

Python Implementation Strategies

Jeremy Hylton
Python / Google

Python language basics

- High-level language
 - Untyped but safe
 - First-class functions, classes, objects, &c.
 - Garbage collected
 - Simple module system
 - Lexically scoped
- Std primitive objects: str, num, dict, list, tuple
- Object-orientation
 - Multiple inheritance, single dispatch
 - Metaclass for customization

Introspection in Python

- Nearly every object supports introspection
 - Classes and instances represented as dictionaries
 - Access to local variables via dictionary
 - Modify top-level name bindings in other modules
 - Load bytecode from string
- Originally envisioned for debugging
 - But many projects find ways to abuse
 - Language does not provide encapsulation
 - Makes optimization more challenging

Python implementation

- Standard Python interpreter
 - Classic bytecode interpreter implemented in C
 - Small, straightforward implementation
 - Roughly 230k LOC, a dozen active developers
 - Large, platform-specific subsystems: Mac, Win32
- Jython
 - Python running on the JVM; development stalled
- IronPython
 - Experimental implementation for CLR
- Easy to extend / embed in host environment

Memory management

- All objects are heap allocated
 - Everything is boxed
 - Very high allocation rate
- Reference counting + cyclic trash collection
- Objects consume a lot of memory
 - Doubly-linked list of containers for cycle detection
 - Instances typically used dict to store variables
- Optimizations?
 - Custom small object allocator
 - Type-specific free lists for int, frame

Multithreading

- Uses platform threads
- Only one thread can execute at a time
- Serialized by interpreter mainloop
- Pseudo-preemptive scheduling
 - Switch threads every N bytecodes
 - Release locks around I/O, &c.
- Much code depends on this feature
 - Reference counting

Bytecode interpreter

- Stack-based interpreter, switch stmt dispatch
- Heavyweight opcodes
 - Often correspond to Python source constructs
 - Invoke expensive Python runtime operations
 - Common opcodes take hundreds of cycles
- Examples:
 - `LOAD_ATTR` – fetch attribute by name
 - `LOAD_GLOBAL` – load global variable
 - `BINARY_ADD` – add two objects (untyped)
 - Stack operations – push, pop, rot

Interpreter details

- At entry point
 - periodic checks for interrupts, thread switches
 - check for debugging/profiling hooks
- Decode opcode & arg
 - small set of opcodes branch directly to decode
 - don't call out of mainloop code, on success
- A few opcodes inline decode
 - compare dispatches directly to JUMP on equality
 - saves several unpredictable branches

Example: BINARY_ADD

- pop args off stack
- check for two exact integer args
 - inline addition
 - check for overflow
- check for exact strings
 - call concatenation routine
- fallback to generic PyNumber_Add routine
 - does either arg have a numeric add slot that accomodates both types, can return NotImplemented
 - does either arg have a sequence concat

Function calls

- Python frames are heap allocated
 - But make recursive call to eval for each frame
 - Note: Stackless Python
- Rich function calling conventions
 - Variable args, keyword args, default values
 - Function call code is complex and slow

Calling from C to Python

- Python runtime often calls back to Python
 - Operator overloading, finalizers, metaclasses
 - Delicate to maintain state, ref counts
 - Heavy use of C stack
 - Occasionally hard to propagate errors
- Hard to predict effects
 - GC finalizer can resurrect the object
 - Modify list that is being sorted

IronPython comparisons

- Generally faster than C implementation
 - 40-80% faster on Dhrystone-style benchmark
- Function calls much faster
 - Can use native CLR frames, gives up some introspection
- Non-local name lookup much faster
 - Generates static definition of module
 - Has hooks to support introspection
- Memory management is costly
 - Creates and destroys builtin types much more slowly
 - Microbenchmarks at best 70% of C interpreter

Python futures

- Interpreter issues raised by users
 - Performance
 - Security
- How these are realized by developers
 - It's not much fun to implement Python in C
 - Need better runtime representation for optimization
 - Thread limits a growing liability