Pycon 2018 – Lecture Notes

Raymond Hettinger

May 13, 2018

Contents

My Mission Train thousands of Python Programmers

Contact Info raymond dot hettinger at gmail dot com

Company Mutable Minds, Inc.

Training videos Free to user's of Safari Online: "Modern Python: Big Ideas, Little Code"

Twitter Account @raymondh

1 Dataclasses: The code generator to end all code generators

1.1 Introduction

What is a code generator?

It is a tool that writes code for you if you give it a list of specifications.

This can save time and reduce wordiness. It is also can supports useful defaults that implement best practices.

What are dataclasses for?

There are two views about this:

1. It makes a mutable data holder, in the spirit of named tuples.
2. It writes boiler-plate code for you, simplifying the process of writing the class.

These two world views are reflected in the name, "Dataclasses"

What to think about during this talk?

1. How to use the code generator?
2. What does it write for you?
3. Is that the code you wanted?
4. Do you find the result to be worth the time spent learning the tool?
5. What is the impact on debugging?

History

- In the beginning, there were dicts, tuples, and hand-written classes.
- Later, the standard libary added named tuples and types.SimpleNamespace.
- ORMs (like Django, SQLAlchemy, and PeeWee) pioneered using class attributes to specify rich data structures.
- Third-party tools like the *traitlets* package were a model for using class attributes to specify data validators.
- In Python 3.6, it became possible to supply type annotations using a new and clean syntax, `visitor_count : int`.
- The third-party library, *attrs*, was inspiration for dataclasses.

Goals

- Per the dataclasses PEP, dataclasses are roughly a “Mutable named tuple with defaults”.
- Build on the success of the *attrs* package.
- Write boilerplate code for Python classes.
- Provide an elegant syntax for creating data holder objects.

1.2 Comparison with Named Tuples

Basic Example

The common case is making a structure with named fields, type annotations, and default values.

Dataclass version:

```
from dataclasses import dataclass

@dataclass
class Color:
    hue: int
    saturation: float
    lightness: float = 0.5
```

Namedtuple version:

```
from typing import NamedTuple

class Color(NamedTuple):
    hue: int
    saturation: float
    lightness: float = 0.5
```

So far, the only evident distinction is that dataclasses use a class decorator and named tuples use inheritance with a metaclass.

Working with the dataclass

```
>>> from dataclasses import asdict, astuple, replace

>>> c = Color(33, 1.0)
>>> c
Color(hue=33, saturation=1.0, lightness=0.5)

>>> c.hue
33
>>> c.saturation
1.0
>>> c.lightness
0.5

>>> replace(c, hue=120)
Color(hue=120, saturation=1.0, lightness=0.5)
>>> asdict(c)
{'hue': 33, 'saturation': 1.0, 'lightness': 0.5}
>>> astuple(c)
(33, 1.0, 0.5)

>>> Color.__annotations__
{'hue': <class 'int'>,
 'saturation': <class 'float'>,
 'lightness': <class 'float'>}

>>> c.hue = 66
>>> c
Color(hue=66, saturation=1.0, lightness=0.5)

>>> import sys
>>> sys.getsizeof(c) + sys.getsizeof(vars(c))
168

>>> import timeit
>>> min(timeit.repeat('c.hue', 'from __main__ import c'))
0.0333401049999793
```

Working with the named tuple

```
>>> from typing import NamedTuple

>>> c = Color(33, 1.0)
```

(continues on next page)

(continued from previous page)

```
>>> c
Color(hue=33, saturation=1.0, lightness=0.5)

>>> c.hue
33
>>> c.saturation
1.0
>>> c.lightness
0.5

>>> c._replace(hue=120)
Color(hue=120, saturation=1.0, lightness=0.5)
>>> c._asdict()
OrderedDict([('hue', 33), ('saturation', 1.0), ('lightness', 0.5)])
>>> tuple(c)
(33, 1.0, 0.5)

>>> Color.__annotations__
OrderedDict([('hue', <class 'int'>),
             ('saturation', <class 'float'>),
             ('lightness', <class 'float'>)])

>>> hue, saturation, luminosity = c

>>> sys.getsizeof(c)
72

>>> import timeit
>>> min(timeit.repeat('c.hue', 'from __main__ import c'))
0.06142597799998839
```

Comparison

Dataclass	**NamedTuple**
replace() function	*_replace()* method
asdict() function	*_asdict()* method
converts to regular dict	converted to *OrderedDict*
astuple() function	*tuple()* function
Mutable	Frozen
Unhashable	Hashable
Non-iterable	Iterable and unpackable
No comparison methods	Sortable
Underlying store: instance dict	Underlying store: tuple
168 bytes	72 bytes
33 ns access	61 ns access

1.3 Generated Code

Common case

```python
from dataclasses import dataclass

@dataclass
class Color:
    hue: int
    saturation: float
    lightness: float = 0.5
```

Generated code

```python
from dataclasses import Field, _MISSING_TYPE, _DataclassParams

class Color:
    'Color(hue: int, saturation: float, lightness: float = 0.5)'

    def __init__(self, hue: int, saturation: float, lightness: float = 0.5) -> None:
        self.hue=hue
        self.saturation=saturation
        self.lightness=lightness

    def __repr__(self):
        return (self.__class__.__qualname__ +
                f"(hue={self.hue!r}, saturation={self.saturation!r}, "
                f"lightness={self.lightness!r})")

    def __eq__(self,other):
        if other.__class__ is self.__class__:
            return (self.hue,self.saturation,self.lightness) == (other.hue,other.
↪saturation,other.lightness)
        return NotImplemented

    __hash__ = None

    hue: int
    saturation: float
    lightness: float = 0.5

    __dataclass_params__ = _DataclassParams(
        init=True,
        repr=True,
        eq=True,
        order=False,
        unsafe_hash=False,
        frozen=False)

    __dataclass_fields__ = {
        'hue': Field(default=_MISSING_TYPE,
                     default_factory=_MISSING_TYPE,
                     init=True,
                     repr=True,
                     hash=None,
```

(continues on next page)

(continued from previous page)

```
                    compare=True,
                    metadata={}),
        'saturation': Field(default=_MISSING_TYPE,
                           default_factory=_MISSING_TYPE,
                           init=True,
                           repr=True,
                           hash=None,
                           compare=True,
                           metadata={}),
        'lightness': Field(default=0.5,
                          default_factory=_MISSING_TYPE,
                          init=True,
                          repr=True,
                          hash=None,
                          compare=True,
                          metadata={})
    }
    __dataclass_fields__['hue'].name = 'hue'
    __dataclass_fields__['hue'].type = int
    __dataclass_fields__['saturation'].name = 'saturation'
    __dataclass_fields__['saturation'].type = float
    __dataclass_fields__['lightness'].name = 'lightness'
    __dataclass_fields__['lightness'].type = float
```

Interesting points

- At first, this matches hand-written code for *__init__()*, *__repr__()* and *__eq__()*. Likewise, the docstring matches what would be written by hand.
- The equality comparison not only checks values, it also does an exact class match. This is an important feature.
- The *__hash__* is set to None so that you don't get accidental hashability using identity.
- The generated code includes everything from the original so that the default value shows-up as a class variable, and the type declarations show-up in the class annotations.
- The generated code also includes a lot of metadata that would not have by in the hand-written version. The *__dataclass_params__* attribute records the parameters used to generate the dataclass was created. The *__dataclass_fields__* attribute has the details for each field.

1.4 Freezing and Ordering

For the common case, dataclasses are mutable. That is one of their principal benefits. Accordingly, they are not hashable so their instances cannot be used as set elements or dictionary keys.

Likewise, dataclasses are not orderable by default. The reason for this restriction is to prevent the TypeErrors that ensue when one or more of the fields is unorderable.

When needed, these defaults are easily overridden by specifying that you want immutability, hashability, and ordering.

Creating the dataclass

```python
from dataclasses import dataclass

@dataclass(order=True, frozen=True)
class Color:
    hue: int
    saturation: float
    lightness: float = 0.5
```

Working with the dataclass

Presto, now we have hashability and ordering!

```python
>>> from pprint import pprint

>>> colors = [Color(33, 1.0),
              Color(66, 0.75),
              Color(99, 0.5),
              Color(66, 0.75)]

>>> pprint(sorted(colors))
[Color(hue=33, saturation=1.0, lightness=0.5),
 Color(hue=66, saturation=0.75, lightness=0.5),
 Color(hue=66, saturation=0.75, lightness=0.5),
 Color(hue=99, saturation=0.5, lightness=0.5)]

>>> pprint(set(colors))
{Color(hue=33, saturation=1.0, lightness=0.5),
 Color(hue=66, saturation=0.75, lightness=0.5),
 Color(hue=99, saturation=0.5, lightness=0.5)}
```

Generated code

```python
def __lt__(self, other):
    if other.__class__ is self.__class__:
        return (self.hue, self.saturation, self.lightness) < (other.hue, other.
↪saturation, other.lightness)
    return NotImplemented

def __le__(self, other):
    if other.__class__ is self.__class__:
        return (self.hue, self.saturation, self.lightness) <= (other.hue, other.
↪saturation, other.lightness)
    return NotImplemented

def __gt__(self, other):
    if other.__class__ is self.__class__:
        return (self.hue, self.saturation, self.lightness) > (other.hue, other.
↪saturation, other.lightness)
    return NotImplemented

def __ge__(self, other):
    if other.__class__ is self.__class__:
        return (self.hue, self.saturation, self.lightness) >= (other.hue, other.
↪saturation, other.lightness)
```

(continues on next page)

(continued from previous page)

```
    return NotImplemented

def __setattr__(self, name, value):
    if type(self) is cls or name in ('hue', 'saturation', 'lightness'):
        raise FrozenInstanceError(f"cannot assign to field {name!r}")
    super(cls,  self).__setattr__(name, value)

def __delattr__(self, name):
    cls = self.__class__
    if type(self) is cls or name in ('hue', 'saturation', 'lightness'):
        raise FrozenInstanceError(f"cannot delete field {name!r}")
    super(cls,  self).__delattr__(name)

def __hash__(self):
    return hash((self.hue, self.saturation, self.lightness))
```

Interesting points

- Unlike *functools.total_ordering()*, the comparison methods are type specific and don't need to worry about reversed fallback comparisons.
- Because *object()* has default comparison methods that return *NotImplemented*, the comparison methods do not call *super()*.
- Frozen behavior (immutability) is implemented by extending *__setattr__()* and *__delattr__()* to block writes and deleted to fields.
- This differs from the usual hand-written approach which uses read-only properties. See the *Fractions* module for an example.
- The extended attribute access methods do call *super()* so that they don't unintentionally turn-off important behaviors in a parent class.
- The hash method matches what would be written by hand.
- In named tuples, all of these behaviors come for free because they are inherited from tuple. And being written in C, the named tuple versions are much faster than these pure python implementations.

1.5 Custom Field Specifications

Dataclasses excel at customizing the class creation process.

In this example, we want to create an *Employee* class that stores an *emp_id*, *name*, *gender*, *salary*, *age*, as well as a list of users who have viewed the employee record.

Field factories

Instead of having a fixed default value, sometimes we want to create a new instance of a collection, such as a list, set, or dictionary.

In this example, we need a new list of users who have accessed the employee record. This is implemented with `field(default_factory) = list`.

Custom methods

Adding methods to a dataclass is no different than for any other class.

In this example, we want an *access()* method to record who viewed the employee record.

Limiting hashing to immutable fields

The unchanging parts of the record are the *emp_id*, *name*, and *gender*.

We exclude the other fields from the hash with `field(hash=False)` or by specifying a *default_factory* which is always presumed to be excluded from the hash function.

Limiting which fields are displayed

Ordinarily, all fields are included in *__repr__()*, but this isn't always desirable.

In this case, we want to exclude sensitive data such as *salary* or *metadata* such as the list of people who have viewed the record. We do this by specifying `field(repr=False)`.

Limiting which fields are used in comparisons

We the dataclass has an `order=True` parameter, then all fields get included in the comparison methods.

This isn't always what we want. Some types such as functions and complex numbers aren't orderable and would raise a TypeError if included in the comparison.

Also, we want to exclude metadata such as the list of accessors because those weren't present at the time the object was instantiated.

To exclude a field from comparisons, specify, `field(compare=False)`.

Attaching metadata

In databases, we customarily write "data dictionaries" that give more information about the fields.

Dataclasses support that need using the *metadata* parameter which can be viewed using the *field()* function.

In our example, we want to note that the salary units are measured in bitcoins by specifying `metadata={'units': 'bitcoin'}`.

Creating the dataclass

With dataclasses, it takes more time to discuss the above issues than it takes to implement them:

```python
from dataclasses import dataclass, field
from datetime import datetime

@dataclass(order=True, unsafe_hash=True)
class Employee:
    emp_id: int
    name: str
    gender: str
    salary: int = field(hash=False, repr=False, metadata={'units': 'bitcoin'})
    age: int = field(hash=False)
```

(continues on next page)

(continued from previous page)

```
    viewed_by: list = field(default_factory=list, compare=False, repr=False)

    def access(self, viewer_id):
        self.viewed_by.append((viewer_id, datetime.now()))
```

Working with the dataclass

```
>>> from dataclasses import fields
>>> from pprint import pprint

>>> e1 = Employee(emp_id='3536465054',
             name = 'Rachel Hettinger',
             gender = 'female',
             salary = 22,
             age = 0x30,
    )

>>> e2 = Employee(emp_id='4054646353',
             name = 'Martin Murchison',
             gender = 'male',
             salary = 20,
             age = 0x30,
    )

>>> e1.access('Roger Wastun')
>>> e1.access('Shelly Summers')
>>> pprint(e1.viewed_by)
[('Roger Wastun', datetime.datetime(2018, 5, 12, 14, 26, 44, 439705)),
 ('Shelly Summers', datetime.datetime(2018, 5, 12, 14, 26, 58, 255971))]

>>> pprint(sorted([e1, e2]))
[Employee(emp_id='3536465054', name='Rachel Hettinger', gender='female', age=48),
 Employee(emp_id='4054646353', name='Martin Murchison', gender='male', age=48)]

>>> assignments = {e1: 'gather requirements', e2: 'write tests'}
>>> pprint(assignments)
{Employee(emp_id='3536465054', name='Rachel Hettinger', gender='female', age=48):
↪'gather requirements',
 Employee(emp_id='4054646353', name='Martin Murchison', gender='male', age=48):
↪'write tests'}

>>> fields(e1)[3]
Field(name='salary',
      type=<class 'int'>,
      default=<verbose_dataclasses._MISSING_TYPE object at 0x1104764a8>,
      default_factory=<verbose_dataclasses._MISSING_TYPE object at 0x1104764a8>,
      init=True,
      repr=False,
      hash=False,
      compare=True,
      metadata={'units': 'bitcoin'})
```

Generated code

```
from dataclasses import _HAS_DEFAULT_FACTORY

def __init__(self, emp_id:int, name:str, gender:str, salary:int, age:int, viewed_by:␣
↪list=_HAS_DEFAULT_FACTORY) -> None:
    self.emp_id = emp_id
    self.name = name
    self.gender = gender
    self.salary = salary
    self.age = age
    self.viewed_by = list() if viewed_by is _HAS_DEFAULT_FACTORY else viewed_by

def __repr__(self):
    return (self.__class__.__qualname__ +
            f"(emp_id={self.emp_id!r}, name={self.name!r}, gender={self.gender!r},␣
↪age={self.age!r})")

def __eq__(self, other):
    if other.__class__ is self.__class__:
        return (self.emp_id, self.name, self.gender, self.salary, self.age) == (other.
↪emp_id, other.name, other.gender, other.salary, other.age)
    return NotImplemented

def __lt__(self, other):
    if other.__class__ is self.__class__:
        return (self.emp_id, self.name, self.gender, self.salary, self.age) < (other.
↪emp_id, other.name, other.gender, other.salary, other.age)
    return NotImplemented

def __le__(self, other):
    if other.__class__ is self.__class__:
        return (self.emp_id,self.name,self.gender,self.salary,self.age) <= (other.emp_
↪id,other.name,other.gender,other.salary,other.age)
    return NotImplemented

def __gt__(self, other):
    if other.__class__ is self.__class__:
        return (self.emp_id, self.name, self.gender, self.salary, self.age) > (other.
↪emp_id, other.name, other.gender, other.salary, other.age)
    return NotImplemented

def __ge__(self, other):
    if other.__class__ is self.__class__:
        return (self.emp_id, self.name, self.gender, self.salary, self.age) >= (other.
↪emp_id, other.name, other.gender, other.salary, other.age)
    return NotImplemented

def __hash__(self):
    return hash((self.emp_id, self.name, self.gender))
```

1.6 Closing Thoughts

Challenges

1. Adding `__slots__` to a dataclass is currently incompatible having default values. This will be hard to fix without having the class decorator create a new class rather than modifying the class in-place.
2. A special technique, `__post_init__()` is needed to call a `__init__()` in a parent class.
3. Dataclasses currently don't work well with immutable parent classes where we need to override or extend a `__new__()` method.

Future directions

- Direct support for `__slots__`
- Custom data validators