

- Secrets
- ABAP
- Apex
- C
- C++
- CloudFormation
- COBOL
- C#
- CSS
- Flex
- Go
- HTML
- Java
- JavaScript
- Kotlin
- Objective C
- PHP
- PL/I
- PL/SQL
- Python**
- RPG
- Ruby
- Scala
- Swift
- Terraform
- Text
- TypeScript
- T-SQL
- VB.NET
- VB6
- XML



# Python static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your PYTHON code

All rules 216

Vulnerability 29

Bug 55

Security Hotspot 31

Code Smell 101

Tags ▾

Search by name...

Functions should not have too many lines of code

Code Smell

Track uses of "NOSONAR" comments

Code Smell

Track comments matching a regular expression

Code Smell

Statements should be on separate lines

Code Smell

Functions should not contain too many return statements

Code Smell

Files should not have too many lines of code

Code Smell

Lines should not be too long

Code Smell

Methods and properties that don't access instance data should be static

Code Smell

New-style classes should be used

Code Smell

Parentheses should not be used after certain keywords

Code Smell

Track "TODO" and "FIXME" comments that do not contain a reference to a person

Code Smell

Module names should comply with a naming convention

Code Smell

## Functions and lambdas should not reference variables defined in enclosing loops

Analyze your code

Code Smell Major suspicious

Nested functions and lambdas can reference variables defined in enclosing scopes. This can create tricky bugs when the variable and the function are defined in a loop. If the function is called in another iteration or after the loop finishes, it will see the variables' last value instead of seeing the values corresponding to the iteration where the function was defined.

Capturing loop variables might work for some time but:

- it makes the code difficult to understand.
- it increases the risk of introducing a bug when the code is refactored or when dependencies are updated. See an example with the builtin "map" below.

One solution is to add a parameter to the function/lambda and use the previously captured variable as its default value. Default values are only executed once, when the function is defined, which means that the parameter's value will remain the same even when the variable is reassigned in following iterations.

Another solution is to pass the variable as an argument to the function/lambda when it is called.

This rule raises an issue when a function or lambda references a variable defined in an enclosing loop.

### Noncompliant Code Example


```
def run():
    mylist = []
    for i in range(5):
        mylist.append(lambda: i) # Noncompliant

    def func():
        return i # Noncompliant
    mylist.append(func)


def example_of_api_change():
    """
    Passing loop variable as default values also makes sure
    For example the following code will work as intended with
    Why? because "map" behavior changed. It now returns an iterator
    the lambda when required. The same is true for other functions
    """
    lst = []
    for i in range(5):
        lst.append(map(lambda x: x + i, range(3))) # Noncompliant
    for sublist in lst:
        # prints [4, 5, 6] x 4 with python 3, with python 2
        print(list(sublist))
```

### Compliant Solution


Comments should not be located at the end of lines of code

 Code Smell

Lines should not end with trailing whitespaces

 Code Smell

Files should contain an empty newline at the end

 Code Smell

Long suffix "L" should be upper case

 Code Smell

```
def run():
    mylist = []
    for i in range(5):
        mylist.append(lambda i=i: i) # passing the variable

        def func(i=i): # same for nested functions
            return i
        mylist.append(func)

def example_of_api_change():
    """
    This will work for both python 2 and python 3.
    """
    lst = []
    for i in range(5):
        lst.append(map(lambda x, value=i: x + value, range(3)))
    for sublist in lst:
        print(list(sublist))
```

### Exceptions

No issue will be raised if the function or lambda is directly called in the same loop. This still makes the design difficult to understand but it is less error prone.

```
def function_called_in_loop():
    for i in range(10):
        print((lambda param: param * i)(42)) # Calling the lambda directly

        def func(param):
            return param * i

        print(func(42)) # Calling "func" directly
```

### See

- [The Hitchhiker's Guide to Python - Common Gotchas](#)
- Python documentation - [Function definitions](#)

Available In:

 |  | 