Data Classes in Python 3.7

Brian Stempin | Yiu Ming Huynh

Goals

1. Discuss what dataclasses are
2. Compare/contrast uses of dataclasses
3. Compare/contrast performance of dataclasses

What are Dataclasses?

They're classes that are wrapped with the `dataclass` decorator:

```
from dataclasses import dataclass

@dataclass
class MyExampleClass(object):
     x: int
     y: int = 20
```

Dataclass Features

- The dunder methods: implements __eq__, __repr__, __ne__, __gt__, __lt__, __le__, __ge__
- enables the following properties:
  - Order
  - Frozen
  - Unsafe_hash
- Has post_init functions

Feature Comparison

We want to compare and contrast the features of dataclasses with other solutions so that we know which tool to choose for a given situation.

Pros of Dataclasses vs tuples/namedtuples

Dataclasses as a class have their own names, whereas tuples are always tuples

```
@dataclass
class CartesianPoints:
    x: float
    y: float

@dataclass
class PolarPoints:
    r: float
    theta: float

c = CartesianPoints(1, 2)
p = PolarPoints(1, 2)
>>> print(c == p)
False
```

Dataclasses as a class have their own names, whereas tuples are always tuples

```
c = (1, 2)
p = (1, 2)
>>> print(c == p)
True
```

Namedtuples kinda solve the problem, but then you run into this:

```
CartesianPoint = namedtuple('CartesianPoint', field_names=['x',
'y'])
c = CartesianPoint(x=1, y=2)
p = (1, 2)
>>> print(c == p)
True
```

Tuples are always immutable...

```
>>> s = (1, 2, 3)
>>> s[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

... but dataclasses have options

```
@dataclass
class MutatingMing:
    super_powers: List[str]

@dataclass(frozen=True)
class ForeverMing:
    super_powers: List[str]

m1 = MutatingMing(super_powers=["shapeshifting master"])
m1.super_powers = ["levitation"]

m2 = ForeverMing(super_powers=["stops time"])
m2.super_powers = ["super human strength"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 3, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'super_powers'
```

Dataclasses can inherit from other classes...

```
@dataclass
class Product:
    name: str

@dataclass
class DigitalProduct(Product):
    download_link: URL

@dataclass
class Ebook(DigitalProduct):
    isbn: str
```

But try doing that with a tuple

```
Product = namedtuple(‘Product’, field_names=[‘name’])

DigitalProduct = namedtuple(‘DigitalProduct’, field_names=[‘name’,
‘download_link`])

Ebook = namedtuple(‘Ebook’, field_names=[‘name’, ‘download_link’,
‘isbn’])
```

Dataclasses have class methods...

```
@dataclass
class CartesianPoint:
    x: float
    y: float

    def calculate_distance(self, other):
        ...
```

vs tuples...

```
c1 = Tuple(1,2)

def calculate_distance(c1: Tuple[float, float], c2:
Tuple[float, float]):
    ...
```

Cons of Dataclasses vs tuples

Tuples have even less boiler plate to create than dataclasses

```
t = "hey look", 1, True, "a tuple just like that"

Vs

@dataclass
class ARandomBundleOfAttributes:
    opener: str
    random_number: int
    random_bool: bool
    closing_statement: str

ARandomBundleOfAttributes("but look!", 7, False, "i'm a dataclass!")
```

Misc that I don't wanna do a code demo of

- [spoiler text] (not really) Tuples have better performance... Coming up soon
- Tuples are naturally immutable, so they make a good data structure for multithreading

Pros of Dataclasses vs Dict

Dataclasses have well structured, specified attributes

```
@dataclass
class TemperaturePoint:
    x: float
    y: float
    temperature: float


def create_heatmap(temp_points: List[TemperaturePoint]):
    ...
```

Whereas if you just had dictionaries...

```
temperature_points = [
    {"x": 1.2, "y": 4.5, "temperature": 20.0},
    {"x": 5.4, "temperature": 24.0}]

def create_heatmap(point_temps: List[MutableMapping]):
    ...
```

Dictionaries cannot inherit from other dictionaries

```
species = {
    "name": "mountain toucan"
}

pet = {
    "species": species,
    "name": "billy"
}
```

I'm not gonna try anymore...

Cons of Dataclasses vs attrs

Dicts are super flexible, and syntatically they are easy to construct

```
phones_to_addresses = {
    "+13125004000": {"name": "Billy the Toucan"},
    "+13125004001": {"name": "Polly the Parrot"},
    ...
}
```

Try doing this with a dataclass

```
@dataclass
class PhoneNumberToAddress:
    # you can't even have a string that starts with a symbol
or
    # number as an attribute
    pass
```

I gave up before I even tried.

Dicts are json-serializable by default

```
s = {"+13125000000": "123 Auto Mechanics Inc"}
dumped_string = json.dumps(s)
print(dumped_string)

'{"+13125000000": "123 Auto Mechanics Inc"}'
```

You need to do some massaging with dataclasses

```
@dataclass
class PhoneEntry:
    number: str
    business_name: str

d = dataclasses.asdict(PhoneEntry('+13125000000', 'Paul and Angela's Bistro'))
json.dumps(d)
print(d)
'{"number": "+13125000000", "business_name": "Paul and Angela's Bistro"}'
```

Pros of Dataclasses vs attrs

Pros of Dataclasses vs attrs

Dataclasses come with the standard library; you have to install attrs as a library.

```
# requirements.txt

attrs==17.10.0
```

Cons of Dataclasses vs attrs

Cons of Dataclasses vs attrs

- Attrs can validate the attributes via validators
- Attrs can also convert attributes
- Attrs also has slots, whereas in dataclasses you have to explicitly state the attributes you want to slot (Note: the attrs slots class is actually a totally different class)

Cons of Dataclasses vs attrs

```
@attr.s(slots=True)
class YellowPageEntry:
    phone_number: PhoneNumber =
attr.ib(convert=phonenumbers.parse)
    business_name: str = attr.ib(validator=instance_of(str))
```

So many more features!

Performance in Detail

Performance: Bottom Line Up Front

- dataclasses and attrs are so close in performance that it shouldn't be a factor in choosing one over the other
- dataclasses and attrs come at a very noticeable cost
- tuples (plain and named, in that order) are the all-time performance king
- dicts are far more performant that I expected them to be

Open Performance Questions

- How much of the dataclasses/attrs slow down has to do with the type checking and validation?
- How much of the dataclasses/attrs slow down has to do with how the data is being stored?

Benchmarking Process

- ASV (Airspeed Velocity) was a life saver and was used to measure CPU time and memory usage
- Every benchmark starts with an attribute count ("ac" for the rest of this presentation)
- A list of N random names, types, and values to fit those types are generated and stored. E.g.: `[['a', 'b', 'c'], [int, str, int], [4, '3vdna9s', 9482]]`
- We test creation time by instantiation the data container under test 10,000 with the previously mentioned randoom data
- ASV does this several times to generate averages
- For places where applicable, we test how long an instantition plus mutation costs

Benchmarking Process

- We test creation time by instantiation the data container under test 10,000 times with the previously mentioned random data
- ASV does this several times to generate averages
- Where applicable, we test instantiation plus mutation costs

Performance Tidbits: dataclasses

- Immutability is practically free
- Generally speaking, dataclasses use less memory than attrs despite missing slot support (<4% difference over all values of ac)
- Almost always a smaller memory foot-print than dictionaries (<25% difference for ac <= 125, 40% difference for ac=625)
- Much slower than dict, tupe, and namedtuples when dealing with a large number of attributes

Performance Tidbits: attrs

- Very similar performance characteristics to dataclasses
- Slots save almost nothing for mutable objects, but they save > 10% on memory for immutable objects
- Slotting does not create a noticeable time difference for classes with a small number of attributes
- Mutating classes that use slots is as fast as classes that aren't slotted

Performance Tidbits: dict

- Becomes a memory-hog several hundred elements (twice as much as tuples, 50% more than dataclasses), but they are on-par for attribute counts < 100
- They are faster to mutate and create than dataclasses and attrs, even at small numbers (uses 33% of the time at ac=5, 21% of the time at ac=25)
- Faster to create than named tuples until ac=125

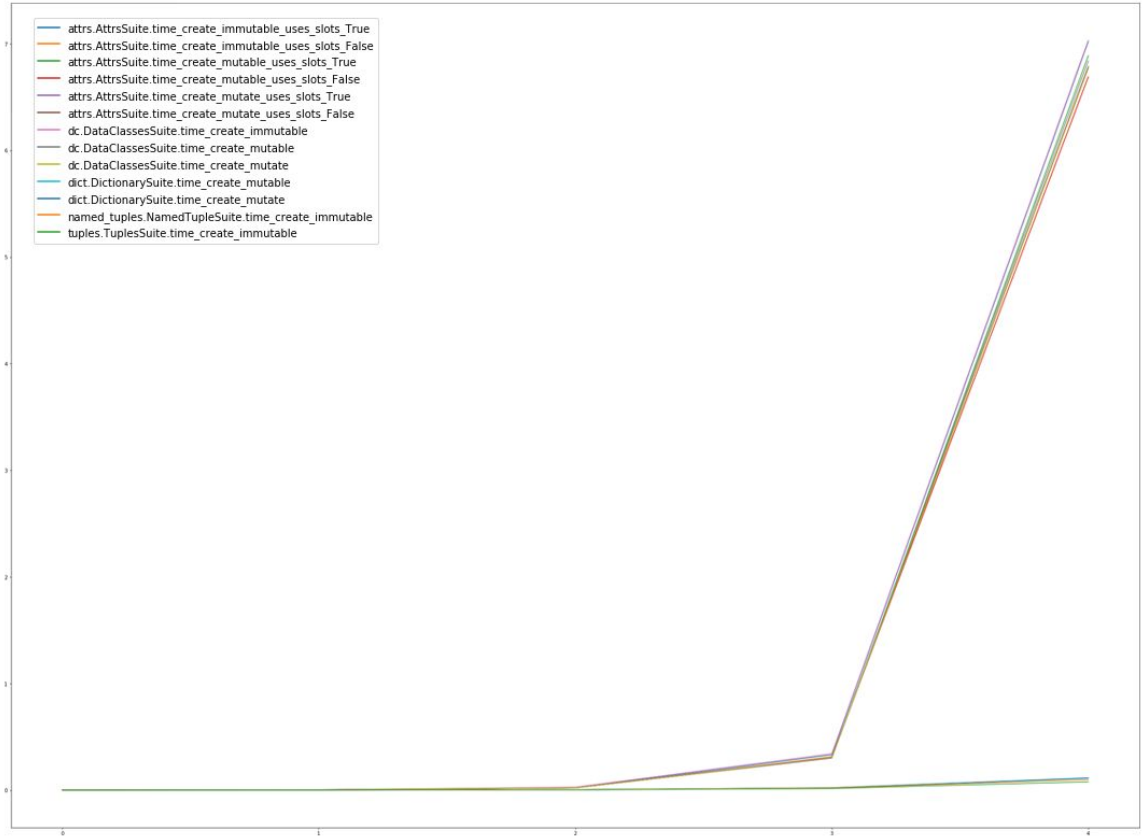
Performance Tidbits: named tuples

- Save around 10% on memory vs dataclasses and attrs
- Use almost the same amount of memory as dicts at small sizes, but have savings > 10% at ac=25
- Saves a significant amount of time vs dataclasses and attrs (64% difference at ac=5 and getts better as ac grows)
- Creation time is slower than dicts until ac=25, then they become faster

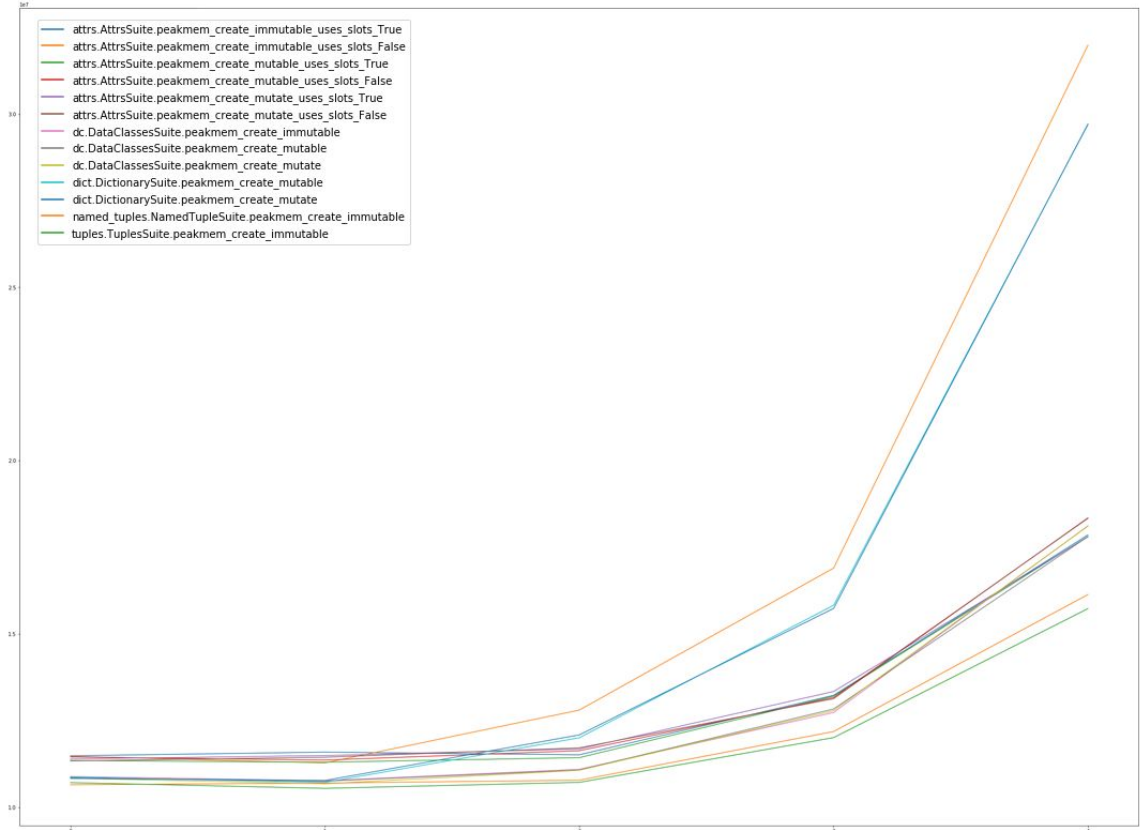
Performance Tidbits: tuples

- Fastest over-all creation time
- Smallest over-all memory footprint (just barely smaller than namedtuples)
- Uses between 50% and 80% of the creation time as a named tuple
- Saves ~10% on memory compared to attrs and dataclasses

CPU Time


attrs.AttrsSuite.time_create_immutable_uses_slots_True
attrs.AttrsSuite.time_create_immutable_uses_slots_False
attrs.AttrsSuite.time_create_mutable_uses_slots_True
attrs.AttrsSuite.time_create_mutable_uses_slots_False
attrs.AttrsSuite.time_create_mutate_uses_slots_True
attrs.AttrsSuite.time_create_mutate_uses_slots_False
dc.DataClassesSuite.time_create_immutable
dc.DataClassesSuite.time_create_mutable
dc.DataClassesSuite.time_create_mutate
dict.DictionarySuite.time_create_mutable
dict.DictionarySuite.time_create_mutate
named_tuples.NamedTupleSuite.time_create_immutable
tuples.TuplesSuite.time_create_immutable

Memory Usage



Key Takeaways

1. Dataclasses are slower than most of the other options

2. Dataclasses are reasonable when it comes to memory usage

3. Dataclasses have no "killer features"

Questions?
Comments?
Complaints?

Thank you