Data Classes (PEP 557)
The effortless tool to define data

Eli Gur
CTO @ RavTech
June 4th, 2018
Agenda

- Some background
- Class example
- Type hints (annotations)
- Dataclasses example
- Field objects
- __post_init__
- Why not use namedtuples?
- Why not use attrs?
- Summary
Coding vs. boilerplate
Example Data: Inventory Item

- Name
- Unit price
- Quantity on hand
- Total cost
Using class: `__init__`

class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    def __init__(self, name, unit_price, quantity_on_hand):
        self.name = name
        self.unit_price = unit_price
        self.quantity_on_hand = quantity_on_hand
Using class: ```__repr__``` (and ```__str__``)

```python
def __repr__(self):
    return \
    f'InventoryItem(name={self.name!r},' \
    f'unit_price={self.unit_price!r},' \
    f'quantity_on_hand={self.quantity_on_hand!r})'
```
Using class: methods

def total_cost(self):
    return self.unit_price * self.quantity_on_hand
Using class: `__eq__`, `sort()` and using as key

```python
item1 = InventoryItem("book1", 2.2, 30)
item2 = InventoryItem("book1", 2.2, 30)
item1 == item2

item3 = InventoryItem("book2", 2.5, 3)
items = [item1, item2, item3]
items.sort()

my_dict[item1] = "more data about book 1"
```
Type hints (annotations)

class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    def __init__(self, name: str,  
                 unit_price: float, 
                 quantity_on_hand: int) -> None:
        self.name = name
        self.unit_price = unit_price
        self.quantity_on_hand = quantity_on_hand
from dataclasses import dataclass

dataclass

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

    def total_cost(self):
        return self.unit_price * self.quantity_on_hand
from dataclasses import dataclass

@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str  # No default value
    unit_price: float
    quantity_on_hand: int = 0  # Default value

    def total_cost(self):
        return self.unit_price * self.quantity_on_hand
What dataclasses generate

_init_
__repr__ (-> __str__)
__eq__, __ne__
__lt__, __le__, __gt__, __ge__
__hash__
Data class parameters

```python
from dataclasses import dataclass
@dataclass(init=True, repr=True, eq=True,
            order=False, unsafe_hash=False, frozen=False)
class InventoryItem:
    ...
```
Generated methods example

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
@dataclass() parameters

init: __init__
repr: __repr__ (-> __str__)
eq: __eq__, __ne__
order: __lt__, __le__, __gt__, __ge__
unsafe_hash: __hash__ (if eq and frozen) (*)
frozen: disable assignment to fields (**)
Using dataclasses: eq, sort() and using as key

```python
item1 = InventoryItem("book1", 2.2, 30)
item2 = InventoryItem("book1", 2.2, 30)
item1 == item2  # eq

item3 = InventoryItem("book2", 2.5, 3)
items = [item1, item2, item3]
items.sort()  # order

my_dict[item1] = "more data about book1"  # unsafe_hash
```
Field parameters

- default (MISSING)
- default_factory (MISSING)
- repr (True)
- hash (None)
- init (True)
- compare (True)
- metadata (None) (*)
field() example

```python
from dataclasses import dataclass, field

class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = field(default_factory=list)
    sum_xyz: int = field(init=False)

    def __post_init__(self):
        self.sum_xyz = self.x + self.y + self.z
```
Inheritance

@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15

# Generated __init__ for C:
def __init__(self, x: int=15, y: int=0, z: int=10):
    ...
Helper functions

```python
fields(InventoryItem)
sadict(item1)
astuple(item1)
replace(item1, unit_price=2.3)
is_dataclass(InventoryItem)
make_dataclass(...)```
dataclass VS. namedtuple (and typing.NamedTuple)

• Strict comparison

  \texttt{Point3D}(2017,6,2) == (2017,6,2)
  \texttt{Point3D}(2017,6,2) == \texttt{DatePoint}(2017,6,2)

• Mutable instances
• Default values
• Control fields in \texttt{__init__}, \texttt{__repr__} etc.
• Support combining by inheritance
• Type hints
dataclass VS. namedtuple (and typing.NamedTuple)

- Easier to add more fields

```python
Time = namedtuple('Time', ['hour', 'minute'])
def get_time():
    return Time(12, 0)

hour, minute = get_time()

# Code will break if second is added
```
dataclass VS. attrs

• attrs is an excellent library
• Many concepts implemented in dataclasses
• Tradeoff to achieve simplicity (no validators, converters etc.)
• If attrs only had to support >= 3.7 and didn’t have any backward compatibility constraints, what would it look like?
• Guido Van Rossum decided for simplicity
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.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Summary

- Pythonic beautiful implementation
- Simple to use
- Write little, get much more
- Readable
Questions?
Thank You!