Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence

APPENDIX A
FORMATIVE STUDY

Table I shows representative discussions about performance impact of nine Python idioms on Stack Overflow. #N represents the number of questions we find for each Python idiom and blue text represents the view times that more than 1k times indicates the questions are popular.

(1) Developers are concerned with the performance of Python idioms. First, developers ask whether idiomatic code is faster or slower than non-idiomatic code. For example, for the star-in-func-call in Table I, developers are interested in whether the star operator affects the performance. Furthermore, developers would like to know how many times idiomatic code may be faster or slower than the non-idiomatic code. For example, for the list-comprehension in Table I, a developer states that the list comprehension is 50% faster than appending to a list with for loop. Finally, developers want to know the reasons why Python idioms causes the performance change. For example, for the for-multiple-targets in Table I, a developer ask why accessing by index slow things down compared to for-multiple-targets.

(2) Many questions lack clear evidence of the performance impact of Python idioms. Only 49.5% questions list code fragments and corresponding execution time to illustrate the performance of Python idioms. For example, for the chain-comparison in Table I, developers write a toy code pair of `x<y<z` and `x<y and y<z` to compare their execution time. Furthermore, for the two questions regarding loop-else and star-in-func-call, they lack code fragments or execution time, but provide only natural language descriptions for the performance of the idioms. For example, for the loop-else in Table I, a developer say "I was introduced to a wonderful idiom in which you can use a for/break/else scheme with an iterator to save both time and lines of code".

(3) Developers are often confused by the controversial descriptions or evidences of the performance of Python idioms. 33 out of 101 question threads present some contradictory performance results. Among the nine examples in Table I, six questions discuss the contradictory performance results, annotated with a *. For example, for the assign-multi-targets in Table I, a developer states that multiple assignment perform at least 30% better than the individual assignment. However, the developer shows that the Python official wiki states that multiple assignment is slower than of individual assignment. Similarly, for the chain-comparison in Table I, a developer finds the chain comparison is slower, but another developer states that the Python official wiki claims that chained comparisons are faster than using "and" operator. As another example, for the truth-value-test in Table I, it is generally understood that this idiom does not impact the performance largely. However, the developer encounter a scenario that the truth-value-test can shave 30% off runtime.

TABLE I: Python Idiom Performance Related SO Questions

Idiom	#N	Question
List-Comprehension*	26	**Question:** How to speed up list comprehension? my understanding is that for-loop is faster than list comprehension. **Comments:** In all cases I've measured the time a list comprehension was always faster than a standard for loop. (2k times)
Set-Comprehension*	12	**Question:** How do python Set Comprehensions work? I tried timeit for speed comparisons, there is quite some difference. **Answer:** List/Dict/Set comprehensions tend to be faster than anything else. (2k times)
Dict-Comprehension*	15	**Question:** Why is this loop faster than a dictionary comprehension? **Comment:** ... I do this with a dictionary with 1000 random keys and values, the dictcomp is marginally slightly faster. (9k times)
Chain-Comparison*	6	**Question:** Is "x < y < z" faster than "x < y and y < z"? In this page, we know that chained comparisons are faster than using the "and" operator. However, I got a different result... It seems that `x < y` and `y < z` is faster than `x < y < z`. (11k times)
Truth-Value-Test*	17	**Question:** bool value of a list in Python. **Answer:** 99.9% of the time, performance doesn't matter as suggested Keith. I only mention this because I once had a scenario, using implicit truthiness testing shaved 30% off the runtime. (31k times)
Loop-Else	7	**Question:** Pythonic ways to use 'else' in a for loop. **Answer:** I was introduced to a wonderful idiom in which you can use a for/break/else scheme with an iterator to save both time and LOC. (1k times)
Assign-Multi-Targets*	8	**Question:** Python assigning two variables on one line I've been looking to squeeze a little more performance out of my code; While browsing this Python wiki page, I found this claim: Multiple assignment is slower than individual assignment. I repeated several times, but the multiple assignment snippet performed at least 30% better than the individual assignment. (3k times)
Star-in-Fun-Call	5	**Question:** What does the star mean in a function call? Does it affect performance at all? Is it fast or slow? (245k times)
For-Mul-Targets	5	**Question:** How come unpacking is faster than accessing by index? (3k times)

Python developers are concerned with the performance of Python idioms. However, their evidences of performance improvement or regression are generally anecdotal based on either toy code or individual project experience. This leads to many controversies about if and when developers should or should not use Python idioms.

APPENDIX B
EMPIRICAL STUDY SETUP

A. Data Collection

Table II summarizes the number of code pairs (non-idiomatic versus idiomatic) in the synthetic dataset and the real-project dataset. For synthetic list/set/dict-comprehension code pairs, 1600 is computed by multiplying 4 numFor, 5 numIf, 5 numIfElse, 2 and 8 values (local or global) of variable scope and `size`. For synthetic chained comparison code, 11968 is computed by multiplying 2992 combinations of 2-5 comparisons from `CompopSet`, 2 and 2 values of variable scope and `isTrue`. For synthetic assign-multiple-targets, when the values are constants, the `isSwap` can only

TABLE II: The statistics of Synthetic and Real-Project Dataset

Idiom	Synthetic	Real-Project
List Comprehension	1600	734
Set Comprehension	1600	282
Dict Comprehension	1600	194
Chain Comparison	11968	2268
Truth Value Test	336	40116
Loop Else	128	198
Assign Multiple Targets	174	10583
Star in Func Call	1920	170
For Multiple Targets	4800	334
Total	24126	54879

be 0, so there are only 174 code pairs instead of 232. For synthetic truth-value-test, 336 is computed by multiplying 3, 2, 14, 2 and 2 values of `TestSet`, `EqSet`, `EmptySet`, scope and `IsTrue`. For synthetic loop-else, the 128 is computed by multiplying the 2, 2, 2, 8 and 2 values of `LoopSet`, `ConditionSet`, variable scope, `size` and `isBreak`. For synthetic star-in-fun-call, the 1920 is computed by multiplying the 30 numSubscript, 2, 2, 2, 2, 2 and 2 values of `hasSubscript`, `hasStep`, `hasLower`, `hasUpper`, variable scope and `isConst`. For synthetic for-multi-targets, the 4800 is computed by multiplying 30 numSubscript, 5 numTarget, 2, 2, 8 values of `hasStarred`, scope and `size`. For real-project code, the number of code pairs are reported by the refactoring tool (accompanied by at least one test case to execute the before- and after-refactoring code).

APPENDIX C
EMPIRICAL ANALYSIS

A. RQ2: How well can code features explain the performance differences caused by Python idioms?

Data size: Fig. 1 shows the relationship between `size` and performance speedup for set-comprehension and dict-comprehension. To see the trend clearly, we take the log of data size. For the set-comprehension, the speedup increases fast when the number of elements increases from 1 to 148(e^5). However, the speedup flattens when the number of elements increases over about 2981 (e^8). For the dict-comprehension, the speedup increases fast when the number of elements increases from 1 to 20 (e^3). However, the speedup flattens when the number of elements increases over about 2981 (e^8).

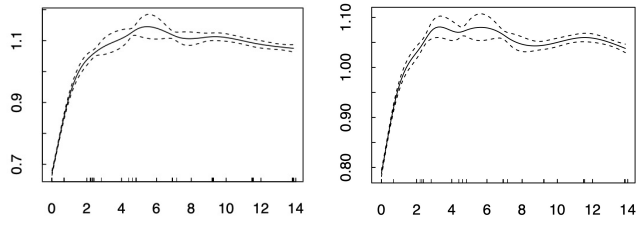

Fig. 1: The Relationship Between `size` and Performance Speedup for Set Comprehension and Dict Comprehension.

B. RQ3: What are the root causes of performance differences caused by Python idioms and what cause the inconsistencies between synthetic and real-project code?

Fig. 2 shows the differences in bytecode instructions between non-idiomatic code and idiomatic code for nine Python idioms.

- **List-comprehension:** From the green box of List-Comprehension in Fig. 2, we could see that the *list-comprehension* needs to execute additional preparation instructions to load listcomp code object and then call it as function before loop iteration. From the blue box of List-Comprehension in Fig. 2, we could see the non-idiomatic code executes the LOAD_* instructions to push data into the stack and then calls the append function to append the element. In contrast, the idiomatic code only executes the LIST_APPEND instruction to append the element.
- **Set-comprehension:** Similar to the *list-comprehension*, *set-comprehension* needs to execute additional preparation instructions to load setcomp code object and then call it as function before loop iteration (green box of Set-Comprehension in Fig. 2). For adding elements, its non-idiomatic code needs to execute the LOAD_* instructions to push data into the stack and then call the add function to add the element, but the idiomatic code only needs to execute the SET_ADD instruction to append the element (blue box of Set-Comprehension in Fig. 2). Besides, non-idiomatic needs to call a function to initialize an empty set, but the *set-comprehension* does not need to do it (red box of Set-Comprehension in Fig. 2).
- **Dict-comprehension:** Similar to the *list-comprehension* and *set-comprehension*, the *dict-comprehension* needs to execute additional preparation instructions to load dictcomp code object and then call it as function before loop iteration (green area of Dict-Comprehension in Fig. 2). However, different from the *list-comprehension* and *set-comprehension*, for appending the element, the non-idiomatic code of the *dict-comprehension* does not need to call a function, but only needs to load the object and store a key-value pair. In contrast, the *dict-comprehension* directly executes MAP_ADD to add the element (blue box of Dict-Comprehension in Fig. 2).
- **Chain-comparison:** The *chain-comparison* replaces one instruction to load the chained comparator o of non-idiomatic code with the instructions to create a reference to the chained comparator o (DUP_TOP) and reference shuffling (ROT_THREE) for the three comparators (n, o, p) (blue box of Chain-Comparison in Fig. 2).
- **Truth-value-test:** From the red box of Truth-Value—Test in Fig. 2, we can see The *truth-value-test* removes instructions of loading the object 0 from `EmptySet` and the comparison operation !=.
- **Loop-else:** The non-idiomatic code determines whether the code executes a break statement by setting a different value to the flagging variable flag. After the for statement is executed, if the flag is True, code executes the body of the if statement. Operations flagging variables and comparison operation is not concise for developers, so the *loop-else* removes the instruc-

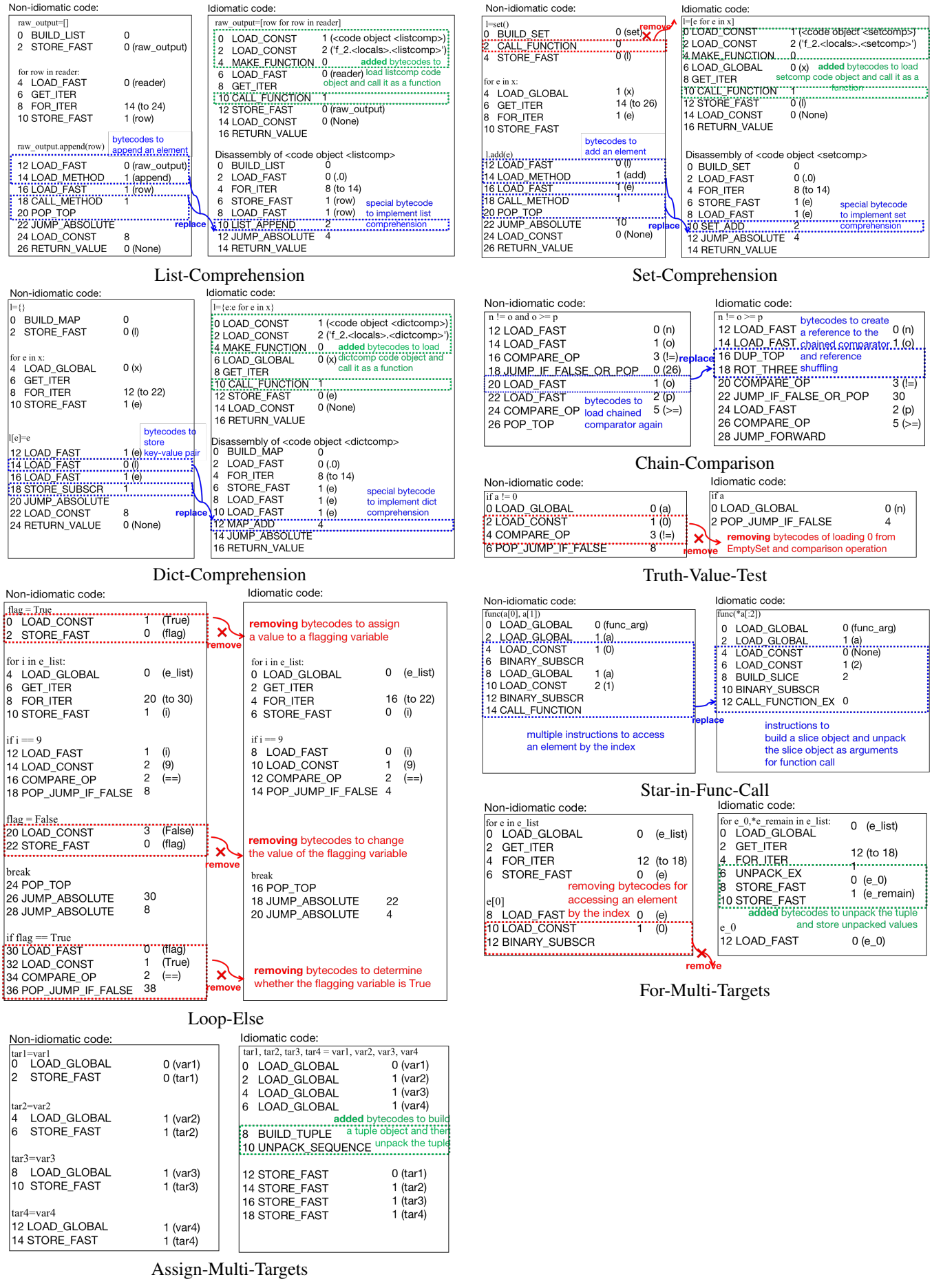

Fig. 2: Bytecode Instructions of Python Idioms (Right) and Corresponding Non-Idiomatic Codes (Left)

tions of setting flagging variables and comparison operation of non-idiomatic code (red box in Loop-Else in Fig. 2).

- **Assign-multi-targets:** Compared to the non-idiomatic code, the idiomatic code additionally executes the BUILD_TUPLE instruction to build a tuple and another UNPACK_SEQUENCE instruction to unpack the sequences to put values onto the stack right-to-left (green box of Assign-Multi-Targets in Fig. 2).
- **Star-in-func-call:** The *star-in-func-call* replaces multiple instructions to load an element by index (BINARY_SUBSCR) with instructions to build a slice object and unpack the slice object into arguments (blue box of Star-in-Func-Call in Fig. 2).
- **For-multi-targets:** In the body of the for statement, the *for-multi-targets* removes instructions to access an element by index (red box of For-Multi-TargetsFig. 2). However, the *for-multi-targets* additionally executes instructions to unpack the object and then store the unpacked values (green box of For-Multi-TargetsFig. 2).

APPENDIX D
CODE EXAMPLES IN TWO DATASETS

Fig. 3 and Fig. 4 show code examples of non-idiomatic code and the corresponding idiomatic code for nine Python idioms in the Synthetic DataSet. And Fig. 5 and Fig. 6 show code examples of non-idiomatic code and the corresponding idiomatic code for nine Python idioms in the Real-Project DataSet. We show the relevant code features in the figures.

```
def func_a():
  x_0 = []
Non-idiomatic code:
  l = []
    for e_0 in x_0:
      l.append(e_0)

Idiomatic code:
l = [e_0 for e_0 in x_0]
```
numFor: 1
numIf: 0
numIfElse: 0
size: 0
scope: Local

```
x_0 = list(range(1, 1001))
x_1 = list(range(1, 1001))

Non-idiomatic code:
l = []
for e_0 in x_0:
    for e_1 in x_1:
        if e_0 % 2:
            l.append(e_0)
        else:
            if e_0 % 2:
                l.append(e_0)
            else:
                l.append(e_0)

Idiomatic code:
l = [e_0 if e_0 % 2 else e_0 if
e_0 % 2 else e_0 for e_0 in x_0
for e_1 in x_1]
```
numFor: 2
numIf:0
numIfElse: 2
size: 1000000
scope: Global

```
x_0 = list(range(1,11))
x_1 = list(range(1,11))
x_2 = list(range(1,11))
x_3 = list(range(1,11))

Non-idiomatic code:
l = []
for e_0 in x_0:
    for e_1 in x_1:
        for e_2 in x_2:
            for e_3 in x_3:
                if e_0//1:
                    if e_0//1:
                        if e_0//1:
                            if e_0%2:
                                l.append(e_0)
                            else:
                                l.append(e_0)

Idiomatic code:
l = [e_0 if e_0 % 2 else e_0 for e_0 in
x_0 for e_1 in x_1 for e_2 in x_2 for
e_3 in x_3 if e_0 // 1 if e_0 // 1 if e_0
// 1]
```
numFor: 4
numIf: 3
numIfElse: 1
size: 10000
scope: Global

List-Comprehension

```
def func_a():
  i_s=0
  e_list=[i for i in range(4)]
Non-idiomatic code:
func_arg(e_list[i_s],e_list[i_s+1],
        e_list[i_s+2])
Idiomatic code:
func_arg(*e_list[i_s: i_s+3])
```
numSubscript: 3
hasSubscript: 1
hasStep: 0
hasLower: 1
hasUpper: 1
isConst: 0
scope: Local

```
e_list=[i for i in range(6)]
Non-idiomatic code:
func_arg(e_list[1],e_list[3])
Idiomatic code:
func_arg(*e_list[1: 4: 2])
```
numSubscript: 2
hasSubscript: 1
hasStep: 1
hasLower: 1
hasUpper: 1
isConst: 1
scope: Global

```
e_list = [i for i in range(4)]

Non-idiomatic code:
func_arg(e_list[0], e_list[1],
e_list[2], e_list[3])

Idiomatic code:
func_arg(*e_list)
```
numSubscript: 4
hasSubscript: 0
hasStep: 0
hasLower: 0
hasUpper: 0
isConst: 1
scope: Global

Star-in-Func-Call

```
def func_a():
  x_0 = list(range(1,1000001))

  Non-idiomatic code:
  l = set()
  for e_0 in x_0:
      if e_0//1:
          if e_0%2:
              l.add(e_0)
          else:
              l.add(e_0)

  Idiomatic code:
  l = {e_0 if e_0 % 2 else e_0
  for e_0 in x_0 if e_0 // 1}
```
numFor: 1
numIf: 1
numIfElse: 1
size: 1000000
scope: Local

```
x_0 = list(range(1,101))
x_1 = list(range(1,101))
x_2 = list(range(1,101))

Non-idiomatic code:
l = set()
for e_0 in x_0:
    for e_1 in x_1:
        for e_2 in x_2:
            if e_0//1:
                if e_0//1:
                    if e_0%2:
                        l.add(e_0 * 100000 +
e_1 * 1000 + e_2)
                    else:
                        l.add(e_0 * 100000 +
e_1 * 1000 + e_2)

Idiomatic code:
l = {e_0 * 100000 + e_1 * 1000 + e_2 if
e_0 % 2 else e_0 * 100000 + e_1 *
1000 + e_2 for e_0 in x_0 for e_1 in
x_1 for e_2 in x_2 if e_0 // 1 if e_0 // 1}
```
numFor: 3
numIf: 2
numIfElse: 1
size: 1000000
scope: Global

```
x_0 = []
x_1 = []
x_2 = []
x_3 = []

Non-idiomatic code:
l = set()
for e_0 in x_0:
    for e_1 in x_1:
        for e_2 in x_2:
            for e_3 in x_3:
                if e_0//1:
                    if e_0//1:
                        if e_0%2:
                            l.add(e_0 * 1000000 +
e_1 * 10000 + e_2 * 100 + e_3)
                        else:
                            if e_0%2:
                                l.add(e_0 * 1000000 +
e_1 * 10000 + e_2 * 100 + e_3)
                            else:
                                l.add(e_0 * 1000000 +
e_1 * 10000 + e_2 * 100 + e_3)

Idiomatic code:
l = {e_0 * 1000000 + e_1 * 10000 + e_2 *
100 + e_3 if e_0 % 2 else e_0 * 1000000 +
e_1 * 10000 + e_2 * 100 + e_3 if e_0 % 2
else e_0 * 1000000 + e_1 * 10000 + e_2 *
100 + e_3 for e_0 in x_0 for e_1 in x_1 for
e_2 in x_2 for e_3 in x_3 if e_0 // 1 if e_0 //
1}
```
numFor: 4
numIf: 2
numIfElse: 2
size: 0
scope: Global

Set-Comprehension

```
def func_a():
    x_0 = list(range(1,101))

    Non-idiomatic code:
    l = dict()
    for e_0 in x_0:
        if e_0%2:
            l[e_0]=e_0
        else:
            l[e_0]=e_0

    Idiomatic code:
    l = {e_0: e_0 if e_0 % 2 else
e_0 for e_0 in x_0}
```
numFor: 1
numIf: 0
numIfElse: 1
size: 100
scope: Local

```
x_0 = list(range(1,101))
x_1 = list(range(1,101))

Non-idiomatic code:
l = dict()
for e_0 in x_0:
    for e_1 in x_1:
        if e_0//1:
            if e_0//1:
                if e_0//1:
                    if e_0%2:
                        l[e_0 * 10000 +e_1]=e_0
                    else:
                        l[e_0 * 10000 +e_1]=e_0

Idiomatic code:
l = {e_0 * 10000 + e_1: e_0 if e_0 % 2
else e_0 for e_0 in x_0 for e_1 in x_1 if
e_0 // 1 if e_0 // 1 if e_0 // 1}
```
numFor: 2
numIf: 3
numIfElse: 1
size: 10000
scope: Global

```
x_0 = list(range(1,101))
x_1 = list(range(1,101))
x_2 = list(range(1,101))

Non-idiomatic code:
l = dict()
for e_0 in x_0:
    for e_1 in x_1:
        for e_2 in x_2:
            if e_0//1:
                if e_0%2:
                    l[e_0 * 100000 + e_1 * 1000 +
e_2]=e_0
                else:
                    if e_0%2:
                        l[e_0 * 100000 + e_1 * 1000 +
e_2]=e_0
                    else:
                        l[e_0 * 100000 + e_1 * 1000 +
e_2]=e_0

Idiomatic code:
l = {e_0 * 100000 + e_1 * 1000 + e_2: e_0 if
e_0 % 2 else e_0 if e_0 % 2 else e_0 for e_0
in x_0 for e_1 in x_1 for e_2 in x_2 if e_0 // 1}
```
numFor: 3
numIf: 1
numIfElse: 2
size: 1000000
scope: Global

Dict-Comprehension

Fig. 3: Code Examples for List/Set/Dict-Comprehension and Star-in-Func-Call in the Synthetic DataSet

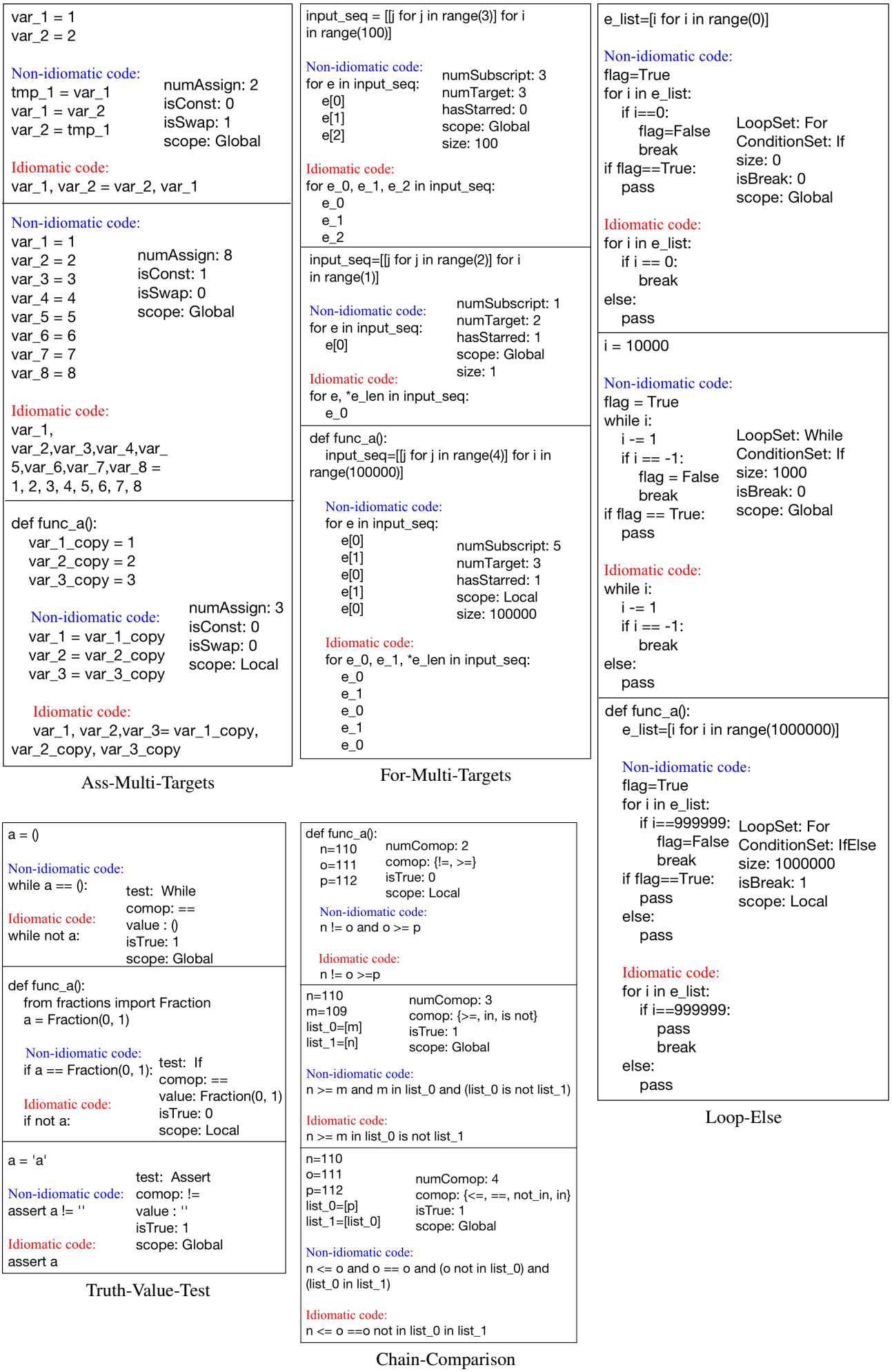

Fig. 4: Code Examples for Assign-Multi-Targets, For-Multi-Targets, Loop-Else, Truth-Value-Test and Chain-Comparison in the Synthetic DataSet

Non-idiomatic code:
```
explicit_bindings = []
for cls in classes:
    if decorators.is_explicitly_injectable(cls):
        for arg_name in get_arg_names_from_class_name(cls.__name__):
            explicit_bindings.append(new_binding_to_class(
                binding_keys.new(arg_name), cls, scoping.DEFAULT_SCOPE,
                lambda cls=cls: locations.get_loc(cls)))
```
numFor: 2
numIf: 1
numIfElse: 0
Involving **function calls, attribute accesses, lambda expression**

Idiomatic code:
```
explicit_bindings = [new_binding_to_class(binding_keys.new(arg_name), cls, scoping.DEFAULT_SCOPE,
lambda cls=cls: locations.get_loc(cls)) for cls in classes if decorators.is_explicitly_injectable(cls) for
arg_name in get_arg_names_from_class_name(cls.__name__)]
```

```
reader = csv.DictReader(cleandata, dialect=dialect)
```

Non-idiomatic code:
```
for row in reader:
    raw_output.append(row)
```
numFor: 1
numIf: 0
numIfElse: 0
reader is a reader object from csv library which **overloads the __next__ method,** which is the bottle neck

Idiomatic code:
```
raw_output = [row for row in reader]
```

Non-idiomatic code:
```
for (it, n) in zip(iterables, offsets):
    if n < 0:
        staggered.append(chain(repeat(fillvalue, -n), it))
    elif n > 0:
        staggered.append(islice(it, n, None))
    else:
        staggered.append(it)
```
numFor: 1
numIf: 0
numIfElse: 2
invovling 0~2 function calls for each append element

Idiomatic code:
```
staggered = [chain(repeat(fillvalue, -n), it) if n < 0 else
islice(it, n, None) if n > 0 else it for (it, n) in zip(iterables,
offsets)]
```

Non-idiomatic code:
```
for cluster in range(2):
    for i in range(length):
        for j in range(i + 1, length):
            edges.append(('{}:{}'.format(cluster, i), '{}:{}'.format(cluster, j)))
```
numFor: 3
numIf: 0
numIfElse: 0
2 function calls

Idiomatic code:
```
edges = [('{}:{}'.format(cluster, i), '{}:{}'.format(cluster, j)) for cluster in range(2) for i in range(length)
for j in range(i + 1, length)]
```

List-Comprehension

Non-idiomatic code:
```
kvstore = {}
for row in reader:
    kvstore[row[0]] = row[1]
```
numFor: 1
numIf: 0
numIfElse: 0
reader is a reader object from csv libaray where overload the __next__ method

Idiomatic code:
```
kvstore = {row[0]: row[1] for row in reader}
```

Non-idiomatic code:
```
ret = {}
for (name, entry) in read_index(f):
    ret[name] = entry
```
numFor: 1
numIf: 0
numIfElse: 0
read_index is a generator function which yield will prevent the function from exiting, until the next time next() is called

Idiomatic code:
```
ret = {name: entry for (name, entry) in
read_index(f)}
```

Non-idiomatic code:
```
rc_dict = {}
for section in config.sections():
    for (name, value) in config.items(section):
        rc_dict[name] = _colon_split(value)
```
numFor: 2
numIf: 0
numIfElse: 0
multiple function calls

Idiomatic code:
```
rc_dict = {name: _colon_split(value) for section in config.sections() for
(name, value) in config.items(section)}
```

Non-idiomatic code:
```
equations = {}
for col in dependent.pandas:
    equations[col] =
(dependent.pandas[[col]],
exog.pandas, endog.pandas,
 instr.pandas)
```
numFor: 1
numIf: 0
numIfElse: 0
multiple attribute accesses

Idiomatic code:
```
equations = {col: (dependent.pandas[[col]], exog.pandas, endog.pandas,
instr.pandas) for col in dependent.pandas}
```

Non-idiomatic code:
```
filter = {}
for (status, res_data) in six.iteritems
(responses):
    if isinstance(status, int) or status.isdigit():
        filter[int(status)] = dict(headers=res_data.get('headers'),
schema=res_data.get('schema'))
```
numFor: 1
numIf: 1
numIfElse: 0
multiple function calls

Idiomatic code:
```
filter = {int(status): dict(headers=res_data.get('headers'),
schema=res_data.get('schema')) for (status, res_data) in
six.iteritems(responses) if isinstance(status, int) or status.isdigit()}
```

Dict-Comprehension

Non-idiomatic code:
```
conjugates = set()
for el in group.generate_dimino():
    for gen in subgr_gens:
        conjugates.add(gen ^ el)
```
numFor: 2
numIf: 0
numIfElse: 0
1. group.generate_dimino() is a generator function which yield will prevent the function from exiting, until the next time next() is called
2. Non-Python built-in objects overload the binary **expression ^**

Idiomatic code:
```
conjugates = {gen ^ el for el in group.generate_dimino() for gen in
subgr_gens}
```

Non-idiomatic code:
```
remove = set()
for (i, p) in enumerate(N):
    for j in range(i + 1, len(N)):
        if sdm_monomial_divides(p[3], N[j][3]):
            remove.add(j)
```
numFor: 2
numIf: 1
numIfElse: 0
Involving **function calls**

Idiomatic code:
```
remove = {j for (i, p) in enumerate(N) for j in range(i + 1, len(N)) if
sdm_monomial_divides(p[3], N[j][3])}
```

Non-idiomatic code:
```
for x in expr.free_symbols:
    if isinstance(x, CoordinateSym) and
x.frame != frame:
        frame_set.add(x.frame)
```
numFor: 1
numIf: 1
numIfElse: 0
Involving **function calls and attribute access**

Idiomatic code:
```
frame_set = {x.frame for x in expr.free_symbols if
isinstance(x, CoordinateSym) and x.frame !=
frame}
```

Non-idiomatic code:
```
t = set()
for n in c:
    if n in required and all((a in s for (a, b) in
connections if b == n)):
        t.add(n)
```
numFor: 1
numIf: 1
numIfElse: 0
Involving **function calls and nest a generation expression which contain for and if nodes**

Idiomatic code:
```
t = {n for n in c if n in required and all((a in s for (a,
b) in connections if b == n))}
```

Set-Comprehension

Non-idiomatic code:
```
self.n_jobs is not None and
self.n_jobs > 1
```
numComop: 2
comop: {is_not, >}
the chained cimparator is an attribute access

Idiomatic code:
```
None is not self.n_jobs > 1
```

Non-idiomatic code:
```
nx >= 0 and ny >= 0 and (nx <
size) and (ny < size)
```
numComop: 4
comop: {>=, >=, <, <}

Idiomatic code:
```
ny >= 0 <= nx < size > ny
```

Non-idiomatic code:
```
status != StatusType.CAPPED and
self.data[k].status == StatusType.CAPPED
```
numComop: 2
comop: {!=, ==}
the chained cimparator is an attribute access and the another comparator is cascading attribute accesses

Idiomatic code:
```
status != StatusType.CAPPED ==
self.data[k].status
```

Non-idiomatic code:
```
 20 != len(decoded) and
len(decoded) != 32
```
numComop: 2
comop: {==, ==}
chained cmparator is a function call

Idiomatic code:
```
20 != len(decoded) != 32
```

Non-idiomatic code:
```
self.id_ is not None and self.name is not None
```
numComop: 2
comop: {==, ==}
non-chained mparators are attribute accesses

Idiomatic code:
```
self.id_ is not None is not self.name
```

Non-idiomatic code:
```
len(ideas_idx_per_cluster[c1]) > 0 and
len(ideas_idx_per_cluster[c2]) > 0
```
numComop: 2
comop: {==, ==}
non-chained cmparators are function calls

Idiomatic code:
```
len(ideas_idx_per_cluster[c1]) > 0 <
len(ideas_idx_per_cluster[c2])
```

Chain-Comparison

Fig. 5: Code Examples for List/Set/Dict-Comprehension and Chain-Comparison int the Real-Project DataSet

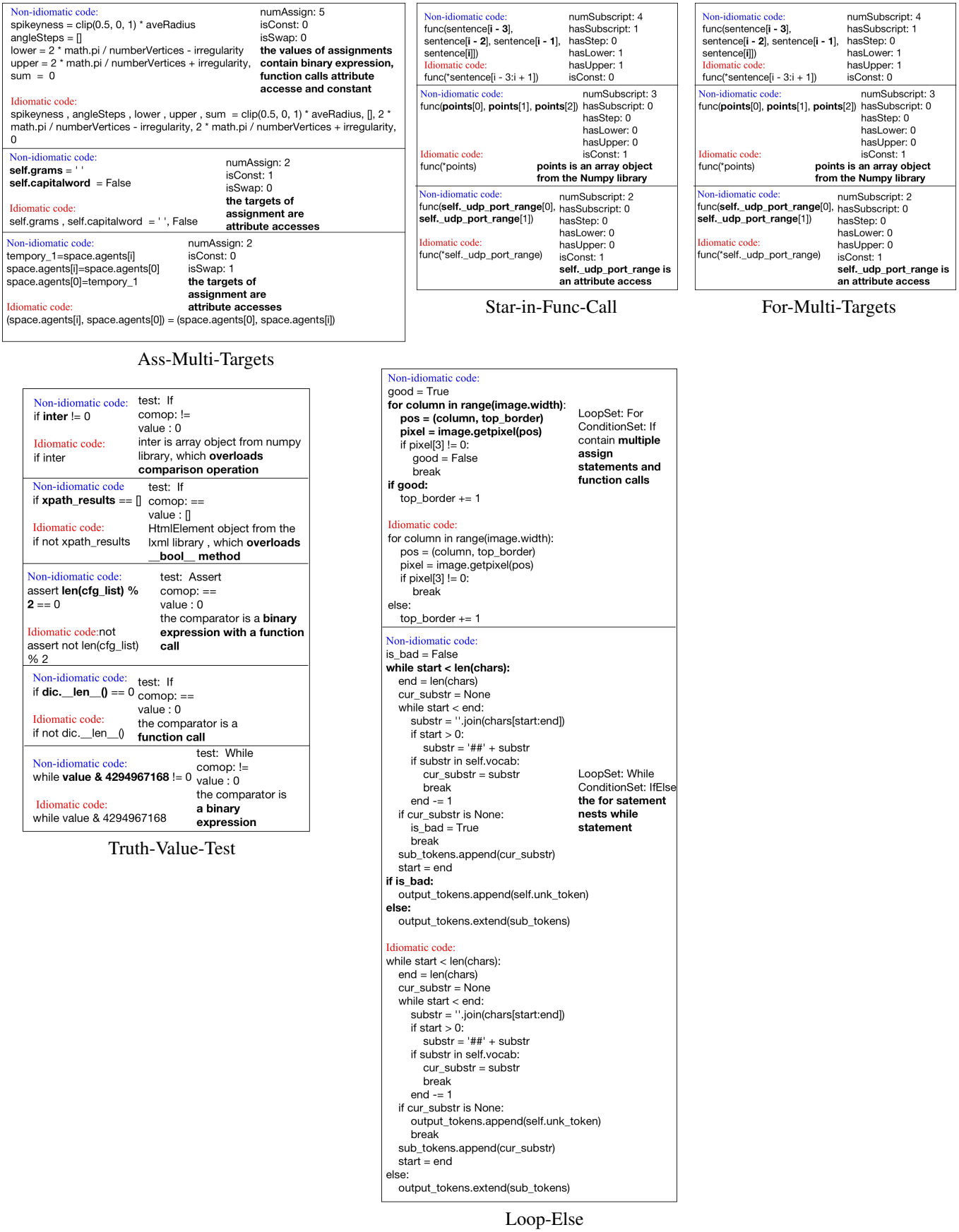

Fig. 6: Code Examples for Assign-Multi-Targets, Star-in-Func-Call, For-Multi-Targets, Truth-Value-Test and Loop-Else int the Real-Project DataSet