

PEP 557 – Data Classes

Author: Eric V. Smith <eric at trueblade.com>

Status: Final

Type: Standards Track

Created: 02-Jun-2017

Python-Version: 3.7

Post-History: 08-Sep-2017, 25-Nov-2017, 30-Nov-2017, 01-Dec-2017, 02-Dec-2017,
06-Jan-2018, 04-Mar-2018

Resolution: Python-Dev message

Notice for Reviewers

This PEP and the initial implementation were drafted in a separate repo:

<https://github.com/ericvsmith/dataclasses>. Before commenting in a public forum please at least read the discussion listed at the end of this PEP.

Abstract

This PEP describes an addition to the standard library called Data Classes. Although they use a very different mechanism, Data Classes can be thought of as “mutable namedtuples with defaults”. Because Data Classes use normal class definition syntax, you are free to use inheritance, metaclasses, docstrings, user-defined methods, class factories, and other Python class features.

A class decorator is provided which inspects a class definition for variables with type annotations as defined in PEP 526, “Syntax for Variable Annotations”. In this document, such variables are called fields. Using these fields, the decorator adds generated method definitions to the class to support instance initialization, a repr, comparison methods, and optionally other methods as described in the Specification section. Such a class is called a Data Class, but there’s really nothing special about the class: the decorator adds generated methods to the class and returns the same class it was given.

As an example:

```
@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
```

The `@dataclass` decorator will add the equivalent of these methods to the `InventoryItem` class:

```
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={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
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
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
```

Data Classes save you from writing and maintaining these methods.

Rationale

There have been numerous attempts to define classes which exist primarily to store values which are accessible by attribute lookup. Some examples include:

- `collections.namedtuple` in the standard library.
- `typing.NamedTuple` in the standard library.
- The popular `attrs` [1] project.
- George Sakkis' `recordType` recipe [2], a mutable data type inspired by `collections.namedtuple`.
- Many example online recipes [3], packages [4], and questions [5]. David Beazley used a form of data classes as the motivating example in a PyCon 2013 metaclass talk [6].

So, why is this PEP needed?

With the addition of PEP 526, Python has a concise way to specify the type of class members. This PEP leverages that syntax to provide a simple, unobtrusive way to describe Data Classes. With two exceptions, the specified attribute type annotation is completely ignored by Data Classes.

No base classes or metaclasses are used by Data Classes. Users of these classes are free to use inheritance and metaclasses without any interference from Data Classes. The decorated classes are truly “normal” Python classes. The Data Class decorator should not interfere with any usage of the class.

One main design goal of Data Classes is to support static type checkers. The use of PEP 526 syntax is one example of this, but so is the design of the `fields()` function and the `@dataclass` decorator. Due to their very dynamic nature, some of the libraries mentioned above are difficult to use with static type checkers.

Data Classes are not, and are not intended to be, a replacement mechanism for all of the above libraries. But being in the standard library will allow many of the simpler use cases to instead leverage Data Classes. Many of the libraries listed have different feature sets, and will of course continue to exist and prosper.

Where is it not appropriate to use Data Classes?

- API compatibility with tuples or dicts is required.
- Type validation beyond that provided by PEPs 484 and 526 is required, or value validation or conversion is required.

Specification

All of the functions described in this PEP will live in a module named `dataclasses`.

A function `dataclass` which is typically used as a class decorator is provided to post-process classes and add generated methods, described below.

The `dataclass` decorator examines the class to find `fields`. A `field` is defined as any variable identified in `__annotations__`. That is, a variable that has a type annotation. With two exceptions described below, none of the Data Class machinery examines the type specified in the annotation.

Note that `__annotations__` is guaranteed to be an ordered mapping, in class declaration order. The order of the fields in all of the generated methods is the order in which they appear in the class.

The `dataclass` decorator will add various “dunder” methods to the class, described below. If any of the added methods already exist on the class, a `TypeError` will be raised. The decorator returns the same class that is called on: no new class is created.

The `dataclass` decorator is typically used with no parameters and no parentheses. However, it also supports the following logical signature:

```
def dataclass(*, init=True, repr=True, eq=True, order=False, unsafe_hash=False, froze
```



If `dataclass` is used just as a simple decorator with no parameters, it acts as if it has the default values documented in this signature. That is, these three uses of `@dataclass` are equivalent:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)
class C:
    ...
```

The parameters to `dataclass` are:

- `init`: If true (the default), a `__init__` method will be generated.
- `repr`: If true (the default), a `__repr__` method will be generated. The generated repr string will have the class name and the name and repr of each field, in the order they are defined in the class. Fields that are marked as being excluded from the repr are not included. For example: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.

If the class already defines `__repr__`, this parameter is ignored.

- `eq`: If true (the default), an `__eq__` method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

If the class already defines `__eq__`, this parameter is ignored.

- `order`: If true (the default is False), `__lt__`, `__le__`, `__gt__`, and `__ge__` methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If `order` is true and `eq` is false, a `ValueError` is raised.

If the class already defines any of `__lt__`, `__le__`, `__gt__`, or `__ge__`, then `ValueError` is raised.

- `unsafe_hash`: If False (the default), the `__hash__` method is generated according to how `eq` and `frozen` are set.

If `eq` and `frozen` are both true, Data Classes will generate a `__hash__` method for you. If `eq` is true and `frozen` is false, `__hash__` will be set to `None`, marking it unhashable (which it is). If `eq` is false, `__hash__` will be left untouched meaning the `__hash__` method of the

superclass will be used (if the superclass is `object`, this means it will fall back to id-based hashing).

Although not recommended, you can force Data Classes to create a `__hash__` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can nonetheless be mutated. This is a specialized use case and should be considered carefully.

If a class already has an explicitly defined `__hash__` the behavior when adding `__hash__` is modified. An explicitly defined `__hash__` is defined when:

- `__eq__` is defined in the class and `__hash__` is defined with any value other than `None`.
- `__eq__` is defined in the class and any non-`None` `__hash__` is defined.
- `__eq__` is not defined on the class, and any `__hash__` is defined.

If `unsafe_hash` is true and an explicitly defined `__hash__` is present, then `ValueError` is raised.

If `unsafe_hash` is false and an explicitly defined `__hash__` is present, then no `__hash__` is added.

See the Python documentation [7] for more information.

- `frozen`: If true (the default is `False`), assigning to fields will generate an exception. This emulates read-only frozen instances. If either `__getattr__` or `__setattr__` is defined in the class, then `ValueError` is raised. See the discussion below.

fields may optionally specify a default value, using normal Python syntax:

```
@dataclass
class C:
    a: int          # 'a' has no default value
    b: int = 0      # assign a default value for 'b'
```

In this example, both `a` and `b` will be included in the added `__init__` method, which will be defined as:

```
def __init__(self, a: int, b: int = 0):
```

`TypeError` will be raised if a field without a default value follows a field with a default value. This is true either when this occurs in a single class, or as a result of class inheritance.

For common and simple use cases, no other functionality is required. There are, however, some Data Class features that require additional per-field information. To satisfy this need for additional information, you can replace the default field value with a call to the provided `field()` function. The signature of `field()` is:

```
def field(*, default=MISSING, default_factory=MISSING, repr=True,
         hash=None, init=True, compare=True, metadata=None)
```

The `MISSING` value is a sentinel object used to detect if the `default` and `default_factory` parameters are provided. This sentinel is used because `None` is a valid value for `default`.

The parameters to `field()` are:

- `default`: If provided, this will be the default value for this field. This is needed because the `field` call itself replaces the normal position of the default value.
- `default_factory`: If provided, it must be a zero-argument callable that will be called when a default value is needed for this field. Among other purposes, this can be used to specify fields with mutable default values, as discussed below. It is an error to specify both `default` and `default_factory`.
- `init`: If true (the default), this field is included as a parameter to the generated `__init__` method.
- `repr`: If true (the default), this field is included in the string returned by the generated `__repr__` method.
- `compare`: If True (the default), this field is included in the generated equality and comparison methods (`__eq__`, `__gt__`, et al.).
- `hash`: This can be a bool or `None`. If True, this field is included in the generated `__hash__` method. If `None` (the default), use the value of `compare`: this would normally be the expected behavior. A field should be considered in the hash if it's used for comparisons. Setting this value to anything other than `None` is discouraged.

One possible reason to set `hash=False` but `compare=True` would be if a field is expensive to compute a hash value for, that field is needed for equality testing, and there are other fields that contribute to the type's hash value. Even if a field is excluded from the hash, it will still be used for comparisons.

- `metadata`: This can be a mapping or `None`. `None` is treated as an empty dict. This value is wrapped in `types.MappingProxyType` to make it read-only, and exposed on the `Field` object. It is not used at all by Data Classes, and is provided as a third-party extension mechanism. Multiple third-parties can each have their own key, to use as a namespace in the metadata.

If the default value of a field is specified by a call to `field()`, then the class attribute for this field will be replaced by the specified `default` value. If no `default` is provided, then the class attribute will be deleted. The intent is that after the `dataclass` decorator runs, the class attributes will all contain the default values for the fields, just as if the default value itself were specified. For example, after:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

The class attribute `C.z` will be `10`, the class attribute `C.t` will be `20`, and the class attributes `C.x` and `C.y` will not be set.

Field objects

`Field` objects describe each defined field. These objects are created internally, and are returned by the `fields()` module-level method (see below). Users should never instantiate a `Field` object directly. Its documented attributes are:

- `name`: The name of the field.
- `type`: The type of the field.
- `default`, `default_factory`, `init`, `repr`, `hash`, `compare`, and `metadata` have the identical meaning and values as they do in the `field()` declaration.

Other attributes may exist, but they are private and must not be inspected or relied on.

post-init processing

The generated `__init__` code will call a method named `__post_init__`, if it is defined on the class. It will be called as `self.__post_init__()`. If no `__init__` method is generated, then `__post_init__` will not automatically be called.

Among other uses, this allows for initializing field values that depend on one or more other fields. For example:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

See the section below on init-only variables for ways to pass parameters to `__post_init__()`. Also see the warning about how `replace()` handles `init=False` fields.

Class variables

One place where `dataclass` actually inspects the type of a field is to determine if a field is a class variable as defined in PEP 526. It does this by checking if the type of the field is `typing.ClassVar`. If a field is a `ClassVar`, it is excluded from consideration as a field and is

ignored by the Data Class mechanisms. For more discussion, see [8]. Such `classVar` pseudo-fields are not returned by the module-level `fields()` function.

Init-only variables

The other place where `dataclass` inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type `dataclasses.InitVar`. If a field is an `InitVar`, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level `fields()` function. Init-only fields are added as parameters to the generated `__init__` method, and are passed to the optional `__post_init__` method. They are not otherwise used by Data Classes.

For example, suppose a field will be initialized from a database, if a value is not provided when creating the class:

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

In this case, `fields()` will return `Field` objects for `i` and `j`, but not for `database`.

Frozen instances

It is not possible to create truly immutable Python objects. However, by passing `frozen=True` to the `@dataclass` decorator you can emulate immutability. In that case, Data Classes will add `__setattr__` and `__delattr__` methods to the class. These methods will raise a `FrozenInstanceError` when invoked.

There is a tiny performance penalty when using `frozen=True`: `__init__` cannot use simple assignment to initialize fields, and must use `object.__setattr__`.

Inheritance

When the Data Class is being created by the `@dataclass` decorator, it looks through all of the class's base classes in reverse MRO (that is, starting at `object`) and, for each Data Class that it finds, adds the fields from that base class to an ordered mapping of fields. After all of the base class fields are added, it adds its own fields to the ordered mapping. All of the generated methods will use this combined, calculated ordered mapping of fields. Because the fields are in insertion order, derived classes override base classes. An example:


```

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

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

```

The final list of fields is, in order, `x`, `y`, `z`. The final type of `x` is `int`, as specified in class `c`.

The generated `__init__` method for `c` will look like:

```

def __init__(self, x: int = 15, y: int = 0, z: int = 10):

```

Default factory functions

If a field specifies a `default_factory`, it is called with zero arguments when a default value for the field is needed. For example, to create a new instance of a list, use:

```

l: list = field(default_factory=list)

```

If a field is excluded from `__init__` (using `init=False`) and the field also specifies `default_factory`, then the default factory function will always be called from the generated `__init__` function. This happens because there is no other way to give the field an initial value.

Mutable default values

Python stores default member variable values in class attributes. Consider this example, not using Data Classes:

```

class C:
    x = []
    def add(self, element):
        self.x += element

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x

```

Note that the two instances of class `c` share the same class variable `x`, as expected.

Using Data Classes, *if* this code was valid:

```
@dataclass
class D:
    x: List = []
    def add(self, element):
        self.x += element
```

it would generate code similar to:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x += element

assert D().x is D().x
```

This has the same issue as the original example using class `c`. That is, two instances of class `D` that do not specify a value for `x` when creating a class instance will share the same copy of `x`. Because Data Classes just use normal Python class creation they also share this problem. There is no general way for Data Classes to detect this condition. Instead, Data Classes will raise a `TypeError` if it detects a default parameter of type `list`, `dict`, or `set`. This is a partial solution, but it does protect against many common errors. See [Automatically support mutable default values in the Rejected Ideas section](#) for more details.

Using default factory functions is a way to create new instances of mutable types as default values for fields:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

Module level helper functions

- `fields(class_or_instance)`: Returns a tuple of `Field` objects that define the fields for this Data Class. Accepts either a Data Class, or an instance of a Data Class. Raises `ValueError` if not passed a Data Class or instance of one. Does not return pseudo-fields which are `ClassVar` or `InitVar`.
- `asdict(instance, *, dict_factory=dict)`: Converts the Data Class `instance` to a dict (by using the factory function `dict_factory`). Each Data Class is converted to a dict of its fields, as `name:value` pairs. Data Classes, dicts, lists, and tuples are recursed into. For example:

```

@dataclass
class Point:
    x: int
    y: int

@dataclass
class C:
    l: List[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'l': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}

```

Raises `TypeError` if instance is not a Data Class instance.

- `astuple(*, tuple_factory=tuple)`: Converts the Data Class instance to a tuple (by using the factory function `tuple_factory`). Each Data Class is converted to a tuple of its field values. Data Classes, dicts, lists, and tuples are recursed into.

Continuing from the previous example:

```

assert astuple(p) == (10, 20)
assert astuple(c) == ((0, 0), (10, 4)),)

```

Raises `TypeError` if instance is not a Data Class instance.

- `make_dataclass(cls_name, fields, *, bases=(), namespace=None)`: Creates a new Data Class with name `cls_name`, fields as defined in `fields`, base classes as given in `bases`, and initialized with a namespace as given in `namespace`. `fields` is an iterable whose elements are either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, `typing.Any` is used for `type`. This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the `dataclass` function to convert that class to a Data Class. This function is provided as a convenience. For example:

```

C = make_dataclass('C',
                  [ ('x', int),
                    'y',
                    ('z', int, field(default=5)) ],
                  namespace={'add_one': lambda self: self.x + 1})

```

Is equivalent to:

```

@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1

```

- `replace(instance, **changes)`: Creates a new object of the same type of `instance`, replacing fields with values from `changes`. If `instance` is not a Data Class, raises `TypeError`. If values in `changes` do not specify fields, raises `TypeError`.

The newly returned object is created by calling the `__init__` method of the Data Class. This ensures that `__post_init__`, if present, is also called.

Init-only variables without default values, if any exist, must be specified on the call to `replace` so that they can be passed to `__init__` and `__post_init__`.

It is an error for `changes` to contain any fields that are defined as having `init=False`. A `ValueError` will be raised in this case.

Be forewarned about how `init=False` fields work during a call to `replace()`. They are not copied from the source object, but rather are initialized in `__post_init__()`, if they're initialized at all. It is expected that `init=False` fields will be rarely and judiciously used. If they are used, it might be wise to have alternate class constructors, or perhaps a custom `replace()` (or similarly named) method which handles instance copying.

- `is_dataclass(class_or_instance)`: Returns True if its parameter is a dataclass or an instance of one, otherwise returns False.

If you need to know if a class is an instance of a dataclass (and not a dataclass itself), then add a further check for `not isinstance(obj, type)`:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

Discussion

python-ideas discussion

This discussion started on python-ideas [9] and was moved to a GitHub repo [10] for further discussion. As part of this discussion, we made the decision to use PEP 526 syntax to drive the discovery of fields.

Support for automatically setting `__slots__`?

At least for the initial release, `__slots__` will not be supported. `__slots__` needs to be added at class creation time. The Data Class decorator is called after the class is created, so in order to add `__slots__` the decorator would have to create a new class, set `__slots__`, and return it. Because this behavior is somewhat surprising, the initial version of Data Classes will not support automatically setting `__slots__`. There are a number of workarounds:

- Manually add `__slots__` in the class definition.
- Write a function (which could be used as a decorator) that inspects the class using `fields()` and creates a new class with `__slots__` set.

For more discussion, see [11].

Why not just use namedtuple?

- Any namedtuple can be accidentally compared to any other with the same number of fields. For example: `Point3D(2017, 6, 2) == Date(2017, 6, 2)`. With Data Classes, this would return `False`.
- A namedtuple can be accidentally compared to a tuple. For example, `Point2D(1, 10) == (1, 10)`. With Data Classes, this would return `False`.
- Instances are always iterable, which can make it difficult to add fields. If a library defines:

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

Then if a user uses this code as:

```
hour, minute = get_time()
```

then it would not be possible to add a second field to `Time` without breaking the user's code.

- No option for mutable instances.
- Cannot specify default values.
- Cannot control which fields are used for `__init__`, `__repr__`, etc.
- Cannot support combining fields by inheritance.

Why not just use `typing.NamedTuple`?

For classes with statically defined fields, it does support similar syntax to Data Classes, using type annotations. This produces a namedtuple, so it shares `namedtuple`'s benefits and some of its downsides. Data Classes, unlike `typing.NamedTuple`, support combining fields via inheritance.

Why not just use `attrs`?

- `attrs` moves faster than could be accommodated if it were moved in to the standard library.
- `attrs` supports additional features not being proposed here: validators, converters, metadata, etc. Data Classes makes a tradeoff to achieve simplicity by not implementing these features.

For more discussion, see [12].

post-init parameters

In an earlier version of this PEP before `InitVar` was added, the post-init function `__post_init__` never took any parameters.

The normal way of doing parameterized initialization (and not just with Data Classes) is to provide an alternate classmethod constructor. For example:

```
@dataclass
class C:
    x: int

    @classmethod
    def from_file(cls, filename):
        with open(filename) as fl:
            file_value = int(fl.read())
        return C(file_value)

c = C.from_file('file.txt')
```

Because the `__post_init__` function is the last thing called in the generated `__init__`, having a classmethod constructor (which can also execute code immediately after constructing the object) is functionally equivalent to being able to pass parameters to a `__post_init__` function.

With `InitVars`, `__post_init__` functions can now take parameters. They are passed first to `__init__` which passes them to `__post_init__` where user code can use them as needed.

The only real difference between alternate classmethod constructors and `InitVar` pseudo-fields is in regards to required non-field parameters during object creation. With `InitVars`, using `__init__` and the module-level `replace()` function `InitVars` must always be specified. Consider the case where a `context` object is needed to create an instance, but isn't stored as a field. With alternate classmethod constructors the `context` parameter is always optional, because you could still create the object by going through `__init__` (unless you suppress its creation). Which approach is more appropriate will be application-specific, but both approaches are supported.

Another reason for using `InitVar` fields is that the class author can control the order of `__init__` parameters. This is especially important with regular fields and `InitVar` fields that have default values, as all fields with defaults must come after all fields without defaults. A previous design had all init-only fields coming after regular fields. This meant that if any field had a default value, then all init-only fields would have to have default values, too.

asdict and astuple function names

The names of the module-level helper functions `asdict()` and `astuple()` are arguably not PEP 8 compliant, and should be `as_dict()` and `as_tuple()`, respectively. However, after discussion [13] it was decided to keep consistency with `namedtuple._asdict()` and `attr.asdict()`.

Rejected ideas

Copying `init=False` fields after new object creation in `replace()`

Fields that are `init=False` are by definition not passed to `__init__`, but instead are initialized with a default value, or by calling a default factory function in `__init__`, or by code in `__post_init__`.

A previous version of this PEP specified that `init=False` fields would be copied from the source object to the newly created object after `__init__` returned, but that was deemed to be inconsistent with using `__init__` and `__post_init__` to initialize the new object. For example, consider this case:

```
@dataclass
class Square:
    length: float
    area: float = field(init=False, default=0.0)

    def __post_init__(self):
        self.area = self.length * self.length

s1 = Square(1.0)
s2 = replace(s1, length=2.0)
```

If `init=False` fields were copied from the source to the destination object after `__post_init__` is run, then `s2` would end up being `Square(length=2.0, area=1.0)`, instead of the correct `Square(length=2.0, area=4.0)`.

Automatically support mutable default values

One proposal was to automatically copy defaults, so that if a literal list `[]` was a default value, each instance would get a new list. There were undesirable side effects of this decision, so the final decision is to disallow the 3 known built-in mutable types: list, dict, and set. For a complete discussion of this and other options, see [14].

Examples

Custom `__init__` method

Sometimes the generated `__init__` method does not suffice. For example, suppose you wanted to have an object to store `*args` and `**kwargs`:

```

@dataclass(init=False)
class ArgHolder:
    args: List[Any]
    kwargs: Mapping[Any, Any]

    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

a = ArgHolder(1, 2, three=3)

```

A complicated example

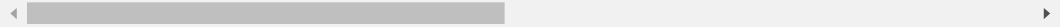
This code exists in a closed source project:

```

class Application:
    def __init__(self, name, requirements, constraints=None, path='', executable_lin
        self.name = name
        self.requirements = requirements
        self.constraints = {} if constraints is None else constraints
        self.path = path
        self.executable_links = [] if executable_links is None else executable_links
        self.executables_dir = executables_dir
        self.additional_items = []

    def __repr__(self):
        return f'Application({self.name!r},{self.requirements!r},{self.constraints!r}

```



This can be replaced by:

```

@dataclass
class Application:
    name: str
    requirements: List[Requirement]
    constraints: Dict[str, str] = field(default_factory=dict)
    path: str = ''
    executable_links: List[str] = field(default_factory=list)
    executable_dir: Tuple[str] = ()
    additional_items: List[str] = field(init=False, default_factory=list)

```

The Data Class version is more declarative, has less code, supports typing, and includes the other generated functions.

Acknowledgements

The following people provided invaluable input during the development of this PEP and code: Ivan Levkivskiy, Guido van Rossum, Hynek Schlawack, Raymond Hettinger, and Lisa Roach. I thank them for their time and expertise.

A special mention must be made about the `attrs` project. It was a true inspiration for this PEP, and I respect the design decisions they made.

References

- [1] attrs project on github (<https://github.com/python-attrs/attrs>)
- [2] George Sakkis' recordType recipe (<http://code.activestate.com/recipes/576555-records/>)
- [3] DictDotLookup recipe (<http://code.activestate.com/recipes/576586-dot-style-nested-lookups-over-dictionary-based-dat/>)
- [4] attrdict package (<https://pypi.python.org/pypi/attrdict>)
- [5] StackOverflow question about data container classes (<https://stackoverflow.com/questions/3357581/using-python-class-as-a-data-container>)
- [6] David Beazley metaclass talk featuring data classes (<https://www.youtube.com/watch?v=sPiWg5jSoZI>)
- [7] Python documentation for `__hash__` (https://docs.python.org/3/reference/datamodel.html#object.__hash__)
- [8] ClassVar discussion in PEP 526
- [9] Start of python-ideas discussion (<https://mail.python.org/pipermail/python-ideas/2017-May/045618.html>)
- [10] GitHub repo where discussions and initial development took place (<https://github.com/ericvsmith/dataclasses>)
- [11] Support `__slots__`? (<https://github.com/ericvsmith/dataclasses/issues/28>)
- [12] why not just attrs? (<https://github.com/ericvsmith/dataclasses/issues/19>)
- [13] PEP 8 names for `asdict` and `astuple` (<https://github.com/ericvsmith/dataclasses/issues/110>)
- [14] Copying mutable defaults (<https://github.com/ericvsmith/dataclasses/issues/3>)

Copyright

This document has been placed in the public domain.

Source: <https://github.com/python/peps/blob/main/peps/pep-0557.rst>

Last modified: 2023-09-09 17:39:29 GMT