Python (2.5) Virtual Machine

A guided tour

Peter Tröger

April 2008

Python

- Started Dec 1989 by Guido van Rossum
  - First public release Feb 1991 on USENET
  - Intended as scripting language for Amoeba
  - Benevolent Dictator For Life (BDFL)
- Google, Zope, Plone, Pixar, Mailman, BitTorrent, Yahoo, NASA, …
- Python Enhancement Proposal (PEP)
- Free software, ”Pascal of the 2000s”

Benefits

- Combination of good ideas from other languages – C, C++, Modula, ABC, Icon, …
- Interpreter approach, no explicit typing, memory management, first-class object model, portability of source and byte code, in-build module documentation
- ”Batteries included”
  - Platform-neutral system calls, built-in standard library (network, files, GUI, databases, …)
- Interactive mode vs. script mode

Python Language

- Meanwhile focus on stability of language
- EVERYTHING is an object
  - Variables, constants, types, functions, …
- EVERYTHING has a namespace
- EVERYTHING can be printed and introspected
- Dynamic typing, static scoping
- Powerful built-in data types: strings, numerical types, lists, dictionaries, boolean, …
- Specialties: `>>>, self, global, ….`

Python Language - Structure

- Definitions and statements can be combined to an importable *module*
  - Dotted module names for *package* concept
  - Import test / From test import x / From test import * / Import test as theTest
- Indentation to show block structure
- Classes
  - `def [methodName] (self, …):`
  - Constructor denoted by `__init__` method
  - `global` statement

Python Language

- Control structures (`if, while, ...`)
  - Collection iterator (`for`)
  - `try: … / except: …` statements
- No switch statement
  - Instead use of `if…elif…elif…else`
  - See PEP3103 for in-depth discussion

Python Language - Collections

- String, lists, and tuples are *sequences*
- Lists – Dynamic arrays (indexing, sub-ranges)
  - Similar to Pascal arrays
  - Mutable collection of data of the same type
- Tuples – light-weight, immutable lists
  - Similar to Pascal records or C structs
  - Immutable collection of data of different type
- Dictionaries – Associative arrays
  - Keys must be immutable

Python Language - Collections

- Collection iterator `for item in coll:`
  - Dictionary, File and String are iterators
- Slicing
- Nesting
- Single-item tuple

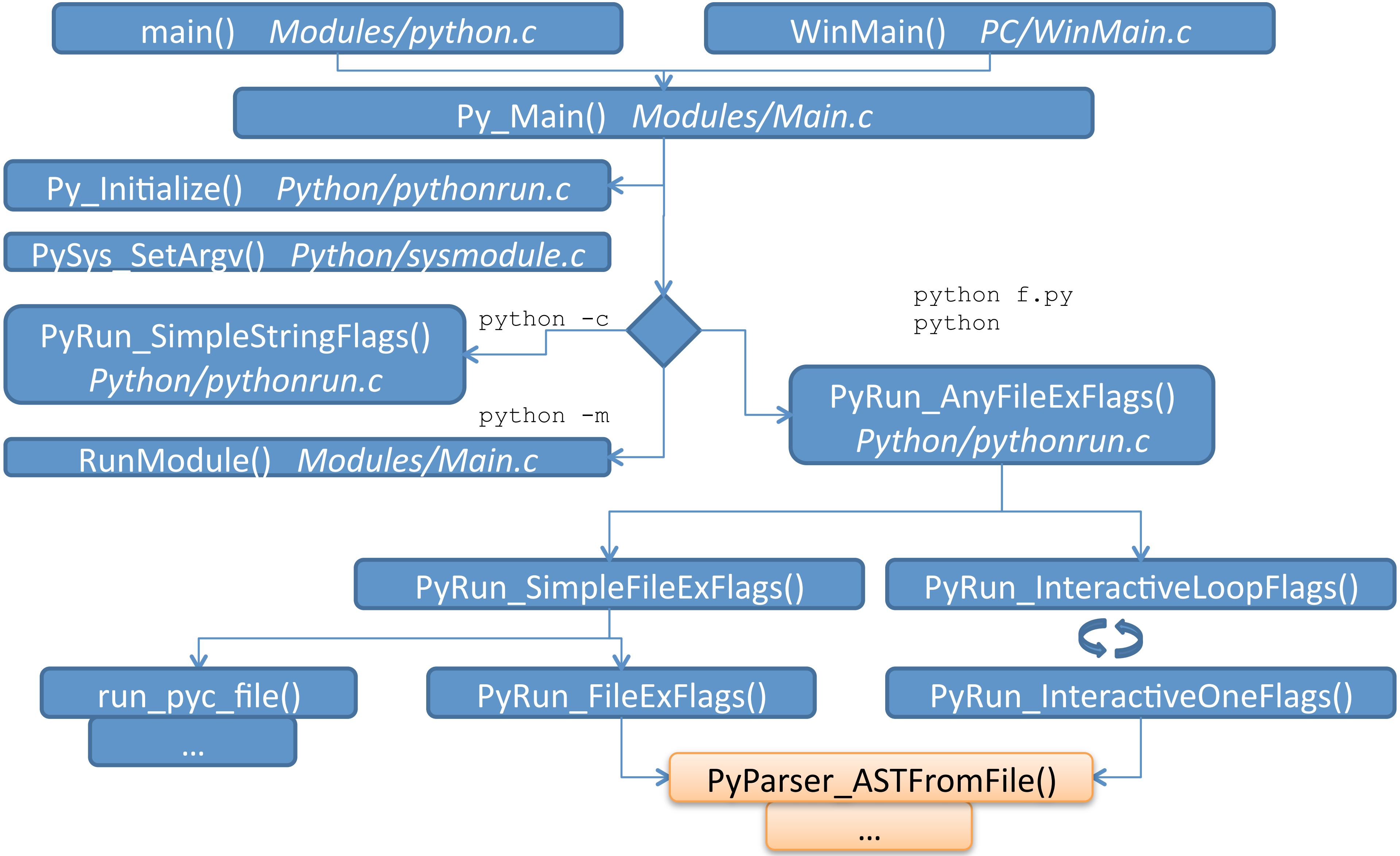
Python Language - Advanced

- Generator function – returns an iterator
  - ```
    def generateItems(seq):
        for item in seq:
            yield 'item: %s' % item
    ```
- List comprehension
- There is some more … (Python 101, books)
    ```

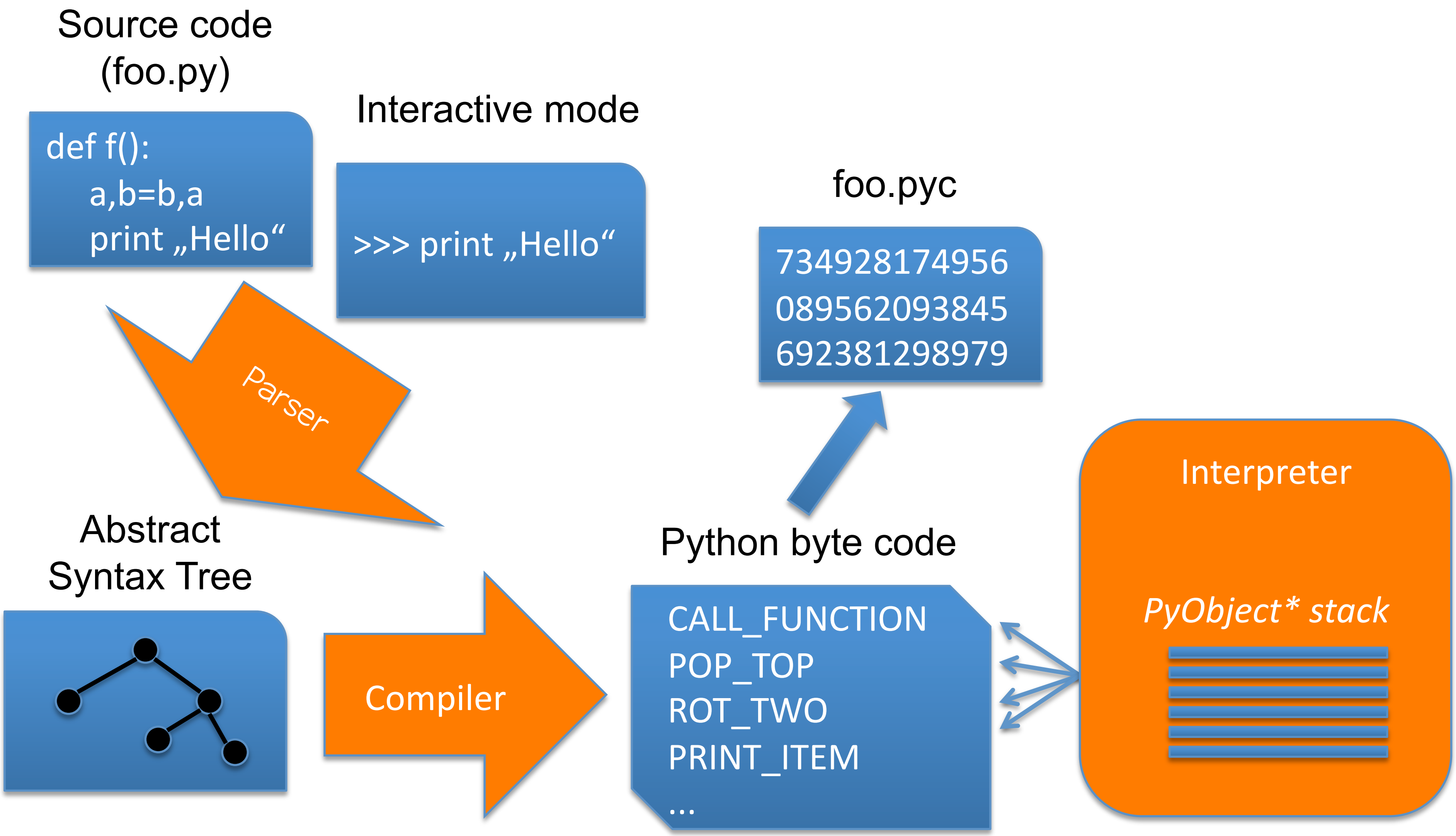
Running Python code

- Interpreter vs. Compiler discussion
- Source code is compiled to some kind of byte code and executed by a virtual machine
  - CPython: Source code -> Python byte code
    - VM written in C, standard implementation
  - Jython: Source code -> Java byte code
    - JVM as runtime environment
  - IronPython: Source code -> IL (".NET") byte code
    - (".NET") CLR as runtime environment
- PyPy: Runtime environment written in Python

Starting CPython



CPython Compiler & Interpreter



CPython 2.5

- Parser transforms the strings of tokens into a concrete syntax tree (ECST)
  - Lexer and LL(1) parser
- Transformer package then translates this into a more genuine AST (`compiler.transformer`)
- Input to byte code compiler in `compile.c`
  - Byte code is generated by parsing this AST
  - Visitor pattern allows hooking in optimizations
- Compilation result for imported code (not top-level file) is stored in `.pyc` file

Python AST (Python.asdl)

- Representation of the program without code
- As usual, every node in the AST represents a syntactic construct
  - Tree root is the `Module` object
  - Statements (If, While, With, Print, Class, Global, Return, Raise, Import, Exec, …)
  - Expressions (BinOp, Dict, Yield, Str, …)
  - Specialized types (excepthandler, alias, slice, …)

Python VM

- Virtual machine part of interpreter executes Python byte code
  - Simple stack machine
  - `PyObject*` stack – byte codes operate on objects
    - Stack frames are allocated on the heap
  - C stack frames point to heap stack frames
- Some high-level byte codes (”PRINT”)
- VM knows nearly nothing about C representation of specific Python types
- Python objects know nothing about the VM

Objects in the Runtime

- Every Python object instance has a corresponding C type instance
  - Even basic types are allocated on the heap
  - All C types contain `PyObject_HEAD` struct members (e.g. see `intobject.h / object.h`)
  - Runtime can therefore treat every Python object as variable of the type `PyObject*`
  - `PyObject_HEAD` contains number of references on the object (`ob->ob_refcnt`)
- C modules need to use `Py_INCREF(obj)` and friends

Objects in the Runtime

- `ob_type` member of `PyObject_HEAD` holds pointer to `PyTypeObject` struct
  - Determines C functions to call on activities with that object – *type methods*
  - Some operations are not feasible for some types

```
PyTypeObject PyInt_Type = {
        PyVarObject_HEAD_INIT(&PyType_Type, 0)
        "int",
        sizeof(PyIntObject),
        0,
        (destructor)int_dealloc,                /* tp_dealloc */
        (printfunc)int_print,                   /* tp_print */
        0,                                      /* tp_getattr */
        0,                                      /* tp_setattr */
        (cmpfunc)int_compare,                   /* tp_compare */
…
```

Interpreter Loop (ceval.c)

- Big switch statement, working on an incoming `PyFrameObject*` structure
  - Many opcode implementations call C helper functions (see `PyEval_EvalFrameEx`)
- For every byte code instruction, management of reference counting needed
- Few byte code prediction mechanism
  - E.g.: *COMPARE_OP* often followed by *JUMP_IF_**
  - Successful interpreter prediction might lead to successful processor branch prediction

Frame Objects

- `PyFrameObject` - argument for interpreter
  - Contains caller frame, code, variables, and return address
  - Executed by the interpreter loop
  - Represents an anonymous *code object in execution*

– Function calls are mapped to frames

– Parameters of function calls are mapped to local variables in a frame

– Functions are a programming language construct, independent from frame concept

Function objects

- Function definition (`def`) binds function name in local namespace to function object
- Function objects are first class objects
- Contains default value arguments
  - Computed at definition time, which is relevant when using mutable objects
- Functions store their body as code object in the *func_code* attribute

Code objects

- Immutable object representing executable byte code
  - *co_name*: Function name
  - *co_nlocals, co_varnames*: Local variables
  - *co_names*: Names used in the byte code
  - *co_freevars*: Names of free variables
  - *co_code, co_consts, co_filename, co_firstlineno, …*
- Contains no reference to mutable objects
- Modules and interactive mode rely on separation of code and functions

Python Byte Code

- Documented in `DIS` module
- Special terminology
  - *TOS*: Top-of-stack item
  - *TOS1*: Second top-most stack item
  - *TOS2*: Third top-most stack item
- Operations normally take top item(s) from stack and put back the result item
  - Basic stack operations are inline C code
- Basic operations on primitive types are usually thread-safe

Basic Stack Operation

- *POP_TOP*: Remove *TOS*
  - Implies decreasing of TOS reference counter
- *ROT_TWO, ROT_THREE, ROT_FOUR*
  - Rotate up the top-most stack items
  - e.g. tuple unpacking : `a,b=b,a`
- *DUP_TOP*: Duplicate TOS reference
- Unary operations on TOS (e.g. *UNARY_POSITIVE*)
- Binary operations on TOS and TOS1 (*BINARY_POWER, _MULTIPLY, _DIVIDE, …)*

Variables

```
x = 1
def f():
    y = x+1
```

- Dictionary for variables in a scope
  - Lookup for ”x” needs to check in local scope, global scope and built-in scope
- Local variables statically decidable by compiler
  - Rule: All assigned variables are local (assignment operator, import)
  - Can be accessed by index, instead of dict lookup

Variables

- *STORE_/LOAD_/DELETE_GLOBAL name*
  - Access to global variables
- *STORE_ / LOAD_ / DELETE_FAST varnum*
  - Fast access by using array of local variables
  - Uses *co_varnames[var_num]* array from code object
- *STORE_ / LOAD_ / DELETE_NAME namei*
  - Used when scope is not determinable (e.g. *import* in function)
  - Code object has *co_names* list with name indices
  - Compilers uses *_FAST / *_GLOBAL whenever possible
- *STORE_ / LOAD_ / DELETE_ATTR nami*
  - Attributes names are also stored in *co_names* list
- *LOAD_ /CONST consti*
  - Relies on *co_consts[consti]* list of constants in code object

In-Place / Slices

- In-place operations are like binary operations, but demand a 'self-mutable' object on TOS1
  - *INPLACE_POWER, INPLACE_MULTIPLY, …*
  - TOS1 then becomes new TOS
  - For operations such as +=, -= (see PEP 203)
- Slice operations
  - *SLICE+0*: TOS=TOS[:]
  - *SLICE+1*: TOS=TOS1[TOS:]
  - *SLICE+2*: TOS=TOS1[:TOS]
  - *SLICE+3*: TOS=TOS2[TOS1:TOS]

Slices / Lists

- Slice assignment
  - Changes TOS / TOS1 / TOS2 in-place
  - *STORE_SLICE+0*: TOS[:]=TOS1
  - *STORE_SLICE+1*: TOS1[TOS:]=TOS2
  - *DELETE_SLICE+*…
  - *STORE_SUBSCR*: TOS1[TOS]=TOS2
  - *DELETE_SUBSCR:* del TOS1[TOS]
- Collection types in Python are cheap and fast !

Printing

- *PRINT_EXPR* : Implements expression statement for interactive mode
- *PRINT_ITEM*: Print TOS to `sys.stdout`
  - *PRINT_ITEM_TO*: Print TOS1 to TOS file-like object
  - Needs to consider Unicode and soft space
- *PRINT_NEWLINE, PRINT_NEWLINE_TO*

Control Structures

- *BREAK_LOOP, CONTINUE_LOOP:*
  Ends the current block
- *RETURN _VALUE:*
  Return with TOS to the caller of a function
- *YIELD_VALUE:*
  Remove TOS and yield it from a generator
- *END_FINALLY:* Interpreter then either re-raises exception, returns from function or continues
- *JUMP_FORWARD delta, JUMP_IF_TRUE delta, JUMP_IF_FALSE delta, JUMP_ABSOLUTE target*

Directly mapped concepts

- *EXEC_STMT*: run Python code
- *LIST_APPEND:* fast list comprehension
- *IMPORT_STAR*: from [module@TOS] import *
- *UNPACK_SEQUENCE count*: Unpack TOS
- *FOR_ITER* : Treat TOS as iterator, call *next()* method, push yielded value on stack
- *RAISE_VARARGS*: Raises an exception, parameters are on stack

Directly mapped concepts

- *BUILD_TUPLE n*
  - Create tuple of n top-most items and push result
- *BUILD_LIST n*
- *BUILD_MAP*
  - Create empty dictionary on the stack
- *BUILD_CLASS :* Creates a new class object
  - TOS: Dictionary of methods
  - TOS1: Tuple of base class names
  - TOS2: Class name

CALL_FUNCTION argc

- *CALL_FUNCTION argc*
  - *argc* low byte contains number of positional parameters
  - *argc* high byte contains number of keyword parameters
  - Parameters itself are on the stack, after them the function object to call
- *MAKE_FUNCTION argc*
  - Pushes new function object, TOS is the code, *argc* default parameters are below TOS
- And a lot more …. (read ceval.c)

Introducing a new byte code

- Extend official list of opcodes (`Include/opcode.h; Lib/opcode.py`)
- Increase MAGIC number (`Python/import.c`)
- Change AST->byte code compiler (`Python/compiler.c`)
- Change byte code interpreter (`Python/ceval.c`)

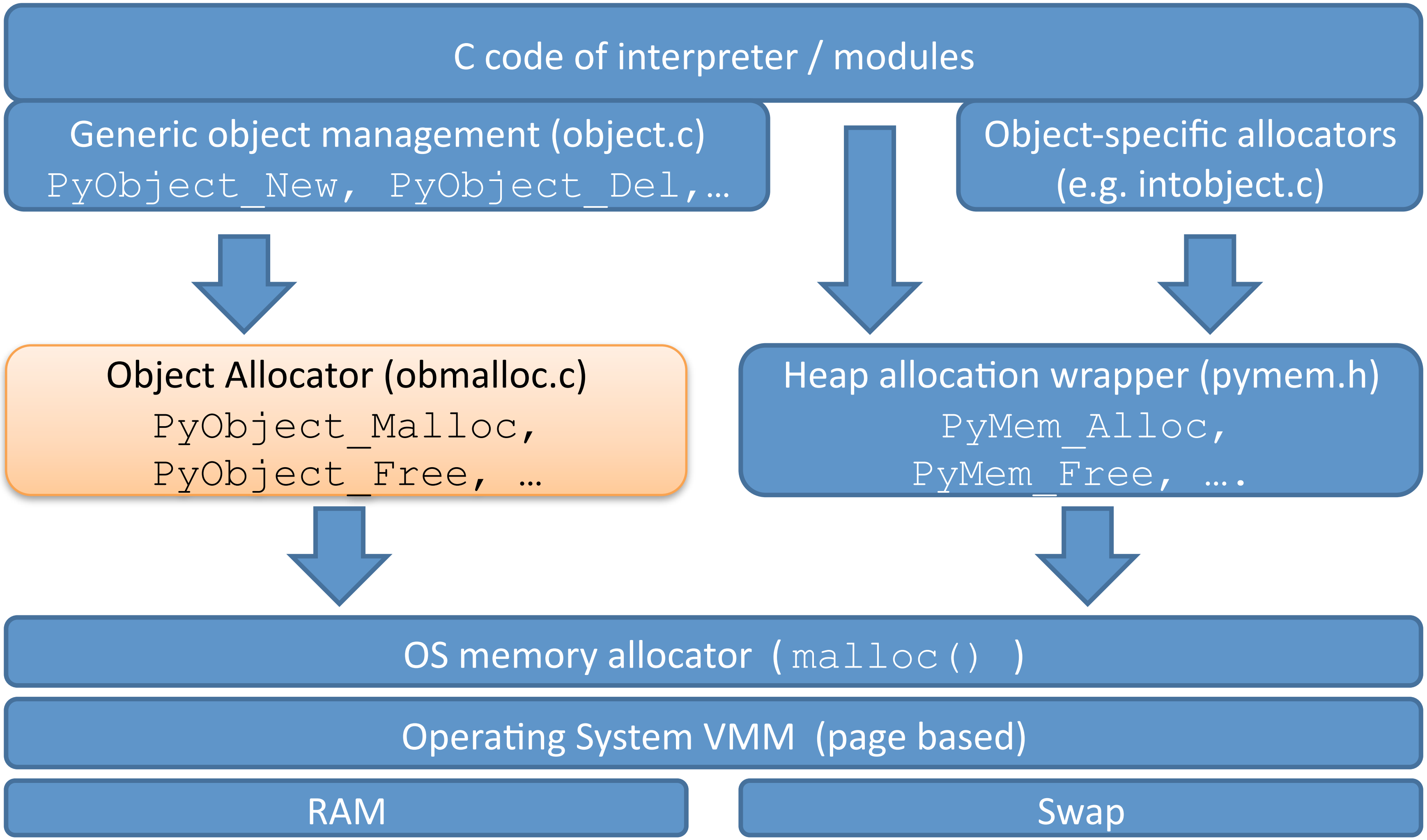
Threading in the VM

- Python threads are true OS threads
- Global interpreter lock
  - Only one (Python) thread is running interpreter code at the same time
  - Regular context switch after a number of executed instructions or with long-running operations
  - Problem with multi-core CPU's
  - Released for long-running C code (e.g. system call)
  - Performance advantage
- Byte code instructions are atomic

Stackless Python

- Every function call creates a C stack frame
  - Subroutines vs. coroutines
- Python functions can act as coroutines or *tasklets*, concurrently executed by scheduler
  - Support for channel communication
  - Act as lightweight threads
- Redesigned interpreter loop to avoid the C stack frame creation on function call

Memory Management



Object Allocator

- By design, many very small allocation requests
  - Everything is an object !
- Special optimization for performance (obmalloc.c)
  - Requests >256 bytes handled by `malloc`
  - Smaller requests sizes are grouped (8 bytes apart)
    - Memory pools of 4k length each (VMM page size), with own free list
    - Pools are used by different request size allocators
    - 8 Byte alignment of returned address

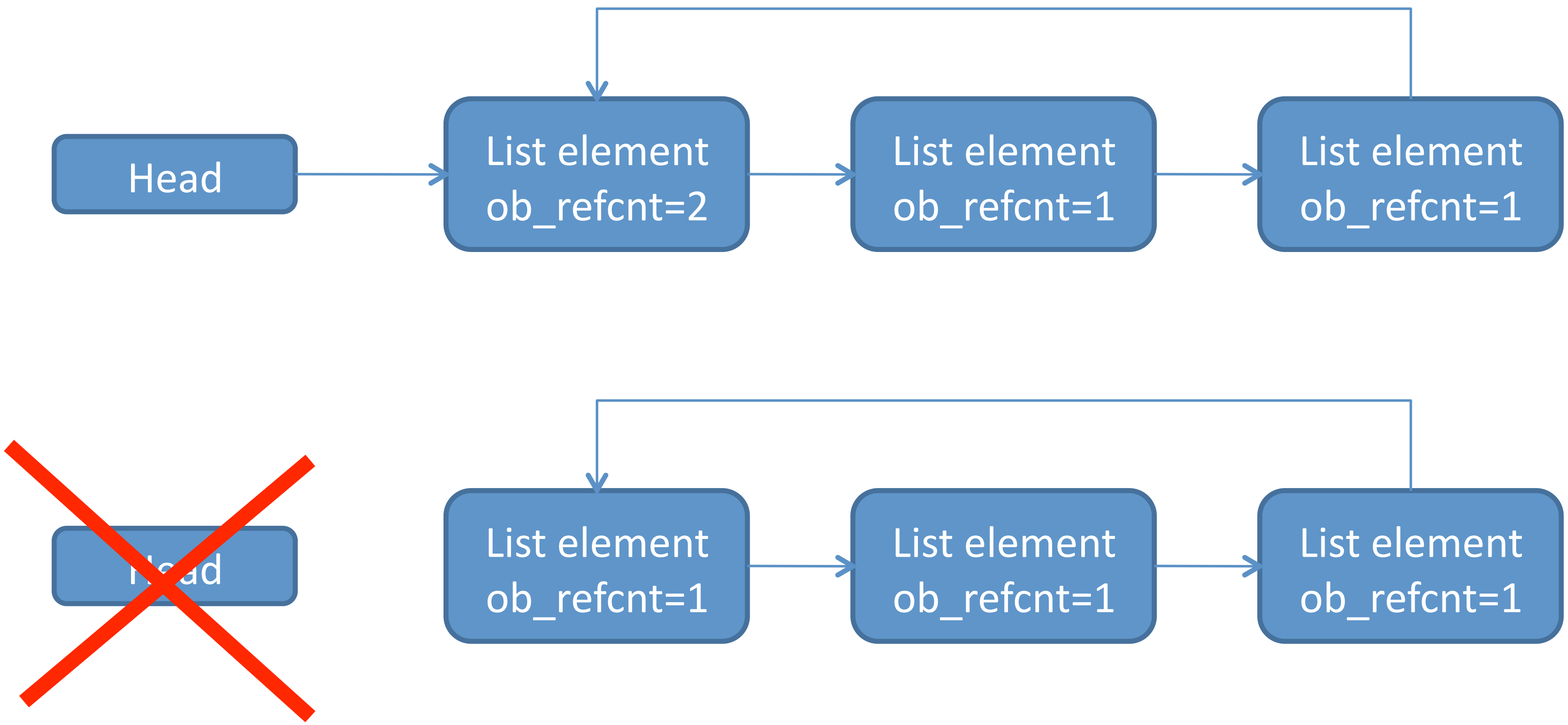
Garbage Collector

- Traditional garbage collection (e.g. mark and sweep) would demand a set of root objects
  - Extension modules can create own Python objects
  - GC for allocated C objects not really portable
  - Traversing all objects is expensive
- Instead: Simple reference counting
  - In `ob->ob_refcnt` from `PyObject_HEAD`
  - Works with every `malloc()` / `free()`
- `Py_DECREF()` - (object.h)
  - Calls finalizer when reference count comes to zero

Garbage Collector

- Functions that create an object set the `ob_refcnt` to 1, and store it - or destroy it by calling `Py_DECREF`
  - Some store functions therefore don't increase the reference counter (e.g. `PyList_SetItem()`)
- Objects can be stucked in tracebacks
- Weakref module (PEP 205)
  - For object caches (weak dicitionaries)
  - For circular references (DOM node relations)

Circular References



Cyclic Garbage Collector

- Reference cycle: Unused object(s) even though reference counter is not zero
  - Test is only relevant for container types
- Usage of double-linked list of all container objects (`gc_next , gc_pref`)
  - Determine all containers which are only referenced by them self
- Objects in cycles with finalizers `__del()__` are added to set of uncollectable objects
  - Order of finalizer calls in the cycle unclear

Sources

- Use the source, Luke (Python SVN trunk, March 2008)
- Mark Lutz, Programming Python. O'Reilly 2006
- http://mail.python.org/pipermail/python-list
- http://www.python.org – PEP's, Python Tutorial,
  Extending and Embedding the Python Interpreter Tutorial, Python FAQ
- http://docs.python.org/lib/bytecodes.html
- http://docs.python.org/api/threads.html
- http://www.voidspace.org.uk/python/articles/code_blocks.shtml
- http://effbot.org/pyref/type-code.htm
- http://www.devshed.com/c/a/Python/How-Python-Runs-Programs/
- http://www.rexx.com/~dkuhlman/python_101/python_101.html
- http://arctrix.com/nas/python/gc/
- http://www.stanford.edu/class/cs242/slides/2006/python-vanRossum.pdf