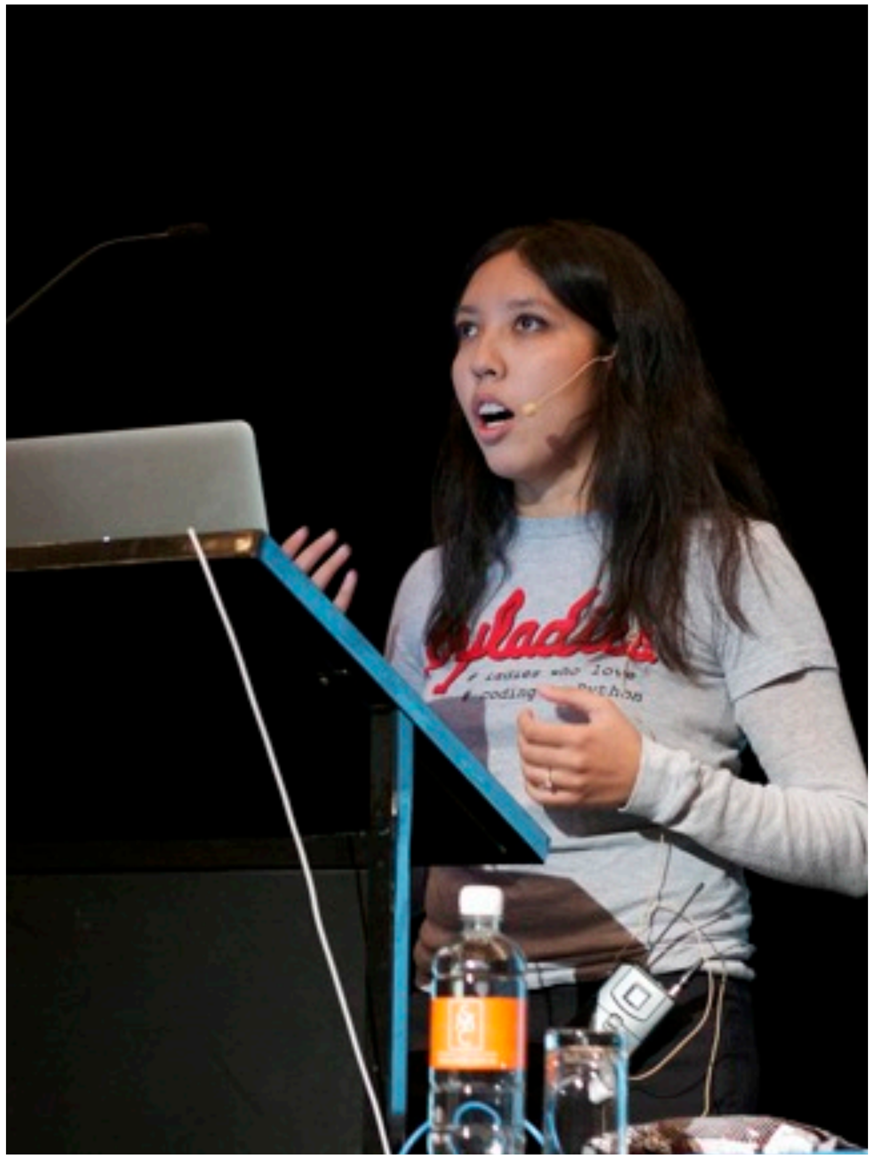
PYCON PHILIPPINES 2012

Audrey Roy
audreyr@cartwheelweb.com
http://www.audreymroy.com
@audreyr

PYTHON TRICKS THAT YOU CAN'T LIVE WITHOUT

ABOUT ME

- Principal at Cartwheel Web
- Massachusetts Institute of Technology EECS (winter 2005)
- Filipina-American and very proud to be here



flickr.com/photos/chrisjrn/6102009780/

I ♥ PYTHON

- OpenComparison core dev, and contributor to various open-source projects
- Co-founded PyLadies
- Helped organize #pyconph
- Python Software Foundation member

Audrey Roy
@audreyr

I ♥ PYTHON

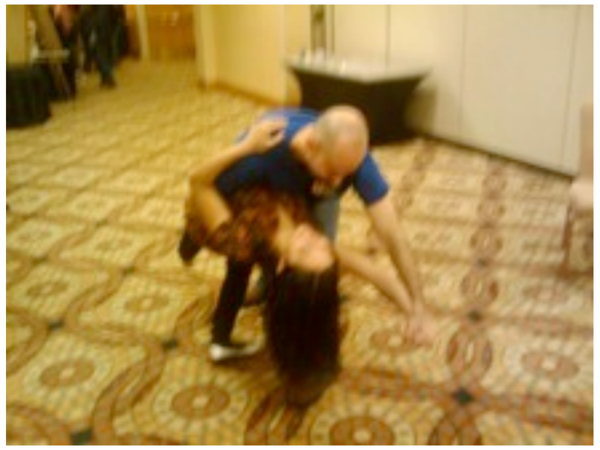
- OpenComparison core dev, and contributor to various open-source projects
- Co-founded PyLadies
- Helped organize #pyconph
- Python Software Foundation member

- I even met my fiancé Daniel Greenfeld at PyCon!

Audrey Roy
@audreyr

I ♥ PYTHON

- OpenComparison core dev, and contributor to various open-source projects
- Co-founded PyLadies
- Helped organize #pyconph
- Python Software Foundation member
- I even met my fiancé Daniel Greenfeld at PyCon!


http://www.flickr.com/photos/47628826@N05/4374285165/

Audrey Roy
@audreyr

OVERVIEW

- Code readability
- Linters and code checkers
- Where to find free reusable Python libraries
- How to package your code for reuse

Audrey Roy
@audreyr

CODE READABILITY

The #1 trick to being a great Python developer is writing clear, understandable code.

CODE READABILITY

- The best Python code is compact, but not too compact
- Write self-documenting code
  - And document it anyway :)

Audrey Roy
@audreyr

CODE READABILITY

Can this be made cleaner?

```
def is_even(x):
    if x % 2 == 0:
        return True
    else:
        return False
```

CODE READABILITY

Can this be made even cleaner?

```
def is_even(x):
    if x % 2 == 0:
        return True
    return False
```

CODE READABILITY

That’s better, but what’s missing?

```
def is_even(x):
    return x % 2 == 0
```

CODE READABILITY

Don’t forget your docstrings

```
def is_even(x):
    """ Returns True if x is even, and
        False if x is odd. """

    return x % 2 == 0
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this
```

```
The Zen of Python, by Tim Peters
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

ZEN OF PYTHON

Keep in mind Python’s philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

ZEN OF PYTHON

Keep in mind Python's philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
```

ZEN OF PYTHON

Keep in mind Python’s philosophy as you code.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
...
```

PEP8

- Python style guide
  - 4 spaces. No tabs!
  - Blank lines between function & class defs
  - Much more...

Audrey Roy
@audreyr

TOOLS FOR CODE READABILITY

Kind of like spell check, but for code

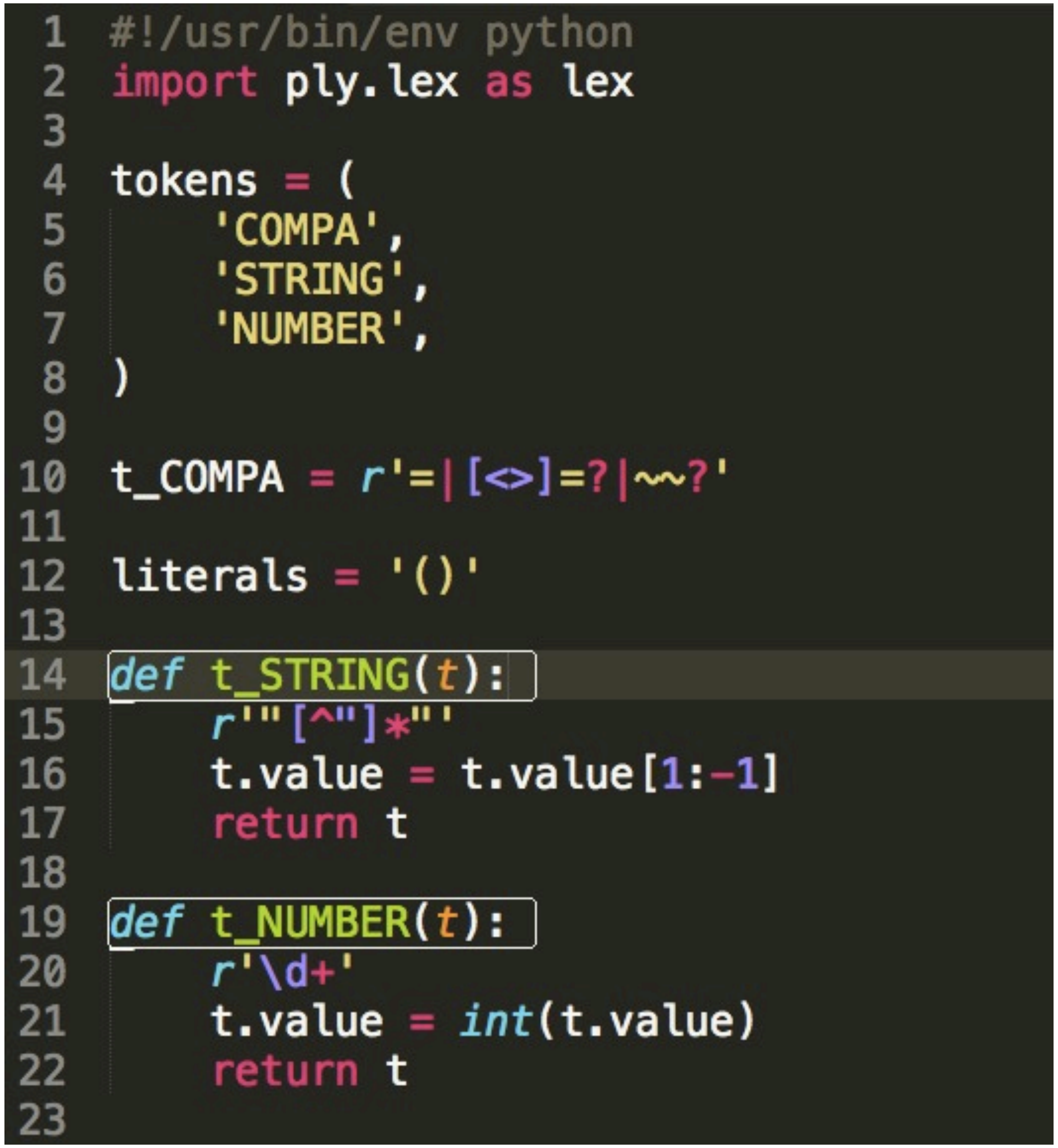
SUBLIME TEXT 2 + PLUGINS

SublimeLinter

Highlights lines of code that are not PEP8-compliant.

Catches potential style issues or errors.

(also for CSS, JS, PHP, Ruby, etc.)



```python
#!/usr/bin/env python
import ply.lex as lex

tokens = (
    'COMPA',
    'STRING',
    'NUMBER',
)

t_COMPA = r'=|[<>]=?|~~?'

literals = '()'

def t_STRING(t):
    r'"[^"]*"'
    t.value = t.value[1:-1]
    return t

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

```

SUBLIME TEXT 2 + PLUGINS

By the way, Sublime Text 2 plugins are simple Python files

To write a plugin, you put a Python file in Sublime's “Packages” directory

PEP8.PY

A command-line PEP8 checker.

http://pypi.python.org/pypi/pep8

PEP8.PY

A command-line PEP8 checker.

```
$  pep8 test2.py
```

http://pypi.python.org/pypi/pep8

PEP8.PY

A command-line PEP8 checker.

```
$  pep8 test2.py
test2.py:13:1: E302 expected 2 blank lines, found 1
```

http://pypi.python.org/pypi/pep8

PEP8.PY

A command-line PEP8 checker.

```
$  pep8 test2.py
test2.py:13:1: E302 expected 2 blank lines, found 1
test2.py:20:1: W391 blank line at end of file
```

http://pypi.python.org/pypi/pep8

PEP8.PY

A command-line PEP8 checker.

```
$  pep8 test2.py
test2.py:13:1: E302 expected 2 blank lines, found 1
test2.py:20:1: W391 blank line at end of file
```

http://pypi.python.org/pypi/pep8

PYLINT

Advanced Python source code analyzer.

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
F:  1,0: Unable to import 'django.db.models'
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
F:  1,0: Unable to import 'django.db.models'
C:  3,0: Invalid name "compa2lookup" (should match (([A-Z_][A-
Z0-9_]*)|(__.*__))$)
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
F:  1,0: Unable to import 'django.db.models'
C:  3,0: Invalid name "compa2lookup" (should match (([A-Z_][A-
Z0-9_]*)|(__.*__))$)
C: 13,0:p_expression_ID: Invalid name "p_expression_ID" (should
match [a-z_][a-z0-9_]{2,30}$)
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
F:  1,0: Unable to import 'django.db.models'
C:  3,0: Invalid name "compa2lookup" (should match (([A-Z_][A-
Z0-9_]*)|(__.*__))$)
C: 13,0:p_expression_ID: Invalid name "p_expression_ID" (should
match [a-z_][a-z0-9_]{2,30}$)
C: 13,0:p_expression_ID: Invalid name "p" (should match [a-z_][a-
z0-9_]{2,30}$)
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
F:  1,0: Unable to import 'django.db.models'
C:  3,0: Invalid name "compa2lookup" (should match (([A-Z_][A-
Z0-9_]*)|(__.*__))$)
C: 13,0:p_expression_ID: Invalid name "p_expression_ID" (should
match [a-z_][a-z0-9_]{2,30}$)
C: 13,0:p_expression_ID: Invalid name "p" (should match [a-z_][a-
z0-9_]{2,30}$)
C: 13,20:p_expression_ID: Invalid name "p" (should match [a-z_][a-
z0-9_]{2,30}$)
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
F:  1,0: Unable to import 'django.db.models'
C:  3,0: Invalid name "compa2lookup" (should match (([A-Z_][A-
Z0-9_]*)|(__.*__))$)
C: 13,0:p_expression_ID: Invalid name "p_expression_ID" (should
match [a-z_][a-z0-9_]{2,30}$)
C: 13,0:p_expression_ID: Invalid name "p" (should match [a-z_][a-
z0-9_]{2,30}$)
C: 13,20:p_expression_ID: Invalid name "p" (should match [a-z_][a-
z0-9_]{2,30}$)
C: 18,4:p_expression_ID: Invalid name "d" (should match [a-z_][a-
z0-9_]{2,30}$)
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
$ pylint test2.py
No config file found, using default configuration
************* Module test2
C:  1,0: Missing docstring
F:  1,0: Unable to import 'django.db.models'
C:  3,0: Invalid name "compa2lookup" (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C: 13,0:p_expression_ID: Invalid name "p_expression_ID" (should match [a-z_][a-z0-9_]{2,30}$)
C: 13,0:p_expression_ID: Invalid name "p" (should match [a-z_][a-z0-9_]{2,30}$)
C: 13,20:p_expression_ID: Invalid name "p" (should match [a-z_][a-z0-9_]{2,30}$)
C: 18,4:p_expression_ID: Invalid name "d" (should match [a-z_][a-z0-9_]{2,30}$)
W: 19,11:p_expression_ID: Used * or ** magic
```

http://pypi.python.org/pypi/pylint

PYLINT

Advanced Python source code analyzer.

```
Report
======
8 statements analysed.

Messages by category
--------------------
+-----------+-------+---------+-----------+
|type       |number |previous |difference |
+===========+=======+=========+===========+
|convention |6      |NC       |NC         |
+-----------+-------+---------+-----------+
|refactor   |0      |NC       |NC         |
+-----------+-------+---------+-----------+
|warning    |1      |NC       |NC         |
+-----------+-------+---------+-----------+
```

http://pypi.python.org/pypi/pylint

FINDING PYTHON LIBRARIES

"Free stuff for Python developers!"

FINDING CODE TO REUSE

Where to get FREE reusable Python libraries:

1. Python Standard Library
   - Many great essentials, already on your system!
   - http://docs.python.org/library/index.html
2. Python Package Index
   - 21,000+ packages to download!
   - http://pypi.python.org/

Audrey Roy
@audreyr

WHY REUSE CODE?

- Python helps you avoid reinventing the wheel
  - “Not Invented Here” syndrome

Audrey Roy
@audreyr

MORE ABOUT THE PYTHON STDLIB

A collection of highly useful modules

- No need to install
- Just import and start using them!

Audrey Roy
@audreyr

STDLIB EXAMPLE: MATH

STDLIB EXAMPLE: MATH

```
>>> import math
```

STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
```

STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
```

STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
```

STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
2.0
```

STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
2.0
>>> math.log(32,2)
```

STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
2.0
>>> math.log(32,2)
5.0
```

STDLIB EXAMPLE: MATH

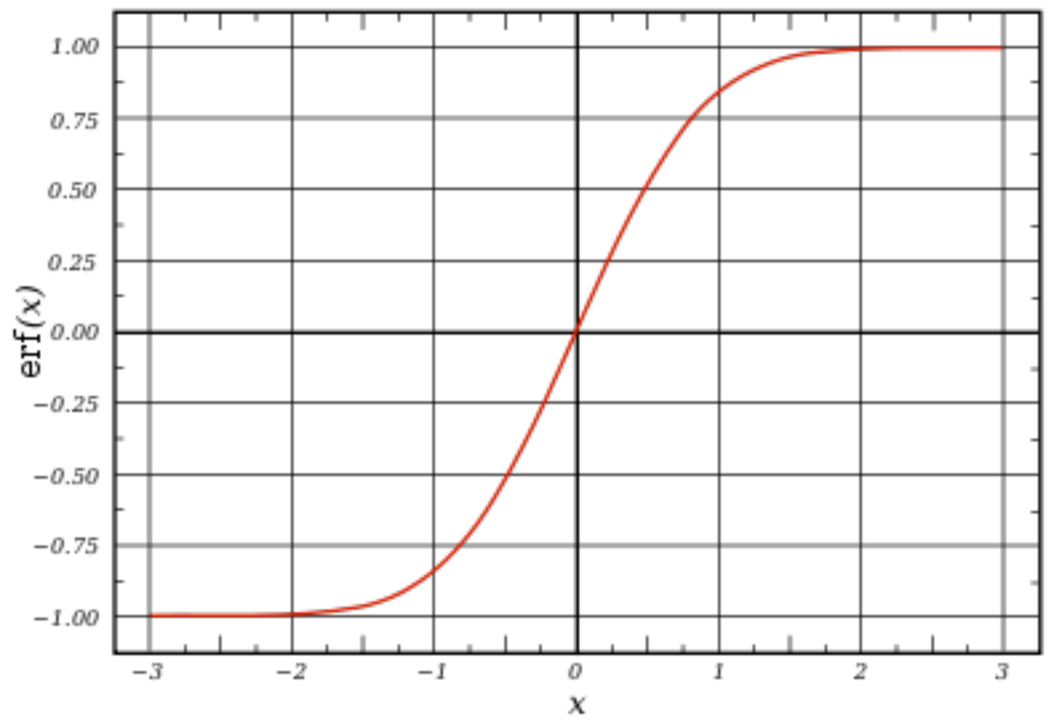
```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
2.0
>>> math.log(32,2)
5.0
>>> math.erf(0.5)
```

STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
2.0
>>> math.log(32,2)
5.0
>>> math.erf(0.5)
0.5204998778130465
```

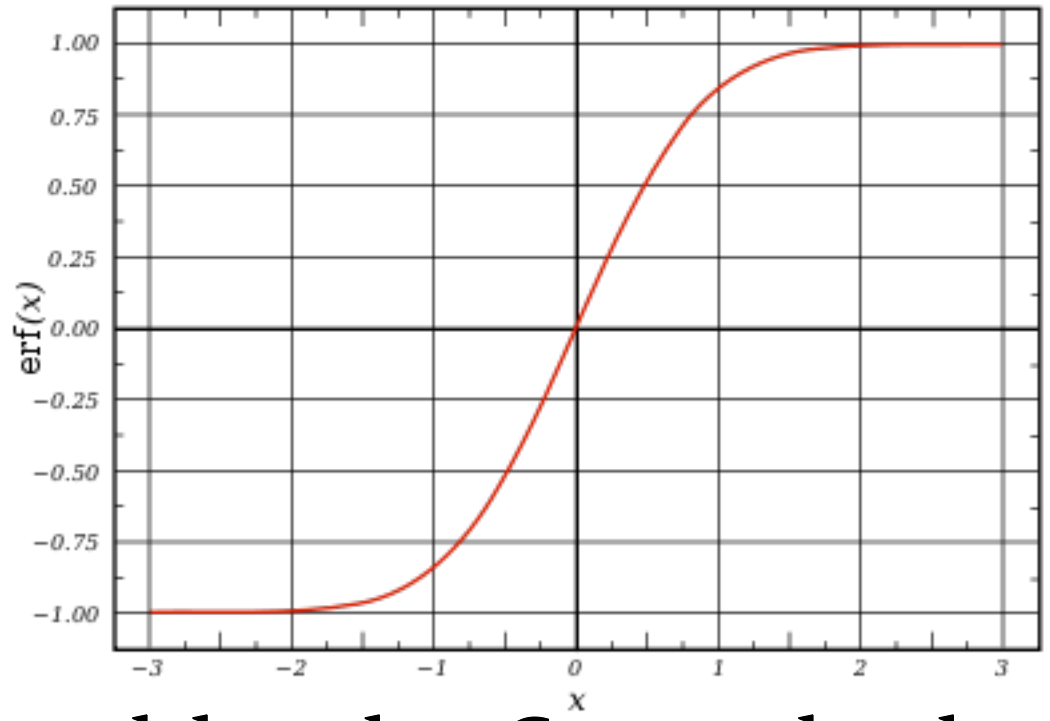
STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
2.0
>>> math.log(32,2)
5.0
>>> math.erf(0.5)
0.5204998778130465
```



STDLIB EXAMPLE: MATH

```
>>> import math
>>> math.ceil(2.03)
3.0
>>> math.floor(2.99)
2.0
>>> math.log(32,2)
5.0
>>> math.erf(0.5)
0.5204998778130465
```



Mathematical functions defined by the C standard

STDLIB EXAMPLE: RANDOM

STDLIB EXAMPLE: RANDOM

```
>>> import random
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
0.12863367604888531
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
0.12863367604888531
>>> random.uniform(0,100)
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
0.12863367604888531
>>> random.uniform(0,100)
25.374019279313988
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
0.12863367604888531
>>> random.uniform(0,100)
25.374019279313988
>>> math.floor(random.uniform(0,100))
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
0.12863367604888531
>>> random.uniform(0,100)
25.374019279313988
>>> math.floor(random.uniform(0,100))
77.0
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
0.12863367604888531
>>> random.uniform(0,100)
25.374019279313988
>>> math.floor(random.uniform(0,100))
77.0
>>> random.randrange(0,100)
```

STDLIB EXAMPLE: RANDOM

```
>>> import random
>>> random.random()
0.12863367604888531
>>> random.uniform(0,100)
25.374019279313988
>>> math.floor(random.uniform(0,100))
77.0
>>> random.randrange(0,100)
69
```

MORE ABOUT PyPI

- PyPI is “Python Package Index”
- 21,000+ packages
  - All created by community members like you
- http://pypi.python.org

Audrey Roy
@audreyr

PyPI EXAMPLES

- You saw some great examples already from PyPI (Python Package Index)
  - pep8: Simple PEP8 syntax checker
  - pylint: Advanced source code analyzer

STDLIB VS. PYPI

- The stdlib is conservative
  - Few additions/changes/deprecations
- On PyPI, anything goes!

STDLIB VS. PyPI

- Sometimes PyPI packages are better than the equivalent stdlib ones
  - e.g. requests is better than urllib2
- If in doubt, ask around

INSTALLING PYTHON PACKAGES

The wrong way, and the right way

THE WRONG WAY

- Systemwide installation of Python libraries is generally bad
- You can make a mess of your system

THE RIGHT WAY

You really should be using these 2 tools:

- pip - a good package installer
- virtualenv - create isolated Python envs

I strongly recommend virtualenvwrapper too.

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
(consumer_io) $ python manage.py runserver
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
(consumer_io) $ python manage.py runserver
(consumer_io) $ ...
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
(consumer_io) $ python manage.py runserver
(consumer_io) $ ...
(consumer_io) $ deactivate
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
(consumer_io) $ python manage.py runserver
(consumer_io) $ ...
(consumer_io) $ deactivate
$ cd ../../experiments
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
(consumer_io) $ python manage.py runserver
(consumer_io) $ ...
(consumer_io) $ deactivate
$ cd ../../experiments
$ workon experiments
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
(consumer_io) $ python manage.py runserver
(consumer_io) $ ...
(consumer_io) $ deactivate
$ cd ../../experiments
$ workon experiments
(experiments) $ python somethingelse.py
```

THE RIGHT WAY: VIRTUALENV

Create isolated virtualenvs for different projects.

```
$ workon consumer_io
(consumer_io) $ cd consumer_io/proj/
(consumer_io) $ python manage.py runserver
(consumer_io) $ ...
(consumer_io) $ deactivate
$ cd ../../experiments
$ workon experiments
(experiments) $ python somethingelse.py
(experiments) $ ...
```

THE RIGHT WAY: PIP

THE RIGHT WAY: PIP

Use pip to install packages into virtualenvs.

THE RIGHT WAY: PIP

Use pip to install packages into virtualenvs.

```
(experiments) $ pip install Django==1.4
```

THE RIGHT WAY: PIP

Use pip to install packages into virtualenvs.

```
(experiments) $ pip install Django==1.4
```

pip is like easy_install, but much better.

THE RIGHT WAY: PIP+VIRTUALENV

SCENARIO:
You use Django 1.3 for work, but you want to experiment with Django 1.4.

With pip and virtualenv, you can switch between 1.3 and 1.4 on the same computer.

PIP REQUIREMENTS FILES

You should pin your dependencies in requirements.txt!

```
$ pip install -r requirements.txt

# Your requirements.txt file
Flask==0.8
glue==0.2.5
Pillow==1.7.7
Django==1.4
```

Use `pip install PackageName==1.0.4` for experimentation only.

AFTER INSTALLATION?

Once installed, you can import Python code from modules:

```
from collections import deque
```

Or from submodules:

```
from os.path import abspath
```

WRITING REUSABLE CODE

How code reuse works in Python

MODULES

A module is a file containing Python definitions and statements.

Like this:

```
# divisible.py

def is_even(x):
    """ Returns True if x is even, and
        False if x is odd. """

    return x % 2 == 0
```

PACKAGES

A Python package is a collection of modules.

PACKAGES

A Python package is a collection of modules.

```
sound/
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
    effects/
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
    effects/
        __init__.py
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
    effects/
        __init__.py
        echo.py
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
    effects/
        __init__.py
        echo.py
        surround.py
```

PACKAGES

A Python package is a collection of modules.

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
    effects/
        __init__.py
        echo.py
        surround.py
```

PACKAGES

A sample import from this package:

```
from sound.formats.wav import read_wav
```

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
    effects/
        __init__.py
        echo.py
        surround.py
```

INTRA-PACKAGE IMPORTS

Relative imports work between submodules of a package:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

INTRA-PACKAGE IMPORTS

Absolute imports work between submodules of a package:

```
# Use this from anywhere in the package
from sound.effects import echo
```

package root

IMPORTING FROM OUTSIDE A PACKAGE

- Can’t use absolute/relative imports
- What to do? One of these:
  - Good: Add the package to PYTHONPATH [edit env var or use sys.path.append()]
  - Better: Install the package into your active virtualenv.

BETTER PACKAGING

- Recommended reading: "The Hitchhiker's Guide To Packaging"
  - http://guide.python-distribute.org
  - Learn to make packages that are downloadable & installable from PyPI

THANK YOU

- Find me if you have questions
- Introduce yourself - I'd love to meet you!
- Twitter: @audreyr
- Email: audreyr@cartwheelweb.com

Audrey Roy
@audreyr