Faster CPython Documentation

Release 0.0

Victor Stinner

Mar 17, 2021

Contents

Contents:

CHAPTER 1

CPython Garbage Collection

See also *Tracing GC for CPython*.

1.1 Terminology

- GC: garbage collection, i.e. automatic freeing of unused memory
- GIL: the CPython global interpreter lock (prevents multiple threads from corrupting things).
- RC: a reference counting GC or the reference count for an object
- Tracing: tracing based GC. A tracing GC works by starting from roots, following object references to determine all alive objects.
- Mark and Sweep (M&S): a form of tracing GC. It is one of the older methods and without elaborations doesn't work well for large programs (collection time is O(n) where n is heap size).
- Root: In tracing collection, an object pointer that is followed to determine which objects are alive. A precise (non-conservative) GC needs more help to find the roots.
- Conservative: a GC that does not 100% accurately identify roots and will therefore sometimes not free memory that it could. Needs little or no help to find the roots. Libgc is a conservative GC.
- Precise: a GC is that non-conservative. I.e. it can 100% accurately identify all roots.
- Libgc: The Boehm-Demers-Weiser conservative garbage collector
- Moving: a moving GC is allowed to move objects within the heap. Once moved, all references to that object must be repaired by the GC. Libgc is an example of a non-moving GC.
- Copying: a copying GC is one type of moving GC
- Incremental: an incremental GC can process part of the heap in one step, rather than doing a complete GC pass at once
- Concurrent: a concurrent GC will allow other program (sometimes called mutator) execution while the GC is doing work. Typically some locking would still be involved but the GC is designed to try to maximize concurrency

- Generational: a GC that divides the set of objects into different sets based on their age. Typically there would be about three generations. The youngest generation collected the most often.
- Finalizer: a function or method that runs when an object is being freed by the GC. In Python __del__ methods or weak-reference callbacks. The running of these functions and methods is called "finalization".
- Weakref: weak reference, a reference to an object that does not keep the object alive. If the object is freed, the weakref returns None

1.2 Summary of Discussion on Oct 19, 2020

During the online Python core dev sprint.

There was a fair amount of discussion on how CPython's internal GC might be changed to a tracing collector. General consensus is that we need to support existing C extensions without having them change many lines of code. Something like HPy could be functional but would require them to change all lines of code that interact with Python objects.

A mark & sweep collector as an initial proof-of-concept was suggested by Neil and Pablo. Mark Shannon noted that a pure M & S will be horribly slow for a large heap and you likely need an incremental GC. We believe an incremental collector is possible but it takes more work. A generational collector would be good too but you might need "write barriers".

Thomas Wouters believes the best way to provide a compatible C API is to have proxy objects mirroring the internal objects. The proxy objects would have refcnt fields and the internal objects would not. One advantage of this design is that it could be better if GIL is removed (i.e. for concurrent updates of refcnts). Neil asked if the refcnt can be put in the internal object and the proxy be eliminated. Mark Shannon believes objects having different headers will be a pain.

There was some discussion if Py_INCREF could be changed to mean "create a new root". It seems maybe a full refcnt would be needed to provide a compatible API. Mark observed that if you don't care about concurrency, RC is quite a performant way to do memory management.

Regarding providing a compatible API, Thomas stated:

> I think the fundamental requirement for that approach to work is that extension modules don't poke at internal data of Python objects that are not defined by themselves. I'm not entirely sure that holds true, but it should be the most common case. the difference with PyPy is, of course, that C code that defines and creates PyObjects and that doesn't use the GC API would just keep using the existing objects. They pay the cost of proxying, but only for their own objects.

Also regarding compatibility, Thomas stated:

> numpy is a giant mess in all this, because it's so tightly coupled with CPython implementation details (it copies type structs from builtin types), and it exposes its own C API. Any change is going to break them.

There was an extended discussion with Guido, talking about ways of providing a compatible API and how the "facade" or "proxy" objects would work. There was also discussion about the GIL and how locking must be done to avoid segfaults if it is removed. In the Java world, the JVM memory model provides a solid specification of the memory behavior and so code running with those guarantees is easier to make safe in a multi-threaded context.

1.3 Summary of Discussion on Oct 20, 2020

Neil suggested a "straw-man" model of how the "proxy" objects might be implemented, in order to provide a compatible API for extensions. Inspired by Ruby's fixed size, 40 byte objects: make these fixed sized objects do the job of the proxy objects that Thomas Wouters was proposing in his libgc experiment. I.e. any PyObject pointers you give out to external C-API users would point to these fixed size object slots. The Python tracing GC would not move them. It would mean if you have a PyTuple, you can't get at the tuple elements just by casting the structure and moving a fixed offset. Instead, you would have to follow the pointer to where the array of elements is actually stored. And, the GC is free to move that array. Unlike the Pypy model, CPython would always use those fixed size objects. So the entries in a PyTuple would be object pointers going to the fixed size objects (same as what is returned to external API users). That avoids the API boundary problem Pypy has where it has to allocate proxies for all of the PyList or PyTuple elements, just in case someone looks at them.

1.4 Python GC Discussion, Meet on Oct 22, 2020

1.4.1 Participating

- Neil Schemenauer
- Larry Hastings
- Pablo Galindo Salgado
- Joannah Nanjekye
- Eric Snow
- Lewis Gaul

1.4.2 Discussion summary

- Big questions: What are we trying to do? Why?
- Big goal: Replace internal reference counting GC with tracing collector
- We must continue to support existing C extensions, without major source code changes.
  - Support of old API could be done with "shim layer" and proxy objects, like is done with PyPy cpyext. Might be slower but likely not as slow as cpyext.
  - Supporting extensions that use custom allocators for PyObjects? We don't care if we force them to use CPython's runtime to allocate all PyObjects. Pretty rare for extensions to use custom allocators.
  - Another issue is switching the underlying allocator used by obmalloc (Python runtime). Not sure if you do that with an envvar or you can actually switch while Python is running. Could be hard to support.
  - Can we force extensions to stop using macros, stop looking inside of certain PyObjects, e.g. inside a list, stop using borrowed references?
  - If we get NumPy and Cython to update to a changed API, it will help a lot.
  - NumPy might be very slow to adapt to C API changes, example was removal of some globals done for sub-interpreters. Seems NumPy will take a long time to adapt to that.
- Tracing GC options
  - Moving vs non-moving
  - Conservative?
  - Adopt existing library/toolkit or build our own?
- How to support existing C extensions (proxy?, similar to what PyPy does with cpyext)
- How does CPython internals have to change to support tracing GC? Will it be too disruptive?
- Aside from technical implementation challenge two issues:

    - Some kind of breakage for external programs/extensions is virtually certain. Can we convince the community to back change? Need "carrot" for them to want it, otherwise why would they? Suggested "carrot": removal of the GIL.
    - Larry thinks the GIL removal and the breakage should come together. Otherwise, you break things with only the promise of better things later. Later good things might not come.
- Existing GC libraries, frameworks that might be integrated
    - Eclipse OMR (C++)
    - MMTk (high performance, cutting edge GC, Rust backend, C API)
- Handling finalizers, e.g. calling finalizers in reference cycles (Boehm GC doesn't call them).
    - __del__ is extremely hard to support in the same style as current CPython. Armin Rigo is apparently working on some ideas for PyPy.
    - Likely that GC libraries or toolkits will not provide the "finalizer" hooks that could support CPython's __del__ behavior.
    - Can we just deprecate __del__? Have been telling people for years it is dangerous and not recommended to use.
    - Replace with callback() API like is used for Weakref(). You get alive objects to your function and you can't resurrect things.
- How much code breaks if RC is removed and objects don't die immediately after going out of scope? Could be as much or more than the code broken by deprecating __del__.
- Joannah plans to do some work towards us achieving tracing garbage collection as part of her PHD.

1.5 Summary of Discussion on Oct 22-23, 2020

There was a fair amount of discussion on the challenges of supporting the full behavior of Python's __del__ methods. Since PEP 442, CPython will call finalizers for objects that are part of garbage cycles. For non-cycles they get called because the RC naturally gives a topological sorting and so they get called in the right order. For cycles, there is no topological ordering and so the GC has no idea what order to call them in. I.e. you have a set of garbage objects (ready to call tp_clear on), some of them have finalizers, which finalizer to call first and after you call it, is it safe to proceed with collection?

The libgc library has a feature to topologically sort garbage before calling finalizers. So, good so far. However, if there are cycles, Palbo suggested that libgc will not call the finalizers. I.e. libgc would act like Python before PEP 442 was implemented. Thomas suggested later that libgc can be configured to give a warning and still call the finalizer in that case. If true, it seems libgc gives us finalization that is close to what we need for __del__.

There was some suggestion (not by Larry ;-) of just killing off support for __del__ methods. The general consensus seems to be, __del__ methods are a pain and are often misused, we should not remove them and that somehow a tracing GC can support the CPython behavior we currently have.

Greg asked if taking away __del__ really is worse than having to repeatedly tell users that they're trying to use __del__ wrong (it doesn't do what they thought when they thought)?

Neil speculates that supporting __del__ might require a second GC pass after the finalizers run. If we have a generational collector, maybe we can put things we think are garbage in the youngest generation, run finalizers, then run a collection of the youngest generation. The generational GC mechanisms (e.g. write barriers) would ensure we don't have to re-examine the entire heap for the second pass and so in theory should be similar to the cost of the PEP 442 extra GC pass.

Neil and Thomas discussed the idea of introducing a safer alternative to __del__. If we use an API similar to weakref.finalize(), some "foot guns" are avoided. One issue with __del__ is thatthe finalizer method gets 'self' when really it likely only needs a few properties of self. The straw-man API is something like:

```
class Foo:
    def __init__(self, filename):
        self.fp = gc.with_finalizer(self, open(filename))
```

The with_finalizer() function can use the context manager methods of 'file' to ensure cleanup. This seems a nice approach because the finalization logic is put right where 'fp' is created and assigned. 'with_finalize()' can be read as "if X dies, do context manager cleanup on Y". Perhaps a first step is to write a 3rd party PyPI package that provides this API. Then, we could consider moving it to 'gc' and then finally recommend that users switch to it rather than using '__del__'.

with_finailizer() can be implemented using weakref.finalizer(). However, that might not be the best approach. Neil thinks the weakref mechanism would not be a required implementation approach and there might be a more efficient way. The implementation of weakrefs in CPython is pretty complicated and not so elegant. Also, there is an issue with using the weakref approach: if weakref are part of garbage cycles, the callback is not called. Our argument for that behavior is there is no defined topological sort and so we could claim the weakref died first and therefore the callback should not be called. We would need a different behavior because we still would like the PEP 442 style finalization rules. I.e. your file will still be closed, even if the file and the thing to clean it up are part of a garbage cycle.

There was some discussion of disallowing object resurrection from __del__ methods. It's not so clear how this would work. With refcnts, we might check the refcnt to see if something was revived. With a tracing GC, maybe a write barrier can catch the resurrection. If something is resurrected, then what? You can print a warning or raise an error. However, the GC likely needs to proceed anyhow and that seems to require another GC pass, similar to how PEP 442 is implemented.

Thomas wonders how difficult it would be to restructure code to use 'with_finalizer()' or a similar thing. If the __del__ method calls methods on 'self', there is no easy translation of that. Those methods can access any attribute of 'self'. Neil suggested that while painful, that restructuring does have value in that it forces the programmer to more explicitly decide what attributes of 'self' the finalizer needs to access. An object with a lot of attributes and methods is a bit like a program with lots of global variables. You lose track of what depends on what.

Nathaniel Smith notes that:

> async generator finalization relies heavily on resurrection: the interpreter gives all async generator objects a shim __del__ which calls back to the coroutine runner to schedule atask to run the actual cleanup (also note that every generator that contains a yield inside a try or with implicitly has a __del__, so there are a lot more __del__ implementations out there than most people realize).

The code for the '__del__' is auto generated by it calls into user code by stepping the generator. So, that can call arbitrary user code from within the GC run.

1.5.1 Notes and Collection of References

Libgc (Boehm collector) experiment

Below is the discussion thread for Thomas Wouters "libgc" patch. The thread is quite long and there are a lot of details about how Python's GC might be changed or why we shouldn't try. Tim Peters has a number of interesting comments on the page:

https://discuss.python.org/t/switching-from-refcounting-to-libgc/1641

Status of Thomas's "proof of concept" patch: it hooks into libgc's finalizer machinery so that libgc is calling finaliers. It removes calls to _Py_Dealloc(). The collect() function in gcmodule.c no longer does cyclic GC and instead calls libgc collection.

Neil is planning to port Thomas's branch to be on top of CPython 3.9 or the master branch. A lot of conflicts in merge/rebase because of API renames and header file reorganization (e.g. pycore_*). Thomas is not planning to work on the branch in the near future and doesn't mind either a rebase or a merge to bring it closer to the cpython head. Thomas says the code in his published branch is the most up-to-date work he has.

Using Tracing GC based on existing gcmodule logic (tp_traverse)

Neil and Palbo discussed the idea of changing gcmodule.c to be a tracing (e.g. mark and sweep) collector. Quite a few pieces of a M&S collector are already there, tested and working. Major challenge is to identify "GC roots". These are global C variables, C variables on the stack and CPU registers holding PyObject pointers.

We discussed the idea of collecting GC roots when an extension module is imported. E.g. have a special GC_Head list where you put all of the new objects created while importing. Everything on the list after import is considered a root by the GC. This is a clever idea but after we realized there are a number of problems with it, maybe fatal.

Libgc solves this "finding roots" problem by using its own tricks. E.g. walking C stack and having a heuristic test to find object pointers there. This is called a "conservative" approach since non object pointers (e.g. integers that look like pointers) will keep things alive that should be freed.

The easier step at this point is to use libgc as a prototype tracing GC CPython. Likely it doesn't perform too well but will provide us some feedback. Also, it would unblock Larry to restart a GIL removal project.

Providing non-moving object references to external API users

Some good GC optimizations (compaction of heap, new object nursery for fast object allocation) require that the GC can move objects. After the object is moved, all references to the object are automatically updated by the GC. That design requires the GC is absolutely sure of all references, otherwise you have crashes. The "conservative" approach doesn't work with a moving GC. Also, if we can't get 3rd party extensions to tell us the roots, we can't move those objects. This is a good article on pro and con of conservative GC, what must be done to be precise:

https://blog.mozilla.org/javascript/2013/07/18/clawing-our-way-back-to-precision/

The Spidermonkey documentation is quite extensive and provides a lot of detail on how embedders of Spidermonkey can mark roots and also details about how the GC works:

https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

Pablo suggested taking a look at what Ruby does with GC and extension APIs. Ruby is quite interesting in that every Ruby object is exactly 40 bytes long. That fact vastly simplifies memory management in ways. The downside is you have an extra indirection if object data doesn't fit in that 40 bytes. E.g. the bytes of a string would be allocated elsewhere.

A neat trick with the Ruby 40 byte objects is that they can individually pin objects, so the GC doesn't move them. That's just a bit inside the ruby object block. Things that are not pinned can get compacted by the GC. Here is video explaining design:

https://youtu.be/H8iWLoarTZc

The basic idea for allowing a moving collector internally is to have "proxy" objects that mirror every internal PyObject. The proxy objects would be at a fixed position in memory and not moved. Since the internal object doesn't need a reference count, the field could be moved tothe proxy and we save one word in the case no proxy is needed (i.e. the object is never revealed through the reference counting API). This is very close to what pypy cpyext does. If you don't move internal objects at all, having a separate proxy object is probably not required. The ref count can be in the internal PyObject and would be zero if the object was never exposed to the ref counted API. Such a design would be faster than pypy cpyext for heavy API use because you don't have to create and keep in-sync the proxies. The downside is you can't use a collector similar to the Pypy 'incminimark' one (incremental, generational, moving collector with new object nursery). Some info on PyPy GC options:

https://doc.pypy.org/en/latest/gc_info.html

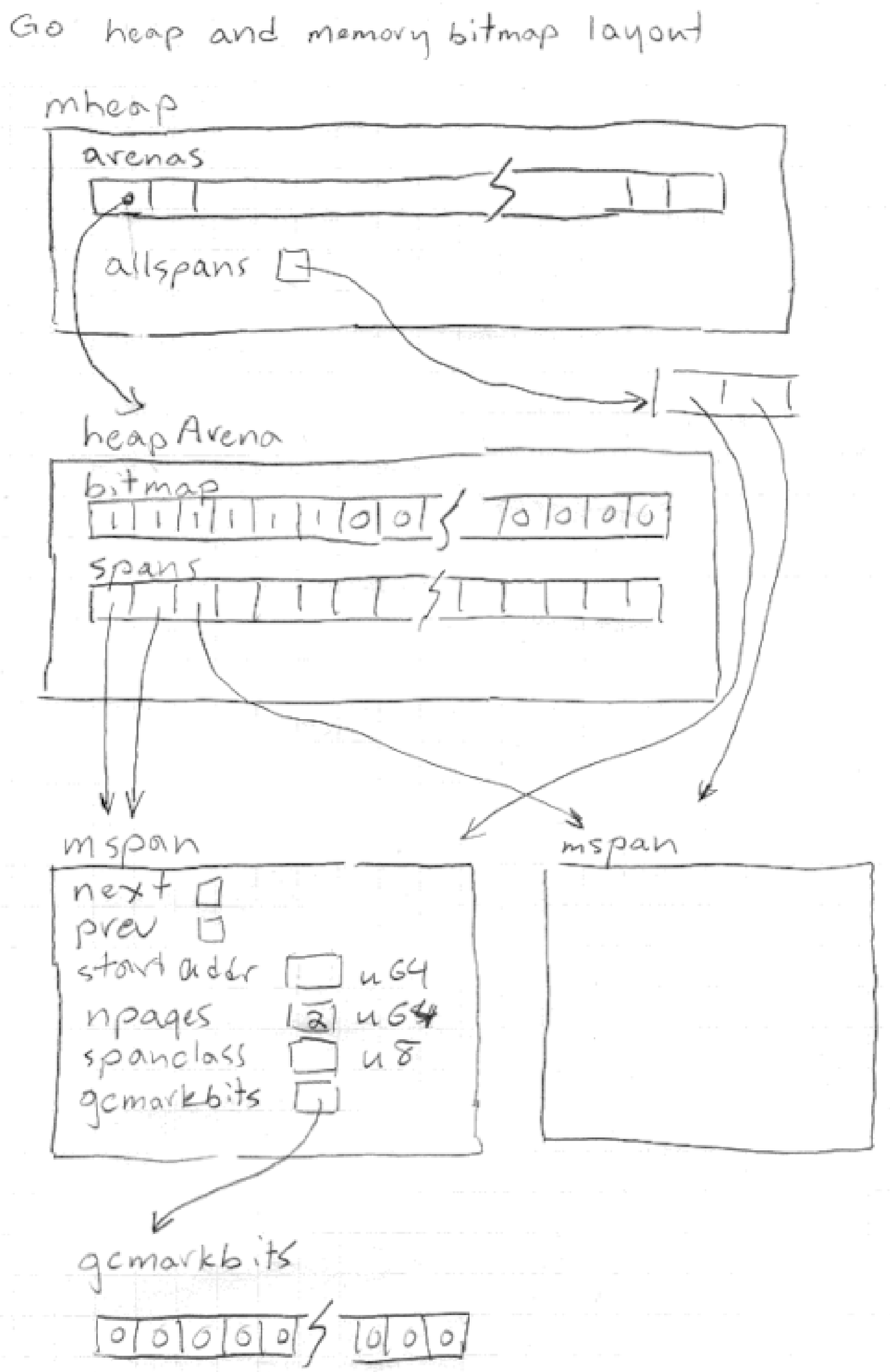
Using memory bitmaps to track GC related info

Neil was initially interested in this technique when exploring the idea of removing the PyGC_Head data currently allocated before every GC object. Very briefly, the idea is to pack PyGC_Head data tightly in arrays at the heads of obmalloc arenas (or something like that). See this discussion:

https://discuss.python.org/t/removing-pygc-head-for-small-objects/1743

Neil's radix tree obmalloc patch is a step towards that design. It adds the radix tree for memory arenas. With that change, there is a fast way to find the arena object for an arbitrary pointer. The next step would be to add bitmaps for individual PyObjects and some place to store the GC informat (GC refs, GC colors). You could also have a bitmap for 'pinned' (i.e. GC cannot move the object). The golang source code is a trove of info about how to do this. See the following in golang.org/src/runtime/: mheap.go, mbitmap.go.

Here is a hand-drawn sketch of the relevant bits of the Go runtime.



- mheap is singleton global
- one arena covers 64 MB, 8192 pages of 8192 bytes
- arenas array uses 32 MB for 48 bit addr space
- each 8192 byte page has a mspan
- mspan can cover more pages and cross arenas
- size of gcmarks array depends on sizeclass, allocated only while GC needs.

The design enables a fast ‘spanOf(p)’ function, where ‘p’ is an object pointer. Inside the span structure that is returned, you can store the GC_Head info as described above.

The Go runtime system is an example of the memory allocator and garbage collector being tightly integrated. They actually split it into more parts: heap manager (who owns what, provides the spanOf() function), page allocator (like memory allocator but deals with pages), memory allocator (uses page allocator, gives out large or small blocks of memory), garbage collector (decides when things can be freed). The Go runtime is concurrent and if we are going to remove the GIL, we would have to solve similar problems as they do.

An interesting nugget in the Go runtime is the “packed GC pointer bitmaps”, aka GC programs. See “func runGCProg” in the mbitmap.c source file. It is sort of a little bytecode interpreter that the GC runs to learn which pointers inside an object are pointers to other objects. Is it possible this could be used for speeding up tp_traverse or tp_clear somehow? It seems Go only uses it if the object has lots of internal pointers. Comment from source file is below:

```
// Packed GC pointer bitmaps, aka GC programs.
//
// For large types containing arrays, the type information has a
// natural repetition that can be encoded to save space in the
// binary and in the memory representation of the type information.
//
// The encoding is a simple Lempel-Ziv style bytecode machine
// with the following instructions:
//
// 00000000: stop
// 0nnnnnnn: emit n bits copied from the next (n+7)/8 bytes
// 10000000 n c: repeat the previous n bits c times; n, c are varints
// 1nnnnnnn c: repeat the previous n bits c times; c is a varint

// runGCProg executes the GC program prog, and then trailer if non-nil,
// writing to dst with entries of the given size.
//
// If size == 1, dst is a 1-bit pointer mask laid out moving forward from
// dst.
//
// If size == 2, dst is the 2-bit heap bitmap, and writes move backward
// starting at dst (because the heap bitmap does). In this case, the caller
// guarantees that only whole bytes in dst need to be written.
```

Pablo had the following idea inspired by “GC Progs”:

> Maybe a simple way of trying this idea is to add a 0-terminated array or something like that to the GH_HEAD struct and if that is not present for an object we use the tp_traverse to compute it. The array would contain the offset to every visited object from the object itself: `pointer_to_visited - pointer_to_object`. Then, our visit functions can iterate over the array instead of calling tp_traverse. The downside is obviously the memory usage.

Neil suggested that a heap structure like the Go mheap/mspan scheme would reduce the memory cost of these pointers to arrays. Each “span” object would have a pointer to the array of offsets. So rather than each object with a GC_Head having a pointer to the offset table, there would be a pointer per “span”. A span covers one “page” of memory (not necessarily OS page, typically 8 kB) and so there would be many fewer pointers. You would need to consolidate the offset array for all objects contained in the span. In the Go source code, the “gcbitmaps” pointer is the relevant thing. Go has a bitmap instead of an array of offsets. The bitmap is probably more efficient (memory and speed wise).

CHAPTER 2

Tracing GC for CPython

See also *CPython Garbage Collection*.

2.1 CPython GC Meeting, Oct 24, 2020

During the online Python core dev sprint.

2.1.1 Attending

- Neil Schemenauer
- Joannah Nanjekye
- Pablo Galindo Salgado

2.1.2 Agenda

- How to do the roots for tracing GC? Explain to Neil how could work.
- Define why a tracing a GC for CPython would be worthwhile (for CPython community, for research institution, for corporpate sponser)
- What are the constraints for changing CPython (backwards compatibility, support from core Python devs)

2.1.3 CPython tracing GC design, Constraints

- Must allow 3 rd party C extensions to compile and work without major source code changes
- Must allow most pure Python code to run, without changes. Some breakage expected (e.g. it assumes immediate deletion as given by RC).
  - Code that relies on 'gc' module is okay to break.

  - Would be nice to provide gc.get_referents(), gc.get_referents().
  - Support __del__ behavior, perhaps limit ability to resurrect objects? Must still call __del__ if there is a reference cycle.
  - Must support weakrefs and weakref callbacks.
- It is okay to change internal CPython C code and require extensive source code changes
- Ideally, it would only require C99 (or maybe newer ANSI C) and not C++ runtime
  - bringing C++ runtime into CPython could be problem (cannot intermix C++ runtimes?)
  - C++ doesn't have stable ABI
  - Some platforms don't even support C++ runtime
- It will be at least as fast as existing CPython and not use much more memory
- Wishlist item: friendly copy-on-write behavior
  - Some people think fork() is evil and copy-on-write doesn't matter (not useful on platforms other than Linux, BSDs)
  - Others (Instagram) rely heavily on this behavior, dirty pages from RC causes issue them

2.1.4 GC Properties

- GC pause time (current Python GC is stop-the-world but collection of younger generations is very fast, collection of oldest generation is much slower and can have pretty long pause times)
- Memory fragmentation (can be bad with non-moving GC)
- Portability (e.g. write barries that require OS support, userfaultfd(), stack crawling to find roots)
- Support for weakrefs
- Support for powerful finalizers

2.1.5 Which kind of GC can meet contraints above

- It could be moving GC but needs to provide stable object pointers for external C API
- Incremental GC would be good to reduce pauses•
- Generational GC is good so that GC time is not too long for large heaps
- Concurrent GC seems highly desirable, to make use of multiple-cores for speedup

2.1.6 Define why a tracing a GC for CPython would be worthwhile

- Allow finer grain locking (remove big GIL lock)
- RC on multi-core machines is not efficient, even if GIL remains
- A non-RC C API will allow internal optimizations for CPython and also allow other Python implementions to provide a common C API.
- Might be faster but a speedup vs RC does not seem a given (given all constraints)
- Can be good for attracting research interest in improving CPython, raise CPython as a platform for GC research.
- Python is now one of top used languages, even small improvement would be noteworthy research result

- Lessons learned can be re-used for future work
  - e.g. how can a compatible C API be provided if a tracing GC is used
  - Some of the solutions for that can be re-used even if initial GC is not ready to merge into offical CPython distribution (e.g. breaks C API too badly)
  - If prototype GC shows some good properties (runs faster if tested with pure Python benchmarks), gives us information on future path.

CHAPTER 3

Projects to optimize CPython 3.7

See also *Projects to optimize CPython 3.6*.

3.1 Big projects

- Multiple interepters per process
  - “solving multi-core Python”
  - https://mail.python.org/pipermail/python-ideas/2015-June/034177.html
  - http://ericsnowcurrently.blogspot.fr/2016/09/solving-mutli-core-python.html
- PyParallel
- Gilectomy: GIL-less CPython
- Add a JIT to CPython? :-) (see Pyston and Pyjion)

3.2 Smaller projects

- **MERGED**: Issue #26110: LOAD_METHOD and CALL_METHOD
  - Issue #29263: Implement LOAD_METHOD/CALL_METHOD for C functions
- Issue #28158: Implement LOAD_GLOBAL opcode cache
  - Issue #26219: implement per-opcode cache in ceval
  - Issue #10401: Globals / builtins cache
- Free list for single-digits ints
- FASTCALL

  - **REJECTED**: tp_fastcall: Issue #29259: Add tp_fastcall to PyTypeObject: support FASTCALL calling convention for all callable objects
  - **REJECTED**: Add tp_fastnew and tp_fastinit to PyTypeObject, 15-20% faster object instanciation
- Convert more C functions to METH_FASTCALL and Argument Clinic
  - Argument Clinic should understand *args and **kwargs parameters
  - Argument Clinic: Fix signature of optional positional-only arguments
  - _struct module
  - **DONE**: print() function. TODO: convert to Argument Clinic (need `*args`).
  - Search for Argument Clinic open issues
- Better bytecode/AST?
  - Issue #1346238: A constant folding optimization pass for the AST
  - Issue #11549: Build-out an AST optimizer, moving some functionality out of the peephole optimizer
- Split PyGC_Head from object (ML thread)
  - sizeof 1-tuple becomes (1 (pointer to gc head) + 3 (PyVarObject) + 1) words from (3 (gc head) + 3 + 1) words.
- Embed some tuples into code object.
  - When `co_consts` is `(None,)`, code object uses 8 (or 6 if above optimization is land) words for the tuple and the pointer to it. It can be 2 words (length and one PyObject*).
  - It may reduce RAM usage and improve cache utilization.
- Optimize option for stripping `__annotation__`.
  - Reduces one dict for each (annotated) functions.
  - `-O3` may be OK, but individual optimization flag (e.g. `-Odocstring`) would be better. It affects PEP 488.
- Interned-key only dict: Most name lookup uses interned string. If dict contains only interned keys only, lookup can see only pointer, and hash can be dropped from dict entries. This can reduce memory usage and cache utilization of namespece dicts.
- Global freepool: Many types has it's own freepool. Sharing freepool can increase memory and cache efficiency. Add `PyMem_FastFree(void* ptr, size_t size)` to store memory block to freepool, and `PyMem_Malloc` can check global freepool first.

CHAPTER 4

Projects to optimize CPython 3.6

See also *Projects to optimize CPython 3.7*.

4.1 Complete or almost complete projects

- **MERGED**: Wordcode
  - New format of bytecode which will allow to fetch opcode+oparg in a single 16-bit operation.
- *FAT Python*: PEP 509, PEP 510, PEP 511, fat and fatoptimizer.
  - Owner: Victor Stinner.
  - Speed-up: unknown :-(
- CPython build options for out-of-the box performance
  - Owner: Alecsandru Patrascu
  - Speed-up: unknown.
- **MERGED**: Change PyMem_Malloc to use PyObject_Malloc allocator?
  - Owner: Victor Stinner
  - Speed-up: up to 6% faster in fastpickle of perf.py (up to 22% faster on unpickle_list of perf.py, according to Intel run of perf.py).

4.2 Micro optimizations

4.2.1 Open

- Speedup method calls 1.2x
  - Owner: Yury Selivanov

- python-dev: Opcode cache in ceval loop
- python-dev: Speeding up CPython 5-10%
- Speedup: up to 21% faster on specific perf.py macro (micro?) benchmarks (call_method, call_method_slots, call_method_unknown).
- Related to implement per-opcode cache in ceval

- Globals / builtins cache
  - Owner: Antoine Pitrou
  - Speedup: 35% faster on a microbenchmark (LOAD_GLOBAL)
- ceval: Optimize list[int] (subscript) operation similarly to CPython 2.7
  - Owners: Yury Selivanov, Zach Byrne
  - Speed-up: up to 30% faster on microbenchmark.
- Free list for single-digits ints
  - Owners: Serhiy Storchaka, Yury Selivanov
  - Speedup: up to 18% faster on microbenchmark.
- Faster bit ops for single-digit positive longs
  - Owner: Yury Selivanov
  - Speedup: between 30% and 55% faster on a microbenchmark

: Closed ——

- [CLOSED, REJECTED] ceval.c: implement fast path for integers with a single digit
  - Owners: many authors :-)
  - Speedup: up to 26% on microbenchmark, unclear status on macrobenchmark. Unclear status for types other than int and float (slow-down or not?).

4.3 Experimental projects

- co_stacksize is calculated from unoptimized code
- FASTCALL: avoid creation of temporary tuple/dict when calling C and Python functions
  - Add a new _PyObject_FastCall() function which avoids the creation of a tuple or dict for arguments
  - property_descr_get:
    * segfault due to null pointer in tuple
    * Correct reuse argument tuple in property descriptor
    * property_descr_get reuse argument tuple
  - Tuple creation is too slow
  - C implementation of functools.lru_cache
- Change bytecode to optimize MAKE_FUNCTION, maybe also CALL_FUNCTION:
  - http://comments.gmane.org/gmane.comp.python.devel/157321

  - See also the optimization on CALL_FUNCTION with keyword parameters, but it requires FAT Python: https://bugs.python.org/issue26802#msg263775
- More efficient and/or more compact bytecode?
  - [Python-ideas] Wordcode v2, moved from -dev
  - [Python-ideas] More compact bytecode
  - Owner: Demur Rumed? Serhiy Storchaka?
  - Speed-up: unknown.
  - See also Speed-up oparg decoding on little-endian machines (speedup: 10% faster on microbenchmark)
- New peephole optimizer written in pure Python: bytecode.peephole_opt, requires the PEP 511.
  - Speed-up: probably negligible, and the Python optimizer is much slower than the C optimizer.
- INCA: Inline Caching meets Quickening in Python 3.3

CHAPTER 5

Notes on Python and CPython performance, 2017

- Python is slow compared to C (also compared to Javascript and/or PHP?)
- Solutions:
  - Ignore Python issue and solve the problem externally: buy faster hardware, buy new servers. At the scale of a computer: spawn more Python processes to feed all CPUs.
  - Use a different programming language: rewrite the whole application, or at least the functions where the program spend most of its time. Dropbox rewrote performance critical code in Go, then Dropbox stopped to sponsor Pyston.
  - Optimize CPython: solution discussed here. The two other options are not always feasible. Rewriting OpenStack in a different language would be too expensive for "little gain". Buying more hardware can become too expensive at very large scale.
- Python optimizations are limited by:
  - Weak typing: function prototypes don't have to define types of parameters and the return value. Annotations are fully optional and there is no plan to make type checks mandatory.
  - Python semantics: Python has powerful features which prevents optimizations. Examples: introspection and monkey-patching. A simple instruction like `obj.attr` can call `type(obj).__getattr__(attr)` or `type(obj).__getattribute__(attr)`, but it also requires to handle descriptors: call `descr.__get__(obj)...` It's not always a simple dictionary lookup. It's not allowed to replace `len("abc")` with `3` without a guard on the `len` global variable and the `len()` builtin function.
- CPython optimizations are limited by:
  - Age of its implementation: 20 years ago, phones didn't have 4 CPUs.
  - CPython implementation was designed to be simple to maintain, performance was not a design goal.
  - CPython exposes basically all its internal in a "C API" which is *widely* used by Python extensions modules (written in C) like numpy.
  - The C API exposes major implementation design choices:
    - Reference counting

    * A specific implementation of garbage collector
    * Global Interpreter Lock (GIL)
    * C structures: C extensions *can* access structure members, not everything is hidden in macros or functions.

5.1 JIT compiler

"Rebase" Pyston on Python 3.7 and continue the project?

Efficient optimizations require type information and assumptions. For example, function inlining requires that the inlined function is not modified. Guards must be added to deoptimize if something changed, and these guards must be checked at runtime.

Collecting information on types can be done at runtime. Type annotation might help, but Numba and PyPy need more precise types.

PyPy is a very efficient JIT compiler for Python, it is fully compatible with the Python language, but its support of the CPython C API is still incomplete and slower (the API is "emulated").

Failure of previous JIT compilers for CPython:

- Pyjion (not completely dead yet), written with Microsoft CLR
- Pyston (Dropbox doesn't sponsor it anymore), only support Python 2.7
- Unladen Swallow (dead)

Explanation of these failures:

- Unladen Swallow: LLVM wasn't as good as expected for dynamic languages like Python. Unladen Swallow contributed a lot to LLVM.
- LLVM API evolving quickly.
- Lack of sponsoring: it's just to justify working on Python performances. (see: "Spawn more processes! Buy new hardware!")
- Optimizing Python is harder than expected?

Notes on a JIT compiler for CPython:

- compatibility with CPython must be the most important point, PyPy took years to be fully compatible with CPython, compatibility was one reason of Pyston project failure
- must run Django faster than CPython: Django, not only microbenchmarks
- must keep compatibility with the C API
- be careful of memory usage: major issue in Unladen Swallow, and then Pyston

5.2 Optimization ahead of time (AoT compiler)

See FAT Python project which adds guards checked at runtime.

5.3 Break the C API?

The stable ABI created a subset of the CPython C API and hides most implementation details, but not all of them. Sadly, it's not popular... not sure if it really works in practice. Not sure that it would be feasible to use the stable ABI in numpy for example?

The Gilectomy project (CPython without GIL but locks per object) proposes to add a new compilation mode for extensions compatible with Gilectomy, but keep backward compatibility.

5.4 New language similar to Python

PHP has the Hack language which is similar but more strict and so easier to optimize in HHVM (JIT compiler for PHP and Hack).

Monkey-patching is very popular for unit tests, but do we need it on production applications?

Some parts of the Python language are very complex like getting an attribute (`obj.attr`). Would it be possible to restrict such feature? Would it allow to optimize a Python implementation?

CHAPTER 6

FAT Python

6.1 Intro

The FAT Python project was started by Victor Stinner in October 2015 to try to solve issues of previous attempts of “static optimizers” for Python. The main feature are efficient guards using versionned dictionaries to check if something was modified. Guards are used to decide if the specialized bytecode of a function can be used or not.

Python FAT is expected to be FAT… maybe FAST if we are lucky. FAT because it will use two versions of some functions where one version is specialised to specific argument types, a specific environment, optimized when builtins are not mocked, etc.

See the fatoptimizer documentation which is the main part of FAT Python.

The FAT Python project is made of multiple parts:

- The fatoptimizer project is the static optimizer for Python 3.6 using function specialization with guards. It is implemented as an AST optimizer.
- The fat module is a Python extension module (written in C) implementing fast guards. The `fatoptimizer` optimizer uses `fat` guards to specialize functions. `fat` guards are used to verify assumptions used to specialize the code. If an assumption is no more true, the specialized code is not used. The `fat` module is required to run code optimized by `fatoptimizer` if at least one function is specialized.
- Python Enhancement Proposals (PEP):
  - PEP 509: Add a private version to dict
  - PEP 510: Specialized functions with guards
  - PEP 511: API for AST transformers
- Patches for Python 3.6:
  - PEP 509: Add ma_version to PyDictObject
  - PEP 510: Specialize functions with guards
  - PEP 511: Add sys.set_code_transformers()
  - Related to the PEP 511:
    - *DONE*: PEP 511: Add test.support.optim_args_from_interpreter_flags()
    - *DONE*: PEP 511: code.co_lnotab: use signed line number delta to support moving instructions in an optimizer
    - *DONE*: PEP 511: Add ast.Constant to allow AST optimizer to emit constants
    - *DONE*: Lib/test/test_compileall.py fails when run directly
    - *DONE*: site ignores ImportError when running sitecustomize and usercustomize
    - *DONE*: code_richcompare() don't use constant type when comparing code constants

Announcements and status reports:

- Status of the FAT Python project, January 12, 2016
- 'FAT' and fast: What's next for Python: Article of InfoWorld by Serdar Yegulalp (January 11, 2016)
- [Python-Dev] Third milestone of FAT Python
- Status of the FAT Python project, November 26, 2015
- [python-dev] Second milestone of FAT Python (Nov 2015)
- [python-ideas] Add specialized bytecode with guards to functions (Oct 2015)

6.2 Getting started

Compile Python 3.6 patched with PEP 509, PEP 510 and PEP 511:

```
git clone https://github.com/vstinner/cpython -b fatpython fatpython
cd fatpython
./configure --with-pydebug CFLAGS="-O0" && make
```

Install fat:

```
git clone https://github.com/vstinner/fat
cd fat
../python setup.py build
cp -v build/lib*/fat.*so ../Lib
cd ..
```

For OS X users, use `./python.exe` instead of `./python`.

Install fatoptimizer:

```
git clone https://github.com/vstinner/fatoptimizer
(cd Lib; ln -s ../fatoptimizer/fatoptimizer .)
```

`fatoptimizer` is registed by the `site` module if `-X fat` command line option is used. Extract of `Lib/site.py`:

```
if 'fat' in sys._xoptions:
    import fatoptimizer
    fatoptimizer._register()
```

Check that fatoptimizer is registered with:

```
$ ./python -X fat -c 'import sys; print(sys.implementation.optim_tag)'
fat-opt
```

You must get `fat-opt` (and not `opt`).

6.3 How can you contribute?

The fatoptimizer project needs the most love. Currently, the optimizer is not really smart. There is a long TODO list. Pick a simple optimization, try to implement it, send a pull request on GitHub. At least, any kind of feedback is useful ;-)

If you know the C API of Python, you may also review the implementation of the PEPs:

- PEP 509: Add ma_version to PyDictObject
- PEP 510: Specialize functions with guards
- PEP 511: Add sys.set_code_transformers()

But these PEPs are still work-in-progress, so the implementation can still change.

6.4 Play with FAT Python

See *Getting started* to compile FAT Python.

6.4.1 Disable peephole optimizer

The `-o noopt` command line option disables the Python peephole optimizer:

```
$ ./python -o noopt -c 'import dis; dis.dis(compile("1+1", "test", "exec"))'
  1           0 LOAD_CONST               0 (1)
              3 LOAD_CONST               0 (1)
              6 BINARY_ADD
              7 POP_TOP
              8 LOAD_CONST               1 (None)
             11 RETURN_VALUE
```

6.4.2 Specialized code calling builtin function

Test fatoptimizer on builtin function:

```
$ ./python -X fat
>>> def func(): return len("abc")
...

>>> import dis
>>> dis.dis(func)
  1           0 LOAD_GLOBAL              0 (len)
              3 LOAD_CONST               1 ('abc')
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 RETURN_VALUE

>>> import fat
>>> fat.get_specialized(func)
[(<code object func at 0x7f9d3155b1e0, file "<stdin>", line 1>,
[<fat.GuardBuiltins object at 0x7f9d39191198>])]

>>> dis.dis(fat.get_specialized(func)[0][0])
  1           0 LOAD_CONST               1 (3)
              3 RETURN_VALUE
```

The specialized code is removed when the function is called if the builtin function is replaced (here by declaring a `len()` function in the global namespace):

```
>>> len=lambda obj: "mock"
>>> func()
'mock'
>>> fat.get_specialized(func)
[]
```

6.4.3 Microbenchmark

Run a microbenchmark on specialized code:

```
$ ./python -m timeit -s 'def f(): return len("abc")' 'f()'
10000000 loops, best of 3: 0.122 usec per loop

$ ./python -X fat -m timeit -s 'def f(): return len("abc")' 'f()'
10000000 loops, best of 3: 0.0932 usec per loop
```

Python must be optimized to run a benchmark: use `./configure && make clean && make` if you previsouly compiled it in debug mode.

You should compare specialized code to an unpatched Python 3.6 to run a fair benchmark (to also measure the overhead of PEP 509, 510 and 511 patches).

6.5 Run optimized code without registering fatoptimizer

You have to compile optimized .pyc files:

```
# the optimizer is slow, so add -v to enable fatoptimizer logs for more fun
./python -X fat -v -m compileall

# why does compileall not compile encodings/*.py?
./python -X fat -m py_compile Lib/encodings/{__init__,aliases,latin_1,utf_8}.py
```

Finally, enjoy optimized code with no registered optimized:

```
$ ./python -o fat-opt -c 'import sys; print(sys.implementation.optim_tag, sys.get_
↪code_transformers())'
fat-opt []
```

Remember that you cannot import .py files in this case, only .pyc:

```
$ echo 'print("Hello World!")' > hello.py
$ ENV/bin/python -o fat-opt -c 'import hello'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: missing AST transformers for 'hello.py': optim_tag='fat-opt',␣
↪transformers tag='noopt'
```

6.6 Origins of FAT Python

- *read-only Python*
- Dave Malcolm wrote a patch modifying Python/eval.c to support specialized functions. See the http://bugs.python.org/issue10399

6.7 See also

- Ruby: Deoptimization Engine

CHAPTER 7

Everything in Python is mutable

7.1 Problem

Developers like Python because it's possible to modify (almost) everything. This feature is heavily used in unit tests with unittest.mock which can override builtin function, override class methods, modify "constants, etc.

Most optimization rely on assumptions. For example, inlining rely on the fact that the inlined function is not modified. Implement optimization in respect of the Python semantics require to implement various assumptions.

7.1.1 Builtin functions

Python provides a lot of builtins functions. All Python applications rely on them, and usually don't expect that these functions are overriden. In practice, it is very easy to override them.

Example overriden the builtin `len()` function:

```
import builtins

def func(obj):
    print("length: %s" % len(obj))

func("abc")
builtins.len = lambda obj: "mock!"
func("abc")
```

Output:

```
length: 3
length: mock!
```

Technically, the `len()` function is loaded in `func()` with the `LOAD_GLOBAL` instruction which first tries to lookup in frame globals namespace, and then lookup in the frame builtins namespace.

Example overriding the `len()` builtin function with a `len()` function injected in the global namespace:

```
def func(obj):
    print("length: %s" % len(obj))

func("abc")
len = lambda obj: "mock!"
func("abc")
```

Output:

```
length: 3
length: mock!
```

Builtins are references in multiple places:

- the `builtins` module
- frames have a `f_builtins` attribute (builtins dictionary)
- the global `PyInterpreterState` structure has a `builtins` attribute (builtins dictionary)
- frame globals have a `__builtins__` variable (builtins dictionary, or builtins module when `__name__` equals `__main__`)

7.1.2 Function code

It is possible to modify at runtime the bytecode of a function to modify completly its behaviour. Example:

```
def func(x, y):
    return x + y

print("1+2 = %s" % func(1, 2))

def mock(x, y):
    return 'mock'

func.__code__ = mock.__code__
print("1+2 = %s" % func(1, 2))
```

Output:

```
1+2 = 3
1+2 = mock
```

7.1.3 Local variables

Technically, it is possible to modify local variable of a function outside the function.

Example of a function `hack()` which modifies the `x` local variable of its caller:

```
import sys
import ctypes

def hack():
    # Get the frame object of the caller
    frame = sys._getframe(1)
    frame.f_locals['x'] = "hack!"
```

(continues on next page)

(continued from previous page)

```
    # Force an update of locals array from locals dict
    ctypes.pythonapi.PyFrame_LocalsToFast(ctypes.py_object(frame),
                                          ctypes.c_int(0))

def func():
    x = 1
    hack()
    print(x)

func()
```

Output:

```
hack!
```

7.1.4 Modification made from other modules

A Python module A can be modified by a Python module B.

7.1.5 Multithreading

When two Python threads are running, the thread B can modify shared resources of thread A, or even resources which are supposed to only be access by the thread A like local variables.

The thread B can modify function code, override builtin functions, modify local variables, etc.

7.1.6 Python Imports and Python Modules

The Python import path `sys.path` is initialized by multiple environment variables (ex: `PYTHONPATH` and `PYTHONHOME`), modified by the `site` module and can be modified anytime at runtime (by modifying `sys.path` directly).

Moreover, it is possible to modify `sys.modules` which is the "cache" between a module fully qualified name and the module object. For example, `sys.modules['sys']` should be `sys`. It is posible to remove modules from `sys.modules` to force to reload a module. It is possible to replace a module in `sys.modules`.

The eventlet modules injects monkey-patched modules in `sys.modules` to convert I/O blocking operations to asynchronous operations using an event loop.

7.2 Solutions

7.2.1 Make strong assumptions, ignore changes

If the optimizer is an opt-in options, users are aware that the optimizer can make some compromises on the Python semantics to implement more aggressive optimizations.

7.2.2 Static analysis

Analyze the code to ensure that functions don't mutate everything, for example ensure that a function is pure.

Dummy example:

```
def func(x, y):
    return x + y
```

This function `func()` is pure if *x* and *y* are *int*: it has no side effect, the output only depends on the inputs. This function will not override builtins, not modify local variables of the caller, etc. It is safe to call this function from anywhere using guards on the type of *x* and *y* arguments.

It is possible to analyze the code to check that an optimization can be enabled.

7.2.3 Use guards checked at runtime

For some optimizations, a static analysis cannot ensure that all assumptions required by an optimization will respected. Adding guards allows to check assumptions during the execution to use the optimized code or fallback to the original code.

CHAPTER 8

Optimizations

See also fatoptimizer optimizations.

8.1 Inline function calls

Example:

```
def _get_sep(path):
    if isinstance(path, bytes):
        return b'/'
    else:
        return '/'

def isabs(s):
    """Test whether a path is absolute"""
    sep = _get_sep(s)
    return s.startswith(sep)
```

Inline `_get_sep()` into `isabs()` and simplify the code for the `str` type:

```
def isabs(s: str):
    return s.startswith('/')
```

It can be implemented as a simple call to the C function `PyUnicode_Tailmatch()`.

Note: Inlining uses more memory and disk because the original function should be kept. Except if the inlined function is unreachable (ex: “private function”?).

Links:

- Issue #10399: AST Optimization: inlining of function calls

8.2 CALL_METHOD

See issue #26110: Speedup method calls 1.2x

8.3 Move invariants out of the loop

Example:

```
def func(obj, lines):
    for text in lines:
        print(obj.cleanup(text))
```

Become:

```
def func(obj, lines):
    local_print = print
    obj_cleanup = obj.cleanup
    for text in lines:
        local_print(obj_cleanup(text))
```

Local variables are faster than global variables and the attribute lookup is only done once.

8.4 C functions using only C types

Optimizations:

- Avoid reference counting
- Memory allocations on the heap
- Release the GIL

Example:

```
def demo():
    s = 0
    for i in range(10):
        s += i
    return s
```

In specialized code, it may be possible to use basic C types like `char` or `int` instead of Python codes which can be allocated on the stack, instead of allocating objects on the heap. `i` and `s` variables are integers in the range `[0; 45]` and so a simple C type `int` (or even `char`) can be used:

```
PyObject *demo(void)
{
    int s, i;
    Py_BEGIN_ALLOW_THREADS
    s = 0;
    for(i=0; i<10; i++)
        s += i;
    Py_END_ALLOW_THREADS
    return PyLong_FromLong(s);
}
```

Note: if the function is slow, we may need to check sometimes if a signal was received.

8.5 Release the GIL

Many methods of builtin types don't need the *GIL*. Example: `"abc".startswith("def")`.

8.6 Replace calls to pure functions with the result

Examples:

- `len('abc')` becomes `3`
- `"python2.7".startswith("python")` becomes `True`
- `math.log(32) / math.log(2)` becomes `5.0`

Can be implemented in the AST optimizer.

8.7 Constant propagation

Propagate constant values of variables. Example:

Original	Constant propagation
`def func()`	`def func()`
`    x = 1`	`    x = 1`
`    y = x`	`    y = 1`
`    return y`	`    return 1`

Implemented in fatoptimizer.

Read also the Wikipedia article on copy propagation.

8.8 Constant folding

Compute simple operations at the compilation. Usually, at least arithmetic operations (a+b, a-b, a*b, etc.) are computed. Example:

Original	Constant folding
`def func()`	`def func()`
`    return 1 + 1`	`    return 2`

Implemented in fatoptimizer and the *CPython peephole optimizer*.

See also

- issue #1346238: A constant folding optimization pass for the AST

- Wikipedia article on constant folding.

8.9 Peephole optimizer

See *CPython peephole optimizer*.

8.10 Loop unrolling

Example:

```
for i in range(4):
    print(i)
```

The loop body can be duplicated (twice in this example) to reduce the cost of a loop:

```
for i in range(0,4,2):
    print(i)
    print(i+1)
i = 3
```

Or the loop can be removed by duplicating the body for all loop iterations:

```
i=0
print(i)
i=1
print(i)
i=2
print(i)
i=3
print(i)
```

Combined with other optimizations, the code can be simplified to:

```
print('0')
print('1')
print('2')
i = 3
print('3')
```

Implemented in fatoptimizer

Read also the Wikipedia article on loop unrolling.

8.11 Dead code elimination

- Replace `if 0:  code` with `pass`
- `if DEBUG: print("debug")` where `DEBUG` is known to be False

Implemented in fatoptimizer and the *CPython peephole optimizer*.

See also Wikipedia Dead code elimination article.

8.12 Load globals and builtins when the module is loaded

Load globals when the module is loaded? Ex: load “print” name when the module is loaded.

Example:

```
def hello():
    print("Hello World")
```

Become:

```
local_print = print

def hello():
    local_print("Hello World")
```

Useful if `hello()` is compiled to C code.

fatoptimizer implements a “copy builtins to constants optimization” optimization.

8.13 Don’t create Python frames

Inlining and other optimizations don’t create Python frames anymore. It can be a serious issue to debug programs: tracebacks are an important feature of Python.

At least in debug mode, frames should be created.

PyPy supports lazy creation of frames if an exception is raised.

CHAPTER 9

Python bytecode

9.1 CPython peephole optimizer

Implementation: Python/peephole.c

Optmizations:

- *Constant folding*
- *Dead code elimination*
- Some other optimizations more specific to the bytecode, like removal of useless jumps and optimizations on conditional jumps

Latest enhancement:

```
changeset:   68375:14205d0fee45
user:        Antoine Pitrou <solipsis@pitrou.net>
date:        Fri Mar 11 17:27:02 2011 +0100
files:       Lib/test/test_peepholer.py Misc/NEWS Python/peephole.c
description:
Issue #11244: The peephole optimizer is now able to constant-fold
arbitrarily complex expressions.  This also fixes a 3.2 regression where
operations involving negative numbers were not constant-folded.
```

Compiler enhancement to reduce the number of stupid jumps:

```
changeset:   92460:c0ca9d32aed4
user:        Antoine Pitrou <solipsis@pitrou.net>
date:        Thu Sep 18 03:06:50 2014 +0200
files:       Lib/test/test_dis.py Misc/NEWS Python/compile.c
description:
Closes #11471: avoid generating a JUMP_FORWARD instruction at the end
of an if-block if there is no else-clause.

Original patch by Eugene Toder.
```

Should be rewritten as an *AST optimizer*.

9.2 Bytecode

- bytecode
- byteplay: byteplay documentation (see also the old byteplay hosted on Google Code)
- diving-into-byte-code-optimization-in-python
- BytecodeAssembler

CHAPTER 10

Python C API

10.1 Intro

CPython comes with a C API called the "Python C API". The most common type is `PyObject*` and functions are prefixed with `Py` (and `_Py` for private functions but you must not use them!).

10.2 Historical design choices

CPython was created in 1991 by Guido van Rossum. Some design choices made sense in 1991 but don't make sense anymore in 2015. For example, the *GIL* was a simple and safe choice to implement multithreading in CPython. But in 2015, smartphones have 2 or 4 cores, and desktop PC have between 4 and 8 cores. The GIL restricts peek performances on multithreaded applications, even when it's possible to release the GIL.

10.3 GIL

CPython uses a Global Interpreter Lock called "GIL" to avoid concurrent accesses to CPython internal structures (shared resources like global variables) to ensure that Python internals remain consistent.

See also *Kill the GIL*.

10.4 Reference counting and garbage collector

The C structure of all Python objects inherit from the `PyObject` structure which contains the field `Py_ssize_t ob_refcnt;`. This is a simple counter initialized to `1` when the object is created, increased each time that a variable has a strong reference to the object, and decreased each time that a strong reference is removed. The object is removed when the counter reached `0`.

In some cases, two objects are linked together. For example, A has a strong reference to B which has a strong reference to A. Even if A and B are no more referenced outside, these objects are not destroyed because their reference counter is still equal to `1`. A garbage collector is responsible to find and break *reference cycles*.

See also the PEP 442: Safe object finalization implemented in Python 3.4 which helps to break reference cycles.

10.5 Popular projects using the Python C API

- numpy
- PyQt
- Mercurial

CHAPTER 11

AST Optimizers

11.1 Intro

An AST optimizer rewrites the Abstract Syntax Tree (AST) of a Python module to produce a more efficient code.

Currently in CPython 3.5, only basic optimizations are implemented by rewriting the bytecode: *CPython peephole optimizer*.

11.2 fatoptimizer

fatoptimizer project: AST optimizer implementing multiple optimizations and can specialize functions using guards of the `fat` module.

11.3 pythran AST

pythran.analysis.PureFunctions of pythran project, depend on ArgumentEffects and GlobalEffects analysis: automatically detect pure functions.

11.4 PyPy AST optimizer

https://bitbucket.org/pypy/pypy/src/default/pypy/interpreter/astcompiler/optimize.py

11.5 Cython AST optimizer

https://mail.python.org/pipermail/python-dev/2012-August/121300.html

- Compiler/Optimize.py

- Compiler/ParseTreeTransforms.py
- Compiler/Builtin.py
- Compiler/Pipeline.py

11.6 Links

11.6.1 CPython issues

- Issue #2181: optimize out local variables at end of function
- Issue #2499: Fold unary + and not on constants
- Issue #4264: Patch: optimize code to use LIST_APPEND instead of calling list.append
- Issue #7682: Optimisation of if with constant expression
- Issue #11549: Build-out an AST optimizer, moving some functionality out of the peephole optimizer
- Issue #17068: peephole optimization for constant strings
- Issue #17430: missed peephole optimization

11.6.2 AST

- instrumenting_the_ast.html
- the-internals-of-python-generator-functions-in-the-ast
- tlee-ast-optimize branch
- ast-optimization-branch-elimination-in-generator-functions

CHAPTER 12

Register-based Virtual Machine for Python

12.1 Intro

registervm is a fork of CPython 3.3 using register-based bytecode, instead of stack-code bytecode

More information: REGISTERVM.txt

Thread on the Python-Dev mailing list: Register-based VM for CPython.

The project was created in November 2012.

12.2 Status

- Most instructions using the stack are converted to instructions using registers
- Bytecode using registers with all optimizations enable is usually 10% faster than bytecode using the stack, according to pybench
- registervm generates invalid code, see TODO section below, so it's not possible yet to use it on the Python test suite

12.3 TODO

12.3.1 Bugs

- Register allocator doesn't handle correctly conditional branches: CLEAR_REG is removed on the wrong branch in test_move_instr.
- Fail to track the stack state in if/else. Bug hidden by the register allocator in the following example:

  def func(obj): obj.attr = sys.modules['warnings'] if module is None else module
- Don't move globals out of if. Only out of loops? subprocess.py:

```
if mswindows:
    if p2cwrite != -1:
        p2cwrite = msvcrt.open_osfhandle(p2cwrite.Detach(), 0)
```

But do move len() out of loop for:

```
def loop_move_instr():
    length = 0
    for i in range(5):
        length += len("abc") - 1
    return length
```

- Don't remove duplicate LOAD_GLOBAL in "LOAD_GLOBAL . . . ; CALL_PROCEDURE . . . ; LOAD_GLOBAL . . . ": CALL_PROCEDURE has border effect
- Don't remove duplicate LOAD_NAME if a function has a border effect:

```
x=1
def modify():
    global x
    x = 2
print(x)
modify()
print(x)
```

12.3.2 Improvments

- Move LOAD_CONST out of loops: it was done in a previous version, but the optimization was broken by the introduction of CLEAR_REG
- Copy constants to the frame objects so constants can be used as registers and LOAD_CONST instructions can be simplify removed
- Enable move_load_const by default?
- Fix moving LOAD_ATTR_REG: only do that when calling methods. See test_sieve() of test_registervm: primes.append().

```
result = Result()
while 1:
    if result.done:
        break
    func(result)
```

- Reenable merging duplicate LOAD_ATTR
- Register allocation for locale_alias = {. . . } is very very slow
- "while 1: . . . return" generates useless SETUP_LOOP
- Reuse locals?
- implement register version of the following instructions:
  - DELETE_ATTR
  - try/finally
  - yield from

  - CALL_FUNCTION_VAR_KW
  - CALL_FUNCTION_VAR
  - operators: `a | b, a & b, a ^ b, a |= b, a &= b, a ^= b`
- DEREF:
  - add a test using free variables
  - Move LOAD_DEREF_REG out of loops
- NAME:
  - test_list_append() of test_registervm.py
  - Move LOAD_NAME_REG out of loop
- Handle JUMP_IF_TRUE_OR_POP: see test_getline() of test_registervm
- Compute the number of used registers in a frame
- Write a new test per configuration option
- Factorize code processing arg_types, ex: disassmblers of dis and registervm modules
- Add tests on class methods
- Fix lnotab

12.4 Changelog

2012-12-21

- Use RegisterTracker to merge duplicated LOAD, STORE_GLOBAL/LOAD_GLOBAL are now also simplified

2012-12-19

- Emit POP_REG to simplify the stack tracker

2012-12-18

- LOAD are now only moved out of loops

2012-12-14

- Duplicated LOAD instructions can be merged without moving them
- Rewrite the stack tracker: PUSH_REG don't need to be moved anymore
- Fix JUMP_IF_TRUE_OR_POP/JUMP_IF_FALSE_OR_POP to not generate invalid code
- Don't move LOAD_ATTR_REG out of try/except block

2012-12-11

- Split instructions into linked-blocks

2012-11-26

- Add a stack tracker

2012-11-20

- Remove useless jumps
- CALL_FUNCTION_REG and CALL_PROCEDURE_REG are fully implemented

2012-10-29

- Remove "if (HAS_ARG(op))" check in PyEval_EvalFrameEx()

2012-10-27

- Duplicated LOAD_CONST and LOAD_GLOBAL are merged (optimization disabled on LOAD_GLOBAL because it is buggy)

2012-10-23

- initial commit, 0f7f49b7083c

12.5 CPython 3.3 bytecode is inefficient

- Useless jump: JUMP_ABSOLUTE <offset+0>
- Generate dead code: RETURN_VALUE; RETURN_VALUE (the second instruction is unreachable)
- Duplicate constants: see TupleSlicing of pybench
- Constant folding: see astoptimizer project
- STORE_NAME 'f'; LOAD_NAME 'f'
- STORE_GLOBAL 'x'; LOAD_GLOBAL 'x'

12.6 Rationale

The performance of the loop evaluating bytecode is critical in Python. For Python example, using computed-goto instead of switch to dispatch bytecode improved performances by 20%. Related issues:

- use computed goto's in ceval loop
- Faster opcode dispatch on gcc
- Computed-goto patch for RE engine

Using registers of a stack reduce the number of operations, but increase the size of the code. I expect an significant speedup when all operations will use registers.

12.7 Optimizations

Optimizations:

- Remove useless LOAD_NAME and LOAD_GLOBAL. For example: "STORE_NAME var; LOAD_NAME var"
- Merge duplicate loads (LOAD_CONST, LOAD_GLOBAL_REG, LOAD_ATTR). For example, "lst.append(1); lst.append(1)" only gets constant "1" and the "lst.append" attribute once.

Misc:

- Automatically detect inplace operations. For example, "x = x + y" is compiled to "BINARY_ADD_REG 'x', 'x', 'y'" which calls PyNumber_InPlaceAdd(), instead of PyNumber_Add().
- Move constant, global and attribute loads out of loops (to the beginning)
- Remove useless jumps (ex: JUMP_FORWARD <relative jump to 103 (+0)>)

12.8 Algorithm

The current implementation rewrites the stack-based operations to use register-based operations instead. For example, “LOAD_GLOBAL range” is replaced with “LOAD_GLOBAL_REG R0, range; PUSH_REG R0”. This first step is inefficient because it increases the number of operations.

Then, operations are reordered: PUSH_REG and POP_REG to the end. So we can replace “PUSH_REG R0; PUSH_REG R1; STACK_OPERATION; POP_REG R2” with a single operatiton: “REGISTER_OPERATION R2, R0, R1”.

Move invariant out of the loop: it is possible to move constants out of the loop. For example, LOAD_CONST_REG are moved to the beginning. We might also move LOAD_GLOBAL_REG and LOAD_ATTR_REG to the beginning.

Later, a new AST to bytecote compiler can be implemented to emit directly operations using registers.

12.9 Example

Simple function computing the factorial of n:

```
def fact_iter(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f
```

Stack-based bytecode (20 instructions):

```
      0 LOAD_CONST             1 (const#1)
      3 STORE_FAST             'f'
      6 SETUP_LOOP             <relative jump to 46 (+37)>
      9 LOAD_GLOBAL            0 (range)
     12 LOAD_CONST             2 (const#2)
     15 LOAD_FAST              'n'
     18 LOAD_CONST             1 (const#1)
     21 BINARY_ADD
     22 CALL_FUNCTION          2 (2 positional, 0 keyword pair)
     25 GET_ITER
>>   26 FOR_ITER               <relative jump to 45 (+16)>
     29 STORE_FAST             'i'
     32 LOAD_FAST              'f'
     35 LOAD_FAST              'i'
     38 INPLACE_MULTIPLY
     39 STORE_FAST             'f'
     42 JUMP_ABSOLUTE          <jump to 26>
>>   45 POP_BLOCK
>>   46 LOAD_FAST              'f'
     49 RETURN_VALUE
```

Register-based bytecode (13 instructions):

```
      0 LOAD_CONST_REG         'f', 1 (const#1)
      5 LOAD_CONST_REG         R0, 2 (const#2)
     10 LOAD_GLOBAL_REG        R1, 'range' (name#0)
     15 SETUP_LOOP             <relative jump to 57 (+39)>
     18 BINARY_ADD_REG         R2, 'n', 'f'
```

(continues on next page)

(continued from previous page)

```
     25 CALL_FUNCTION_REG    4, R1, R1, R0, R2
     36 GET_ITER_REG         R1, R1
>>   41 FOR_ITER_REG         'i', R1, <relative jump to 56 (+8)>
     48 INPLACE_MULTIPLY_REG 'f', 'i'
     53 JUMP_ABSOLUTE        <jump to 41>
>>   56 POP_BLOCK
>>   57 RETURN_VALUE_REG     'f'
```

The body of the main loop of this function is composed of 1 instructions instead of 5.

12.10 Comparative table

```
Example      |S|r|R|            Stack                 |            Register
------------+-+-+-+----------------------------------+--------------------------------
↪--------------------
append(2)    |4|1|2| LOAD_FAST 'append'               | LOAD_CONST_REG R1, 2 (const#2)
             | | | | LOAD_CONST 2 (const#2)           | ...
             | | | | CALL_FUNCTION (1 positional)     | ...
             | | | | POP_TOP                          | CALL_PROCEDURE_REG 'append',␣
↪(1 positional), R1
------------+-+-+-+----------------------------------+--------------------------------
↪--------------------
l[0] = 3     |4|1|2| LOAD_CONST 3 (const#1)           | LOAD_CONST_REG R0, 3 (const#1)
             | | | | LOAD_FAST 'l'                    | LOAD_CONST_REG R3, 0 (const#4)
             | | | | LOAD_CONST 0 (const#4)           | ...
             | | | | STORE_SUBSCR                     | STORE_SUBSCR_REG 'l', R3, R0
------------+-+-+-+----------------------------------+--------------------------------
↪--------------------
x = l[0]     |4|1|2| LOAD_FAST 'l'                    | LOAD_CONST_REG R3, 0 (const#4)
             | | | | LOAD_CONST 0 (const#4)           | ...
             | | | | BINARY_SUBSCR                    | ...
             | | | | STORE_FAST 'x'                   | BINARY_SUBSCR_REG 'x', 'l', R3
------------+-+-+-+----------------------------------+--------------------------------
↪--------------------
s.isalnum() |4|1|2| LOAD_FAST 's'                    | LOAD_ATTR_REG R5, 's', 'isalnum
↪' (name#3)
             | | | | LOAD_ATTR 'isalnum' (name#3)     | ...
             | | | | CALL_FUNCTION (0 positional)     | ...
             | | | | POP_TOP                          | CALL_PROCEDURE_REG R5, (0␣
↪positional)
------------+-+-+-+----------------------------------+--------------------------------
↪--------------------
o.a = 2      |3|1|2| LOAD_CONST 2 (const#3)           | LOAD_CONST_REG R2, 2 (const#3)
             | | | | LOAD_FAST 'o'                    | ...
             | | | | STORE_ATTR 'a' (name#2)          | STORE_ATTR_REG 'o', 'a' (name
↪#2), R2
------------+-+-+-+----------------------------------+--------------------------------
↪--------------------
x = o.a      |3|1|1| LOAD_FAST 'o'                    | LOAD_ATTR_REG 'x', 'o', 'a'␣
↪(name#2)
             | | | | LOAD_ATTR 'a' (name#2)           |
             | | | | STORE_FAST 'x'                   |
------------+-+-+-+----------------------------------+--------------------------------
↪--------------------
```

Columns:

- "S": Number of stack-based instructions
- "r": Number of stack-based instructions exclusing instructions moved out of loops (ex: LOAD_CONST_REG)
- "R": Total number of stack-based instructions (including instructions moved out of loops)

CHAPTER 13

Read-only Python

13.1 Intro

A first attempt to implement guards was the readonly PoC (fork of CPython 3.5) which registered callbacks to notify all guards. The problem is that modifying a watched dictionary gets a complexity of O(n) where n is the number of registered guards.

readonly adds a `modified` flag to types and a `readonly` property to dictionaries. The guard was notified with the modified key to decide to disable or not the optimization.

More information: READONLY.txt

Thread on the python-ideas mailing list: Make Python code read-only.

The project was mostly developed in May 2014. The project is now dead, replaced with *FAT Python*.

13.2 READONLY

This fork on CPython 3.5 adds a machinery to be notified when the Python code is modified. Modules, classes (types) and functions are tracked. At the first modification, a callback is called with the object and the modified attribute.

This machinery should help static optimizers.

Examples of such optimizers:

- astoptimizer project: replace a function call by its result during the AST compilation
- Learn types of function paramters and local variables, and then compile Python (byte)code to machine code specialized for these types (like Cython)

13.3 Issues with read-only code

- Currently, it's not possible to allow again to modify a module, class or function to keep my implementation simple. With a registry of callbacks, it may be possible to enable again modification and call code to disable optimizations.
- PyPy implements this but thanks to its JIT, it can optimize again the modified code during the execution. Writing a JIT is very complex, I'm trying to find a compromise between the fast PyPy and the slow CPython. Add a JIT to CPython is out of my scope, it requires too much modifications of the code.
- With read-only code, monkey-patching cannot be used anymore. It's annoying to run tests. An obvious solution is to disable read-only mode to run tests, which can be seen as unsafe since tests are usually used to trust the code.
- The sys module cannot be made read-only because modifying sys.stdout and sys.ps1 is a common use case.
- The warnings module tries to add a __warningregistry__ global variable in the module where the warning was emited to not repeat warnings that should only be emited once. The problem is that the module namespace is made read-only before this variable is added. A workaround would be to maintain these dictionaries in the warnings module directly, but it becomes harder to clear the dictionary when a module is unloaded or reloaded. Another workaround is to add __warningregistry__ before making a module read-only.
- Lazy initialization of module variables does not work anymore. A workaround is to use a mutable type. It can be a dict used as a namespace for module modifiable variables.
- The interactive interpreter sets a "_" variable in the builtins namespace. I have no workaround for this. The "_" variable is no more created in read-only mode. Don't run the interactive interpreter in read-only mode.
- It is not possible yet to make the namespace of packages read-only. For example, "import encodings.utf_8" adds the symbol "utf_8" to the encodings namespace. A workaround is to load all submodules before making the namespace read-only. This cannot be done for some large modules. For example, the encodings has a lot of submodules, only a few are needed.

13.4 STATUS

- Python API:
  - new function.__modified__ and type.__modified__ properties: False by default, becomes True when the object is modified
  - new module.is_modified() method
  - new module.set_initialized() method
- C API:
  - PyDictObject: new "int ma_readonly;" field
  - PyTypeObject: a new "int tp_modified;" field
  - PyFunctionObject: new "int func_module;" and "int func_initialized;" fields
  - PyModuleObject: new "int md_initialized;" field

13.5 Modified modules, classes and functions

- It's common to modify the following attributes of the sys module:

  - sys.ps1, sys.ps2
  - sys.stdin, sys.stdout, sys.stderr
- "import encodings.latin_1" sets "latin_1" attribute in the namespace of the "encodings" module.
- The interactive interpreter sets the "_" variable in builtins.
- warnings: global variable __warningregistry__ set in modules
- functools.wraps() modifies the wrapper to copy attributes of the wrapped function

13.6 TODO

- builtins modified in initstdio(): builtins.open modified
- sys modified in initstdio(): sys.__stdin__ modified
- structseq: types are created modified; same issue with _ast types (Python-ast.c)
- module, type and function __dict__:
  - Drop dict.setreadonly()
  - Decide if it's better to use dict.setreadonly() or a new subclass (ex: "dict_maybe_readonly" or "namespace").
  - Read only dict: add a new ReadOnlyError instead of ValueError?
  - sysmodule.c: PyDict_DelItemString(FlagsType.tp_dict, "__new__") doesn't mark FlagsType as modified
  - Getting func.__dict__ / module.__dict__ marks the function/module as modified, this is wrong. Use instead a mapping marking the function as modified when the mapping is modified.
  - module.__dict__ is read-only: similar issue for functions.
- Import submodule. Example: "import encodings.utf_8" modifies "encoding" to set a new utf_8 attribute

13.7 TODO: Specialized functions

13.7.1 Environment

- module and type attribute values:
  - ("module", "os", OS_CHECKSUM)
  - ("attribute", "os.path")
  - ("module", "path", PATH_CHECKSUM)
  - ("attribute", "path.isabs")
  - ("function", "path.isabs")
- function attributes
- set of function parameter types (passed as indexed or keyword arguments)

13.7.2 Read-only state

Scenario:

- 1: application.py is compiled. Function A depends on os.path.isabs, function B depends on project.DEBUG
- 2: application is started, "import os.path"
- 3: os.path.isabs is modified
- 4: optimized application.py is loaded
- 5: project.DEBUG is modified

When the function is created, os.path.isabs was already modified compared to the OS_CHECKSUM.

13.7.3 Example of environments

- The function calls "os.path.isabs":
  - rely on "os.path" attribute
  - rely on "os.path.isabs" attribute
  - rely on "os.path.isabs" function attributes (except __doc__)
- The function "def mysum(x, y):" has two parameters
  - x type is int and y type is int
  - or: x type is str and y type is str
  - ("type is": check the exact type, not a subclass)
- The function uses "project.DEBUG" constant
  - rely on "project.DEBUG" attribute

13.7.4 Content of a function

- classic attributes: doc, etc.
- multiple versions of the code:
  - required environment of the code
  - bytecode

13.7.5 Create a function

- build the environment
- register on module, type and functions modification

13.7.6 Callback when then environment is modified

xxx

13.7.7 Call a function

xxx

13.8 LINKS

- http://legacy.python.org/dev/peps/pep-0351/ : Get an immutable copy of arbitrary objects
- http://legacy.python.org/dev/peps/pep-0416/ : add a new frozendict type => types.MappingProxy added to Python 3.3

CHAPTER 14

History of Python optimizations

- 2002: Creation of the psyco project by Armin Rigo
- 2003-05-05: psyco 1.0 released
- Spring 1997: Creation of Jython project (initially called JPython) by Jim Hugunin
- 2006-09-05: Creation of IronPython project by Jim Hugunin
- Creation of PyPy, spin-off of psyco
- mid-2007: PyPy 1.0 released.
- 2009-03: Creation of Unladen Swallow project by some Google employees
- 2010-Q1: Google stops funding Unladden Swallow
- 2012-09: Creation of the astoptimizer project by Victor Stinner
- 2012-11: Creation of the *registervm* project by Victor Stinner
- 2014-04-03: Creation of Pyston project by Kevin Modzelewsk and the Dropbox team
- 2014-05: Creation of *read-only Python* PoC by Victor Stinner
- 2015-10: Creation of the *FAT Python* project by Victor Stinner
- 2016-01: Creation of Pyjion by Brett Canon and some Microsoft employes
- 2017-01-03 : Brett Cannon add a note to say to expect sporadic progress from the project
- 2017-01-31: Dropbox stops funding Pyston

CHAPTER 15

Misc

15.1 Ideas

- PyPy CALL_METHOD instructor
- Lazy formatting of Exception message: in most cases, the message is not used. AttributeError(message) => AttributeError(attr=name), lazy formatting for str(exc) and exc.args.

15.2 Plan

- Modify CPython to be *notified when the Python code is changed*
- *Learn types* of function parameters and variables
- *Choose between bytecode and specialized code* at runtime

Other idea:

- registervm: My fork of Python 3.3 using register-based bytecode, instead of stack-code bytecode. Read REGISTERVM.txt
- *Kill the GIL?*

15.3 Status

See also the status of individual projects:

- READONLY.txt
- REGISTERVM.txt

15.3.1 Done

- Fork of CPython 3.5: be notified when the Python code is changed: modules, types and functions are tracked. My fork of CPython 3.5: readonly; read READONLY.txt documentation.

Note: "readonly" is no more a good name for the project. The name comes from a first implementation using read-only code.

15.4 Why Python is slow?

15.4.1 Why the CPython implementation is slower than PyPy?

- everything is stored as an object, even simple types like integers or characters. Computing the sum of two numbers requires to "unbox" objects, compute the sum, and "box" the result.
- Python maintains different states: thread state, interperter state, frames, etc. These informations are available in Python. The common usecase is to display a traceback in case of a bug. PyPy builds frames on demand.
- Cost of maintaince the reference counter: Python programs rely on the garbage collector
- ceval.c uses a virtual stack instead of CPU registers

15.4.2 Why the Python language is slower than C?

- modules are mutable, classes are mutable, etc. Because of that, it is not possible to inline code nor replace a function call by its result (ex: len("abc")).
- The types of function parameters and variables are unknown. Example of missing optimizations:
  - "obj.attr" instruction cannot be moved out of a loop: "obj.attr" may return a different result at each call, or execute arbitrary Python code
  - x+0 raises a TypeError for "abc", whereas it is a noop for int (it can be replaced with just `x`)
  - conditional code becomes dead code when types are known
- obj.method creates a temporary bounded method

15.4.3 Why improving CPython instead of writing a new implementation?

- There are already a lot of other Python implementations. Some examples: PyPy, Jython, IronPython, Pyston.
- CPython remains the reference implementation: new features are first implemented in CPython. For example, PyPy doesn't support Python 3 yet.
- Important third party modules rely heavily on CPython implementation details, especially the *Python C API*. Examples: numpy and PyQt.

15.4.4 Why not a JIT?

- write a JIT is much more complex, it requires deep changes in CPython; CPython code is old (+20 years)
- cost to "warm up" the JIT: Mercurial project is concerned by the Python startup time

- Store generated machine code?

15.5 Learn types

- Add code in the compiler to record types of function calls. Run your program. Use recorded types.
- Range of numbers (predict C int overflow)
- Optional paramters: forceload=0. Dead code with forceload=0.
- Count number of calls to the function to decide if it should be optimized or not.
- Measure time spend in a function. It can be used to decide if it's useful to release or not the GIL.
- Store type information directly in the source code? Manual type annotation?

15.6 Emit machine code

- Limited to simple types like integers?
- Use LLVM?
- Reuse Cython or numba?
- Replace bytecode with C functions calls. Ex: instead of PyNumber_Add(a, b) for a+b, emit PyUnicode_Concat(a, b), long_add(a, b) or even simpler code without unbox/box
- Calling convention: have two versions of the function? only emit the C version if it is needed?
  - Called from Python: Python C API, `PyObject* func(PyObject *args, PyObject *kwargs)`
  - Called from C (specialized machine code): C API, `int func(char a, double d)`
  - Version which doesn't need the GIL to be locked?
- Option to compile a whole application into machine code for proprietary software?

15.6.1 Example of (specialized) machine code

Python code:

```python
def mysum(a, b):
    return a + b
```

Python bytecode:

```
0 LOAD_FAST                0 (a)
3 LOAD_FAST                1 (b)
6 BINARY_ADD
7 RETURN_VALUE
```

C code used to executed bytecode (without code to read bytecode and handle signals):

```
/* LOAD_FAST */
{
    PyObject *value = GETLOCAL(0);
    if (value == NULL) {
        format_exc_check_arg(PyExc_UnboundLocalError, ...);
        goto error;
    }
    Py_INCREF(value);
    PUSH(value);
}

/* LOAD_FAST */
{
    PyObject *value = GETLOCAL(1);
    if (value == NULL) {
        format_exc_check_arg(PyExc_UnboundLocalError, ...);
        goto error;
    }
    Py_INCREF(value);
    PUSH(value);
}

/* BINARY_ADD */
{
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    if (PyUnicode_CheckExact(left) &&
             PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(left, right, f, next_instr);
        /* unicode_concatenate consumed the ref to v */
    }
    else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
}

/* RETURN_VALUE */
{
    retval = POP();
    why = WHY_RETURN;
    goto fast_block_end;
}
```

Specialized and simplified C code if both arguments are Unicode strings:

```
/* LOAD_FAST */
PyObject *left = GETLOCAL(0);
if (left == NULL) {
    format_exc_check_arg(PyExc_UnboundLocalError, ...);
    goto error;
}
```

(continues on next page)

(continued from previous page)

```
Py_INCREF(left);

/* LOAD_FAST */
PyObject *right = GETLOCAL(1);
if (right == NULL) {
    format_exc_check_arg(PyExc_UnboundLocalError, ...);
    goto error;
}
Py_INCREF(right);

/* BINARY_ADD */
PyUnicode_Append(&left, right);
Py_DECREF(right);
if (sum == NULL)
    goto error;

/* RETURN_VALUE */
retval = left;
why = WHY_RETURN;
goto fast_block_end;
```

15.7 Test if the specialized function can be used

Write code to choose between the bytecode evaluation and the machine code.

Preconditions:

- Check if os.path.isabs() was modified:
  - current namespace was modified? (os name cannot be replaced)
  - namespace of the os.path module was modified?
  - os.path.isabs function was modified?
  - compilation: checksum of the os.py and posixpath.py?
- Check the exact type of arguments
  - x type is str: in C, PyUnicode_CheckExact(x)
  - list of int: check the whole array before executing code? fallback in the specialized code to handle non int items?
- Callback to use the slow-path if something is modified?
- Disable optimizations when tracing is enabled
- Online benchmark to decide if preconditions and optimized code is faster than the original code?

CHAPTER 16

Kill the GIL?

See the *Global Interpreter Lock*.

16.1 Why does CPython need a global lock?

Incomplete list:

- Python memory allocation is not thread safe (it should be easy to make it thread safe)
- The reference counter of each object is protected by the GIL.
- CPython has a lot of global C variables. Examples:
  - `interp` is a structure which contains variables of the Python interpreter: modules, list of Python threads, builtins, etc.
  - `int` singletons (-5..255)
  - `str` singletons (Python 3: latin1 characters)
- Some third party C libraries and even functions the C standard library are not thread safe: the GIL works around this limitation.

16.2 Kill the GIL

- Require deep changes of CPython code
- The current Python C API is too specific to CPython implementation details: need a new API. Maybe the stable ABI?
- Modify third party modules to use the stable ABI to avoid relying on CPython implementation details like reference couting
- Replace reference counting with something else? Atomic operations?

- Use finer locks on some specific operations (release the GIL)? like operations on builtin types which don’t need to execute arbitrary Python code. Counter example: dict where keys are objects different than int and str.

See also pyparallel.

CHAPTER 17

Implementations of Python

17.1 Faster Python implementations

- PyPy
  - AST optimizer of PyPy: astcompiler/optimize.py
- Pyston
- Hotpy and Hotpy 2, based on GVMT (Glasgow Virtual Machine Toolkit)
- Numba: JIT implemented with LLVM, specialized to numeric types (numpy)
- pymothoa uses LLVM ("don't support classes nor exceptions")
- WPython: 16-bit word-codes instead of byte-codes
- Cython

17.2 Fully Python compliant

- PyPy
- Jython based on the JVM
- IronPython based on the .NET VM
- Unladen Swallow, fork of CPython 2.6, use LLVM. No more maintained
  - Project announced in 2009, abandonned in 2011
  - ProjectPlan
  - Unladen Swallow Retrospective
  - PEP 3146
- Pyjion

17.3 Other

- Replace stack-based bytecode with register-based bytecode: old registervm project

17.4 Fully Python compliant??

- psyco: JIT. The author of pysco, Armin Rigo, co-created the PyPy project.

17.5 Subset of Python to C++

- Nuitka
- Python2C
- Shedskin
- pythran (no class, set, dict, exception, file handling, . . .)

17.6 Subset of Python

- pymothoa: use LLVM; don't support classes nor exceptions.
- unpython: Python to C
- Perthon: Python to Perl
- Copperhead: Python to GPU (Nvidia)

17.7 Language very close to Python

- Cython: "Cython is a programming language based on Python, with extra syntax allowing for optional static type declarations."
  - based on Pyrex

CHAPTER 18

Benchmarks

- speed.pypy.org: compare PyPy to CPython 2.7 (what about Python 3?)
- Intel Language Performance (CPython, 2.7 and default branches)
- CPython benchmarks, come from Unladen Swallow?

See also:

- Python benchmark sizes

CHAPTER 19

Random notes about PyPy

19.1 What is the problem with PyPy?

PyPy is fast, much faster than CPython, but it's still not widely used by users. What is the problem? Or what are the problems?

- Bad support of the *Python C API*: PyPy was written from scratch and uses different memory structures for objects. The cpyext module emulates the Python C API but it's slow.
- New features are first developped in CPython. In january 2016, PyPy only supports Python 2.7 and 3.2, whereas CPython is at the version 3.5. It's hard to have a single code base for Python 2.7 and 3.2, Python 3.3 reintroduced `u'...'` syntax for example.
- Not all modules are compatible with PyPy: see PyPy Compatibility Wiki. For example, numpy is not compatible with PyPy, but there is a project under development: pypy/numy. PyGTK, PyQt, PySide and wxPython libraries are not compatible with PyPy; these libraries heavily depend on the Python C API. GUI applications (ex: gajim, meld) using these libraries don't work on PyPy :-(Hopefully, a lot of popular modules are compatible with PyPy (ex: Django, Twisted).
- PyPy is slower than CPython on some use cases. Django: "PyPy runs the templating engine faster than CPython, but so far DB access is slower for some drivers." (source of the quote)

If I understood correctly, Pyjston will have same problems than PyPy since it doesn't support the Python C API neither. Same issue for Pyjion?

CHAPTER 20

Talks

Talks about Python optimizations:

- *PyCon UK 2014 - When performance matters … <http://www.egenix.com/library/presentations/PyCon-UK-2014-When-performance-matters/>* by Marc-Andre Lemburg

CHAPTER 21

Links

21.1 Misc links

- "Need for speed" sprint (2006)
- ceval.c: use registers?
  - Java: Virtual Machine Showdown: Stack Versus Registers (Yunhe Shi, David Gregg, Andrew Beatty, M. Anton Ertl, 2005)
  - Lua 5: The Implementation of Lua 5.0 (Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, 2005)
  - Python-ideas: Register based interpreter
  - unladen-swallow: ProjectPlan: "Using a JIT will also allow us to move Python from a stack-based machine to a register machine, which has been shown to improve performance in other similar languages (Ierusalimschy et al, 2005; Shi et al, 2005)."
- Use a more efficient VM
- WPython: 16-bit word-codes instead of byte-codes
- Hotpy and Hotpy 2: built using the GVMT (The Glasgow Virtual Machine Toolkit)
- Search for Python issues of type performance: http://bugs.python.org/
- Volunteer developed free-threaded cross platform virtual machines?

21.2 Other

- ASP: ASP is a SEJITS (specialized embedded just-in-time compiler) toolkit for Python.
- PerformanceTips