Improving Mutation Testing Process of Python Programs

Anna Derezinska and Konrad Hałas

Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
A.Derezinska@ii.pw.edu.pl
halas.konrad@gmail.com

Abstract. Mutation testing helps in evaluation of test suite quality and test development. It can be directed to programs of different languages. High cost of a mutation testing process limits its applicability. This paper focuses on mutation testing of Python programs, discussing several issues of mutant creation and execution. It was showed how they can be effectively handled in the Python environment. We discuss introduction of first and higher order mutation in an abstract syntax tree with use of generators, dealing with code coverage with AST, executing mutants via mutant injection into tests. The solutions were used in reengineering of MutPy - a mutation testing tool for Python programs. The improvements were positively verified in mutation experiments.

Keywords: mutation testing, Python, dynamically typed language

1 Introduction

Python is an interpreted high-level programing language [1] gaining more and more concern in different application areas, including web development, internet protocols, software engineering tools, or scientific computing [2]. Python programs, like in other general purpose languages, can be companioned with a test suite, especially unit tests. Test quality evaluation and their development can be assisted by mutation testing [3].

In mutation testing, a small change, so-called *mutation*, is introduced into a program code. A type of changes is specified by *mutation operators*. If a mutated program, a *mutant,* is modified by one mutation operator at one place, we speak about first order mutation. Introducing of many mutations into the same mutant is called higher order mutation. Mutants are run against tests. A mutant status is determined by the output of its test runs, a mutant can be *killed* by tests, *alive*, or *equivalent* – not to be killable by any tests. *Mutation score* of a test set is calculated as the number of killed mutants divided by the number of all considered nonequivalent mutants.

Usage of mutation operators in Python is restricted in comparison to other languages, like Java or C#. Python belongs to dynamically typed languages which raises a problem of *incompetent mutants,* mutants with possible type inconsistencies [4]. We proposed a pragmatic approach, in which some incompetent mutants are generated

The original publication is available at http://link.springer.com/chapter/10.1007/978-3-319-18473-9_23
A. Derezińska, K. Hałas, Improving Mutation Testing Process of Python Programs, R. Silhavy et. al. (eds.) Software Engineering in Intelligent Systems, AISC vol. 349, pp 233-242, Springer, 2015, doi 10.1007/978-3-319-18473-9_23

and run against tests. A set of mutation operators applicable in this context to Python is discussed in [5]. It was experimentally shown that the number of incompetent mutants was limited (about 5% on average) and the mutation results were accurate.

Details of a mutation testing process depend on different factors, such as a program level at which mutations are introduced (e.g. source code, syntax tree, intermediate code), selection of mutation operators, limitation of a mutant number due to different reduction criteria, selection of tests to be run, etc. [6,7]. The main problem of a mutation testing process is its high labor consumption. There are many attempts to reduce the cost, focusing on different aspects of the process [3], [8-12]. Some methods of mutant number reduction are associated with the less accurate results of the mutation testing process. One of approaches, in which less mutants are created and run, is higher order mutation [13]. Experiences of higher order mutation of Python programs are reported in [14].

Dynamic typing makes it possible to encounter incompetent mutants. Therefore, mutation in Python can be done either in the run-time or we can deal with incompetent mutants during test execution. In this work, based on our experiences [5], [14], the second approach is continued. Mutations are introduced into the program code and during code interpretation incompetent mutants are discarded.

In this paper, we focus on the improvements to the mutation testing process that do not decline the mutation score. We present how using different Python structures, selected steps of the process could be realized more effectively. The approaches were applied in refactoring of the primary version of MutPy – a mutation testing tool for Python [15]. Experiments confirmed benefits for mutation testing of Python programs.

This paper is organized as follows: we describe selected features of Python programs in the next Section. Section 3 presents various problems and their solutions for the mutation testing of Python programs. Tool support for the process and experimental results are discussed in Section 4. Finally, Section 5 describes related work and Section 6 concludes the paper.

2 Fundamentals of Python Programs

The Python programing language [1] supports the notions of procedural and object-oriented programming. Python is an interpreted language. A program is translated into its intermediate form that is interpreted during a program run. During the translation process, at first an Abstract Syntax Tree (AST) of a Python program is created. The tree is further converted into the bytecode form. Below, we recall briefly selected characteristics of the Python language.

2.1 Iteration and Generators

In Python, loop *for* is used in iterating over collection elements. An object used as an argument in a *for* loop is required to have a special *iterator* method. An iterator returns a next element of the collection or a special exception at the collection end.

Using this schema, a *generator function* can be created. It is designated by usage the keyword *yield* instead of the *return* operation. Each calling of the function, its iterator is returned, so-called *generator iterator,* or *generator* in short. An iteration step corresponds to execution of the function body till encountering of a *yield* statement. The function status is saved. In the next iteration step, the function state is resumed, and the code is executed until the subsequent *yield* statement or the function end occurs. In this way, an iterated object (generator) is created and can be used in a *for* loop.

2.2 Module Import

A module is a basic unit of a program execution. Each module has its namespace. Elements of a namespace, defined in a module or imported from other modules, are accessed by a module *dictionary*. Import of Python modules is supported by the interpreter system. The import algorithm consists of the following main steps:

1. A module is imported via *import* statement. If the module was already imported, its reference exists in the appropriate dictionary and can be further used. Finish.
2. Otherwise, the Python interpreter searches for the desired module by iterating over a system list of *meta_paths*. For each element of the list a so-called *finder* is called. The finder returns a *loader* object if the module was found. If all finders were asked for the module and none found it, the procedure is finished. The appropriate exception (*ImportError*) is raised indicating the missing module.
3. After the module was found, the *loader* is used to load the module. The reference of the loaded module is added to the dictionary. The procedure is completed and the module can be used.
4. It can happen that a loader cannot load the module. This situation is also pointed out by the exception *ImportError*.

A Python user can expand this functionality by designing appropriate *finder* and *loaders* objects.

3 Mutation Testing Process Improvements

In this section we discuss the improvements in the selected steps of the mutation testing process: introduction of mutations into the code, execution mutants with tests, and application of code coverage data in the mutant creation and execution.

3.1 Introducing First Order Mutations into Python AST

Application of a mutation into a code is based on a mutation operator. An operator specifies a kind of a source code that can be mutated and a kind of a code change that should be introduced.

One idea of application of mutation operators is manipulation of a code on the Abstract Syntax Tree (AST) level. Using the standard *ast* module we can manipulate the tree and apply mutation operators. A simple process of a mutant creation can consist of the following steps:

1. An original code is converted into its AST form (once per a program).
2. For each mutation operator, a copy of the AST is made:
   (a) a place of a mutation is found in the tree,
   (b) information about the mutation place is stored,
   (c) the tree is modified according to a given mutation operator,
   (d) the modified tree is adjusted - the line numbers are revised,
   (e) the tree is passed to the further translation steps and the intermediate code of the mutant is created.

Considering one place of AST many mutants can be created due to different mutation operators. The basic input of a module accomplishing a mutation operator is an AST node. The output of the mutation is a modified node, or information about a node deletion, or information about resining of the mutation (e.g. raising of an appropriate exception). A node that was already used by all mutation operators is marked as a recently visited code. Therefore, the same node will not be modified again during creation of subsequent mutants.

This simple introduction of mutations into an AST has disadvantages. The first one is copying of a syntax tree. It is performed in order to keep the original tree unmodified to be used for creation of subsequent mutants. However this coping is memory and time-consuming. An alternative to creation of a copy can be a reverse transformation of the tree. The modified tree fragments are restored to the original form.

The next problem is traversing of the whole tree while searching of the mutation place. This operation can take a considerable time amount in dependence of the number of tree nodes. Straightforward searching in a tree results in visiting of nodes modified in mutants that have already been generated during the mutation process.

Therefore, an improved and more efficient algorithm of mutation injection can solve these problems. A syntax tree is not to be copied for each step. The search engine of mutation places starts with the recently visited place. These ideas can be efficiently realized in Python with assistance of the *generator* concept (Sect. 2.1).

A new mutation engine can be served by a *MutationOperator* base class which is specialized by various classes responsible for particular mutation operators. A mutation method of the base class is a generator function. Therefore, we can iterate other the results of this operation. Input of the mutation method is the original AST to be mutated. Using the generator mechanism, succeeding mutants in the AST form are returned and a context is stored. The context is used by the mutation algorithm to memorize a mutation place in a tree. After a subsequent call of the generator function, the AST can be restored and the searching for a next mutation place can be continued.

At first, a possibility of being mutated is checked for a current node. If a mutation is allowed at this place the node is mutated. Then, the *yield* method returns a new version of the mutated node and suspends mutant generation process according to the *generator* principle. If a node cannot be mutated, its children nodes are visited. For each child a mutation generator is recursively activated in the *for* loop. A child node is substituted by a mutated node returned by the recursive call of the mutation generator. After the next iteration of the generator the AST is reconstructed.

More complex cases, are also taken into account. A node can be inspected for the next time, even if it was already modified. Moreover, an examined child of the current node can consist not only of a single node but also of a whole collection of nodes.

Other functionality performed on an AST is mutation of nodes based on the context information. A context of a node is specified mainly by its predecessor. Node successors are simply accessible in a tree. In order to access a node predecessor an AST supported by the standard Python library should be extended. Working on a node context, we can easily omit selected nodes that should not be mutated, e.g. documentation nodes that are equal to string nodes but are placed in a different context.

3.2 Higher Order Mutation

Mutations discussed in the previous subsection referred to the first order mutation (FOM). In higher order mutation, two or more changes are provided to the same mutant. A higher order mutant (HOM) can be created using several algorithms, such as FirstToLast, Between-Operators, Random or Each-Choice [13], [16]. For example, using a FirstToLast strategy, a 2nd order mutant is a combination of a not-used FOM from the top of the list with a not-used FOM from the list end.

Higher order mutation can also be applied to Python programs [14]. While building a HOM, changes of a program are also forced into the AST form. At the beginning, a list of all first order mutations is generated. The general idea of creating a HOM is based on merging of AST node changes defined by the first order mutation. If different HOM algorithms are handled, the strategy design pattern can be used.

In general, conflict situations where more than one mutation have to modify the same node are possible. We assumed that an AST node is modified only once in a given HOM. In this way, a situation is avoided when a next modification overwrites the previous one. Therefore, the merging algorithms were modified, taking into account the next first order mutation from the list.

3.3 Mutant Injection into Tests

Execution of mutants with tests can be effectively realized by a mutant “injection” into tests. Based on this operation, a test will be re-run with the mutated code, not an original one. It can be performed in the operational memory, without storing a mutant to a disk.

During evaluation of mutants, two program modules are available: a mutant module and a test module. If a mutant has to be tested, a reference in the test namespace should refer to the mutant functions instead of those from the original program. A valid reference to a mutated function is resolved by a concept of mutant injection.

A simple approach to mutant injection can be based on the comparison between namespaces of a mutant module and a test module. If an object belongs to both workspaces, its mutated version is injected into the namespace of the test module. This operation can be implemented by a simple assignment. However, this solution is not sufficient when a mutated module is imported inside a function or a method. Import mechanism is often used in such a context for different purposes. During a re-run of tests, a test method will import an original code but not its mutated version.

A new approach on the mutant injection was based on the extension of the import module mechanism (Sect. 2.2). Additional *finder* and *loader* were created that can be activated after an attempt to import a mutant module. Therefore, a mutation engine overrides importing a mutated module in a function or method. A dedicated class can combine the functionality of a finder and a loader. The class is treated as an *importer* in the Python system. Before a test is launched, our new importer should be added at the beginning of a system list (*meta_path*). If so, during the test execution this importer will be called at first in the import routine. Therefore, a mutated version of the module will be fetched instead of the original one.

3.4 Code Coverage Evaluation on AST

Code coverage is an important criterion of test quality. Reachability of a mutation place by a test is a one of necessary conditions to kill a mutant [17].

Performing mutation on the AST level is reasonable to resolve the code coverage also on the same program level. The solution is inspired by an instrumentation of AST [18]. The idea of the algorithm can be summarized in the following steps:

1. Numbering of all nodes of an AST.
2. Injection a special nodes to the AST. Calling of these nodes results in memorizing a number of a considered node.
3. Transformation of the modified AST into the intermediate form of a program.
4. Running the program (i.e., interpreting it) and collecting information about reachable nodes. A code of a reached AST node is counted to be covered.

Injected nodes are related to instructions that add a node number to a special set. These so-called *coverage instructions* have various types and are injected into different places in accordance to a considered original AST node:

- *a simple expression (line)* – a coverage instruction is injected just before the node; it adds a set of the numbers of all nodes included in the expression,
- *a definition of a function, method or a class* – a coverage instruction is injected at the beginning of the body of the node; it adds the number of the node definition,
- *a loop or conditional instruction* - a coverage instruction is injected just before the node; it adds a set of the numbers of all nodes included in the loop or conditional instruction,
- *other entities (e.g. nodes inside expressions)* – no instruction is injected, as the node number is already included in an instruction injected for a simple expression.

A mutant generation engine can use information about coverage of AST nodes. Mutations that would modify non-covered nodes can be omitted. Hence, the number of mutants to be generated is reduced.

Analyzing code coverage we can also obtain information by which test cases a particular node is covered. Using this information we can run only those tests that cover the node modified by a considered mutant. Hence the number of test runs is lowered.

3.5 Reduction of Test Runs

If all tests are run with each mutant (or only tests covering the mutation) we can gather detailed information about test effectiveness [19]. Though, if we are only interested in the mutation score, the testing process can be performed quicker. The testing of a mutant is interrupted if any test kills the mutant or a mutant is classified as an incompetent one. The remaining tests can be omitted for this mutant.

4 Tool and Experiments

The solutions discussed in Sect. 3 were implemented in the newest version of the MutPy tool [15]. MutPy was the first mutation testing tool supporting object-oriented mutations of Python programs and written in the Python language. Its earlier version (v. 0.2) was published in 2011. Based on the gathered experience, the tool was considerably refactored in order to assist the more effective mutation testing process. The main functionality supported by the current version of the MutPy tool (v. 0.4) is the following:

- generation of mutants with selected mutation operators, 20 operators including structural, object-oriented, and Python specific operators [5],
- first- or high-order mutation operators of a selected algorithm of HOMs: FirstToLast, Between-Operators, Random and Each-Choice [14].
- mutation of all code, or mutation of the code covered by tests of a given test suite,
- selection of a code that can be omitted while a mutation is introduced, e.g. a code of test cases,
- storing of mutants on disk (optionally),
- view of original and mutated Python source code and intermediate code,
- selection of tests that cover the mutated parts of code,
- running mutants with unit tests,
- categorization of mutants after test runs, detection of mutants alive, incompetent, killed by tests and by timeout,
- evaluation of mutation score and generation of reports on mutation results.

Results of two versions of the mutation testing process are compared in Table 1. The table summarizes the mutation testing of four projects: *astmonkey, bitstring, dictsect* and *MutPy* (v. 0.2) [14]. The data refers to first order mutation without considering code coverage data, as in the previous process only this kind of mutation was supported. The second column distinguishes a version of the mutation testing process with MutPy 0.2 (old), and the new one.

The first two rows present the total time of the mutation testing process. This time includes time of mutant generation, execution of test runs and evaluation of results.

We can observe that in the new version the time is 3 to 7 times shorter. The highest benefits were for the biggest project - *bitstring*, with the highest number of tests and the longest execution time. However, there are several factors that contribute to these times. The first one is the number of mutants caused by the revised set of mutation operators and avoidance of creation many incompetent mutants [5]. The number of mutants influences both the time of all mutant creation and all mutant execution. The second factor is the number of tests. It was lowered due to the strategy of deciding a mutant status after killing it by any test (Sect. 3.5).

Another factor is a cost of a mutant creation. The impact of this factor can be calculated independently. Average times of a mutant generation are compared in the bottom rows of the table. This improvement is independent on the mutant number and follows from the new algorithm of mutation introduction. The average time of a mutant generation was from few to even more than hundred times shorter that previously. In the former solution, generation time depends strongly on the size of a program, therefore, for the biggest project (*bitstring*) the improvement is the most substantial. The current algorithm of an AST manipulation caused that this average time is of the similar magnitude for different projects (0.06-0.02s).

The new approach to mutant injection into tests has not visible effect in the performance measures, but has made it possible to mutate some projects that were previously not allowed.

Table 1. Comparison of two versions of mutation testing process performed with MutPy

Measure	Version	astmonkey	bitstring	dictset	MutPy_old	Sum
Time of mutation process [s]	old	635.6	19405.4	297.8	337.8	20676.6
	new	98.7	2700.0	60.4	108.1	2967.2
Mutant number	old	1010	5444	572	703	7729
	new	521	3774	311	270	4876
Test run number	old	66660	2482464	98956	60458	2708538
	new	22230	733260	28500	16843	800833
Average time of mutant generation [s]	old	0.53	2.42	0.38	0.23	-
	new	0.05	0.02	0.05	0.06	-

Other improvements in the process performance are obtained by usage of code coverage information and higher order mutation. Usage of code coverage data reduced the mutant number of 6 % on average. The higher impact has the code coverage criterion in selecting of tests. The number of test runs dropped about 74-94 % for these projects. Summing up, only due to the code coverage inspection both in mutant generation and execution the mutation time was lowered, taking 44-88 % of the time without this data.

The detailed results of 2nd and 3rd order mutation with different HOM algorithms concerning these projects are given in [14]. In general, the number of mutants were halved for the 2nd order and about 33 % for the 3rd order. The resulting mutation time was equal 40-83 %, and 25-90 % of the 1st order mutation time, for 2nd and 3rd order mutation accordingly.

5 Related Work

Manipulation of the syntax tree in the mutant generation step was applied for different languages. Mutation of C# programs was performed at AST of the source code [20], but also at a syntax tree of the Intermediate Language form of .NET [21]. In this paper, the general ideas are combined with the Python-specific utilities. The most of other tools dealing with mutation testing of Python programs has a limited functionality and is restricted to few standard mutation operators, with no support to object-oriented ones. Their discussion can be found in [14]. Simple application of AST for introduction of Python mutations was also reported in the Elcap tool [22].

The problems of mutating dynamically typed languages were addressed in [4]. Mutations introduced in run-time were analyzed and illustrated by JavaScript examples. In the contrast, this paper was focused on the mutation in the source code, as a solution giving practically justified results.

An interesting approach to verification of Python programs was presented in [23], where mutations are introduced into the intermediate code. However, the purpose of mutation is different from discussed here. The authors compare error location abilities of two tools: UnnaturalCode for Python and the Python translator. UnnaturalCode locates errors in an instrumented program during its run-time. Many mutations introduced to the code could be detected during translation of the source code, and in this sense they are not suitable for evaluation of tests, as in the typical mutation testing.

Application of coverage data in the step of mutant generation was beneficially performed in the CREAM tool for mutation testing of C# programs [20]. In PROTEUM, the mutation environment for C programs, the code instrumentation was used and the reachability of mutation statements by tests were observed [6]. However, there is no information about an prior reduction of a test set based on the coverage data, like performed in the process discussed here. The integration of code coverage with mutation testing approach was also promoted in Pitest – a mutation tool for Java programs.

6 Conclusion

In this paper, we proposed improvements to the mutation testing process of Python programs. They concerned mutant generation based on modifications of an abstract syntax tree and execution of mutants with tests. First and higher order mutations were used. We explained how different Python language structures, like iteration with generators and an importer class that extends module import, were applied in the mutation tool reengineering. The advances implemented in the MutPy tool resulted in the considerable lowering the mutation time.

References

1. Python Programming Language, http://python.org
2. Pérez, F.; Granger, B.E. ; Hunter, J.D. Python: An Ecosystem for Scientific Computing. Computing in Science & Engineering. 13(2), 13-21 (2011)

3. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Softw. Eng. 37(5), 649-678 (2011)
4. Bottaci, L.: Type Sensitive Application of Mutation Operators for Dynamically Typed Programs. In: Proceedings of 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp 126-131. IEEE Comp. Soc. (2010)
5. Derezinska, A., Hałas, K.: Analysis of Mutation Operators for the Python Language. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J.: (eds.) DepCos-RELCOMEX 2014. AISC, vol. 286, pp. 155-164. Springer Int. Pub. Switzerland (2014)
6. Vincenzi, A.M.R., Simao, A.S., Delamro, M.E., Maldonado, J.C.: Muta-Pro: Towards the Definition of a Mutation testing Process. J. of the Brazilian Computer Society, 12(2) 49-61 (2006)
7. Mateo, P.R., Usaola, M.P., Offutt, J.: Mutation at the Multi-Class and System Levels. Science of Computer Programming 78, pp.364-387, Elsevier (2013)
8. Usaola, M.P., Mateo, P.R.: Mutation Testing Cost Reduction Techniques: a Survey. IEEE Software 27(3), pp. 80–86 (2010)
9. Offut, J., Rothermel, G., Zapf, C.: An Experimental Evaluation of Selective Mutation. In: 15th International Conference on Software Engineering, pp. 100-107 (1993)
10. Zhang, L., Gligoric, M., Marinov, D., Khurshid, S.: Operator-Based and Random Mutant Selection: Better Together. In: 28th IEEE/ACM Conference on Automated Software Engineering, pp. 92-102. Palo Alto (2013)
11. Derezinska. A., Rudnik M.: Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs. In: C.A. Furia and S. Nanz (eds.): TOOLS Europe 2012. LNCS, vol. 7304, pp. 42–57. Springer, Berlin (2012)
12. Bluemke, I., Kulesza, K.: Reduction in Mutation Testing of Java Classes. In: Holzinger, A., Libourel, T., Maciaszek, L.A., Mellor, S. (eds.) Proceedings of the 9th International Conference on Software Engineering and Applications, pp. 297-304. SCITEPRESS (2014)
13. Polo, M., Piattini, M., Garcıa-Rodrıguez, I.: Decreasing the Cost of Mutation Testing with Second-Order Mutants. Software Testing Verification & Reliability, 19(2),111–131 (2009)
14. Derezinska, A., Hałas, K.: Experimental Evaluation of Mutation Testing Approaches to Python Programs. In: Proceedings of IEEE International Conference on Software Testing, Verification, and Validation Workshops, pp. 156-164. IEEE Comp. Soc. (2014)
15. MutPy, https://bitbucket.org/khalas/mutpy
16. Mateo, P.R., Usaola M.P., Aleman J. L. F.:Validating 2nd-order Mutation at System Level, IEEE Trans. Softw. Eng., 39(4), 570–587 (2013)
17. DeMillo, R.A., Offutt, A.J: Constraint based automatic test data generation. IEEE Transactions on Software Engineering, 17(9), 900–910 (1991)
18. Dalke, A.: Instrumenting the AST. http://www.dalkescientific.com/writings/diary/archive/2010/02/22/instrumenting_the_ast.html
19. Derezinska A.: Quality Assessment of Mutation Operators Dedicated for C# Programs. In: Proceedings of 6th International Conference on Quality Software, QSIC'06, pp. 227-234. IEEE Computer Society Press, California (2006)
20. Derezinska, A, Szustek, A.: F Evaluation of the C# Mutation System. In: Szmuc, T., Szpyrka, M., Zendulka, J. (eds.) CEE-SET 2009. LNCS vol.7054, pp. 229-242. Springer (2012)
21. VisualMutator, http://visualmutator.github.io/web/
22. Elcap, http://githup.com/sk-/elcap
23. Campbell, J., C., Hindle, A., Amaral, J., N.: Python: Where the Mutants Hide, or Corpus-based Coding Mistake Location in Dynamic Languages, http://webdocs.cs.ualberta.ca/~joshua2/python.pdf