Python Interpreter Performance Deconstructed

Gergö Barany
Institute of Computer Languages
Vienna University of Technology
gergo@complang.tuwien.ac.at

ABSTRACT

The Python programming language is known for performing poorly on many tasks. While to some extent this is to be expected from a dynamic language, it is not clear how much each dynamic feature contributes to the costs of interpreting Python. In this study we attempt to quantify the costs of language features such as dynamic typing, reference counting for memory management, boxing of numbers, and late binding of function calls.

We use an experimental compilation framework for Python that can make use of type annotations provided by the user to specialize the program as well as elide unnecessary reference counting operations and function lookups. The compiled programs run within the Python interpreter and use its internal API to implement language semantics. By separately enabling and disabling compiler optimizations, we can thus measure how much each language feature contributes to total execution time in the interpreter.

We find that a boxed representation of numbers as heap objects is the single most costly language feature on numeric codes, accounting for up to 43 % of total execution time in our benchmark set. On symbolic object-oriented code, late binding of function and method calls costs up to 30 %. Redundant reference counting, dynamic type checks, and Python's elaborate function calling convention have comparatively smaller costs.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Interpreters*

Keywords

interpreters, dynamic programming languages, unboxing, reference counting, Python

1. MOTIVATION

Python is a popular programming language, but it has a reputation of being slow. Projects such as the PyPy just-in-time compiler [BCFR09] show that it is possible to execute some Python programs up to 50 times faster while remaining faithful to its semantics.

Why, then, is the standard Python interpreter (often called CPython) so slow? Is it due to interpretation overhead? Due to dynamic typing and late binding of operations? Due to the boxing of numbers, or due to the overhead of automatic memory management by reference counting?

We attempt to answer some of these questions by measuring the execution times of Python programs with and without enabling the corresponding language features. Simply turning off language features is obviously infeasible in the standard interpreter, but we sidestep this problem in a novel way by using a compiler to specialize programs while still running them inside the interpreter. Type annotations from the user and simple program analysis allow us to turn off language features where it is safe to do so, but to still adhere to Python's dynamic semantics otherwise.

This kind of analysis has several advantages over implementing dynamic optimizations in the interpreter and measuring their impact. First, relying on annotations is simpler than performing static or dynamic type checking; second, we obtain results that quantify *only* the overheads of the language features of interest, without incurring unknown costs of the guards needed when performing dynamic program specialization; third, we do not suffer from the perturbations and some other complications we would have to expect if we tried to measure the costs of dynamic language features by instrumenting the Python interpreter with timing code.

Our results show that dynamic typing and especially boxing of arithmetic operands and containers can easily account for more than half of the total execution time of programs that perform a large number of numeric computations. For programs written in an object-oriented style, with large numbers of method calls, late binding of such calls can take up 20 % and more of total time. Naïve reference counting, a feature for which Python is often criticised, shows a uniform but small impact of up to 7 % of execution time.

Besides the concrete data we obtained, the main contribution of this paper is a general technique for quantifying the costs of language features in dynamic programming languages. While our implementation and the findings are specific to Python, the general approach is applicable to any interpreted dynamic programming language that is capable of calling arbitrary foreign code.

The rest of the paper is structured as follows. Section 2 explains our approach to simulating the Python interpreter's behavior using compilation. Section 3 describes our experi-

To appear in Dyla'14, Edinburgh, UK, June 2014.

```
LOAD_FAST i
LOAD_FAST j
BINARY_SUBTRACT
```

(a) Bytecode

```
Py_INCREF(i)
Py_INCREF(j)
PyNumber_Sub(i, j)
↳ long_sub(i, j)
  ↳ sub %i, %j
Py_DECREF(i)
Py_DECREF(j)
```

(b) Python interpreter

```
PyNumber_Sub(i, j)
↳ long_sub(i, j)
  ↳ sub %i, %j
```

(c) With optimized reference counting

```
Py_INCREF(i)
Py_INCREF(j)
long_sub(i, j)
↳ sub %i, %j
Py_DECREF(i)
Py_DECREF(j)
```

(d) With static type information

```
sub %i, %j
```

(e) Unboxed

Figure 1: Evaluation of the Python expression `i-j` on integers.

mental evaluation and the results we obtained on the cost of dynamic language features in Python. Section 4 discusses related work, Section 5 mentions possibilities for future work, and Section 6 concludes.

2. COMPILED INTERPRETER SIMULATION

Our method for evaluating different aspects of an interpreter's performance is based on generating compiled program parts that simulate the internal API calls executed by the interpreter. The following sections describe how Python uses function calls to implement much of the language semantics, and how we can simulate the same or different sequences of function calls with the same externally observable behavior.

2.1 Dynamic overheads in Python: example

CPython is a stack-based bytecode interpreter written in C using dispatch based on token threaded code. All program data, including numbers, is manipulated through pointers to heap structures that share the common `PyObject` interface. There is no additional static type information. All operations, including basic arithmetic, are dispatched dynamically based on operand types. Automatic memory management is based primarily on reference counting, with a mark and sweep garbage collector as a backup to collect cyclic objects.

This simple architecture is reflected in the internal API used by the interpreter: The implementations of most bytecodes are simple sequences of reference count increment and decrement operations for operands and a single call to an API function. Figure 1 shows an example bytecode snippet for computing the expression `i-j` in Figure 1(a) and the corresponding sequence of API calls executed by the interpreter in Figure 1(b). (The `Py_INCREF` and `Py_DECREF` operations are actually macros that expand to inline code, not function calls.) For this example, `i` and `j` are assumed to be local variables of Python's arbitrary-precision integer type, which is called `int` at the language level but, for historical reasons, `long` within the interpreter. Accordingly, the generic `PyNumber_Subtract` function internally dispatches to the `long_sub` function that performs the actual operation. In the fastest case, this operation amounts to a single CPU `sub` instruction. We write `%i` and `%j` for the CPU registers that hold the numeric values of the objects `i` and `j`, respectively.

The interpreter executes each bytecode instruction in isolation. However, interesting optimization opportunities appear by viewing just a few bytecodes as a group. In the example, the reference counting operations on the subtraction's operands are obviously redundant: Since `i` and `j` are local variables, they have non-zero reference counts when entering this piece of code. The subtraction operation does not consume reference counts. Thus even if the reference count increment/decrement pairs are removed as in Figure 1(c), neither operand will be inadvertently deallocated.

Alternatively, a typing oracle might inform us that the subtraction's operands will always be integers when this snippet is executed. In that case it is not necessary to go through the generic subtraction instruction, and a direct call to `long_sub` can be used to implement the subtraction. This transformation is shown in Figure 1(d). The latter two optimizations can be combined to avoid both reference counting and dynamic dispatch (not shown here). Finally, if the oracle guarantees that the two integers will always fit into machine registers, the operation can be completely unboxed and executed as a single CPU instruction as in Figure 1(e).

2.2 Other dynamic overheads

Besides boxed arithmetic and reference counting, other overheads of interest to us that appear as sequences of API calls in the Python interpreter include the boxing of containers, the dynamic allocation of call stack frames on the heap, and the late binding of function calls from the global scope or from object attributes (i. e., method calls).

By *boxing of containers* we mean primarily accesses to Python lists and arrays. Python's lists are indexable consecutive heterogeneous sequences (vectors) of boxed objects. Arrays are homogeneous vectors of unboxed numbers. When accessing a list or array by numeric index, the index must be unboxed to perform the address computation to access the underlying memory; for arrays, additionally, numbers must be type checked and unboxed on stores, and boxed on loads. Both list and array accesses also perform bounds checks on the index.

For handling coroutines ('generators'), for certain introspective programming techniques, and for easier stack unwinding on exceptions, the Python interpreter manages a *dynamically allocated call stack* on the heap. Each call from a Python function to another causes a frame to be allocated dynamically, and positional arguments are passed by copying them into the frame's variable slots. These frames are destroyed on function return. If instead of returning, the function executes a `yield` statement, its execution is suspended and can be resumed later. In this case, its execution state is encapsulated in its stack frame, which can not be deleted yet. For this reason, stack frames must be allocated dynamically. The cost to support coroutines, which are rare,

must thus be paid in full for every function call, which are frequent.

Python's highly dynamic nature also extends to function calls, which typically use *late binding*. Python uses dynamic scoping, so every use of a global object incurs the overhead of a lookup in the global hash table for the current context (plus another lookup in the builtins table if the global lookup misses). Further, method calls are always virtual and are passed on the stack in a boxed representation as a pair of a function and a `self` pointer. Boxing/unboxing of methods alone can account for about 10 % of highly object-oriented programs' execution time [Bar13].

2.3 Compiling Python functions with `pylibjit`

To evaluate the costs of dynamic language features, we need to be able to execute Python programs with certain features turned on and off; further, since we want to quantify the costs of features *as implemented by CPython*, our programs must be executed within (or at least linked to) the CPython runtime. Finally, modifying the interpreter directly to turn features on and off would be both difficult and possibly plagued by second-order effects due to orthogonal issues such as the ordering of bytecode implementations in the interpreter [MG11].

For all these reasons, we decided to use `pylibjit`, a framework for compiling Python program fragments to run on top of the Python interpreter [Bar14]. `pylibjit` uses function annotations ('decorators') to mark functions to be compiled to machine code and to provide type annotations for arguments, the return value, and variables occurring in the function. For example, the naïve Fibonacci function in Python can be compiled to machine code using unboxed arithmetic by simply decorating it with appropriate annotations:

```
@pyjit.compile(return_type=jit.Type.int,
               argument_types=[jit.Type.int])
def fib_int(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Due to unboxing, this compiled version runs about 50 times as fast as the interpreted version. Note that the Python source code was not changed and is still parsed by the Python interpreter; the annotation can be removed at any point to switch back to purely interpreted execution. Type annotations can name unboxed machine types, Python's boxed number types `float` and `int`, list or array types, or any Python class name. Using these annotations, the compiler is able to specialize the code's behavior as described below.

In any case except for unboxed operations, the compiled code implements Python semantics by calling the same API functions that the interpreter also uses.

2.4 `pylibjit` Optimizations

The optimizations performed by `pylibjit` on request aim directly at eliminating the sources of overheads discussed above.

Reference counting optimization removes unnecessary reference count increment/decrement operations. Pairs of such operations on the same object within a basic block, without intervening operations that might reduce the reference count, are redundant; they can be identified and eliminated at compile time. Reference counting operations that are not removed are generated as inlined code, as in CPython (which uses C macros for inlining).

Static dispatch of arithmetic operations relies on user-provided type annotations to identify arithmetic on boxed `int` or `float` objects and generate calls directly to the type-specific functions rather than the generic `PyNumber` functions that perform dynamic dispatch at run time.

Unboxing of numbers uses type annotations to directly use machine operations on integers and floating-point numbers that fit into machine registers. This avoids storing every number in a heap object and managing that memory.

As a special case, this also turns loops of the very common form `for i in range(n)`, which would normally use an iterator to enumerate boxed numbers from 0 to `n`, into unboxed counting loops if `i` is an unboxed machine integer.

Unboxing of containers, also driven by type annotations, specializes read and write accesses to `list` and `array` objects and elides bounds checks if the index is an unboxed machine integer. (`pylibjit` does not currently have a switch to separate unboxing of containers from bounds checking.) Boxing/unboxing operations on array accesses are also eliminated if the operand is otherwise used unboxed.

Call stack frame removal executes calls between any two compiled functions directly and uses the machine's calling convention rather than dynamically allocated stack frames. This means that we cannot compile coroutines or catch exceptions, but these cases do not occur in our benchmarks, so handling the frames is unnecessary overhead.

Early binding of function calls resolves the addresses of compiled or built-in functions or methods at compile time (guided by type annotations for methods' receiver objects) and generates direct CPU call instructions.

Most of these optimizations are orthogonal in that they can be enabled and disabled independently of each other. This allows us to isolate and study language features such as the tightly interwoven but separable issues of dynamic typing and boxing of numbers.

However, there are relevant interactions between optimizations that influence their respective behavior. For example, optimization of reference counting operations applies in fewer cases if entire boxed objects are optimized away by unboxing. As another example, unboxing of containers is almost useless if numbers are not unboxed because direct memory accesses are only generated if the index is an unboxed number. In this case, only accesses with constant indices would be unboxed.

3. EXPERIMENTS

Building on the infrastructure described above, we evaluated the dynamic overheads of the Python language features discussed in Sections 2.1 and 2.2.

Table 1: Benchmark characteristics

Benchmark	Types of computations
`crypto_pyaes`	small int arithmetic, array operations, method calls
`fannkuch`	list manipulations, method calls
`float`	float arithmetic, attribute accesses, method calls
`go`	list accesses, method calls
`hexiom2`	list accesses, method calls
`meteor-contest`	list manipulations, function calls
`nbody-modified`	float arithmetic, list accesses
`pidigits`	arbitrary-precision int arithmetic
`richards`	attribute accesses, method calls
`scimark_fft`	float arithmetic, array accesses
`scimark_lu`	float arithmetic, array accesses, method calls
`spectral_norm`	small int/float arithmetic, function calls

3.1 Experimental setup

The basis of the evaluation was Python 3.2.3 on an Intel Atom N270 CPU running Linux 3.2.0. We used a number of benchmarks from the PyPy benchmark suite[1], which unfortunately contains mainly programs in the Python 2 dialect. Many, but not all, could be translated to Python 3 using the standard `2to3` tool or simple manual transformations. A few benchmarks had to be excluded because the manual annotation overhead to obtain meaningful optimizations with `pylibjit` was too large, or because they used generators (coroutines) or `try-catch-finally` constructs which the compiler cannot handle so far. A single benchmark (`raytrace-simple`) had to be excluded due to an as yet unresolved bug in `pylibjit`.

The remaining benchmarks are listed in Table 1 along with brief descriptions of the types of computations they perform.

We obtained costs for each of the language features under consideration by measuring execution time with and without the feature enabled (wall clock time, as reported by the timing infrastructure used by the benchmark suite). Each configuration was run 10 times on an otherwise idle machine and the minimum taken as the representative time. In our experience, the minimum is very close to the time taken by most runs; any outliers are particularly slow runs, there are no fast outliers.

To measure some of the language features' impacts, we had to implement corresponding 'pessimizations'. For example, to measure the impact of static (boxed) typing of arithmetic, we needed a 'fully dynamic typing' switch which ignores all of the user's number type annotations by rewriting them to refer to the universal type `object`. The compiler is then forced to emit generic `PyNumber` calls to implement any arithmetic operation. As another example, we measure the impact of late binding on function calls by calling the appropriate lookup functions of the API.

As mentioned above, some optimizations interact with each other. In the experiments described below, such interactions were resolved in the way that reflects each language feature's costs most faithfully. In particular, the cost of unnecessary reference counting was evaluated with fully dynamic typing and no unboxing; the cost of container boxing was evaluated with unboxed numbers.

[1] `https://bitbucket.org/pypy/benchmarks/`

3.2 Results

Table 2 summarizes our results. The cost of each language feature is given as the percentage (rounded to whole numbers) of the interpreter's total execution time on the program without any compilation whatsoever. The absolute cost of a language feature f is obtained by benchmarking two copies of the program, one with f enabled and one with it disabled. For example, to measure the costs of the boxing of numbers, we first measure execution times with boxed numbers (but also with static type checking and reference counting elision enabled), then measure the same programs with numbers unboxed as much as possible. The difference between execution times is just the cost of boxed arithmetic, not including the cost of other related features such as automatic memory management or dynamic type checks.

If executing a program in the interpreter takes time t_i, compiling it with some costly dynamic feature f enabled takes time t_f and compiling it without f takes $t_{\neg f}$, then the fraction given in the table is

$$\frac{t_f - t_{\neg f}}{t_i}.$$

The numerator gives the total time spent in implementing dynamic feature f, and the denominator normalizes that to total interpreter execution time. While the numerator is a time difference between *compiled* programs, the extra time is spent in Python interpreter API calls. It is therefore a useful proxy for determining the time the interpreter would spend executing those same operations.

3.3 Discussion

Unsurprisingly, different language features have different effects on the benchmarks. There is, however, some strong clustering depending on benchmark type, especially on the results of number boxing for the very 'numeric' benchmarks `spectral_norm`, `crypto_pyaes`, and `scimark_fft`. The latter two are also heavily based on arrays and incur correspondingly high costs from boxing and bounds checking on containers. `nbody-modified` similarly spends much of its time in list accesses to get at arithmetic operands.

Besides boxing operations, late binding has the largest effects. This is to be expected for object-oriented benchmarks such as `go`, `hexiom2` (both of which play games), and `richards` (a scheduling simulation). The arithmetic-heavy `float` benchmark performs a very large number of method calls and thus also incurs large overheads due to late binding. The 'numeric' `scimark_lu` uses a Python class to implement matrices and therefore also has to perform many late-bound method calls to access matrix elements.

The third group of benchmarks shows less dynamic language feature overhead in the table. This is because these programs spend much of their time in compiled code within the Python standard library, not in the interpreter. `fannkuch` and `meteor-contest` compute on large numbers of list slices and other built-in data structures. `pidigits` spends almost all of its time on arbitrary-precision integer arithmetic that cannot be unboxed. Reference counting has a small impact here, but no other dynamic language feature is relevant because this benchmark exercises only Python's big integer library.

Table 2: Cost of various Python language features and implementation choices as fraction of interpreter execution time.

Benchmark	Ref. counting	Dynamic typing	Number boxing	Container boxing	Call stack	Late binding
`crypto_pyaes`	7 %	5 %	34 %	24 %	1 %	3 %
`fannkuch`	5 %	3 %	1 %	5 %	0 %	0 %
`float`	0 %	1 %	32 %	0 %	2 %	30 %
`go`	6 %	2 %	2 %	0 %	10 %	16 %
`hexiom2`	3 %	0 %	2 %	3 %	8 %	18 %
`meteor-contest`	5 %	0 %	1 %	0 %	2 %	4 %
`nbody-modified`	2 %	15 %	19 %	12 %	0 %	5 %
`pidigits`	1 %	0 %	0 %	0 %	0 %	0 %
`richards`	7 %	0 %	0 %	0 %	10 %	23 %
`scimark_fft`	7 %	9 %	38 %	24 %	0 %	0 %
`scimark_lu`	2 %	1 %	6 %	6 %	0 %	16 %
`spectral_norm`	5 %	15 %	43 %	0 %	6 %	3 %

From the point of view of individual language features, unnecessary reference counting has the least variation in cost of 0–7 % of total execution time. Interpreter optimizations that remove the low-hanging fruit of unnecessary reference counting operations could hope for small but uniform performance improvements. Somewhat less uniform but much larger gains could be expected from optimizing method calls, possibly due to more aggressive caching of dynamic method lookup results. Optimizing the dynamic call stack would improve a few benchmarks but not make a difference on most of them.

4. RELATED WORK

There is a large body of work on optimizing various aspects of the Python interpreter, such as type speculation and unboxing of numbers [Bru10a, Bru10b, Bru13], optimization of method calls [Bar13, MKC+10], or more general interpreter optimizations [HH09, PR13]. These papers optimize one or more dynamic language features, typically inserting guards along the way to catch cases of mis-speculation. However, these papers, with the recent exception of Brunthaler [Bru13], do not attempt to quantify the *potential* impact of their optimizations: How fast could their programs become with perfect speculation, i. e., without any guards? In this sense the present paper differs from all of the above by trying to quantify the costs of dynamic language features rather than the benefits of concrete optimizations.

Similarly, this paper does not quite fit into the literature on Python compilers. Some, like PyPy [BCFR09] or Cython [Sel09], reimplement Python to compile to standalone executables. This is very useful to maximize performance and highlight that it is possible to execute Python code very quickly, but it does not shed light on where in the interpreter how much of that performance is lost. Others, like numba [num], are more similar to the `pylibjit` compiler we use [Bar14] in that they also run compiled code inside the interpreter, but again with a focus on performance, not on faithful modeling of Python language semantics.

5. FUTURE WORK

The results presented above suggest two primary avenues of research on optimizing Python, namely a focus on number unboxing and on method lookups, respectively. Both can be addressed fully dynamically to some extent: Brunthaler [Bru13] describes a Python interpreter that uses dynamic type information to generate type-specialized, unboxed code at run time and achieves speedups of up to 4 ×. Barany [Bar13] addresses a sub-problem of method lookups, namely the fact that due to Python's calling convention, method objects must be passed around in a boxed representation. This boxing can often be eliminated, giving speedups of up to about 10 %.

Both of these optimizations work with completely unmodified Python code. However it appears that both of these optimizations, as well as the actual process of dynamically looking up methods, would profit from more type information. Python's flexible syntax allows such information to be expressed as decorators or as annotations in a function's head, which makes it an interesting target for gradual typing, i. e., the mixing of static and dynamic types within a single program [VSB]. Gradual typing information can be used to perform unboxing and early binding without the guards needed if only dynamic type information is available.

Finally, our measurements of the costs of language features can be refined further to gain even more insight into optimization potentials. In particular, the costs of 'late binding' should be separated into the late binding of global identifiers vs. object attributes, and in the case of method lookups, the cost of method boxing should be determined in more detail.

6. CONCLUSIONS

We have presented the first limit study that tries to quantify the costs of various dynamic language features in Python. The results show that for arithmetic-heavy programs, boxing of operands and containers is the largest performance bottleneck; dynamic type checks themselves (separate from boxing) have a comparatively smaller impact.

For programs heavy on method calls, lookup and boxing of methods are the most costly language feature, but the calling convention involving dynamic allocation of call stack frames is also sometimes relevant.

The basic technique of simulating interpreters by compilation is applicable to interpreters for any dynamic programming language and should enable their developers to gain similar insights.

7. REFERENCES

[Bar13] Gergö Barany. Static and dynamic method unboxing for python. In *6. Arbeitstagung Programmiersprachen (ATPS 2013)*, number 215 in Lecture Notes in Informatics, 2013.

[Bar14] Gergö Barany. `pylibjit`: A JIT compiler library for Python. In *Software Engineering 2014 (Workshops)*, number 1129 in CEUR Workshop Proceedings, 2014.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[Bru10a] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 1–14, New York, NY, USA, 2010. ACM.

[Bru10b] Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 429–451. Springer, 2010.

[Bru13] Stefan Brunthaler. Speculative staging for interpreter optimization. *CoRR*, http://arxiv.org/abs/1310.2300, 2013.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In Bernard Mans, editor, *ACSC*, volume 91 of *CRPIT*, pages 17–25. Australian Computer Society, 2009.

[MG11] Jason McCandless and David Gregg. Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures: Or: How I learned to stop worrying and love hill climbing. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 161–170. ACM, 2011.

[MKC+10] Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelsohn, Priya Nagpurkar, and Peng Wu. Understanding the potential of interpreter-based optimizations for python. Technical Report 2010-14, UCSB, 2010.

[num] numba: NumPy aware dynamic Python compiler using LLVM. https://github.com/numba/numba.

[PR13] Russell Power and Alex Rubinsteyn. How fast can we make interpreted Python? *CoRR*, http://arxiv.org/abs/1306.6047, 2013.

[Sel09] Dag Sverre Seljebotn. Fast numerical computations with Cython. In *Proceedings of the 8th Python in Science conference (SciPy 2009)*, 2009.

[VSB] Michael M. Vitousek, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. http://wphomes.soic.indiana.edu/jsiek/files/2014/03/retic-python.pdf. [Online draft; accessed April 4, 2014].

Acknowledgements

This work was funded in part by the Austrian Science Fund (FWF) under contract no. P23303, *Spyculative*.

The author would like to thank the anonymous reviewers for their helpful comments.