Tracing Dynamic Features in Python Programs

Beatrice Åkerblom,
Stockholm University
Sweden

Jonathan Stendahl,
Mattias Tumlin
Stockholm University

Tobias Wrigstad
Uppsala University
Sweden

ABSTRACT

Recent years have seen a number of proposals for adding (retrofitting) static typing to dynamic programming languages, a natural consequence of their growing popularity for non-toy applications across a multitude of domains. These proposals often make assumptions about how programmers write code, and in many cases restrict the way the languages can be used.

In the context of Python, this paper describes early results from trace-based collection of run-time data about the use of built-in language features which are inherently hard to type, such as dynamic code generation. The end goal of this work is to facilitate static validation tooling for Python, in particular retrofitting of type systems.

Categories and Subject Descriptors

D.3 Programming Languages [**D.3.4 Interpreters**]: D.2.8 Metrics

General Terms

Languages

Keywords

Dynamic languages, open source, Python, dynamic features

1. INTRODUCTION

Interest in dynamic languages has increased in the last decade, both in industry and in academia [10]. Dynamic languages are popular for their flexibility, expressivity and succinctness, typically paid in weaker performance and safety [10]. The growing interest has resulted in numerous efforts to introduce static type systems for them in various ways, often restricting the way that the languages can be used.

Dynamic languages lack a static type system. Static type systems make it possible exclude certain errors from programs, provide verified documentation through type annotations and enable compilers to optimize programs. Static type system also come with disadvantages, for example, inherent conservative approximation is likely to reject some valid programs that would run without errors as not typeable.

Not having to support a static type system frees dynamic languages up for including powerful dynamic features which would be problematic in a typed setting, *e.g.,* reflective behaviour and metaprogramming facilities. Commonly available reflective mechanisms include support for checking available fields/methods, adding and removing fields/methods, without the need to restart or rebuild the running program, and runtime code generation. A common use of reflection is to extract all methods prefixed "test" in unit test frameworks, but also to generate names of attributes from program input.

It is easy to see that the ability to change types at run-time can make type-checking at compile-time extremely difficult as types are no longer stable. Types in a statically typed object-oriented language are usually defined as classes and allowing classes to change at runtime would make for a very unstable base for reasoning about programs' behaviours.

Earlier work on retrofitting type systems to Python have taken different approaches to handling the abovementioned difficulties, ranging from assuming that certain language features are not used in practise [3], that classes and types do not change after initialisation [8] or that the use is *e.g.,* limited to a certain phase [2]. Another approach is to allow parts of the program to fall back to runtime checks [1]. This work aims to provide hard numbers to asses the validity of these approaches, and guide future work in this direction.

Type systems can simplify development, especially through documentation which is verified to correspond to the code, and are the most scalable verification tools used in software development today. Additionally, type systems enable efficient compilation and data representation, especially of primitive data types, which can greatly impact performance by allowing direct manipulation of bit sequences for *e.g.,* arithmetic on numbers, instead of indirected, dynamically dispatched interaction with objects wrapping *e.g.,* an integer.

Understanding how dynamic languages are used when writing program code is important for many reasons. Our main goal here is aiding or investigating the feasibility of retrofitting static type systems onto Python. Additionally, understanding how language features are used in practise is crucial for design of interpreters, compilers and other tools that will be used with the language *e.g.,* for refactoring, and to influence education. This kind of understanding could also be helpful when exploring unknown code.

This work contributes to the understanding of actual dynamic program behaviour in real-world Python code in terms of what dynamic features are used, whether they are used in

MSR'14, May 31 – June 1, 2014, Hyderabad, India
ACM 978-1-4503-2863-0/14/05
http://dx.doi.org/10.1145/2597073.2597103

program-specific code or library code, and during start-up or normal execution, along with an analysis of our early results.

An additional contribution of this work is the development of tracing additions to an existing Python interpreter which without the performance penalties and unmanageably large logs of previous work [5], which has hampered validity of previously available results.

2. METHODOLOGY

This study was carried out using trace-based dynamic data collection using an instrumented version of the CPython interpreter for Python 2.6.6 [4]. We instrumented the interpreter to collect runtime data about uses of all builtin Python functions that can be used for introspection, object changes, code generation and library loading [4] (see Figure 1).

There were two main reasons for choosing version 2.6.6: it offered a larger collection of mature programs and it was the version shipped with Debian stable when the study was initialised, and Debian stable was a hard requirement to study certain proprietary code excluded in this paper.

Dynamic data collection has both advantages and disadvantages compared to a static collection based only on the code of the programs. Tracing is able to more precisely describe actual uses (*e.g.,* hard numbers of uses) of a certain feature than purely static analysis, and is sensitive to different paths taken in a program due to input. Furthermore, which is extremely relevant for dynamic languages, it easily captures events (an occurrence of a use of a feature of interest) in code which is generated or loaded dynamically in ways which are difficult or impossible to resolve statically. The main disadvantage of the dynamic approach is the difficulty of choosing representative input or interaction strategies for running programs which will give acceptable code coverage.

In this paper, we focus on the language features found in Figure 1. The names in `tele type` are language constructs/features that are part of standard Python. We have grouped these features into four high-level categories as follows. For brevity, we omit a description of the individual constructs and refer to the Python Reference Manual [4].

Introspection mechanisms used to examine the state of an object or the local scope at runtime (*e.g.,* checking for the availability of fields or methods).

Object Changes features that update or change the state of an object; update, add or remove fields in a way that may depend on the program state (using strings to store attribute names).

Code Generation features that evaluate code generated or imported in text format during runtime.

Library Loading constructs that can load or reload arbitrary libraries[1] at runtime. This allows f.ex. deferring decisions such as what library should be loaded according to user input or underlying hardware.

For this initial study we selected 19 open source programs (see list in Figure 2) from SourceForge[2] for traced runs. First of all, all selected programs were entirely developed in Python and currently actively developed (production/stable). Other criteria were that they were all interactive programs with a graphical user interface and useful programs (used by others than the developers with > 1000 downloads) that run under Debian GNU/Linux. The last criterium was that the programs had to be possible to use without special hardware (*e.g.,* microscopes and cameras) subscriptions or specialist knowledge in the application domain.

[1]Reloading a library can have arbitrary side-effects in Python programs and may update existing namespaces.

[2]Chosen for its simple manual search function.

Figure 1: Categorising dynamic features from Python.

Categories			
Introspection	Object Changes	Code Generation	Library Loading
`hasattr`	`del` *attribute*	`eval`	`_import_`
`getattr`	`delattr`	`exec`	`reload`
`_getattr_`	`_delattr_`	`execfile`	
`_getattribute_`	`setattr`		
`vars`	`_setattr_`		

The only modification to the programs is a manually inserted signal to our logging framework that the program's start-up is finished, which was typically when the graphical user-interface has been loaded and just before entering the main loop of the program, following [5]. Arguments exist in the literature [2, 5, 7] that programs exhibit more dynamism during startup than the remainder of their run-time, as well as arguments to the contrary. For example, Ancona et al.'s RPython allows unrestricted dynamic changes to classes during start-up, but assumes programs at some point stabilise and are possible to express using a more restricted subset of the Python language. The manually inserted signal makes is used to distinguish between events occurring at start-up and events occurring at run-time in our tracing logs.

All runs of the programs were documented with use cases and had the goal to use as much of the programs' functionality as practically possible. The progams were run manually by us with the intention of using the programs in a normal way (not randomly) making sure that all buttons were clicked and all menu alternatives were used. Exceptional functionality *e.g.,* automatic update checks may not have been run, though. Sadly, we did not obtain code coverage metrics from these traces as they omitted everything else than the information about the dynamic features encountered during execution.

In addition to collecting trace data, we searched program-specific code statically in the source files for occurrences of our features of interest. We did not use this analysis to exclude any programs from our corpus. It did however provide some understanding of the paths executed in the program during tracing, especially in the cases where features could be shown to exist statically, but not show up in traces.

3. RESULTS AND ANALYSIS

We traced the execution of 19 programs in our corpus collecting a total of 7.4 million events (5.6 million of these were collected for just one program) in tracing logs. To obtain data representative for different classes of programs, we chose programs of different sizes and complexity, from project planning tools to simple system tools for cleaning away caches and logs. Due to the small number of programs used to obtain these early results, we do not further discuss the nature of the programs or the correlation between use of certain features and program kind but this will be left for future work.

All programs in the study used many (between 5 and 9 out of 15) of the features in Figure 1, and all programs used features from all categories. The extent of this usage of each feature in the different programs was extremely varied, ranging from 2 uses to over 5 million uses.

In our results we focus on occurrences rather than counting numbers of calls traced. In these interactive programs, the

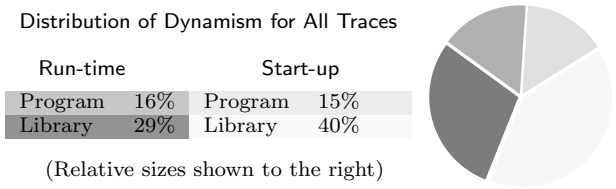

Figure 3: Distribution of dynamic features over libraries and program-specific code during start-up and run-time. Shading corresponds to Figure 2.

number of calls made to a dynamic feature is likely to be dependent on the running time and the number of times specific program functionality was used. A way to further this would be splitting calls traced on calls made in the code.

Each category in Figure 1 had exactly one feature whose use we did not find in our traces: `_delattr_`, `_getattribute_`, `execfile` and `reload`.

Figure 2 shows that traces for all programs contain features from all categories: introspection, object changes, dynamic imports, and dynamic code generation. In 17 of 19 programs, access attempts were made to attributes which were not defined using introspection. This suggests that such behaviour is a common Python idiom.

Figure 3 shows that dynamic features are used both in library code and in application-specific code, during the start-up phase as well as the run-time phase.

Dynamic feature use are only slightly more common in the start-up phase than in the run-time phase (55% of the dynamic features were found during startup and 45% during runtime). This result support that what other have concluded before us [5]—uses of dynamic features is higher during start-up than during "normal execution".

The number of uses of dynamic features is higher in library code than in program specific code (69% were found in library code and 31% in program specific code), but dynamism is by no means encapsulated in libraries.

In our study, the difference between library code and application-specific code (69%/31%) is larger than the difference between start-up and run-time (55%/45%) where many previous works expect differences to be found. This may be a result of using certain libraries that perhaps are more prone to use dynamic features than others. The differentiation between library code and program-specific code was not made in Holkner's and Harland's study and has not to our knowledge been made in any other trace-based dynamic study.

Figure 3 shows that 40% of all use of dynamic features were found in library code and executed during startup. All feature categories from Figure 1 are represented in Figure 4. Still in Figure 3, we see that 15% of the traced use of dynamic features originated in program-specific code during startup (see Figure 3) and the occurances were also found in fewer programs than for library startup. Except for dynamic import, the individual features were found only in less than half of the programs. The only feature that is more common in the program-specific code at startup is the feature `vars`, which checks for what variable names are defied in *e.g.,* a class, object or the local scope but it was only found traces from two programs.

The big difference between start-up library code and run-time library code is that run-time code generation (which was used by all programs in library code during start-up) was found in only in two programs in library run-time. One

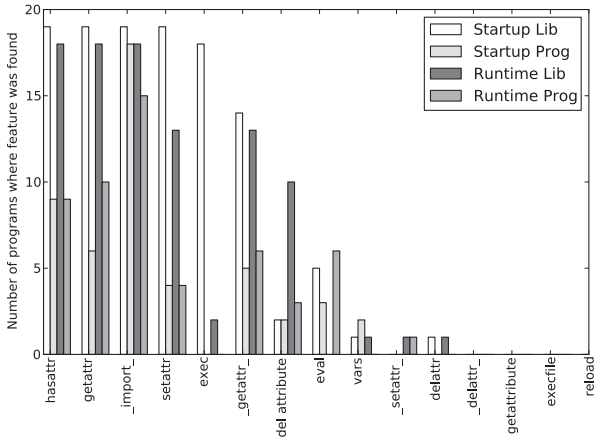

Figure 4: Occurrence frequency for dynamic features.

individual feature also differed from what we saw in library code during startup, that is the `del` *attribute* which is seen in five times as many programs (10 instead of 2) during runtime than during startup in library code.

Figure 4 shows that, compared with program-specific code at startup, the same number of programs or more use dynamic features in progam-specific code during runtime with only one exception being dynamic import.

Use of `_setattr_` has only been found in traces from one program's library code and from one program in program-specific code, both traced during run-time. It allows addition or change of an object's attributes using string-valued attribute names, possibly computed at run-time. Consequently, uses of `_setattr_` may change an object in ways which are difficult (or impossible) to track statically and may lead to inconsistencies *e.g.,* in a type system's view of an object and the object's run-time type. A similar argument applies to all features in the Object Changes category. For example, uses of `delattr` may remove fields from objects in running programs, which is not the same as assigning it to `None`, the Python equivalent of `null`. For example, `delattr` can affect shadowing and overriding, thereby changing an object's behaviour.

Uses of `delattr` were observed in one program during start-up and in one program during run-time. Both uses were traced from library code.

In the table below we see the median, average, minimum and maximum for all features per program separated on start-up library code, start-up application-specific code, run-time library code and run-time application-specific code.

The table shows that for all programs the library start-up code contains calls to dynamic features. For the other categories, on the other hand, there are programs that use no dynamic features at all (library run-time, application-specific start-up and application-specific run-time). The range between minimum and maximum are always very large, in

Per program dynamic feature usage

	Median	Avg	Min	Max
Entire programs	5.8 K	390 K	214	6.7 M
Library start-up	674	4.5 K	81	56 K
Library run-time	883	350 K	0	6.6 M
Program-specific start-up	508	3.2 K	0	33 K
Program-specific run-time	154	33 K	0	610 K

Figure 2: Raw data from traces. Cell colours same as Figure 3. Numbers denote the number of times the feature was used. For uses greater than 1000, we write 1 K, 1 M etc. rounded down for brevity. We do not show `_delattr_`, `getattribute`, `execfile` and `reload` since we found no uses of these in our corpus.

Program	hasattr		getattr		_getattr_		vars		del		delattr		setattr		_setattr_		eval		exec		_import_	
	Introspection								Object Changes								Code Generation				Lib.Load.	
Anomos	0	0	1.3 K	0	0	0	0	0	0	0	0	0	0	27	0	0	0	0	0	0	14	714
	1.2 K	231	1.4 K	234	10	7	0	12	42	0	0	0	106	63	0	0	0	12	0	84	23 K	1.9 K
Bleachbit	3	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	140	283
	37	28	11	31	1	2	0	0	3	0	0	0	12	17	0	0	0	0	16	6	322	442
Comix	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	227
	202	16	437	53	3	5	0	0	18	0	0	0	12	3	0	0	0	0	0	22	211	324
Convertall	607	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	261	0	0	0	65
	192	2	386	10	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	192	66
Elltube	6	0	56	0	0	0	0	0	0	0	0	0	136	0	0	0	0	0	0	0	0	58
	188	12	137	19	8	0	0	0	0	0	0	0	108	3	0	0	0	1	0	22	45	168
Exaile	268	43	870	20	16	23	0	0	0	0	0	0	4	0	0	0	35	1	0	0	22	579
	1.3 K	212	207	158	2	50	0	0	330	0	0	0	762	380	0	0	0	1	0	28	2.4 K	0.9 K
Kodos	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	106
	13	4	7	5	1	0	0	0	0	0	0	0	12	26	0	0	0	0	0	16	28	112
Mcomix	35	1	19	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52	505
	320	25	0.9 K	544	3	3 K	0	0	216	0	0	0	287	3	0	0	0	0	0	22	1.7 K	420
Photofilmstrip	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	205
	4 K	24	1037	12	1	3	0	0	3	0	0	0	460	3	0	0	0	0	0	193	210	439
Pysol	112	1 K	4.3 K	14	0	0	0	0	0	2	0	0	0	111	0	0	0	0	0	0	349	1244
	4.9 K	2.8 K	8.5 K	1.1 K	0	3	0	0	16	1	0	0	0	3	0	0	0	0	0	246	3.2 K	5 K
Radiotray	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	614
	15 K	342	165	576	17	54	0	0	0	0	0	0	204	10	0	0	0	0	0	56	640	890
Rednotbebook	32	217	313	72	0	6	0	16	2	0	0	0	0	11	0	0	0	0	0	0	7312	232
	224	182	6	55	0	2	0	0	0	0	0	0	0	3	0	0	0	0	0	23	184	403
Retext	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
	0	4	0	2	0	0	0	0	0	0	0	0	0	26	0	0	0	0	0	16	0	163
Sbackup	0	0	85	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	69	353
	413	143	3	28	0	58	0	0	3	0	0	0	0	62	0	0	0	0	0	44	458	1 K
Solfege	0	1	1353	0	606	0	0	0	2	0	0	0	54	0	66	0	16	0	0	0	67	509
	6	25	14	87	0	2	0	0	0	0	0	0	0	171	0	0	0	1	0	22	25	309
Taskcoach	0.1 M	2.2 K	0.1 M	192	0.4 M	17 K	0	2	115	2	0	0	305	208	0	0	1 K	141	0	0	41	1.1 K
	724	197	836	748	23	872	0	0	8	16	0	0	5 K	444	0	0	0	1	0	205	49	748
TorrentSearch	723	130	31	31	3	2	0	0	0	0	0	0	0	0	0	0	218	0	0	0	261	380
	661	366	450	314	372	51	0	0	0	0	18	6	648	504	0	0	0	0	0	28	66	559
Wikidpad	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	88 K	1.6 K	1 K	2.4 K	5.6 M	49 K	0	0	381	0	0	0	145	173	330	0	0	0	3	41	0.86 M	2.6 K
Zmail	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	23
	3	5	18	21	3	0	0	0	0	0	0	0	36	29	0	0	0	0	0	16	38	179

the table above ranging from 0 to 6.6M during library runtime. For entire programs, we see that if we add all occurrances for all features, the smallest, median and largest number of occurrences were 214, 5.8 K and 6.6 M respectively. This clearly indicates that real Python programs see reflective calls in the hundreds or thousands per program run.

4. CONCLUSIONS AND FUTURE WORK

Our results show that Real-world Python programs *do leverage* built-in dynamic features to exhibit behaviour which is inherently hard to type. When retrofitting type systems for Python code, f.ex., widespread use of reflection to create attributes from string values—so that a class' interface is not know at "compile time"—must be taken into account to make the type system work with actual Python code.

Our results show that dynamic behaviour is neither buried in library code, nor predominantly occurs at program start-up time. This is in slight contrast to earlier results in the literature [5], and mean that restricting use of reflection once programs stabilise is not possible, nor is the use of the powerful features of typed Scheme/Racket [9] which allows typing certain modules which interact with untyped modules.

In our on-going work, we aim to further investigate how Python program's leverage dynamism and "duck typing" in practise, similar to work done for Javascript [7], Smalltalk [6], and to some extent for Python [5]. We also study polymorphism of Python programs, and how various notions of typing (*e.g.,* nominal, structural, etc.) apply to existing code.

The work described in this article continues with additional traces and programs, further studies of the relationship between library and program code, along with qualitative investigations of the usage patterns for dynamic features, and to what extent they could be removed by trivial refactorings, to yield code which would be easier to type statically.

5. REFERENCES

[1] D. An, A. Chaudhuri, J.S. Foster and M. Hicks, Michael, "Dynamic Inference Of Static Types For Ruby", In POPL 2011.

[2] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. "RPython: Reconciling Dynamically And Statically Typed OO Languages", In DLS, 2007.

[3] J. Aycock. "Aggressive Type Inference", In Proceedings of the 8th International Python Conference, pp. 11-20, 2000.

[4] G. van Rossum and F.L. Drake, "PYTHON 2.6 Reference Manual", CreateSpace, Paramount, CA, 2009.

[5] A. Holkner and J. Harland, "Evaluating The Dynamic Behaviour Of Python Applications", Proceedings of ACSC '09, 2009.

[6] O. Callaú, R. Robbes, É. Tanter and D. Röthlisberger, "How Developers Use The Dynamic Features Of Programming Languages: The Case Of Smalltalk", Mining Software Repositories, 2011.

[7] G. Richards, S. Lebresne, B. Burg and J. Vitek, "An Analysis Of The Dynamic Behavior Of JavaScript Programs", in PLDI, 2010.

[8] P. Thiemann, "Towards A Type System For Analyzing JavaScript Programs", in ESOP, pages 408-422, 2005.

[9] Sam Tobin-Hochstadt And Matthias Felleisen, "Interlanguage Migration: From Scripts To Programs", in DLS'06, 2006.

[10] L. Tratt, "Dynamically Typed Languages", Advances in Computers 77. Vol, pp. 149-184, ed. Marvin V. Zelkowitz, 2009.