Deep dive into Pylint

Łukasz Rogalski

Agenda

1. Static analysis
2. Pylint as static analysis tool
3. Checkers
4. Inference engine
5. Known issues
6. Alternatives
7. Summary
8. Q&A

Static analysis

Definition

> Static program analysis is the analysis of computer software that is performed without actually executing programs. In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code.
> The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension, or code review. Software inspections and software walkthroughs are also used in the latter case.
> Static program analysis - Wikipedia

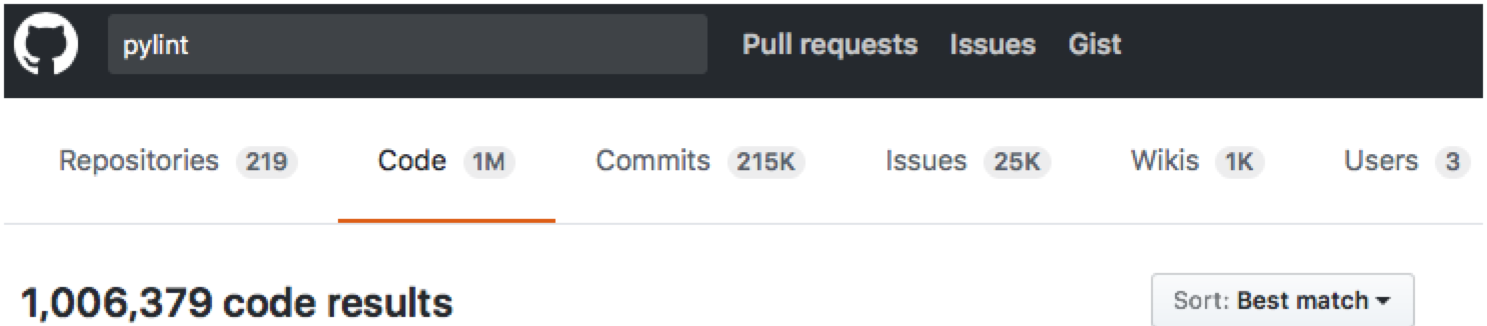
Pylint as static analysis tool

PyCQA/pylint: A Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells

- Dated back as far as 2001 (Python 2.2), originally authored by Logilab
- First commit in Git: forget the past. · PyCQA/pylint@4becf6f (committed on 26 Apr 2006)
- I wasn’t able to find old SVN repo :(

Quite popular:

[Search · pylint · GitHub]



How to install:

```
$ pip3 install pylint
```

How to run:

```
$ pylint my_package
```

Sample output:

```
No config file found, using default configuration
************* Module flow.graph
C: 64, 0: Line too long (102/100) (line-too-long)
C:164, 0: Wrong continued indentation (add 1 space).
                                    isinstance(node.statement, ImplicitReturnStmt))
                                    ^| (bad-continuation)
C:  1, 0: Missing module docstring (missing-docstring)
C:  9, 0: Missing class docstring (missing-docstring)
W: 21, 4: Useless super delegation in method '__init__' (useless-super-delegation)
R: 20, 0: Too few public methods (0/2) (too-few-public-methods)
W: 46, 0: Dangerous default value [] as argument (dangerous-default-value)
W: 46,24: Unused argument 'ast_node' (unused-argument)
R: 76, 4: Unnecessary "else" after "return" (no-else-return)
W:132, 0: Dangerous default value [] as argument (dangerous-default-value)
R:296, 4: Too many return statements (7/6) (too-many-return-statements)

------------------------------------------------------------------
Your code has been rated at 7.06/10 (previous run: 7.06/10, +0.00)
```

Checkers

Definition

Checkers implement code verification logic.

- pluggable architecture (new checker is a `BaseChecker` subclass, registered in linter before analysis start)
- exact API depends on type of checks checker performs

Two groups of checkers:

- token-based checkers
- AST-based checkers

Token-based checkers

Tokens

> A Python program is read by a parser. Input to the parser is a stream of tokens, generated by the lexical analyzer.

See lexical analysis for details.

`tokenize` module in standard library allows to generate tokens based on module source.

simple_module.py:

```python
def is_something(a: int) -> bool:
    return a >= 2
```

run_tokenize.py:

```python
import tokenize

module_name = "simple_module.py"
with open(module_name) as fh:
    for token in tokenize.generate_tokens(fh.readline):
        print(token)
```

Output

```
TokenInfo(type=1 (NAME), string='def', start=(1, 0), end=(1, 3), line='def is_somet
TokenInfo(type=1 (NAME), string='is_something', start=(1, 4), end=(1, 16), line='de
TokenInfo(type=53 (OP), string='(', start=(1, 16), end=(1, 17), line='def is_someth
TokenInfo(type=1 (NAME), string='a', start=(1, 17), end=(1, 18), line='def is_somet
TokenInfo(type=53 (OP), string=':', start=(1, 18), end=(1, 19), line='def is_someth
TokenInfo(type=1 (NAME), string='int', start=(1, 20), end=(1, 23), line='def is_som
TokenInfo(type=53 (OP), string=')', start=(1, 23), end=(1, 24), line='def is_someth
TokenInfo(type=53 (OP), string='->', start=(1, 25), end=(1, 27), line='def is_somet
TokenInfo(type=1 (NAME), string='bool', start=(1, 28), end=(1, 32), line='def is_so
TokenInfo(type=53 (OP), string=':', start=(1, 32), end=(1, 33), line='def is_someth
TokenInfo(type=4 (NEWLINE), string='\n', start=(1, 33), end=(1, 34), line='def is_s
TokenInfo(type=5 (INDENT), string='    ', start=(2, 0), end=(2, 4), line='    retur
TokenInfo(type=1 (NAME), string='return', start=(2, 4), end=(2, 10), line='    retu
TokenInfo(type=1 (NAME), string='a', start=(2, 11), end=(2, 12), line='    return a
TokenInfo(type=53 (OP), string='>=', start=(2, 13), end=(2, 15), line='    return a
TokenInfo(type=2 (NUMBER), string='2', start=(2, 16), end=(2, 17), line='    return
TokenInfo(type=4 (NEWLINE), string='\n', start=(2, 17), end=(2, 18), line='    retu
TokenInfo(type=6 (DEDENT), string='', start=(3, 0), end=(3, 0), line='')
TokenInfo(type=0 (ENDMARKER), string='', start=(3, 0), end=(3, 0), line='')
```

Sample line from output:

```
TokenInfo(type=53 (OP), string='>=', start=(2, 13),
          end=(2, 15), line='    return a >= 2\n')
```

Each token has some some basic data associated with themselves:

- token type
- string value
- row and column indices for beginning and end of token
- actual line that token comes from

Checker API

```
from pylint.checkers import BaseChecker

class BaseTokenChecker(BaseChecker):
    """Base class for checkers that want to have access to

    def process_tokens(self, tokens):
        """Should be overridden by subclasses."""
        raise NotImplementedError()
```

Typical checks

- whitespace violations
- mixed indentation (tabs/spaces)
- etc.

Essentially, those checks rely on information which are missing from abstract syntax tree.

AST-based checkers

Abstract syntax tree

> In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax.
> Abstract syntax tree - Wikipedia

Python ASTs

`ast` module in standard library allows to generate syntax tree based on module source.

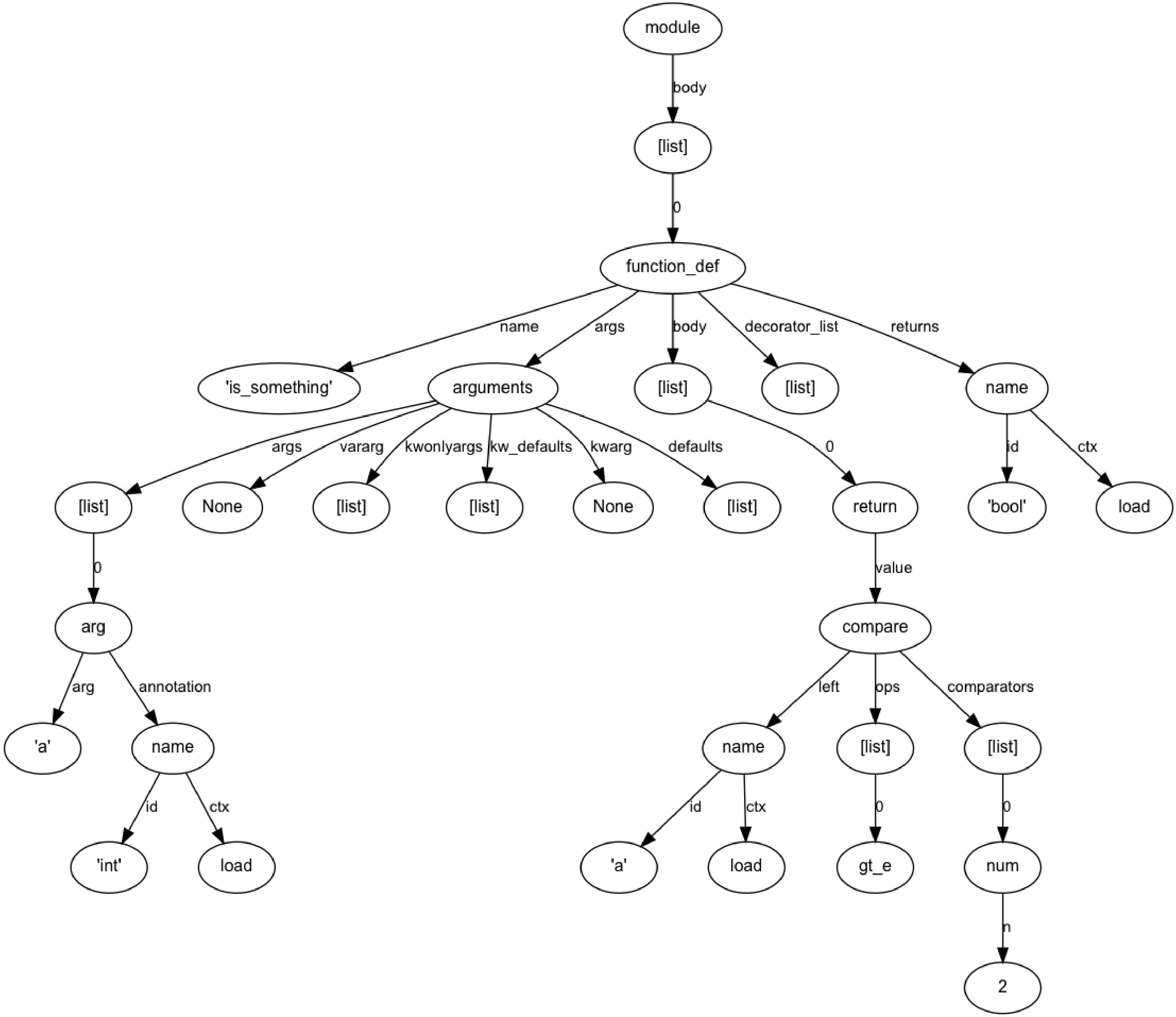
simple_module.py

```python
def is_something(a: int) -> bool:
    return a >= 2
```

run_ast.py

```python
import ast

module_name = "simple_module.py"
with open(module_name) as fh:
    ast_root = ast.parse(fh.read(), filename=module_name)
```


module
body
[list]
0
function_def
name
args
body
decorator_list
returns
'is_something'
arguments
[list]
[list]
name
args
vararg
kwonlyargs
kw_defaults
kwarg
defaults
0
id
ctx
[list]
None
[list]
[list]
None
[list]
return
'bool'
load
0
value
arg
compare
arg
annotation
left
ops
comparators
'a'
name
name
[list]
[list]
id
ctx
id
ctx
0
0
'int'
load
'a'
load
gt_e
num
n
2

AST-based checker

- is fed with AST tree
- walks over inputted tree (*visitor* pattern)
- invokes `visit_{nodeclass}` method
- those method acts accordingly based on visited nodes

```
class IterableChecker(BaseChecker):
    @check_messages('not-an-iterable')
    def visit_for(self, node):
        self._check_iterable(node.iter)

    @check_messages('not-an-iterable', 'not-a-mapping')
    def visit_call(self, node):
        for stararg in node.starargs:
            self._check_iterable(stararg.value)
        for kwarg in node.kwargs:
            self._check_mapping(kwarg.value)
```

Example

bad_super_call.py

```
class MyClass(object):
    def __init__(self):
        pass  # some implementation

class DerivedClass1(MyClass):
    def __init__(self):
        # call superclass __init__ (?)
        super(type(self), self).__init__()
        self.x = 1

class DerivedClass2(MyClass):
    def __init__(self):
        # call superclass __init__ (?)
        super(self.__class__, self).__init__()
        self.x = 2
```

- Invalid when `DerivedClass1` or `DerivedClass2` is subclassed!
- Infinite recursion on instantiation of subclass

Let's implement a checker for this case!

Algorithm:

1. When `FunctionDef` node is visited
2. And this node is a method
3. Find all `Call` nodes that belong to this method
4. If called function is a `Name` node and it's value is `"super"`
5. Check first argument of called function:
    - If first argument is `Call` node and called function is a `Name` with value `"type"` and it's single argument is first argument of method (`self`) or
    - If first argument is an `Attribute` access, where accessed member is `__class__` and object is first argument of method (`self`)
6. Emit a message

Obviously not *that* easy

1. `FunctionDef` AST node is shared between methods and functions - how to differentiate between them?
2. We don't want to walk through FunctionDef body and collect `Call` nodes blindly - sometimes they belong to another scope (e.g. another function defined in method)
3. We implicitly assumed that `super` and `type` names are bound to original built-in functions. What if they are not?

Inference engine

Problem statement

- both token-based and AST-based checkers knows only about structure of the source code (token-based: how the code was written, AST-based: how the code was parsed)
- it’s desirable to have actual knowledge about execution (*without* running the code)

Solution

- astroid (AST on steroids) - [PyCQA/astroid: A common base representation of python source code for pylint and other projects] `astroid`

> It provides a compatible representation which comes from the `_ast` module. It rebuilds the tree generated by the builtin `_ast` module by recursively walking down the AST and building an extended AST. The new node classes have additional methods and attributes for different usages. They include some support for static inference and local name scopes. Furthermore, `astroid` builds partial trees by inspecting living objects.

`astroid` basic functionality

Parent / child relationship of nodes

parent_children_demo.py

```
SOME_GLOBAL = 1  #@


def func(x):  #@
    y = x ** 2
    return y  #@
```

run_parent_children.py

```
import astroid

with open('parent_children_demo.py') as fh:
    nodes = astroid.extract_node(fh.read())
    assign_node, function_node, return_node = nodes

print(assign_node)
# Assign(targets=[<AssignName.SOME_GLOBAL l.1 at 0x0>],
#        value=<Const.int l.1 at 0x0>)
print(assign_node.parent)
# Module(name='',
#        doc=None,
#        file='<?>',
#        path='<?>',
#        package=False,
#        pure_python=True,
#        future_imports=set(),
#        body=[<Assign l.1 at 0x0>, <FunctionDef.func l.4
```

```
print(function_node)
# FunctionDef.func(name='func',
#                  doc=None,
#                  decorators=None,
#                  args=<Arguments l.4 at 0x0>,
#                  returns=None,
#                  body=[<Assign l.5 at 0x0>, <Return l.6
children = list(function_node.get_children())
print(children)
# [<Arguments l.4 at 0x0>, <Assign l.5 at 0x0>, <Return l
print(return_node in children)
# True
```

Naming scopes

run_naming_scopes.py (1)

```python
import astroid

with open('simple_module.py') as fh:
    module_node = astroid.parse(fh.read())

function_node = module_node.body[0]
function_node.lookup('a')
# (<FunctionDef.is_something l.1 at 0x106c70400>,
#  [<AssignName.a l.1 at 0x106c5a128>])
module_node.lookup('is_something')
# (<Module l.0 at 0x106c70390>,
#  [<FunctionDef.is_something l.1 at 0x106c70400>])
module_node.lookup('bool')
# (<Module.builtins l.0 at 0x1063abd68>,
#  [<ClassDef.bool l.0 at 0x1068795c0>])
```

Understanding common language constructs

Method resolution order (MRO)

mro_demo.py

```python
class A: pass
class B: pass
class C(A, B): pass
class D(B): pass
class E(D, C): pass
print(E.mro())
```

Output

```
[<class '__main__.E'>, <class '__main__.D'>,
 <class '__main__.C'>, <class '__main__.A'>,
 <class '__main__.B'>, <class 'object'>]
```

run_mro_demo.py

```
import astroid

with open('mro_demo.py') as fh:
    my_module = astroid.parse(fh.read())

e_class = my_module.locals['E'][0]
print(e_class.mro())
```

Output

```
[<ClassDef.E l.6 at 0x0>, <ClassDef.D l.5 at 0x0>,
 <ClassDef.C l.4 at 0x0>, <ClassDef.A l.2 at 0x0>,
 <ClassDef.B l.3 at 0x0>, <ClassDef.object l.0 at 0x0>]
```

Dunder methods

dunders_demo.py

```python
class A:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        return A(self.value + other.value)

a1 = A(3)
a2 = A(5)
a3 = a1 + a2
assert a3.value == 8

class CtxMgr:
    def __enter__(self):
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        pass

with CtxMgr() as obj:
    pass
```

run_dunders_demo.py

```
import astroid

with open(dunders_demo.py) as fh:
    my_module = astroid.parse(fh.read(),
                              module_name='dunders')

a3_node = my_module.locals['a3'][0]
inferred_node = next(a3_node.infer())
print('a3', inferred_node)

obj_node = my_module.locals['obj'][0]
inferred_node = next(obj_node.infer())
print('obj', inferred_node)
```

Output

```
a3: Instance of dunders.A
obj: Instance of dunders.CtxMgr
```

Slicing

slicing_demo.py

```
sequence = [1, 1, 2, 5, 8, 13]
sequence2 = sequence[::2]
sequence  #@
sequence2  #@
```

run_slicing_demo.py

```
import astroid

with open(slicing_demo.py) as fh:
    seq_node, seq2_node = astroid.extract_node(fh.read())

inferred_seq1 = next(seq_node.infer())
inferred_seq2 = next(seq2_node.infer())
print(inferred_seq1)
print([v.value for v in inferred_seq1.elts])
print(inferred_seq2)
print([v.value for v in inferred_seq2.elts])
```

Output

```
List.list(ctx=<Context.Load: 1>,
          elts=[ <Const.int l.1 at 0x0>,
            <Const.int l.1 at 0x0>,
            <Const.int l.1 at 0x0>,
            <Const.int l.1 at 0x0>,
            <Const.int l.1 at 0x0>,
            <Const.int l.1 at 0x0>])
[1, 1, 2, 5, 8, 13]
List.list(ctx=None,
          elts=[ <Const.int l.1 at 0x0>,
            <Const.int l.1 at 0x0>,
            <Const.int l.1 at 0x0>])
[1, 2, 8]
```

Understanding language constructs - summary

- In-depth understanding of Python execution model
- Helps a lot with catching errors

Transforms, `astroid.brain`

- Interpreter core code is implemented in C - no source code to even start reasoning about inference
- Fancy code, even if implemented in Python, is also really tricky to infer
- We need handcrafted set of rules to improve inference engine
- Two ways of doing it:
  - custom inference tips
  - node transforms
- Easy API for registering new transforms:

```
MANAGER.register_transform(node_class,
                           transform_callable,
                           predicate)
```

Custom inference tips

- *quasi* node transform - it sets internal attribute on AST node,
- this attribute is later on used for inferring actual values

```
def inference_tip(infer_function):
    def transform(node, infer_function=infer_function):
        node._explicit_inference = infer_function
        return node
    return transform
```

- `infer_function` should be a generator of inferred values

Inference tip demo

```python
def infer_bool(node, context=None):
    """Understand bool calls."""
    if len(node.args) > 1:
        # Invalid bool call.
        raise UseInferenceDefault

    if not node.args:
        return nodes.Const(False)

    argument = node.args[0]
    try:
        inferred = next(argument.infer(context=context))
    except InferenceError:
        return util.Uninferable
    if inferred is util.Uninferable:
        return util.Uninferable

    bool_value = inferred.bool_value()
    if bool_value is util.Uninferable:
        return util.Uninferable
    return nodes.Const(bool_value)
```

Node transforms

- modification of AST, which actually modifies output tree
- handcrafted rules for "tricky" modules (`enum`, `namedtuple`, `six`, `typing` to name a few)
- example: add extra names to module scope (if namespace is populated dynamically)

Inference engine summary

Inference engine is a *gift and the curse* of Pylint.

- Powerful inference engine means we can catch more problems in code comparing to other tools
- Unfortunately, mistakes during inference causes much higher false positive ratio

Known Pylint problems

Flow analysis

inference_flow.py

```
def func(arg):
    if not arg:
        return None
    return 2 * arg

x = func(3)
x  #@
```

run_no_flow_control.py

```
import astroid

with open('inference_flow.py') as fh:
    node = astroid.extract_node(fh.read())

for value in node.infer():
    print(value)
```

Output

```
Const.NoneType(value=None)
Const.int(value=6)
```

Observations

- Even in this simple example we are still deducing that return value *may possibly* be `None`
- Important when flow control statements are used (conditionals, exceptions, `continue` / `break` on loops etc.)

Conclusions

- As of today, Pylint default config would avoid to emit warnings in case of ambiguity
- Workaround, not a solution...

Dynamic code constructs

Example: [issue28082: use IntFlag for re constants · python/cpython@deec4d0]()

Before patch

```
# flags
A = ASCII = sre_compile.SRE_FLAG_ASCII # assume ascii "loc
I = IGNORECASE = sre_compile.SRE_FLAG_IGNORECASE # ignore
L = LOCALE = sre_compile.SRE_FLAG_LOCALE # assume current
U = UNICODE = sre_compile.SRE_FLAG_UNICODE # assume unicod
M = MULTILINE = sre_compile.SRE_FLAG_MULTILINE # make anch
S = DOTALL = sre_compile.SRE_FLAG_DOTALL # make dot match
X = VERBOSE = sre_compile.SRE_FLAG_VERBOSE # ignore white

# sre extensions (experimental, don't rely on these)
T = TEMPLATE = sre_compile.SRE_FLAG_TEMPLATE # disable bac
DEBUG = sre_compile.SRE_FLAG_DEBUG # dump pattern after co
```

After patch

```
class Flag(enum.IntFlag):
    ASCII = sre_compile.SRE_FLAG_ASCII # assume ascii "loc
    IGNORECASE = sre_compile.SRE_FLAG_IGNORECASE # ignore
    LOCALE = sre_compile.SRE_FLAG_LOCALE # assume current
    UNICODE = sre_compile.SRE_FLAG_UNICODE # assume unicod
    MULTILINE = sre_compile.SRE_FLAG_MULTILINE # make anc
    DOTALL = sre_compile.SRE_FLAG_DOTALL # make dot match
    VERBOSE = sre_compile.SRE_FLAG_VERBOSE # ignore whites
    A = ASCII
    I = IGNORECASE
    L = LOCALE
    U = UNICODE
    M = MULTILINE
    S = DOTALL
    X = VERBOSE
    # sre extensions (experimental, don't rely on these)
    TEMPLATE = sre_compile.SRE_FLAG_TEMPLATE # disable bac
    T = TEMPLATE
    DEBUG = sre_compile.SRE_FLAG_DEBUG # dump pattern afte
globals().update(Flag.__members__) # (!!!)
```

Alternatives

pycodestyle (formerly known as PEP8)

> `pycodestyle` is a tool to check your Python code against some of the style conventions in PEP 8.

- No-AST policy (as lightweight as possible), checks are based on tokenization and regular expression
- Mostly limited to style-based checks

[PyCQA/pycodestyle: Simple Python style checker in one Python file]

pyflakes

> A simple program which checks Python source files for errors.
> Pyflakes analyzes programs and detects various errors. It works by parsing the source file, not importing it, so it is safe to use on modules with side effects. It’s also much faster.
> Pyflakes is also faster than Pylint or Pychecker. This is largely because Pyflakes only examines the syntax tree of each file individually. As a consequence, Pyflakes is more limited in the types of things it can check.

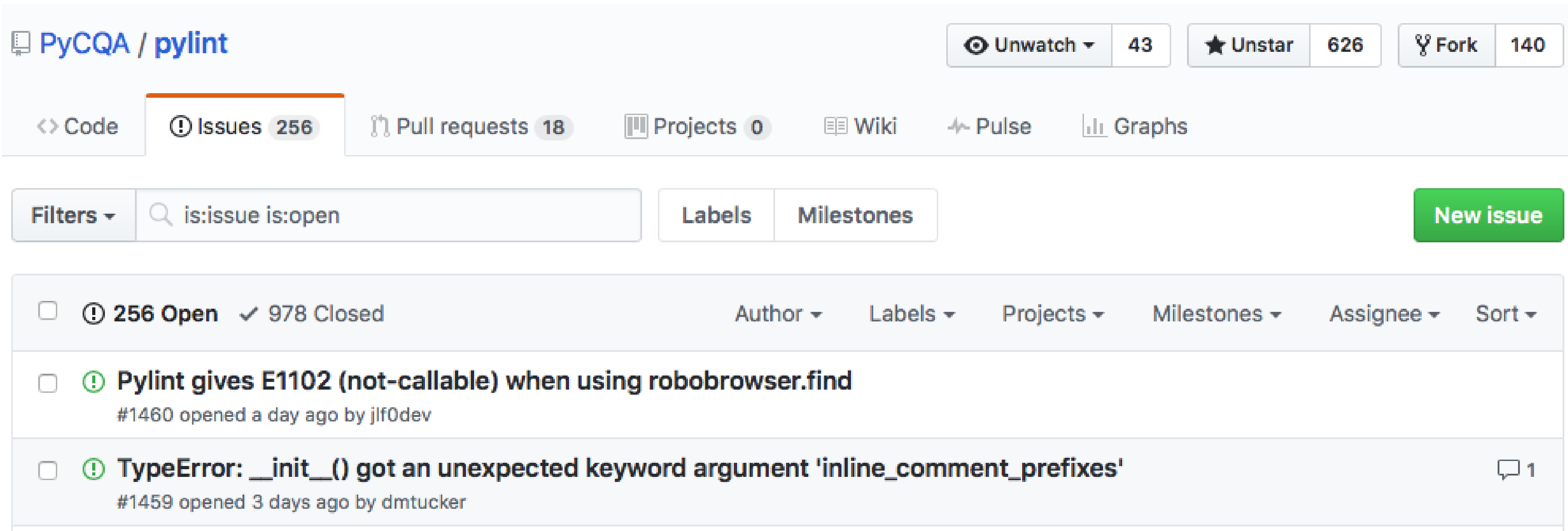
- Simpler
- With AST-based checks, without inference engine

[PyCQA/pyflakes: A simple program which checks Python source files for errors.]

Summary

- Static analysis: definition and basic principles
- Pylint checkers as a validation performing logic
- Two basic types of checkers: token-based and AST-based
- Tokens and ASTs are two different ways to look at source code
- Token-based checker: looks at code formatting
- AST-based checker: looks at code structure
- Inference engine as a crucial part of analysis
- Other analysis tools and it's comparision to Pylint

We still have a lot of issues unresolved



Pull requests are welcome!

Thanks!

- Slides: https://github.com/rogalski/pygda-pylint-talk
- My LinkedIn: https://www.linkedin.com/in/lukasz-rogalski/

Q&A