Preventing headaches with linters and automated checks

Flávio Juvenal
@flaviojuvenal
vintasoftware.com

Slides are here:

bit.ly/djangocon-linters

Linters and automated checks:

1. **What** are they
2. **Why** using them
3. **How** to implement them
4. **When** to run them
5. **Which** ones exist

What?


lint

lint *(uncountable)*

1. clinging fuzzy fluff that accumulates...
and we hate.

en.wiktionary.org/wiki/lint
(not exactly)

linter

Also valid for software...

linter *(countable)*

1. tool to analyze code to find flaws and errors, helping to remove that clinging fuzzy fluff we hate.

en.wikipedia.org/wiki/Lint_(software)
(not exactly)

```
$ cat my_sum.py
def my_sum(x, x):
    return x + y

$ pyflakes my_sum.py
my_sum.py:1: duplicate argument 'x' in
function definition
my_sum.py:2: undefined name 'y'
```

Why?

What's wrong?

```
from django.db import models


class PersonQuerySet(models.QuerySet):
    def admin_authors(self):
        self.filter(
            role='A', is_admin=True)
```

Can a linter really detect this…?

Yes!

We'll see how

Linters **prevent**:

- bad style (e.g. pycodestyle)
- bad patterns (e.g. flake8-bugbear)
- bugs (e.g. pylint)
- security vulnerabilities (e.g. bandit)

Expert devs know language/framework/library-related quirks and common mistakes. They should **write this knowledge down as linter checks to perpetuate it**.

For orgs, linters
consolidate knowledge
in executable form

- enforce long-lasting good practices
- automate code quality checks
- help to train new developers

Orgs are doing this!

- twisted/twistedchecker
- openstack-dev/hacking
- edx/edx-lint
- saltstack/salt-pylint
- all via custom checks plugins that tools support: pycodestyle, flake8, pylint, bandit, coala, etc.

Orgs are checking:

- blacklisted modules, e.g. test-only libs
- inconsistent string formatting
- `i18n` calls on non-literal strings
- missing try/except on optional third-party imports
- prohibit `locals()` calls
- missing `super()` call on unittest `setUp`/`tearDown`
- too broad `assertRaises` on tests
- etc…

Linters make **better UX** for libraries and frameworks

- error prevention is good UX
- Django does that...

python manage.py check

docs.djangoproject.com/en/dev/topics/checks/

```
$ cat example_project/urls.py
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^app1/', include('app1.urls', namespace='app1')),
    url(r'^app2/', include('app2.urls', namespace='app1')),
]

$ python manage.py check
System check identified some issues:

WARNINGS:
?: (urls.W005) URL namespace 'app1' isn't unique. You may
not be able to reverse all URLs in this namespace

System check identified 1 issue (0 silenced).
```

***class* `ArrayField`(*base_field*, *size=None*, ***options*)**

If you give the field a default, **ensure it's a callable**. Incorrectly using `default=[]` creates a mutable default shared between all instances of `ArrayField`.

docs.djangoproject.com/en/1.11/ref/contrib/postgres/fields/#arrayfield

Suggestion for Django:

Perhaps every "ensure", "remember to", "don't forget", or similar warnings on Django docs should become a new system check…

How?

dynamic analysis

or

static analysis

dynamic analysis

- performed by executing code, not necessarily at runtime
- in order to work, certain checks need to be dynamic, i.e., they need to execute Django. E.g., a check for unapplied migrations
- Django system check framework is dynamic

Let's solve this with a dynamic check:

```
from django.db import models
from django.contrib.postgres.fields import ArrayField

class Post(models.Model):
    tags = ArrayField(
        models.CharField(max_length=200),
        default=[])
```

Demo!

system_checks/checks.py

static analysis

- performed without actually executing code
- safer and more general than dynamic analysis, it can analyse all code flows
- similar to code review, but performed by machines

static analysis types

1. text/regex-based
2. token-based
3. AST-based
4. Inference-based

text/regex-based

- Ideal for simple checks
- Example: dodgy, looks at Python code to search for things which look "dodgy" such as passwords or diffs

token-based

```
$ cat tokenize_me.py
print ("Hi")

$ python -m tokenize tokenize_me.py
...
1,0-1,5:            NAME           'print'
1,6-1,7:            OP             '('
...
```

whitespace_before_parameters @ pycodestyle.py#L699

token-based

- Better to get structure than raw text
- Does not lose info:

  `x == untokenize(tokenize(x))`
- Ideal for style checks
- Example: pycodestyle (formerly pep8), part of flake8

Abstract Syntax Tree-based

```
import ast, astor

tree = ast.parse(
'''
def add(x, y):
    return x + y
''')

print(astor.dump(tree))
```

AST-based

```
Module(
    body=[
        FunctionDef(name='add',
            args=arguments(
                args=[arg(arg='x'),
                      arg(arg='y')]),
            body=[Return(
                value=BinOp(
                    left=Name(id='x'),
                    op=Add,
                    right=Name(id='y')))])])
```

greentreesnakes.readthedocs.io/en/latest/nodes.html

AST-based

- Abstract Syntax Tree: tree structure that represents code
- Abstracts some info
  (e.g. "if-elif-else" becomes nested "if-else"s)
- …

AST-based

- Ideal for checks that need to analyze structure of the code as a whole, checking the relationships between parts, like logic errors
  (e.g. "undefined name")
- Example: pyflakes, part of flake8

AST is made for walking ♪♬♩

```
class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        print(node.name)
        self.generic_visit(node)

FuncLister().visit(tree)
```

Let's solve this with AST walking:

```
from django.db import models


class PersonQuerySet(models.QuerySet):
    def admin_authors(self):
        self.filter(
            role='A', is_admin=True)
```

Let's solve this with AST walking:

class **Call**(*func, args, keywords, starargs, kwargs*)

A function call. `func` is the function, which will often be a `Name` or `Attribute` object. (...)

greentreesnakes.readthedocs.io/en/latest/nodes.html#Call

Let's solve this with AST walking:

class **Expr**(*value*)

When an expression, such as a function call, appears as a statement by itself (an expression statement), with its return value not used or stored, it is wrapped in this container.

`value` holds one of the other nodes in this section (...)

greentreesnakes.readthedocs.io/en/latest/nodes.html#Expr

Demo!

ast_qs.py

What if...

```
class PersonQuerySet(models.QuerySet):
    def authors(self):
        return self.filter(role='A')

    def admin_authors(self):
        self.authors().filter(
            is_admin=True)
```

Would be great if we could infer the type of `self.authors()`…

...actually, we can!

Inference-based

- Can infer info from AST nodes, like imports/variables/attrs resolution, literal op results, classes MRO, types, etc.
- "An interpreter that doesn't execute the code"
- Example: pylint, thanks to astroid lib

Demo!

ast_qs_inferred.py

What about **mypy** type inference?

fjsj
commented 3 days ago

It would be great to have a specific API for the type inference capabilities of mypy. I imagine feeding this API with a typed_ast and getting inferred types for relevant nodes. This is very useful for powerful static analysis, allowing the implementation of even smarter linters for Python.

gvanrossum
commented 3 days ago

Yes, that would be very useful and powerful. It would also be complicated to implement and maintain, so this isn't something we're expecting to be working on in the near (or even medium) future.

Not yet :(

github.com/python/mypy/issues/2097

However...

disobeying_guido_on_mypy.py

google/pytype

- Used in 500+ internal Google projects, all in Python 2
- Only initial Python 3 support

pycharm

- Can work with incomplete ASTs
- Implemented in Java

Both

- Make use of type annotations
- Have inference capabilities

When?

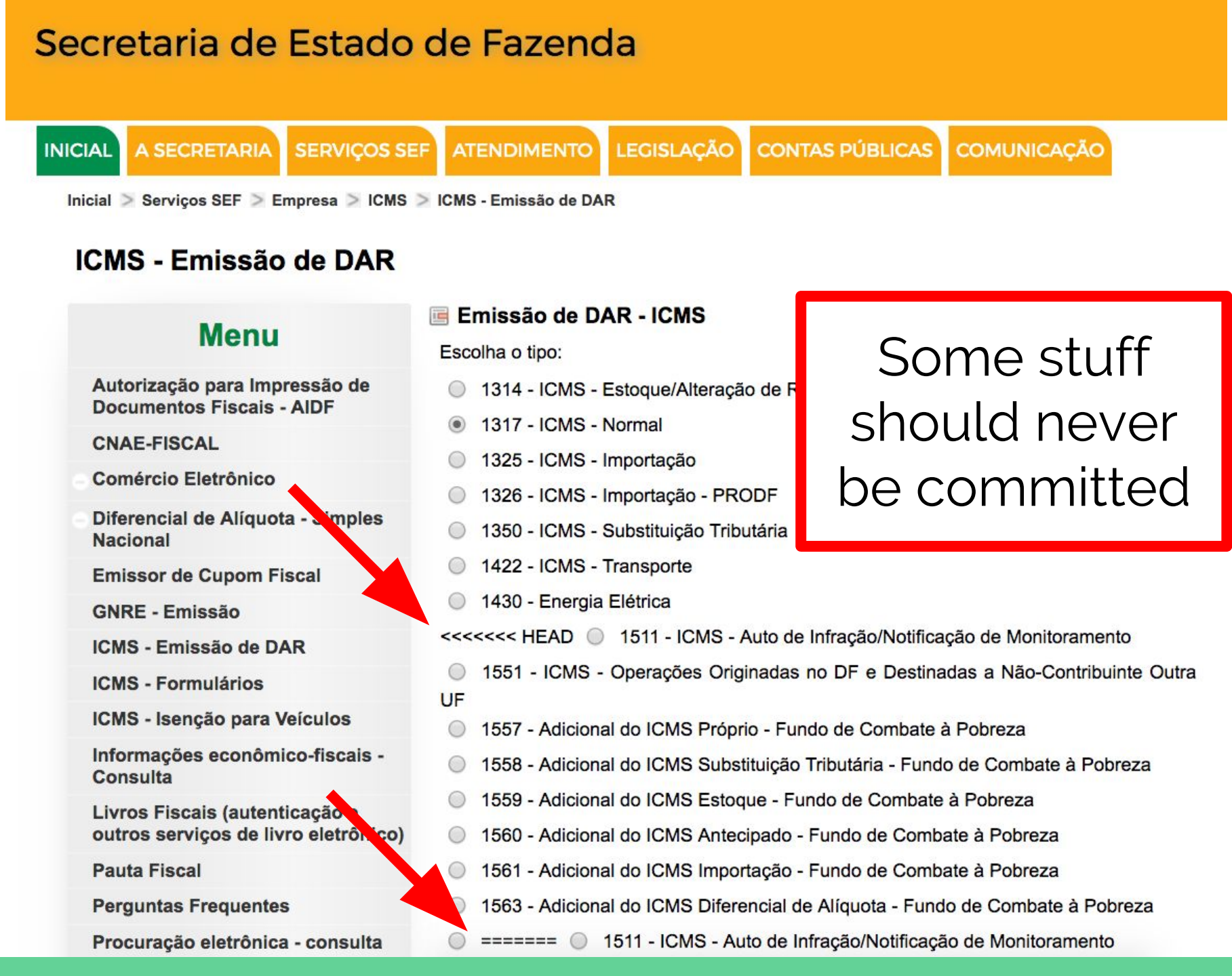
when to run linters

1. programming-time
2. commit-time
3. continuous integration-time
4. code review-time

programming-time

```
23      def visit_expr(self, node):
24          if isinstance(node.value, astroid.Call):
25              try:
26                  for qs in node.value.infer():
27                      if qs.is_subtype_of('django.db.models.query.QuerySet'):
28                          self.add_message ('queryset-expr-not-assigned', node=node)
29                          return
30              except astroid.InferenceError as e:
31                  return
```

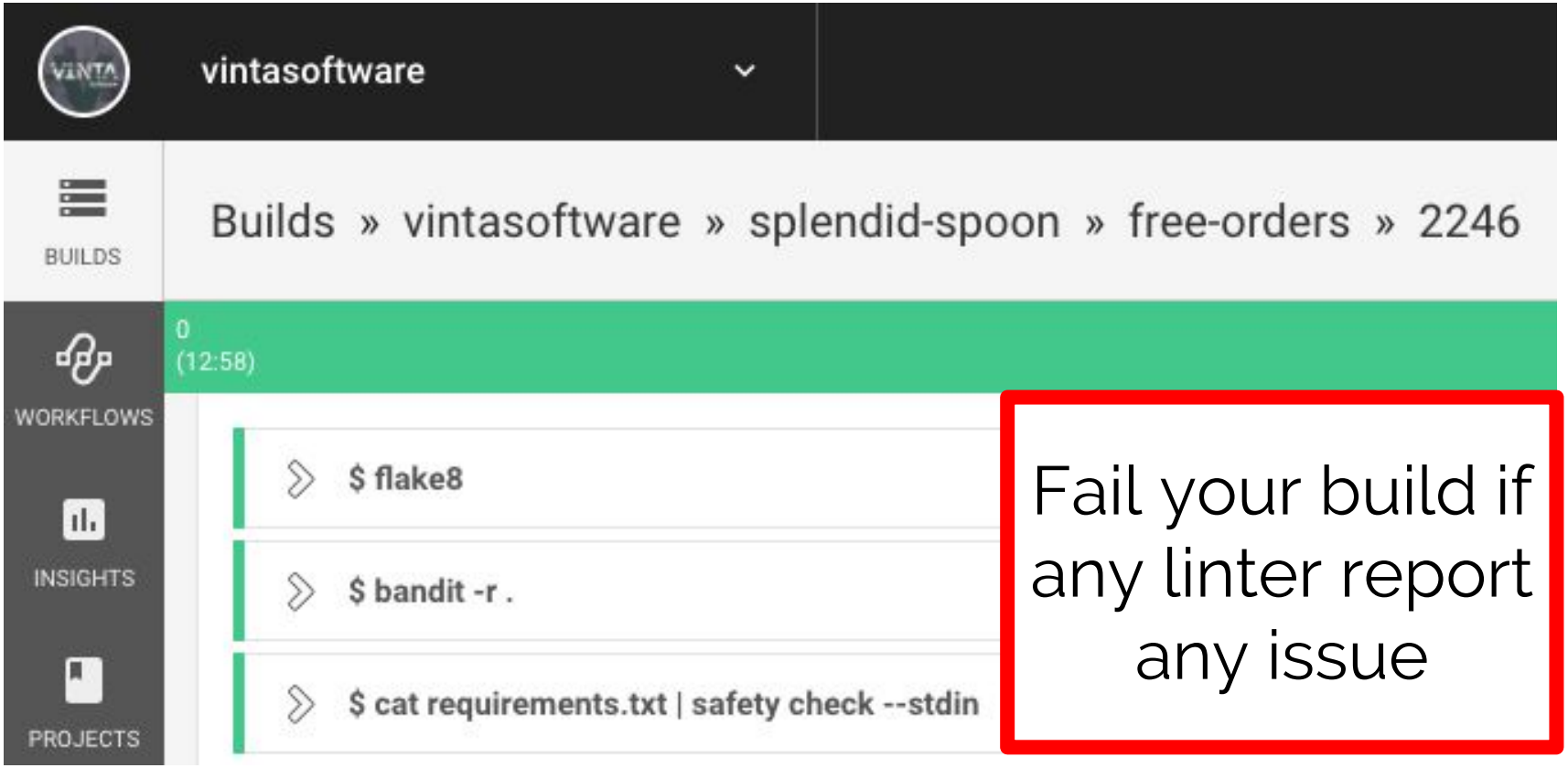
E211 — whitespace before '('


Secretaria de Estado de Fazenda
INICIAL
A SECRETARIA
SERVIÇOS SEF
ATENDIMENTO
LEGISLAÇÃO
CONTAS PÚBLICAS
COMUNICAÇÃO
Inicial > Serviços SEF > Empresa > ICMS > ICMS - Emissão de DAR
ICMS - Emissão de DAR
Menu
Autorização para Impressão de Documentos Fiscais - AIDF
CNAE-FISCAL
Comércio Eletrônico
Diferencial de Alíquota - Simples Nacional
Emissor de Cupom Fiscal
GNRE - Emissão
ICMS - Emissão de DAR
ICMS - Formulários
ICMS - Isenção para Veículos
Informações econômico-fiscais - Consulta
Livros Fiscais (autenticação e outros serviços de livro eletrônico)
Pauta Fiscal
Perguntas Frequentes
Procuração eletrônica - consulta
Emissão de DAR - ICMS
Escolha o tipo:
1314 - ICMS - Estoque/Alteração de R
1317 - ICMS - Normal
1325 - ICMS - Importação
1326 - ICMS - Importação - PRODF
1350 - ICMS - Substituição Tributária
1422 - ICMS - Transporte
1430 - Energia Elétrica
<<<<<<< HEAD
1511 - ICMS - Auto de Infração/Notificação de Monitoramento
1551 - ICMS - Operações Originadas no DF e Destinadas a Não-Contribuinte Outra UF
1557 - Adicional do ICMS Próprio - Fundo de Combate à Pobreza
1558 - Adicional do ICMS Substituição Tributária - Fundo de Combate à Pobreza
1559 - Adicional do ICMS Estoque - Fundo de Combate à Pobreza
1560 - Adicional do ICMS Antecipado - Fundo de Combate à Pobreza
1561 - Adicional do ICMS Importação - Fundo de Combate à Pobreza
1563 - Adicional do ICMS Diferencial de Alíquota - Fundo de Combate à Pobreza
=======
1511 - ICMS - Auto de Infração/Notificação de Monitoramento
Some stuff should never be committed

commit-time

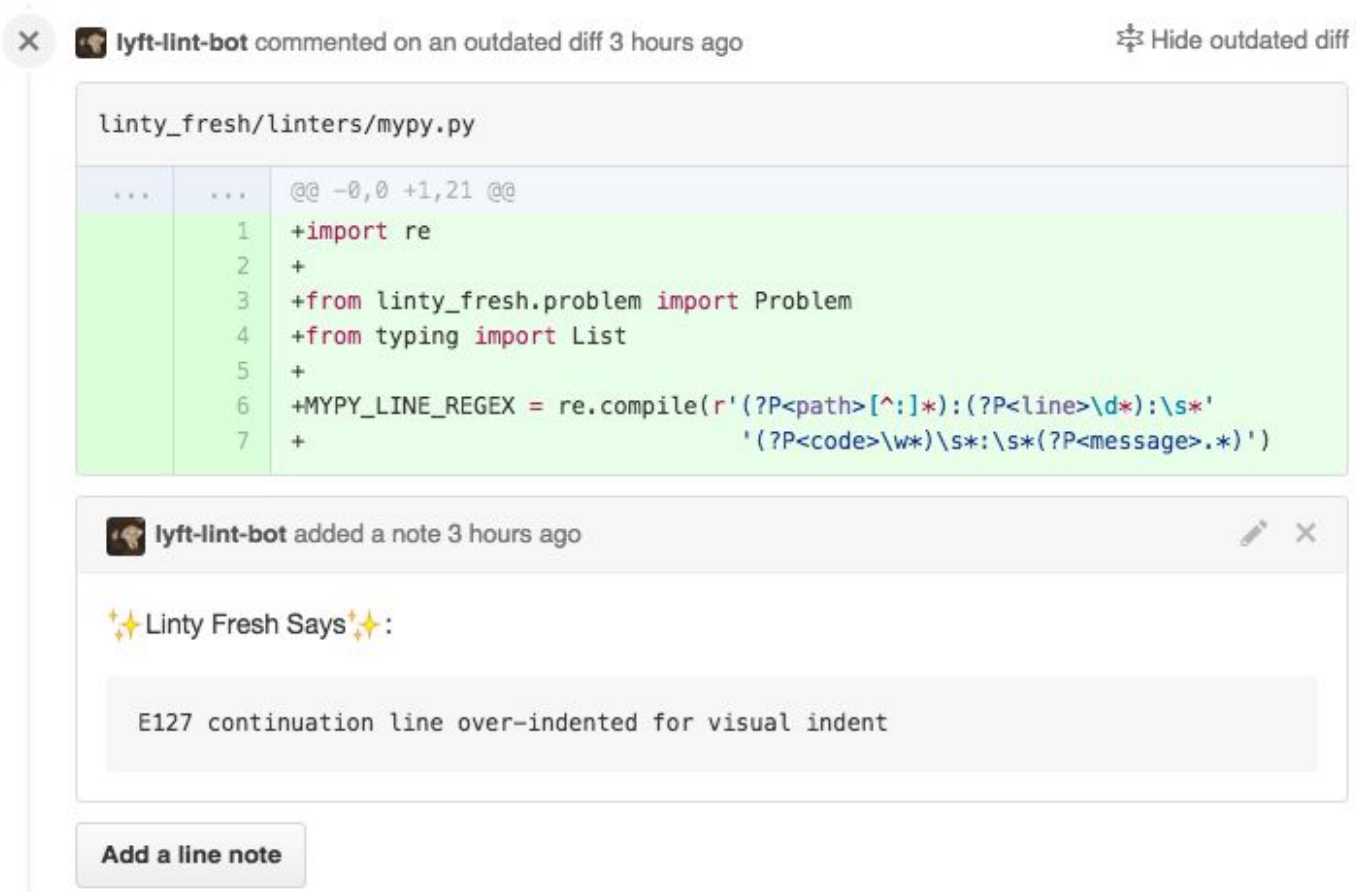
- Git hook to run linters before commit
- pre-commit.com - written in Python!

continuous integration-time



github.com/vintasoftware/django-react-boilerplate

code review-time



github.com/lyft/linty_fresh

Which?

dozens of linters available!

- **quality**: flake8, flake8-bugbear, pylint, pydiatra, vulture
- **imports**: isort, pycycle
- **docs**: pydocstyle
- **security**: bandit, dodgy, pyt, hacking, safety, dependency-check
- **packaging**: pyroma, check-manifest
- **cpython**: cpychecker
- **spelling**: scspell3k
- **typing**: mypy
- wrap them all with prospector (python-only) or coala (general)

github.com/vintasoftware/python-linters-and-code-analysis

some ideas for new Django checks:

1. string formatting in `raw`/`extra` queries
2. `null=True` in `CharField`/`TextField`
3. `null=True` in BooleanField instead of `NullBooleanField`
4. method that overrides save doesn't use commit argument
5. `ModelForm` save doesn't return saved instance
6. Celery task call with ORM model instance as argument
7. `ATOMIC_REQUESTS=True` + Celery `delay` inside view
8. etc…

Let's sprint on those for **django-bug-finder**

If it's difficult for code analysis to understand your code, maybe it'll be difficult for your fellow developers too…

Thanks! Questions?

Feel free to reach me:
twitter.com/flaviojuvenal
flavio@vinta.com.br
vintasoftware.com

Let's contribute:
django-bug-finder
python-linters-and-code-analysis

Slides for this and other Vinta talks at:
bit.ly/vinta2017

References

- Pylint - an overview of the static analysis tool for Python, Claudiu Popa
  https://www.youtube.com/watch?v=p1wPOIYt8Ws
- 12 years of Pylint (or How I learned to stop worrying about bugs)
  https://www.youtube.com/watch?v=0jKbNpEjkhI
  Slides: http://pcmanticore.github.io/pylint-talks/#slide:1
- Andrey Vlasovskikh - Static analysis of Python
  https://www.youtube.com/watch?v=lJtED-xN-HE
- Dave Halter - Identifying Bugs Before Runtime With Jedi
  https://www.youtube.com/watch?v=yPSmj2kmX8g
- Static Code Analysis with Python
  https://www.youtube.com/watch?v=mfXIJ-Fu5Fw
- Writing custom checkers for Pylint
  https://breadcrumbscollector.tech/writing-custom-checkers-for-pylint/
- Writing pylint plugins
  https://nedbatchelder.com/blog/201505/writing_pylint_plugins.html
- Pylint and dynamically populated packages
  http://blog.devork.be/2014/12/pylint-and-dynamically-populated.html

References

- To AST and Beyond by Curtis Maloney
  https://www.youtube.com/watch?v=N_Q3i3oaZ6w
- Andreas Dewes - Learning from other's mistakes: Data-driven analysis of Python code - PyCon 2015
  https://www.youtube.com/watch?v=rN0kNQLDYCI
- Why Pylint is both useful and unusable, and how you can actually use it
  https://codewithoutrules.com/2016/10/19/pylint/
- Interview: Claudiu Popa – Using Pylint for Python Static Analysis
  https://blog.sqreen.io/interview-pylint-for-python-static-analysis/
- Andrey Vlasovskikh - Static analysis of Python
  https://www.youtube.com/watch?v=lJtED-xN-HE
- Static Code Analysis for All Languages - coala (by Lasse Schuirmann)
  https://www.youtube.com/watch?v=oFawYQ0EonY
- Hacking Python AST: checking methods declaration
  https://julien.danjou.info/blog/2015/python-ast-checking-method-declaration

References

- How Python Linters Will Save Your Large Python Project https://jeffknupp.com/blog/2016/12/09/how-python-linters-will-save-your-large-python-project/
- https://github.com/jwilk/check-all-the-things/blob/master/data/python.ini

Extra slides

fixers

- some linters have the ability to automatically fix the code, this is great UX!
- e.g. isort, autoflake, autopep8, etc.
- Coala integrates checkers with fixers seamlessly

more examples of dynamic analysis

- executes in a real-like env or in the real env
- real-like env: pyroma, which runs setup.py to check Python packages health
- real env: a check for unapplied migrations in Django