Evaluating the dynamic behaviour of Python applications

Alex Holkner **James Harland**

School of Computer Science
RMIT University,
Australia,
Email: {alexander.holkner,james.harland}@rmit.edu.au

Abstract

The Python programming language is typical among dynamic languages in that programs written in it are not susceptible to static analysis. This makes efficient static program compilation difficult, as well as limiting the amount of early error detection that can be performed. Prior research in this area tends to make assumptions about the nature of programs written in Python, restricting the expressiveness of the language. One may question why programmers are drawn to these languages at all, if only to use them in a static-friendly style. In this paper we present our results after measuring the dynamic behaviour of 24 production-stage open source Python programs. The programs tested included arcade games, GUI applications and non-interactive batch programs. We found that while most dynamic activity occurs during program startup, dynamic activity after startup cannot be discounted entirely.

Keywords: Dynamic languages, Python and Compilers.

1 Introduction

Many of the features of a dynamic language such as Python that make it so appealing for rapid development and prototyping make traditional static code analysis intractable. This means that programs written in such a language cannot be checked for type safety, compiled down to efficient machine code or run without supervision in a sandboxed environment.

This problem is typically approached by making assumptions about the source code, essentially restricting the expressiveness of the language to make it susceptible to static analysis. It may be that these assumptions are reasonable and correct for a large number of programs. For example, Aycock (2000) writes, "Giving people a dynamically-typed language does not mean that they write dynamically-typed programs."

A compelling alternative is RPython (Ancona et al. 2007), which allows the full use of Python up to a predetermined entry point (typically the *main* function), after which only a restricted subset of Python is permitted. We suppose that the developers of RPython would argue that most uses of dynamic features are required only during program startup, while classes are being initialised, after which only non-dynamic features are required. RPython may be a useful general purpose programming language providing similar expressiveness to Python, but only if this argument is justifiable.

We investigated the validity of these assumptions by recording the dynamic activity of a range of open source applications written in Python. By measuring the actual run-time behaviour over a sample of programs we hope to provide evidence to evaluate the following (somewhat contradictory) anecdotal hypotheses:

1. That programs written in Python generally do not make use of dynamic features, or that if they do, they can be trivially rewritten in a more static style.
2. That while programs written in Python use dynamic features, they do so mostly during program startup, and afterwards behave like a statically-compiled program.

Our results show that many of the tested programs actually do make use of dynamic features throughout their lifetime. The majority of this use is during program startup, after which programs behave relatively statically. We believe that many traditional static analyses may be feasible in Python at run time after startup. Nevertheless, we believe that even RPython's approach is too restrictive, as all programs we analyse exhibit some dynamic behaviour after startup.

The following section details the specific features we count as dynamic in our tests. Section 3 describes how the tests were constructed, and the types of programs examined. We present our early results in Section 4, cover the related material in Section 5 and make our concluding remarks in Section 6.

2 Background

We broadly categorise the dynamic features of Python into four sets: *reflection*, *dynamic typing*, *dynamic objects* and *dynamic code*.

Reflection refers to the use of an object using meta-object facilities, such as getting and setting an attribute by name, invoking a method by name, and inspecting an object and its class. Reflection is not unique to dynamic languages; for example, all of these facilities are available in Java, and some are also possible in C++. Python differs from these languages only in the ease of use of reflection, which possibly encourages its use.

Dynamic typing refers to the fact that variables in Python are not given a type; instead, they take the type of whichever object is currently assigned to them. Some uses of dynamic typing can be trivially reduced to static typing with type inference, and union types can be introduced to deal with cases where a variable takes on multiple types (Anderson

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Thirty-Second Australasian Computer Science Conference (ACSC2009), Wellington, New Zealand. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 91. Bernard Mans, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

et al. 2003). Dynamic types are gradually being introduced into other languages, both as placeholders for variables whose type is inferred at compile time, and as actual run-time dynamic types (Abadi et al. 1991, Hamilton 2006).

A Python program makes no type declarations: classes are constructed at run time before use. This complicates static type inference schemes for Python; typically some class declarations are extracted from the program source code, though this is not comprehensive and does not match the semantics of the language. Further complicating matters, both classes and objects can undergo almost any mutation at run time; for example, objects can change class, classes can change base classes, additional methods and attributes can be added to objects and classes, and existing methods and attributes can be modified or even deleted from a class or object. We refer to these issues regarding the Python object model as *dynamic objects*.

Finally, a common feature of all dynamic languages, including Python, is the ability to construct code at run time from source code. For example, in Python this is provided by the *eval* function and *exec* statement. Modules can also be imported by name, which is a related functionality. Clearly the type, validity and security of such an evaluation can never be discovered ahead of run time. We call this feature *dynamic code*.

Each of the specific dynamic features described in the next section falls into one of these categories.

3 Methodology

Our experiments consisted of recording the dynamic activity of a number of Python applications while they were running. We began by selecting a small selection of programs suitable for instrumentation. These programs were run in a modified Python interpreter that collected information on the use of 14 specific dynamic features.

3.1 Program selection

We required a number of standalone open source Python applications (i.e., programs with accompanying source code that would run without additional programming) for our analysis. Ideally these applications would be diverse in their type, authorship, style and use of frameworks and libraries. We used the Python Package Index (`http://pypi.python.org`), which lists over 4,500 packages. Of these, around 1000 are marked as being in a production or stable state. We filtered this list to those made for an X11 or console environment under Linux, and targeted towards end-users rather than developers. These decisions were made either arbitrarily, to reduce the size of the sample set, or to specifically to expediate the testing process. This left us with around 50 packages. After discarding packages that were not standalone or would not run in our test environment, we were left with 24 programs.

These programs cover a broad spectrum of application domains. We expected that our results would differ significantly between different types of programs, so we classified the applications according to the nature of interaction they require. The classifications are:

Game 6 programs that are graphics-intensive and require large amounts of processing time. These programs do not require input to continue processing, as they are largely driven by the system clock.

Interactive 12 programs that require interactive user input to run. These included several GUI applications, a web server and a text-based application.

Non-interactive 6 programs that take a single input file and then run to completion.

Since the execution of all of these programs is dependent on the inputs they are given, we tried to run them in "typical" scenarios. For the interactive applications, this meant interacting with them (opening and editing files, saving them, opening preference dialogs, and so on) for some time before exiting normally. The games were run for less than a minute, as the data recorded from these quickly became difficult to manage. The results we collected suggest that this time limit does not affect the run-time profile. The non-interactive programs were given a single medium-sized input file.

3.2 Dynamic features

The Python virtual machine operates on a machine-independent stack-based bytecode stream, which governs the execution of the program. For example, there are bytecodes to push and pop values on and off the stack, to load constants and local variables onto the stack, to perform attribute accesses and function calls, and so on.

Our analysis was at this bytecode level. As each bytecode is interpreted, our code is invoked and checked the specific opcode and current stack for dynamic behaviour. If the operation is classified as dynamic, the current frame of execution is labelled accordingly. We refer to such frames as "dynamic frames."

We chose frames as the level of granularity to record at because many optimisations performed by compilers operate over entire methods. By marking a frame as "dynamic", we suppose that that frame's method would not have been amenable to optimisation for this invocation. Another interesting possibility for analysis would have been to record basic block invocations; however the Python virtual machine does not expose this abstraction, and so would require significantly more work to instrument.

We recorded the following dynamic features:

attr_add An attribute was added to an object outside of its constructor.

attr_del An attribute was deleted from an object.

attr_mutate_generalize An attribute on an object was modified, with the new value having a more general type than the old value. This would not be a type safe operation in a statically-typed language, unless the attribute was typed initially with the new value's type.

attr_mutate_type An attribute on an object was modified, with the types of the new and old values having no common type ancestor.

attr_mutate_none An attribute on an object was modified, with either the old or new values being *None* (roughly equivalent to "null" in other languages).

call_execfile The *execfile* function was called, which dynamically loads and executes another Python source file.

call_reload The *reload* function was called, which reloads a module from its source file to obtain the latest changes.

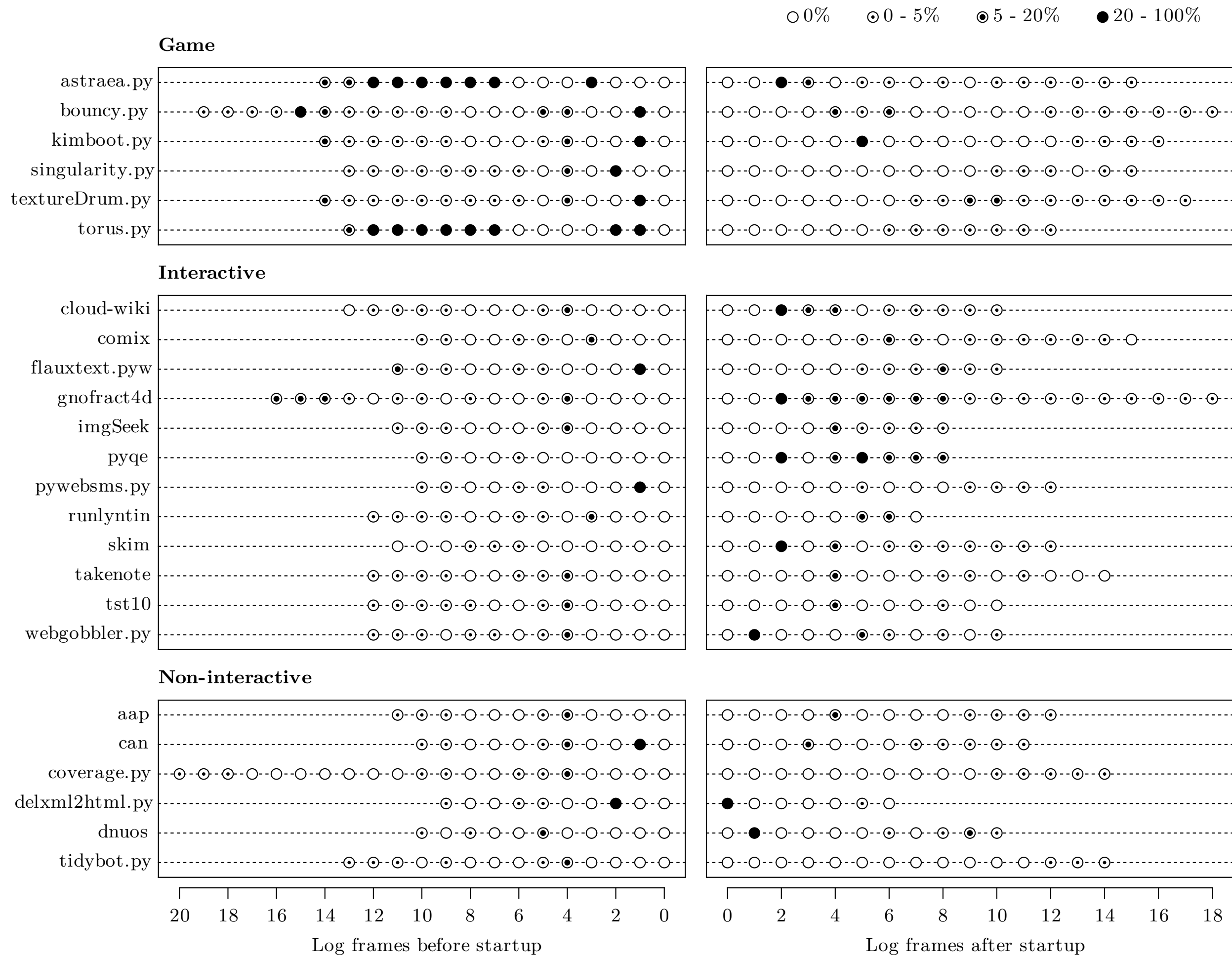


Figure 1: Summary of dynamic activity for all programs. The left and right panels show the startup and run-time phases of each program, respectively. The program's running time is measured in activation frames and runs from right to left along the X axis on a log scale of base 2. Each sample is represented by a circle showing the proportion of dynamic frames within that time period. Darker circles represent parts of the program that have more dynamic activity.

call_getattr The *getattr* function was called, which retrieves an object's attribute by name.

call_setattr The *setattr* function was called, which sets the value of an object's attribute by name.

call_delattr The *delattr* function was called, which deletes an attribute from an object by name.

call_locals The *locals* function was called, which returns a mutable mapping of the current frame's local variables to their values.

call_globals The *globals* function was called, which returns a mutable mapping of the current frame's global variables to their values.

call_eval The *eval* function was called, which evaluates a string as a Python expression.

exec_stmt The *exec* statement was used, which executes a string or file as a Python compound statement.

These features can all be classified as making use of *reflection*, *dynamic typing*, *dynamic objects* or *dynamic execution*. While *attr_mutate_none* is not a type-safe operation in any language, many statically-typed object-oriented languages such as Java and C++ still permit it.

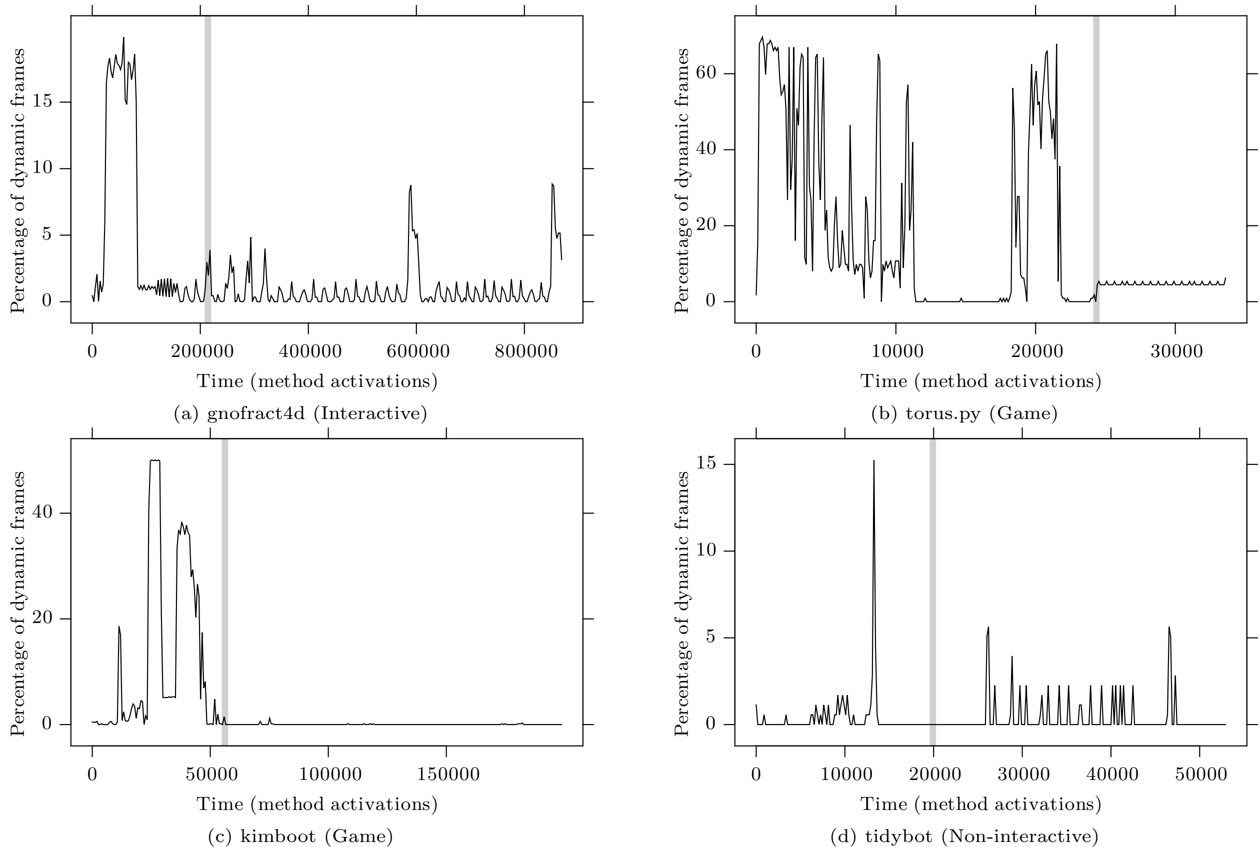
3.3 Instrumentation

In order to instrument the programs, we modified the Python 2.5.2 interpreter to add a bytecode-tracing function. This is similar to the built-in line-tracing function, but operates on individual bytecodes rather than source lines. The function calls back into our instrumentation code—written in Python—with the current frame, VM stack and opcode about to be executed. By examining the values on the stack our program can make a determination about whether a given instruction is dynamic.

Some of the *call_** dynamic activities were recorded by creating function proxies for the corresponding built-in function; however this turned out to be applicable in only a few cases, as many of these functions depend on the lexical scope of the caller.

Due to the run-time overhead in instrumenting, many programs run much slower in our test environment than they would otherwise. This was particularly problematic when measuring the games, as they schedule their behaviour based on wall-clock time. In order to work around this issue we also proxied the *sys.time* function, having it return an artificial time based on the number of instructions processed so far. While the games continued to run quite slowly, they at least behaved correctly over time.

In order to distinguish between dynamic activity during and after startup, we inserted a marker into the source code of each program which could



(a) gnofract4d (Interactive)

(b) torus.py (Game)

(c) kimboot (Game)

(d) tidybot (Non-interactive)

Figure 2: Aggregate dynamic activity over several programs. The progression of time is shown on the X axis, while the proportion of frames for a given sample is shown on the Y axis. A vertical line in each plot shows where the startup phase ends and the main event loop is entered.

be picked up by our instrumentation code. We attempted to be consistent in the placement of this marker between the various styles of programs. For interactive applications and games the marker was inserted just before entry into the main event loop. For the non-interactive programs the marker was inserted after command-line arguments were parsed.

4 Results

Our intention is to investigate the dynamic behaviour of production Python programs, with a view of exploring the anecdotal arguments described in the background. We begin by looking at the overall dynamic profile of all programs, grouped by classification. We then investigate the aggregate run-time dynamic activity of some representative programs in greater detail, and then investigate the use of specific dynamic features. Finally, we introduce the notion of "non-dynamic frame depth," and some preliminary results relating to it.

4.1 Overview

A summary plot of our results is shown in Figure 1. Each line of the plot shows the execution of a program over time, with the left panel showing its startup period and the right panel showing the run-time period. The more filled in a circle is, the higher the proportion of frames in that sample that were found to be dynamic.

The summary shows a range of dynamic behaviours across the programs. Recall that we classified each program according to its interaction protocol. Only the programs classified as games show a clear clustering of dynamic behaviour in the startup phase compared to the run-time phase. In fact, we found that 70% of all tested programs had a higher proportion of dynamic activity in their startup phase than at run time. We note that most of those programs that do not confirm this hypothesis are relatively short-running—70% of them had running times less than the median.

4.2 Dynamic activity over time

Figure 2 shows the dynamic behaviour for a selection of programs. We can see the appearance of regular patterns of dynamic activity in each of the interactive or game programs (Figures 2a–c) in the run-time phase, corresponding to iterations of the run loop or repeated event handling. The non-interactive program (Figure 2d) does not exhibit periodic behaviour. These results are typical of all the programs we tested.

While the aggregate information shown in these plots gives a quick overview of the dynamics of a program, it does not shed any light onto why a program would behave this way. By plotting each dynamic feature separately we can start to see patterns emerge that explain some of the peaks.

For example, Figure 3 shows the detailed run-time behaviour for *gnofract4d*, an interactive program that allows the user to explore fractal images. The program generates C code on the fly according to the current fractal being evaluated. The dynamic activities *call_eval* and *call_getattr* are both used to create and link to the generated code.

Figure 4 shows the detailed behaviour for *torus.py*, a simple demonstration program for a graphics library that displays a spinning torus. Note the high use of reflective and dynamic object features during startup, followed by just the *call_getattr* feature during run time. This run-time dynamic behaviour is due en-

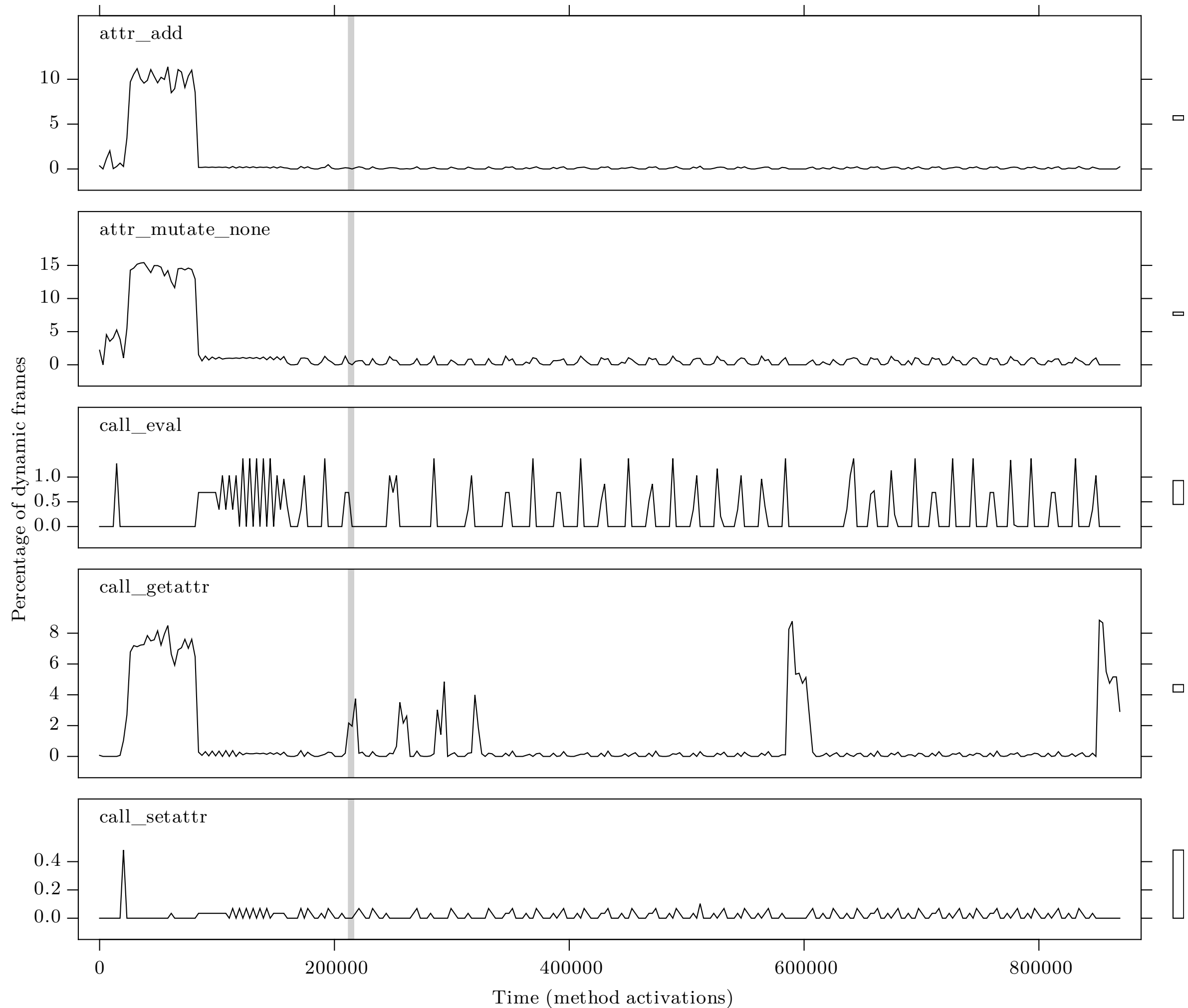


Figure 3: Detailed dynamic behaviour for *gnofract4d*. Each trace shows the proportion of frames having the labelled dynamic feature. The vertical reference line shows the division between startup time and run time. The rectangles on the right side of each trace show the relative scaling; the vertical lengths represent the same percentage on each plot.

tirely to a lazy module loading system in one of the program's dependent libraries, which uses it to resolve a module name at run time. Due to limitations in Python's import mechanism, there is no way to remove this dynamic call without changing the application code or forcing the modules to be loaded eagerly.

In both of these cases it's clear that the programs use dynamic features in ways that cannot be easily translated into non-dynamic code. This indicates that RPython is not a suitable language for these particular programs.

It's also interesting to note in Figure 4 the apparent correlation of dynamic activity during startup; for example, the *call_setattr*, *call_getattr* and *attr_add* series look very similar. This trend was common among the programs, and could be an interesting avenue for future research.

4.3 Use of dynamic features

Table 1 shows the number of programs out of the 24 sampled that used each dynamic feature at least once after startup. We describe the reasons for these uses that we discovered by examining the programs' source code.

attr_add is used by almost all programs during main processing. The high number of programs using this feature is due to it being used in several places in the standard library. While some of these occurrences are due to delayed initialisation (and so it is conceivable that the programmer could have made a simple code modification to provide a default value), there are some occurrences in which the attribute is added to the object ad-hoc. Translating this code to a static language would probably require a named mapping attached to the object, and the corresponding extra syntactical baggage that implies.

Feature	Programs
attr_add	22
attr_del	3
attr_mutate_generalize	4
attr_mutate_type	15
attr_mutate_none	23
call_execfile	0
call_reload	0
call_getattr	21
call_setattr	13
call_delattr	0
call_locals	4
call_globals	6
call_eval	5
exec_stmt	4

Table 1: Number of programs using each measured dynamic feature at least once after program startup.

Most uses of *attr_del* were essentially syntactic

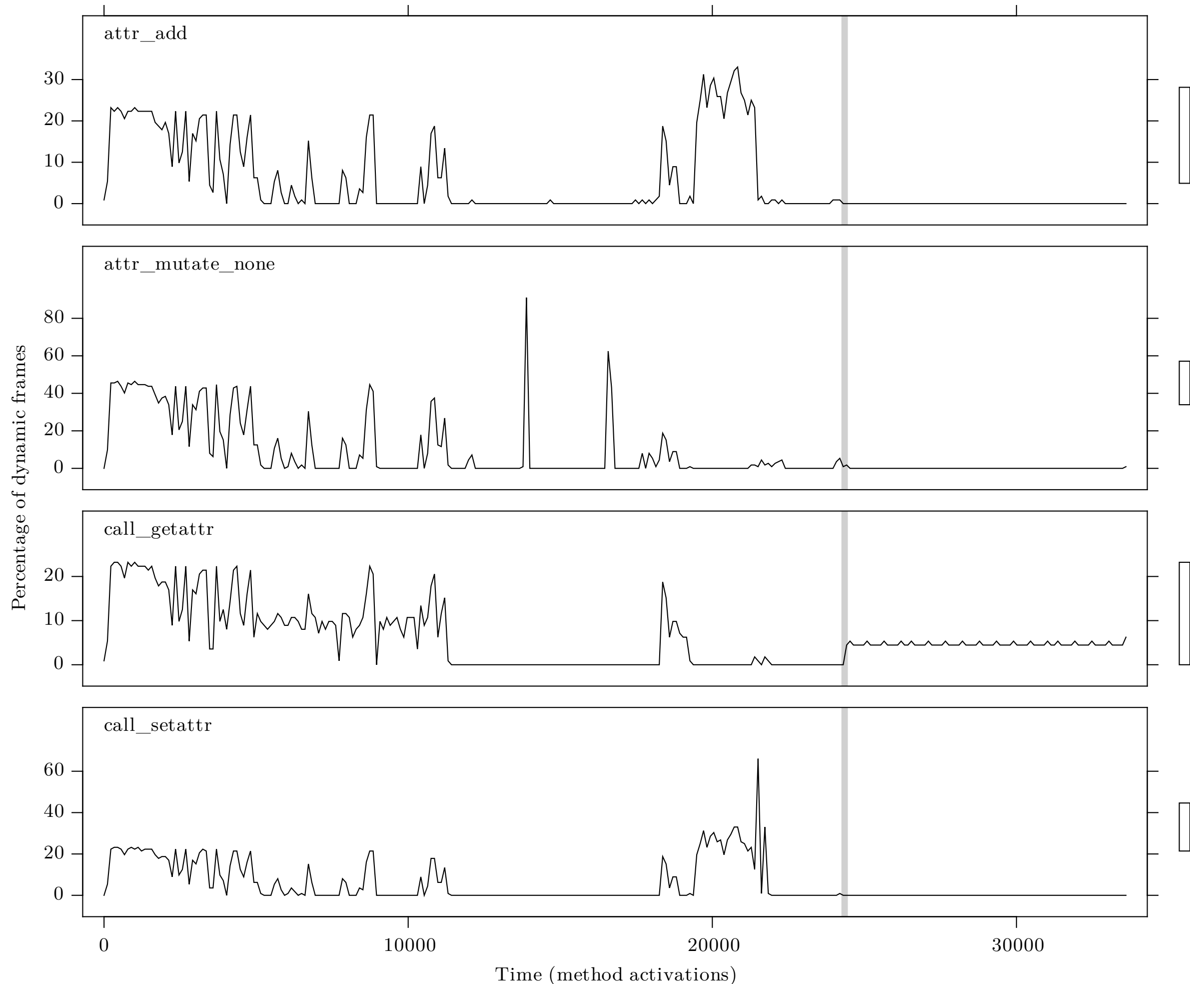


Figure 4: Detailed dynamic behaviour for *torus.py*. Each trace shows the proportion of frames having the labelled dynamic feature. The vertical reference line shows the division between startup time and run time. The rectangles on the right side of each trace show the relative scaling; the vertical lengths represent the same percentage on each plot.

equivalents to setting the attribute to *None*, and so could be discounted from dynamic activity. The only exception was in the standard library, in which a class constructor uses reflection to copy the local variables onto attributes of the object, then deletes an attribute that should not have been copied.

We were surprised to see how many programs used *attr_mutate_type*; some quite often in their main run loop. The use of this feature would normally imply that an object's attribute is changing its interface at run time, which makes reasoning about the program difficult. Upon closer inspection, however, we discovered that the vast majority of these uses were in coercing integers to floats, or 8-bit character strings to Unicode strings, or replacing functions with callable objects (these are distinguished in Python's introspection, but have equivalent semantics). We will try to rule out as many of these special cases as possible in our next evolution of the testing framework.

We did note some other uses of *attr_mutate_type* in a handful of programs, though we suspect that these are actually programming errors. Both *attr_mutate_none* and *attr_mutate_type* were omitted from the summary and aggregate results presented earlier, as they could be seen to artificially boost the dynamic profile of a program.

The use of *call_globals* was mostly found to be in support of *exec_stmt*, or during module initialisation of modules loaded after startup. For the module initialisation case, the feature is used as a tool in metaprogramming, such as for modifying the public interface of the module. In one case *call_globals* was used simply to modify a global variable (Python has an explicit syntax for doing this). In another case it was used to dynamically replace a module's contents with proxy objects to facilitate logging.

call_eval and *exec_stmt* were used to load plugins and for metaprogramming (constructing a string and then evaluating it). The *cloud-wiki* program uses *exec_stmt* to implement a simple embedded language which is used to construct web pages returned to the user.

In summary, we saw cases among the programs where the dynamic feature was used gratuitously, and could be trivially replaced with simpler, static code. A not insignificant number of programs also seemed to require the use of dynamic features, however, and could not have easily been rewritten in a static style.

4.4 Depth of non-dynamic frames

One of the original hypotheses that lead to this investigation was that while programs may make use of dynamic features at run time, there will still be large sections of code that are essentially static, with respect to the environment that has already been created. One could imagine taking some method call site in a program and compiling it "just-in-time", fully knowing the existing type environment and without having to make any assumptions—if no code below

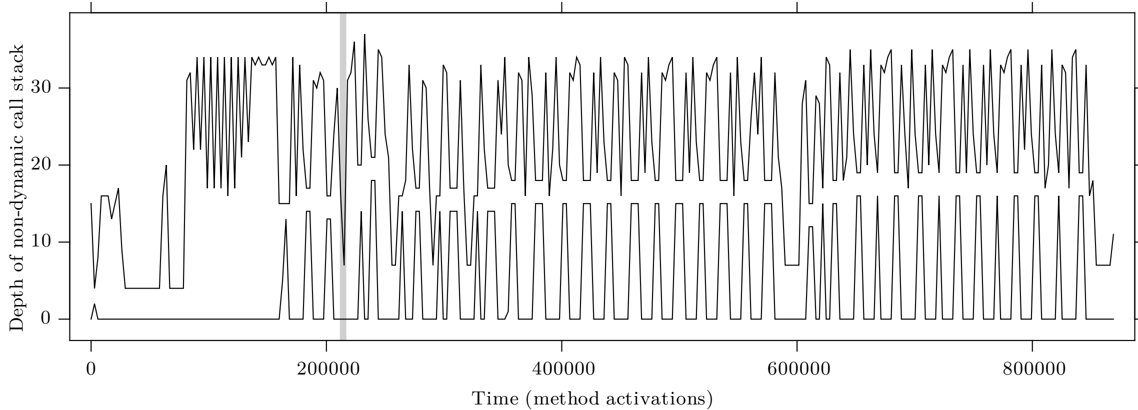


Figure 5: Minimum and maximum depth of non-dynamic frames over time, for the program *gnofract4d*. High values indicate the current call stack has non-dynamic frames for a large number of parent frames. A zero value indicates the frame is dynamic. The vertical reference line shows the division between startup time and run time.

that call site in the call graph uses a dynamic feature, and so could be analysed.

We have begun preliminary testing of this hypothesis by measuring how deep the call stack can go without encountering a dynamic feature. We do this in two phases. First, we mark all frames with a dynamic feature as dynamic (as described in Section 3), and any frame that called a dynamic frame as dynamic. We then traverse up the call stack for each non-dynamic frame until we reach a dynamic frame, counting how many times we can traverse up. If we were able to traverse all the way up to the program entry point, that would indicate the program had zero dynamic frames.

Note that this process runs offline, after all dynamic activity has been recorded. Where a method was entered recursively, only one frame of that method was counted.

We are then able to plot for each program the depth of non-dynamic frames over time. An example is shown in Figure 5. In this particular program the depth of non-dynamic frames varies between 5 and 18. We found that 20 programs had depths of 4 or greater for at least 50% of their frames after startup. All programs had at least one frame with depth 9 or greater, indicating that some static-style analysis could be performed on a non-trivial portion of the program.

4.5 Discussion

The results presented so far should be viewed as preliminary. There is a lot of scope for future investigation in this area. For example, we have not yet measured the use of dynamically-typed variables and functions, and the number of programs tested is small in comparison to the amount of Python code available. Nevertheless, we believe some reasonable conclusions can be drawn.

Firstly, the hypothesis that programs written in Python generally do not use dynamic features is clearly invalidated. Even disregarding dynamic variable types (which we did not test), we found that all programs used non-reflective dynamic features, such as dynamic code execution and using objects dynamically. Any static analysis tool, such as a static type checker or an efficient native code compiler, must therefore be necessarily imprecise in typical programs.

The usefulness of RPython as a replacement language for Python depends on the extent of use of dynamic features after the *main* function has been entered. We have shown that among the programs tested, 70% have less dynamic activity after startup. This supports the supposition that RPython is in some cases well-suited as a replacement language for Python.

However, the use of dynamic features after startup is not insignificant, and in many cases cannot easily be replaced with non-dynamic code. Nearly all programs used reflective features after startup, and over 20% of the programs executed dynamic code. This contrasts with the view perpetuated by static analysis research in Python that these features are rarely used.

We attributed some dynamic object modification (besides reflective modification) to delayed initialisation: i.e., attributes were added to objects outside of their constructor, but still within some clearly indicated initialisation method. Some object modifications are more "dynamic" than others; a change to a class or its inheritance hierarchy is more significant than a change to a single object, from a static analysis point of view. Salib (2004) writes, "in most Python programs, [...] methods are not added after a class object has been created." In our current test framework we are unable to easily identify these situations, but it is something we will investigate in the future.

5 Related work

There is something of a tradition in research for proscribing particular language features: the so-called "...considered harmful" papers (Dijkstra 1968). These papers have a certain relevance to our work, as they ask the question "Which language features are really necessary?" However, while these papers adopt a theoretical or best-practice viewpoint, we look only at the commonality of use of features in existing software.

Research into optimizing compilers has often led to profile data from program runs being fed into the compiler to provide it with more information about the program's expected run-time behaviour (Pettis & Hansen 1990, Gupta et al. 2002, Chang et al. 1992). These so-called profile-guided techniques are used to optimise instruction selection, branch prediction, code placement, function inlining, and of course JIT invocation. Code profiling is also a common technique for optimising and debugging programs by hand. The research into these techniques focusses on using gathered information to make informed choices for optimisation, and has not contributed much knowledge regarding the overall trends of usage of various programming features, though such data could be obtained from such a profiling tool.

One notable exception is the profile-guided receiver-class prediction by Grove et al. (1995). The authors profiled a handful of C++ and Cecil applications to discover the use of polymorphism within these programs, including how that use changed over development time, and over multiple inputs. Their results over the sample of tested programs showed a preference for a small number of receiver classes even in a polymorphic application, which can be exploited for better performance with with a dispatch cache.

Tempero et al. (2008) conducted an empirical study of a large number of Java programs to determine the use of inheritance in practice. Their aims are similar to our own, in that they wish to discover actual usage patterns in published software.

Our work is somewhat related to research into type inference in dynamic languages, as most static analysis and optimisation requires accurate type information for variables. The papers describing these systems are often accompanied by a short study into the use of a particular dynamic feature, though none have investigated the use of a range of features as we have, across a diverse range of applications.

Aycock (2000) proposes a static type inference system for Python that makes aggressive assumptions about the use of dynamic features in Python. In justifying these assumptions, Aycock performed both a static analysis of a small set of programs to look for occurrences of *exec* and *eval*, and used an instrumented interpreter to find occurrences of dynamically-typed local and global variables at run time. He found that up to 7% of variable stores caused a change of type, and that these stores were localised in up to 5% of the total number of variables. Aycock did not consider dynamic object features as we have, but nevertheless concluded that static type inference is viable in Python, given additional source annotations.

StarKiller (Salib 2004) is a type inference system for Python that makes several assumptions about the underlying programs. In the presentation of this system Salib performed a static analysis on the Python standard library looking for uses of *eval*. In that work, almost all uses of *eval* were found to be superfluous, and easily replaceable with non-dynamic code. However, the study was biased in focussing only on the standard library, which is not representative of Python application code. The study also ignores other dynamic features StarKiller is unable to handle, such as *exec_stmt*.

Löwis (1998), in introducing a non-destructive change to the Python object layout, profiled a single large application to determine how often object attributes are retrieved from the instance, the class, or a superclass. This study is focussed on a specific optimisation, and does not suggest any strong link with type inference.

Several other type inference systems are worth mentioning for their implicit assumptions about the static nature of programs written in dynamic languages.

Cannon (2005) introduces static type inference to Python for local, atomic variables only. This is possible without making assumptions about the program, but his results on four Python programs indicated that very little performance benefit can be gained from the type information gleaned.

Cuni & Ancona (2007) suggest conservatively creating "fast paths" in the compiled bytecode when a Python program appears to reuse a particular class interface. They use a "tainted" flag to mark classes that have mutated beyond the original static analysis, and so will be able to handle unmodified Python programs.

Anderson (2006) describes a static type-inferencing algorithm for JavaScript, which has a similar object model to Python. Anderson's algorithm is sound, though not precise, in the presence of attribute addition and deletion, but does not address object delegation (a mechanism in JavaScript playing a similar role to class inheritance in Python) or dynamic code execution.

Furr et al. (2008) have developed DRuby, a variant of Ruby that supports type inference and type annotations. Their implementation, while unable to deal with reflection or dynamic code execution, was able to statically find type errors in a small number of modified Ruby programs.

The idea that programs behave differently after startup is closely related to program "phases," which have been extensively studied in the CPU simulation domain. For example, several techniques exist for automatically locating the places in source code that correspond to a phase change (Sherwood et al. 2002). This would include, but not be limited to, the program startup point, and we hope to be able to make use of these techniques in the future to replace our current manual approach for placing the start marker.

6 Conclusion

There is clearly a great desire amongst Python programmers and researchers for tools that can provide type checking or optimisation. It is well-known that these tools cannot operate precisely and accurately for arbitrary Python programs, but it is often said that many programs do not pose a problem.

In this paper we have tested 24 production open source Python programs to measure the extent to which they are amenable to static analysis. In every case the program would need significant modification to be rewritten into a language without dynamic features. However, many analyses typically performed statically could be performed at run time after startup.

We have discovered that game-like programs in particular have a characteristic high level of dynamic activity during startup, after which they use very few if any dynamic features.

The dynamic features of Python can be classified into several broad categories. Reflective features were highly used in all programs, even after startup. Dynamic code execution was used in 90% of programs during startup, and in 20% of programs after startup. It is unclear from our tests so far what the use of dynamic object modification is.

The next steps in our research are to broaden the scope of our test framework. In particular, we would like to track object modification more closely to look for changes to the class hierarchy. We also intend to measure the use of dynamic variables and functions. We would like to find the extent to which dynamic dispatch is actually required, which follows a similar investigative track as prior research into the degree of polymorphism in a program. For example, Grove et al. (1995) profiled the run-time behaviour of several C++ and Cecil programs in order to characterise the degree of polymorphism used in real object oriented programs. Their results indicated that receiver classes can be strongly predicted for a given call site; a trend that many just-in-time and profile-guided compilers use for optimising virtual and dynamic method dispatch. Having shown that run-time behaviour in Python programs is not completely dynamic, an obvious next step would be to reproduce this experiment and discover if dynamic dispatch targets can actually be predicted.

Our current instrumentation method does not record activity in threads besides the main thread. While we believe none of the tested programs make extensive use of multithreading, we hope to overcome this limitation in future work.

While we make these changes to the test framework we also intend to analyse more programs. So far we have omitted libraries and web services from the sample set, but we hope to be able to add them later.

Eventually we would also like to expand our research into more dynamic languages, such as Ruby, JavaScript and Perl, in order to discover if the trends we are observing are language-specific or a general artifact of dynamic programming.

7 Acknowledgments

We would like to thank James Noble and Bill Appelbe for their helpful suggestions during the preparation of this paper, and the reviewers for their useful feedback.

References

Abadi, M., Cardelli, L., Pierce, B. & Plotkin, G. (1991), 'Dynamic typing in a statically typed language', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **13**(2), 237–268.

Ancona, D., Ancona, M., Cuni, A. & Matsakis, N. D. (2007), 'RPython: a step towards reconciling dynamically and statically typed OO languages', *Proceedings of the 2007 symposium on Dynamic languages* pp. 53–64.

Anderson, C. (2006), Type Inference for JavaScript, PhD thesis.

Anderson, C., Barbanera, F. & Dezani-Ciancaglini, M. (2003), 'Alias and Union Types for Delegation', *Ann. Math., Comput. & Teleinformatics* **1**(1).

Aycock, J. (2000), 'Aggressive Type Inference', *Proceedings of the 8th International Python Conference* pp. 11–20.

Cannon, B. (2005), Localized Type Inference of Atomic Types in Python, PhD thesis, California Polytechnic State University.

Chang, P. P., Mahlke, S. A., Chen, W. Y. & Hwu, W. W. (1992), 'Profile-guided Automatic Inline Expansion for C Programs', *Software - Practice and Experience* **22**(5), 349–369.

Cuni, A. & Ancona, D. D. (2007), Towards a More Effective Implementation of Python on Top of Statically Typed Virtual Machines.

Dijkstra, E. W. (1968), 'Letters to the editor: go to statement considered harmful', *Communications of the ACM* **11**(3), 147–148.

Furr, M., An, J. D., Foster, J. S. & Hicks, M. (2008), 'Static Type Inference for Ruby'.

Grove, D., Dean, J., Garrett, C. & Chambers, C. (1995), 'Profile-guided receiver class prediction', *ACM SIGPLAN Notices* **30**(10), 108–123.

Gupta, R., Mehofer, E. & Zhang, Y. (2002), 'Profile Guided Compiler Optimizations', *The Compiler Design Handbook: Optimizations and Machine Code Generation* pp. 143–174.

Hamilton, G. (2006), JSR-292: Supporting Dynamically Typed Languages on the Java Platform, Technical report.

Löwis, M. (1998), 'Virtual Method Tables in Python', `http://foretec.com/python/workshops/1998-11/proceedings/papers/lowis/lowis.html` .

Pettis, K. & Hansen, R. C. (1990), 'Profile guided code positioning', *ACM SIGPLAN Notices* **25**(6), 16–27.

Salib, M. (2004), Starkiller: A Static Type Inferencer and Compiler for Python, PhD thesis, Massachusetts Institute of Technology.

Sherwood, T., Perelman, E., Hamerly, G. & Calder, B. (2002), 'Automatically characterizing large scale program behavior', *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* pp. 45–57.

Tempero, E., Noble, J. & Melton, H. (2008), 'How Do Java Programs Use Inheritance?: An Empirical Study of Inheritance in Java Software', *Lecture Notes in Computer Science* **5142**, 667–692.