

# Python Coding Standards Compliance



This document was drawn from [Python Standards Checking](#) on the Trac/Wiki, with some light edits (including some re-formatting for Confluence). The content on this page should be verified before the Trac/Wiki page is removed.

## Policy

Upon receiving a Standards Compliance report, the developer is expected to resolve the issues. Most LSST Rules may be 'broken' if the developer feels there are good reasons for not following the Standards. This caveat might be tightened up if the developers aren't primarily holding to the Standards.

## Coding Standards Compliance Tool

LogiLab **PyLint** was selected as the most comprehensive Coding Standards Checker for Python available. An important feature is its design strategy allowing user-coded add-ons to the Rule base.

## Documentation

- PyLint User Manual: [http://www.logilab.org/card/pylint\\_manual](http://www.logilab.org/card/pylint_manual)
- PyLint error messages sorted by;
  - [Message text](#)
  - [Error code](#)

## Rules

Currently the only changes to the LogiLab **PyLint** Rule base is the disabling of some Rules to better reflect the [LSST Python Coding Standards](#). (**Note:** the `pylintrc` startup file containing the list of disabled errors is missing.)

## Suppressing an Infraction

**PyLint** provides a variety of methods to either permanently or temporarily turn-off an infraction report covering a single line or a block of lines. The option to permanently disable all infractions against specific Rules are also provided in the [.pylintrc](#) initialization script and has been used to refine the LSST Infraction Report to those Rules of interest.

Provided below is the **PyLint** example/tester on suppressing an infraction report within the user's code. If you need to inhibit an infraction on a single line, review **meth3** shown below.

```
"""pylint option block-disable-msg"""

class Foo(object):
    """block-disable-msg test"""

    def __init__(self):
        pass

    def meth1(self, arg):
        """this issues a message"""
        print self

    def meth2(self, arg):
        """and this one not"""
        # pylint: disable-msg=W0613
        print self\
            + "foo"

    def meth3(self):
        """test one line disabling"""
        # no error
        print self.bla # pylint: disable-msg=E1101
        # error
        print self.blop

    def meth4(self):
        """test re-enabling"""
        # pylint: disable-msg=E1101
        # no error
        print self.bla
        print self.blop
```

```

# pylint: enable-msg=E1101
# error
print self.blip

def meth5(self):
    """test IF sub-block re-enabling"""
    # pylint: disable-msg=E1101
    # no error
    print self.bla
    if self.blop:
        # pylint: enable-msg=E1101
        # error
        print self.blip
    else:
        # no error
        print self.blip
    # no error
    print self.blip

def meth6(self):
    """test TRY/EXCEPT sub-block re-enabling"""
    # pylint: disable-msg=E1101
    # no error
    print self.bla
    try:
        # pylint: enable-msg=E1101
        # error
        print self.blip
    except UndefinedName: # pylint: disable-msg=E0602
        # no error
        print self.blip
    # no error
    print self.blip

def meth7(self):
    """test one line block opening disabling"""
    if self.blop: # pylint: disable-msg=E1101
        # error
        print self.blip
    else:
        # error
        print self.blip
    # error
    print self.blip

def meth8(self):
    """test late disabling"""
    # error
    print self.blip
    # pylint: disable-msg=E1101
    # no error
    print self.bla
    print self.blop

```

## Suppressing Multiple Infractions

Multiple infractions on a single line are suppressed using the syntax:

```
<statement> # pylint: disable-msg=<error#>[,<error#>]*
```

For example:

```
from sdqaLib import * # pylint: disable-msg=W0401,W0403
```

## Common or Curious Error Reports

## "Statement seems to have no effect" Error

Python does not require a ';' at the end of most statements but it doesn't object should you gratuitously add one. On the other hand, Pylint reports such an occurrence is an error:

```
self.policy.set("singleKernelClipping", False);
```

produced the following errors - as a result of the extra semi-colon:

```
examples/JackknifeResampleSpatialKernel.py:136: [W0104,
DiffimTestCases.jackknifeResample] Statement seems to have no effect
examples/JackknifeResampleSpatialKernel.py:136: [C0321,
DiffimTestCases.jackknifeResample] More than one statement on a single line
```

## Relative Import Error

PyLint reported:

```
python/lsst/ip/diffim/createPsfMatchingKernel.py:5: [W0403] Relative import 'createKernelFunctor'
```

The python source:

```
# all the c++ level classes and routines...
import diffimLib

# all the other diffim routines
from createKernelFunctor import createKernelFunctor
```

The solution:

```
# all the c++ level classes and routines...
import diffimLib

# all the other diffim routines
from .createKernelFunctor import createKernelFunctor
```

Note the leading period. The warning is about the new-to-python 2.5 feature of relative imports as described in PEP 328: <http://www.python.org/dev/peps/pep-0328/> and the forthcoming change in how the ambiguity of absolute imports will change. The old import scheme that most of us have used for years has a serious flaw. Suppose you have the following file hierarchy:

```
mypkg/
  __init__.py
  sub1/
    __init__.py
    foo.py
    bar.py
  sub2/
    __init__.py
    why.py
```

and `bar.py` starts with:

```
import foo
```

This is fine unless you install a new package named `"foo"` and you want to access it from `bar.py`. Then it's a headache because the local `foo` shields the main-level package `foo` FOR NOW. In a few generations of Python the main-level `foo` will shadow the local `foo` instead!!! Thus the warning.

You **should** use the new relative import syntax:

```
from . import foo
```

You can use more dots as needed:

```
from ..sub2 import why
from ...mypkg.sub2 import why # stupid, but shows dots to go back up again
```

You can even get the new absolute import syntax (preferring global to local module names) but you have to use `"from future import absolute_import"`. This may only be necessary if you really do want to access a global module whose name is shadowed by a local module. I suspect any code that does not will start raising deprecation warnings when we switch to Python 2.6 and in the long run such code could break.