Warsaw University of Technology

Operators for Mutation Testing of Python Programs

by

Anna Derezińska, Konrad Hałas

ICS Research Report 2/2014

Institute of Computer Science

Nowowiejska 15 / 19, 00-665 Warsaw, Poland

Operators for Mutation Testing of Python Programs

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. Mutation testing that supports design of high quality test suites of a program. It is based on mutation operators that specify small changes introduced into a program. Python is an interpreted programming language with dynamic typing system. In dynamically typed programs, type consistency rules cannot be statically verified. A mutated program can be an incompetent mutant that manifests an incorrect behavior at runtime due to violation of typing rules. In this paper we specify a set of mutation operators that could be applied to Python programs. Most of them correspond to selected operators used in strongly typed general purpose languages, covering structural and object-oriented mutation operators. They were adopted to the Python language. Operators designed for Python features are also discussed. We try to avoid operators that generate many incompetent mutants. All operators can be applied before a mutated program execution and possible incompetent mutants are to be automatically detected at runtime.

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

1 Introduction

Mutation testing is a powerful technique that supports design of a high quality test suite of a program [1]. A program under concern is modified by automatic introduction of small changes. Specifications of these modifications are called mutation operators. *Standard* (also called *structural* or *traditional*) mutation operators relate to common expressions used in general purpose languages [2-5]. *Object-oriented* mutation operators are devoted to object-oriented concepts in OO languages [5-9]. There are also various operators specialized to specific language features [8,10,11].

Python is an interpreted programming language with dynamic typing system [12]. While applying mutation operators in dynamically typed programs, type consistency rules cannot be statically verified. A mutated program can manifest an incorrect behavior at runtime due to violation of typing rules, and will be an *incompetent mutant* [13]. An incompetent mutant can be treated in a similar way as an invalid mutant in the context of strongly typed programming languages, e.g. not compiling.

A part of this Research Report is published in A.Derezińska, K.Hałas *Analysis of Mutation Operators for Python Programs,* AISC, Springer (2014) available at link.springer.com

In this paper we consider the mutation testing technique related to Python programs. A set of mutation operators that could be applied to Python programs is presented. Most of them correspond to selected operators used in strongly typed general purpose languages, covering structural and object-oriented mutation operators. They were adopted to the Python language. Operators designed for Python features are also discussed. We try to avoid operators that generate many incompetent mutants. All operators can be applied before a mutated program execution, but possible incompetent mutants are to be automatically detected at runtime.

The mutation operators discussed in the paper were implemented and experimentally examined using MutPy - a mutation testing tool for Python programs [14,15].

The reminder of this paper is organized as follows. In the next Section we discuss mutation testing of dynamically typed languages. Section 3 presents standard and object-oriented operators that can be used in Python programs. In Section 4, mutation operators that were designed for the Python language features are introduced. Section 5 discusses briefly MutPy and other tools for mutation testing of Python programs, and Section 6 concludes the paper.

2 Mutation of Dynamically Typed Programming Languages

Generation of mutants in a dynamically typed language can provide problems of type control. Without a knowledge about types of variables, a mutation tool could introduce an invalid mutation, i.e. a mutant execution will end with a type related exception. This problem can be illustrated by a Python code example. The original program includes the following code:

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

This function can be calculated in different ways. We show below three exemplary calling and their results returned by the function. In each case a different kind of addition was executed, namely arithmetic sum of numbers, concatenation of strings, and concatenation of lists. Variable *x* has a different type, but the addition operation is available for all these types.

```
>>>add(2)
4
>>>add(`abc`)
`abcabc`
>>>add([1, 2, 3])
[1, 2, 3, 1, 2, 3]
```

The code of the *add* function can be modified using a standard mutation operator. For example, if the AOR (*Arithmetic Operator Replacement*) mutation operator is applied the following mutant can be generated:

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

This mutant, similarly as the original function, can be run with various parameters. In the first case, where *x* has a numeric type, the mutant performs a subtraction operation which is defined for numbers. The result (0) differs from the value (4) of the original program, but the mutant is valid (competent).

However, subtraction is not specified for string or list type. Therefore in case of two remaining examples no value will be returned. After a call like this, Python will raise *TypeError* exception. Before the program execution, we could not determine type of variable *x* and, therefore, a subset of mutants generated by AOR would be *incompetent*.

Mutation that require substituting of variables could also generate incompetent mutants. Substituted variables can be of different types, which results in an error at run-time. During a program execution, it is also possible to change an object type assigned to an identifier. Therefore mutation of a dynamically typed language program could be considered to be applied during the program run-time, as proposed in [13].

Mutation at run-time could be realized using a meta-mutant approach. A mutation is specified by a meta-operator – a special function that is called at a mutation place. In this function all possible type combinations could be verified and omitted those that generate incompetent mutants. There are several problems associated with this solution. First, the number of type combinations can be high. Second, registration of all operations performed on the passed arguments is necessary in order to decide whether a mutated operation is allowed in a given context. It is especially difficult for the built-in objects.

Another drawback of a meta-mutant approach is a difficulty in transformation of the code with meta-mutants into a modified source code. The same code can have different representations in dependence on an object type used in a mutant call.

A mutant could have a different performance profile than the original program. Time relations of a program execution can be substantially different, causing problems, for example, in detection of an endless loop that is controlled by timeout.

Finally, application of meta-mutation does not avoid catching incompetent mutants during executing a mutant with tests. In Python, incompetent mutants can be detected after capturing the *TypeError* exception. A mutation system should omit such a mutant, and does not count its results to the overall mutation outcomes.

Therefore, in the MutPy mutation tool [14,15], a more pragmatic approach was implemented. All mutations are introduced into a source code (in fact into a code in the form of a syntax tree), and incompetent mutants are detected and eliminated at run-time. This mutation process requires a proper set of mutation operators suitable for Python programs.

3 Adaptation of Mutation Operators to the Python Language

The analysis of existing mutation operators was founded on operators designed for strongly typed general purpose languages. Operators implemented in the following four mutation testing tools, successfully used in many experiments, were taken into account:

- Mothra for Fortran [3],
- Proteum for C [4],
- MuJava for Java [7],
- CREAM for C# [16, 17].

A systematic review of the operators implemented in the tools was performed. Selecting an operator, we took into account its applicability to Python and reflecting 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 the program static analysis and preliminary experiments performed using the previous version of the MutPy tool.

Selection of the standard and object-oriented mutation operators is discussed in the next subsections. We also present description of all operators that were chosen to be implemented in MutPy version 3.0 [14,15].

3.1 Standard Mutation Operators Applied to Python

In strongly typed programming languages, a code replacement defined by a standard mutation operator can be easily 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 in the run-time.

A set of all standard operators implemented in four tools mentioned above was at first reduced by elimination of the overlapping functionality. If different operators deal with the same mutation those with three-letter names were chosen. If a scope of an operator A (i.e. mutants generated by A) is included in another operator (B), then the B operator was selected.

Next, an operator reduction concerned language constructions that do not exist in Python. For example, there are no *goto* and *do-while* statements, hence C-like operators that change such instructions are not applicable for Python programs.

In the following step, operators that generate many incompetent mutants were removed. Mutants of this kind can be detected and removed at runtime, but the whole process would be more costly to be performed. 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. Without knowing a type of a programming item to be substituted, 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 such mutants would be incompetent. Therefore, operators of this kind were not further considered.

A reduced set of standard mutation operators consists of 19 operators and is listed in **Table 1**. Operators from this set were analyzed with the reference to the Python language specification. The operators were combined with those implemented in the previous MutPy tool (version 0.2) and the scope of operators was examined in order to follow the mostly used conventions of operator names.

Table 1. Standard (structural) mutation operators - a preliminary reduced set

Operator		Mothra	Proteum	MuJava	CREAM
AOD	Arithmetic Operator Deletion	x		x	
AOR	Arithmetic Operator Replacement	x	x	x	x
ASR	Assignment Operator Replacement		x	x	x
COD	Conditional Operator Deletion			x	
COI	Conditional Operator Insertion		x	x	
COR	Conditional Operator Replacement			x	
CRP	Constant Replacement	x	x		
LCR	Logical Connector Replacement	x	x		x
LOD	Logical Operator Deletion			x	
LOR	Logical Operator Replacement		x	x	x
OAEA	Arithmetic Assignment by Plain Assignment		x		
ROR	Relational Operator Replacement	x	x	x	x
RSR	Return Statement Replacement	x	x		
SBRC	Break Replacement by Continue		x		
SCRB	Continue Replacement by Break		x		
SDL	Statement Deletion	x	x		
SMVB	Move Brace up and down		x		
SOR	Shift Operator Replacement		x	x	
UOR	Unary Operator Replacement				x

Operators OAEA and RSR were omitted due to a rare applicability to Python programs. The operator SMVB that changes a position of the block ending sign (‘}’) was not implemented, because of different approach to block denoting in Python, where an indentation corresponds to a block boarder. Functionality of SCRB was covered by the BCR (*Break Continue Replacement*) operator. Furthermore, the SOR operator functionality was added to the LOR operator.

The finally selected standard operators are discussed below. Each operator is also illustrated by an example of an original and mutated Python code.

AOD Arithmetic Operator Deletion.

The AOD operator deletes an unary arithmetic operator ‘+” or ‘-‘. Usage of ‘+’ operator seems to be superfluous, but the program behavior might not be the same as after a simple calling of *x* operand. In Python, operator '+' is usually used for forcing a calling of a special method__*pos*__ on an operand. This method can be implemented in a class, and overloads the unary operator '+' of class instances.

Examples before mutation: *after mutation:.*

```
- x        x
+ x        x
```

AOR Arithmetic Operator Replacement.
This mutation operator substitutes an arithmetic operator with another one. Substitution rules mimic possible errors of developers corresponding to similar arithmetic operations (like '/' and '//') or opposite ones (like '+' and '-'). The rules are given in **Table 2**.

Table 2. Substitution rules of arithmetic operators

Operator before mutation	+	-	*	*	*	/	/	//	//	**	%
Operator after mutation	-	+	/	//	**	*	//	*	/	*	*

Examples before mutation: *after mutation:*

```
x + y        x - y
x / y        x // y
```

ASR Assignment Operator Replacement.
The ASR operator replaces an extended assignment operator with another one according to the rules analogues to the AOR operator (**Table 2**).

Examples before mutation: *after mutation:*

```
x += y        x -= y
x *= y        x /= y
```

BCR Break Continue Replacement.
This operator swaps keywords *break* and *continue* in a loop body.

Example before mutation: *after mutation:*

```
1 for x in y:          for x in y:
2     …….                  …..
3     break                continue
```

COD Conditional Operator Deletion.
The COD operator deletes unary negation operator *not*. It also deletes the negation of an membership operator *not in*.

Example before mutation: *after mutation:*

```
if not x:                    if  x:
```

COI Conditional Operator Insertion
The opposite action to the previous operator is realized by the COI operator. However, in order to avoid creating of many incompetent mutants, its application was limited to headers of a conditional instruction *if* and a loop *while*, as well as a membership operator *in*.

Example before mutation: *after mutation:*

```
if x:                        if not x:
```

CRP Constant Replacement .
This operator changes numeric and string literals. A numeric literal is substituted by an incremented value. A string literal is substituted by another string or an empty one (2 mutants).

Examples before mutation: *after mutation:*

```
x = 1                               x = 2
z = `test`                          z = `mutpy`
```

LCR Logical Connector Replacement
The LCR operator swaps logical operators *and* with *or* and vice versa.

Examples before mutation: *after mutation:*

```
if x or y:                         if x and y:
while x and y:                     while x or y:
```

LOD Logical Operator Deletion
This operator deletes an unary operator of bit negation `~`.

Example before mutation: *after mutation:*

```
~ x                                 x
```

LOR Logical Operator Replacement
The LOR operator substitutes bitwise operators. Program mutations intend to mimic potential mistakes, analogous to the case of the AOR operator. Substitution rules are given in **Table 3**.

Table 3. Substitution rules of logical operators

Operator before mutation	&	|	^	<<	>>
Operator after mutation	|	&	&	>>	<<

Examples before mutation: *after mutation:*

```
x & y                                x | y
x << y                               x >> y
```

OIL One Iteration Loop.
After application of the OIL operator, a loop can be executed only once. The operator is implemented by adding a *break* statement at the end of a *for* or *while* loop body. This operator was finally recognized as injecting an artificial program modification and was classified as a trail mutation operator.

Example before mutation: *after mutation:*

```
1 for x in y:                    for x in y:
2   ……….
3                                    break
```

ROR Relational Operator Replacement
This mutation operator swaps relational operators. Substitution rules implemented by this operator (**Table 4**) refer to similar relational operators (eg. `<' vs `<=`) or opposite ones (eg. `<` vs `>`).

Table 4. Substitution rules of relational operators

Operator before mutation	<	<	>	>	<=	<=	>=	>=	==	!=
Operator after mutation	>	<=	<	>=	>=	<	<=	>	!=	==

Examples before mutation: *after mutation:*

```
if x < y:                            if x > y:
while i >= 10:                       while i > 10:
```

ZIL Zero Iteration Loop.
The ZIL operator interrupts realization of a loop during its first iteration. This operator is implemented by a substitution of a loop body by a *break* statement.

Example before mutation: *after mutation:*

```
for x in y:                      for x in y:
  ……….                               break
```

3.2 Object-Oriented Mutation Operators Applied to Python

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

The Python language supports basic object-oriented concepts, thus we also try to adopt object-oriented mutation operators used in Java or C#. At first, a set of 36 object-oriented operators implemented in MuJava [7] or CREAM [16,17] was examined.

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, these object-oriented operators were omitted that could be defined in Python, but require type-based verification during application. For example, operators PRM (*Property Replacement with Member Field*) and PRV (*Reference Assignment with other Compatible Type*) were omitted. Without type verification, these operators would mostly generate incompetent mutants.

Below we present object-oriented mutation operators that were included in the final operator set and implemented in the MutPy tool.

EXD Exception Handler Deletion
The EXD operator deletes an exception handler block. In the Mutpy implementation, this effect is realized by inserting of a *raise* statement into an *except* block. In this way we can re-raise the exception. In result, the exception is raised, but not handled.

Example before mutation: *after mutation:*

```
try:                             try:
    …….                                  …..
except Exception:                except Exception:
  ….                                 raise
```

EXS Exception Swallowing
After applying the EXS operator, an exception handler block is not performed, similarly to a mutant created with the EXD operator. But in case of EXS, the exception is captured. This functionality was realized by substitution of an exception handling block with a *pass* statement, which stands in Python for a nothing-to-do instruction.

Example before mutation: *after mutation:*

```
try:                              try:
    …….                               …..
except Exception:                 except Exception:
  ….                                  pass
```

IHD Hiding Variable Deletion
This operator deletes a class variable that was re-assigned in a subclass and, therefore, hides the variable from the superclass. In Python, a variable can be declared in a class in two following ways:

1) inside a class constructor (__*init*__), these variables are accessible by calling of a class instance,
2) inside a class definition, these variables are accessed within the class (the concept is similar to *static* fields used in Java or C#).

While considering a variable defined in a constructor (case 1), it is sometimes impossible to determine which variable is hidden, due to dynamic typing in Python and the necessity of a run-time analysis of the constructor execution. Therefore, the IHD operator was limited to the second case, where a class (static) variable is hidden.

In the Mutpy tool, the operator is implemented by substitution of a given statement by a *pass* statement.

Example before mutation: *after mutation:*

```
class X:                        class X:
    y = 1                           y = 1
class Z(X):                     class Z(X):
    y = 2                           pass
```

There is an operator complementary to IHD used in Java and C#. It is IHI (*Hiding Variable Insertion*). IHI hides a super class variable in a subclass when variables are of the same name. Python has other rules then Java and C#. A method or a variable in a subclass always hides an element from the superclass. Therefore, IHI was not applied to Python.

IOD Overriding Method Deletion
Operator IOD deletes a method that was overridden in a subclass. The implementation is based on the substitution of the method by a *pass* statement, like in the IHD operator.

Example before mutation: *after mutation:*

```
class X:                        class X:
    def foo(self):                  def foo(self):
class Y(X):                     class Y(X):
    def foo(self):              pass
```

IOP Overridden Method Calling Position Change
A method of a super class can be overwritten in its subclass. The original method, i.e. a method from the base class, can be still called in the body of the method in subclass. After application of the IOP operator, a calling of the original method is placed in another position in the subclass. The original method calling is moved to the end of the method in the subclass. If the calling have already been at the end, it is moved to the beginning of the subclass method. If the overridden method consists of only one statement, namely the super class calling, the operator is not applied. If the operator had been applied in a case like that, an equivalent mutant would have been generated.

Example before mutation: *after mutation:*

```
class X:                     class X:
    def foo(self):               def foo(self):
      …………                         …….
class Y(X):                  class Y(X):
    def foo(self):               def foo(self):
      super().foo()                …………. 
      ………….                       super().foo()
```

SCD Super Calling Deletion.
The operator deletes a calling of an original method from a method body that hides the original one. This operator covers the functionality of IPC (Explicit Call of a Parent's Constructor Deletion) and ISD (Super Keyword Deletion) applied for Java. The SCD operator deletes the calling of a super class method not only from a constructor (as in IPC) but also from any other method. In Python, a calling of a superclass method is realized in the same way using a build-in *super* method.

Example before mutation: *after mutation:*

```
class X:                     class X:
    def foo(self):               def foo(self):
      …………                       …….
class Y(X):                  class Y(X):
    def foo(self):               def foo(self):
      super().foo()                ………….
```

SCI Super Calling Insert.
This operator is complementary to SCD. It inserts a calling of an original method (a superclass method) to the body of the method that overwrites the superclass-method in a subclass.

Example before mutation:

```
class X:
    def foo(self):
      …………
class Y(X):
    def foo(self):
        ………….
```

after mutation:

```
class X:
    def foo(self):
    …….
class Y(X):
    def foo(self):
        super().foo()
```

SVD Self Variable Deletion.
The SVD operator deletes a *self* variable from a method body. It is analogous to the JTD operator (*This Keyword Deletion*) used for Java and C#, which deletes a keyword *this*. In Python, there is no *this* keyword, but a reference to a class instance can be passed as a first method argument. It is usually called *self*. Syntactically the mechanism is similar to those from Java and C#, but a usage of the *self* reference is obligatory and not optional as in Java and C#. In result, many invalid mutants can be generated. Therefore, the SVD operator was finally classified as a trial one.

Example before mutation:

```
class X:
    def foo(self):
      return  self.x
```

after mutation:.

```
class X:
    def foo(self):
        return x
```

Another operator dealing with *this* keyword in Java was JTI (This Keyword Insertion). It was not transformed to Python, because the obligatory usage of keyword *self* makes functionality of this operator pointless.

4 Python-Related Mutation Operators

Apart from the mutation operators used in the above-mentioned languages, operators related to the Python language were proposed. Various programing structures were considered, such as decorator, index slice, loop reversing, membership relation, exception handling, variable and method handling, *self* variable [12]. Some Python-specific notions have already been taken into account in the scope of operators discussed in the previous Section. Here, we discussed new mutation operators designed especially for Python.

4.1 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: *after mutation:* .

```
class X:                          class X:
    @classmethod
    def foo(cls):                   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 during the program 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 not included into the final set of Python mutation operators. Both operators insert a decorator to a method definition, namely CDI inserts *@classmethod* and SDI *@staticmethod.* However, according to the program analysis and preliminary experiments, such operators inserting a new element into a program are usually a source of incompetent mutants. Insertion-based mutation of this kind is syntactically correct and can be applied in many places of a program, but it is difficult to indicate whose places where this change is reasonable justified.

4.2 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 *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 such removal gives a syntactically correct slice.

Example before mutation:	after mutation:
`x[1:-1]`	`x[:-1]`

4.3 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 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):  :`

5 MutPy - a Tool for Mutation Testing Python Programs

MutPy [14,15] is a mutation testing tool for Python 3.x source code. Twenty mutation operators specified in the previous Sections were implemented in the current version of the tool (**Table 5**). Operator category is denoted by S - structural and OO - object-oriented. Two operators (DDL and SDL) could be classified to both categories. The table includes operators adopted to Python and new Python-related operators, the latter marked in the *Python-related* column.

Table 5. 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		
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/OO		v
SVD	Self Variable Deletion	OO		v
ZIL	Zero Iteration Loop	S		v

The last column indicates trial operators that were considered for Python programs, but finally did not include in the recommended set of mutation operators supported by the MutPy tool. According to the program analysis and preliminary experiments they provide many incompetent mutants and/or do not significantly contribute to the quality evaluation of tests or mutation-based test generation.

The set of mutation operators implemented in MutPy is so far the most comprehensive among all mutation tools for Python programs. It is the only tool supporting object-oriented operators in Python. The tool applies mutation on the abstract syntax tree (AST) level. MutPy supports higher order mutation [1] and code coverage analysis as mutation process boost techniques. It could generate reports in YAML and HTML format.

The first tool for mutation testing of Python programs - Pester [18] was very simple and did not consider any mutations that could generate incompetent mutants. Moreover, it has been not updated for few years.

Other tools stand for a proof of concept with 2-3 operators. PyMuTester [19] supports two operators *IfCondition Negation* and *Loop Skipping*. Mutant [20] implements three mutation operators: *ComparisonMutation*, *ModifyConstantMutation*, and *JumpMutation*.

The most mature tool, except MutPy, is Elcap [21]. It supports eight structural operators, such as: *StringMutator*, *NumberMutator*, *ArithmeticMutator*, *ComparisonMutator*, *LogicalMutator*, *FlowMutator*, *YieldMutator* and *BooleanMutator*.

6 Conclusion

Operators presented in the paper can be applied to Python programs. They cover a wide range of possible mistakes of developers, concerning all basic programming expressions, object-oriented language features and specific constructions of Python. Therefore, they can be effective in evaluation of a test suite.

Appling mutation to Python programs before run-time, incompetent mutants can be generated. Existence of incompetent mutants decreases the efficiency of the mutation process. Incompetent mutants are run against tests but their outcome is not calculated to the final mutation results. However, executing and handling these mutants increases the overall mutation time.

Selecting the discussed mutation operators, their ability not to generate the majority of incompetent mutants was taken into account. Experiments performed on different programs using this set of operators confirmed that on average only 5.3% of generated mutants were incompetent [14]. Moreover, the incompetent mutants were handled by the MutPy tool and did not influence the mutation results.

References

1. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649-678 (2011)
2. Voas, J.M., McGraw G.: Software Fault Injection, Inoculating Programs against Errors, John Wiley & Sons Inc. (1998)
3. King K. N., Offutt A. J.: A Fortran Language System for Mutation-based Software Testing. Software - Practice and Experience, 21(7), pp. 685-718 (1991)

4. Delamaro M. E., Maldonado J. C.: Proteum-A tool for the assessment of test adequacy for C programs. In: Proc. of the Conference on Performability in Computing Systems (PCS 96), pp. 79–95 (1996)
5. 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)
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), pp. 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. Bradbury J.S., Cordy J.R., Dingel J.: Mutation Operators for Concurrent Java (J2SE 5.0). In: Proc. of the Workshop on Mutation Analysis (Mutation 2006), pp. 83-92 (2006)
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. Python Programming Language, http://python.org
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. 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)
15. MutPy, https://bitbucket.org/khalas/mutpy
16. 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, Springer Berlin Heildelberg, pp. 42–57 (2012)
17. CREAM, http://galera.ii.pw.edu.pl/~adr/CREAM
18. Jester - the JUnit test tester, http://jester.sourceforge.net/
19. PyMutester,http://miketeo.net/wp/index.php/projects/python-mutant-testing-pymutester
20. Mutant, http://github.com/mikejs/mutant
21. Elcap, http://githup.com/sk-/elcap