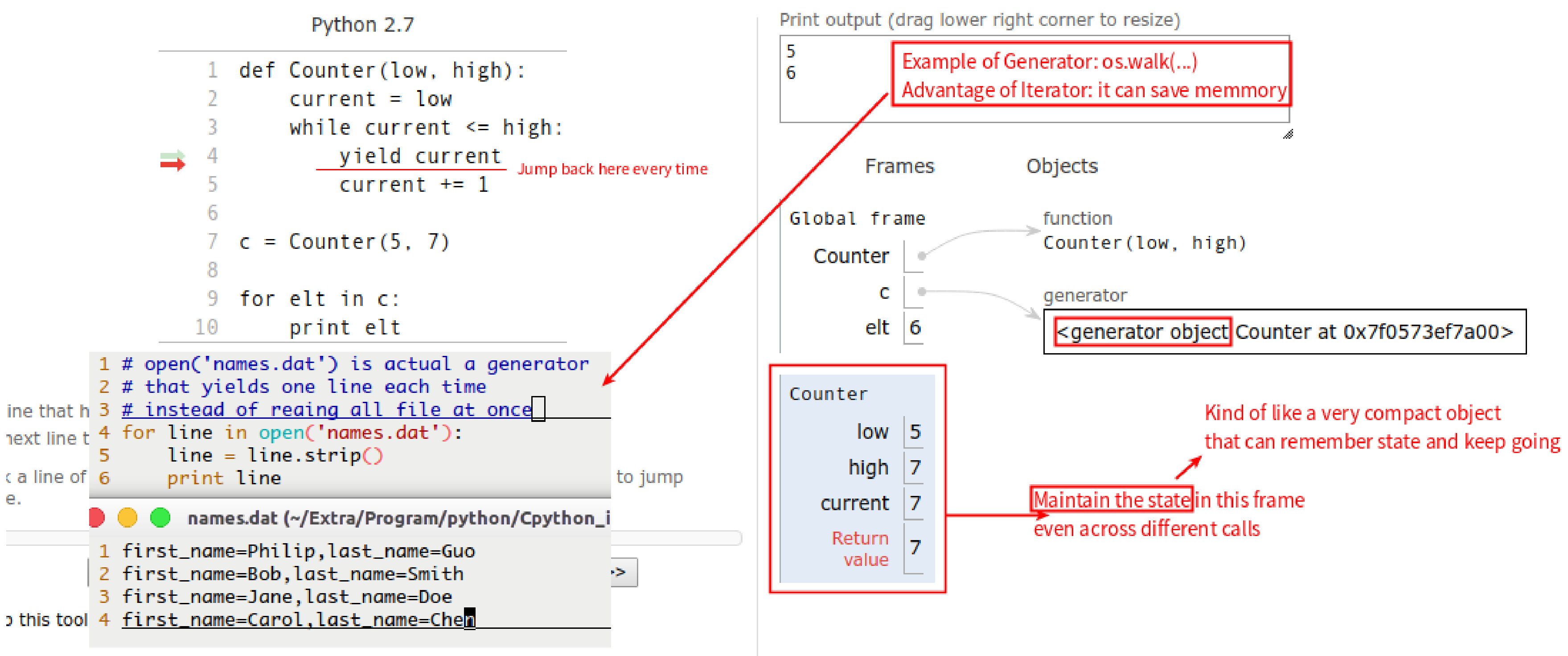
- Source Code associated with creation of Insatance

```
/* line 444-484 in Objects/classobject.c:
 * definition of PyClass_Type - Type Object of the class object
 */
PyTypeObject PyClass_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "classobj",
    sizeof(PyClassObject),
        ...
    /* called by Instruction CALL_FUNCTION
     * following the process described above
     * when instantiaze a Class
     */
    PyInstance_New,                             /* tp_call */
        ...
};
/* line 549-598 in Objects/classobject.c:
 * implementation of function PyInsatnce_NEW
 */
PyObject *
PyInstance_New(PyObject *klass, PyObject *arg, PyObject *kw)
{
    register PyInstanceObject *inst;
    PyObject *init;
    static PyObject *initstr;

    if (initstr == NULL) {
        initstr = PyString_InternFromString("__init__");
        if (initstr == NULL)
            return NULL;
    }
    /* create a unintialized Instance first */
    inst = (PyInstanceObject *) PyInstance_NewRaw(klass, NULL);
    /* get the attribute 'init' from the instance*/
    init = instance_getattr2(inst, initstr);
    if (init == NULL) {
            ...
            return null
        } else {
        /* Apply Method Object 'init' on the new instance to initialize it
         * with arguments given by Customer code, like Counter(5, 7)
         *
         * As 'init' is a Method Object and keeps a pointer(im_self) to the
         * instance('inst') itself, function PyEval_CallObjectWithKeyword
         * doesn't need get 'inst' as a parameter to modify the 'inst'
         *
         * 'res' is return value of the __init__() method, should be None
         */
        PyObject *res = PyEval_CallObjectWithKeywords(init, arg, kw);
        ...
        }
    return (PyObject *)inst;
}
```

Lecture 9 - Generators

1. Objective: To see how generators are a more general kind of iterator
    - recap of the Counter iterator class
    - Counter re-implemented as a generator
    - Diving into how CPython implements generators
2. Concepts about generator

◦



Python 2.7

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

c = Counter(5, 7)

for elt in c:
    print elt
```

```python
# open('names.dat') is actual a generator
# that yields one line each time
# instead of reaing all file at once
for line in open('names.dat'):
    line = line.strip()
    print line
```

names.dat (~/Extra/Program/python/Cpython_i

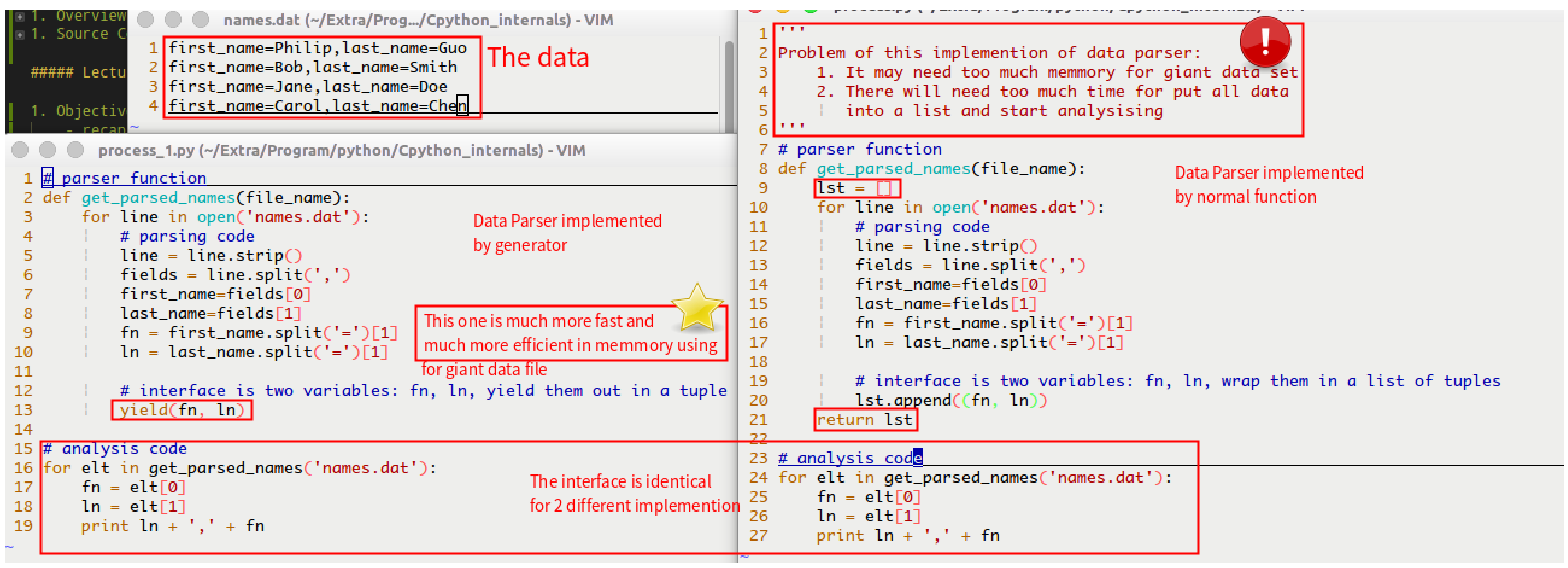
```
first_name=Philip,last_name=Guo
first_name=Bob,last_name=Smith
first_name=Jane,last_name=Doe
first_name=Carol,last_name=Chen
```

Print output (drag lower right corner to resize)

```
5
6
```

Jump back here every time
Example of Generator: os.walk(...)
Advantage of Iterator: it can save memmory
Frames
Objects
Global frame
Counter
c
elt 6
function
Counter(low, high)
generator
<generator object Counter at 0x7f0573ef7a00>
Counter
low 5
high 7
current 7
Return value 7
Kind of like a very compact object
that can remember state and keep going
Maintain the state in this frame
even across different calls
ine that h
next line t
a line of
e.
this tool
to jump

◦



names.dat (~/Extra/Prog.../Cpython_internals) - VIM

```
first_name=Philip,last_name=Guo
first_name=Bob,last_name=Smith
first_name=Jane,last_name=Doe
first_name=Carol,last_name=Chen
```

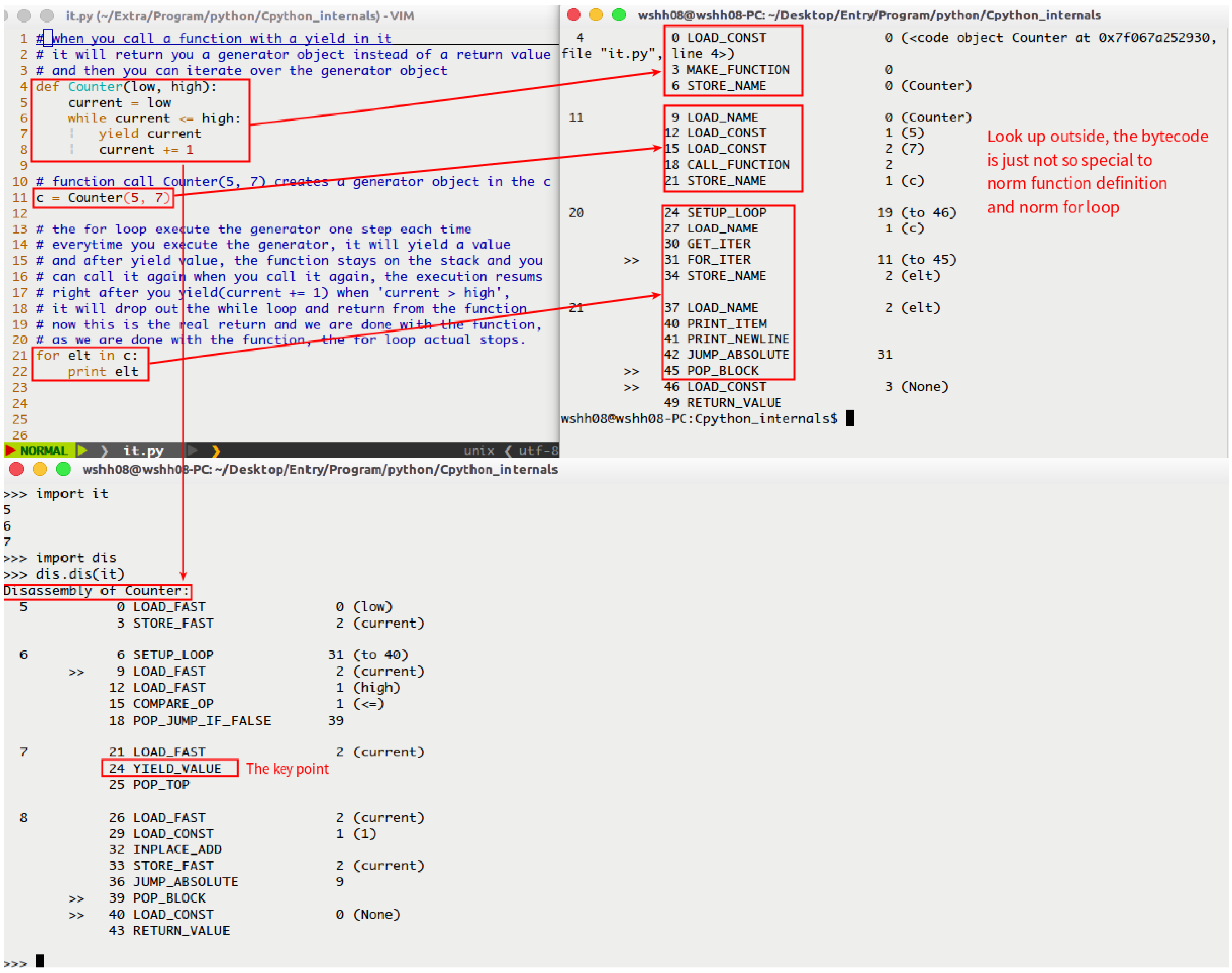
process_1.py (~/Extra/Program/python/Cpython_internals) - VIM

```python
# parser function
def get_parsed_names(file_name):
    for line in open('names.dat'):
        # parsing code
        line = line.strip()
        fields = line.split(',')
        first_name=fields[0]
        last_name=fields[1]
        fn = first_name.split('=')[1]
        ln = last_name.split('=')[1]

        # interface is two variables: fn, ln, yield them out in a tuple
        yield(fn, ln)

# analysis code
for elt in get_parsed_names('names.dat'):
    fn = elt[0]
    ln = elt[1]
    print ln + ',' + fn
```

```python
'''
Problem of this implemention of data parser:
    1. It may need too much memmory for giant data set
    2. There will need too much time for put all data
       into a list and start analysising
'''
# parser function
def get_parsed_names(file_name):
    lst = []
    for line in open('names.dat'):
        # parsing code
        line = line.strip()
        fields = line.split(',')
        first_name=fields[0]
        last_name=fields[1]
        fn = first_name.split('=')[1]
        ln = last_name.split('=')[1]

        # interface is two variables: fn, ln, wrap them in a list of tuples
        lst.append((fn, ln))
    return lst

# analysis code
for elt in get_parsed_names('names.dat'):
    fn = elt[0]
    ln = elt[1]
    print ln + ',' + fn
```

3. Bytecode about generator

◦


it.py (~/Extra/Program/python/Cpython_internals) - VIM
wshh08@wshh08-PC: ~/Desktop/Entry/Program/python/Cpython_internals
NORMAL
it.py
unix
utf-8
wshh08@wshh08-PC: ~/Desktop/Entry/Program/python/Cpython_internals

```python
# when you call a function with a yield in it
# it will return you a generator object instead of a return value
# and then you can iterate over the generator object
def Counter(low, high):
    current = low
    while current <= high:
        yield current
        current += 1

# function call Counter(5, 7) creates a generator object in the c
c = Counter(5, 7)

# the for loop execute the generator one step each time
# everytime you execute the generator, it will yield a value
# and after yield value, the function stays on the stack and you
# can call it again when you call it again, the execution resumes
# right after you yield(current += 1) when 'current > high',
# it will drop out the while loop and return from the function
# now this is the real return and we are done with the function,
# as we are done with the function, the for loop actual stops.
for elt in c:
    print elt
```

```
  4           0 LOAD_CONST               0 (<code object Counter at 0x7f067a252930,
file "it.py", line 4>)
              3 MAKE_FUNCTION            0
              6 STORE_NAME               0 (Counter)

 11           9 LOAD_NAME                0 (Counter)
             12 LOAD_CONST               1 (5)
             15 LOAD_CONST               2 (7)
             18 CALL_FUNCTION            2
             21 STORE_NAME               1 (c)

 20          24 SETUP_LOOP              19 (to 46)
             27 LOAD_NAME                1 (c)
             30 GET_ITER
        >>   31 FOR_ITER                11 (to 45)
             34 STORE_NAME               2 (elt)

 21          37 LOAD_NAME                2 (elt)
             40 PRINT_ITEM
             41 PRINT_NEWLINE
             42 JUMP_ABSOLUTE           31
        >>   45 POP_BLOCK
        >>   46 LOAD_CONST               3 (None)
             49 RETURN_VALUE
wshh08@wshh08-PC:Cpython_internals$
```

Look up outside, the bytecode is just not so special to norm function definition and norm for loop

```
>>> import it
5
6
7
>>> import dis
>>> dis.dis(it)
Disassembly of Counter:
  5           0 LOAD_FAST                0 (low)
              3 STORE_FAST               2 (current)

  6           6 SETUP_LOOP              31 (to 40)
        >>    9 LOAD_FAST                2 (current)
             12 LOAD_FAST                1 (high)
             15 COMPARE_OP               1 (<=)
             18 POP_JUMP_IF_FALSE       39

  7          21 LOAD_FAST                2 (current)
             24 YIELD_VALUE
             25 POP_TOP

  8          26 LOAD_FAST                2 (current)
             29 LOAD_CONST               1 (1)
             32 INPLACE_ADD
             33 STORE_FAST               2 (current)
             36 JUMP_ABSOLUTE            9
        >>   39 POP_BLOCK
        >>   40 LOAD_CONST               0 (None)
             43 RETURN_VALUE

>>>
```

The key point

4. Diving into the source about Generator Object and instruction `YIELD_VALUE` (NOT VERY CLEAR!!)
    - line 12-27 in **Include/gebobject.h**: definition of the Generator Object

```c
typedef struct {
    PyObject_HEAD
    /* The gi_ prefix is intended to remind of generator-iterator. */

    /* Note: gi_frame can be NULL if the generator is "finished"
     * The frame will be kept in scope until the generator is "finished"
     * to maintain the local state, the frame will be poped out the frame stack
     * after yield, but it's still alive and the generator keeps track on it
     */
    struct _frame *gi_frame;

    /* True if generator is being executed. */
    int gi_running;

    /* The code object backing the generator */
    PyObject *gi_code;

    /* List of weak reference. */
    PyObject *gi_weakreflist;
} PyGenObject;
```

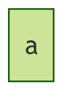
    - line 331-380 in **Objects/genobject.c**: definition of Type Object of Generator Object, the function `gen_iternext(PyGenObject *gen)` will be called while iterating on a Generator Object to get the next value

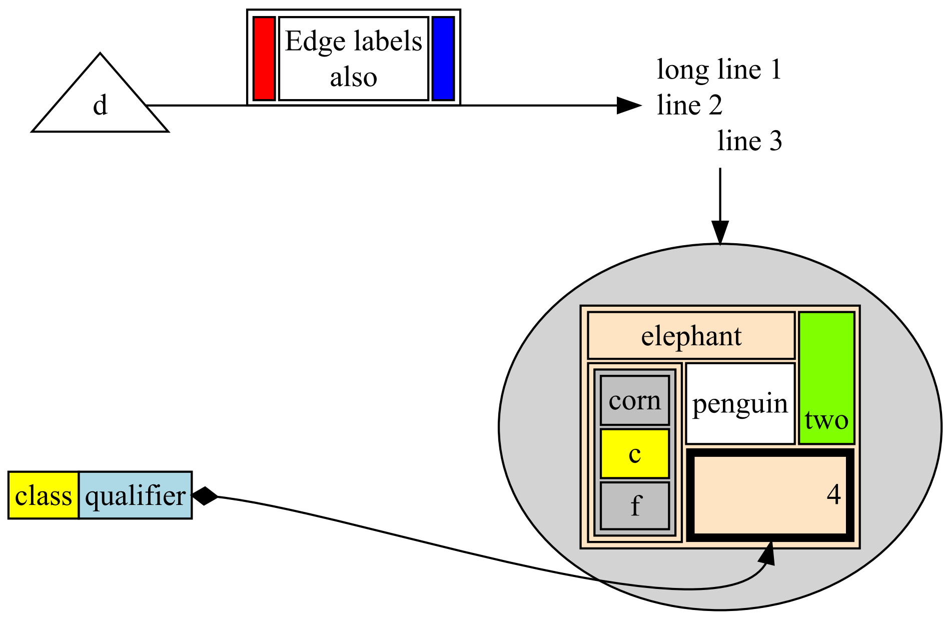
```c
PyTypeObject PyGen_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "generator",                                /* tp_name */
    ...
    /* return the generator object itself as a iterator object? */
    PyObject_SelfIter,                          /* tp_iter */
    /* called by the FOR_ITER instruction while iterating on a generator object */
    (iternextfunc)gen_iternext,                 /* tp_iternext */
    ...
}
/* line 280-284 in Objects/genobject.h: implemention of gen_iternext(...) */
static PyObject *
gen_iternext(PyGenObject *gen)
{
    return gen_send_ex(gen, NULL, 0);
}
/* line 44-113 in Objects/genobject.h: implemention of gen_send_ex(...) */
static PyObject *
gen_send_ex(PyGenObject *gen, PyObject *arg, int exc)
{
    PyThreadState *tstate = PyThreadState_GET();
    /* Grab the frame out from the generator object */
    PyFrameObject *f = gen->gi_frame;
    PyObject *result;

    if (gen->gi_running) {
        PyErr_SetString(PyExc_ValueError,
                        "generator already executing");
        return NULL;
    }
    if (f==NULL || f->f_stacktop == NULL) {
        /* Only set exception if called from send() */
        if (arg && !exc)
            PyErr_SetNone(PyExc_StopIteration);
        return NULL;
    }

    if (f->f_lasti == -1) {
        if (arg && arg != Py_None) {
            PyErr_SetString(PyExc_TypeError,
                            "can't send non-None value to a "
                            "just-started generator");
            return NULL;
        }
    } else {
        /* Push arg onto the frame's value stack */
        result = arg ? arg : Py_None;
        Py_INCREF(result);
        *(f->f_stacktop++) = result;
    }

    /* Generators always return to their most recent caller, not
     * necessarily their creator. */
    f->f_tstate = tstate;
    Py_XINCREF(tstate->frame);
    assert(f->f_back == NULL);
    f->f_back = tstate->frame;

    gen->gi_running = 1;
    /* !!Go back to the interpreter main loop
     * to execute the code in the generator
     */
    result = PyEval_EvalFrameEx(f, exc);
    gen->gi_running = 0;

    /* Don't keep the reference to f_back any longer than necessary.  It
     * may keep a chain of frames alive or it could create a reference
     * cycle. */
    assert(f->f_back == tstate->frame);
    Py_CLEAR(f->f_back);
    /* Clear the borrowed reference to the thread state */
    f->f_tstate = NULL;

    /* If the generator just returned (as opposed to yielding), signal
     * that the generator is exhausted. */
    if (result == Py_None && f->f_stacktop == NULL) {
        Py_DECREF(result);
        result = NULL;
        /* Set exception if not called by gen_iternext() */
        if (arg)
            PyErr_SetNone(PyExc_StopIteration);
    }

    if (!result || f->f_stacktop == NULL) {
        /* generator can't be rerun, so release the frame */
        Py_DECREF(f);
        gen->gi_frame = NULL;
    }

    return result;
}
```

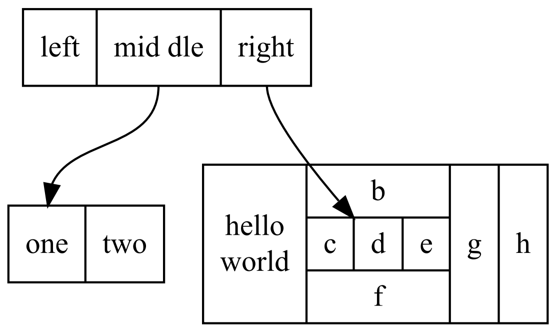
- line 1881-1885 and line 2975-3021 in **Python/ceval.c**: `YIELD_VALUE` - the only special instruction of a generator from a regular function

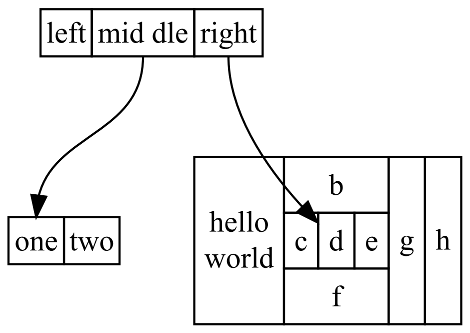
```
    case YIELD_VALUE:
        /* pop up the return value from the value stack */
        retval = POP();
        f->f_stacktop = stack_pointer;
        why = WHY_YIELD;
        goto fast_yield;
        ...
        ...
    fast_yield:
        ...
        if (tstate->frame->f_exc_type != NULL)
            reset_exc_info(tstate);
        else {
            assert(tstate->frame->f_exc_value == NULL);
            assert(tstate->frame->f_exc_traceback == NULL);
        }
    /* pop frame */
    exit_eval_frame:
        Py_LeaveRecursiveCall();
        tstate->frame = f->f_back;

        return retval;
    }
```

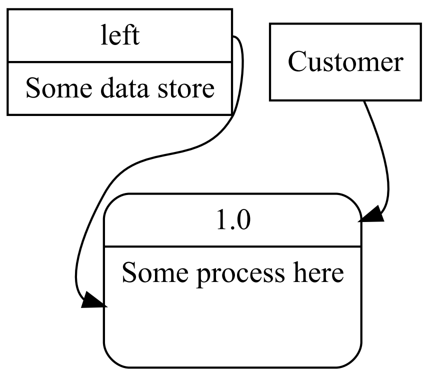
没有这个↓↓↓在markdown_preview_enhanced中预览前面的mermaid图都会乱(但是在Chrome中打开后显示正常)，不知为何











VIM 快捷键：
首先，可以在命令模式下输入v进入自由选取模式，选择需要剪切的文字后，按下d就可以进行剪切了。
其他命令模式下剪切命令：
dd：剪切当前行

ndd：n表示大于1的数字，剪切n行
dw：从光标处剪切至一个单子/单词的末尾，包括空格
de：从光标处剪切至一个单子/单词的末尾，不包括空格
d$：从当前光标剪切到行末
d0：从当前光标位置（不包括光标位置）剪切之行首
d3l：从光标位置（包括光标位置）向右剪切3个字符
d5G：将当前行（包括当前行）至第5行（不包括它）剪切
d3B：从当前光标位置（不包括光标位置）反向剪切3个单词
dH：剪切从当前行至所显示屏幕顶行的全部行
dM：剪切从当前行至命令M所指定行的全部行
dL：剪切从当前行至所显示屏幕底的全部行

yy：复制当前行
nyy：n表示大于1的数字，复制n行
yw：从光标处复制至一个单子/单词的末尾，包括空格
ye：从光标处复制至一个单子/单词的末尾，不包括空格
y$：从当前光标复制到行末
y0：从当前光标位置（不包括光标位置）复制之行首
y3l：从光标位置（包括光标位置）向右复制3个字符
y5G：将当前行（包括当前行）至第5行（不包括它）复制
y3B：从当前光标位置（不包括光标位置）反向复制3个单词