Analysis of Mutation Operators for the Python Language

Anna Derezińska 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. A mutation introduced into a source code of a dynamically typed program can generate an incompetent mutant. Such a mutant manifests a type-related error that cannot be detected before the mutant execution. To avoid this problem, a program mutation can be provided at run-time, or incompetent mutants should be automatically detected and eliminated. We showed that the latter solution can effectively be applied providing selected mutation operators. This paper discusses mutation operators to be used for mutation testing of Python programs. Standard and object-oriented mutation operators were applied to the Python language. Python-related operators dealing with decorators and collection slices were proposed. The operators were implemented in MutPy, the tool for mutation testing of Python programs, and experimentally evaluated.

Keywords: mutation testing, mutation operators, Python, dynamically typed programming language.

1 Introduction

The Python programming language [1] belongs to the eight most popular programming languages over the last decade [2]. As a dynamically typed language it might be more sensitive to small mistakes of programmers and more difficult to be tested in comparison to strongly typed programming languages, such as C++, Java or C#. Nevertheless, the efficient testing of Python programs is indispensable. Therefore, we tried to apply the mutation testing approach to Python programs.

Mutation testing is recognized as a beneficial method for evaluating of a test case suite as well as creating of effective test cases [3]. The main idea originates from the fault injection approaches in which a fault is intentionally introduced and should be revealed by adequate tests. A code of a mutated program is slightly changed in comparison to the original code. If only one change is applied we speak about *first-order mutation*. Introduction of many changes to the same program is called *higher-order mutation*. A modified program, which is named a *mutant*, can be run against a set of test cases. If after running a test a mutant behavior differs from the original program, the mutant is said to be *killed* by the test. If the test suite did not reveal the change,

The original publication is available at http://link.springer.com/chapter/10.1007/978-3-319-07013-1_15
W.Zamojski et al. (eds), *Proceedings of the Ninth International Conference DepCoS-RELCOMEX*, Advances in Intelligent Systems and Computing 286, doi: 10.1007/978-3-319-07013-1_15, © Springer International. Publishing Switzerland 2014, pp. 155-164.

either these test cases are insufficient and should be supplemented, or the mutant is *equivalent* one and no test could kill it.

The changes of programs are specified with *mutation operators*. *Standard* operators, also called *traditional* or *structural* ones, are easily applied in any general purpose strongly typed programming languages, e.g. arithmetic '+' is substituted by '-'. In object-oriented languages, several additional operators are used, which are devoted to object-oriented features. There are several studies on mutation operators concerning strongly typed languages, including structural languages [4,5] and object-oriented ones [6-10]. Applicability of specific language features to mutation testing operators was discussed in [11]. Analogies and differences among operators of different languages were shown in [12]. The question is whether these operators are applicable in a dynamically typed language, especially in Python. The Python language has also some specific aspects that should be tested, and therefore they could be considered by Python-related mutation operators.

The main obstacle of applying the mutation approach to a dynamically typed language is a fact that a mutant often cannot be verified before the run-time. The type relations in a programming statement and their correctness are not known before the mutant is called. A mutant can be syntactically correct but violets dynamic binding properties of the type system at runtime, hence is *incompetent* one. The general solution would be providing all mutations during the program execution. This approach to mutation of dynamically typed programming languages was considered by Botacci in [13]. The ideas were discussed on JavaSript examples, but were not put into a praxis.

In this paper, we followed a more practical approach. Selected mutation operators only sometimes generate incompetent mutants. This sort of cases can be automatically detected at run-time, and a mutant like this will be not counted to results of mutation testing. Experiment results showed, that the overhead cost concerning the elimination of such incompetent mutants is not very big for this set of mutation operators. Incompetent mutants can be handled by a mutation tool.

Based on mutation operators used in different programming languages (Fortran, C, Java, C#) and on Python programming features, a set of operators was determined that is applicable to Python programs. In this paper, we presented this set, basic principles of operator selection and detailed description of some Python-related operators. An exhaustive discussion about adaptation of all operators to the Python statements is omitted due to brevity reasons and can be found in [14].

Mutation operators discussed in this paper were implemented and experimentally examined using MutPy - a mutation testing tool for Python programs [15,16]. The set of operators in MutPy is so far the most comprehensive among all mutation tools for Python programs. It is the only tool supporting OO operators in Python. Other tools are either simple and not updated like Pester [17] or a proof of concept with 2-3 operators like PyMuTester [18] and Mutant [19]. The only exception is a recently developed Elcap [20] that introduces mutations at the abstract syntax tree level, similarly to MutPy. Elcap supports 8 structural mutation operators.

The rest of this paper is structured as follows. In the next Section we discuss mutation testing of dynamic languages as well as standard and OO operators that can be used in Python programs. Section 3 introduces mutation operators that were designed

for selected features of the Python language. Experiments on mutation operators with Python programs are presented in Section 4, and Section 5 concludes the paper.

2 Adaptation of Mutation Operators to the Python Language

In strongly typed programming languages, a code replacement defined by a standard mutation operator can be verified by a static analysis. Types of data, variables and operators, as well as type consistency rules are known at the compilation time. Therefore, we can assure that a mutated program will be correctly compiled, and a mutation is not a source of type-related errors encountering at run-time.

The application of some object-oriented mutation operators is more complicated because they can depend on various conditions, e.g. other classes in an inheritance path [8,9]. However, while checking sufficient correctness conditions it is possible to avoid invalid code modifications.

2.1 Mutation of Dynamically Typed Programming Languages

Generation of mutants in a dynamically typed language can provide problems of type control [13]. Without a knowledge about types of variables, a mutation tool could introduce an invalid mutation, i.e. a mutant execution ends with a type related exception. This problem can be illustrated by a Python code example. The original program includes the following code, shown on the leftt hand side. If AOR (*Arithmetic Operator Replacement*) mutation operator is applied, the following mutant can be generated:

Example before mutation:

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

after mutation:

```
def add(x, y):
  return x - y
```

A mutant like this could be run with various parameters. If *add* is called with integer values, e.g. *add(2,2)*, the mutation will be valid. The mutant outputs value of *0*, whereas the original program ends with *4*. However, the function *add* could also be executed with string parameters. For example, calling of *add('a', 'b')* gives in the original program concatenation of those two strings - 'ab'. On the other hand, mutated version of the function cannot be executed because there is no string subtraction operation. After this kind of call, the *TypeError* exception is raised. Before the program execution, we could not determine types of variables *x* and *y*, therefore a part of mutants generated by AOR would be *incompetent*.

2.2 Mutation Operators from other Languages Applied to Python

The analysis of existing mutation operators was founded on operators designed and successfully applied in mutation testing tools for the following languages: Fortran (Mothra [4]), C (Proteum [5]), Java (MuJava [7]), and C# (CREAM [21]). There are many structural operators, but considering a different syntax of the Python language,

the set of structural operators was preliminarily reduced. For example, there are no *goto* and *do-while* statements, hence C-like operators that change this sort of instructions are not applicable for Python programs.

Then, a systematic review of the operators was performed on an initial set of potential operators. The set consisted of 19 traditional operators and 36 object-oriented ones. Selecting an operator, we took into account its applicability to Python and a manifestation of a possible fault made by a program developer. Moreover, we try to avoid operators that can generate a substantial number of incompetent mutants. This assessment was based on a program static analysis and preliminary experiments performed using a previous version of the MutPy tool.

In general, operators that demand to add a new element to a mutant are risky ones. More safe are operators that delete or change a program element. This idea can be illustrated by an example. A statement `y = ~ x` can be mutated by omitting a bitwise negation operator. The outcome is a valid mutant `y = x`. The reverse mutation, i.e. adding '~' operator, would generate in most cases an incompetent mutant.

Other mutation operators that generate mostly incompetent mutants deal with substitution of variables, objects or constants, with other variables, objects or constants. If a type of a programming item to be substituted is not known, a number of potentially generated mutants will be very high; much higher than in a strongly typed language. On the other hand, the most of mutants like these would be incompetent. Therefore, such operators were not included into the final set of operators to be implemented.

The Python language supports basic object-oriented concepts, thus we also try to adopt object-oriented mutation operators used in Java or C#. However, while analyzing these operators, we can perceive that many of them are not suitable for Python. Several keywords are not used in the way as in Java or C#, for example *override, base, static* before a class field. Some constructions, common to strongly typed languages, like type casting or method overloading are also not used. Therefore, about 20 object-oriented mutation operators were excluded.

In the next step, we omitted object-oriented operators that could be defined in Python, but require type-based verification during application. For example, operator PRV (*Reference Assignment with other Compatible Type)* was omitted. Without type verification, these operators would mostly generate incompetent mutants.

Other operators refer to an optional usage of selected structures. For example, in Java and C# a keyword *this* can in many cases be used or not. The analogous keyword *self* is obligatory used in Python. Therefore, the operator JTI that deletes this kind of keyword is not applicable to Python.

After the analysis, remaining operators adapted for Python were implemented in the mutation tool. They are listed in Table 1, indicating an operator category: S - structural and OO - object-oriented.; and one operator falls in both categories.

New operators related to specific constructions of the Python language were also proposed (*Python-related* column). They are discussed in Section 3.

The last column indicates trial operators that were considered for Python programs, but finally did not include into the recommended set of mutation operators supported by the MutPy tool. According to the program analysis and experiments they provide

many incompetent mutants and/or do not significantly contribute to the quality evaluation of tests or mutation-based test generation.

A detailed specification of all operators can be found in [14].

Table 1. Mutation operators for Python implemented in MutPy v 0.3

	Operator	Category	Python-related	Trial
AOD	Arithmetic Operator Deletion	S		
AOR	Arithmetic Operator Replacement	S		
ASR	Assignment Operator Replacement	S		
BCR	Break Continue Replacement	S		
COD	Conditional Operator Deletion	S		
COI	Conditional Operator Insertion	S		
CRP	Constant Replacement	S		
DDL	Decorator Deletion	S, OO	v	
EHD	Exception Handler Deletion	OO		
EXS	Exception Swallowing	OO		
IHD	Hiding Variable Deletion	OO		
IOD	Overriding Method Deletion	OO		
IOP	Overridden Method Calling Position Change	OO		
LCR	Logical Connector Replacement	S		
LOD	Logical Operator Deletion	S		
LOR	Logical Operator Replacement	S		
ROR	Relational Operator Replacement	S		
SCD	Super Calling Deletion	OO		
SCI	Super Calling Insertion	OO		
SIR	Slice Index Remove	S	v	
CDI	Classmethod Decorator Insertion	OO	v	v
OIL	One Iteration Loop	S		v
RIL	Reverse Iteration Loop	S	v	v
SDI	Staticmethod Decorator Insertion	OO	v	v
SDL	Statement Deletion	S		v
SVD	Self Variable Deletion	OO		v
ZIL	Zero Iteration Loop	S		v

3 Python-Related Mutation Operators

Various notions of the Python language [1], such as decorator, index slice, loop reversing, membership relation, exception handling, variable and method hiding, *self* variable were considered for mutation operators. New operators designed for some features, and selected after preliminary experiments are presented below. Other Py-

thon features were used in the scope of the adopted operators, e.g. the membership operator *in* can be negated by the COI operator (*Conditional Operator Insertion*) [14].

3.1 Decorators

Decorator is a function wrapper, which transforms input parameters and a return value of an original function. In Python, decorators are mostly used to add some behavior to a function without violating its original flow. A decorator is a callable object that takes a function as an argument and returns also a function as a call result. In a Python program, all decorators should be listed before a function definition and ought to be started with '@' sign.

The following example presents a simple decorator, which prints all positional arguments and a return value of a given function.

```
def print_args_and_result(func):
    def func_wrapper(*args):
        print('Arguments:' , args)
        result = func(*args)
        print('Result:', result)
        return result
    return func_wrapper
```

This decorator can be applied to a simple *add* function as follows:

```
@print_args_and_result
def add(x, y):
    return x + y
```

After calling the decorated function, e.g. *add(1,2)*, we obtain the following output:

```
Arguments: (1,2)
Result: 3
```

Three mutation operators dealing with the decorator notion were proposed: DDL *Decorator Deletion*, CDI *Classmethod Decorator Insertion*, and SDI *Staticmethod Decorator Insertion*.

The DDL operator deletes any decorator from a definition of a function or method.

Example before mutation:

```
1 class X:
2     @classmethod
3     def foo(cls):
```

after mutation:

```
class X:

    def foo(cls):
```

While using the *@classmethod* decorator, a method of a specialized class object is assigned as a first argument. If this decorator is deleted, the first argument is a default one. The method can be called from an object instance *x.method()*, where *x* is an instance, or using a class *X.method()*, where *X* is a class name. The latter case results in an incompetent mutant, but both cases can only be distinguished at run-time.

The DDL operator can also delete the *@staticmethod* decorator. The number of arguments accepted by the method without this decorator is incremented. The mutant will be valid if the modified method has a default argument that was not given in a method call. Otherwise, the mutant will be incompetent.

The DDL operator also deletes other decorators. If more than one decorator is assigned to an artifact all of them will be deleted.

The remaining two operators dealing with delegates were classified as trial ones. Both operators insert a decorator to a method definition, namely CDI inserts *@classmethod* and SDI *@staticmethod.* Operators inserting a new element into a program are usually a source of incompetent mutants, and the same occurred here.

3.2 Collection slice

An element of a collection can be referred in Python programs using the access operator '[]' with an appropriate index (or a key in case of the *dict* type). It is also possible to use this operator for obtaining a subset of a collection, so called *slice*. The syntax of the '[]' operator usage is the following:

```
collection[start:end:step]
```

where *start* is the index of the first element of the slice, *end* the last element, and *step* a metric between two consecutive elements of a slice. Each of these three arguments is optional. The slice elements are taken from the beginning of a collection if *start* is missing, and will go the collection end if no *end* element is defined explicitly. The following examples are valid slices of collection *x*:

```
x[3:8:2]    x[3:8]      x[::2]    x[3:]
```

Mutation operator SIR *Slice Index Removal* deletes one argument of the slice definition*: start*, *end* or *step*. Each removal gives a syntactically correct slice.

Example before mutation: after mutation:

```
x[1:-1]                    x[:-1]
```

3.3 Loop Reversing

A loop in a Python program is realized by the *for* statement. Any loop can be iterated in the reverse order, which is completed by function *reversed()*, a built-in function of the Python language.

Mutation operator RIL *Reverse Iteration Loop* changes direction of a loop iteration. However, the preliminary evaluation of the operator indicates that many equivalent mutants can be generated by the operator.

Example before mutation: after mutation:

```
for x in y:                for x in reversed(y):
```

4 Experiments

The mutation operators listed in **Table 1**. were implemented in MutPy - a mutation testing tool for Python 3.x source code [15,16]. It applies mutation on the abstract syntax tree (AST) level of a program. MutPy also supports higher order mutation and code coverage analysis [22].

In this section, results of the first order mutation testing in dependence on different mutation operators are presented. We carried out experiments on four Python programs [15,22]. The results are summarized in **Table 2**. The number of all mutants generated using a particular mutation operator is shown in the column *All*. In further columns, a distribution of this number is given, including the number of mutants that were killed by tests, killed by timeout, classified as incompetent mutants and remain not killed. Mutants are *killed by tests* when a test output differs from the output of the original program. Mutants are *killed by timeout* when the execution time exceeds a time limit calculated as approximately five times the original execution time.

Table 2. Mutation testing results

Operator	Mutant number						Mutation Score
	All	Killed by tests	Killed by timeout	Incompetent		Not killed	
AOD	38	25	1	0	0.0%	12	68.4%
AOR	740	441	5	117	15.8%	177	71.6%
ASR	82	67	4	3	3.7%	8	89.9%
BCR	14	7	4	0	0.0%	3	78.6%
COD	141	121	0	8	5.7%	12	91.0%
COI	601	511	5	39	6.5%	46	91.8%
CRP	2378	1435	3	47	2.0%	893	61.7%
DDL	16	0	0	7	43.8%	9	0.0%
EHD	37	21	0	9	24.3%	7	75.0%
EXS	55	36	0	4	7.3%	15	70.6%
IHD	27	11	0	0	0.0%	16	40.7%
IOD	47	25	0	6	12.8%	16	61.0%
IOP	4	0	0	0	0.0%	4	0.0%
LCR	61	36	0	15	24.6%	10	78.3%
LOD	7	7	0	0	0.0%	0	100.0%
LOR	136	121	1	0	0.0%	14	89.7%
ROR	512	423	4	5	1.0%	80	84.2%
SCD	4	1	0	0	0.0%	3	25.0%
SCI	43	7	0	8	18.6%	28	20.0%
SIR	85	61	4	0	0.0%	20	76.5%
Sum	5028	3356	31	268	5.3%	1373	71.2%

A mutant is classified as *incompetent* if the *TypeError* exception is raised during running it with tests. The exception is detected by the mutation tool, and this mutant is not counted to the overall mutation score of the program. The whole time of the mutation process is increased due incompetent mutants, proportionally to their number. However, as we can observe in **Table 2**, there were only 5.3% of incompetent mutants in comparison to all generated mutants. The percentage of incompetent mutants is different for various mutation operators. There are few operators for which about 15% - 25% of mutants was incompetent. A high number of incompetent mutants (43.8%) was also detected for the DDL operator. This is one of Python-related operators included into the final operator set, and it confirmed suspicions about possible many incompetent mutants. On the other hand, the handling of incompetent mutants is transparent to a user, and the overall overhead (few %) can be accepted.

The last column gives the *mutation score*. This measure reflects the ability of tests to kill the mutants. The mutation score is calculated as a sum of mutants killed by tests and killed by timeout divided by all generated and competent mutants. The exact calculation of mutation score should exclude *equivalent* mutants, i.e. mutants that have behavior equivalent to the original program and cannot be killed. In the current version of the MutPy tool, automatic classification of equivalent mutants is not supported. Therefore, the results can be treated as a lower bound of the mutation score, so-called *mutation score indicator*. However, it can be observed that CRP operator generated extremely many mutants. Many of them were not killed but also were not equivalent to the original program, as, for example, they change a string value to be displayed and this message content was typically not verified by tests.

5 Conclusion

The selection of discussed mutation operators were based on two general premises. The operator should be firstly useful and secondly effectively applicable in the mutation testing process of a dynamically typed language. The considered mutation process assumed application of operators into an intermediate form of a program before the run time, and an automatic detection of incompetent mutants.

All mutation operators presented in the paper were implemented in the mutation tool and experimentally verified. Experiments showed that only a few percent of generated mutants were incompetent, which gave an upper bound of the time overhead. This confirms the practical solution used in the MutPy tool providing a wide scope of various operators. Further experiments on mutating Python programs consider some techniques of mutation cost reduction and higher-order mutation [22].

An open question remains an extension of the mutation process with mutation operators used in strongly typed languages, but omitted due to the tendency of generating many incompetent mutants. If a mutant like this deals with a programming feature that could be easily verified with static analysis, then the operator is not worthwhile to be implemented. Otherwise, if such an operator is creating valid mutants apart from the incompetent ones, and these mutants could be used for verifying important lan-

guage features, then it could be beneficial to create an additional environment for the run-time generation of this kind of mutants.

References

1. Python Programming Language, http://python.org
2. TIOBE Programming Community Index, www.tiobe.com [visited Jan 2014]
3. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering, 37(5), 649-678 (2011)
4. King K. N., Offutt A. J.: A Fortran Language System for Mutation-based Software Testing. Software - Practice and Experience, 21(7), 685-718 (1991)
5. Delamaro M. E., Maldonado J. C.: Proteum-A tool for the Assessment of Test Adequacy for C Programs. In: Proc. of the Conf. on Performability in Computing Systems (PCS 96), pp. 79–95 (1996)
6. Ma, Y-S., Kwon, Y-R., Offutt, A.J.: Inter-Class Mutation Operators for Java. In: Proc. of Inter. Symp. on Soft. Reliability Eng., ISSRE'02, pp. 352-363. IEEE (2002)
7. Ma Y-S., Offutt J., Kwon Y-R.: MuJava: an Automated Class Mutation System. Software Testing, Verification and Reliability 15(2), 97-133 (2005)
8. Derezińska, A.: Advanced Mutation Operators Applicable in C# Programs. In: Sacha, K. (ed.) IFIP vol. 227, Software Engineering Techniques: Design for Quality, pp. 283-288. Springer, Boston (2006)
9. Derezińska A.: Quality Assessment of Mutation Operators Dedicated for C# Programs. In: Proc of 6th Inter. Conf. on Quality Software, QSIC'06, pp. 227-234. IEEE Computer Soc. Press, Los Alamitos California (2006)
10. Derezińska, A., Kowalski K.: Object-Oriented Mutation Applied in Common Intermediate Language Programs Originated from C#. In: Proc. of 4th Inter. Conf. Software Testing Verification and Validation Workshops (ICSTW), pp. 342 - 350. IEEE Comp. Soc. (2011)
11. Derezińska, A.: Analysis of Emerging Features of C# Language Towards Mutation Testing. In: Mazurkiewicz, J., Sugier, J., Walkowiak, T., Zamojski, W. (eds.). Models and Methodology of System Development, Monographs of System Dependability vol. 1, pp. 47-59. Publishing House of Wrocław University of Technology, Wrocław (2010)
12. Boubeta-Puig, J., Medina-Bulo, I., Garcia-Dominguez, A.: Analogies and Differences between Mutation Operators for WS-BPEL 2.0 and Other Languages. In: Proc. of 4th Inter. Conf. on Soft. Testing, Verif. and Validation Workshops, pp. 398-407. IEEE (2011)
13. Bottaci, L.: Type Sensitive Application of Mutation Operators for Dynamically Typed Programs. In: Proc. of 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp 126-131. IEEE Comp. Soc. (2010)
14. Derezińska, A., Hałas, K.: Operators for Mutation Testing of Python Programs. Res. Rep. 2014, Inst. of Comp. Science Warsaw Univ. of Technology (2014)
15. Hałas, K.: Cost Reduction of Mutation Testing Process in the MutPy Tool. Master thesis, Institute of Computer Science, Warsaw University of Technology (2013) (in polish)
16. MutPy, https://bitbucket.org/khalas/mutpy
17. Jester - the JUnit test tester, http://jester.sourceforge.net/
18. PyMutester,http://miketeo.net/wp/index.php/projects/python-mutant-testing-pymutester
19. Mutant, http://github.com/mikejs/mutant
20. Elcap, http://githup.com/sk-/elcap

21. Derezińska. 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 Heildelberg (2012)
22. Derezińska, A., Hałas, K.: Experimental Evaluation of Mutation Testing Approaches to Python Programs. In: Proc. of IEEE Inter. Conf. on Software Testing, Verification, and Validation Workshops, pp. 156-164, IEEE Comp. Soc. (2014)