

PEP yourself: 10 PEPs you should pay attention to

Another look at known and lesser known PEPs

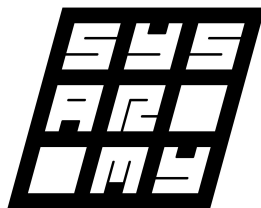
Juan Manuel Santos || godlike
EuroPython 2019 / 2019-07-12

Doctor who?



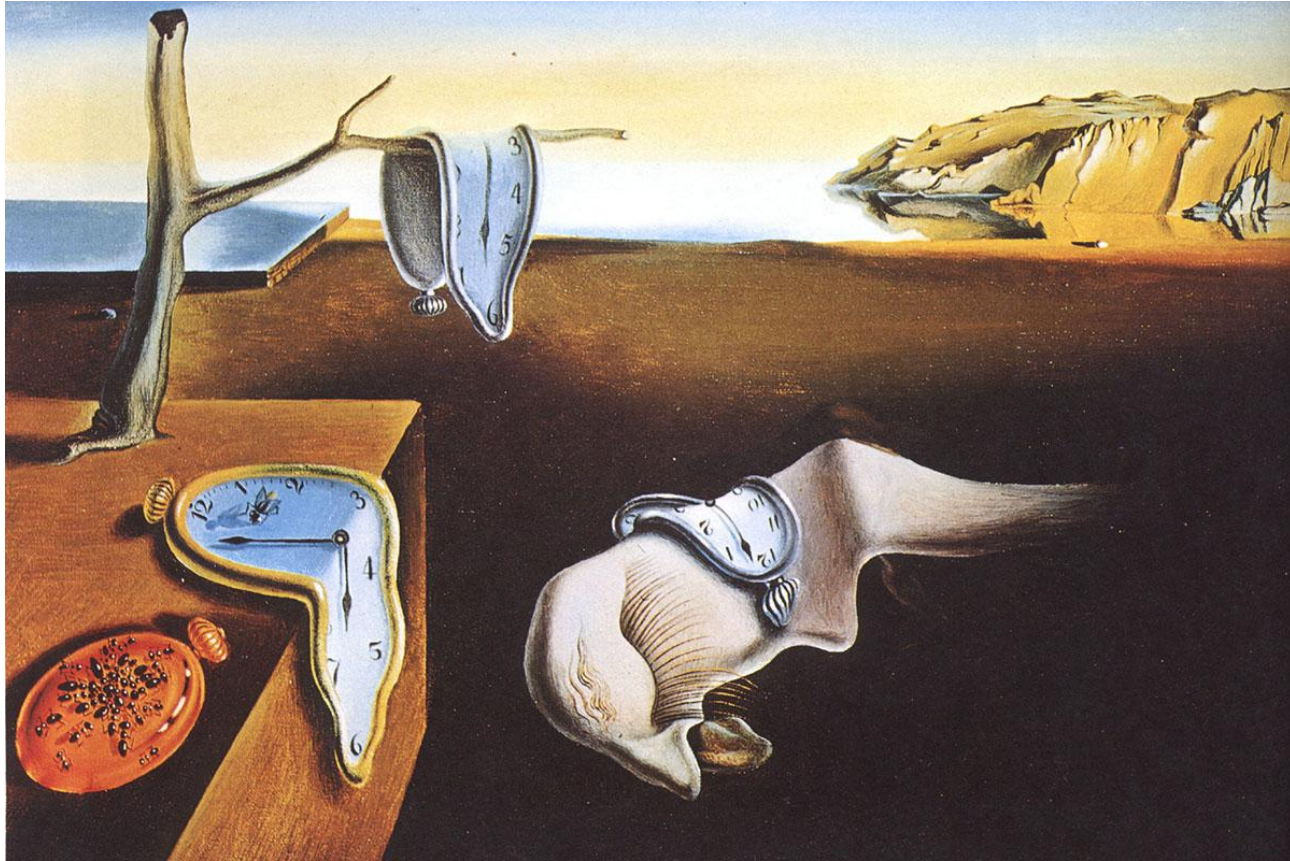
- Juan Manuel Santos.
 - IRC: godlike.
 - Twitter: godlike64.
 - Principal Technical Support Engineer @ Red Hat.
-
- Linux.
 - Python.





NERDEAR.LA

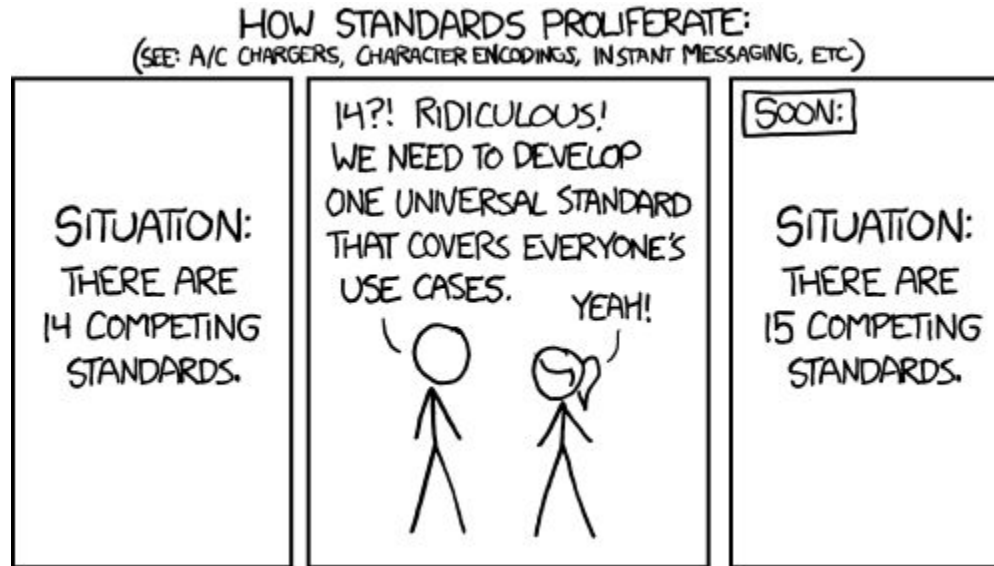




o/?

Why?

- Always liked standards.
- A way of moving things forward.



Why?

- Standards are a way of communication.
- Improve, refine, finalize an idea.
- Put it in a box.
- Share it with others.
- Improve some more.
- Seal and stamp the box.



How?

- I love Python but I am no Python expert superhero.
- I also love Open Source.
- ...
- Wait!

How?



GODLiKE
@godlike64



If you use Python, which would you say are the most relevant PEPs? (8, 20 & 257 excluded)
please RT, trying to reach as many users as possible

1:50 PM · Nov 28, 2018 · [Twitter for Android](#)

How?



What is a PEP?

- Python Enhancement Proposal.
- Design documents that provide information to the community.
- New features, processes, environment.

“We intend PEPs to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.”

Mandatory Monty Python reference



Why PEPs?

- Enhance Python.
- Implement new features.
- Better language → moar people!

General PEPs

PEP 8

PEP 8 -- Style Guide for Python Code

- Seed for this talk.
- Covers a lot of things on how to write your Python code:
 - Indentation.
 - Naming.
 - Imports.
 - Write code with other Python implementations in mind.

PEP 8 -- Style Guide for Python Code

- **Line length.**
- 79 characters max.

“Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.”

PEP 8 -- Style Guide for Python Code

```
1
2 import webbrowser
3 import hashlib
4
5 webbrowser.open("https://xkcd.com/353/")
6
7 def geohash(latitude, longitude, datedow):
8     '''Compute geohash() using the Munroe algorithm.
9
10     >>> geohash(37.421542, -122.085589, b'2005-05-26-10458.68')
11     37.857713 -122.544543
12
13     '''
14     # https://xkcd.com/426/
15     h = hashlib.md5(datedow).hexdigest()
16     p, q = [('%.f' % float.fromhex('0.' + x)) for x in (h[:16], h[16:32])]
17     print('%d%s %d%s' % (latitude, p[1:], longitude, q[1:]))
```

PEP 8 -- Style Guide for Python Code

- *Trey Hunner: “Craft Your Python Like Poetry.”*
 - <https://treyhunner.com/2017/07/craft-your-python-like-poetry/>
- It is not a technical limitation.
- It is a **human imposed limitation**.
- Humans read shorter lines better (think newspapers).
- **Python isn't prose, it's poetry.**
- Craft it as such.

PEP 8 -- Style Guide for Python Code

```
1 employee_hours = [schedule.earliest_hour for employee in self.public_employees for schedule in employee.schedules]
2
```

PEP 8 -- Style Guide for Python Code

```
1 employee_hours = [schedule.earliest_hour
2                     for employee in self.public_employees
3                     for schedule in employee.schedules]
4
```

PEP 8 -- Style Guide for Python Code

```
1 employee_hours = [  
2     schedule.earliest_hour  
3     for employee in self.public_employees  
4     for schedule in employee.schedules  
5 ]  
6
```


PEP 8 -- Style Guide for Python Code

```
1 books = Book.objects.filter(author__in=favorite_authors).select_related('author', 'publisher').order_by('title')
2
```

PEP 8 -- Style Guide for Python Code

```
1 books = Book.objects.filter(  
2     author__in=favorite_authors).select_related(  
3     'author', 'publisher').order_by('title')
```

PEP 8 -- Style Guide for Python Code

```
1 books = (Book.objects
2         .filter(author__in=favorite_authors)
3         .select_related('author', 'publisher')
4         .order_by('title'))
5
```

PEP 8 -- Style Guide for Python Code

```
1 books = (  
2     Book  
3     .objects  
4     .filter(author__in=favorite_authors)  
5     .select_related('author', 'publisher')  
6     .order_by('title')  
7 )  
8
```

PEP 257

PEP 257 -- Docstring Conventions

- *“Working software over comprehensive documentation.”*



PEP 257 -- Docstring Conventions

- *“If you violate these conventions, the worst you'll get is some dirty looks.”*



PEP 257 -- Docstring Conventions

- The `__doc__` attribute.
- All modules, all exported functions and classes from a module, as well as all public methods **should** have a docstring.
- Your docstring → actual docs!

PEP 257 -- Docstring Conventions

```
def kos_root():  
    """Return the pathname of the KOS root directory."""  
    global _kos_root  
    if _kos_root: return _kos_root  
    ...
```

PEP 257 -- Docstring Conventions

```
def complex(real=0.0, imag=0.0):  
    """Form a complex number.
```



```
    Keyword arguments:
```

```
    real -- the real part (default 0.0)
```



```
    imag -- the imaginary part (default 0.0)
```

```
    """
```

```
    ...
```

PEP 257 -- Docstring Conventions

```
>>> import os
>>> print(os.__doc__)
OS routines for NT or Posix depending on what system we're on.
```

This exports:

- all functions from posix or nt, e.g. unlink, stat, etc.
- os.path is either posixpath or ntpath
- os.name is either 'posix' or 'nt'
- os.curdir is a string representing the current directory (always '.')
- os.pardir is a string representing the parent directory (always '..')
- os.sep is the (or a most common) pathname separator ('/' or '\\')
- os.extsep is the extension separator (always '.')
- os.altsep is the alternate pathname separator (None or '/')
- os.pathsep is the component separator used in \$PATH etc
- os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
- os.defpath is the default search path for executables
- os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being portable between different platforms. Of course, they must then only use functions that are defined by all platforms (e.g., unlink and opendir), and leave all pathname manipulation to os.path (e.g., split and join).

PEP 257 -- Docstring Conventions

```
>>> print(os.path.__doc__)  
Common operations on Posix pathnames.
```

Instead of importing this module directly, import `os` and refer to this module as `os.path`. The "`os.path`" name is an alias for this module on Posix systems; on other systems (e.g. Mac, Windows), `os.path` provides the same operations in a manner specific to that platform, and is an alias to another module (e.g. `macpath`, `ntpath`).

Some of this can actually be useful on non-Posix systems too, e.g. for manipulation of the pathname component of URLs.

PEP 257 -- Docstring Conventions

```
>>> print(os.path.exists.__doc__)
```

```
Test whether a path exists. Returns False for broken symbolic links
```

PEP 257 -- Docstring Conventions

- reStructuredText Docstring Format
 - <https://www.python.org/dev/peps/pep-0287/>
- Napoleon
 - <https://sphinxcontrib-napoleon.readthedocs.io/en/latest/>
- *Good programmers write code that humans can understand.* — Martin Fowler.
- *If there's something developers respect, it's code.* — Hynek Schlawack.

PEP 3099

PEP 3099 -- Things that will Not Change in Python 3000

“If you think you should suggest any of the listed ideas it would be better to just step away from the computer, go outside, and enjoy yourself. Being active outdoors by napping in a nice patch of grass is more productive than bringing up a beating-a-dead-horse idea and having people tell you how dead the idea is. Consider yourself warned.”

PEP 3099 -- Things that will Not Change in Python 3000

- Rationale always explained.
- Or link to long mailing list discussion
- pYtHoN wOn'T bE cAsE iNsEnSiTiVe
- Slices and extended slices are here to stay.
- Maximum line length will stay at 80 characters.



PEP 3099 -- Things that will Not Change in Python 3000

- *“The interpreter prompt (>>>) will not change. It gives Guido warm fuzzy feelings.”*



Fun PEPs

PEP 202

PEP 202 -- List Comprehensions

- Generate lists in one line! No indentation required!
- **Faster.**
- Take an iterable → generate a list.

```
>>> print([i for i in range(10)])  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

PEP 202 -- List Comprehensions

- Generate lists in one line! No indentation required!
- **Faster.**
- Take an iterable → generate a list.

```
>>> print()
```

PEP 202 -- List Comprehensions

- Generate lists in one line! No indentation required!
- **Faster.**
- Take an iterable → generate a list.

```
>>> print([])
```


PEP 202 -- List Comprehensions

- Generate lists in one line! No indentation required!
- **Faster.**
- Take an iterable → generate a list.

```
>>> print([for i in range(10)])
```

PEP 202 -- List Comprehensions

- Generate lists in one line! No indentation required!
- **Faster.**
- Take an iterable → generate a list.

```
>>> print([i for i in range(10)])  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

PEP 202 -- List Comprehensions

- Can have filtering:

```
>>> print([i for i in range(20) if i%2 == 0])  
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- Or apply a transformation to all elements:

```
print([i if i%2 == 0 else 'odd' for i in range (20)])  
[0, 'odd', 2, 'odd', 4, 'odd', 6, 'odd', 8, 'odd', 10,  
'odd', 12, 'odd', 14, 'odd', 16, 'odd', 18, 'odd']
```

PEP 202 -- List Comprehensions

- More than one source iterable:

```
>>> nums = [1, 2, 3, 4]
>>> fruit = ["Apples", "Peaches", "Pears", "Bananas"]
>>> print([(i, f) for i in nums for f in fruit])
[(1, 'Apples'), (1, 'Peaches'), (1, 'Pears'), (1,
'Bananas'), (2, 'Apples'), (2, 'Peaches'), (2, 'Pears'), (2,
'Bananas'), (3, 'Apples'), (3, 'Peaches'), (3, 'Pears'), (3,
'Bananas'), (4, 'Apples'), (4, 'Peaches'), (4, 'Pears'), (4,
'Bananas')]
```

PEP 202 -- List Comprehensions

- There's also a younger brother: dict comprehensions!

```
>>> print({i : chr(65+i) for i in range(4)})  
{0: 'A', 1: 'B', 2: 'C', 3: 'D'}
```

PEP 202 -- List Comprehensions

- There's also a younger brother: dict comprehensions!

```
>>> print({i : chr(65+i) for i in range(4)})  
{0: 'A', 1: 'B', 2: 'C', 3: 'D'}
```

PEP 202 -- List Comprehensions

- There's also a younger brother: dict comprehensions!

```
>>> print({i : chr(65+i) for i in range(4)})  
{0: 'A', 1: 'B', 2: 'C', 3: 'D'}
```

PEP 234

PEP 234 -- Iterators

- Controlled for loops.
1. A method produces an **iterator object**.
 2. The iterator object provides a **next()** method.
 3. **next()** will return one element at a time, until no more elements are available.
 4. **StopIteration**.

PEP 234 -- Iterators

- Iteration interface already implemented in all for loops.
- This allows:

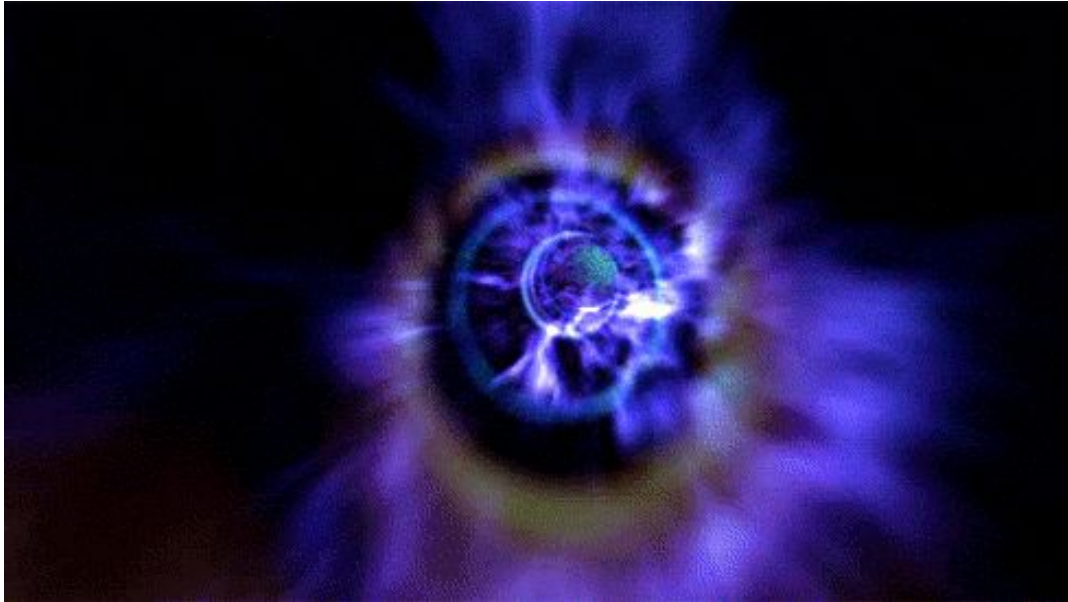
```
for line in file:  
    ...
```

- And also:

```
for k in dict:  
    ...
```

PEP 234 -- Iterators

- Infinite collection:



PEP 255

PEP 255 -- Simple Generators

- Resumable functions.
- Introduces the **yield** statement.
- Makes use of the iterator protocol.
 - Call **next()**.
 - Run until **yield**.
 - **Freeze execution, return control to the caller.**
 - **Retains local state!**

PEP 255 -- Simple Generators

```
def fib():  
    a, b = 0, 1  
    while 1:  
        yield b  
        a, b = b, a+b
```

PEP 255 -- Simple Generators

```
>>> my_fib = fib()
>>> for item in my_fib:
...     if item > 100:
...         break
...     print(item)
...
1
1
2
3
5
8
13
21
34
55
89
>>> 
```

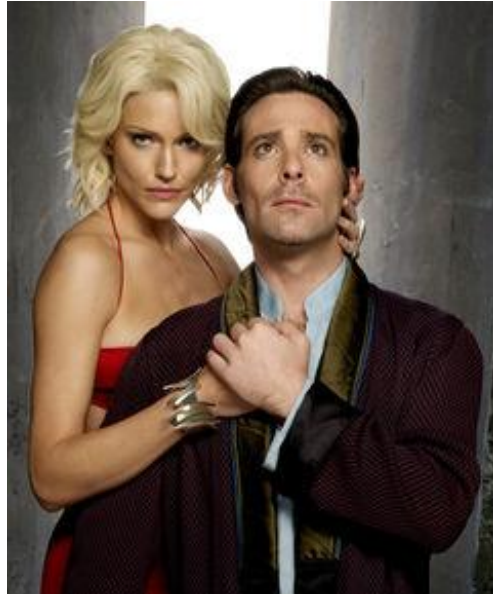
PEP 255 -- Simple Generators

- Fun with Iterators and Generators - Malcolm Tredinnick
 - <https://www.youtube.com/watch?v=vD-JJD5tllg>

PEP 498

PEP 498 -- Literal String Interpolation

- The one true way of doing strings in Python 3.6+.



PEP 498 -- Literal String Interpolation

- Why? Before f-strings came along we had:

- %-formatting:

```
>>> msg = 'disk failure'  
>>> 'error: %s' % msg  
'error: disk failure'
```

- str.format():

```
>>> value = 4 * 20  
>>> 'The value is {value}.'.format(value=value)  
'The value is 80.'
```

PEP 498 -- Literal String Interpolation

- Why? Before f-strings came along we had:

PEP 498 -- Literal String Interpolation

- Why? Before f-strings came along we had:
 - Concatenation with +:

```
>>> value = 'HORRIBLE'  
>>> 'The value is ' + value  
'The value is HORRIBLE'
```

PEP 498 -- Literal String Interpolation



PEP 498 -- Literal String Interpolation

- Simply, prepend f:

```
>>> f'Hello world!'
'Hello world!'
```

- Use braces to insert any variable:

```
>>> name = 'world'
>>> f'Hello {name}!'
'Hello world!'
```

PEP 498 -- Literal String Interpolation

- ... or just about any Python expression that you feel like inserting:

```
>>> import math  
>>> f'The square root of 500 is {math.sqrt(500)}'  
'The square root of 500 is 22.360679774997898'
```




PEP 498 -- Literal String Interpolation

- ... or just about any Python expression that you feel like inserting:

PEP 498 -- Literal String Interpolation

- ... or just about any Python expression that you feel like inserting:

```
>>> f'One hundred digits of pi: {math.pi:.100f}'  
'One hundred digits of pi:  
3.1415926535897931159979634685441851615905761718750000000000000000000000000000000  
000000000000000'  
>>>
```



PEP 498 -- Literal String Interpolation

- Cannot be used in docstrings.
- Cannot be used with `gettext()`.



Advanced PEPs

PEP 484

PEP 484 -- Type Hints

“Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.”

- Enables static analysis.
- Some day, runtime type-checking (optional, not enforced!).
- Also useful for documentation purposes.

PEP 484 -- Type Hints

- Makes use of PEP 3107-style annotations:

```
def greeting(name: str) -> str:  
    return 'Hello ' + name
```

PEP 484 -- Type Hints

- Also, type aliases:

```
from typing import TypeVar, Iterable, Tuple

T = TypeVar('T', int, float, complex)
Vector = Iterable[Tuple[T, T]]

def inproduct(v: Vector[T]) -> T:
    return sum(x*y for x, y in v)
def dilate(v: Vector[T], scale: T) -> Vector[T]:
    return ((x * scale, y * scale) for x, y in v)
vec = [] # type: Vector[float]
```


PEP 557

PEP 557 -- Data Classes

“Mutable namedtuples with defaults.”

- Define class attributes and types.
- Generates `__init__`, `__repr__`, comparison methods.

PEP 557 -- Data Classes

```
@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

PEP 557 -- Data Classes

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0) -> None:
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand

def __repr__(self):
    return f'InventoryItem(name={self.name!r}, unit_price={self.unit_price!r},
quantity_on_hand={self.quantity_on_hand!r})'

def __eq__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) == (other.name,
other.unit_price, other.quantity_on_hand)
    return NotImplemented
```

PEP 557 -- Data Classes

```
def __ne__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) != (other.name,
other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __lt__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) < (other.name,
other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __le__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) <= (other.name,
other.unit_price, other.quantity_on_hand)
    return NotImplemented
```

PEP 557 -- Data Classes

```
def __gt__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) > (other.name,
other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __ge__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) >= (other.name,
other.unit_price, other.quantity_on_hand)
    return NotImplemented
```

PEP 557 -- Data Classes

- Saves **a lot** of boilerplate code.
- Does not replace **attrs**.
 - Validation.
 - Converters.
 - Slotted classes.
 - Moar.
- Flavio Curella: “Dataclasses and attrs: when and why.”
 - <https://www.revsys.com/tidbits/dataclasses-and-attrs-when-and-why/>

PEP 572

PEP 572 -- Assignment Expressions

- Name the result of an expression → allows reuse!
- Programmers value writing fewer lines of code over shorter (but possibly indented) lines.
 - Coincidentally, comprehensions ^.

PEP 572 -- Assignment Expressions

`name := expression`



PEP 572 -- Assignment Expressions

- Those 'if <something> is not None':

```
match = pattern.search(data)
if match is not None:
    # Do something with match
```

- Now turn into:

```
if (match := pattern.search(data)) is not None:
    # Do something with match
```

PEP 572 -- Assignment Expressions

- Usage with `any()` or `all()`:

```
if any((comment := line).startswith('#') for line in lines):  
    print("First comment:", comment)  
else:  
    print("There are no comments")
```

- Or in a comprehension:

```
total = 0  
partial_sums = [total := total + v for v in values]  
print("Total:", total)
```

PEP 572 -- Assignment Expressions

- Examples from Python's standard library:
 - site.py changed this:

```
env_base = os.environ.get("PYTHONUSERBASE", None)
if env_base:
    return env_base
```

Into this:

```
if env_base := os.environ.get("PYTHONUSERBASE", None):
    return env_base
```

PEP 572 -- Assignment Expressions

- Examples from Python's standard library:
 - copy.py changed this:

```
reductor = dispatch_table.get(cls)
if reductor:
    rv = reductor(x)
else:
    reductor = getattr(x, "__reduce_ex__", None)
    if reductor:
        rv = reductor(4)
    else:
        reductor = getattr(x, "__reduce__", None)
        if reductor:
            rv = reductor()
        else:
            raise Error(
                "un(deep)copyable object of type %s" % cls)
```

PEP 572 -- Assignment Expressions

- Examples from Python's standard library:

- Into this:

```
if reductor := dispatch_table.get(cls):
    rv = reductor(x)
elif reductor := getattr(x, "__reduce_ex__", None):
    rv = reductor(4)
elif reductor := getattr(x, "__reduce__", None):
    rv = reductor()
else:
    raise Error("un(deep)copyable object of type %s" % cls)
```

PEP 572 -- Assignment Expressions

- Less indentation.
- Less lines.
- Happy programmer.



Thank you!

