
perflint

downloads 9.2k/month downloads 9.2k/month

A Linter for performance anti-patterns

This project is an early beta. It will likely raise many false-positives in your code.

Installation

```
1 pip install perflint
```

Usage

Perflint can be used as a standalone linter:

```
1 perflint your_code/
```

Or as a `pylint` linter plugin:

```
1 pylint your_code/ --load-plugins=perflint
```

VS Code

Add these configuration properties to your `.vscode/settings.json` file (create if it doesn't exist):

```
1 {
2     "python.linting.pylintEnabled": true,
3     "python.linting.enabled": true,
4     "python.linting.pylintArgs": [
5         "--load-plugins",
6         "perflint",
7         "--rcfile",
8         "${workspaceFolder}/.pylintrc"
9     ],
10 }
```

Rules

W8101: Unnecessary list() on already iterable type (unnecessary-list-cast)

Using a `list()` call to eagerly iterate over an already iterable type is inefficient as a second list iterator is created, after first iterating the value:

```
1 def simple_static_tuple():
2     """Test warning for casting a tuple to a list."""
3     items = (1, 2, 3)
4     for i in list(items): # [unnecessary-list-cast]
5         print(i)
```

W8102: Incorrect iterator method for dictionary (incorrect-dictionary-iterator)

Python dictionaries store keys and values in two separate tables. They can be individually iterated. Using `.items()` and discarding either the key or the value using `_` is inefficient, when `.keys()` or `.values()` can be used instead:

```
1 def simple_dict_keys():
2     """Check that dictionary .items() is being used correctly. """
3     fruit = {
4         'a': 'Apple',
5         'b': 'Banana',
6     }
7
8     for _, value in fruit.items(): # [incorrect-dictionary-iterator]
9         print(value)
10
11    for key, _ in fruit.items(): # [incorrect-dictionary-iterator]
12        print(key)
```

W8201: Loop invariant statement (loop-invariant-statement)

The body of loops will be inspected to determine statements, or expressions where the result is constant (invariant) for each iteration of a loop. This is based on named variables which are not modified during each iteration.

For example:

```
1 def loop_invariant_statement():
2     """Catch basic loop-invariant function call."""
3     x = (1,2,3,4)
4
5     for i in range(10_000):
```

```

6         # x is never changed in this loop scope,
7         # so this expression should be evaluated outside
8         print(len(x) * i) # [loop-invariant-statement]
9         #         ^^^^^^^

```

`len(x)` should be evaluated outside the loop since `x` is not modified within the loop.

```

1 def loop_invariant_statement():
2     """Catch basic loop-invariant function call."""
3     x = (1,2,3,4)
4     n = len(x)
5     for i in range(10_000):
6         print(n * i) # [loop-invariant-statement]

```

The loop-invariance checker will underline expressions and sub-expressions within the body using the same rules:

```

1 def loop_invariant_statement_more_complex():
2     """Catch basic loop-invariant function call."""
3     x = [1,2,3,4]
4     i = 6
5
6     for j in range(10_000):
7         # x is never changed in this loop scope,
8         # so this expression should be evaluated outside
9         print(len(x) * i + j)
10    #         ^^^^^^^^^^^^^ [loop-invariant-statement]

```

Methods are blindly considered side-effects, so if a method is called on a variable, it is assumed to have possibly changed in value and therefore not loop-invariant:

```

1 def loop_invariant_statement_method_side_effect():
2     """Catch basic loop-invariant function call."""
3     x = [1,2,3,4]
4     i = 6
5
6     for j in range(10_000):
7         print(len(x) * i + j)
8         x.clear() # x changes as a side-effect

```

The loop-invariant analysis will walk up the AST until it gets to the whole loop body, so an entire branch could be marked. For example, the expression `len(x) > 2` is invariant and therefore should be outside the loop. Also, because `x * i` is invariant, that statement should also be outside the loop, therefore the entire branch will be marked:

```

1 def loop_invariant_branching():
2     """Ensure node is walked up to find a loop-invariant branch"""
3     x = [1,2,3,4]
4     i = 6

```

```
5
6     for j in range(10_000):
7         # Marks entire branch
8         if len(x) > 2:
9             print(x * i)
```

Notes on loop invariance Functions can have side-effects (print is a good example), so the loop-invariant scanner may give some false-positives.

It will also highlight dotted expressions, e.g. attribute lookups. This may seem noisy, but in some cases this is valid, e.g.

```
1 from os.path import exists
2 import os
3
4 def dotted_import():
5     for _ in range(100_000):
6         return os.path.exists('/')
7
8 def direct_import():
9     for _ in range(100_000):
10        return exists('/')
```

`direct_import()` is 10-15% faster than `dotted_import()` because it doesn't need to load the `os` global, the `path` attribute and the `exists` method for each iteration.

W8202: Global name usage in a loop (loop-global-usage)

Loading globals is slower than loading "fast" local variables. The difference is marginal, but when propagated in a loop, there can be a noticeable speed improvement, e.g.:

```
1 d = {
2     "x": 1234,
3     "y": 5678,
4 }
5
6 def dont_copy_dict_key_to_fast():
7     for _ in range(100000):
8         d["x"] + d["y"]
9         d["x"] + d["y"]
10        d["x"] + d["y"]
11        d["x"] + d["y"]
12        d["x"] + d["y"]
13
14 def copy_dict_key_to_fast():
15     i = d["x"]
```

```
16     j = d["y"]
17
18     for _ in range(100000):
19         i + j
20         i + j
21         i + j
22         i + j
23         i + j
```

`copy_dict_key_to_fast()` executes 65% faster than `dont_copy_dict_key_to_fast()`

R8203 : Try..except blocks have a significant overhead. Avoid using them inside a loop (loop-try-except-usage).

Up to Python 3.10, `try...except` blocks are computationally expensive compared with `if` statements.

Avoid using them in a loop as they can cause significant overheads. Refactor your code to not require iteration specific details and put the entire loop in the body of a `try` block.

W8204 : Looped slicing of bytes objects is inefficient. Use a `memoryview()` instead (memoryview-over-bytes)

Slicing of `bytes` is slow as it creates a copy of the data within the requested window. Python has a builtin type, `memoryview` for zero-copy interactions:

```
1 def bytes_slice():
2     """Slice using normal bytes"""
3     word = b'A' * 1000
4     for i in range(1000):
5         n = word[0:i]
6         # ^^^^^^^^^ memoryview-over-bytes
7
8 def memoryview_slice():
9     """Convert to a memoryview first."""
10    word = memoryview(b'A' * 1000)
11    for i in range(1000):
12        n = word[0:i]
```

`memoryview_slice()` is 30-40% faster than `bytes_slice()`

W8205 : Importing the “%s” name directly is more efficient in this loop. (dotted-import-in-loop)

In Python you can import a module and then access submodules as attributes. You can also access functions as attributes of that module. This keeps your import statements minimal, however, if you use this method in a loop it is inefficient because each loop iteration it will load global, load attribute and then load method. Because the name isn't an object, “load method” falls back to load attribute via a slow internal path.

Importing the desired function directly is 10-15% faster:

```
1 import os # NOQA
2
3 def test_dotted_import(items):
4     for item in items:
5         val = os.environ[item] # Use `from os import environ`
6
7 def even_worse_dotted_import(items):
8     for item in items:
9         val = os.path.exists(item) # Use `from os.path import exists`
            instead
```

W8301 : Use tuple instead of list for a non-mutated sequence. (use-tuple-over-list)

Constructing a tuple is faster than a list and indexing tuples is faster. When the sequence is not mutated, then a tuple should be used instead:

```
1 def index_mutated_list():
2     fruit = ["banana", "pear", "orange"]
3     fruit[2] = "mandarin"
4     len(fruit)
5     for i in fruit:
6         print(i)
7
8 def index_non_mutated_list():
9     fruit = ["banana", "pear", "orange"] # Raises [use-tuple-over-list]
10    ]
11    print(fruit[2])
12    len(fruit)
13    for i in fruit:
14        print(i)
```

Mutation is determined by subscript assignment, slice assignment, or methods called on the list.

W8401 : Use a list comprehension instead of a for-loop (use-list-comprehension)

List comprehensions are 25% more efficient at creating new lists, with or without an if-statement:

```
1 def should_be_a_list_comprehension_filtered():
2     """A List comprehension would be more efficient."""
3     original = range(10_000)
4     filtered = []
5     for i in original:
6         if i % 2:
7             filtered.append(i)
```

W8402 : Use a list copy instead of a for-loop (use-list-copy)

Use either the `list()` constructor or `list.copy()` to copy a list, not another for loop:

```
1 def should_be_a_list_copy():
2     """Using the copy() method would be more efficient."""
3     original = range(10_000)
4     filtered = []
5     for i in original:
6         filtered.append(i)
```

W8403 : Use a dictionary comprehension instead of a for-loop (use-dict-comprehension)

Dictionary comprehensions should be used in simple loops to construct dictionaries:

```
1 def should_be_a_dict_comprehension():
2     pairs = (("a", 1), ("b", 2))
3     result = {}
4     for x, y in pairs:
5         result[x] = y
6
7 def should_be_a_dict_comprehension_filtered():
8     pairs = (("a", 1), ("b", 2))
9     result = {}
10    for x, y in pairs:
11        if y % 2:
12            result[x] = y
```