dataclasses — Data Classes

Source code: Lib/dataclasses.py

This module provides a decorator and functions for automatically adding generated special methods such as `__init__()` and `__repr__()` to user-defined classes. It was originally described in **PEP 557**.

The member variables to use in these generated methods are defined using **PEP 526** type annotations. For example this code:

```
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 = 0

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

Will add, among other things, a `__init__()` that looks like:

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: i
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Note that this method is automatically added to the class: it is not directly specified in the `InventoryItem` definition shown above.

New in version 3.7.

Module-level decorators, classes, and functions

@dataclasses.**dataclass**(**, init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False*)

This function is a decorator that is used to add generated special methods to classes, as described below.

The `dataclass()` decorator examines the class to find `field`s. A `field` is defined as class variable that has a type annotation. With two exceptions described below, nothing in `dataclass()` examines the type specified in the variable annotation.

The order of the fields in all of the generated methods is the order in which they appear in the class definition.

The `dataclass()` decorator will add various “dunder” methods to the class, described below. If any of the added methods already exist on the class, the behavior depends on the parameter, as documented below. The decorator returns the same class that is called on; no new class is created.

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_ha
class C:
   ...
```

The parameters to `dataclass()` are:

- `init`: If true (the default), a `__init__()` method will be generated.

  If the class already defines `__init__()`, this parameter is ignored.
- `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 `TypeError` is raised.
- `unsafe_hash`: If `False` (the default), a `__hash__()` method is generated according to how `eq` and `frozen` are set.

  `__hash__()` is used by built-in `hash()`, and when objects are added to hashed collections such as dictionaries and sets. Having a `__hash__()` implies that instances of the class are immutable. Mutability is a complicated property that depends on the programmer's intent, the existence and behavior of `__eq__()`, and the values of the `eq` and `frozen` flags in the `dataclass()` decorator.

  By default, `dataclass()` will not implicitly add a `__hash__()` method unless it is safe to do so. Neither will it add or change an existing explicitly defined `__hash__()` method. Setting the class attribute `__hash__ = None` has a specific meaning to Python, as described in the `__hash__()` documentation.

  I f `__hash__()` is not explicit defined, or if it is set to `None`, then `dataclass()` *may* add an implicit `__hash__()` method. Although not recommended, you can force `dataclass()` 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.

Here are the rules governing implicit creation of a `__hash__()` method. Note that you cannot both have an explicit `__hash__()` method in your dataclass and set `unsafe_hash=True`; this will result in a `TypeError`.

If `eq` and `frozen` are both true, by default `dataclass()` 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, since it is mutable). 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).

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

`field`s 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.

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

For common and simple use cases, no other functionality is required. There are, however, some dataclass 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. For example:

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

As shown above, 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`. No code should directly use the `MISSING` value.

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. I f 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 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.

class dataclasses.**Field**

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.

dataclasses.**fields**(*class_or_instance*)

Returns a tuple of Field objects that define the fields for this dataclass. Accepts either a dataclass, or an instance of a dataclass. Raises TypeError if not passed a dataclass or instance of one. Does not return pseudo-fields which are ClassVar or InitVar.

dataclasses.**asdict**(*instance*, ***, *dict_factory=dict*)

Converts the dataclass `instance` to a dict (by using the factory function `dict_factory`). Each dataclass is converted to a dict of its fields, as `name: value` pairs. dataclasses, dicts, lists, and tuples are recursed into. For example:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: list[Point]

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

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

Raises `TypeError` if `instance` is not a dataclass instance.

dataclasses.**astuple**(*instance*, ***, *tuple_factory=tuple*)

Converts the dataclass `instance` to a tuple (by using the factory function `tuple_factory`). Each dataclass is converted to a tuple of its field values. dataclasses, 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 dataclass instance.

dataclasses.**make_dataclass**(*cls_name*, *fields*, ***, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*)

Creates a new dataclass 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 each either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, `typing.Any` is used for `type`. The values of `init`, `repr`, `eq`, `order`, `unsafe_hash`, and `frozen` have the same meaning as they do in `dataclass()`.

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 dataclass. 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
```

dataclasses.**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 dataclass. 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.

dataclasses.**is_dataclass**(*class_or_instance*)

Return `True` if its parameter is a dataclass or an instance of one, otherwise return `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)
```

Post-init processing

The generated `__init__()` code will call a method named `__post_init__()`, if `__post_init__()` is defined on the class. It will normally be called as `self.__post_init__()`. However, if any `InitVar` fields are defined, they will also be passed to `__post_init__()` in the order they were defined in the class. 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 of two places 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 dataclass mechanisms. 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 dataclasses.

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, dataclasses 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 dataclass 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 dataclass 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:

```
mylist: 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 dataclasses:

```
class C:
    x = []
    def add(self, element):
        self.x.append(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 dataclasses, *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 dataclasses just use normal Python class creation they also share this behavior. There is no general way for Data Classes to detect this condition. Instead, dataclasses 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.

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
```

Exceptions

exception `dataclasses.`**`FrozenInstanceError`**

Raised when an implicitly defined `__setattr__()` or `__delattr__()` is called on a dataclass which was defined with `frozen=True`.

© Copyright 2001-2020, Python Software Foundation. Quick search Go |
The Python Software Foundation is a non-profit corporation. Please donate.

Last updated on Nov 19, 2020. Found a bug?
Created using Sphinx 2.4.4.