CUSTOM CODE CHECKS

Anton Marchukov

PyCon Israel 2017

ABOUT ME @martchukov

- Senior Software Engineer at Red Hat.
- oVirt Community Infra team.
- CI and related infrastructure.
- Lots of automation in Python.
- DevOps advocate.

oVirt is free, open-source virtualization management platform based on the KVM hypervisor.

WHY DO AUTOMATED CODE CHECKS?

- Let computer do the work for you.
- Let computer do the work without you.
- Reproducible set of rules.

\+ All points for doing code checks at all.

WHY WRITE CUSTOM CODE CHECKS?

- Standards do not cover local conventions.
- Standards do not cover all good coding practices .
- Like unit tests, but for the code itself.

USE CASE: ALLOW ONLY KEYWORD ARGUMENTS

~~Service.do_action(4242)~~

Service.do_action(obj_id=4242)

HOW TO FORCE? PYTHON 2

```
def do(**kwargs):
    # raises TypeError when missing.
    param = kwargs.pop('param')
```

HOW TO FORCE? PYTHON 3

```
def do(*, param):
    # Python 3 syntax (PEP 3102) to force
    # 'param' to be passed as kwarg only.
    # Raises TypeError if not.
```

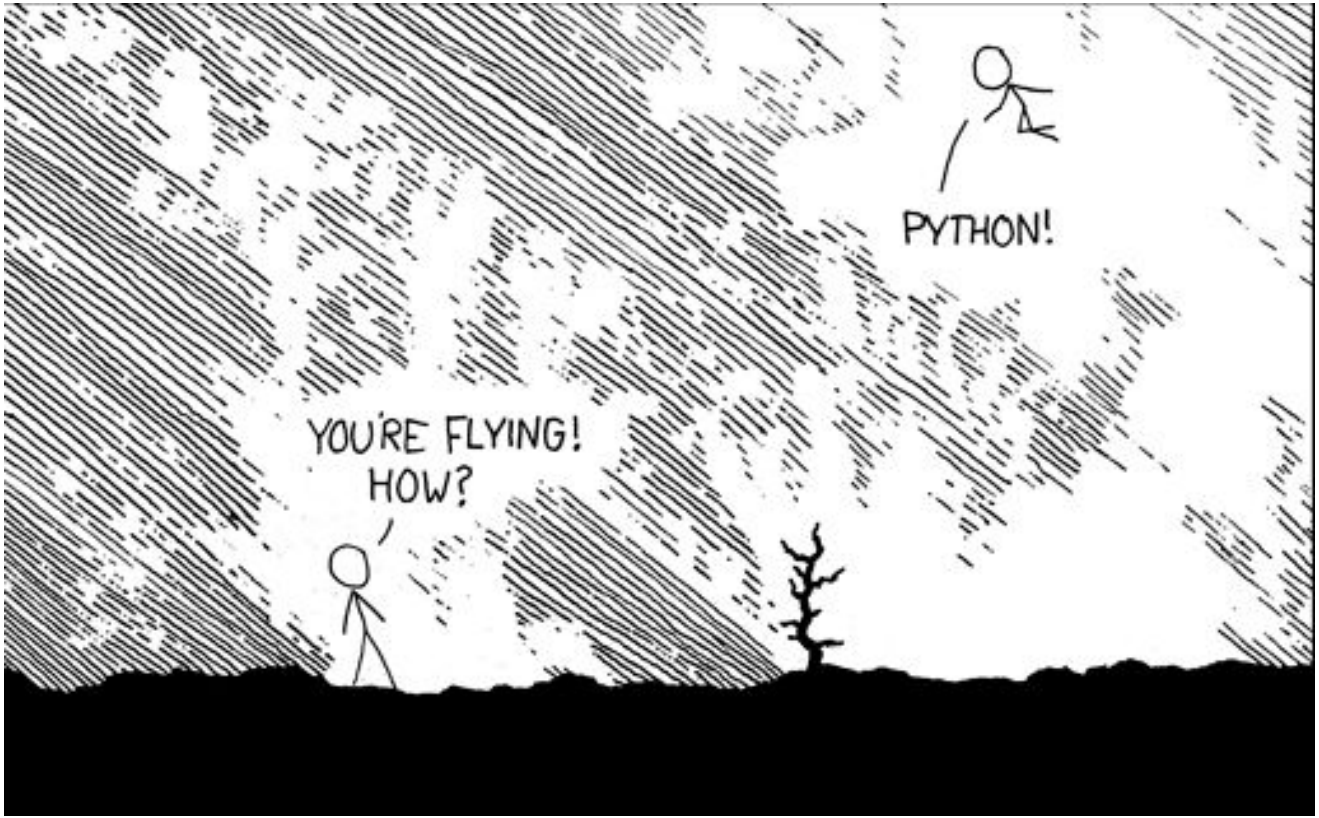
USE CASE: foobar/code_example.py

```
def do(self, *, obj_id, action):
def do_py2(self, **kwargs):
def do_py2_wrong(self, *args, **kwargs):
def do_noargs(self):
def do_wrong(self, param):
def do_one_arg_wrong(param):
```

HOW CPYTHON WORKS

1. Source Code
2. Parse Tree
3. **Abstract Syntax Tree**
4. Control Flow Graph
5. Byte Code

SCARY? BUT WE CAN FLY. PYTHON!



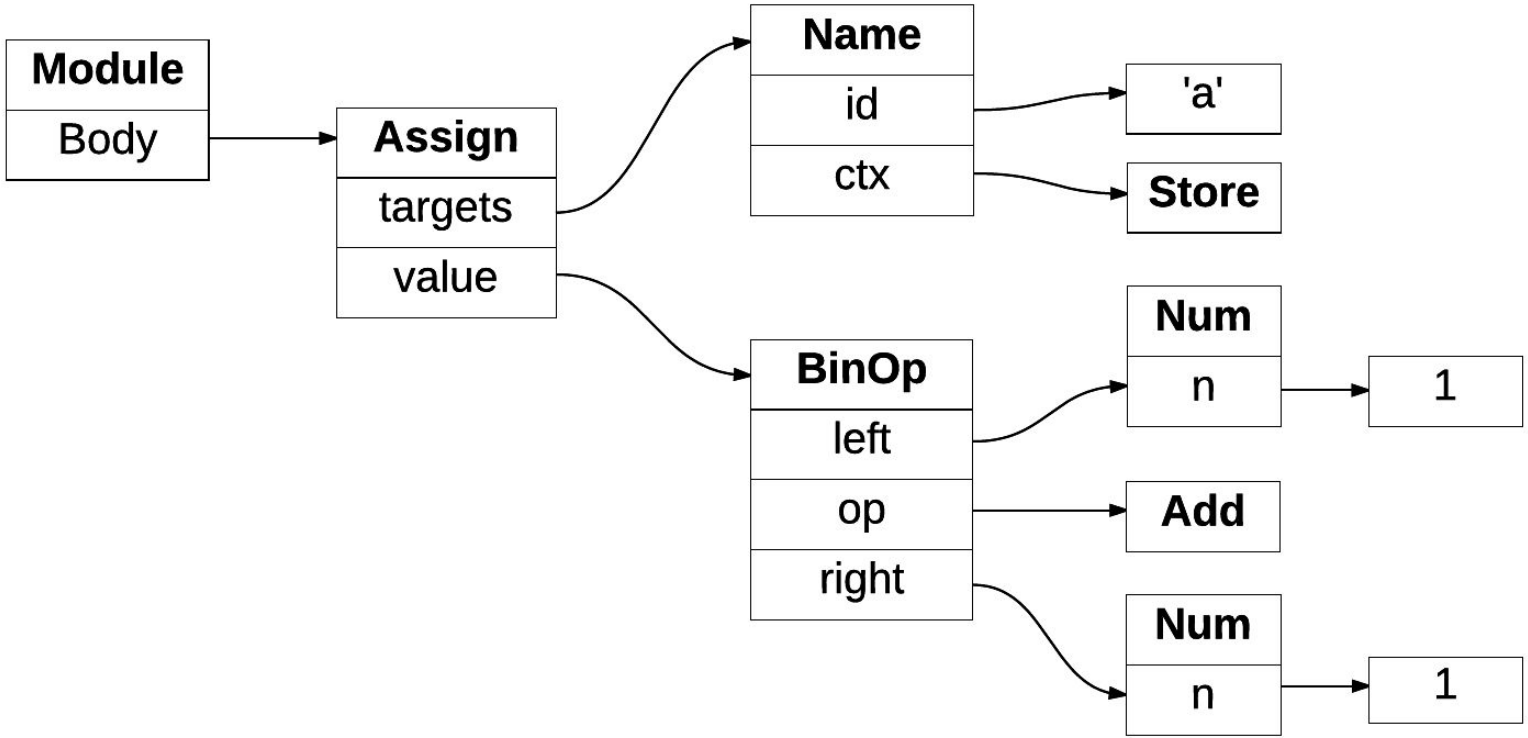
 CUSTOM CODE CHECKS. Image from https://xkcd.com/353/ under CC BY-NC.

ABSTRACT SYNTAX TREE MODULE (ast)

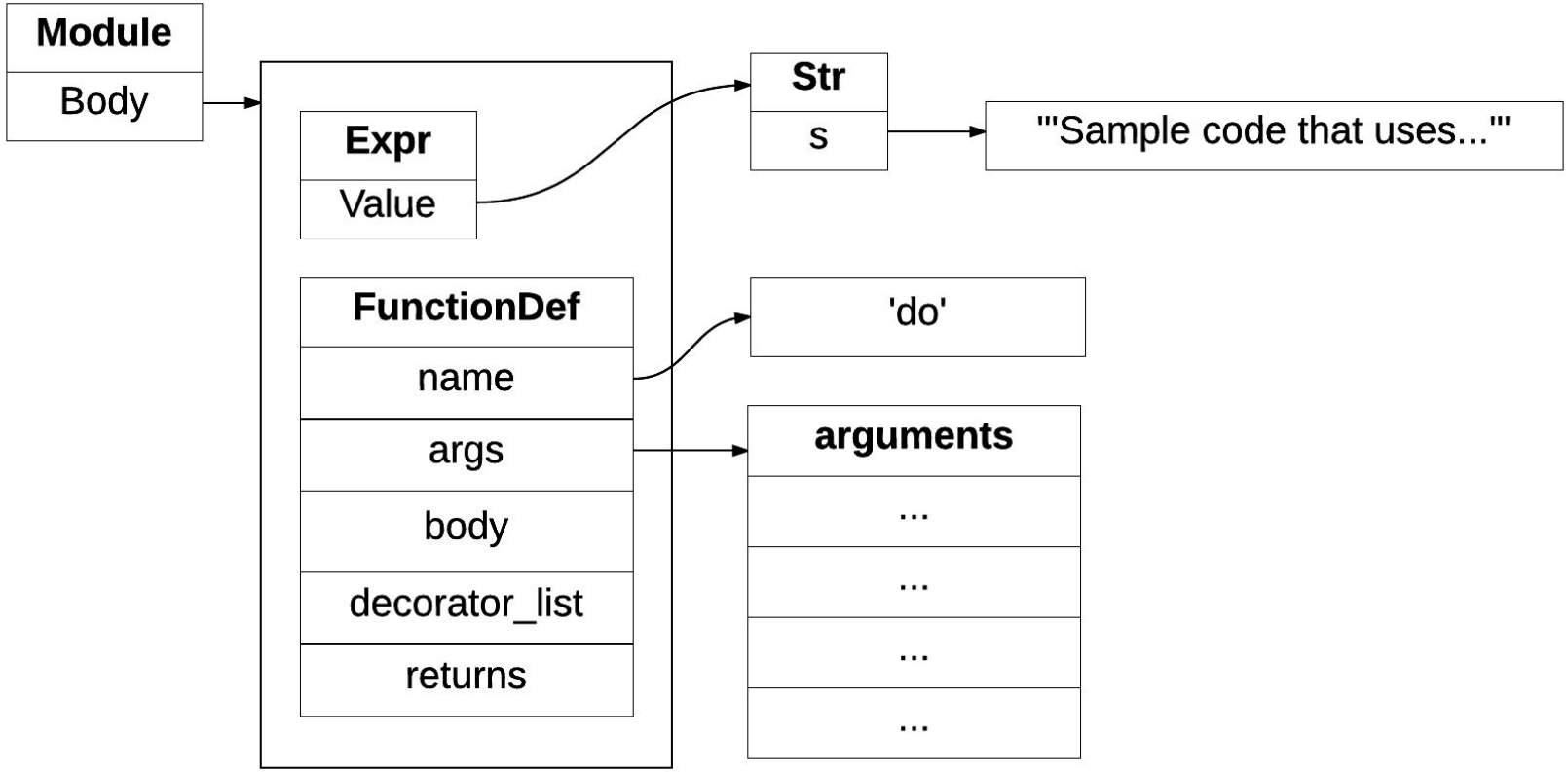
```
>>> import ast
>>> ast.dump(ast.parse("a = 1 + 1"))
```

```
"Module(body=[Assign(targets=[Name(id='a',
ctx=Store())], value=BinOp(left=Num(n=1),
op=Add(), right=Num(n=1)))])"
```

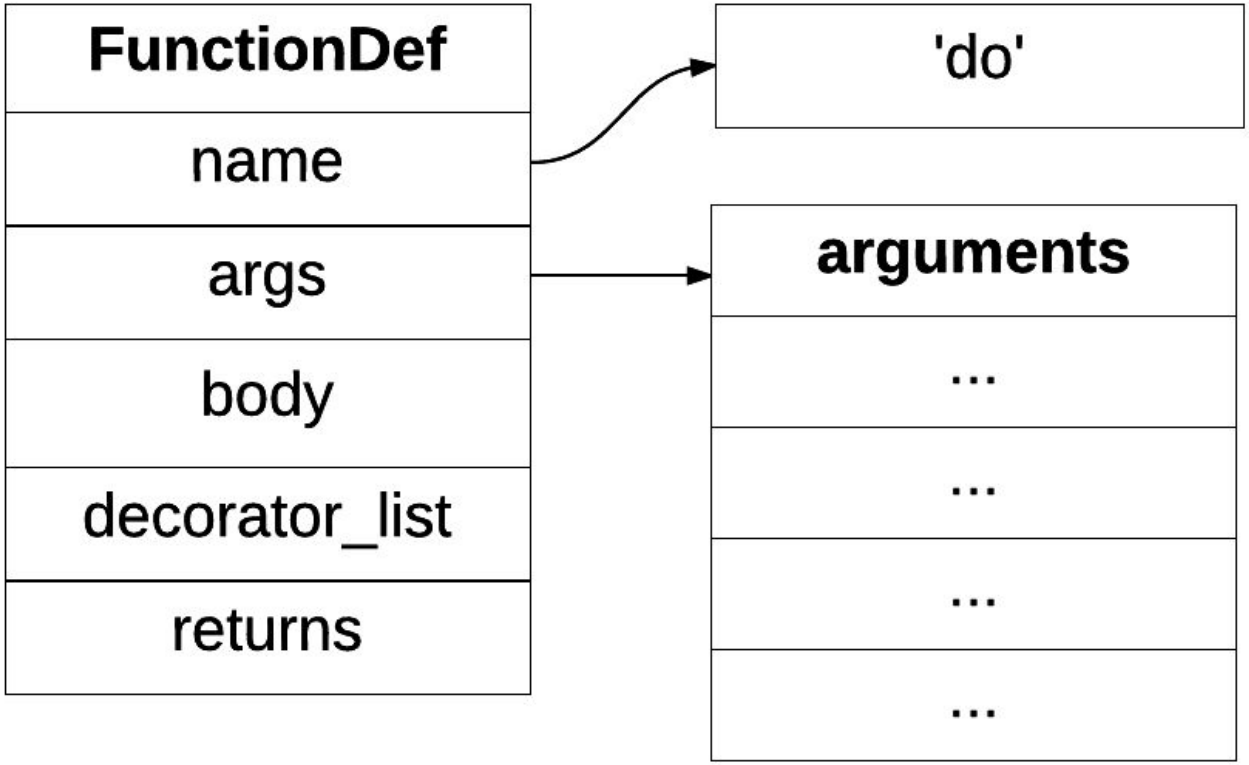
AST FOR a = 1 + 1


Module
Body
Assign
targets
value
Name
id
ctx
'a'
Store
BinOp
left
op
right
Num
n
1
Add
Num
n
1

HOW FUNCTION DEFINITION LOOKS IN AST



HOW FUNCTION DEFINITION LOOKS IN AST



CODE THAT LOOKS FOR FUNCTION DEFINITIONS

```
with open("code_example.py") as source:
    for node in ast.walk(ast.parse(source.read())):
        # Skip everything that is not a function definition
        if not isinstance(node, ast.FunctionDef):
            continue
        # Skip non-public definitions (and constructors in this case)
        if node.name.startswith('_'):
            continue
```

...

CODE THAT LOOKS FOR FUNCTION DEFINITIONS

```
with open("code_example.py") as source:
    for node in
        ast.walk(ast.parse(source.read())):
            # Skip non-public definitions (and constructors in this case)
            if node.name.startswith('_'):
                continue
```

...

CODE THAT LOOKS FOR FUNCTION DEFINITIONS

```
with open("code_example.py") as source:
  for node in ast.walk(ast.parse(source.read())):
    # Skip everything that is not a function definition
    if not isinstance(node, ast.FunctionDef):
        continue
    if node.name.startswith('_'):
      continue
```

...

CODE THAT LOOKS FOR FUNCTION DEFINITIONS

```
with open("code_example.py") as source:
    for node in ast.walk(ast.parse(source.read())):
        # Skip everything that is not a function definition
        if not isinstance(node, ast.FunctionDef):
            continue
```

```
if node.name.startswith('_'):
        continue
```

HOW TO RUN IT? FLAKE8 + TOX

```
├ foobar
│   ├ code_example.py <- code sample
│   └ flakes
│       └ __init__.py <- flake plugin
├ setup.py              <- entry points
└ tox.ini               <- build config
```

FLAKE8 AST PLUGIN: foobar/flakes/_init_.py

```
class KwArgsChecker(object):

    name = 'KwArgsChecker'
    version = '0.1'

    def __init__(self, tree):
        self.tree = tree

    def run(self):
        # our check goes here, AST is at self.tree
```

...

FLAKE8 AST PLUGIN: foobar/flakes/_init_.py

```
class KwArgsChecker(object):
    name = 'KwArgsChecker'
    version = '0.1'
    self.tree = tree

    def run(self):
        # our check goes here, AST is at self.tree
```

...

redhat

FLAKE8 AST PLUGIN: foobar/flakes/_init_.py

```
class KwArgsChecker(object):

    def __init__(self, tree):
        self.tree = tree

    def run(self):
        # our check goes here
```

ENTRY POINTS FOR KwArgsChecker PLUGIN: setup.py

```
from setuptools import setup

setup(name='foobar', version='0.1',
    packages=['foobar', 'foobar.flakes'],
    entry_points={
        'flake8.extension': [
            'X001 = foobar.flakes:KwArgsChecker'
        ]
    }
)
```

...

ENTRY POINTS FOR KwArgsChecker PLUGIN: setup.py

```
from setuptools import setup

packages=['foobar', 'foobar.flakes'],

    entry_points={
        'flake8.extension': [
            'X001 = foobar.flakes:KwArgsChecker'
        ]
    }
)
```

...

ENTRY POINTS FOR KwArgsChecker PLUGIN: setup.py

```
from setuptools import setup

entry_points={
  'flake8.extension': [
    'X001 = foobar.flakes:KwArgsChecker'
  ]
}
```

FLAKE8 AST PLUGIN: foobar/flakes/_init_.py

```
def run(self):
    for node in ast.walk(self.tree):
        # Skip everything that is not a function definition
        if not isinstance(node, ast.FunctionDef):
            continue
        # Skip non-public methods (including constructors)
        if node.name.startswith('_'):
            continue

        arguments = node.args
```

...

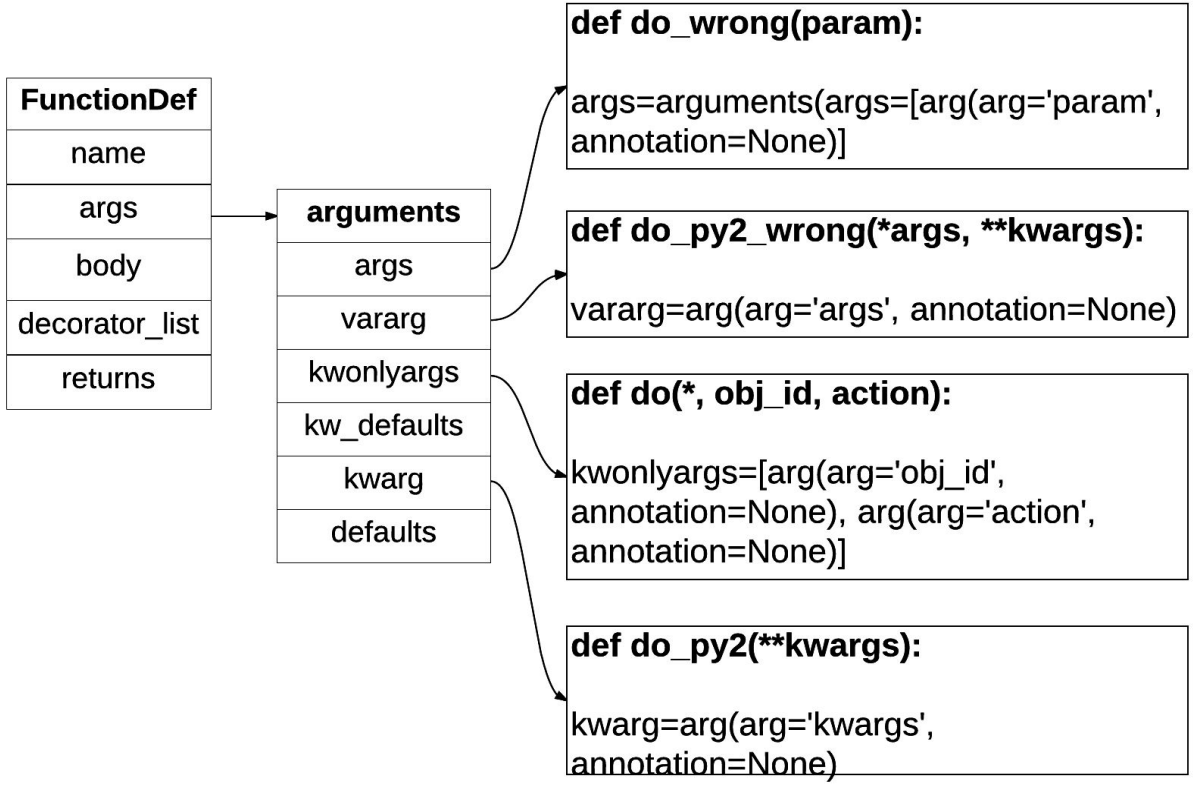
FLAKE8 AST PLUGIN: foobar/flakes/_init_.py

```
def run(self):
    for node in ast.walk(self.tree):

        # ... skipped ...

        arguments = node.args
```

HOW FUNCTION ARGUMENTS LOOK IN AST



...

TERMINOLOGY DISCLAIMER

Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it.

(Python Programming FAQ).

...

POSITIONAL (ARGS)

arguments
args
vararg
kwonlyargs
kw_defaults
kwarg
defaults

```
def do_wrong(param):

args=arguments(args=[arg(arg='param',
annotation=None)]
```

...

VARIABLE (VARARG)

arguments
args
vararg
kwonlyargs
kw_defaults
kwarg
defaults

```
def do_py2_wrong(*args):

vararg=arg(arg='args', annotation=None)
```

...

KEYWORD-ONLY (KWONLYARGS)

arguments
args
vararg
kwonlyargs
kw_defaults
kwarg
defaults

```
def do(*, obj_id, action):

kwonlyargs=[arg(arg='obj_id',
annotation=None), arg(arg='action',
annotation=None)]
```

...

KEYWORD (KWARG)

arguments
args
vararg
kwonlyargs
kw_defaults
kwarg
defaults

```
def do_py2(**kwargs):

kwarg=arg(arg='kwargs', annotation=None)
```

LET'S CHECK FOR WHAT IS WRONG

~~POSITIONAL (ARGS)~~
~~VARIABLE (VARARG)~~
KEYWORD-ONLY (KWONLYARGS)
KEYWORD (KWARG)

But object methods are a bit harder...

CHECKING ARGUMENTS: foobar/flakes/_init_.py

```
# If we have positional arguments defined
if arguments.args:
    # More than one positional argument
    if len(arguments.args) > 1:
        yield self._failure_msg(node)
    # Only one positional argument. We allow only 'self' or 'cls'
    if arguments.args[0].arg not in ['self', 'cls']:
        yield self._failure_msg(node)

# If we take '*args'
if arguments.vararg:
    yield self._failure_msg(node)
```

...

CHECKING ARGUMENTS: foobar/flakes/_init_.py

```
        # If we have positional arguments defined
        # If we have positional arguments
        if arguments.args:

            if arguments.args[0].arg not in ['self', 'cls']:
                yield self._failure_msg(node)

        # If we take '*args'
        if arguments.vararg:
            yield self._failure_msg(node)
```

...

redhat

CHECKING ARGUMENTS: foobar/flakes/_init_.py

```
# If we have positional arguments defined
if arguments.args:
    # More than one positional argument
    if len(arguments.args) > 1:
        yield self._failure_msg(node)
    yield self._failure_msg(node)

# If we take '*args'
if arguments.vararg:
    yield self._failure_msg(node)
```

...

CHECKING ARGUMENTS: foobar/flakes/_init_.py

```
# If we have positional arguments defined
if arguments.args:
```

```
# Only one positional argument
if arguments.args[0].arg
    not in ['self', 'cls']:
        yield self._failure_msg(node)
```

```
if arguments.vararg:
    yield self._failure_msg(node)
```

...

CHECKING ARGUMENTS: foobar/flakes/_init_.py

```
# If we have positional arguments defined
if arguments.args:
    # More than one positional argument
    if len(arguments.args) > 1:
        yield self._failure_msg(node)
```

```
# If we take '*args'
if arguments.vararg:
    yield self._failure_msg(node)
```

```
    yield self._failure_msg(node)
```

REPORTING FAILURE: foobar/flakes/_init_.py

```
def _failure_msg(self, node):
 return (node.lineno,
            node.col_offset,
            'X001 only keyword arguments \
            are allowed in public methods',
            self.name)
```

“CUSTOM” MEANS STRANGE THINGS TOO :-)

```
# Have some fun. It’s up to you!
if 4 <= node.lineno <= 42:
    yield self._failure(node)
```

OR SOMETHING MORE USEFUL

```
def __init__(self, tree, filename):
    self.tree = tree
    self.filename = filename
```

...

redhat

OR SOMETHING MORE USEFUL

```
if self.filename.endswith('/important.py'):
    yield self._failure(node)
```

OUR CHECK IS EXECUTED DURING TOX RUN

```
GLOB sdist-make: /Users/antonm/work/pyconil2017-ast-checks/setup.py
flake8 inst-nodeps: /Users/antonm/work/pyconil2017-ast-checks/.tox/dist/foobar-0.1.zip
flake8 installed: flake8==3.3.0,foobar==0.1,mccabe==0.6.1,pycodestyle==2.3.1,pyflakes==1.5.0
flake8 runtests: PYTHONHASHSEED='2928859785'
flake8 runtests: commands[0] | flake8
```

./foobar/code_example.py:14:1: X001 only keyword arguments are allowed in public methods

```
./foobar/code_example.py:24:1: X001 only keyword arguments are allowed in public methods
./foobar/code_example.py:44:5: X001 only keyword arguments are allowed in public methods
./foobar/code_example.py:52:5: X001 only keyword arguments are allowed in public methods
./foobar/code_example.py:56:5: X001 only keyword arguments are allowed in public methods
ERROR: InvocationError: '/Users/antonm/work/pyconil2017-ast-checks/.tox/flake8/bin/flake8'
```

WHAT HAPPENED DURING TOX RUN?

1. **Tox** creates a **virtual environment** and calls **setuptools**.
2. **Setuptools** installs **foobar** and **foobar.flakes** and register a **plugin** as an **entry point**.
3. **Tox** runs **flake8**.
4. **Flake8** calls **pkg_resources** to enumerate plugins listed in an **entry point**.
5. **Flake8** calls our **plugin** for each python file it finds.

CONCLUSION

- Writing custom code checks is well possible.
- Can be done abstract from specific syntax.
- Can be included into a source tree like unit and other tests.
- Can be run along with other checks using Python build and run tools.

QUESTIONS?

@martchukov
amarchuk@redhat.com
https://github.com/marchukov/talk-ast-checks

REFERENCES

1. Python Software Foundation. Design of CPython's Compiler - Python Developer's Guide. Retrieved from https://docs.python.org/devguide/compiler.html.
2. Ian Cordasco. Writing Plugins for Flake8. Retrieved from http://flake8.pycqa.org/en/latest/plugin-development/index.html.
3. Python Packaging Authority. Package Discovery and Resource Access using pkg_resources. Retrieved from https://setuptools.readthedocs.io/en/latest/pkg_resources.html.
4. Python Software Foundation. ast - Abstract Syntax Trees - Python 3.6.1 documentation. Retrieved from https://docs.python.org/3/library/ast.html.
5. Python Software Foundation. PEP 3102 - Keyword-Only Arguments. Retrieved from https://www.python.org/dev/peps/pep-3102/.
6. Python Software Foundation. What is the difference between arguments and parameters. Programming FAQ. Retrieved from https://docs.python.org/3/faq/programming.html#faq-argument-vs-parameter.
7. Julien Danjou. 2016. The Hacker's Guide to Python. Lulu.com.
8. Brett Slatkin. 2015. Effective Python: 59 Specific Ways to Write Better Python. Addison-Wesley Professional.
9. Keith Cooper, Linda Torczon. 2011. Engineering a Compiler. Elsevier.